Visual Objects

For
Windows 2000® and Windows XP®

Programmer's Guide
Version 2.7

Contents

Chapter 1: Introduction

Chapter 2: From Character Mode to Windows

Chapter 3: Program Structure and Flow

Chapter 4: Standard Components—Classes, Objects, and Libraries

Chapter 5: Object Linking and Embedding

Chapter 6: Justifying Database Access Choices

Chapter 7: Data Server Classes

Chapter 8: Using DBF Files

Chapter 9: Concurrency Control

Chapter 10: Justifying User Interface Choices

Chapter 11: GUI Classes

Chapter 12: Other Features of the GUI Classes

Chapter 13: Printing

Chapter 14: Error and Exception Handling

Chapter 15: File Handling

Chapter 16: Hyperlabels

Chapter 17: Operating Environment

Chapter 18: Third-Party Components

Chapter 19: How the Visual Objects Two-Level Preprocessor Works

Chapter 20: Overview of Language Elements

Chapter 21: Data Types

Chapter 22: Variables, Constants, and Declarations

Chapter 23: Operators and Expressions

Chapter 24: Arrays

Chapter 25: Objects, Classes, and Methods

Chapter 26: Code Blocks

Chapter 27: Functions and Procedures

Appendix A: RDD Specifics

Appendix B: Reserved Words

Index

Chapter

1

Introduction

Note: For detailed information about the new Visual Objects class libraries, as well as updates to existing classes, properties, and methods, refer to the *Visual Objects Help*.

The *Programmer's Guide* is loosely divided into three parts:

- Chapters 1-3, *Moving from Character Mode to Windows,* gives you an overview of developing applications in Visual Objects, focusing in particular on the differences between programming in the DOS and Microsoft Windows environments.
- Chapters 4-19, *Subsystems* is about components. It contains a general discussion of the concept of components and how they fit into the architecture of Visual Objects and then goes on to discuss the standard components in detail.
- Chapters 20-27, *Language Elements and Program Structure* is a combination of reference material and programming topics. It contains specific information about the Visual Objects language that you will not find in any of the other reference guides, such as specifics on data types and operators. It also presents other aspects of the language by grouping together components that are more exhaustively defined in other reference guides.

This guide is organized into the following chapters and appendices:

Moving from Character Mode to Windows

Chapter 1, Introduction, 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, From Character Mode to Windows, discusses some of the major differences between programming in a character mode environment and Windows.

Chapter 3, Program Structure and Flow, discusses how to properly structure a Visual Objects application and explains how the visual development tools help you build such an application.

Subsystems

Chapter 4, Standard Components—Classes, Objects, and Libraries, discusses the standard components in Visual Objects and describes the framework for constructing, modifying, and using them.

Chapter 5, Object Linking and Embedding, discusses the use of object linking and embedding (OLE) in Visual Objects. With the support of OLE 2.0 in Visual Objects, you can use a whole world of pre-built ready-to-use components provided by the third-party community.

Chapter 6, Justifying Database Access Choices, outlines the issues involved in choosing a database access format and a programming technique (object-oriented or procedural) for applications that use DBF files.

Chapter 7, Data Server Classes, provides an overview of the built-in data server classes, discussing the common philosophy behind their designs and suggesting ways to exploit the similarities in order to make your applications as data-independent as possible.

Chapter 8, Using DBF Files, provides an overview of using DBF files from both the procedural and the object-oriented perspectives.

Chapter 9, Concurrency Control, tells you how to access database files in shared and exclusive mode, how to obtain and release locks in shared mode, and how to resolve locking and file open failures.

Chapter 10, Justifying User Interface Choices, outlines the issues involved in choosing a user interface programming technique—terminal emulation or Graphical User Interface (GUI) classes—for your applications.

Chapter 11, GUI Classes, describes the standard components of Visual Objects that deal with the GUI classes, focusing on the some of the more common arrangements and uses of windows in GUI applications.

Chapter 12, Other Features of the GUI Classes, discusses some less typical ways in which to use the GUI classes in your applications.

Chapter 13, Printing, discusses the various printing techniques that you can use in your applications.

Chapter 14, Error and Exception Handling, discusses the requirements for making an application robust and limiting the impact of exception conditions, as well as the reasons why traditional thinking needs to be extended to accommodate the complex structures of modern applications. It then describes the solution for exception handling in Visual Objects.

Chapter 15, File Handling, deals with the issue of handling files in your programs.

Chapter 16, Hyperlabels, describes how the information in an object's hyperlabel is used by the system at runtime.

Chapter 17, Operating Environment, discusses both the development and delivery platforms; topics include using DLLs and how to distribute an application.

Chapter 18, Third-Party Components, weighs the decision of developing or buying a new component and discusses the implications of buying third-party components.

Chapter 19, How the Visual Objects Two-level Preprocessor Works, describes the Visual Objects two-level preprocessor that consists of a Clipper compatible preprocessor and the Visual Objects preprocessor.

Language Elements and Program Structure

Chapter 20, Overview of Language Elements, introduces you to the basic language elements that go together to make up a program.

Chapter 21, Data Types, helps you analyze your data and choose appropriate data types from those that are available in the language.

Chapter 22, Variables, Constants, and Declarations, introduces you to variables and constants, shows you how to create and declare them, and explains how they come to be associated with a particular data type.

Chapter 23, Operators and Expressions, defines all of the operators that are available to you and shows you how to use them to build expressions.

Chapter 24, Arrays, explains the array data type in greater detail and discusses how to use arrays.

Chapter 25, Objects, Classes, and Methods, explains the object data type in greater detail and discusses the language components that you will need to create objects and use them in an application.

Chapter 26, Code Blocks, explains the code block data type in greater detail and discusses how to use code blocks.

Chapter 27, Functions and Procedures, provides a brief discussion of procedural programming in terms of defining and using functions and procedures in an application.

Appendix A, RDD Specifics, presents a table of specifications for the various RDDs supplied with Visual Objects.

Appendix B, Reserved Words, lists all reserved words in the Visual Objects language in alphabetical order.

Index

What You Need to Know

This guide is intended to help programmers understand the implications of programming in the Windows environment and assumes that you have a basic knowledge of DOS programming.

In addition to an understanding of basic programming concepts, this guide assumes that you are familiar with 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.

Some of the chapters build on information presented in other chapters. Chapter 22, "**Variables, Constants, and Declarations**" for example, assumes you understand the material presented in **Data Types** Where this is the case, the dependencies will be discussed in the introductory sections of the chapters.

In order to properly interpret syntax and programming examples, you will also need to understand certain standard conventions, which are described in the following sections.

Metasymbol and Variable Name Prefixes

Prefixes are used in syntax to denote the data type of parameters and arguments and in examples to denote the data type of variables and constants. The prefix of a variable name always appears in lowercase, followed by a logical descriptor in mixed case.

For example, *cCustomerName* is a string holding a customer name, *cbEval* is a code block to evaluate, and *nRecordNumber* is a record number.

The prefixes are shown in the following table. For more information on a particular data type, refer to **Data Types**

Prefix	Represents
a a<*type*>	ARRAY The <*type*> indicates the data type of the array elements and may be any prefix listed in this table. For example, *acNames* represents an array of strings and *adwWords* represents an array of double words.
b	BYTE
c	STRING (can also be a memo field)
cb	CODEBLOCK
d	DATE
dw	DWORD
f	FLOAT
i	INT
id	Literal identifier
k	System-defined constant
l	LOGIC
li	LONGINT
n	Numeric
o	OBJECT
psz	PSZ
ptr	PTR
r4	REAL4
r8	REAL8
si	SHORTINT
struc	STRUCTURE
sym	SYMBOL
u	USUAL Used when more than one data type is allowed.
w	WORD

x<*type*>	Either a literal identifier or an expression enclosed in parentheses (called an extended expression). The <*type*> indicates the data type of the value and may be any prefix listed in this table. For example, *xcFileName* is an extended string expression.

General Syntax Conventions

The following conventions are used to represent particular conditions in the syntax:

< >

Indicates an item you supply (like a variable name). For example, you would supply an alias name specified as an extended character expression for the following syntax representation:

```
CLOSE <xcAlias>
```

such as:

```
CLOSE Customer
```

or:

```
CLOSE (cDataFile)
```

( )

If shown within syntax representation, parentheses are required. For example, given the following syntax representation:

```
SetDeleted(<lToggle>)
```

The following command would be acceptable:

```
SetDeleted(TRUE)
```

but the following would not:

```
SetDeleted FALSE
```

,

Separates arguments and must be entered as shown. Two consecutive commas in a function, method, or procedure call indicate that an argument is being omitted.

[ ]

Indicates an optional item or list of items. (If you enter the optional item, do not type the brackets.) In the following syntax representation, the keyword STATIC is optional:

```
[STATIC] DEFINE <idConstant> := <uValue>
```

Thus, both of the following statements would be acceptable:

```
DEFINE ONE := 1
STATIC DEFINE TWO := 2
```

Square brackets are also used to define the number of elements in each dimension of an array syntax specification. When shown as part of an array specification, they are a literal part of the syntax and must be entered as shown.

→ Used in function and method syntax representations to indicate the type of value returned. In the following example, the return value for the AllTrim() function is *cTrimString*:

```
AllTrim(<cString>) → cTrimString
```

... Indicates that you can repeat the preceding element. In this syntax representation, your can specify several TO...INTO clauses:

```
SET RELATION TO [<uRecID> INTO <xcAlias>]
     [, [TO] <uRecID> INTO <xcAlias>...]
```

Also used to indicate intervening code:

```
BEGIN SEQUENCE
     <Statements>...
     IF lBreakCond
          BREAK
     ENDIF
RECOVER
     <RecoveryStatements>...
END SEQUENCE
```

| Separates a list of mutually exclusive choices—you must choose one. For example:

```
SET DELETED ON | OFF
SET ORDER TO <nPosition> | TAG <xcOrder>
```

@ Indicates that an argument must be passed by reference. The @ symbol is a literal part of the syntax and must be entered as shown. For example, for the following syntax:

```
FRead(<ptrHandle>, @<cBufferVar>, <nBytes>)
```

you might enter:

```
FRead(ptrFileOne, @cTextBuff, 254)
```

General Typographic Conventions

This guide 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.

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 and DBServer)
- 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 'Subsystems' 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 online Help, 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

From Character Mode to Windows

GUI: Graphical User Interface

If you are used to developing programs in a character mode environment, you will undoubtedly find Windows programming different. Windows is a Graphical User Interface (GUI). Things behave differently. Users expect different things. The ambition level for a Windows application is higher. Things you never considered before are now par for the course.

Programming in this new environment has both good and bad sides. There are wonderful new toys, new opportunities, and new ways to structure the solutions, and the end result can be much more satisfying—to you and to your clients. On the other hand, there are new programming challenges to overcome, new terminology to learn, and new ways to make mistakes.

By discussing some of the major differences between programming in character mode and Windows, this chapter gives you some general ideas of the challenges you face when programming for a GUI environment.

Chapter 3, "Program Structure and Flow," takes those issues one step further, offers specific programming solutions for creating a high-quality, well-behaved GUI application in Visual Objects.

Application Behavior and Structure

Most applications designed to run in a character mode environment have the same basic hierarchical structure, in which higher-level routines call lower-level routines based on user input, such as menu selections.

In such a hierarchical application, the program is in control, not the user. The user has certain flexibility in choosing what path to take, but the application has complete control over the sequence of events and what paths are available to the user.

For a character mode application, this hierarchical structure works well. However, it is not well suited for a Windows application. Applications designed like this can certainly be made to run in the Windows environment, but they do not behave the way users have come to expect a Windows application to behave.

This is because in a Windows application, the user, not the program, controls the sequence of events. Decisions, such as the order in which fields should be filled or even if they should be filled at all, are made by the user. While editing a record in one window, the user might jump to another window to edit a different record, open a new window to add another record, or switch to another application that might work on the same database.

Event: Something that happens (particularly user input)

In Windows, actions—like opening a file or printing a document—are referred to as *events*. For example, when the user clicks a toolbar button or chooses an item from a menu, he/she generates an event.

Event-driven: Application that is controlled by events at any time (particularly user input)

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 character mode applications, in which the program has control.

For a well-behaved Windows application to be event-driven, the program structure of the application must turn "inside out." Rather than a traditional hierarchical structure, it must execute in tiny atomic units that can start up at any time, do their task quickly, and finish. The higher-level routines do not call the lower-level ones directly. Instead, they yield control to the GUI environment, which in turn dispatches events to the various action routines, the *event handlers*. Such an event-driven application has no life except in these event handlers: all activity in the application occurs in response to events passed on from the operating system.

This type of behavior requires very modular programming. Unless the event handlers are well isolated from one another, there is no way to manage the behavior of the program: side effects will spread like rings on a pond, with interactions multiplying explosively and unpredictably.

It also requires very "defensive" programming. No piece of code can make any expectations about its environment. Just because the situation was good a moment ago, you cannot assume it is good now—who knows what happened in the interim?

In summary, porting a character mode application to the Windows environment without making significant modifications is possible, but if you want to design a well-behaved Windows application, you must structure it differently. Good structure is the key to developing a successful Windows application.

Multiple Users, Tasks, and Windows

Another difference between a character mode environment and Windows is that a Windows application must be designed with multi-user situations in mind, even if it will be used only on a stand-alone computer.

Under DOS, it is possible to program an application for a single user. Because DOS is basically a single-user, single-task operating system, you can know exactly what to expect. If the application is not designed for a network, you can ignore interaction with other programs.

On the other hand, under Windows, even a single user operating on a stand-alone computer, requires that applications are designed for a multi-user (actually multi-tasking) environment. For example, the user may want to access the same database with multiple applications (or multiple instances of the same application).

MDI: Multiple Document Interface

Indeed, even a single application can do many things at the same time. Most Windows applications, for example, have a Multiple Document Interface (MDI). You can open multiple documents (databases, records, or whatever) into multiple windows and work with them simultaneously. You can even open the same document in several different windows, without fear that one window will lose track of what you are doing in another one.

Think about the implications of this. Allowing a user access to multiple records might mean accessing several orders for the same customer (or different customers) at the same time. This means that you must open the same database in multiple work areas, present each work area in a separate window, and keep independent positions and record locks.

Concurrency control: Managing conflicts between different programs or tasks

It means that both the window and the program that does the work must be self-contained units that can exist in several copies without "stepping" on themselves. It means that windows must be notified when another window that looks at the same or related data does something, such as moving or changing the data. The program must protect against conflicting access and data corruption—this is called *concurrency control*.

The problem lies in the fact that although most Windows applications need behave like this, few Windows development systems make it easy to do. Designing a robust, multi-user application in which each individual component can safely coexist with itself is not so easy—you need to obtain record locks or file locks, or open files exclusively, and alert the user when data is inaccessible. In this area, traditional hierarchical programming techniques are woefully inadequate.

In summary, because Windows is a multi-tasking environment, your applications must provide solutions for concurrency control and the *multi-instance* scenario presented here.

User Interface

Finally, there is the issue of user interface. Users have come to expect that their Windows applications have a particular look and feel about them.

Part of the user interface issue is simply appreciating the distinction between programming a character-based user interface and programming a graphical one. The purpose of the GUI is to increase *bandwidth*. That is, to increase the amount of usable information the user can view at one time and to increase the speed with which the user can cause the application to do something.

You increase the bandwidth by presenting information in the form of graphs or pictures. Even when you present text, you can convey more information by using color, type styles, and other forms of decoration and by placing the text in meaningful groupings. You also increase the bandwidth by enabling the user to exercise more flexible control.

Another point to factor into the user interface issue is that while the Visual Objects terminal emulation layer gives you basic compatibility and support for DOS-style terminal I/O, you need to realize that users' expectations are much higher than they were in the DOS world. As a developer, your ambition level must rise to meet those expectations—for example:

- Modal menus in which the user must make one selection that is exclusive of all others are no longer acceptable. Users want menu access to do anything the application has to offer at any time, no matter what else they may be doing at the time.
- Users expect to operate your application without the manual and, at the same time, expect speed, convenience, and power.
- Every menu item, every button, and every control must be labeled, display a descriptive message when selected, and have online help.
- Mouse support is no longer an option, it is a requirement.
- Applications that take over the CPU and do not yield control to other applications when asked are not acceptable.

And it is not just ease-of-use or ergonomics. Applications are supposed to be pretty. You must deal with color choices, typography, layout, and the latest design fads. As if programming was not hard enough, there are entire texts to read on the subject of user interface design.

In short, users expect applications to take full advantage of the Windows environment and to provide as many bells and whistles as possible, without compromising speed or integrity. They expect a simple and predictable look and feel with both power and flexibility. However, accomplishing this is not at all simple for the developer.

New Tools for the New Approach

Despite all we've said in this chapter, it's not all bad news. Meeting the challenges of Windows programming using traditional tools and techniques is hard indeed, but the new tools of Visual Objects take care of many of the details for you (you'll learn more about this in the next chapter).

While you must still deal with the design challenges, the Visual Objects tools set up a framework for you and automatically take care of many of the more complex tasks, such as memory management, event routing, multi-tasking, and concurrency control. This enables you to focus on the business tasks and on the appearance and behavior of your solution.

Chapter

3

Program Structure and Flow

To describe what it is like to program in Visual Objects, you must look at a number of aspects: the IDE and its repository, the programming language itself, the visual development tools, and the class libraries. However, the most important aspect of the system is the structure and control flow of a prototypical application.

In fact, the principal difference between a traditional Xbase application and a Visual Objects application is the way in which they are structured. The language you use to write your business logic is familiar and the database operations are the same (even though you have the option to dress them up in an object-oriented "wrapper"). The program structure, however, is different.

This chapter discusses how to properly structure a Visual Objects program and explains how the visual development tools help you build such an application. After reading it, you will see how the class libraries provided by Visual Objects are designed to facilitate programming a GUI-based, event-driven, MDI application, as well as how the IDE tools leverage the class libraries to generate code that is not only usable but pedagogical.

The Objectives

Visual Objects is aimed at producing a full-fledged Windows application. This means:

- Multi-document interface, with no constraints on opening several databases or the same database in different windows
- Event-driven operation, with no limitations on user flexibility and control
- Top-flight graphical appearance
- Full-fledged annotation, prompting, and help
- Support for Windows conventions and subsystems, such as clipboard, drag-and-drop, and help
- Self-configuring behavior

- Maintainability, allowing extension of the system without destabilizing it or requiring wholesale changes

This is a high ambition level that you cannot realistically reach if you must program every detail explicitly. With Visual Objects, you get a development system that takes care of the implementation details, allowing you to concentrate on the business solution. In this system, object orientation holds the key, and the class libraries provide built-in support for these expectations.

The prototypical application is based on four class libraries: the RDD and SQL class libraries for database processing, the GUI Classes library for user interface processing and the OLE library for object linking and embedding. These libraries are designed to work together and are interconnected through automatic routing of message traffic and event notification. You need not be concerned with the details of the relationship, but it is good to know which things you *do not* have to worry about.

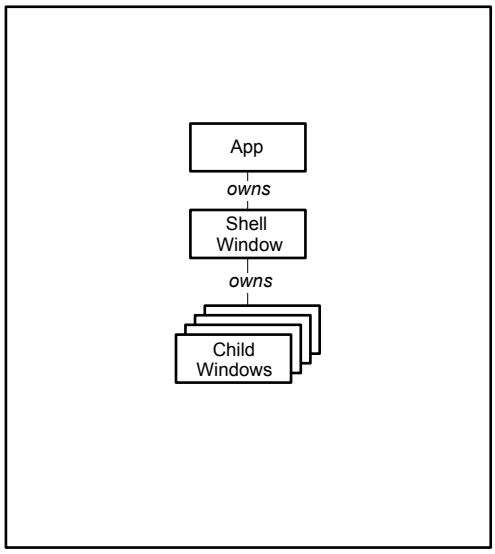
Windows and Controls

To understand the structure and control flow in an event-driven, multi-instance application, you need to look at the components that make up a program. Rather than looking at code and calling sequences, you will examine the different objects and their relationships to each other. Later, you will see how these relationships are embodied in code.

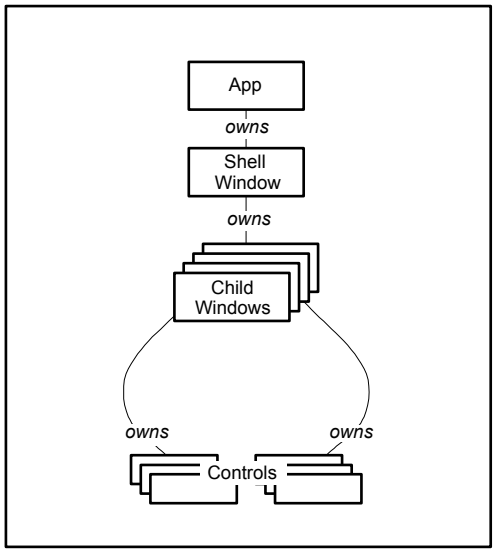
Ownership Relationships

Everything in a Visual Objects GUI application has an *owner*. Each owned object, by convention, can tell you who its owner is via a property called Owner. This is a very important concept that controls much of the action in the application.

Consider a typical MDI application, with a *shell window* and a number of *child windows*. The shell is owned by the *App*, an invisible object that controls the basic event processing of the system:



On the window sit *controls* of many types, from simple text fields (also called edit controls) to push buttons and spreadsheet-like tables:



Windows also own menus and toolbars in a similar structure.

Meaning of Ownership Relationships

The ownership relationships hold the key to many important aspects of the application, from message routing, to the display of prompts and diagnostics, to error handling.

In an object-oriented application, these relationships serve much the same role that the call stack serves in a procedural program. Under event-driven execution, the action routines are called from the *dispatcher*, so returning up the call stack does not lead to a component with larger knowledge of the application. To reach the higher authorities, those parts of the application with broader knowledge, you look to your *owner* rather than your *caller*.

Visual Signs of Ownership

Ownership is not necessarily reflected in the visual display: while an MDI child window can *own* a menu, under the Windows convention this menu is actually *displayed* in the shell window. The ownership affects operation: as different child windows are created or switch focus, the menu in the shell window changes to reflect the type and status of the *currently active* child.

(Of course, in most cases the controls that a window owns are also displayed within the window, and the child windows that a shell window owns are displayed within it. Such a visual relationship is common, but not necessary, to an ownership relation.)

Ownership vs. Inheritance

Note that in the preceding diagrams, the relationship depicted is *ownership* and not *class hierarchy*. The distinction is represented by two different verbs, *has-a* and *is-a*. "A shell window *has a* child window" is a correct statement, but "a shell window *is a* child window" is incorrect.

Ownership is a dynamic, runtime relationship between *objects,* while inheritance is a static, compile-time relationship between *classes*. Both ownership and inheritance are one-to-many: each object has only one owner and each class has only one parent, but an object can own many things and a class can have many subclasses.

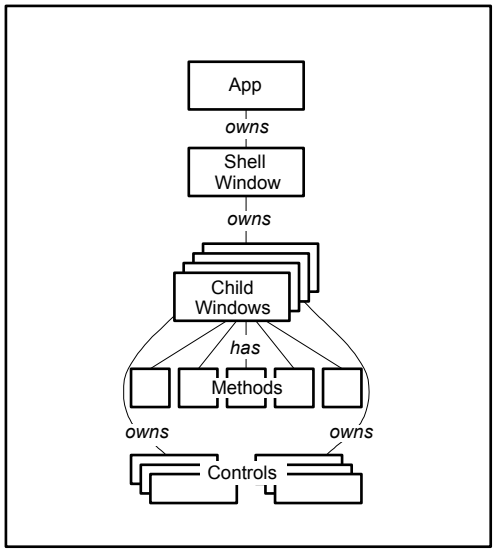
Client-Server

Another important relationship is *uses-a*, which reflects the client-server relationship (a client *uses a* server). You can say, for example, that "a child window *uses a* data server." Like *has-a*, *uses-a* is a one-to-many (each client can have only one server, although each server can have many clients), runtime relationship, but it does not necessarily imply ownership. (These and other relationships are discussed in greater detail in Chapter 11, "GUI Classes".)

Event Generation and Handling

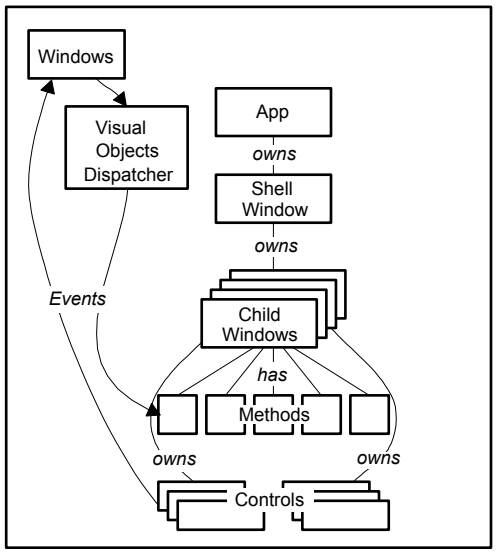
In a Visual Objects application, the controls that sit on a window (such as push buttons, list boxes, or menu items) own data and initiate *events*. For example, clicking on a push button generates a command event.

It is important to note that the controls themselves do not perform any serious processing logic—that belongs instead to the *event handlers*, or *methods*, of the window that owns the controls:



It is natural and correct to think of the controls as *generating* the events. However, internally, events are actually directed first to Windows and then to Visual Objects before being dispatched to the window and ultimately its event handlers for processing.

This idea is illustrated below:



For example, since clicking on a push button generates a command event, a push button labeled "OK" might be used to close a dialog window after the user is finished with it. To handle the command event generated by this push button, its owner dialog window would need a method that would take care of shutting itself down.

A window can have other components that generate events or display information, such as a *menu*, a *toolbar,* and a *status bar*. As with controls, these are secondary—the window does the processing. The Visual Objects dispatcher, in combination with the default methods of the GUI classes, ensure intelligent routing of the events to the object that is interested..

Smart Windows, Dumb Controls

This arrangement—controls generating events and windows processing them—is one of the key structural principles of the GUI classes provided by Visual Objects: that of "smart" windows and "dumb" controls.

This is not the only possible way to arrange the logic of an application (it is possible to attach event handlers to controls). But this approach has several important advantages:

- It facilitates data sharing within the processing logic.
- It sets up natural scoping and referral rules.
- Perhaps most important, it fits with a traditional concept of what a program is and thus minimizes the conceptual relearning required. Because the windows own both the data and the code that works on it, everything is naturally connected.

Do not look down on the dumb controls, however. They are not all that dumb. For example, many controls, such as push buttons, edit controls, toolbar buttons, and menu items, have a Description property, which is a textual prompt that appears in the status bar. When a user goes to one of these controls, the control signals this *focus change* and causes the prompt to be displayed in the status bar. These controls also have a Help Context property that can be used in an context-sensitive, online Help system. When a user requests context-sensitive help about one of these controls, the control starts up the help system at the appropriate topic.

Types of Windows

There are six main kinds of windows, each of which is implemented as a class in the GUI Classes library. They are introduced individually in the following section.

TopAppWindow

TopAppWindow is the top window of a Single Document Interface (SDI) application. It is structured around displaying one document (for example, file or record) at a time.

SDI applications are the closest to traditional DOS applications and are relatively rare in GUI environments.

ShellWindow

ShellWindow is the main window in an MDI application. MDI is the Windows convention for structuring an application around the presentation of multiple documents simultaneously in many windows.

MDI applications typically use a shell window (based on the ShellWindow class) 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 and are typically derived from either the ChildAppWindow or DataWindow class.

Note: While MDI is a specific convention for Windows, other GUI environments support applications that are structured in a similar way from a logical perspective, even if the visual presentation is different.

ChildAppWindow

ChildAppWindow is a "child" window in either an MDI or an SDI application. In the former, child windows are usually owned by the shell window and in the latter by the top window.

DialogWindow

DialogWindow is a secondary, usually transient, child window used to collect or display utility information. Dialog windows can be modal or modeless. They are most often *modal*, which means that the user must respond before the application can proceed (menus and controls in other windows are unavailable while a modal dialog window is open). *Modeless* dialog windows do not impose this constraint and can, therefore, stay open longer.

Any kind of window can own a dialog window, including other dialog windows (the standard Print dialog which can usually call a Setup dialog is an example of nested dialogs).

It might appear that the difference between a modeless dialog window and a child window is small, and, visually, this is indeed the case (although there are some visual clues such as the standard border treatment).

The main difference lies in how they behave by default and in how they are intended to be used. For example, the child windows currently open in an MDI application are typically listed in the Window menu and can be rearranged with the Tile and Cascade menu commands. This is not the case for dialog windows—whether modal or modeless, they are independent of the MDI structure.

DataWindow

DataWindow is used to represent a *data-aware* window: this type of window can optionally be connected, or linked, to one or more databases. When connected to a database, a data window "knows" about the database upon which it is intended to operate: the controls that sit on the window—simple edit controls as well as spreadsheet-like tables—are connected to fields in the database. Furthermore, the data window has built-in behavior for standard operations, such as Skip Forward and Skip Backward, Go Top and Go Bottom, and Delete and Insert.

In most database applications, the data windows represent a significant portion of the business logic; therefore, this type of window is discussed more fully in the next section.

DataDialog Window

DataDialog windows can be created either be modal or modeless. They can also have a data server attached to them. There is no real benefit to creating modal DataDialog windows because their behavior would be almost identical to that of Datawindows.

Using Data Windows and Data Servers

A data window can be used as a top window, a child window, or a dialog window. When it is instantiated, the data window determines its context and creates the type of underlying window it needs. Thus, if the owner of the data window is an application, it acts as a top window; if the owner is a shell window (which is the most common configuration), it acts as an MDI child window; in other cases, it becomes a dialog window.

This is important and bears repetition: once you have defined a data window, you can use it all by itself, creating a simple application with only this window. Or you can combine it with other windows and allow opening as many copies as you want under a shell window. There is no difference in how the window is defined: short of ensuring that the database is opened in shared mode, no special effort is required to allow the window or its database to be opened many times.

A data window acts as a combination of a dialog window and an application window: internally, the display and navigation among its controls are handled automatically by the dialog manager, but the external behavior is handled like the corresponding top window or child window.

Data Links

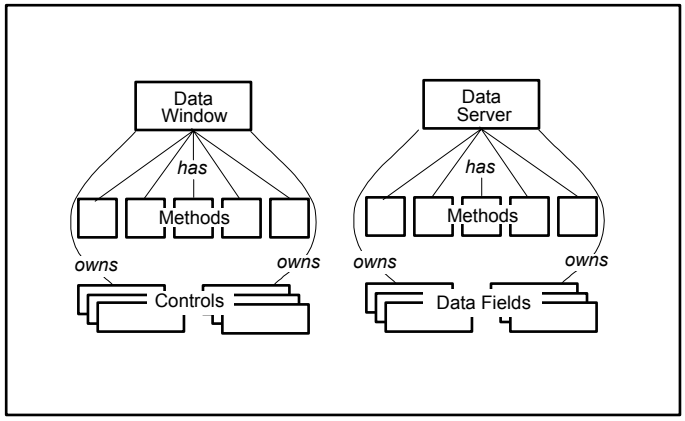
When a link is established between a data window and a data server, each control on the window is automatically matched with each field of the database based on common names. In this example, the CustomerWindow and CustomerDB classes have been created by the Window and DBServer Editors, respectively; you simply tell the window to *use* the data server, and the link between the CustNo control and the CustNo field is established automatically:

```
CLASS CustomerWindow INHERIT DataWindow
     // Generated code...

CLASS CustomerDB INHERIT DBServer
     // Generated code...

METHOD OpenCustomerWindow() CLASS EmptyShellWindow
     LOCAL oCustomerWindow, oCustomerDB AS OBJECT
     oCustomerWindow := CustomerWindow{SELF}
     oCustomerDB := CustomerDB{}
     oCustomerWindow:Use(oCustomerDB)
     ...
```

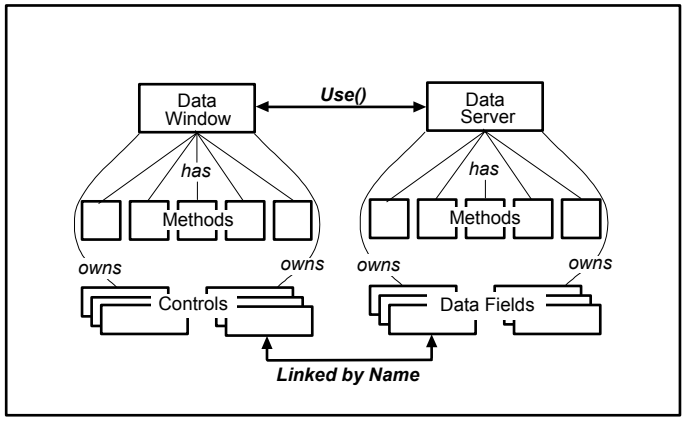
Once this relationship has been cemented, you can refer to oCustomerWindow:CustNo and oCustomerDB:CustNo, and get the same value. Because the fields are linked, changes in one are reflected in the other.

Parallel Structure

The data window and the data server have a parallel structure. They represent a record as a whole, an aggregate of data elements, and contain components that represent individual data elements. The relationships between the components are exactly the same on both sides:



When you link the two, the link relationships between them mirror this structure:



Business Processing

The business logic of your application sits in the event handling methods of the window, which is where most of your classical Xbase language will go. Since the window also owns the controls, this means that the business logic can use data fields without requiring complex, object-based referencing. This was a key objective of Visual Objects: to allow the business logic to have a regular appearance without requiring new, object-oriented styles of referencing fields and variables.

For example, if you want a method for calculating and displaying tax withholdings for income fields in the data window, you would create a push button and associate it with this method:

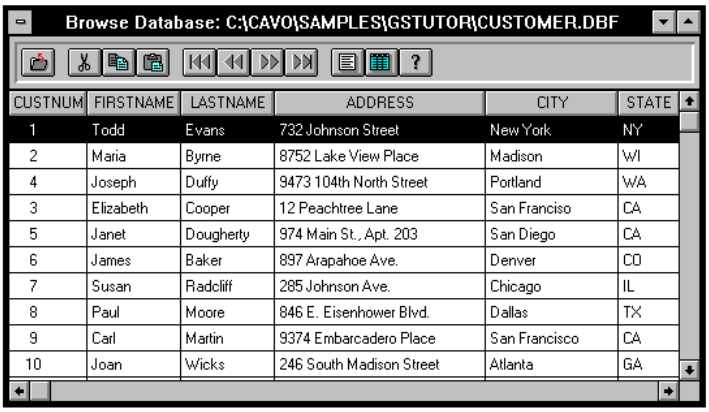
```
METHOD CalcSalesTax() CLASS CustomerWindow
    SalesTax := TaxRate * (TotalSales + Shipping)
```

Because this is a method of the data window, the references to the values are correctly identified with the controls on the window which, in turn, are linked to the database fields.

In fact, the control values are ACCESS and ASSIGN methods rather than instance variables. They intercept the assignment or reference and ensure that the values are properly propagated to and from the data server, just like the Get objects do in traditional @...SAY...GET processing.

DataBrowser: A Spreadsheet-Like Table

DataBrowser is a class that lets you represent data using a spreadsheet-like table, often called a *browse view*. It is a very powerful facility with extensive built-in behavior: it handles scrolling, editing, inserting, and deleting automatically and provides for resizing and rearranging of columns, as well as a split bar for dividing the table into two independently scrollable parts. Both keyboard and mouse interfaces are fully automatic.



CUSTNUM	FIRSTNAME	LASTNAME	ADDRESS	CITY	STATE
1	Todd	Evans	732 Johnson Street	New York	NY
2	Maria	Byrne	8752 Lake View Place	Madison	WI
4	Joseph	Duffy	9473 104th North Street	Portland	WA
3	Elizabeth	Cooper	12 Peachtree Lane	San Francisco	CA
5	Janet	Dougherty	974 Main St., Apt. 203	San Diego	CA
6	James	Baker	897 Arapahoe Ave.	Denver	CO
7	Susan	Radcliff	285 Johnson Ave.	Chicago	IL
8	Paul	Moore	846 E. Eisenhower Blvd.	Dallas	TX
9	Carl	Martin	9374 Embarcadero Place	San Francisco	CA
10	Joan	Wicks	246 South Madison Street	Atlanta	GA

A data browser is populated by DataColumn objects, which hold the data and provide formatting, validation, help, and so on. The behavior and programming interface of the DataBrowser class is patterned on that of the TBrowse class introduced by CA-Clipper, although it offers more display flexibility.

A data browser is very much like a data window from an internal viewpoint (with respect to such things as data linkage and handling of many data-oriented events), but it is more like a control from an external viewpoint (with respect to such things as control flow and message traffic).

Form and Browse View

A data browser is much like a data window. A data window can switch between *form view*, the standard data window view in which one record is displayed in the window, and *browse view*. The standard data browser view displays multiple records in the window. This feature, which is activated by a message like any other data window action, is often invoked by View Table and View Form commands on the main menu. Of course, you may choose not to provide this option for a particular window, making it available in only one view or the other.

Advance #1: Automatic Layout

When switching from a predefined browse view into form view or vice versa, there may be no predefined layout—you may have defined only one layout for this window. In that case, the system automatically generates a default layout for the form view or the browse view, as required. Indeed, it is possible to instantiate a view that has no predefined layout, in which case both layouts are automatically generated. It is also possible to define both layouts explicitly.

If you look at the program presented earlier in this chapter that linked the predefined CustomerWindow data window to the predefined CustomerDB data server, you can see how you could have done it without any subclasses at all, with completely automatic self-configuration:

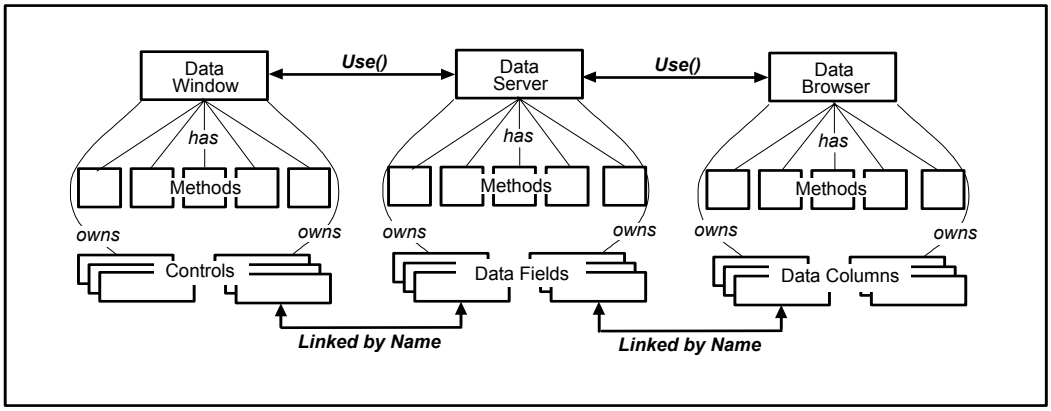
```
METHOD Start() CLASS App
     LOCAL oCustomerWindow AS OBJECT
     oCustomerWindow := DataWindow{SELF}
     oCustomerWindow:Use(DBServer{"customer"})
     oCustomerWindow:Show()
```

Parallel Structure

A data window has controls, and in a typical database-linked window most of the controls are linked to specific database fields. Similarly, a data browser has columns, and most of them are linked to database fields.

This similarity reaches further: a column responds to the same data-oriented messages as a control, including validation. A column also has the same annotation features as a control: a caption, a description/prompt message, a help context, and so on.

Thus, there is a parallel structure among all three types of objects: the DataServer, DataWindow, and DataBrowser on the higher level, and the fields, controls, and columns on the lower level:



FieldSpecs

The properties of a database field are grouped together in a FieldSpec object, which includes data type and size, annotation such as a caption (label), a prompt and context-sensitive help, formatting, validation rules, error messages for the validation rules, and help for the error messages.

But these properties are also of interest on the GUI level. In fact, controls on a data window and columns on a data browser also have their properties grouped in such FieldSpec objects. The developer can provide a field specification explicitly for the control or column. This is the approach most similar to traditional Xbase programming, where VALID and PICTURE clauses are specified with the @...SAY...GET statement.

Advance #2: Inheritance of FieldSpecs

But unlike the Xbase Get system, the control and column automatically pick up a field specification from the database field they are linked to if none is provided. On the next level, if you provide no explicit field specification for the database fields, they generate a standard one based on record layout. Thus, none of this infrastructure of documentation and processing rules is required: simply opening an existing database and linking it with a data window will work. The FieldSpec extensions represent *opportunities,* not *requirements,* to manage data, its presentation, and its behavior in an organized way.

Sub-Data Windows

A data window can also be used as a subwindow by placing it on another window. The sub-data window is designed independently using the Window Editor, and may be displayed either in Form View or Browse View. It can be linked to a database and has its own processing logic. But from the viewpoint of the host data window that owns it, it behaves like a complex *custom control*.

In other words, from an internal perspective (with respect to such things as data linkage and handling of many data-oriented events) a sub-data window is like any other window. From an external perspective, (with respect to such things as control flow and message traffic), a sub-data window is like a control.

Like any data window, a sub-data window can be switched between browse and form view. In practice, a very common approach is to limit a sub-data window to the browse view. This is useful in producing a standard master-detail relationship, such as the classical Customer-Orders.

Sub-data windows may be placed on data windows only, not on other types of windows that are not data-aware. Only a data window knows what to do with a sub-data window.

Note that sub-data windows are not a special kind of window. Any kind of data window can be used as a sub-data window or as an independent window. You specify how you want the window to behave when you create it: normally it is an independent window, but if you give it a placement parameter (a position or a resource ID) like a control, it becomes a control.

Command Events

Command events are generated by four sources: push buttons, menus, toolbars, and accelerators.

- A push button can be placed on any dialog or data window and is directly linked to a particular command event.
- A menu can be owned by a TopAppWindow, ShellWindow, or an MDI-type ChildAppWindow, or by a DataWindow that is not being used as a sub-data window. Note that sub-data windows and dialog windows cannot have menus.
- A toolbar is logically like a menu. It is owned by the window and generates command events. However, a toolbar differs from a menu in that any window or sub-data window can have one.
- An accelerator is just a keystroke sequence that is associated with a particular menu item; the menu item does not even have to be visible on any menu, and thus an accelerator can be seen as a direct keystroke sequence for generating a command event.

Event Routing by Name

By default, command events are linked to a method of the owner window through a symbolic name. Thus, each menu item and each push button are given a symbolic *event name*, and the window has a method of that name. For example, on the File menu there is a menu item called Print; it is automatically linked to the Print() method of the child window through its symbolic name #Print.

(The native Windows resource IDs, 16-bit integers, are required by the compiler but are not exposed to the developer. Routing is done by symbolic name.)

Control Flow

A command event is sent first to the lowest level window that has focus. If that window does not want to deal with the event, it passes the event up the ownership chain to its owner, who passes it on up. If no one wants the event, it will do nothing.

This automatic propagation is quite useful. In an MDI application, for example, the File Save and File Print menu commands should be handled by the specific child window that handles each document, while File Open and File Print Setup should be handled by the shell window.

Multiple Instantiation

As described so far, the structure provides little more than the Xbase Get system. In a graphics mode: you can design data windows, place controls on them, and link database fields to the controls in such a way that data is automatically propagated back and forth between window and database. You can also specify validation rules for the fields.

Advance #3: Multiple Windows

The major advantage with this object-oriented approach, however, is that the entire structure can be implemented many times. For example, the simple little program that creates a data window and links it to a data server can easily be converted into an MDI application that allows you to open any number of customer windows with different records in them. For the sake of simplicity, a menu item is added to the system menu:

```
METHOD Start() CLASS App
     LOCAL oWindow, oMenu AS OBJECT
     oWindow := ShellWindow{SELF}
     oMenu := oWindow:EnableSystemMenu()
     oMenu:AppendItem(100, #New)
     oWindow:Show()
     SELF:Exec()               // Start the app!

METHOD New() CLASS ShellWindow
     LOCAL oCustomerWindow AS OBJECT
     oCustomerWindow := DataWindow{SELF}
     oCustomerWindow:Use(DBServer{"customer",       ;
          DBSHARED})
     oCustomerWindow:Show()
```

Note: In this example, the DBServer object is instantiated using DBSHARED as its second argument to allow the database file to be opened multiple times. In your own application, you can specify DBSHARED in the data server's Init() method so that when the data server is instantiated, its open mode will be properly set. Specifying DBSHARED overrides the default behavior of all Visual Objects applications, which is to open files exclusively unless otherwise specified. Another solution is to SetExclusive(FALSE) in the application's Start() method—this changes the default open mode to shared.

The key to this application is the name *New*. The new system menu item has a label #New. Selecting this menu item will automatically invoke the New() method (as described earlier in the Event Routing by Name section), which creates a customer window and links it to a customer data server.

Both the DataWindow and the DataServer classes are built with multi-instance support. No matter how many new customer windows you open, they will operate independently, with the database opened multiple times in separate work areas until you run out of work areas or DOS file handles.

This is a real advance over the traditional Xbase approach.

How much will this program do? With just a few short lines of code, it will allow you to do the following:

- Open the customer databases in independent windows. The databases will be shown in browse view.
- Allow you to browse up and down in the database, rearrange the columns, and modify the database by just typing into the fields.

Twelve lines of code produce a multi-window browser/editor for the Customer database. A few more lines will allow you to dress it up with a real menu and a few other things.

The Standard Application

The ShellWindow class provides a basic shell window with no frills, something you need to have available. But you will often build standard applications that follow common practice. For example, MDI applications usually have a menu with File, Edit, View, Window, and Help items on it, and these menus have standard items, standard accelerator keys, and so on.

Advance #4: Built-in Behavior

Because this is a good starting point for most typical programs, the system can generate a default framework for you, called the Standard Application. With the Application Wizard, simply accept all the defaults when creating a new application, and you get a working standard application. With the Application Gallery, select StandardMDI on the Standard tab page and then click OK.

The Standard Application provides a higher level of built-in behavior by creating a subclass of ShellWindow called StandardShellWindow and implementing several methods that serve as event handlers for standard menus. (There are two menus, EmptyShellMenu and StandardShellMenu, that are also part of the Standard Application.)

In the Start module, the StandardShellWindow is instantiated as follows:

```
METHOD Start() CLASS App
     LOCAL oMainWindow AS StandardShellWindow
     oMainWindow := StandardShellWindow{SELF}
     oMainWindow := Show(SHOWCENTERED)
     SELF:Exec()
```

Note: When creating the Standard Application from the Application Gallery, there is an extra line of code after the LOCAL statement, SELF:Initialize( ), which has no bearing on this discussion.

Using the Standard Application, you do not have to bother adding a New item to the system menu. The StandardShellWindow has a fully functional system menu with standard options, such as Restore, Move, Size, Minimize, and Maximize. It also has a toolbar and standard menu commands, such as File Open, Close, and Print Setup, Edit Cut, Copy, and Paste, View Form and Table, and Window Tile and Cascade.

The File Open command is similar to the New system menu item in the previous example. It presents you with a directory list box from which you can open any database file in a self-configuring data window that defaults to browse view.

The Standard Application is a good prototype for a real-world application. It comes with a large amount of built-in behavior. For any operation that you want to add, simply write a method for the StandardShellWindow class and hook the method, by name, into the appropriate menu using the Menu Editor. For any operation that you do not want, simply remove it from the menu definition and, optionally, delete the corresponding method. For any operation that you want to change, simply edit the appropriate event handler method to meet your specific needs.

Database-Oriented Actions

You have looked at data field linkage between the window and the database. What about data operations, such as navigation through Skip and Goto or actions such as Delete and Append?

The data server has methods with these names, so you can write a method of the window that invokes these methods for the connected server:

```
METHOD GetRidOfJones() CLASS CustomerWindow
     DO WHILE !SELF:Server:EOF
          IF SELF:Server:LastName == "Jones"
               SELF:Server:Delete()
          ELSEIF SELF:Server:Credit > 1000
               SELF:Server:Credit += 500
          ENDIF
          SELF:Server:Skip()
     ENDDO
```

However, the DataWindow class also has methods for the basic database operations:

```
METHOD GetRidOfJones() CLASS CustomerWindow
          DO WHILE !SELF:Server:EOF
          IF LastName = "Jones"
               SELF:Delete()
          ELSEIF Credit > 1000
               Credit += 500
          ENDIF
          SELF:Skip()
     ENDDO
```

There are some advantages to using these window methods instead of the database methods. Perhaps the most important benefit is that the window is aware of what is going on and can respond accordingly. You can also configure the data window to handle the Skip() operation differently when changes have been made to the data, if it automatically updates the database, discards the changes, or presents a dialog box and asks the user if the data should be saved or discarded.

The data window provides extensive built-in intelligence to handle many practical situations like this. The most sophisticated is automatic event notification.

Event Notification

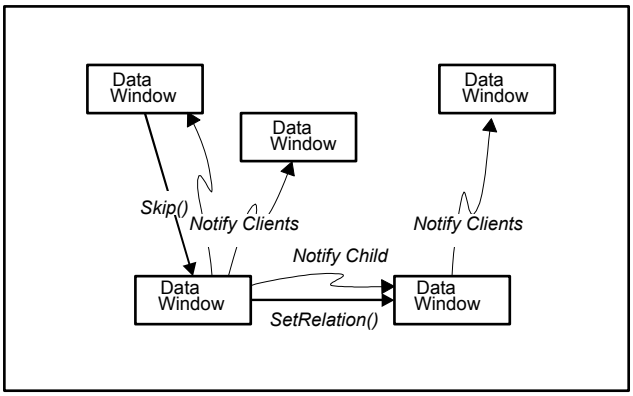
In an MDI application, you often need to coordinate between independent windows. If two windows show data on the same record or on records in separate but related (linked) databases, updates and movements in one window should be propagated between them. For example, if you have a general customer review window and the ability to open a customer credit history window for the same record, skipping forward and back in the main window should be reflected in the other window.

It is possible to do this manually, by one window calling the other one whenever data should be propagated. Indeed, this is the standard approach in many development systems. But this approach has some serious disadvantages:

- It is a hassle.
- It tends to dictate a particular structure so that data and actions need be propagated in one direction only. For example, that the general customer record is the main window and the credit history window is secondary.

  But what if the credit history window can be used as a child window from the general window, as a child window from the order processing window, or as a stand-alone utility? Who should communicate with whom? And even among the two of them, who instigates the action? Who sends the instruction to whom?
- It is very difficult to handle arbitrarily complex, dynamic situations. The different windows need to learn to communicate with different windows.
- It is difficult to add a new feature to the system. If you add a customer sales history window, you need to modify lots of other pieces of code to insert the communication handles. It prevents you from dynamically extending a system without destabilizing changes.

The automatic event notification facility of the class libraries handles all these situations without requiring explicit coding. Windows are notified when there is an action in a data server to which they are linked, whether the action originated in the current window or in another:



Automatic Data Propagation

Changed data is automatically sent to the data server before a move and copied back up to the window after a move, in the Xbase style. However, the data window supports several other modes of behavior, including:

- Save, discard, or prompt before a move
- Conservative or optimistic concurrency control, with single record locks, multiple record locks, or no record locks but reread-and-verify

Advance #5: Event Notification Among Windows with Shared Data Servers

If two or more windows share a data server, changes are automatically propagated among them, regardless of where they originated. If a user types in one window, it is reflected in the other; likewise, if a user invokes a Skip() method, a Delete() method, or a Delete(<*scope*>) method, all windows reflect the change.

In all these cases, the windows and data server coordinate by exchanging event notification messages. For example, after a field change, the windows are given a NotifyFieldChange message; after a single record operation, such as Skip(), they are given a NotifyRecordChange message; after a wholesale change, such as Delete(<*scope*>), they are given a NotifyFileChange message.

Similarly, before a move is made, the windows are given a NotifyIntentToMove message to give them a chance to save their data or prompt the user.

The data server keeps track of its clients so that all interested parties are informed of the action, regardless of where it originated. Because there is no direct relationship between the different windows (they are related only indirectly, through their server), new windows can be added to an existing application without undue effort or risk.

Advance #6: Event Notification Among Windows with Related Data Servers

If two Xbase databases have a SetRelation link, a movement in one will affect the other. In this case, the event notification will cross between the data servers and indirectly find its way to all the affected windows, automatically.

Visual Development Tools

Visual Objects is a complete Windows and Windows NT development environment, but it is also designed to provide you with a migration path for your character mode applications. To make the move from character mode to Windows, you must truly rethink the way you program. Exploiting the visual development tools to the fullest extent possible is your key to making this transition.

The visual development tools (such as the Window Editor, Menu Editor, and DBServer Editor) allow you to lay out visual elements, define data elements interactively, and generate the code that embodies the design. The code generators are built to leverage the class libraries described above. They generate subclass definitions (essentially, a class declaration, a resource, an Init() method, and a number of ACCESS/ASSIGN methods) to instantiate the window or menu. The action methods can be methods inherited from the standard classes or custom code—you enter the code using the Source Code Editor, which is invoked by whatever editor you are using. In any case, these methods are straightforward code and do not involve any generation.

The code generated by these tools is not only useful because you are saved the step of writing it, it is also of pedagogical value. Painting a window produces a usable program, a starting point for modification into a complex program, an educational sample, or all of the above. Many of the topics discussed in this chapter are covered in greater detail in Chapters 4 through 19, which include parts on database programming, user interface programming, and other topics such as exception and file handling. Using these resources and what you have already learned, you should be well on your way to meeting the new programming challenges.

Chapter 4

Standard Components—Classes, Objects, and Libraries

In this chapter, you will see how Visual Objects provides a *framework* for constructing, modifying, and deploying standard components, and includes a large library of components that meet many of the common needs of applications. You will also see how it provides a flexible architecture in the form of a standard framework, a skeleton of control flow on which you can hang the muscles of your specific business solutions.

Note: If any of the OOP terminology used in this chapter is unfamiliar to you, refer to Chapter 25 "Objects, Classes, and Methods." That chapter will also provide you with implementation-level details of OOP in Visual Objects.

Why You Need Components

Components + Relationships + Logic = Applications

What are components? What relationships do they have? What is architecture? How do you construct software quickly? How do you construct software that stays constructed? Where does the business logic go?

Components

When constructing a building, you always use prefabricated components: girders, door frames, sink units. When constructing a computer, you assemble prefabricated integrated circuits, power supplies, disk drives. Nobody would consider producing such complex artifacts from concrete and raw lumber, from transistors and sheet metal.

Yet, when constructing software applications, often equally complex, the tradition of using prefabricated components is not as well established. Using components improves both productivity and quality of software construction. But you need some tools and techniques to make it practical.

Along with OLE Automation, OLE Controls (OCXs) are perhaps the most obvious example of software component reuse. Controls represent a major advance in component software since they represent well-defined, encapsulated behavior that can be reused in multiple development environments independently of the language used for programming them.

Architecture

Just as you would not consider a complex construction project without the use of components, you would not start assembling a building or a computer without a plan—an overall architecture—that guides the entire construction. The success of a project often hinges on the quality of the architecture, and this is as true of software as of any other discipline.

Yet, software construction differs from many other human endeavors in its need for continuous adjustment and elaboration. For this reason, the standard components must be easy to modify and adapt to new uses and circumstances, and the architecture must allow the continuous rearrangement of components and addition of new components.

What Is Architecture?

Architecture is the set of principles by which you construct an application. It both dictates and depends on the set of tools, techniques, and prefabricated components that together help the application to grow along its architectural guidelines.

The primary goal of the architecture described here is to enable you, the developer, to create applications that can sustain *incremental enhancement*. Incremental enhancement is the principle and technique by which you iterate through these steps:

1. Start with an application in which you have complete confidence
2. Make a small change to it
3. Finish with an improved application in which you still have complete confidence

What Are Components?

Software *components* used to consist only of functions, or libraries of cooperating or related functions. The set of DML functions in Xbase is an example of a widely used software component.

Visual Objects offers other types of components, such as classes. In fact, it supports the encapsulation of a whole collection of classes as a compiled library (technically, it is implemented as a Windows DLL). The interface to the library consists of just the list of properties and methods of the classes within.

Although libraries are discussed in the context of classes here, a compiled library can of course contain many other types of components (for example, functions, defined constants, and global variables).

Essentially, software components are the analog of door frames and window units in buildings. Constructing a building would take a lot more effort if you had to build every single component by hand. Similarly, developing a software application would be much more difficult without the help of some standard components. When you have finished reading this portion of the *Programmer's Guide*, you will agree that the set of components bundled with Visual Objects goes a long way towards automating the tasks that are common to a majority of business applications.

Superficially, software components fill the same role as prefabricated components in any other form of construction. They save you time because they implement large chunks of your application without your having to program them. But software components differ from doors and kitchen units in the fact that you can adapt their appearance and behavior.

Visual Objects specializes in managing and enhancing components. It supports classes and inheritance. This is *object orientation*.

Plugging Components Together

Software components are extremely valuable because they implement useful default behavior and because they support encapsulation and inheritance uniquely well. But, how do you plug them together to make applications?

Class Relationships

Classes relate to each other in several important ways, each of which is discussed in this section. The interesting effects of these relationships occur at runtime, when objects manufactured by the components cooperate with each other. To cause this cooperation, the components themselves—the classes and libraries—must conform to some minimal conventions.

Inheritance

Inheritance is also known as an *is-a* relationship. For example, if Employee inherits from Person, you would say that an Employee *is a* kind of Person.

Tree: A set of nodes that can be reached from a single branch-point, in exactly one way

Inheritance forms a *tree*: each class can have at most one ancestor that it inherits from, although that parent class may of course inherit from its ancestor, etc. Each class can have any number of descendants. (Many people use the term *class hierarchy* instead; this is correct but less precise, since a tree is a specific type of hierarchy.) The internal states of objects also form tree structures.

Ownership

Particularly in GUI programming, ownership is an important relationship. Suppose FancyWindow, a subclass of DataWindow, owns a number of controls, such as push buttons. Syntactically, the controls plug into the DataWindow by the fact that you (or the Window Editor) declare them as internal variables within the FancyWindow class. Dynamically, you or the Window Editor must construct each control in the Init() method of the FancyWindow class.

Ownership brings with it the responsibilities of controlling the lifetime of, and handling the exceptions raised by, the owned objects.

Ownership is also known as a *has-a* relationship, because a FancyWindow *has a* PushButton that it manages, a Customer *has an* Order, or an Order *has a* line item.

In the case of the Visual Objects components, ownership is taken one step further. Each owned object, by convention, remembers who its owner is. In fact, it has an access called *Owner* that returns the object's owner, whenever you ask it to.

The special keyword, SELF, refers to the object itself. In the following typical code, SELF is the object, for example FancyWindow, that is creating the push button and is therefore its owner:

```
oPushButton := PushButton{SELF, SOMEBUTTON_ID}
```

Code using the push button can then say something like:

```
oPushButton:Owner:Close()
```

to close the FancyWindow. You can regard owned objects as constituents of their owner. The owned objects are sometimes called *subobjects*. Subobjects implement the internal state of an object. (Do not confuse subobjects with subclasses. Subclasses are subordinate to this class in an inheritance hierarchy; subobjects are subordinate to this object in the ownership tree at runtime.)

Client-Server

The client-server relationship is used typically for the relationship between applications and databases. The database object is the server and it can have many clients. The clients cause changes to the database by invoking methods of the database object. The database notifies its clients whenever its data has changed by invoking their methods.

Conventionally, the Use() method establishes the link between a client window and its data server. This is called a *uses-a* relationship, because a client *uses a* server.

Each client can have only one server, although each server can have many clients. Thus, the client-server relationship also constitutes a tree.

In the following typical code, the server, *oDBServer*, has already been created and the DataWindow links to it by invoking its own Use() method:

```
SELF:Use(oDBServer)
```

Database Relations

The classical technique for linking tables in Xbase databases is the *relation*, which establishes that movement in one table causes corresponding movements in other tables. There is a one-to-many relationship: from one table, you can set up relationships to many tables, but each table can be the child in only one relation.

Thus, this classical structure also forms a tree.

Importance of Tree Structures

Each of the fundamental structures in the Visual Objects application architecture is a tree.

- The inheritance tree builds up at compile time, and you can walk it using the Repository Explorer when grouped by class view. The Repository Explorer draws many trees on their side (that is, fallen over), with each root on the left and branches that fork to the right.
- The ownership tree builds up at runtime, as your application creates objects which, in turn, create more objects.
- The client-server tree builds up at runtime as you connect windows to data servers.
- The database relation tree builds up at runtime as you define relations between databases.

If you store references to objects in state variables of other objects, without any specific rules (or *architecture)*, you have something which is no longer a tree. Technically, it is called an arbitrary graph. Trees, which are a special, constrained kind of graph, are easier to understand and handle than arbitrary graphs.

Branch-point: A node that contains a list of nodes

Node: A leaf or a branch-point with no further structure, as far as the tree is concerned

Structuring an application as a number of trees makes its behavior more predictable because the most common programming error is to attempt to use an object outside its lifetime. In the tree structure, each *branch-point* is responsible for creating and destroying the *nodes* that it contains. Therefore, the lifetime of objects follows the tree structure. If the classes are also well-encapsulated, an object is not even visible outside of its lifetime.

Another common problem is memory or resource leaks, where the application gradually uses up more and more of the resources of the machine as it runs, eventually causing the system to grind to a halt. The automatic garbage collection scheme of Visual Objects takes care of the resources that are within its reach, but sometimes other resources must be freed as objects are destroyed.

In many environments, notably Windows, the sequence in which objects are destroyed is very sensitive. Again, a tree structure makes it simpler to ensure that all resources are freed in the correct order.

One last word about tree structures. The Xbase languages deal with tables. The primary concept is that you have a number of tables and you can add or remove rows in them. The tables are an elegant and efficient way of dealing with a large number of structurally identical things, such as customer records. The other kind of data structure found in Xbase is the array. Conceptually, an array is very similar to a one dimensional table, or list. Xbase does not provide any more complex structure than that.

In moving from Xbase to GUI development, you move from a realm where the main metaphor is a table—huge numbers of similar items—to one where there are small numbers of dissimilar items. There may be many dialogs in an application, but each has some unique behavior. This small population, high diversity world needs tree structures rather than tables. So Visual Objects programs maintain their data in tables, as in Xbase, but present their data through an architecture dominated by trees.

Summary

Keep in mind that the architecture of Visual Objects components is a convention, not an absolute law. To support these simple mechanisms for plugging things together, rigorous encapsulation and lifetime management of the components supplied with Visual Objects are enforced, and standard methods, such as Use(), are provided to strengthen the convention.

By convention, the important relationships between components are inheritance, ownership, and client-server. All of them form tree structures. One line of code suffices to establish any of these relationships. Built into the design of each component is the expectation that it will plug together with other components according to these conventions.

A Tour of the Visual Objects Components

The Visual Objects standard components mostly fall into two large groups: those that deal with the GUI and those that deal with the database. There are also other components designed to handle aspects of your application do not neatly fit either of these two categories. This short section gives an overview of all the components and how they cooperate.

Data Server Classes

The DataServer class is an abstraction of anything that supports both indexed and sequential access. It could be an ISAM file. It could be a database. It could be an array. It could be a DDE link to a server that supports this protocol. You can add your own subclasses of DataServer to support anything you like, so long as what you add behaves superficially like an ISAM file.

There are two built-in subclasses of DataServer: *DBServer* to support DBF databases and *SQLSelect* to support SQL databases. The methods of DBServer and SQLSelect fall into three groups:

1. Those that are inherited from DataServer. These methods are required for DataWindow and DataBrowser support—these objects cannot *use-a* data server unless it supports at least this set of methods.
2. Those that have common names between the two classes. These methods enable much of your code to be independent of the database technology you choose.
3. Those that are unique to one implementation or the other. For example, it is not meaningful to reindex an SQL database, so SQLSelect has no Reindex() method. You should isolate calls to these unique methods to small regions of your code, thereby isolating the code that you must change if you decide on a different database technology.

GUI Classes

GUI classes account for the majority of classes supplied with Visual Objects. Very roughly, the GUI classes include one class for every kind of object that you encounter in a GUI. There are Window and Dialog classes. There are Control classes, such as ScrollBar and PushButton. There are Menus and Toolbars.

Portability

Just as the data server classes make your application portable between DBF and SQL databases, the GUI classes make it inherently portable to any GUI—not just Microsoft Windows variants. The Visual Objects runtime system does not yet run on other GUIs, but the GUI classes ensure that any application code you write will have a good chance of being portable in the future.

SDI vs. MDI Windows

Every application must have a window (if you do not explicitly specify one, Visual Objects supplies one for you). Windows can own further windows, and that is how the application grows. The window you start with can be a ShellWindow or a TopAppWindow. Choose ShellWindow if you require Multiple Document Interface (MDI) behavior. Choose TopAppWindow for Single Document Interface (SDI) behavior. Either kind of window can own ChildAppWindows and DialogWindows. DialogWindows can own Controls.

Windows have two sets of methods.

1. Those that are invoked by your application code. This set of methods does things to the window, such as positioning it or drawing things on its canvas.
2. The *event handlers*. These methods are usually invoked by the Visual Objects dispatcher in response to some action that the user took, such as pressing a push button or choosing a menu action.

Data-Aware Windows

Supplementing these purely GUI classes, there are data-aware GUI classes. For example, DataWindow *is a* kind of ChildAppWindow that can be a client of a data server. The DataBrowser is a tabular display that may also be connected to a data server as a client.

When you develop an application to use a database, you can use the Window Editor to lay out data-aware windows that your application uses. The Window Editor generates code to manage the whole operation, including setting up the default behavior of a field based on the database server to which the field maps.

The Standard Application

It was mentioned earlier that if you do not specify a window in your application, Visual Objects will create one for you. Indeed, whenever you include the GUI Classes library in an application's search path, Visual Objects creates not only a window in which to run the application, but a complete, working application called the Standard Application.

The Standard Application exhibits standard MDI behavior. It allows the user to view one or more databases in table form. The user can change the *view*, causing the program to present the data in a self-configuring data window. In other words, the data window's default implementation is that it lays out a form automatically fitted to the data in the database—even though it has never seen the database before.

When you code your application to handle exceptions such as "the database layout changed, but nobody told me," you can fall back on the elegant default behavior of the GUI classes to respond to the user. Similarly, you can prototype much of an application's behavior using only the DataWindow and DataBrowser classes.

Classes for Annotation

There is an automatic system for escalating exceptions upwards through the ownership hierarchy, so that your application can deal with exceptions where they occur, but still provide more general handling if necessary. This mechanism works by connecting the Error class to the GUI Classes library's event system. (See Chapter 14, "Error and Exception Handling" for more information.)

DataField and FieldSpec classes provide for automatic validation and type checking, without changing the contents of existing databases.

The HyperLabel family of classes integrates the context-sensitive help system with your program. Hyperlabels trigger status prompts that appear on the status bar and hypertext topics that appear on the help window when the user requests help.

The GUI classes also implement a wide range of stock objects, such as useful icons, bitmaps, and colors.

Business Logic

As discussed in detail in the first part of this guide, a move from Character Mode to Windows requires a move away from traditional program structure. The event-driven nature of Windows (or any other GUI) requires you to use methods as event handlers throughout your applications. The modeless behavior required of world-class applications dictates that you create objects that respond independently to the events.

Modeless behavior means that the application never stops to wait for some specific user action—it is always able to respond to *any* user action. On very rare occasions, you must use modal behavior (for example, when you have failed to handle an exception and you require the user to make a decision before it is safe to do anything else).

Despite these sweeping and necessary architectural changes, your business logic remains surprisingly intact. The code you write to fetch data from the database, to validate data that you capture, and to commit it back to the database looks very similar to the code of traditional Xbase programs. The skeleton is entirely new, but the muscles of business logic that make it effective remain unchanged.

It is a fairly trivial matter to partition the source code of your application into classes and methods. That, in itself, gives you the ability to find code quickly and to control changes to the source code more effectively. The tough job consists in controlling changes to the dynamic structure of your application—the structure that it exhibits at runtime.

The standard components supplied with Visual Objects work to control changes to the dynamic structure. That is why this chapter has stressed the ownership tree so much. The goal of the Visual Objects architecture is to enable you to develop your applications by a process of safe, incremental enhancement.

OLE Controls

OLE Controls is a technology that allows standardizing of plug-in functionality for development tools. Regardless of the functionality that individual controls impart, all controls have a general architecture in common. For example, all controls have properties that can be accessed or assigned which direct their functionality. In addition, most controls also cause events to be communicated to the container (client) housing them based on input from the end-user. The OLE Controls specification is a formal attempt to standardize how this communication between a control and its container (your control-aware window) takes place.

OLE Automation Servers

OLE Automation Servers are applications such as Microsoft Excel which can be used as stand-alone applications and can also be controlled programmatically from any development tool that is an OLE Automation Controller (client). Let's say you program Excel to load the spreadsheet template, fill in the appropriate input variables as collected by the user interface portion of your application, and perform a recalculation returning the result for display back to your user interface. All of this can be done without accessing Excel's user interface. In other words, Excel can be launched and made to load and recalculate a spreadsheet completely invisibly. In this manner the business logic of Excel itself, which is the ability to manipulate spreadsheet data, has been exposed to anyone interested in using it in the form of an OLE Automation Server software component.

OLE Automation

OLE Automation is a technology designed to exploit the natural (logical) separation of user interface and business logic which occurs in software. The assumption is that although the user interface code typically represents the most expensive part of the application in terms of time, it is the business logic that is the most valuable code as it is the most likely to be reused. OLE Automation is a concentrated effort to package this logic as a separate software component so that it may be reused over and over again by any other client software component that needs it.

You Can Develop Components

You have seen that the standard components bundled with Visual Objects address many aspects of application development. Particularly, they address those areas that most business applications have in common. But they do not address aspects that would be specific to a particular industry type, such as banking or manufacturing.

You can appreciate the great benefits that standard components bring to application development. You can amplify these benefits and extend them to other areas of your development effort by designing your own components. You can also copy the overall style of the built-in system components, which are thoroughly worked out and tested.

Generally you should start by designing subclasses of DataWindow that uniquely address your industry. You can also very simply design subclasses of DataServer that attach to different data sources. You can use the concepts of *is-a*, *has-a*, and *uses-a* to extend the Visual Objects architecture into new areas. It requires some familiarity and experience to do this, but using the standard components as a role model, it is not so difficult.

Chapter

5

Object Linking and Embedding

In this chapter, you will learn how Visual Objects accommodates object linking and embedding (OLE). With the support of OLE 2 in Visual Objects, you can utilize a variety of prebuilt, third-party components. This chapter provides an in-depth explanation of what OLE is all about and how you can easily apply (including sample codes) linking and embedding, custom controls, and automation servers to your application.

OLE Overview

OLE is not a new technology—it existed before the release of Windows. It was used primarily in Microsoft Office applications to create compound documents—one or more foreign applications linked or embedded inside another application. For example, creating a Microsoft Excel spreadsheet file inside a Microsoft Word document enables you to directly change the linked spreadsheet file in the Microsoft Word document, if changes are made to the spreadsheet file.

Microsoft has now developed OLE 2, which is a central component of Windows. OLE 2 is a full 32-bit technology, which does more than facilitate compound documents. It includes subtechnologies, such as automation, data transfer, memory allocation, file management, OLE controls (OCXs), and other new technologies that are being developed. Visual Objects is a comprehensive OLE client, which supports all of OLE 2 technologies.

Component Object Model (COM)

A new architecture, called Component Object Model (COM), serves as the foundation for OLE 2. COM employs a binary interface, which allows components supplied by different Independent Software Vendors (ISVs) to interoperate in a reliable, controlled manner. It lays the groundwork for developers to build specialized software components that interface commonly.

With COM, software vendors do not have to exchange specifications, or in any way coordinate the design and assembly of their specialized software components. By merely adhering to the COM-based OLE standards, the different ISV software components will be able to interoperate automatically.

For example, it is not necessary for an OCX designer to communicate with the vendor of a specific OCX-aware window painter tool. The two components will naturally be able to interoperate as long as each follows the COM-based OLE standards. That is, the Window Editor (OLE Control "container") will be able to interrogate the OCX to determine what functionality it possesses and provide an interface to that functionality.

Basic COM Terminology

Components consist of one or more objects, and each object is a collection of one or more *interfaces*. Interfaces are sets of semantically related functions or *methods*. OLE is nothing more than a specification for a number of these interfaces. For example, if a component is to support the OLE drag-and-drop capability, it must implement the OLE interfaces which provide that capability (IDropSource and IDropTarget). Likewise, if that same application wants to support uniform data transfer under OLE, then it must implement the interfaces, which provide those capabilities (IDataObject, etc.).

So, at its lowest level COM is the specification for how an interface is defined, whereas OLE is a specification for a set of interfaces. COM specifies the details of how interfaces are structured in memory, whereas OLE defines a collection of useful COM interfaces.

All COM interfaces have the same general memory layout, which is consistent across all COM implementations, regardless of the operating system. Basically, a COM interface is nothing more than a pointer to a pointer to a function table, which contains pointers to the actual functions (the implementation) of the interface. While this memory layout may seem unusual, it is structured this way for historic reasons—namely because it mirrors the layout of a C++ object in memory. It is, therefore, not surprising that COM interfaces can be implemented directly using C++ objects, since that language has become the most popular development language in recent years among systems-level programming professionals.

However, it is important to understand that using C++ to implement a COM object is not required. COM is a binary standard, which means it does not assume any specific language. Instead, all it assumes is that the implementation language can construct a memory layout (as shown above) and be able to deference a pointer to a pointer to a function pointer.

Components are little more than a container of windows objects. In the simple case where a component is made up of only one object, the distinction between the two disappears and the terms component and object become interchangeable.

In any case, a developer never really works with an object directly. In fact, a developer can never have a physical pointer to an object. Instead, a pointer to an interface on the object is obtained. By obtaining such an interface pointer for the first time, the object is automatically and dynamically loaded and remains in memory as long as the interface pointer is still in use.

By formally hiding the physical underlying object from the developer and forcing him to access the object only by a published *interface* of functions, complete object encapsulation is achieved. The COM specification does not allow for the developer to access the state information of the object directly, because it does not specify how that state information is laid out in memory. This implementation detail is left up to the implementation of the object. Therefore, all access to object data must be done through the interface functions, providing a safe environment for the object to exist in as well as a greatly simplified COM specification.

COM as an Object-Based Model

It is important to understand that COM is not an object-oriented model. It is object "based". Therefore, *implementation inheritance* is not a property of COM. Rather, it is a model based solely on the idea of *encapsulating* behavior and data in such a way as to make it available in a standard binary form. While it is very possible that the internal construction of a COM component is object-oriented in implementation, the presentation of that component is not. In other words, while it is meaningful to speak of COM objects (or components), the COM specification does not provide for a way to create a new COM object by inheriting from an existing COM object.

This seeming omission is by design. Many developers believe that it is impossible to support implementation inheritance in an object model without placing restrictions on how the model can be used. Specifically, they assume that COM components are not constructed in any particular language. Hence, the standard is not language-specific but binary-specific. If a COM component is defined based on the definition of another COM component (also known as inheritance) then it would be possible for a relationship to exist between two components from separate companies. This would be fine until the vendor of one of the components decided to update their component. The problem is that the "contract" or relationship between components in an implementation hierarchy is not clearly defined; it is implicit and ambiguous. When the behavior of a component changes unexpectedly, the behavior of related components can become undefined.

An implementation hierarchy works well when all of the components in the hierarchy are under complete control, such being written by the same vendor in the same language. However, in a distributed environment where components come from different vendors written in different languages, and where source code is not available, implementation of inheritance is not a viable option. Inheritance violates the principal of encapsulation which is by far the more important aspect of any object technology. So, Microsoft has purposely left implementation inheritance capabilities out of the COM specification.

COM Interfaces

Component objects are programmed through interface pointers (pointers to pointers to function tables of pointers). Objects can be composed of several unrelated interfaces, each providing some useful functionality on behalf of that object. Interfaces simply represent "function sets"—logically related collections of functionality that are combined in a manner that allows them to be reused.

It is very important to understand that an interface is a specification of behavior only. It has no *particular* implementation. You can think of it as an abstract class, the individual services (functions) of which must be implemented by you. This means that it is entirely possible that different developers will implement different versions of the same interface. While the implementation details are free to change from developer to developer, and environment to environment, what must remain consistent is the intended behavior.

For example, consider a fictitious COM interface called *IStack*. Such an interface might have the services *Push* and *Pop* defined. The interface's formal specification would then detail such things as the order the function pointers appear in the function table. For example the pointer to the Push function in the first slot and the pointer to the Pop function in the second slot. It would also detail the semantic meaning of each of the functions. This would then represent everything a developer who required such an interface would need to implement it. Details such as the data structure to implement the stack (array, linked-list, etc.) and the language the implementation is written in, would be left entirely up to the implementation of the interface.

OLE and COM

So COM is a model by which developers can build objects made up of reusable interfaces. OLE, a collection of standard services, is defined completely in terms of COM interfaces. OLE is a broad specification covering many facets of windows programming. However, it is generally thought of as a collection of interface specifications for common Windows-specific services, which are most advantageous for Windows applications. These services include memory allocation, object persistence (structured data storage), uniform data transfer, object linking and embedding, and drag and drop.

However, OLE is not limited to the specification of these interfaces. In many cases, OLE has execution, which is part of the operating system in the form of DLLs. So, the developer does not have to implement all of the interfaces, but can use the default implementations instead. Nonetheless, consistent with the openness of the specification, you are still free to replace one of the default implementations with your own if you find it is not meeting your needs. In general, however, you will find their implementations perfectly adequate.

OLE is COM, plus a detailed specification of general-purpose Windows services in the form of interfaces, and some default implementations of those interfaces.

Issues of a Component-Based System

Let's review the design goals of COM.

There are four classic challenges facing component-based systems development:

- Interoperability
- Versioning
- Language independence
- Transparent remoting

COM was designed to overcome each of these difficulties.

Interoperability

A fundamental concern with the development of any component object model is compatibility. Components supplied by different ISVs must be able to interoperate safely without their developers having prior knowledge of the specifications of each other. Component software units require that other component software units have common required services.

COM solves this problem by making it possible for any object to *query* another object for a given service (interface) at runtime. For example, by being able to ask whether a software component supports drag-and-drop (implements the IDropTarget for example), another software component can dynamically determine how it should behave as an IDropSource in relation to that component.

If a particular object is made up of a set of interfaces, COM specifies that by obtaining a pointer to any one of them, a developer can query for and obtain a pointer to any other interface on that object. This works because all COM interfaces, by definition, support a *QueryInterface()* service which allows it to ask its owner (the object which holds the interface) if it holds any other interface. The result of the query is either "no such interface exists," or a pointer to the interface of interest is obtained. In this manner, given any interface on an object, a developer can find out if it supports—and then actually obtain a pointer to—any other required interface on the object by simply querying for it.

Using this query system, components can interrogate each other for common features and exploit them if found, or safely ignore them. By designing components to optionally use services that are available and to default behavior intelligently if those same services are missing, robust software components can be built that are reusable in many diverse situations.

Versioning

How can one software component be updated without affecting others that are related to it? How can you provide a new version of a component and still guarantee backwards compatibility? We have all witnessed at one time or another the versioning problems DLLs present in the Windows environment. Two applications from different vendors rely on a DLL from a third vendor and all three vendors are working with different versions of the DLL. When a more recent version of the DLL is overwritten by an older version because one of the application vendors is not up to date, code relying on the functionality of the most recent version of the DLLs breaks. This is currently a huge problem and it will only get worse in a component-based system unless a solution is found. Again, COM solves this problem.

COM provides a versioning mechanism that allows seamless evolution of components. When one component of the system is upgraded it is not necessary to replace, recompile, or even notify other components in that system of the change. This is accomplished by forcing interfaces to be *immutable*. This means that once you have published a specification for an interface you can never add to, delete from, or otherwise modify the functional specification of that interface in any way.

Consider our earlier example of the *IStack* interface. After designing this interface and publishing the specification internally within my organization, I may realize that I need to add some additional behaviors to the interface. For example, *Duplicate* and *Exchange,* which would allow me to duplicate the topmost entry on the stack and exchange the positions of the top two entries on the stack, respectively. However, the COM specification strictly forbids me to change the interface specification.

So, the only solution I have is to create a new interface called *IStack2,* which implements the original two services *Push* and *Pop* as well as the two new services *Duplicate* and *Exchange*. This may seem awkward at first, but in reality it is a very efficient solution. The new version of the stack implementation would then support not one but two interfaces, *IStack* and *IStack2.* Systems that were built using *IStack* and do not know or care about the new functionality of *IStack2* will continue to work as before. However, new systems that want to take advantage of the new *Duplicate* and *Exchange* functionality can query for and use *IStack2* instead.

It may appear that the forced duplication of common services will unnecessarily bloat future versions of interfaces. However, an interface designer can control all aspects of exactly how the interface is implemented and so avoid duplication of common services. All he must do is ensure that COM interface architecture (pointers and function tables, etc.) is adhered to. In the case of this new interface, the duplicated *Push* and *Pop* services will simply point to the implementation of the previous interface, thereby reusing that code and avoiding redundancy.

By strictly enforcing that interfaces are immutable, COM ensures that legacy systems do not break, since old interfaces are supported forever, and new users can query for and use new interfaces. The obvious benefit of this form of version control is that more care will go into the designing of interfaces to ensure robust specifications that are sufficient to avoid having to version the interface itself.

Language Independence

How is it possible for a software component written in one language to use the functionality of a software component written in another? COM ensures this by defining a binary rather than a source code standard for interoperable objects. This means it does not rely on any given language's object model. By employing an object model that can be supported by a number of different languages (including those that are not otherwise object-based, such as C), the choice of which language to write a component object in becomes an unimportant implementation detail.

Language independence is one of the advantages of using COM. COM objects are more reusable than source-code objects, such as C++, since those objects can typically only interact with other C++ objects. Even within a given language, different vendors can create subtle differences, which do not allow for proper interoperation. For example, it is possible to build a C++ DLL with Microsoft Visual C++ and not be able to use it within a Borland C++ project. With COM, however, the reusability transcends the source code and such differences in object representation are avoided.

COM has minimal language requirements to be able to work with component software units. For your language of choice to be able to *implement* a COM (*create* component objects) it must be able to support the creation of a table of function pointers. For your language of choice to be able to *access* a COM interface (and, therefore, work with component objects) it must support the ability to call functions by dereferencing pointers. Given these modest requirements in today's development environment, COM's proliferation is almost assured.

Transparent Remoting

How can clients communicate with component objects without concern for where those components physically reside? COM provides for full distribution. A component object is accessible regardless of where it resides on the network. More than a simple "internet-like" technology, which brings the component to the local machine to call it, COM actually runs the component remotely wherever it resides and only returns the result to the client. This allows for a true client/server distributed architecture.

COM components (also known as COM *Servers*) can be implemented in one of three ways:

- In-Process
- Local
- Remote

Clients who make use of these servers do not need to know which type of server they are calling. COM *provides transparent cross-process interoperability* by abstracting away and separating the calling transport mechanism from the component itself.

In-Process Server

In the case of an *in-process* server, which uses a traditional DLL, there is little work to be done. When the client loads the component, it is done so in the same address space as the client (the program). Therefore, the two components share the same memory space and file handles, and can communicate with each other with the highest efficiency.

Local Server

Implementation of a component that is packaged as an .EXE on the same physical machine as the client (program) is called a *local* server. Linking from the client to the component is a more complicated process than that of in-process servers. Calls to the server component (and their method parameters and return values) need to be packaged and transferred across the process boundary in both directions since the two components do not, in general, share memory (assuming 32-bit components). This transfer process is performed automatically and invisibly by COM using Lightweight Remote Procedure Call (LRPC) technology.

Remote Server

When the server is *remote* (resides on another physical machine somewhere on the network) conceptually the exact same transfer process needs to take place. The only difference is the transport protocol that is used. With local servers LRPC is used. With remote servers, true RPC across the network is used.

In all cases, neither the server nor the client need to worry about their proximity to one another. Neither server is written to assume a certain transport mechanism. All of this can be determined by COM at runtime based on various operating system settings, which means that clients use the same simple programming model when calling components regardless of where they are running. Therefore, a developer could build and test a component locally, and, once it is production ready, it can be moved out onto a remote machine on the network, and none of the test code needs to be changed.

Achieving cross-process interoperability is the key to solving the component software problem. It would be relatively easy to design a component software architecture that assumed all component interactions occurred within the same process space. The COM library encapsulates all the legwork associated with finding and launching components and with managing the communication between components. Because components are insulated from location differences, when a new component is released with support for cross-network interaction, existing component objects will be able to work in a distributed fashion without requiring any source code changes, recompilation, or redistribution of any kind.

OLE 2 Features

As mentioned before, OLE is a comprehensive technology that consists of several subtechnologies like:

- Linking and Embedding
- Controls
- Automation

Linking and Embedding

Linking and embedding objects into "containers" or "compound documents" is the basic foundation of OLE. These two methods store items, which were created by one application, inside a document of another application. The application that created the object is called the server application and the application that stores the object is called the container or client application.

Embedding

Embedding is the more common of the two methods. For example, creating a picture or a spreadsheet and placing it inside a Microsoft Word document. There are several ways of embedding an object into a container. An object can be pasted using the Paste or Paste Special menu items, dragged and dropped into the container, or inserted via the Insert Object menu item. Whichever way you choose, the end result is the same; the object created by a server, such as Paint or Excel, will appear to be part of the client's (Microsoft Word's) document.

An object has two states, passive and active. After the object is placed in the container, it is in the passive state. An object will stay in this state until it is necessary to make modifications to it, in which case it will become active by some user action defined by the client. Activating the object, to edit it, launches the application (for example, Microsoft Paint or Microsoft Excel) used to create the object.

The application can be launched in two ways, depending on how the client application has implemented activation support. The first way is for the client to start the server as a separate application in a separate window. The second way is for the client to become the server application for the duration of the object editing session. In this case, the client will change into the server. This change encompasses menus, toolbars, status bars, and any palette windows. Microsoft Word is a good example of this behavior, called "In-Place Activation" or "Visual Editing."

There are also two models for activating objects: *outside-in activation* and *inside-out activation*. Outside-in activation requires an explicit action to activate the object, such as choosing the Object Edit menu command or double-clicking the object itself. This model is the most common because it reduces the risk of inadvertently activating an object, which can take a significant amount of time to load and dismiss. Inside-out activation requires no extra user interaction; simply placing the cursor on the object will activate it. Thus, inside-out objects are indistinguishable from the client application's native data. This model can be used when the overhead of activating the object is small.

Linking

An object can also be linked to the client application. A linked object is a representation of (or pointer to) the actual object which resides elsewhere (either in the same document or in a different document).

An object can be linked by selecting the Paste Link option in the Paste Special dialog box. When you choose to paste link the object to the client, a picture of the contents of the clipboard is inserted into your document. The Paste Link option creates a link to the source file so that changes to the source file will be reflected in your document. The Insert Object dialog box also supports linking of objects. When you select the Link check box, along with the Create From File radio button in this dialog, a new object will be created that is linked to a selected file. A picture of the file contents will be inserted into your document. This picture will be linked to the file so that changes to the file will be seen in your document.

Editing a linked object is very similar to editing an embedded object. The only difference is that the data for a linked object remains in the document that created the object, whereas the data for an embedded object travels with the object to the client application. Additionally, editing a linked object is always done "Out-of-Place."

Controls and Control Containers

An OLE custom control (OCX) is a special kind of OLE 2 object. It is an embedded OLE 2 object with an extended interface that lets it behave like a Windows control. An OLE control container is an application that can support OLE controls. Thus, OLE custom controls are a set of extensions that turn simple OLE 2 containers and objects into more powerful *Control Containers* and *Controls*.

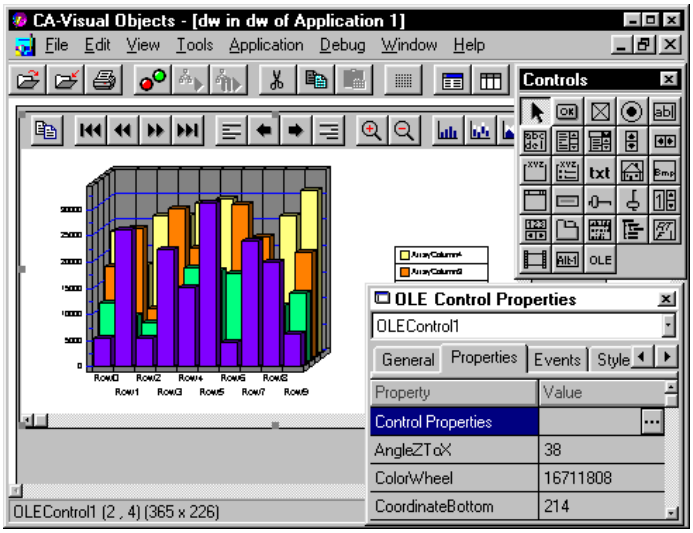
The OLE 2 standard for compound documents meets many of the requirements of both control and control containers, but not all of them. Writing a control involves some other issues, primarily those dealing with OLE automation. A control must expose its events, methods, and properties to a control container. And a control container must expose *ambient properties* and its own events to the control. Ambient properties are named characteristics or values of the container itself that generally apply to all controls in the container. Some examples of ambient properties are default colors, font, and whether the container is in *design mode* or *run mode*.

The difference between controls and simple OLE objects is that controls generally do not need a lot of user interface components, like toolbars and menus. However, they do have additional needs for event capture—such as focus and keystrokes.

OLE Support Inside Visual Objects Window Editor

To support OLE 2, the Visual Objects Window Editor has become both an OLE object and an OCX container. Thus, it will have design and runtime modes, dragging and dropping of objects and controls from the Window Editor palette, embedding and linking of objects, in-place activation and editing of embedded objects, and the ability to access all the methods, properties, and events of an OCX.

The following figure shows the Window Editor containing a business graphic OCX. Note that the OCX events and properties are listed inside the OLE Control Properties window.

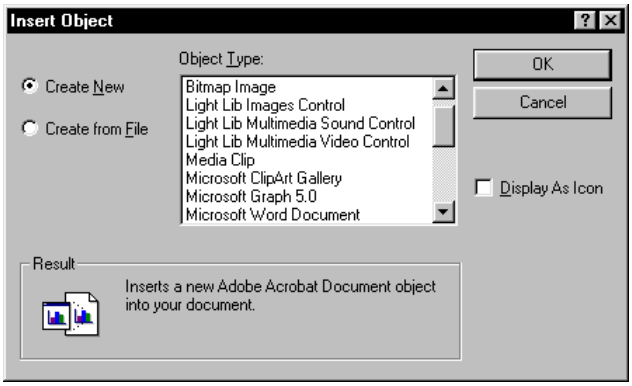


When the Window Editor is invoked, the Visual Objects IDE View menu extends to include "Test Mode...", thus giving the Window Editor two modes or states, design mode and run (or test) mode. OLE controls will not be *active* at design time (they will not have a window handle yet), but will appear only as a representation of contents. This is done in order to increase performance and memory by minimizing the number of open windows. When in test mode, however, the controls will be activated.

The Visual Objects IDE Edit menu now includes several OLE-related menu items, including Insert OLE Object, Insert OLE Control, and Setup OLE Control.

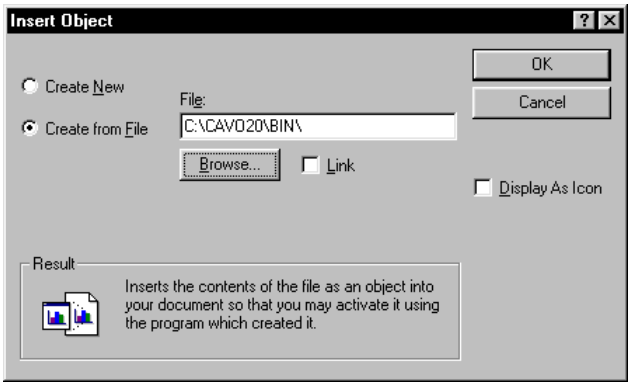
Insert OLE Object

Selecting the Insert OLE Object menu item launches the Insert Object standard dialog box:



The Insert Object dialog box is a common Windows dialog box. To create a new OLE object, click on the Create New radio button and select the new Object type from the Object Type list box. The Result group box describes the object that is selected.

If the OLE object to be inserted is from an existing file (for example, to use 256color.bmp file, which comes with Windows as an OLE object), the Create from File radio button should be selected. By clicking this radio button, the Insert Object dialog box removes the Object Type list box and replaces it with a File edit control for the path and object filename, as shown in the next figure:

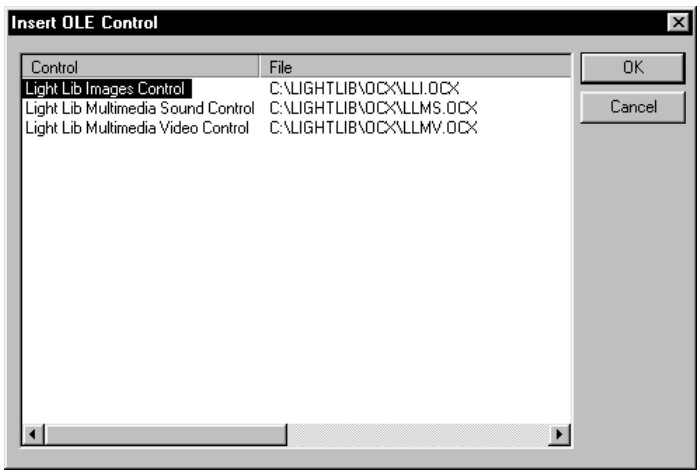


The Link check box, if selected, creates a linked object instead of an embedded object. This means that if there are changes to the object, in this case is 256Color.bmp, the linked object is updated in the window right away with the new changes. This is where the difference between linking and embedding can be seen clearly. An embedded object does not have a pointer to the actual file, but a linked object does.

Another option that can be selected in the Insert Object dialog box is Display as Icon. This check box is only available for Create New and Create from File modes. It is used to insert OLE objects as icons instead of the look and feel of actual objects. For example, instead of having the actual spreadsheet shown in your data window, the inserted spreadsheet object can be displayed as an icon. If you need to see the spreadsheet file, the icon can be double-clicked and your data window displays the spreadsheet file.

Insert OLE Control

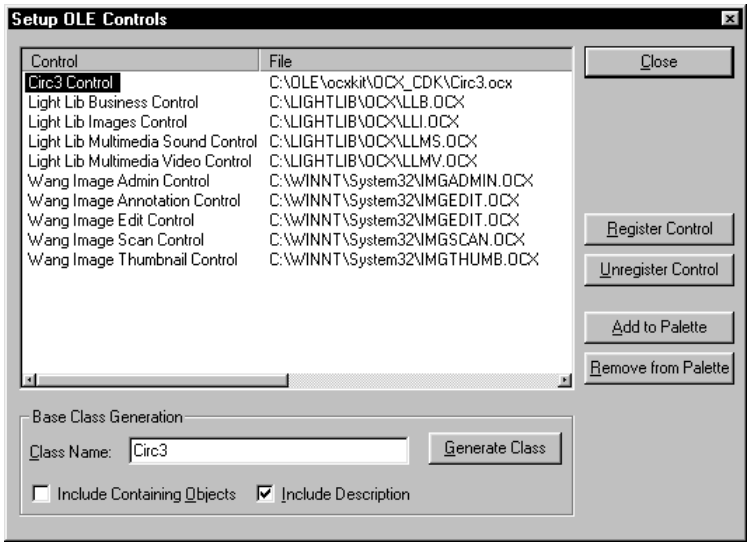
Selecting the Insert OLE Control menu item launches a dialog box with a list of all the OCXs that are registered in the database on your machine:



The only action required is to select the OLE control and press the OK button or double-click on the OLE control that is going be inserted into your window. These OLE controls are either automatically registered to the registry or can be registered manually, as described in the next section, in the Setup OLE Controls dialog box.

Setup OLE Control

The Setup OLE Control menu item launches a dialog box with all OCXs—which you can register or unregister, and generate the OLEControl class from them:



The Setup OLE Controls dialog box is similar to Insert OLE Controls, except it has more functionality. Several actions can be done in this dialog box at once:

- Registering OCXs
- Unregistering OCXs
- Adding registered OCXs to the tool palette
- Removing OCXs from tool palette
- Generating an "OLEControl" class

A new OCX would not be useful unless it is registered to the Windows Registry. Registering an OCX can be done in two ways-either through this Setup OLE Controls dialog box or by installation. The procedure to install OCXs is usually the same as the procedure to install any regular software, except OCX installation registers the OCXs automatically.

Similarly, you can unregister an OCX by pressing the UnRegister button or doing an uninstall, which usually comes with the installation package for the control.

To add registered controls to the tool palette in the Window Editor, press the Add to Palette button. This action also adds the controls to the Select From Palette command in the Edit menu.

And of course, any OCXs that are added to the Tool Palette and the Select From Palette command can also be removed by clicking on the Removing from Palette button.

OLE controls can also have all of their functions included in the source code by pressing the Generate push button. This action is similar to generating an OLE automation server, which will be discussed later in this chapter. Visual Objects will create a specific class for the OCX, which inherits from an OLEControl class. Note that you can program OCXs without having to generate a base class through the OLEControl class, but for performance and compile time checking reasons you may decide to generate a base class. Base classes for OCXs inherit from the OLEControl class.

Once the control is inserted into the Window Editor, clicking on the new control reveals its associated properties box. Since it is an OCX, the control will be queried for its properties. The control's methods and events are also queried and displayed in the standard Visual Objects properties box. Thus, clicking on a particular event from the properties box launches the Source Code Editor so that code can be written to handle the event.

An object browser, similar to the Visual Objects Class Browser, will also be available so that a particular control's properties, methods, and events can be seen without inserting the control onto the Window Editor. The object browser queries the registration database and the object itself for this information.

OLE Automation

OLE automation is very different from the original OLE features of linking and embedding. While the concept of compound documents is central to linking and embedding, OLE automation does not compound documents. But internally it uses the same techniques as linking and embedding for the communication between clients and servers. It allows one application to drive another application. Therefore, automation is also considered a part of OLE.

The basic idea behind automation is to define a standard for cross-application macro languages. Most standard applications, for example word processors or spreadsheets, incorporate some kind of programmability through a macro language. If this macro language is proprietary, it can only be used inside the application itself. It is not possible to control or program the application from another application.

Establishing a standard for accessing the macro language allows applications to start some other, maybe specialized, application and program it to perform a certain task. This is the basic idea behind OLE automation. Note that OLE automation only standardizes the protocol of the language, but not the language syntax. Different automation controllers might implement the same macro language using a completely different syntax. Actually, some OLE automation enabled applications might be programmable only through other applications. Visual Objects, of course, uses its own syntax.

When discussing OLE automation, the two most important terms are “client” and “server.” An automation server is an application that implements an automation interface. This automation interface enables other applications or automation clients, also called “controllers,” to program it. The automation interface is often referred to as IDispatch or dispatch interface. In order for this to work, the automation server must “expose “its interface to the automation client. The interface has an object-oriented structure consisting of functions, parameters, its return type, and also its properties, which might allow read/write or read only access. The benefit of exposing IDispatch is to provide functionality that is useful for other applications. For example, a word processor might expose its spell-checking functionality so that other programs can use it.

Automation servers are added to the registry, so any application can find out what automation servers are available by accessing the registry. Once a server has been found, the application can obtain the IDispatch through the OLE API or OLE2VIEW.EXE, which comes with the Win32 SDK.

This information of IDispatch becomes a component of the automation object. Here you will find functions being mapped to methods along with properties, and variables being mapped to access and assign methods.

Note: A server without IDispatch or type information cannot be used with Visual Objects. However, most automation servers provide the controller with type information.

Visual Objects and Automation

Since IDispatch has an object-oriented structure, representing OLE servers within a Visual Objects class seems natural. This can be done in two ways: either using a generic OLE automation object, or a specific object that is instantiated from a server-specific class. Visual Objects includes a tool that creates this specific automation object from the IDispatch of an automation server. This tool, which will be discussed later, is called *Automation Server Generator*.

Automation functions support a similar flexibility as late bound (untyped) Visual Objects methods—parameters might be polymorphic or optional. Additionally, automation supports named arguments, to pass parameters in any order. For these reasons, automation methods, properties, and variables can only be mapped to late bound (untyped) Visual Objects methods.

Runtime Automation Handling

To demonstrate OLE automation, the Microsoft Word automation server will be used as an example. A Microsoft Word Automation server IDispatch is called WordBasic. In order for WordBasic to be used as an automation server, we need to know its program ID or PROGID. This information can either be obtained from the documentation that comes with an automation server, or by directly looking into the registry with a tool like OLE2VIEW.EXE.

The first way of running WordBasic as an automation server from Visual Objects is to create an OLEAutoObject and pass the PROGID to the Init() method. The following lines of code instantiate a WordBasic automation server:

```
FUNCTION Start()
LOCAL oAuto AS OLEAutoObject
oAuto:=OLEAutoObject{ "word.basic" }
```

If you run this code, nothing will seem to happen. The reason is that WordBasic is started as an automation server and comes up invisible by default. This behavior depends on the server you are instantiating. Some servers may come up visible, others may come up invisible. Bringing up an automation server invisibly is better because the application can start an automation server and do several tasks without the user noticing.

In order to implement OLE automation, you need automation server documentation because the descriptions of those parameters are not available in IDispatch. In some automation server documentation, all the methods, properties, and variables of the server including argument names, types, and descriptions are listed. For WordBasic, take a look at the WordBasic.hlp which comes with Microsoft Word versions 6 and 7.

Using an automation server is not easy if method names and arguments are not available. This is when tools like OLE2VIEW.EXE or Visual Objects Automation Server Generator (which will be explained later in this chapter) are useful.

Compile-Time Automation Handling

So far, all the automation logic has been handled exclusively at runtime. The automation object was created through the generic OLEAutoObject class and the runtime system figures out the specifics of the automation object at the time you create it. This is similar to creating a DBServer on the fly instead of using the DB Server Editor at development time.

Using a generic OLEAutoObject is very flexible, but there are also disadvantages to it. At the time the object is instantiated, the runtime reads and processes all the type information for the server. Since OLE is resource intensive, reading the IDispatch is a time-consuming process which involves several interprocess calls (at least if you are not using an in-proc server that runs in your applications address space) for each method, access, assign, or variable. Interprocess calls are also called Local Remote Procedure Calls, or LRPCs. For Microsoft Word, which has 1000 functions, this process might take more than a minute.

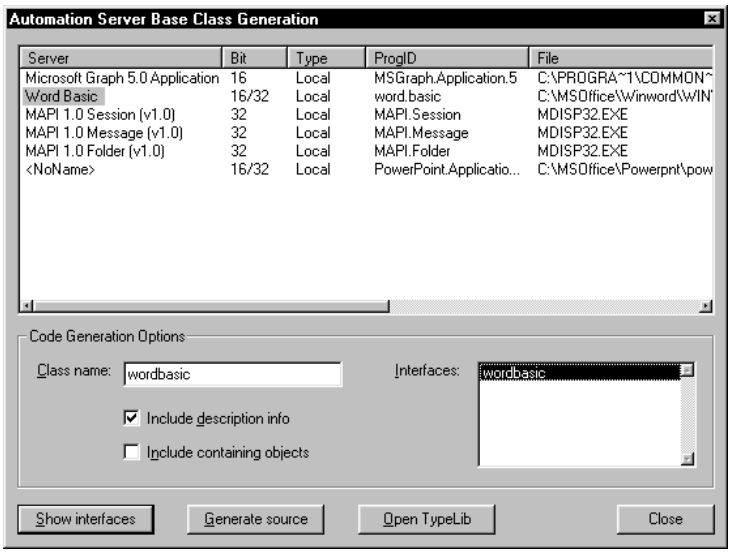
Another disadvantage is that compile-time error checking cannot be performed for automation code that uses OLEAutoObject. Since all the methods you call in the code do not exist at compile time, you will receive warnings about unknown methods. But you are not able to catch misspelled method names.

The alternative to using OLEAutoObject is to use a pregenerated class that inherits from OLEAutoObject. This pregenerated class is specifically generated for each automation server you want to use. Visual Objects provides the tool, Automation Server Generator, to pregenerate the specific class. It is a simple concept similar to using the DB Server Editor or creating a specific class, that inherits from the DBServer class.

Automation Server Generator

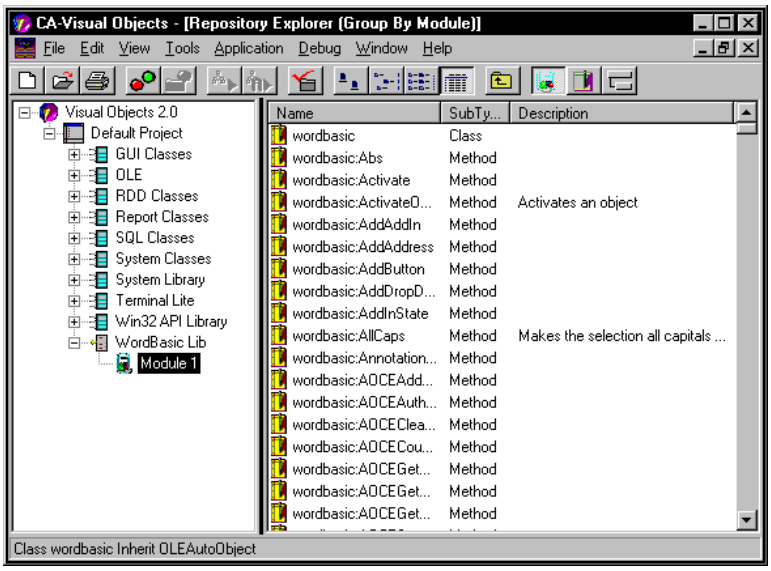
To use Automation Server Generator the following steps must be performed:

1. The Application must have GUI classes and OLE libraries included in the application properties.
2. Focus must be on the module where the automation object will get created.
3. Select the Automation Server command from the Tools menu.
4. The Automation Server Base Class Generation dialog box appears:



5. Select the server name from the server list and click the Show Interfaces button or simply double-click the server name to show the selected automation server's interface(s).
6. Select the interface to be generated by the automation server code generator. One interface can be generated at a time.
7. Type the class name to be generated. By default, Visual Objects puts the interface name as the class name.
8. Click the Generate Source button.
9. Close the Automation Server Base Class Generation dialog box by clicking the Close button after the code generation process is complete.

In the next figure we will show a module that contains an automation server object, its methods and properties that are created from the WordBasic interface. Since the class name to be created is WordBasic, Visual Objects creates a WordBasic class, which inherits from OLEAutoObject class. The Automation Server Generator supports the OLE-provided help document string. This string might be part of the type information for a method, giving a brief explanation of what the method does. If the description for the interface is available, it will be created along with each entity, as long as Include Description Info check box is selected in the Automation Server Generator dialog box.



Using a Pre-Generated Automation Class

As long as only one interface class is involved, things are pretty simple. Instead of using OLEAutoObject, we simply use the generated class and, since the class knows its server, we do not have to pass any arguments on the instantiation. After the WordBasic class is generated, we can bring up the WordBasic server with the following code:

```
FUNCTION Start()
LOCAL oWordBasicApp AS WordBasic
      oWordBasicApp:=WordBasic{}
```

Notice that in the instantiation of the WordBasic class, nothing is passed as parameters. This is because in the Init() method of the WordBasic class, as shown in the following lines of code, are generated from the Automation Server Generator.

```
METHOD Init(ObjID) CLASS WordBasic
      IF(ObjID=NIL)
            ObjID:= "word.basic"
      ENDIF
      Super:Init(ObjID,0,.T.)
```

An automation server may have several interfaces. Combining several interface classes makes the process complex. There is no way for the Automation Server Generator to find out which specific interface is returned by a method. Although OLE provides the information that a dispatch interface returns, it does not tell which one. This is not a Visual Objects limitation, but a general OLE problem. The generated code creates another OLEAutoObject for a returned dispatch interface. The generated code is functional, but since it uses the generated OLEAutoObject class, it sub-optimal. In the next section, we will explain how to get around this problem.

Changing Generated Automation Server Code

If you are dealing with a dispatch interface hierarchy, the generated code for methods that return IDispatch may be changed manually. This involves changing a class name from OLEAutoObject to the name of the class you generated for the specific dispatch interface.

Within a generated access, you will find the following lines of code close to the end. This line is the same for all methods, accesses, and assigns returning a dispatch interface. Unfortunately, the WordBasic automation server only has one IDispatch.

```
// To use a pre-defined class here, change
// OLEAutoObject to desired class name
uRetValue := If(uRetValue:pInterface!=NULL_PTR,;    OLEAutoObject{uRetValue},;
     NULL_OBJECT)
```

To return a specific automation object, you have to change OLEAutoObject in the IF() statement into the desired class name. Since this is not applicable to WordBasic, we use a make-believe DICAVO automation server which is a subclass of OLEAutoObject. The changes would look like the following lines of code:

```
uRetValue := If(uRetValue:pInterface!= NULL_PTR,;
     DICAVO{uRetValue},;
     NULL_OBJECT)
```

Similarly, you have to change OLEAutoObject to the generated automation server class name (DICAVO) in each method to type the whole interface hierarchy. There are still more changes that need to be done, and they should be changed in the Start function as follows:

```
FUNCTION Start
     LOCAL oDICAVO AS DICAVO
     LOCAL oInterface2 AS DICAVOInterface2
     LOCAL oFunc1 AS DICAVOInterface2
     oDICAVO:=DICAVO{}
     oInterface2:=DICAVO:Interface2
     oFunc1:=oInterface2:Func1()
     oFunc1:Visible:=True
     oFunc1:AddText( "New Text" )
     oDICAVO:Quit()
```

The only change to the pregenerated automation server code is to change OLEAutoObject to the new generated class.

Advantages of Using a Pre-Generated Automation Class

There are several advantages of using a pre-generated automation class.

- Enhanced compile-time checking for nonexistent or misspelled servers or methods. Such errors can be avoided since compiler warning would be shown.
- Instantiating an automation server object is much faster using a generated class since all the type information is already available. If OLE automation is done at run time, the application must pull the information together each time an OLEAutoObject is created.
- Method invocations are faster than that of a generic class. Your application does not have to look up the dispatch ID any time you do a method invocation.
- OLEAutoObject contains a huge array to hold all the type information of the server. This array does not get created if a generated automation server is used. However, this does not save memory since the generated automation server code adds to your .EXE and also requires storage space.

Tip: The WordBasic example contains one thousand methods. If only a small amount of methods are used, then it makes sense to generate the WordBasic class in a separate application and copy the class definition, the Init() method, and the methods and properties that are going to be used in the application. Visual Objects links all classes and methods found in your application and its search path into the executable.

OLE Automation Collections

Unfortunately, WordBasic does not implement one special feature of OLE automation—collections. Collections are comparable to arrays that are managed through access and assigns. Collection properties do not return or accept a complete array, but they manage the array internally and only return or accept individual elements. This means you have to be able to pass array indices to the access and assign methods encapsulating these properties.

The following code calls the access MyAccess of the object MyObject, expecting it to return an array and use the dimension operator on the returned array, which is not how OLE collections work:

```
? MyObject:MyAccess[1]
```

Therefore, a new syntax for calling accesses and assigns of OLE collections has been created:

```
? MyObject:[MyAccess, 1]
```

This calls the access MyAccess of MyObject and passes 1 as a parameter. This means you are also able to declare access methods with a parameter list as follows:

```
ACCESS MyAccess(x) CLASS MyClass
```

Assigns work the same way:

```
MyObject:[MyAssign, 1] := 87
```

and

```
ASSIGN MyAssign(x, y) CLASS MyClass
```

In this case, MyAssign is called with the value 87 for x and the value 1 for y. This new syntax allows proper handling of OLE collections in the Visual Objects language.

Note: This behavior is only implemented for late-bound (untyped) access and assigns.

Named Arguments

Another feature of OLE automation is the use of named arguments. Automation methods might support named arguments for optional parameters. In this case, any parameter that has a name associated with it, by specifying the name with the argument, allows the passing of arguments in any order, along with omitting arguments completely.

The Visual Objects implementation of OLE automation and named arguments is supported through the NamedArg class. The Init() method of this class takes two parameters; the first one is the argument name (as a symbol) and the second one is the actual value:

```
oAuto:FormatFont(NamedArg{#Points, 18})
```

The above code shows a typical use of named arguments. The FormatFont() method of the WordBasic automation server contains many parameters, as in the following line of code;

```
METHOD FormatFont(Points, Underline, Color,;
     Strikethrough, Superscript, Subscript,;
     HIDDEN_, SmallCaps, AllCaps, Spacing,;
     Position, Kerning, KerningMin, Default,;
     Tab, Font, Bold, Italic) CLASS WordBasic
```

By using named arguments, you only have to specify the property you actually want to change; in this case, the font size to 18. You do not need to pass the rest of the parameters to the FormatFont() method. For example, to change the font to Bold, no commas are necessary before the Bold argument name. You only need:

```
oAuto:FormatFont(NamedArg{#Bold,1})
```

The following lines of code show an example of using the WordBasic OLE automation server, from instantiating to closing the server:

```
FUNCTION Start
     LOCAL o AS WordBasic
     o:=WordBasic{}
     o:FileNewDefault()
     o:ViewZoom100()
     o:Insert("Visual Objects 2.7 is the ")
     o:FormatFont(NamedArg{#Points,24})
     o:Insert("Best ")
     o:FormatFont(NamedArg{#Points,12})
     o:Insert("Application Development Tool")
     o:FileSaveAs("c:¥samples¥VOWord.doc")
     Textbox{,"Visual Objects Automation;
          Sample","Click OK after you are
          done"}:Show()
     o:FileExit()
```

Note: The WordBasic server must be running before your application is executed if you want the user to see the process. The end result is the same and can be viewed by opening the document file, VOWord.doc, in the path specified. The previous example code above instantiates an OLEAutoObject as WordBasic, creates a new document with the Normal.dot template, changes the View to 100%, inserts some text, changes the font to different sizes, saves the document to C:\SAMPLES\VOWord.doc, and closes the server.

OLE Automation and OCXs

OLE Controls (OCXs) are custom controls that use OLE mechanisms to communicate with their owning windows. The 32-bit OCX market is moving rapidly. It compels the creation of sophisticated OCXs for almost every need. To name a few: image processing, business graphics, spreadsheets, word processors, and Word Wide Web browsers.

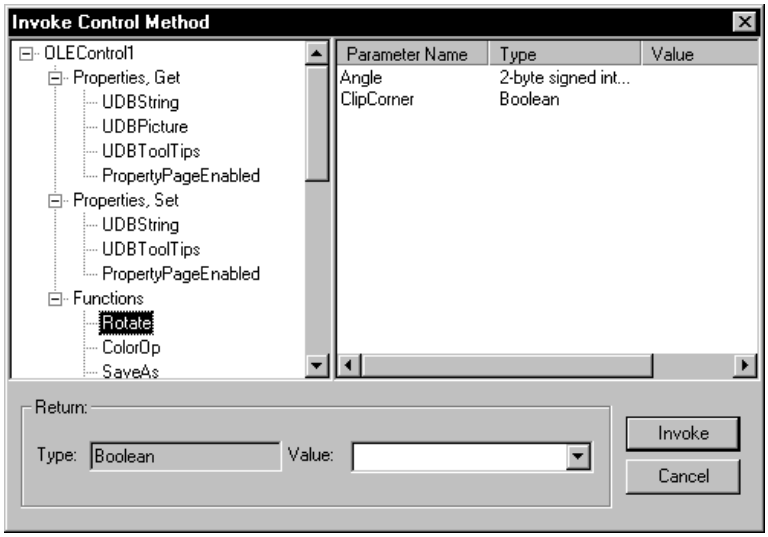
Visual Objects fully supports the use of OCXs. OCXs are often a better choice than full-blown OLE server applications because they are usually less resource intensive and run inside the application's address space, which gets rid of the overhead involved in LRPCs between address spaces.

OCXs are not dumb controls that are simply placed on a form and send some Windows messages to their parents. They are intelligent controls that can be programmed and can fire any kind of events to their owners. As you have probably guessed, programming OCXs happens through OLE automation. Just as with normal automation servers, Visual Objects supports runtime and compile-time OCX handling. Let's take a look at the runtime-based handling first.

Runtime-Based Automation Handling

OCXs are available as controls in the Window Editor. By choosing the Insert OLE Control command from the Edit menu in the Window Editor, you can select an OCX. From inside the window, the OCX is treated like a regular control and, therefore, an object is created in the Init() method of the form. Before we look at the generated code and see how we can talk to the OCX, we can first take a look at the functions and properties that make up an OCX.

After having selected an OCX on the form, choose the OLE Control Methods command from the Edit menu. This brings up the Invoke Control Method dialog box that shows all properties and functions with their parameters for the selected OCX. You can even enter parameters and invoke these methods through this dialog. This way, you can set up the OCX for the initial appearance at runtime and see what certain methods do.



All the methods and properties listed in this dialog can be directly sent to the object representing the OCX control. The class for a simple data window containing a Light Lib OCX is shown below:

```
CLASS OCXWin INHERIT DataWindow
      PROTECT oDCLLibOCX AS OLECONTROL
```

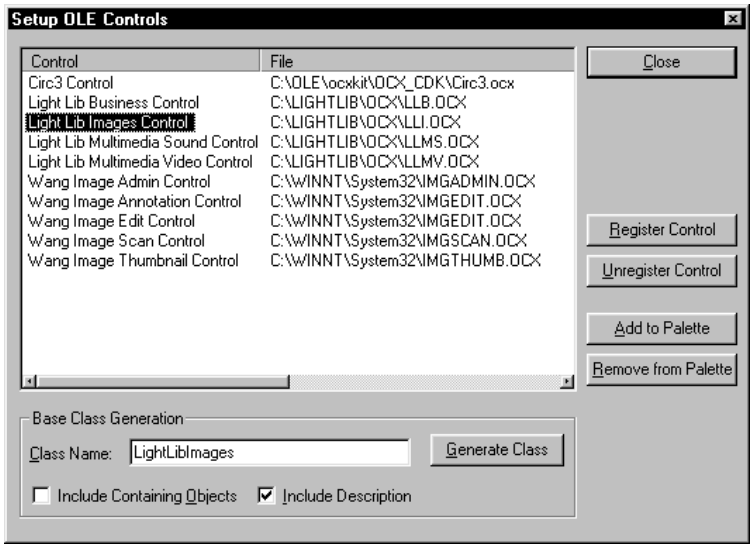
The object representing the OLE object is inherited from OLEControl. Anywhere inside a method of class OCXWin, you can now call methods or set/query properties of the OCX. Adding this code at the end of the generated Init() method, makes the OCX load the specified image any time the form is invoked:

```
oDCLLibOCX :Load( "cai.bmp",0)
```

Usually, the OCX would not load the same image at startup, but the OCX would be prepared in the Window Editor in such a way that the image is already loaded when the control appears.

Compile-Time Automation Handling

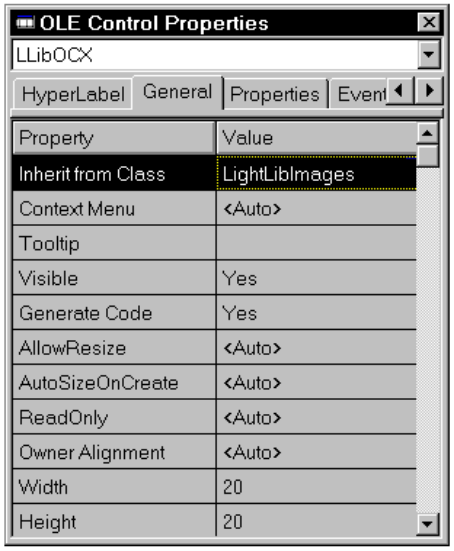
The approach we have taken so far simply sends messages to the control object that was inherited from OLEControl. This is completely runtime-based. To provide compile-time handling, we have to generate a class again. This is done through the Setup OLE Controls dialog box, which is accessible via the Setup OLE Control command from the Tools menu:



To generate a LightLibImages class from LightLibImages Control:

1. Select the OCX (for example, LightLibImages Control).
2. In the Class Name field, enter the name of the new class that you want Visual Objects to generate. By default Visual Objects uses the control name as the class name, such as LightLibImages.
3. Enable the Include Description check box if you would like Visual Objects to generate the descriptions.
4. Click the Generate button.

5. In the Window Editor, after the LightLibImages class is generated, change the Inherit from Class property in the OLE Control Properties window to **LightLibImages**, as shown here:



This causes the Window Editor to generate a class definition for the form to use the generated automation class for the OLEControl as follows:

```
CLASS OCXWin INHERIT DataWindow
    PROTECT oDCLLibOCX AS LightLibImages
```

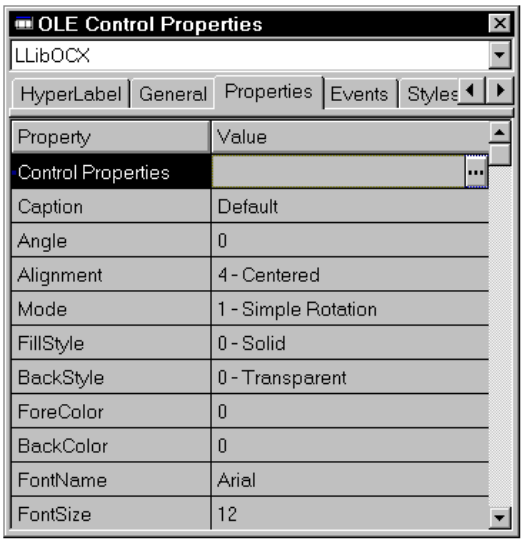
The Init() method of this window class would look like this:

```
METHOD Init(oWindow,iCtlId,oServer) CLASS OCXWin
    ...
    oDCLLibOCX := LightLibImages{SELF,;
        ResourceID{DW_OLECONTROL1,_GetInst()}}
    oDCLLibOCX:Hyperlabel := Hyperlabel{#LLibOCX,;
        NULL_STRING,NULL_STRING,NULL_STRING}
    oDCLLibOCX:CreateFromAppDocStorage(;
        OleAppDocStorage{"C:¥…¥Application 1.MDF",;
        "OCXWin", "LLibOCX"})
```

Events

As mentioned before, OCXs can also fire custom events. This is not really part of automation, but the OCX event handling is available to Visual Objects windows.

OCXs do not necessarily have events; the Light Lib Image OCX (the current version) does not have events. If an OCX supports these events, they do not become part of the OCX object; however, they do become methods of the window that own the OCX. You can generate OCX event handling by selecting the event and clicking the ellipsis (...) button for that particular event in the OLE Control Properties window, as shown below:



This brings up the Source Code Editor with the method declaration, including the parameters that the OCX passes when firing the event. Once again, parameters can be obtained from the OLE type information for the OCX, from either the documentation or the OLE2VIEW.EXE.

Putting OLE to Work

OLE is much more than just a Windows add-on. Additionally, on the programming side, the complexity of OLE is mirrored in the OLE programming interface. OLE objects, servers, and containers communicate by means of interfaces, each of which provide a different set of functionality. Interfaces are arranged in a hierarchical inheritance tree, starting with a root interface called IUnknown. Given a pointer to the IUnknown method, IUnknown::QueryInterface can be used to ask for additional interfaces supported by the OLE object.

The minimum set of interfaces to be implemented for an OLE object or container is quite well defined and varies with the range of functionality you want to support. To illustrate this, a list of the standard interfaces for use with the Visual Objects Window Editor follows. These can be used to create an OCX container:

```
IUnknown
IDispatch
IOleInPlaceUIWindow
IDropTarget
IOleUILinkContainer
IOleInPlaceFrame
IAdviseSink2
IPropertyNotifySink
IOleClientSite
IOleInPlaceSite
IOleControlSite
...
```

Each interface provides a set of standard methods, all of which have to be implemented, even though in many cases they are just dummies returning a NOT_IMPLEMENTED constant.

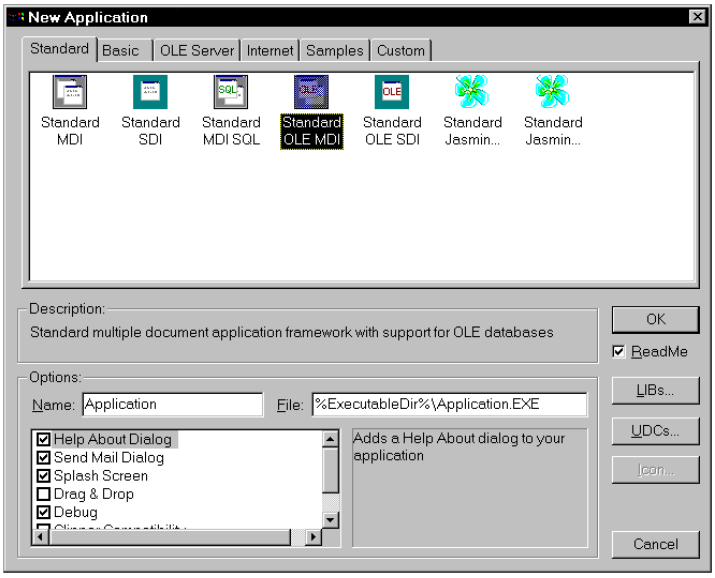
To make things even more challenging OLE works with it's own file format, which are called DocFiles (.DFL files), and memory management, and uses Unicode strings (in 32-bit) throughout, regardless of whether it is running on Windows or Windows NT.

It should be evident that the complexity of OLE calls for some kind of support layer between your application and the OLE basics, especially because most of the infrastructure you are required to provide is always identical and application-independent. Therefore, Visual Objects encapsulates OLE to hide the necessary but always identical basics, such as the needed set of interfaces from the user, and instead exposes a lean and clear interface to OLE.

In the following section, you will see how to use Visual Objects to implement a multiple document interface (MDI) OLE container application, which can manage embedded or linked OLE objects and supports "In-Place" activation. Implementing the container using the Visual Objects OLE classes, requires much less work than doing the same low-level implementation in C or C++.

The Sample Frame Work

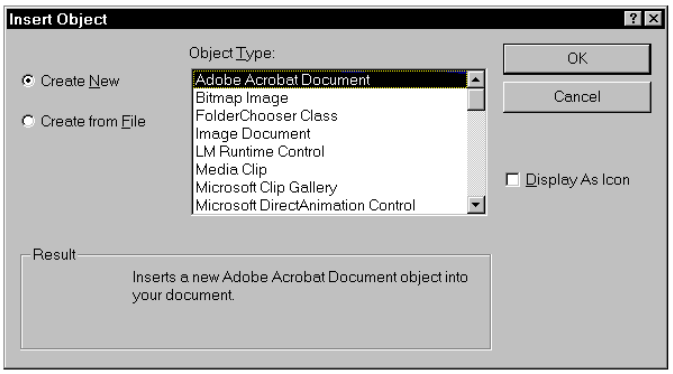
Start by simply select the Standard OLE MDI application on the Standard tab page:



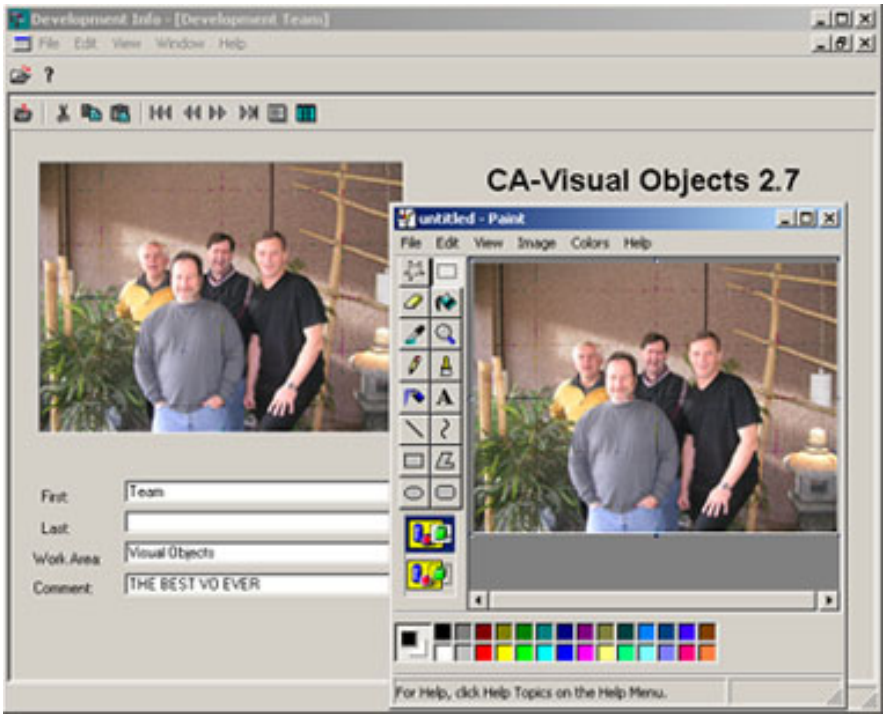
This creates the Standard OLE Menus module, which contains the EmptyShellMenu and StandardShellMenu entities. StandardShellMenu contains OLE menu items such as Insert Object, Paste Special, and Links.

Inserting Objects

The Insert Object menu item calls the InsertObject() method of the DataWindow, which in turn calls OLEObject:CreateFromInsertDialog(). This brings up the Insert Object dialog box:



After successfully creating the object, it is positioned on the form and receives the input focus. By default, out-of-place activation will take place:



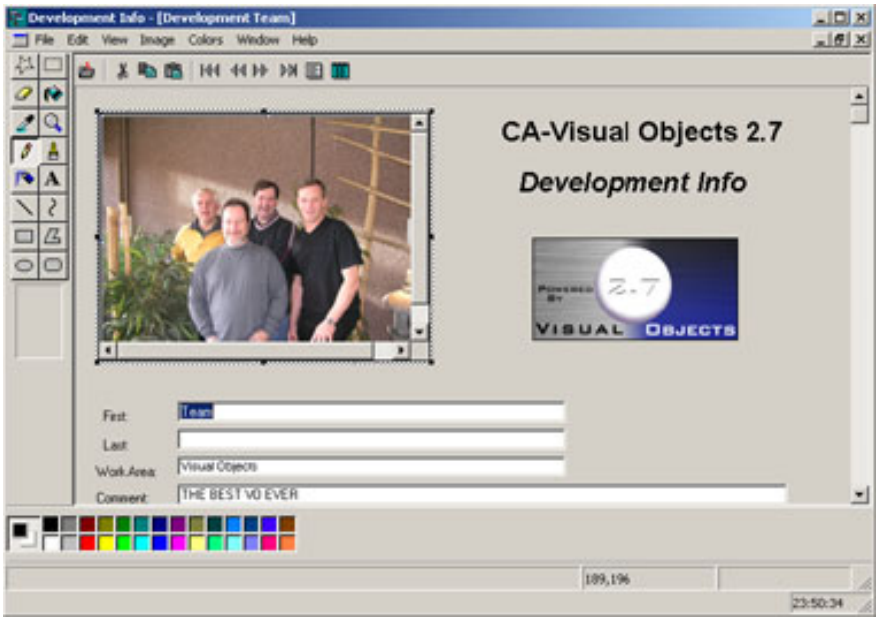
Notice that Microsoft Paint runs in a separate window, and that the OLE object inside the Visual Objects container is grayed out as long as the object is being edited. To close Microsoft Paint during the out-of-place activation, click anywhere on the Visual Objects form.

To demonstrate in-place activation, you will need to modify the StdDataWindow:Init() method as follows:

```
METHOD Init(oParentWindow, sFileName, lReadOnly, oServer) CLASS StdDataWindow
     LOCAL sCaption AS STRING
     LOCAL aKids AS ARRAY
     LOCAL i AS WORD

     SUPER:Init(oParentWindow)

     SELF:Menu := StandardShellWindow{SELF}
     SELF:ToolBar:PressItem(IDM_StandardShellMenu_View_Form_ID)
     sCaption := "Browse Database:"

     IF !IsNil(oServer)
          SELF:Use(oServer)
          SELF:Caption := sCaption + oServer:Name
     ELSE
          SELF:Use(CreateInstance(#DBServer, sFileName, , lReadOnly))
          SELF:Caption := sCaption + sFileName
     ENDIF

     //Retrieve all child objects of the StdDataWindow, including the window's
     //control objects
     aKids := SELF:GetAllChildren()

     FOR i := 1 UPTO ALen(aKids)
          IF IsInstanceOf(aKids[i], #OLEObject)    //If the child is an OLEObject
```

```
                aKids[i]:AllowInPlace := TRUE
                aKids[i]:AllowResize  := TRUE
        ENDIF
    NEXT
```

Note that the *AllowInPlace* and *AllowResize* flags are set to TRUE, meaning that the object is allowed to be edited in-place and the user is allowed to resize the object. Compile this new code and then run the application. Double-clicking on the object itself will demonstrate the in-place editing of the OLE object:



Note: The server can deny in-place activation even if OLEObject:AllowInPlace is set to TRUE. In such cases, out-of-place activation will take place.

You can see that the server's menus are merged with the original Visual Objects container menu, and that the server's toolbars are integrated into the MDI client area, allowing you to work with MS Paint (or any other OLE server) from within your Visual Objects application. From the time when the user double-clicks on the object to the final in-place activation, many activities go on behind the scenes, all of which you would have to code explicitly if you were doing OLE the hard (low-level) way. The server and the container have to communicate and decide how to merge menus, how to use the container's client space, and what's going to happen to the container's and server's toolbars.

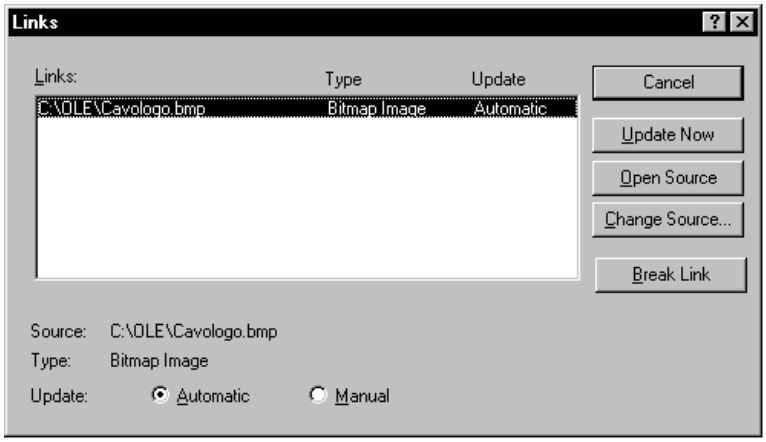
In Visual Objects, all of this is automatically done for you. You do not have to implement 10+ interface classes and you do not have to worry about the OLE basics—all the "magic" is automatically done behind the scenes.

To close, or deactivate, the server (in this case, MS Paint), simply click on the gray area of the form.

Adding Paste and Link Support

Another way of getting an OLE object inside the OLE container is via OLE's Paste Special dialog box.

Both the Insert Object and Paste Special dialog boxes contain options to create links to OLE objects instead of embedding the objects. Visual Objects provides another dialog box, Links, which allows you to manage these links. You can update, edit, change, and break the link to the OLE server in this dialog, as shown below:



You can bring up the Links dialog box by simply calling the DataWindow:Links() method at runtime. This is already being done via the Edit Links… menu item. At design time, this dialog can be displayed by having a linked object in your window, and selecting the Links command from the Edit menu.

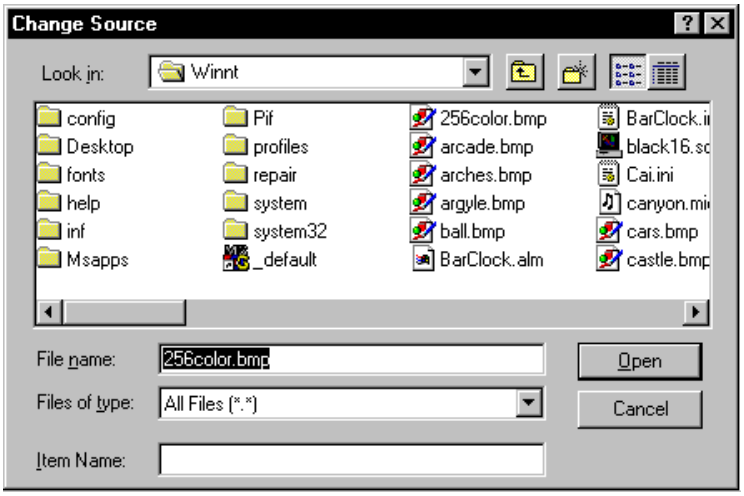
The Update radio buttons are available for you to specify the type of links for the selected linked object. Select Automatic update if you want to have the changes of the linked object to be reflected in your linked object automatically. For example, if there are changes to the CAVOLOGO.BMP file, the changes are updated automatically to the linked object in your window.

To the update data for the selected linked object manually, you select the Manual radio button. The Update Now button is pressed to update the changes in the selected linked object. For example, if there are changes to the CAVOLOGO.BMP file, which is used in your window, you might want to press this button to have that changes take effect in your window.

Tip: You do not need to do this if the Update type is Automatic.

The Open Source button is pressed if you need to make changes (editing) to the linked object. This object will be opened in the application the linked object was created. For example, the CAVOLOGO.BMP file will be opened in Microsoft Paint, an out-of-place activation will take place.

The Change Source button is pressed if you want to specify a different object for the selected linked object. This invokes the Change Source dialog box:



In this dialog box, you can specify the location of the new linked object.

The Break Link button is pressed if you want to disconnect the link between the linked object in your window with the file. The linked object will then be an embedded object, and changes to the file (in our case CAVOLOGO.BMP) cannot be updated in your window anymore. Once the link is disconnected, it cannot be reconnected.

Inserting Objects Using Drag-and-Drop

OLE containers usually allow the user to place new OLE objects inside the container by using drag-and-drop techniques. To achieve this in Visual Objects, you call EnableOleDropTarget(TRUE) for the ChildAppWindow you want to register as an OLE drop target. For the Standard OLE MDI application, this could be done in the StdDataWindow:Init() method by adding SELF:EnableOLEDropTarget(TRUE). After that, any OLE-related drag-and-drop operation generates an OLEDragEvent object and one of the container's methods, OleDragEnter(), OleDragOver(), or OleDragDrop(), is called.

Given the example of an OLE container, which should only handle WordPad documents, you would have to examine the value of OleDragEvent:ObjectName as follows:

```
METHOD OleDragOver(oDE) CLASS StdDataWindow
     RETURN oDE:ObjectName = "WordPad Document"
```

By returning FALSE from OleDragOver(), the drag-and-drop cursor changes to the no-drop shape, indicating to the user that the current object cannot be dropped. When the user releases the mouse button, and thus finishes the drag-and-drop operation, OleDrop() is called:

```
METHOD OleDrop(oOleDragEvent) CLASS StdDataWindow
     LOCAL oOLE AS OLEObject
     IF (oOleDragEvent:ObjectName ="WordPadDocument")
          oOLE := OLEObject{self}
          oOLE:AllowInPlace := TRUE
          oOLE:AllowResize := TRUE
          IF oOLE:CreateFromOLEDragEvent(oOLEDragEvent)
               oOLE:Origin := oOLEDragEvent:Position
               oOLE:Show()
               oOLE:SetFocus()
               RETURN TRUE
          ENDIF
     ENDIF
RETURN FALSE
```

First, we check the object's type, because we only want to allow WordPad objects. Then, the object creation is done similarly to how InsertObject and PasteSpecial are handled internally, except that we now use CreateFromOLEDragEvent for creating the object. Additionally, we get the object's position from the event by initializing oOLE:Origin with OLEDragEvent:Position.

Showing Status Bar Messages

When an OLE object is activated in-place, the container and the server menus are merged. But the container's status bar (if any) is still controlled by the container. In order to allow the activated server to display its status bar messages (for example, to describe its menu items) in the container's status bar, we use EnableOleStatusMessages—a method of ShellWindow, available as soon as you include the OLE library in your search path. Once this is done, the method OnOleStatusMessage() is called whenever the activated server wants to display a status bar message. By default, OnOleStatusMessage() takes the message string parameter and displays it in the ShellWindow's status bar. If this is your intended behavior, you do not have to do anything. If you want to do something different, overload OnOleStatusMessage() in your derived class and you can do anything you want with the message string.

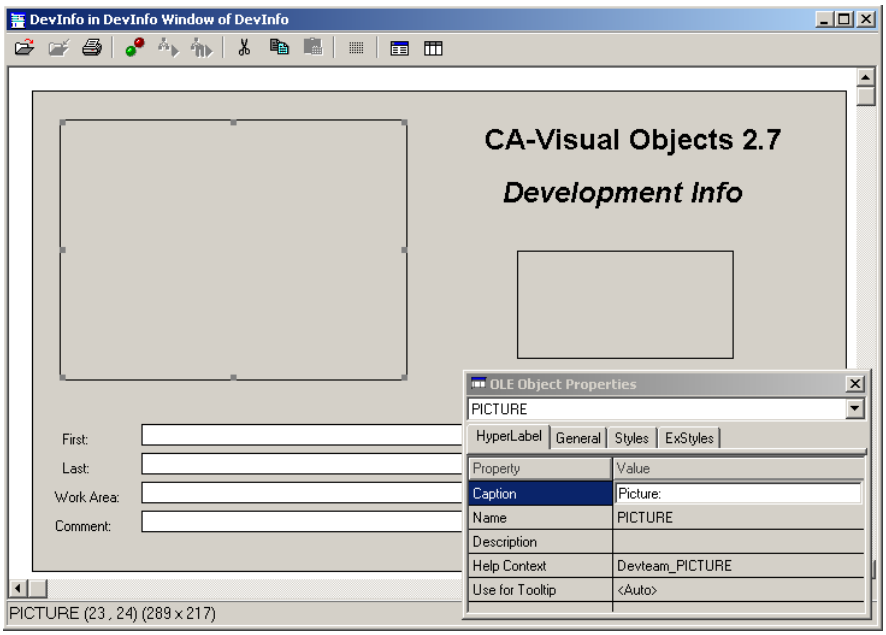
Using OLE in Databases

Since OLE is a big part of modern applications, databases can now contain OLE fields. You can insert objects or OLE controls into the OLE field. Each field in a database can have a different OLE object and it can also be changed at runtime.

An OLE field is similar to a memo field. It has a pointer to another file, called a "DocFile", which has a DFL extension. DFVIEW.EXE, which comes with MSDN, can be used to view the .DFL files.

To create an OLE database, you can use the Visual Objects DB Server Editor. The process of creating a new OLE database is almost the same as that of a regular database. The only difference is that the OLE field has to be an OLE data type. After the server is created, Auto Layout can be performed in the Window Editor for that new server.

Visual Objects Window Editor's Auto Layout feature creates an OLEObject control for each OLE field. At design time, this control looks like a MultiLineEdit control, and it would have all the properties of OLEObject in the OLE Object Properties window. The next figure is an example of placing an OLE field inside of a data window. This example is taken from the DEVINFO.AEF sample application in the \SAMPLES\OLE\DEVINFO subdirectory. In this example, the PICTURE field is the OLE field:



At runtime, there are several things that can be done to the OLEObject field, when the focus is on the OLEObject field, including:

- Inserting a new OLE object

- Inserting a linked object
- Changing the OLE object

As you can see, you can skip to the next record, to the previous record, or to the beginning and end of file just like regular databases. And for each record you can place a different picture or any other OLE objects.

The procedure to insert an OLE object and create links in the OLEObject field is the same as that of inserting and creating links of OLE objects in the Visual Objects Window Editor.

Chapter 6

Justifying Database Access Choices

In deciding how to build new database applications and how to make the most of existing code and data, there are several options. Different circumstances require different approaches. A thorough understanding of the technical tradeoffs will aid in producing the optimal solution. This is the essence of engineering—optimizing under constraints.

From a language standpoint, Visual Objects fully supports both the procedural and object-oriented programming approaches. The procedural approach, the traditional structure of Xbase applications, is not well suited for dealing with the complex demands of GUI environments such as Windows. The object-oriented approach provides all the capabilities of the procedural approach, but also adds structures that fit the event-driven and multi-tasking nature of GUI applications. In practice, a hybrid approach is often the best: an object-oriented structure overall to handle user interfacing and navigation, and procedural programming in the action routines that perform the business tasks.

Another choice is the type of database access to support in your application. Visual Objects supports Xbase-style data manipulation through a replaceable database driver (RDD) technology, allowing you a choice of different file formats driven by a common language interface. This feature allows you to use different file formats within the same application and to tailor your applications so that migrating from one format to another is simple and straightforward.

In addition to Xbase database operations based on the RDD technology, Visual Objects supports SQL databases with Open Database Connectivity (ODBC), the standard replaceable driver technology for SQL under Windows. Although SQL and Xbase databases use different logic, the SQLSelect class provides an interface compliant with the DBServer class for DBF, allowing an application to operate the same way regardless of which database it uses.

The different options presented here—procedural vs. object-oriented programming, different types of Xbase-style databases or SQL—present very similar programming styles, but they do not present similar performance characteristics. The reason several options are provided is that each offers its own strengths and weaknesses. Care should be taken in choosing the appropriate technology for each application.

Visual Objects makes these choices easy by requiring minimal changes in your programs. This chapter outlines the issues involved in choosing a programming method and a database format and will, hopefully, help you make the right decisions for your own situation.

Technology—Object-Oriented or Procedural

Consider the following program, written using standard, procedural, Xbase commands:

```
USE employee NEW
SET INDEX TO empno
...
DO WHILE .NOT. EOF()
     IF Sex == "M"
          DELETE
     ELSE
          Salary += Raise(Name)
     ENDIF
     SKIP
ENDDO
```

This example is trivial, but it illustrates the major operations commonly done with Xbase databases:

- Opening a table in a work area
- Using an index file for ordering
- Checking for end-of-file (in the logical sense, if there are any more records available under the controlling order of this index)
- Using the value of a field (Sex) on each record
- Manipulating records with operations such as DELETE
- Changing the value of a field (Salary)
- Moving the record pointer

Aliased References

Experienced programmers know that this program is not very robust because all the database references and commands refer to the current work area. Imagine that the function Raise() was originally written to prompt the user but was later modified to look up the new value in a table. If Raise() neglects to select the original work area before returning, the SKIP statement after the invocation of Raise()—and all subsequent statements—would apply to the wrong table!

It could be argued that this is bad programming—Raise() should clean up after itself. Defensive programming means not relying on others to clean up but ensuring that your own program is robust enough to survive whatever happens. In any case, as the world gets more complex, with multiple windows and multiple tables open at the same time, it is increasingly unrealistic to expect that every piece of code will be completely free of side effects.

This is especially important under the event-driven paradigm of GUI programming. Between the USE statement and the loop (where the ellipsis represents other code), a GUI program is likely to grant control to the user who might do any other operation, including opening another table in another window.

A good solution to this problem would be to qualify every reference using the alias operator, as in Employee->Salary. However, Xbase commands like SKIP and DELETE cannot be qualified this way. Visual Objects, therefore, supports function equivalents for the database commands (for example, SKIP corresponds to DBSkip() and DELETE to DBDelete()). Fully qualifying both field references and function calls, produces code which is much more robust.

For example:

```
USE employee NEW
SET INDEX TO empno
...
DO WHILE .NOT. Employee->(EOF())
     IF Employee->Sex == "M"
          Employee->(DBDelete())
     ELSE
          Employee->Salary += Raise(Employee->Name)
     ENDIF
     Employee->(DBSkip())
ENDDO
```

This version of the program exploits the universal aliasing feature of Visual Objects to protect every reference from undue influence. In this code, there is no dependence on the current work area remaining selected. Each line of code works on its own terms and is not subject to the vagaries of the context. The original version was fragile because it depended on its state (the current work area) to remain constant. The new version is more robust because it is stateless.

However, it is not robust enough to handle all the demands placed on a Windows application. GUI applications are typically designed for the multi-document, multi-window paradigm: in addition to this code fragment, there are other pieces of code that use other tables in other windows, simultaneously within the single application.

Multi-Tasking, Multiple Documents

Multiple tables in multiple windows does not by itself pose a problem for our code—the explicit alias referencing allows the code to survive even if other routines use other tables at the same time. Indeed, multi-tasking and event-driven processing increase the importance of making every statement sufficient unto itself, since it is even more likely that unanticipated operations have been going on when this program "turns its back."

Important! *In order to open the same table more than once, you must use shared mode; otherwise, a concurrency conflict will occur. The remainder of this discussion assumes that the SetExclusive() flag has been set to FALSE to enable shared mode as the default open mode. See Chapter 9 "Concurrency Control" for more information.*

The problem with the above code lies in the fact that while it can coexist with other code using other tables, what happens if the user wants to run the same function simultaneously in different windows for different departments, all using the same table? Can this code coexist with other copies of itself? Clearly, it cannot. Opening the table twice using the statement:

```
USE employee NEW
```

will fail because the two work areas will, by default, be opened under the same alias (employee), and this is not allowed. According to Xbase procedural programming conventions, all work areas must have unique aliases, and this approach leaves to the programmer the task of managing the different aliases.

It is indeed possible to specify an explicit alias when opening the table, but if you want two different aliases you must write two specific programs with different aliases explicitly written into the code:

```
FUNCTION Func1()
    USE employee ALIAS Emp1 NEW
    ...

FUNCTION Func2()
    USE employee ALIAS Emp2 NEW
    ...
```

However, this is no solution because two aliases may not be enough. What you really need is a program that can be started an arbitrary number of times and that finds a unique alias each time it starts. Although it is possible to write a program that finds a unique alias and uses this in all references, the solution is so technical that the original business logic ends up being completely swamped by the administration of these artificial aliases.

Indeed, this problem with dynamic allocation and management of work areas, aliases, and other components is one of the main reasons most business database programming systems do not support opening the same data window, with the same table, several times in independent windows. These systems thereby fail to meet one of the basic demands of Windows.

You need a solution that takes care of the technological barriers and lets you use the database as you intend. You need a program that can be run many times without stepping on itself. You need what the technologists call a *reentrant* program. In this case, you need one that is reentrant not only in itself but in its management of the database.

Object-Oriented Database Programming

The object-oriented database services of Visual Objects provide a different way of dealing with the database called a *data server*. Data servers are high-level, abstract entities that provide you with an object-oriented interface for databases. They hide the technological artifacts of the alias and the work area and make the database a self-contained object that can be used as many times as you want.

Look at the following comparison between the explicitly aliased version of the program (shown here with some syntactic artifacts removed) and a new, object-oriented program:

```
// Alias-qualified procedural style

USE employee ALIAS Emp NEW
Emp->(DBSetIndex("empno"))
DO WHILE .NOT. Emp->(EOF())
  IF Emp->Sex = "M"
    Emp->(DBDelete())
  ELSE
    Emp->Salary += Raise(Emp->Name)
  ENDIF
  Emp->(DBSkip())
ENDDO
```

```
// Object-oriented style

Emp := DBServer{"employee"}
Emp:SetIndex("empno")
DO WHILE .NOT. Emp:EOF
  IF Emp:Sex = "M"
    Emp:Delete()
  ELSE
    Emp:Salary += Raise(Emp:Name)
  ENDIF
  Emp:Skip()
ENDDO
```

The two programs are remarkably similar. The first statements, which open the table, appear different but do pretty much the same thing. In the object-oriented version, you create a new DBServer object for the Employee file and use the variable Emp to refer to it; in the procedural version, you select a new work area for the Employee file and use the alias Emp to refer to it. The variable Emp contains an object pointer, as opposed to the alias Emp in the traditional case—but this is a technicality. In both cases Emp is the handle by which you refer to this particular table.

But after this introduction, all the operations and field references look about the same: instead of being qualified with Emp followed by the alias operator (->), they are qualified with Emp followed by the *object message send operator* (:). Each function name is the same, except that the object-oriented versions (the methods) do not have the "DB" prefix.

In some cases, such as EOF, the object-oriented version has a virtual variable so the code does not use parentheses. The DBServer object dynamically reconfigures itself when the table is opened to allow references like Emp:Salary that look like exported instance variables.

The benefit of the object-oriented program is that it can be invoked as many times as you want, without stepping on itself. It is self-contained, reentrant, and does not require you to get involved in managing work area numbers or aliases. Many copies of the DBServer object can be instantiated, and each lives in its own world without needing anything from the outside.

This works particularly well in conjunction with object-oriented GUI programming using the GUI Classes library. As you will see in the 'User Interface Programming' section of this guide, a data window is also an object that can be instantiated many times without stepping on itself. The data server fits naturally with this approach: you create a data window and a data server and connect them with each other in a client-server relationship.

Referencing Multiple Databases Simultaneously

As mentioned above, other procedural solutions are possible if all you want to do is open multiple databases at the same time. Indeed, the trivial part of the solution is starting up several copies of the same program.

However, these solutions are not adequate if you need to refer to more than one work area at the same time. Imagine placing a push button on the "Employee Review" data window that copies a salary value from one employee to another. Here you need to refer to the two work areas, something you would traditionally do through the two aliases. Generating unique aliases is one thing, but keeping track of an unspecified number of them is quite another problem.

With the object-oriented approach, however, you can quite simply write a routine that takes two work areas, presented as DBServer objects, and copies the salary from the current record of the first work area to the current record of the second:

```
FUNCTION CopySalary(EmpSource, EmpTarget)
    EmpTarget:Salary := EmpSource:Salary
```

The Right Choice

The decision about what database access technology is right for you depends largely on what you want to do:

- If you have existing applications that use the procedural approach and you see no need to adapt them for operating in a GUI, event-driven environment, they will compile and run as is.
- If you want to upgrade a procedural application to use the data server approach, the task will not be as difficult as you might imagine and may be well worth the effort considering what you gain. For example, the DBServer Editor allows you to quickly create data servers by importing the structure of existing database files.

- If you are developing new applications, using the object-oriented approach from the start will definitely be to your advantage.

Database—DBF or SQL

Thus far in this chapter, the discussion has focused on DBF files and the alternative technologies available for accessing them, but you have another choice in Visual Objects and that is using SQL databases instead of, or in addition to, DBF files.

Visual Objects provides the SQLSelect class, which allows you to access SQL databases. If you are familiar with using traditional SQL embedded in a program, you may recognize that many of the same problems described above exist for SQL database applications as well. In particular, the requirements for specifying *cursor names* (akin to DBF aliases) get in the way of writing reentrant code. SQLSelect overcomes for SQL databases the same difficulties that DBServer overcomes for DBF databases.

The SQLSelect object is functionally equivalent to the DBServer object and using it in a program is identical to using DBServer, except for the way you create the object:

```
Emp := SQLSelect{"SELECT * FROM employee ORDER BY empno"}
...
DO WHILE .NOT. Emp:EOF
     IF Emp:Sex == "M"
          Emp:Delete()
     ELSE
          Emp:Salary += Raise(Emp:Name)
     ENDIF
     Emp:Skip()
ENDDO
```

Of course, there are other, unique ways to operate on a SQL database that have no counterpart in the DBF paradigm. These are supported via the SQLStatement class and include such operations as mass updates and deletions.

Use of these features will make your applications less amenable to changing from one database format to another but, in fact, these types of operations are rarely sufficient for business applications. Complex calculations as well as involvement by the end user generally require creating a cursor (using SQLSelect) and embedding business logic written in a programming language as described above.

SQL databases also have other constraints. For example, many cannot move backwards and, depending on how the cursor was originally defined, a SQL database may not support updates. These constraints are usually covered in the documentation supplied by the database manufacturer.

Thus, from a language standpoint, the decision of which database format to use is a trivial one, because most of your code is reusable no matter what you decide. Your decision to use SQL or DBF databases will probably be based on other issues, such as security, performance, and data volume requirements.

Chapter

7

Data Server Classes

Visual Objects provides several built-in classes that enable you to design applications that are, to a large degree, independent of the type of data storage used (for example, DBServer for DBF files and SQLSelect for SQL database). This chapter provides an overview of the data server classes, discussing the common philosophy behind their designs and suggesting ways to exploit the similarities in order to make your applications as data-independent as possible.

For more information on any of the classes discussed in this chapter, see the *Visual Objects Online Help.*

Data Servers

A *data server* is an object created from a subclass of the DataServer class. Data servers are the object-oriented tools for data windows to interact with databases. There are built-in data servers of different types: the two most important ones, DBServer and SQLSelect, work with databases. Other servers can be written that present arrays and DOS directories in a tabular format that allows them to be manipulated like databases.

All of these data servers provide a common set of methods based on a common database paradigm. These fundamental operations include getting and setting field values, moving forwards and backwards by (logical) record positions, deleting or modifying the current record, inserting a new record, and searching for a given value. The naming conventions used for the various methods make them similar, if not identical, to corresponding Xbase command and function names, simplifying the transition from procedural to object-oriented database programming.

The way a data server is defined is not standardized; it differs among the different types of servers. For example, an SQL database supports quite complex SELECT statements, while an Xbase work area is based on a simple definition, a complex indexing technology, and the concept of relating several work areas. All of these variations are provided in class-specific extensions to the basic protocol.

In particular, the way each class instantiates objects is unique. However, once an object has been created and the database opened, its clients can use the server the same way. Thus, you can instantiate a data server with any one of the first two sets of statements shown below, present it as a server to a data window and, after that, the data window can support a number of standard operations:

```
// Xbase
oServer := DBServer{"customer"}
oServer:SetIndex("custno")

// SQL
oServer := SQLSelect{"SELECT * FROM customer ORDER BY custno"}

oDW := DataWindow{SELF, DWResourceName}
oDW:Use(oServer)
oDW:Skip()
? oDW:CustNo, oDW:CustName, oDW:Address
oDW:CustName := "Jones"
oDW:Seek(12345)
oDW:Delete()
oDW:Skip(5)
oDW:Commit()
```

The data manipulation methods of the data window simply turn around and invoke the corresponding methods of the data server, which works because all the servers support the same basic methods.

The DataServer class hierarchy is designed and implemented specifically to offer you a consistent interface with data presented in different formats. You can also define your own data servers by subclassing the DataServer (or any other data server) class. By following the method naming conventions of the built-in data servers, you maintain the interface consistency and make your applications easier to adapt to changing formats.

DBF Servers

The DBServer class is designed to work with Xbase-style DBF files, giving you all the functionality necessary to manage them, including indexing, setting relations, and performing mass operations, such as total and update.

Instantiating a DBServer object is equivalent to opening a DBF file in a work area, but the work area and alias concepts are not integral to your use of the data server. Because the DBServer class manages the alias and work area information behind the scenes, you can easily design applications that are reentrant and take full advantage of GUI environments.

The DBServer class was discussed briefly in Chapter 6, "Justifying Database Access Choices," which explained how using it solved certain critical problems of GUI programming that are not easily addressed through a procedural approach. The DBServer class is discussed more fully in Chapter 8, "Using DBF Files."

SQL Servers

The SQLSelect class is designed to let you work with SQL databases in your applications. It does for SQL databases what the DBServer class does for DBF files, using the same interface and overcoming the same difficulties that programming in a GUI environment presents. Using SQLSelect takes you a step beyond the ODBC approach to SQL programming, eliminating the need for automatic generation of cursor names that can add an undesirable degree of technological complexity to your application.

The only difference between the way you use SQLSelect and DBServer is the way in which you instantiate the class. With SQLSelect, you use a standard SQL SELECT statement to define the records (or *rows*) you want to select and the order in which you want them presented. From this point on, however, you operate on the SQL database in the same manner as you would a DBF file. For example:

```
Emp := SQLSelect{"SELECT * FROM employee ORDER BY empno"}
DO WHILE .NOT. Emp:EOF
     IF Emp:Sex == "M"
          Emp:Delete()
     ELSE
          Emp:Salary += Raise(Emp:Name)
     ENDIF
     Emp:Skip()
ENDDO
```

If you are familiar with SQL, you might wonder how this is possible. The SQL and Xbase paradigms are quite different in terms of the operations you can perform, and one cannot successfully emulate the other. This is the reason Visual Objects must maintain separate server classes for accessing DBF and SQL databases—two methods (one from each class) can have the same name with completely different implementations.

The approach in the example above, referring to fields and doing Skip until EOF is true, is the traditional Xbase approach. The standard SQL approach is slightly different: you do Fetch until it indicates no more data. Skip is not required because the Fetch operation by itself moves forward. Using Skip or Fetch to move to the next record and make the data available might appear like a trivial distinction. The approaches are also semantically different. Opening a DBF table automatically positions the file pointer to the first record, but an SQL database is not positioned until after the first Fetch operation.

The SQLSelect object also supports the classical SQL-oriented approach:

```
Emp := SQLSelect{"SELECT * FROM employee ORDER BY empno"}
DO WHILE Emp:Fetch()
     IF Emp:Sex == "M"
          Emp:Delete()
     ELSE
          Emp:Salary += Raise(Emp:Name)
     ENDIF
ENDDO
```

As a user of the SQLSelect class, it is not important to you that the methods are implemented in a different manner than in the DBServer class. The important thing is that, because method names are standardized, the code is consistent no matter what data server you use.

Field References in Object-Oriented SQL

References to database fields (SQL calls them *columns*) are done with the object-oriented style introduced above, very similar to the alias-qualified style of Xbase. Standard procedural SQL provides a different approach in which an SQL statement is *bound* to individual variables. In this approach, the variables automatically hold the data after the Fetch operation. This approach does not allow direct updating of the fields—to update the fields of the current record, you must execute an SQL statement something like this:

```
<Open SQL cursor>
<Locate specific record>
UPDATE employee SET Salary = 75000
     WHERE CURRENT OF CursorName
```

Embedded SQL and ODBC provide no simple way of doing this. In contrast, the SQLSelect class allows you to refer to field values as if they were ordinary variables. In this case, you write:

```
Empl := SQLSelect{"SELECT * FROM employee"}
<Locate specific record>
Empl:Salary := 75000
```

The object-oriented framework gives you a very simple way of referring to the contents of different cursors, avoiding the complexity of constructing convoluted SQL strings with dynamically inserted cursor names.

Other SQL Operations

This approach is not the only way of dealing with an SQL database. SQL is a powerful language that allows many operations to be done without program logic.

SQLStatement

To let you perform any SQL statement you want, Visual Objects provides the SQLStatement class. For example, instead of using a DO WHILE loop to process all the records meeting a particular condition (as shown in an earlier example), you could do it with a single statement:

```
SQLStatement{"DELETE FROM customer WHERE Sex = 'M'"}
```

SQLConnection

SQLConnection is another class designed to work with SQL databases. This class allows you to define a specific connection to an SQL database, including a driver, user name, and password. Once instantiated, an SQLConnection object can serve as an argument during SQLSelect and SQLStatement object instantiation.

Data Fields and Field Specifications

The DataField and FieldSpec classes let you associate certain properties with a database field, including data type, context-sensitive help, a prompt, a field caption, formatting, validation rules, error messages for the validation rules, and help for the error messages.

By using DataField and FieldSpec objects in your application you can encapsulate these attributes with the specific data, which makes writing your applications easier. In addition, you can reuse DataFields and FieldSpecs by putting them in libraries or DLLs and including them in our applications or modules.

Data Fields

A DataField object (or a data field) is defined by three properties: *Name*, *FieldSpec*, and *HyperLabel*.

Name

Each database table is defined by several data fields arranged in a specific order. Thus, it is possible to refer to a field by its number or by its name (which is a string)—either will uniquely identify the field for each database. However, because the name is likely to be more invariant over time, the DataField class does not allow access to the field number.

Note: The NameSym property returns the data field name as a symbol instead of a string, which may be more efficient in some circumstances. See Accessing Fields in Chapter 8, "Using DBF Files" for more information on the advantages of using symbols over strings.

FieldSpec

The FieldSpec property (or field specification) is defined as a FieldSpec object. FieldSpec is a comprehensive class that contains a formatting picture and several different validation rules, as well as methods for performing the validations, formatting, and converting data from one type to another. Database people often call this specification a *domain* or an *abstract data type*.

HyperLabel

The HyperLabel property is defined as a HyperLabel object. HyperLabel is a simple class: essentially a set of labels of different types, from a programmer's symbol to a caption, a description, and a unique identifier for linking into a context-sensitive help system. Hyperlabels are attached to most objects in the system (rather like shipping labels on a package) and make it convenient to present the user with meaningful information about what is going on in the application. For example, not only does each DataField object have a hyperlabel, each FieldSpec object has one as well. See Chapter 16, "Hyperlabels" for more information.

A Data Field's Relationship to Its Properties

The relationships between these objects are simple: the DataField *has a* FieldSpec and a HyperLabel. And, since the field specification is independent of the data field definition, it is possible—and common—for several data fields to share the same field specification.

Field Specifications

For example, a database system may have only one field specification defined for employee numbers, because they all have the same type, size, format, validation rules, etc. However, the "Employee" database would have two fields (the number of the employee and the number of the employee's manager) that use the employee number field specification.

Thus, the database system can define several classes:

```
CLASS EmployeeNumberSpec INHERIT FieldSpec

METHOD Init() CLASS EmployeeNumberSpec
     ValType := "C"
     UsualType := STRING
     Length := 10
     Decimals := 0
     Picture := ...
     HyperLabel := HyperLabel{#EmployeeNumber, ;
          "Employee Number", ... }

CLASS EmployeeNumber INHERIT DataField

METHOD Init() CLASS EmployeeNumber
     SUPER:Init(#EmpNo, EmployeeNumberSpec{})

CLASS ManagerNumber INHERIT DataField

METHOD Init() CLASS ManagerNumber
     SUPER:Init(HyperLabel{#MgrNo, ;
          "Manager Number"}, EmployeeNumberSpec{})
```

In this source code, you will note that both the DataField subclasses, EmployeeNumber and ManagerNumber, use the same FieldSpec subclass, EmployeeNumberSpec.

Hyperlabels

This relationship also helps explain why both the data field itself and its field specification have a hyperlabel. The EmployeeNumberSpec class has a hyperlabel that includes generic descriptive information about all employee numbers. EmployeeNumber, as a general employee number, simply uses the hyperlabel of its field specification, but the ManagerNumber is a specific type of employee number and prefers to create its own hyperlabel.

In fact, this structure is littered with hyperlabels. Every validation rule in the field specification (there are several of them, such as data type, maximum size, minimum size, required, range, and other, developer-defined validations) has an error message to display if a value is rejected. This error message is actually a hyperlabel to provide a hook for attaching context-sensitive help to the validation. Any hyperlabel that is not provided is automatically created by the system.

How Data Servers Use Data Fields and Field Specifications

Data fields and field specifications, therefore, are not a physical part of the database file—they are defined by code in your application. You connect them to a database file using a data server (also defined by code in your application) that defines the relationships between the database file and the properties you have defined.

When you use either the DBServer Editor or the SQL Editor to create a data server, the code for the data fields, field specifications, hyperlabels, and data server, is generated automatically based on your design. However, you can use the DataServer, FieldSpec, and HyperLabel classes directly to achieve the same benefits with data servers defined elsewhere.

You can use the generated code for a data server as a model for how to define your own, but the basic idea is to subclass an existing data server and include a DataField instantiation for each field in its Init() method. The data fields, of course, use field specifications and hyperlabels that you must also define.

Note: Keep in mind that defining data fields and field specifications is *completely* optional—you can ignore them altogether. If they are needed but not available, the system will generate data fields and field specifications for any database file based on its record layout. This happens at runtime (for example, when you instantiate the database file as a DBServer object).

How Data Windows Use Data Fields and Field Specifications

In addition to defining field specifications for use with a data server and explicitly linking them to data fields, you can define field specifications for use with a data window (or data browser) and explicitly link them to controls (or columns).

Here again, defining field specifications is optional. When designing a data window, if you do not provide a field specification for a particular control, it automatically picks up one from the data field to which it is linked. And, as stated earlier, a standard field specification is generated at runtime, if necessary, so the data window always has something to fall back on. Thus, the ability to define field properties in this manner represents an opportunity rather than a requirement. The main benefit is that the properties you define as part of the data server are reusable.

Note: Just as it is not necessary to explicitly design a data server with data fields and field specifications geared toward a particular database file, it is also not necessary to explicitly design a data window for a particular database file. The data window can automatically create a default layout based on the characteristics of the linked data server. Thus, you can instantiate *any* database (new or old) as a DBServer object, instantiate a DataWindow object that uses this data server, and view and edit the contents of the database file—even though neither the data window nor the data server has ever seen the database file. (An excellent example of this can be seen in the Standard Application generated by Visual Objects).

It is when you link a data server to a data window that the properties of the data fields come into action. The data window uses the field specifications to properly display controls along with their correct captions and prompts, to give the proper help text when the user requests it, to validate the information entered by the user, and to offer helpful error messages when something goes wrong. See Chapter 11,"GUI Classes" for more information on data windows, data browsers, and their relationship to field specifications.

Other Data Servers

Earlier in this chapter, it was stated that you can define your own data servers and, in fact, this is just what you are doing when you use the data server editors provided in the IDE. With the DBServer Editor, for example, the system automatically creates a subclass of DBServer to hold all of the information that you specify (such as field validation rules, help, formatting, and disk file name and location).

For example, after designing a Customer database file, the following statement would be generated (along, of course, with many others):

```
CLASS Customer INHERIT DBServer
```

Then, to instantiate this class you would use:

```
LOCAL oDBCust AS Customer
    oDBCust := Customer{}
```

All data servers that you design in this manner will be very similar to one another in functionality, even though each has its own unique properties; however, you can also create special purpose data servers by subclassing any class in the DataServer hierarchy, which is the focus of this section.

Joining Tables

The ability to define servers is a very powerful feature that you might want to explore on your own. For example, a server can act as a piping connection to define operations involving two or more servers. Consider joining two databases with SetRelation, for example. When joining the Employee table with the Department table, the Skip operation is well defined. But what does it mean to do a Delete operation on the Employee table? Should you delete the employee, or the employee and the department, or refuse to perform the operation? The answer would be different for Customers and Orders or for Orders and Items. A general purpose data server class cannot guess the correct semantics in each case because they depend on the application logic.

The solution would be to define a specialized data server that is capable of joining other servers. Very briefly, the server would accept as instantiation arguments two predefined servers that you would relate in the Init() method. Then, you would define, using methods, how operations such as Delete should work for the joined tables and how fields should be returned and updated.

Buffered Servers

Another possible extension would be a DBServer that buffered changes, allowing rollback of a large number of changes. The standard DBServer class allows changes to the current record to be discarded and the original field values restored, but once you have moved off the record, the changes are permanent as far as the program is concerned.

Such a buffered server could sit between the client window and the actual data server, storing changed values in memory and only writing them out when the application invokes a Commit operation.

Chapter

8

Using DBF Files

Visual Objects lets you design applications built around the management of Xbase-style databases (called *DBF files*) with a complete set of language facilities designed specifically to manipulate these databases. The language facilities vary slightly, depending on the approach you use.

Note: This was discussed in "Justifying Database Choices," which touched on the differences and similarities between the procedural and object-oriented approaches to database programming and the benefits of using the DBServer class.

This chapter, while not elaborating on every possible database operation available, provides an overview of using DBF files from both the procedural and object-oriented perspectives.

For more specific information, refer to the online help system which includes a list all commands and functions designed for manipulating database files. The Online Help will also provide you with information on the DBServer methods and instance variables.

Databases and Work Areas

The Visual Objects database system is designed around the *work area*, a component for using a single database file and multiple index files. A work area is *occupied* or *unoccupied*, depending on whether it contains an open file. At application startup, all work areas are unoccupied, and work area number one (1) is the current work area.

A DBF file, also called a *table*, consists of one variable length header record that defines the file structure in terms of its field (or *column*) definitions, and zero or more fixed length records (or *rows*) that contain the actual data. Each record has one additional byte for the record delete status flag.

The file structure is defined and added to the DBF file when the file is created. It consists of one or more field definitions describing the name, width, and data type attributes for each column in the table. Normally, you will define database file structures when you are designing the application that will use them. (You can use the DBServer Editor, documented in the *IDE User Guide*, for this purpose). Visual Objects can also operate with DBF files created by other applications, including CA-Clipper, CA-dBFast, and other Xbase systems.

The table rows are added to the file at application runtime using data windows which understand and enforce the file structure and other validation rules.

The data for all field types except memo are stored directly in the DBF file. Memo fields are maintained in a separate file with the same name as the DBF file and a .DBT or .FPT extension. The DBF file contains pointers to the data in the memo file, but all of this is completely transparent to you as a developer and to the end users of your application.

Replaceable Database Drivers

It was stated earlier that Visual Objects can operate with DBF files created by applications made with Visual Objects, Xbase, or by the IDE. This is largely because DBF formats have remained fairly well standardized over the years. Index files, on the other hand, have remained distinct and unique to the product that invented them. CA-Clipper NTX files are different from dBASE IV MDX files, but the DBF file formats for these products are largely compatible.

Visual Objects supports an RDD technology that allows you to choose different file formats, including database, memo and index, within the same application. This feature lets you tailor your applications so that migrating from one format to another is simple and straightforward, and plays an important role in the ability of your Visual Objects applications to share data with other applications (see the Data Sharing section later in this chapter).

The following RDDs are supplied:

RDD Name	Product
DBFCDX	FoxPro
DBFBLOB	Enhanced FoxPro .FPT file support and BLOB (binary large object) file support
DBFMDX	dBASE IV
DBFNTX	Visual Objects and CA-Clipper

See Appendix A, "Rdd Specifics," in this guide for more details on the RDDs and any special characteristics they may have.

Choosing an RDD

All RDDs listed in the table are part of the System Library, so you do not have to do anything special to make them available to your application. You indicate which RDD you want to use by name when you open a database file (for example, USE...VIA "DBFMDX" or as an argument when you instantiate a DBServer object). Thus, you can access multiple file formats in different work areas in the same application. DBFNTX is the default RDD, but you can change the default through the RDDSetDefault() function.

Common Interface

Regardless of the RDD, you will operate on the database using a common interface, minimizing the amount of code you must change to use an application with a variety of file formats and eliminating the need to learn a different syntax for each format. The only limitations in the language are those imposed by the original manufacturer of the database/index format or by restrictions that are inherent in the file structure. For example, CA-Clipper does not support multiple orders per index file. Allowing the supplied RDDs to overcome these limitations would render the database and index file incompatible with the applications originally designed to use them.

Third-Party RDDs

The examples in this chapter use the default RDD but will also work with the other supplied RDDs because of their common language interface. However, because the RDD technology is designed in keeping with Visual Objects open architecture philosophy, it is possible and probable that RDDs from third-party developers will become available. If you choose a third party RDD, it may not work with all the language components because of limited functionality associated with the database. See Chapter 18, "Third-Party Components" for more information about choosing third-party components.

Language Overview

The Visual Objects language provides a complete set of commands and functions for manipulating an open database in a procedural manner. While most of the commands have an equivalent in the form of a DB…() function (and a strongly typed VODB…() function), there are functions that do not have command equivalents, such as EOF() and RecNo(), and commands, such as AVERAGE, that do not have functional equivalents.

The DBServer class provides each of the basic database operations as a method. When corresponding to a DB function, the "DB" is dropped from the method name but, otherwise, the method name is the same as the command or function that it implements. In other words, there are at least three, and in most cases four, ways to perform every database operation:

Style	Example
Command	SKIP <*n*>
Function	DBSkip(<*n*>)
Strongly Typed Function	VODBSkip(<*n*>)
DBServer Method	oDBServer:Skip(<*n*>)

Commands vs. Functions vs. Methods

You will probably use only one of these styles for most of your programming and, for the most part, there is a straightforward mapping between the styles of operation: the command, the DB function, the VODB…() function, and the corresponding method are essentially the same.

In some cases, a command and its corresponding DB function are not exactly the same. For example, the DELETE command differs from the DBDelete() function in that the command allows record scoping. Internally, DELETE is implemented using DBDelete() with the record processing function, DBEval().

In cases like this, the DBServer method provides the greater functionality. For example, the Delete() method lets you operate on the current record or specify record scoping using code blocks.

There are also some database operations that act globally on all open database files (for example, DBCloseAll() and DBCommitAll()) and are, therefore, not methods of the DBServer class. Methods apply to a particular object, not to other objects.

Finally, any function whose primary purpose is to return a value, such as a status flag or database attribute, is provided in the DBServer class as an ACCESS method, which means that you refer to it without parentheses as you would an exported instance variable. Examples of this are BOF, EOF, RecNo, and Header.

File Specifications

Using functions, you normally specify database file names and aliases as strings. Using DBServer methods (and during instantiation), you can also specify them as strings, but the FileSpec class provides a more powerful way of dealing with file names.

In any method of the DBServer class, you can use a FileSpec object instead of a string whenever a file name is required. Using a FileSpec object has certain advantages over using a file name. This is discussed in Chapter 15, "File Handling." Similarly, when a DBServer method requires a reference to another DBServer object, you should normally specify the object itself, but you can also specify an alias as a string or a symbol.

Accessing Fields

Whether you are using the procedural or object-oriented style of database programming, you refer to fields in one of two basic ways:

- Referring directly to the field names
- Using FieldGet() and FieldPut(), with the field identified through a parameter

Field Names

In traditional Xbase programming, you may refer to field names with or without an alias qualifier. It is recommended that you use the alias qualifier, of course, to avoid dependence on the current work area, as shown in the example below:

```
// Traditional alias-qualified field reference
USE customer ALIAS DBCust
? DBCust->CustName
DBCust->CustName := "John Smith"
```

With DBServer, field names are implemented as virtual variables, making field references very similar to traditional Xbase field references with the alias operator:

```
// Virtual instance variable
oDBCust := DBServer{"customer"}
? oDBCust:CustName
oDBCust:CustName := "John Smith"
```

FieldGet() / FieldPut()

The FieldGet() and FieldPut() methods of the data server classes behave essentially the same as the FieldGet() and FieldPut() database functions. One key difference is the extra flexibility in the field selection parameter: the FieldGet() and FieldPut() *methods* allow the specification of the field name in three different ways:

- Field name as a symbol
- Field name as a string
- Field position as a number

For example:

```
oDBCust := DBServer{"customer"}

cName := oDBCust:FieldGet(#CustName)
cName := oDBCust:FieldGet("CustName")
cName := oDBCust:FieldGet(3)

oDBCust:FieldPut(#CustName, cName)
oDBCust:FieldPut("CustName", cName)
oDBCust:FieldPut(3, cName)
```

Using a symbol is often the most practical and efficient approach. By comparison, the two other approaches have some minor disadvantages:

- String comparison is slower than symbol comparison, so specifying the field name as a string imposes some extra overhead in the lookup process. In practice, the overhead is usually insignificant compared to all the other work going on in an application, from disk I/O to screen handling. The difference is of interest only for those who want to squeeze every last fraction of performance out of a program.

  Another issue is programming style: many developers feel that strings should be used to contain only things that the end user sees and that programmer-level information should be held in other data types, such as symbols. You will find this to be true in the system and class libraries and in most of the examples you see in the documentation, but you should make your own choice. As in all matters of programming style, the most important thing is consistency.

- Referring to fields by position makes the application code dependent on the structure of the database. This is generally not a good idea for applications—the customer name field is always called CustName, but it may not always be field number 3. During the lifetime of an application, the database structure may change. An application may also be applied to another table that contains similar data and column names but has a different structure.

  Field reference by position is primarily of value for general purpose utilities rather than business applications; such utilities do not know anything about the contents of each database, and reference by position makes sense without costing anything.

Which Approach to Use

As discussed previously in "Justifying Database Access Choices," commands are not easily adapted to multi-tasked GUI programming and are not recommended. Similarly, since they do not address the problem of easily opening the same database in multiple windows, the DB functions also fall short in GUI applications but are, otherwise, quite flexible and useful.

The DBServer methods are fully polymorphic, at least as flexible and tolerant of parameter values as the DB functions, and often more so. Because they inherently support multi-instance operation, this is the recommended way of doing database operations in a GUI environment.

The strongly typed database functions, such as VODBSkip(), are equivalent to the DB functions except that their parameters and return values are strongly typed and they do not call the runtime error handling system. These functions are quite particular about how they are used, but they are faster and allow for more robust programming. Because all references are *early bound* (that is, everything about them is known at compile time), nothing needs to be resolved at runtime, and many errors are detected at compile time. These functions are recommended for performance-critical database operations that do not involve user interaction.

Hybrid Programming

Although not recommended, it is possible to mix procedural database operations with object-oriented operations in the same program and even for the same database. You can retrieve the alias of a DBServer object and use it to gain direct access to the work area:

```
Select(Emp:Alias)
Salary += 100
```

This approach may be useful as a way of recycling a large piece of existing procedural code within an object-oriented framework. Of course, the approach works only if the procedural code retains control throughout and does not allow another task within the new GUI program structure to subvert the current work area concept.

It is even possible (but rarely a good idea) to retrieve the alias and the work area number of a database server and use them directly in procedural code:

```
EmpAlias := Emp:Alias
EmpWA := Select(EmpAlias)
EmpWA->Salary += 100
```

Two cautions are appropriate when using a hybrid approach:

- Be aware that neither the database server nor its clients are aware of changes made through the procedural path, and such changes are not propagated up to the data window automatically. In general, all of the automatic provisions of the object services are potentially subverted when the program bypasses them.
- Do not close the database and open a new one in a work area that is used by a DBServer object. The DBServer knows the structure and the alias of its database and work area and, if you change this, the methods of the DBServer will malfunction.

Hybrid programming is complicated and not recommended except in the most extreme cases. In general, you should choose a model and stick with it.

Using a mix of procedural and object-oriented database programming, however, may make sense in certain cases. For instance, you may choose to do non-interactive, performance critical database operations using the procedural approach (and the VODB functions) and database operations that require user interaction with the DBServer class. In this case, you would not mix the two styles of programming but would use one or the other depending on the requirements of your application.

Record Scoping

Many database operations can process subsets of records within a work area using a scope and conditional clauses. For any command that allows one, the syntax of the <*Scope*> is as follows:

```
[ALL | NEXT <nRecords> | RECORD <nRecord> | REST]
```

- ALL processes all records
- NEXT processes the current record and the specified number of records
- RECORD processes the specified record
- REST processes all records from the current record to the end-of-file

Commands specified without a scope default to the current record (NEXT 1) or ALL records, depending on the command. For example, DELETE and REPLACE process only the current record, whereas AVERAGE processes all records. Specifying a scope changes this default by indicating how many records to process and where to begin.

The set of records processed can also be restricted using a conditional clause that specifies a subset of records based on a logical condition. The two conditional clauses are FOR and WHILE.

A FOR clause defines a condition that each record within the scope must meet in order to be processed. If no scope is specified, FOR changes the default scope to ALL records.

A WHILE clause defines another condition that each record processed must meet; as soon as a record is encountered that causes the condition to fail, the command terminates. If no scope is specified, WHILE changes the default scope to REST.

Specifying a scope, a FOR clause, and a WHILE clause within the same command syntax can raise questions regarding the order in which the clauses are processed. The scope is evaluated first in order to position the record pointer. Then, the WHILE clause is evaluated and, if the condition is not met, the process terminates. If the WHILE condition is met, the FOR clause is evaluated. If the FOR condition is also met, the record is processed; otherwise, it is not. Either way, the record pointer is moved to the next record within the scope until the scope is exhausted.

Scoping Through Method Parameters

Since using commands is not recommended, the preceding discussion may seem superfluous. However, all of these commands have DBServer method equivalents that support record scoping using a slightly different syntax.

First, commands whose default scope is not ALL have an additional method to support the ALL scope (for example, RecallAll() and DeleteAll()). Commands whose default scope is ALL do not have these methods because they would be redundant (for example, Average() averages all records, by default).

All scope-dependent methods allow explicit specification of a scope and FOR and WHILE clauses. For example, the Recall() method has this syntax:

```
Recall(<cbForBlock>, <cbWhileBlock>, <uScope>)
```

The first parameter is a code block representing the FOR clause:

```
// RECALL FOR Last = "Smith"
oDB:Recall({|| Last = "Smith"})
```

The second parameter is the WHILE clause:

```
// RECALL FOR First = "Jack" WHILE Last = "Smith"
oDB:Recall({|| First = "Jack"}, ;
     {|| Last = "Smith"})
```

The third parameter is the scope clause, and can be used to specify ALL records, the REST of the records, or the NEXT <*n*> records:

```
// RECALL WHILE Last = "Smith" ALL
oDB:Recall(, {|| Last = "Smith"}, DBSCOPEALL)

// RECALL FOR Last = "Smith" REST
oDB:Recall({|| Last = "Smith"},, DBSCOPEREST)

// RECALL WHILE Last = "Smith" NEXT 10
oDB:Recall(, {|| Last = "Smith"}, 10)
```

Note: With this approach, there is no equivalent to processing a specific record number.

With no parameters, the Recall() method is equivalent to the RECALL command and the DBRecall() function, recalling the current record only. RecallAll() is equivalent to RECALL ALL.

Preset Scopes

The DBServer methods provide another way of specifying a scope, one that may be more convenient in many cases. The FOR clause, WHILE clause, and scope can be considered properties of the DBServer object and can be assigned directly:

```
// RECALL WHILE Last = "Smith" ALL
oDB:WhileBlock := {|| Last = "Smith"}
oDB:Scope := DBSCOPEALL
oDB:Recall()

// RECALL FOR Last = "Smith" REST
oDB:ForBlock := {|| Last = "Smith"}
oDB:Scope := DBSCOPEREST
oDB:Recall()

// RECALL WHILE Last = "Smith" NEXT 10
oDB:WhileBlock := {|| Last = "Smith"}
oDB:Scope := 10
oDB:Recall()
```

This approach is very convenient, especially when several operations are to be done with the same scope. It also provides for more structured programming. One caution: the ForBlock, WhileBlock, and Scope are persistent—they remain in effect until reset. Do not forget to reset the scope after using it, or you may accidentally delete all the records!

Indexing

Records are stored in the same order (called the *physical* order) as they are added to the DBF file and, by default, this is the order in which you will access the database. However, the physical order is not usually the ideal way of viewing the database since data is often added in no meaningful order. Most applications require that the records be ordered *logically*, according to the contents of one or more fields. You accomplish this using *orders* that are stored in *index* files. Orders not only let you order the DBF file to suit your application, but also give you quick access to data using keys and allow you to set up complex relationships between DBF files.

The indexing technology used by your application depends on the RDD, as mentioned earlier. For example, using traditional Xbase DBF files, each order is stored in a separate index file, but other database formats support more than one order per file. In cases where only a single order per file is allowed, you will often hear the terms order and index used interchangeably. Regardless of whether you are accessing single-order or multiple-order indexes, the language facilities are very similar—there are DB functions and corresponding methods that work with both types of index.

If the index files are designed properly, you may have to change only the way you open the files, leaving all operational statements the same. For example, if you have several single-order index files, you could use this code to open and select between them:

```
// Object-oriented approach
LOCAL oDBCust := DBServer{"customer"}
oDBCust:SetIndex("custno")
oDBCust:SetIndex("custname")
...
oDBCust:SetOrder("custname")
...
oDBCust:SetOrder("custno")
...
```

```
// Procedural approach
USE customer ALIAS DBCust
DBCust->(DBSetIndex("custno"))
DBCust->(DBSetIndex("custname"))
...
DBCust->(DBSetOrder("custname"))
...
DBCust->(DBSetOrder("custno"))
...
```

If you have a multiple-order index in which the orders have the same names as the single-order index files in the previous example, you would have to change very little to adapt this code:

```
// Object-oriented approach
RDDSetDefault("DBFMDX")
...
LOCAL oDBCust := DBServer{"customer"}
oDBCust:SetIndex("customer")
...
oDBCust:SetOrder("custname")
...
oDBCust:SetOrder("custno")
...
```

```
// Procedural approach
RDDSetDefault("DBFMDX")
...
USE customer ALIAS DBCust
DBCust->(DBSetIndex("customer"))
...
DBCust->(DBSetOrder("custname"))
...
DBCust->(DBSetOrder("custno"))
...
```

Relating Databases

Several DBF files that have related structures and data can be associated, along with their indexes, using DBSetRelation() or SetRelation(). These let you establish relationships between several files and operate on them as a single entity known as a database, or view.

For example, these statements link the Customer and Orders database based on the CustNum field that they have in common:

```
// Object-oriented approach
LOCAL oDBCust := DBServer{"customer"}
LOCAL oDBOrd := DBServer("orders")
oDBOrd:SetIndex("custno")
oDBCust:SetRelation(oDBOrd, #CustNum)
```

```
// Procedural approach
USE customer ALIAS DBCust NEW
USE orders ALIAS DBOrd NEW
DBOrd->(DBSetIndex("custno"))
DBCust->(DBSetRelation("DBOrd", ;
        {||DBCust->CustNum}))
```

Once the link is established, the two files are treated as a single database in the sense that when the parent (Customer) moves, the child (Orders) moves along with it, finding the appropriate CustNum match each time. Using this feature, you can establish master-detail relationships between your DBF files and reflect the relationship using form and browse views.

Selective Relations

The traditional Xbase relation is only a partial solution: it positions the child table to the correct record, but it does not limit operations to those records that match the relation. This means that if you move forward in the child table, the system does not prevent you from moving on to the orders for the next customer. It also does not prevent you from moving backward to orders belonging to other customers.

A *selective relation*, which is established the same way as a traditional selection, adds automatic filtering: the child database behaves as if only records matching the relation existed. Attempts to move beyond the last order for the current customer raise an end-of-file condition, and attempts to move before the first raise the beginning-of-file condition. Similarly, operations like GoTop and GoBottom move only within the group of matching records, and properties like RecCount reflect the number of records in the selection.

This selective relation significantly simplifies programming of the typical *master-detail* applications: the child window can operate as if it deals with a complete table, without regard for matching field values. It also means that a general browser for a table may be used as a child table browser without special programming:

```
oDBCust := DBServer{"customer"}
oDBOrd := DBServer("orders")
oDBOrd:SetIndex("custno")
oDBCust:SetSelectiveRelation(oDBOrd, #CustNum)
oDWCust := CustomerWindow{...}
oDWCust:Use(oDBCust)
oDWOrd := DataWindow{oDWCust}
oDWOrd:Use(oDBOrd)
oDWOrd:Show()
oDWCust:Show()
```

A note about performance: the selective filtering obviously adds some overhead to all regular processing, but not more so than the filtering you would normally have to do in the application.

Note: For scope-sensitive operations like Average, the default scope is subject to the selective relation: if you set up a selective relation and perform an unscoped Average operation, you will get the result from the records in the selection only. An explicitly specified scope overrides the selective relation, however.

Undoing Changes

A common requirement in GUI applications is the Undo or Cancel operation: discarding all changes that have been made to the database with direct field assignments. To implement this behavior, you need a kind of buffered processing, where changes are not made directly to the database but kept in a buffer temporarily until they are either written out to disk or discarded.

In fact, all traditional Xbase database processing buffers the data (in the RDD), but the buffer is not accessible with standard Xbase database commands. Visual Objects adds the DBBuffRefresh() function and the Refresh() method for refreshing the buffer from disk, in effect rereading the old values from the disk and discarding the changes that have been made, but not yet committed.

With this approach, a data window can continue to operate with standard database operations, whether procedural or object-oriented, and the Undo/Cancel operations are implemented using the appropriate refresh operation.

Note, however, that this approach buffers one record only. To produce a more complete buffering approach that maintains all changes in an in-memory buffer until committed, the best approach is to produce a separate buffer server that sits in front of the DBServer object. This approach is discussed briefly in Chapter 7, "Data Server Classes."

Data Sharing

When accessing a database designed for use with an application written in another language, the questions of *compatibility* and *interoperability* arise. Compatibility refers to non-simultaneous access of the same data by Visual Objects and other applications, and interoperability refers to simultaneous access of the same data by Visual Objects and other applications. Both are especially important issues for applications designed to run in the Windows environment.

Compatibility

When sharing data between Windows and DOS applications, you must be aware of the issue of character sets. Many other Xbase development systems, including CA-Clipper, operate under DOS and use the DOS character set, called OEM (Original Equipment Manufacturer) or ASCII. Visual Objects, on the other hand, operates under Windows and therefore uses the Windows character set, called ANSI (American National Standards Institute).

For English letters and numbers, there is no compatibility problem. However, any application that uses words in other languages might be affected by these differences. Names such as François Dônèl, Håkan Strömbäck, Ängström, and Gnädige Müller-Strauß are not represented the same way in DOS and Windows programs. Files written in one environment may not be interpreted correctly by a program in another environment. When sharing data between DOS and Windows applications, care must be taken.

Visual Objects operates with the ANSI character set internally and uses the ANSI character sets in its data files, but it also understands OEM data files. When it detects that a file was created by a DOS application, such as CA-Clipper or dBASE IV, it automatically converts the data when reading and writing the file. This step ensures that the file remains compatible, and indexes remain consistent, with existing DOS applications, even after being updated by Visual Objects.

There are two minor considerations: untranslatable national letters and special characters. ANSI is a larger character set and includes some national letters that do not have representations in the OEM character set. Thus, a user running a Visual Objects application might enter the name Éloise and, after storing it in an OEM file, get it back as Eloise. This is an unavoidable consequence of the difference between DOS and Windows: if the accented letter were stored in the file, the DOS programs would not understand it.

Interoperability

The interoperability of two applications depends on the locking schemes defined in the individual RDDs, and compatibility is not always achievable. The following summarizes the types of applications with which your Visual Objects application can safely share data:

RDD	Product	Interoperable
DBFCDX	FoxPro	Yes
DBFMDX	dBASE IV	Yes
DBFNTX	CA-Clipper	Yes

Note: When using DBFCDX, the .FPT file will remain compatible unless you change the block size, using SET MEMOBLOCK or RDDInfo(_SET_MEMOBLOCKSIZE), to a value that is less than 32.

Of course, any sharing of data requires that both applications are well-behaved, opening files in shared mode to allow simultaneous data access and placing appropriate locks when necessary. (See Chapter 9 "Concurrency Control" for more information.)

Caution! *Unless shared access between applications is explicitly supported as indicated in the table above, database integrity is not guaranteed and index corruption will occur if two programs attempt to write to a database or index file at the same time. For this reason, sharing data between such applications is strongly discouraged and not supported by Computer Associates.*

Chapter

9

Concurrency Control

When developing applications to run under Windows, you should always take into account the issue of simultaneous, or *concurrent*, data access. Not only is it possible for several users to run the same application (and therefore access the same data), but even on a single-user workstation the user may reasonably expect to switch back and forth between applications that use the same data. Thus, it is important that all applications cooperate with one another and share the data resources to the greatest degree possible.

This chapter tells you how to access database files in shared and exclusive mode, how to obtain and release locks in shared mode, and how to resolve locking and file open failures. For more specific reference material on any of the commands, methods, and functions mentioned in this chapter, see the online help system.

Using Shared Mode

The first rule in writing an application that will allow concurrent data access is to *open database files in shared mode unless exclusive mode is required*.

You can use either of these strategies:

- Change the status of the global SetExclusive() flag from TRUE to FALSE in your Start() routine. This will cause database files to be opened in shared mode by default. It also means that you must specify exclusive use of database files when you require it.
- Specify shared access when you open a database file that requires it.

These rules apply whether you are using the DBServer class, Xbase commands, or DB or VODB functions.

Note: The low-level file open functions, FOpen() and FXOpen(), allow you to specify the open mode directly and are unaffected by SetExclusive(). Refer to the online help system for more information on these functions.

When to Obtain Exclusive Use

Certain operations require exclusive use of a database file to function properly. Visual Objects enforces this requirement with an error message (or a return value of FALSE for the VODB functions) if you attempt to use any of the following operations with a database file opened in shared mode:

- Pack (Pack() method, DBPack() and VODBPack() functions, or PACK command)
- Zap (Zap() method, DBZap() and VODBZap() functions, or ZAP command)

Other File Open Operations

In addition to opening database and index files directly, several operations open one or more files in the course of operation and determine the open mode (either exclusive or shared) automatically. As a developer you do not have control over this, but knowing it will help you make the proper allowance in your applications.

There are two general rules that will help you to decide how a given operation works: if it writes to the file, the open mode is exclusive; if it only reads the file, the open mode is shared. For example, the Update() method attempts to open the secondary file in shared mode. If another process has exclusive use of that file, Update() will not be able to open it. Therefore, your programs must anticipate simultaneous updates.

Retrying After an Open Failure

When shared access to data is allowed, the failure of a file open operation becomes a normal possibility, and NetErr() is designed to detect and report open failures as well as certain other concurrency conflicts (see NetErr() in the online help system for details).

You should always check NetErr() immediately after any file open operation that is likely to fail. In this example, the program retries several times before reporting that the file is unavailable and escalating the exception:

```
// Procedural approach
LOCAL iOpenCount AS INT
FOR iOpenCount := 1 UPTO 10
     USE customer ALIAS DBCust SHARED
     IF !NetErr()
          EXIT
     ELSE
          LOOP
     ENDIF
NEXT
IF NetErr()
     ? "File not available in shared mode."
     BREAK
ENDIF
DBCust->DBSetIndex("custno")
DBCust->DBSetIndex("custname")
```

When using a data server, NetErr() will also tell you whether the file open operation was successful, but there is another approach that you may find more flexible—checking the data server's Status flag. This flag will be NULL_OBJECT if the file open operation is successful; otherwise, it will have information about what went wrong, including an error message:

```
// Object-oriented approach
LOCAL oDBCust AS DBServer
LOCAL iOpenCount AS INT
FOR iOpenCount := 1 UPTO 10
     oDBCust := DBServer{"customer"}
     IF oDBCust:Status == NULL_OBJECT
          EXIT
     ELSE
          LOOP
     ENDIF
NEXT
IF oDBCust:Status ◇ NULL_OBJECT          // File open failed
     ? oDBCust:Status:Description
     BREAK
ENDIF
oDBCust:SetIndex("custno")
oDBCust:SetIndex("custname")
```

Note: Your application must always anticipate that the file may not be available, regardless of the open mode. When exclusive mode is requested, another process accessing the file (in either shared or exclusive mode) would make the file unavailable. When shared mode is requested, another process with exclusive use would make the file unavailable.

Locking

When a database file is open in shared mode, you must obtain a lock before performing any update operation (for example Delete, Recall, Replace) on the data; otherwise, the operation will result in an error.

There are two levels of locking: record and file. The one you use will depend on the operation required. If you are updating one record at a time, a record lock will be sufficient; however, if you are performing a mass update, you will need to lock either the entire file or all of the records involved in the operation. (For more specific information on the actual locking schema used, refer to the "RDD Specifics" appendix in this guide and the NewDBLock() and NewIndexLock() functions in the online help system.)

Tip: The DataWindow class provides several levels of automatic locking via its ConcurrencyControl property. See Chapter 11, "GUI Classes" in this guide for more information.

Note: Although it is not required, you may obtain a file lock before performing a read-only operation, such as a report, if your application requires the data to remain unchanged throughout the operation.

File Locking

The FLock() method and the FLock() and VODBFLock() functions attempt to place a file lock on the current database and return a logical value indicating the success or failure of the lock, but they only try once. Because of the possibility of a failure (a file lock will fail if another process has a file or record lock for the same file), you will probably want to make several attempts before admitting defeat.

For example, assuming the Customer class inherits from DBServer, the following method will try to lock the Customer database a specified number of times:

```
// Object-oriented approach
METHOD FileLock(nTimes) CLASS Customer
     // Default to two tries
     nTimes := IF(nTimes == NIL, 2, nTimes)
     // Keep trying until our time's up
     FOR iCount := 1 UPTO nTimes
          IF SELF:FLock()
               RETURN TRUE              // Locked File
          ENDIF
     NEXT
     RETURN FALSE                       // Not locked
```

A similar function which you could call instead of FLock() is illustrated below:

```
// Procedural approach
FUNCTION FileLock(nTimes) AS LOGIC PASCAL
     // Default to two tries
     nTimes := IF(nTimes == NIL, 2, nTimes)

     // Keep trying until our time's up
     FOR iCount := 1 UPTO nTimes
          IF DBCust->FLock()
               RETURN TRUE              // Locked File
          ENDIF
     NEXT

     RETURN FALSE                       // Not locked
```

Record Locking

Record locking is similar to file locking and is accomplished with the RLock() method and the RLock(), DBRLock(), and VODBRLock() functions. All of these make a single attempt to place a record lock on the specified or current record and return a logical value indicating success or failure. Except with the RLock() function which is designed for single record locks only, specifying a particular record as an argument adds to the current record *lock list*, enabling you to lock multiple records at the same time. A record lock will fail if another process has a file lock for the same file or a record lock for the same record.

Using code similar to the examples given above for file locking, you will probably want to create a new record locking method or function that makes several attempts or tries for a certain time span before giving up. This way, you will reduce the probability that the record lock will fail.

Unlocking

Once a lock is in place, an application can write to the file. Other users' attempts to lock the same record or file will fail, but they will still have read access to the data.

The lock remains in place until the application releases it by:

- Explicitly releasing the lock (for example, with the Unlock() method or DBUnlock() function)
- Closing the locked file
- Issuing another lock for the same file
- Terminating the program normally

Because efficient applications seek to minimize the duration of the locks they impose, you should release locks as soon as possible after they have served their purpose.

Resolving a Failure

The code for opening files in shared mode and attempting locks is straightforward and, as suggested already, you may want to try these operations several times before giving up. If, however, a file cannot be opened or a lock cannot be obtained within a reasonable amount of time, the application has to abandon its original intentions and come up with an alternative plan.

You will usually want to give the user a message indicating what has happened along with some options, such as:

- Retry the operation

  This choice simply allows the user to extend the lock or open effort, without seeking to resolve the failure.

- Try the same activity, but with different data

  This choice makes sense for lock failures in many applications. If the target record is not available, use another instead. For example, a data entry operator working on a stack of credit adjustment slips will not mind putting Smith's slip at the bottom of the pile and moving on to Jones or Brown. When the operator gets back to Smith an hour later, that record will probably be free.

- Abandon the current activity

  The situation in which a needed data resource is temporarily unavailable is common, and it is usually treated as an exception when the user chooses to abandon the activity in progress. For a general discussion on how to deal with exceptions, see Chapter 14, "Error and Exception Handling" in this guide.

Update Visibility

When concurrency control is a consideration, it is important to determine when database (and index) updates actually become visible, which differs depending on the *observer*. This section describes the update visibility rules for each of the following possible observers:

- The initiator
- The operating system and other processes
- The physical disk

Note: The rules specified below are for the RDDs supplied with Visual Objects (see Appendix A "Reserved Words" for a complete list). Rules for other drivers may differ.

The Initiator

The *initiator* of an operation is the process that causes an update to occur. All updates appear to the initiator immediately after they are made.

The Operating System and Other Processes

Updates are guaranteed to be visible to other processes when the initiator writes the update to the operating system (although they may become visible sooner). Writing to the operating system can be done in several ways, including releasing a lock or moving the record pointer when a record lock is in place.

The Physical Disk

Writing an update to the operating system does not guarantee that a physical disk write will take place, because the operating system may hold recently written records in memory. These records appear to other processes as if they were on the disk, but if a failure occurs (such as a hardware problem), the records may never be physically written to disk. In this case, updates are lost, and processes to which the updates were visible may be proceeding with erroneous data.

To ensure that updates are written to disk, commit the changes (for example, with the Commit() method or DBCommit() function) or close the database file (which automatically commits the changes). Any of these actions sends a request to the operating system to perform a solid-disk write immediately.

Caution! *Even if you commit changes to disk, there is no guarantee that a physical disk write will actually occur. Some cache programs and network server software will intercept the commit request and postpone the physical disk write. This is a violation of DOS protocol unless the underlying system can guarantee that its method of committing updates is as reliable as writing them to the disk controller hardware.*

Abnormal Termination

When an application terminates normally, all files are closed and all updates are committed to disk.

If the application terminates abnormally, some updates may be lost. The missing updates may potentially include every update to a file since updates to that file were last made visible to the operating system.

If the operating system fails during an application, missing updates may potentially include every update to a file since the last point at which updates to that file were committed to disk.

Chapter

10 Justifying User Interface Choices

In deciding how to program the user interface for your application, Visual Objects provides two choices: the terminal emulation layer and the GUI Classes library. The terminal emulation layer (or terminal emulator) implements the Xbase commands and functions for user interface programming, while the GUI classes provide the object-oriented components for GUI programming.

Why are there two separate paradigms? Why not merge them? How do you choose between them? This chapter attempts to answer these questions by providing an overview of both paradigms and will hopefully help you make the right decisions for your own situation.

Note: In Visual Objects, there is a class library, Console Classes, that provides 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. One major difference between the two types of applications is that a Terminal Lite application will get a new window allocated while a Console Classes application runs in the same window in which it was started. Depending on the command line setting, a Console Classes application can also run in a full screen, while a Terminal Lite application cannot.

The Terminal Emulation Layer

The terminal emulation layer provides a limited subset of the traditional set of Xbase user interface commands and functions and displays their output in a special window called the *terminal emulation window*. The purpose of this layer is to enable existing Xbase applications to run under Windows.

It would seem an easy matter to bind these macros to the Visual Objects Window Editor, enabling existing code to map onto nicely designed forms. However, this approach could only partially simulate the Windows look and feel. Certain inherent behaviors would remain in effect, leaving the user frustrated with an application that, although visually similar, was not quite as good as other Windows applications.

Data Entry Order

The first problem is that in the Xbase code, the order of data entry is predetermined: it is driven by the order in which the @...GETs are programmed. Field-level validation and cross-field validation must occur in a precise order.

Natural Windows behavior, on the other hand, does not have this restriction. In general, the user can enter data in any order. Field-level validation occurs as soon as a field loses edit focus. Cross-field validation is done when the record is ready to be committed to the database. In addition, the application can override this behavior, so that the user has the ability to enter data several times in one field and cross-field validation would be attempted on every focus change. In summary, the behavior of a Windows application is a lot richer than that of a typical Xbase application, even within the confines of a single form.

Modal Behavior

The second problem is that Xbase code is modal. That is, the user cannot open another form in the same application when they are halfway through dealing with this form. The ability to look something up or to make a correction while one is entering data is very valuable behavior, specific to GUIs.

Because it would not help you, or your clients, to supply applications that had Windows look and feel while exhibiting Xbase-like modal behavior, the terminal emulation layer gives existing applications a familiar look and feel. When your users see this look and feel, they know they are back to the old mode in which the program is in control. When they use a different application, they get the modeless behavior and the Windows look and feel and know that they are in control.

GUI Classes

World-class applications require fully modeless GUI behavior, in which the user can do most anything at any time and the application will like it. Why is it necessary to use classes to obtain modeless behavior? Would it not be possible to do something with @...SAY...GET commands to make them modeless?

To get completely modeless, MDI-style behavior, the user should be able to open a data window. Then open another data window on the same database. Then open a few more data windows. Then chop and change, and cut and paste between them. To enable the style of programming that can withstand this behavior, you need an object-oriented approach. Each data window is an object. You can make an arbitrary number of them at any time. Each of them is self-contained and does not interfere with the other ones (except when you explicitly program them to interact).

It would be impossible to achieve this behavior without encapsulation and inheritance. Any approach that attempts to reuse Xbase user interface code in a modeless environment would be a blind alley. Also, using the object-oriented syntax allows for some significant improvements to the Xbase functionality, such as status messages and context-sensitive help.

The GUI Classes Compared to Windows API

Given all the reasons that compel an object-oriented approach, why the GUI classes? One could argue for a thin layer of classes that encapsulate the Windows Application Programming Interface (API).

In fact, the GUI classes do provide a very thin layer over the API. They are highly optimized, so that CPU and memory requirements are minimal. So, why does the GUI Classes library not resemble the API? Because, for the vast majority of Visual Objects developers, the GUI Classes library represents a much better solution.

Here are a few reasons why:

1. The GUI Classes library is small and fast, reducing the size and, therefore, increasing the speed of your applications.
2. It is very easy to learn—about a hundred classes. The Windows API has many, many more functions.
3. It provides a lot of very useful default behavior. If you write no code at all, the system will still generate a standard application.
4. It cooperates with all the code generation tools in the IDE, such as the Window Editor and data server editors.
5. It supplies data-aware classes.
6. It contains error detection and error correction code that will save you from the many pitfalls of Windows programming.
7. The architecture lends itself to useful extensions.
8. It provides a role model that you can safely use to develop your own class libraries.
9. The Visual Objects runtime system does not yet run on other GUIs, but the GUI classes ensure that any application code you write will have a good chance of being portable to other GUIs in the future. The Windows API, by comparison, would impose an awkward, non-portable structure.

The Right Choice

The terminal emulation layer exists as a migration path to permit existing Xbase applications to run under Windows; however, it cannot fulfill the Windows user's reasonable expectations. The GUI Classes library enables you to build world-class applications that meet or surpass users' expectations by providing a simple, successful framework on which to construct your application.

Chapter

11

GUI Classes

GUI classes account for the majority of classes supplied with Visual Objects. Very roughly, the GUI Classes library includes one class for every kind of object that you encounter in a GUI. There are Window and Dialog classes. There are Control classes, such as ScrollBar and PushButton. There are Menus and Toolbars. This chapter describes the standard components of Visual Objects that deal with the GUI.

Events, Event Contexts, and Event Handlers

A GUI application is made up of different types of objects (for example, windows, menus, toolbars, scroll bars, and controls). These objects interact with each other via *events*. In general, the events that you are concerned with as a developer are the ones that are generated when the user does something. For example, a mouse click generates an event and so does a request for help made by pressing F1.

Events are generated and handled as follows:

1. Something happens.
2. Windows sends a *message* to the active window saying what happened.
3. The Visual Objects dispatcher intercepts this message, converts the message into an event that it sends to a special method, called an *event handler*, designed to handle that particular type of event.
4. The event handler method does some special processing to deal with the event.

Events

In Visual Objects, events are objects, too. There are classes defined for the various types of events that commonly occur in a GUI application. The event classes, as expected, are tailored to the specific event for which they are designed. These classes derive from the Event class and have "Event" tacked onto the end of their name.

Event Contexts

Classes that contain event handlers are called *event contexts* because they represent a context in the application in which events can occur. All variants of a window are event contexts. This is reflected in the Window class hierarchy—classes with "Window" tacked to the end of their name derive ultimately from the EventContext class.

Event Handlers

The event class names and the event handler names are closely related. For example, a user's request for help generates a HelpRequestEvent that is passed to the event handler Window:HelpRequest(). Similarly, when the user presses or releases a key, a KeyEvent is generated; it is passed to Window:KeyDown() or Window:KeyUp(), depending on the action. Some event handlers, such as DataWindow:Notify(), can handle several different events, such as NotifyDelete and NotifyIntentToMove, depending on an argument passed to the event handler.

Many of the default event handlers have built-in behavior that is sufficient for most applications; however, it is not possible for the default event handlers to handle every event in the exact way that you want. For this reason, event handlers, like all other methods, can be overridden in your Window subclass if they do not suit your specific needs.

Command Events

For example, it is not at all unreasonable to expect an application to supply customized event handlers to deal with push button clicks, menu command selections, and toolbar button clicks (which are just a special case of menu command selections). The default event handlers for these *command events*, Window:ButtonClick() and Window:MenuCommand(), cannot possibly know the design of your window or menu in order to implement these event handlers for you; instead, they assume you are going to implement your own methods to handle these events individually.

Event Processing by Name

To make processing them easy and straightforward, command events are handled based on the *symbolic name* of the push button or menu item generating the event. For example, you might create a push button and give it the symbolic name PostOrder. Then whenever the user clicks that push button, Window:ButtonClick() retrieves the symbolic name of the push button and looks for a method of that same name (in this case, it looks for the PostOrder() method). The Window and Menu Editors are set up to support this structure and generate the code it requires.

Generated Code

When you define a window or menu using the appropriate visual editor, Visual Objects generates code to assign a HyperLabel object to each push button or menu item (other objects generated by the editors also have HyperLabel objects associated with them). The HyperLabel object holds, among other things, the symbolic name used for command event processing.

For example, the code generated to place the PostOrder push button on a window named InformationWindow would look something like this (slightly simplified):

```
STATIC DEFINE INFORMATIONWINDOW_POSTORDER := 100

RESOURCE InformationWindow DIALOG 4, 19, 174, 101
STYLEDS_MODALFRAME | WS_POPUP |;
     WS_CAPTION | WS_SYSMENU
CAPTION "Information Window"
FONT 8, "MS Sans Serif"
BEGIN
     CONTROL "Post Order",
          INFORMATIONWINDOW_POSTORDER,;
               "Button", BS_PUSHBUTTON | WS_TABSTOP;
                    | WS_CHILD, 69, 16, 35, 12
END

CLASS InformationWindow INHERIT DialogWindow
     PROTECT oCCPostOrder

METHOD Init(oWindow) CLASS InformationWindow
     SUPER:Init(oWindow,;
          ResourceID{"InformationWindow"})
     oCCPostOrder := PushButton{SELF,;
          ResourceID{INFORMATIONWINDOW_POSTORDER}}
     oCCPostOrder:HyperLabel := ;
          HyperLabel{#PostOrder,;
               "Post Order", "Post the order transaction"}
     ...
```

Note: For more examples of this, use the various editors and look at the generated code.

Resource IDs

Here, INFORMATIONWINDOW_POSTORDER is a constant that holds the *resource ID* (a 16-bit number) for the PostOrder push button. This ID is used to uniquely identify the push button control to Windows (in the CONTROL statement of the resource entity) and to tie that object to the corresponding object in your application. You can see this in the InformationWindow:Init() method, where the ID is used to create a ResourceID object for the PostOrder push button.

Resources

The code for the resource entity creates a standard Windows *resource* to define what the window or menu looks like. Within this definition, Windows requires the resource ID mentioned above for each control and menu item (and every other object) that appears on a window or menu.

Symbolic Names

After the PostOrder push button is instantiated, a hyperlabel is created for it. The hyperlabel contains the symbolic name #PostOrder (which is used to control program operation), the text that goes on the face of the button ("Post Order"), and the prompt string that shows up in the status bar ("Post the order transaction").

Note: The hash mark character (#) identifies #PostOrder as a special data type called SYMBOL. The SYMBOL data type provides an internal representation that handles strings very efficiently. See Chapter 21, "Data Types," later in this guide, for more information on this and other data types available in the Visual Objects language.

Windows does not know about symbolic names; however, since each control and menu item in your application knows about its Windows resource ID and its symbolic name, your application can use symbolic names only, ignoring the Windows resource IDs altogether.

Event Handlers

Given such a structure, all you have to do is write the PostOrder() method to handle the event, which you can also do from the Window Editor using the When Clicked property. This brings up the Source Code Editor with the following line of generated code:

```
METHOD PostOrder() CLASS InformationWindow
```

After you write the code for the event handler, there is nothing more to do. Once the window is instantiated, clicking on the PostOrder push button will automatically invoke the PostOrder() method.

With this structure, the Window and Menu Editors and the GUI Classes library provide the overall framework for the control flow of the application. Building an application consists of visually defining the windows and menus (and their associated toolbars), and writing the application logic in the event handling methods.

The standard Window classes provide built-in methods for many of the basic operations, such as destroying and hiding windows. The DataWindow class (discussed later in the Data Windows section of this chapter) provides a richer set of methods designed specifically for data-oriented operations, such as SkipNext(), SkipPrevious(), Append(), and Delete().

To use a built-in event handling method, either assign its name directly to the push button or menu item, or call the method from within your own event handling method using, for example, SELF:EndDialog() or SELF:SkipNext().

The Window Handles Events

When the Visual Objects dispatcher receives messages from Windows, it translates them into events and sends them to the appropriate window. That is, events are not sent to the control or menu that generated the event, but rather to the window that owns the control or menu. (Remember that the Window class inherits from EventContext, but Control and Menu do not; therefore, windows can handle events, but controls and menus cannot.) This architecture has several advantages. Since the event handlers are methods of the window, they can directly reference the data held by the window. If the event handlers were methods of the controls, scoping conflicts would arise: the action code would have to reference the window's data indirectly.

This means that when designing a menu, the individual menu items do not have code attached to them, they only have the name of the event handler to call. When a menu is attached to a window, the link is complete: selecting a menu command sends the corresponding event to the window, causing the corresponding event handler method to be called.

Tip: All windows have a Menu property to identify the menu owned by the window. This makes it easy to call methods of the menu from your event handler methods, as in Menu:CheckItem() or Menu:EnableItem(). Similarly, data windows have a Server property that makes calling methods of the server equally convenient.

Isomorphism:
When different types of objects respond to the same message

This provides for a very useful form of *isomorphism* (a term that is described in greater detail in Chapter 25, "Objects, Classes, and Methods" later in this guide). Consider the Print command on the File menu. Different types of windows can have different code for printing their data: printing a customer record is not the same as printing the orders for that customer. However, the menu does not need to know about this difference: it just sends the Print message, and the appropriate Print() method of each particular window is invoked to do the right thing.

Escalation of Events

On the other hand, it creates the possibility of missing event handlers: if a menu has an item for which no event handler is provided in the window, the menu command will not work.

Consider the Open command on the File menu. The standard interpretation of Open in an MDI application is that opening a new document is equivalent to opening a new window. Keep in mind that in an MDI application, the shell owns the child windows. It would not make sense, therefore, for a child window to handle the Open command, because you do not open the new document in the child window—you open it as a *new* child window in the shell. The Open command should be handled by the shell.

This means that the child windows—for example, the customer window, the order window, the item list window, the credit history window—do not have Open() methods to handle the Open command. So, what happens when the user is sitting on a customer window and selects File Open?

First, when the menu command event is generated, the message goes to the default Window:MenuCommand() event handler, which retrieves the symbolic name of the menu item, Open. Then, the Open message goes to the current window, which is the customer window in this case. Because there is no Open() method, the message is simply passed up to the customer window's owner, which is the shell window. Since the shell window does have an Open() method, it is invoked. In some cases the ownership hierarchy may be longer, but regardless of the structure, an unsupported message is always passed up the ownership chain until something can handle it.

Eventually, if the message reaches the top of the application (the shell window or the top window) and no window had a method that matched, the shell or top window checks if there is a subclass of Window or ReportQueue with a matching name. If there is, it is instantiated with the shell or top window as its owner. Thus, a simple way to open a child window or print a report in an MDI application is to define a menu item with the same name.

Overriding Behavior

Of course, you can write a new MenuCommand() or ButtonClick() method for your Window subclass. Using this approach, you can intercept the event before it goes to the corresponding default event handler and apply processing logic in some other way. In particular, you can write a CASE construct that tests for the various resource IDs that you want to process. You can either supplement the default behavior by calling the same method in the superclass or override it completely.

This example uses the default behavior for all but the File menu commands:

```
METHOD MenuCommand(oMCE) CLASS CustomerWindow
     LOCAL nItemID := oMCE:ItemID

     DO CASE
     CASE nItemID = IDM_MYMENU_FILE_OPEN_ID
          // Process File Open command

     CASE nItemID = IDM_MYMENU_FILE_CLOSE_ID
          // Process File Close command

     CASE nItemID = IDM_MYMENU_FILE_CLOSE_ALL_ID
          // Process File Close All command

          CASE nItemID = IDM_MYMENU_FILE_EXIT_ID
          // Process File Exit command

     OTHERWISE
          SUPER:MenuCommand(oMCE)
     ENDCASE
```

This approach is less convenient and arguably less object-oriented than the matching of methods to symbolic names, but you may prefer it if you are used to this style of programming.

Eventually, if nobody handles the event—if no window has a matching method, if there is no Window or ReportQueue subclass that matches, if no customized MenuCommand() or ButtonClick() method did anything—the event quietly fails. This allows you to do top-down design of the user interface during prototyping: menu items and buttons that have no implementation are ignored and do not raise an error condition of any kind.

The Shell and the Windows It Owns

Thus, one of the main functions of a window in a GUI application is to handle events that are generated when the window has focus. As a developer, you can choose to provide explicit code to handle each type of event or you can leave it to the default behavior of the GUI classes. In addition to processing events, Windows serve two other purposes:

The Canvas for Viewing

First and foremost, the window acts as a viewer for data, text, or graphics. To present its information to the user, it needs the concept of a *canvas* on which the program paints the data or the words. In most GUIs, the canvas area must be rectangular in shape. Thus, a window is just a large rectangle, and it has methods for drawing lines or printing text.

Controls for Initiating Actions

To enable the user to perform actions on the window, the window needs an assortment of *controls*, such as a minimize button or a status bar. Just as the controls for driving a car are on the dashboard, the controls for driving a window occupy convenient locations around the window. The program also manipulates these controls. For example, it can disable controls when their use would be inappropriate for the moment or hide them when they are not relevant.

Programming the User Interface

Programming the user interface starts with the *shell* window (derived from the ShellWindow class), which manages all the other windows in an application. The shell window is the first window that the application constructs.

There is a well-established convention that an application constructs just one shell window, during App:Start(), and that shell is then owned by the App object. The Standard Application generated by Visual Objects follows this convention and is an excellent example of how to structure an MDI application (see "Learning the Basics" in the *Getting Started* guide for more information on the Standard Application). If you abide by the convention, for example by using the Standard Application as your starting point, you will achieve behavior that is typical of many commercial Windows applications.

Top Window

The simplest possible application uses a top window, derived from the TopAppWindow class instead of the ShellWindow class. Like a shell window, a top window is owned by the App object and can own further windows of its own, but the similarity ends there. Top windows present the menu for a whole application, supporting the SDI in Windows—they do not have the automatic arrangement, status bar, and menu management that a shell window does.

The canvas of a top window has all the capabilities of the canvas of a child window. This means that in the simplest of all applications the top window can perform the presentation functions of a child window, which serves well enough in situations where the window is required to present only one view at a time.

Note: For information and a broader understanding of the various types of windows you may want to consult the Microsoft Windows style guide, *The Windows Interface: An Application Design Guide.*

In the Microsoft text, top windows correspond to "application windows," child windows controlled by a shell window correspond to "document windows," and dialog windows correspond to "dialog boxes."

The Shell as Owner

In the conventional style, a single shell window is the ultimate owner of all the active elements of the application. Child windows constructed by the shell are clipped to lie within its canvas. The shell represents the physical boundary of the application on the Windows desktop, and all the activity of the application takes place within the confines of the shell's canvas.

Besides its own canvas, title bar, and border, the shell window conventionally displays a menu, a status bar, and a toolbar. Initially, the menu and status bar refer to the shell itself, but once the shell has constructed further windows, it presents the menu for whichever of its owned windows currently has focus and displays messages from that window on its status bar. Each child window, however, displays its own toolbar.

The shell constructs further child windows that it owns and manages. When the user iconizes a child window, its icon appears at the bottom of the shell's canvas. The ShellWindow:Arrange() method arranges all child windows based on an argument defining the arrangement. For example, the default shell window of the Standard Application has Window Tile and Cascade menu commands that call this method. The ShellWindow class handles all this window management for you.

Customizing the Shell

Visual Objects permits you to place controls on the canvas of the shell. However, this is contrary to convention. By convention, you place controls only on the canvas of a data window or dialog window.

One exception to this is that it is not unusual to put *window scroll bars* on the shell. These are special scroll bars that snap to the right side and bottom of the window. Their conventional significance is to indicate that the window is a partial view of a larger area, and the scroll bars move the partial view to traverse the larger area. The GUI classes do not interpret them in any way. If you have window scroll bars on the shell, you must handle the scroll events that occur by writing explicit code.

When you want to change the behavior of the shell window, it is best to subclass it. In your subclass, you can hide or even remove the status bar. You can remove either (or both) the menu and toolbar. You can change the border style, background and foreground colors, and many other properties. If you look at the list of assign methods for ShellWindow and its ancestors using the Class Browser, you will get a good idea of the variety of things you can change.

Window Relationships

Once you have established your shell window, there are four kinds of windows that it can own.

Child Windows

Child windows, which inherit from the ChildAppWindow class, have practically the same functionality as shell windows except they are constrained to their shell window's canvas. Generally, you would use a child to present graphical views, such as block diagrams or charts. Child windows are commonly used for drawing objects, such as EllipseObject and BitmapObject (see "Other Features of the GUI Classes" in this guide for additional details on this subject).

Data Windows

Another common variety of window is the data window (derived from the DataWindow class). Essentially, they model business forms, and they interact automatically with data servers. Data windows are discussed in more detail in the Data Windows section later in this chapter.

Dialog Windows

If you wish to display or capture information using a formal layout without storing the information in a database, you can use a dialog window (derived from the DialogWindow class). Dialog windows are usually displayed in response to menu command or push button events. Their purpose is to gather additional information needed to process the event.

You create the layout for a dialog window using the Window Editor, just as you would do for a data window, but without making the connection to the database. The Window Editor generates all the necessary code, including a resource entity, resource IDs for the dialog window and all its controls, and an Init() method that creates the necessary control objects and hyperlabels for the controls and the dialog window itself. A good example of this can be found in the code generated for the HelpAbout dialog window in the Standard Application.

In your program, you can connect the dialog window to a push button or menu item of the same name. This is what happens in the Standard Application with HelpAbout. It is linked by name to the Help About menu command. Alternatively, you can instantiate the dialog window and display it directly:

```
HelpAbout{SELF}:Show()
```

Either way, the dialog will appear with the layout that you designed using the Window Editor.

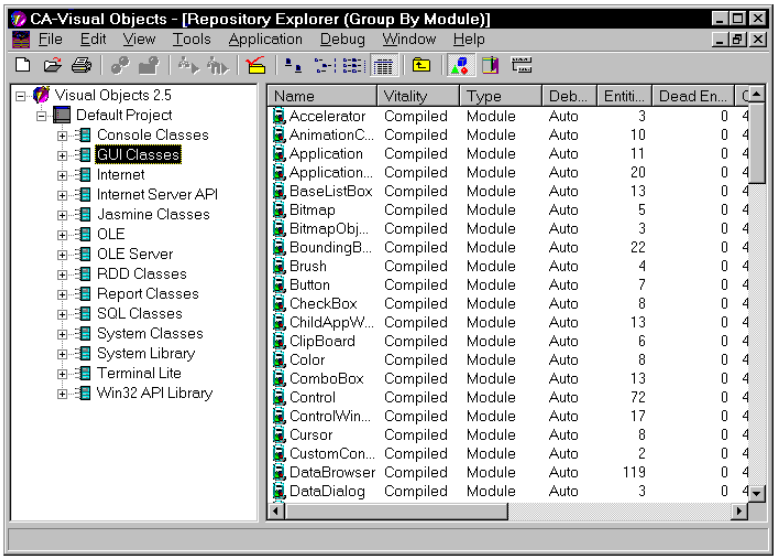
DataDialog Windows

DataDialog windows are dialog windows that can have a data server attached. DataDialog windows are modal Dialogs and unlike normal DataWindows, are not MDI child windows. The behavior for the Modeless DataDialog windows is almost identical to that of regular DataWindows.

DataDialogs expect the underlying window to be an AppWindow and not a Dialog. Therefore, the EnableVerticalScroll and EnableHorizontalScroll methods do nothing for DataDialogs. Assigning a menu causes the menu to be displayed at the top of the Dialog. This is a direct consequence of not being an MDI child.

Controls

As stated earlier, you place controls, by convention, only on the canvas of a data window or dialog window. Detailed interaction between the window and the user takes place via these controls. The most common controls are edit controls and push buttons, but the GUI Classes library supports all of the controls that GUIs use. Typically, the class that implements a particular control has the name that is most frequently used to refer to controls of that type. Hence, there are classes called ComboBox, PushButton, ScrollBar, and so on, as shown here:



You normally place these controls on a window using the Window Editor. When they occur on a data window, they are automatically data-aware. That is, the data window associates certain controls with data fields in the data server using the symbolic names you supplied to the Window Editor.

In addition to their data-aware behavior, Visual Objects controls have a lot of other automatic behavior. You can specify the tabbing sequence among controls and an Alt+key sequence used to select the control using the Window Editor. (Both the tabbing sequence among controls and the Alt+key are coded into the window's resource entity and managed by the Windows *dialog manager* at runtime.) When the control gets focus, the prompt from its hyperlabel appears on the shell window's status bar. Controls have virtual variables, named Value and TextValue, that enable you to assign values directly to and from the control.

Menus

A *menu* is a visible object that enables the user to activate parts of an application. It is the means of navigation in a GUI application.

Visual Objects implements a number of different navigation mechanisms as menus. Whether the user chooses a menu item with the mouse, uses an accelerator key or Alt+key combination, or clicks a button on the toolbar, the only interest for the developer is to know which item the user selected. Visual Objects hides from the application the route by which the user arrived at the choice.

You construct the physical appearance of menus using the Menu Editor. You lay out the hierarchy of the menu, specify icons, accelerator keys, Alt+key combinations, and toolbar buttons, and specify the status messages to appear by default when the user probes each menu item. The Menu Editor generates code that produces the menu, along with its corresponding accelerator table and toolbar. When the menu is instantiated, it is given an owner (a window); it assigns itself, and the accelerator table and toolbar, if any, to the window.

Menu selection event handling has already been discussed in detail in this chapter, but there are two other menu events that may prove of interest to your program. The MenuSelect event occurs when the user moves over an item without choosing it. The default implementation of the Window:MenuSelect() event handler posts the appropriate message to the shell window's status bar. The other menu event is Window:MenuInit(), which occurs just before a menu is to drop down. You only need this one if you create menu items on the fly rather than predefining them using the Menu Editor.

Under most circumstances, the Menu Editor alone gives you sufficient control over the behavior of a menu and its associated toolbar.

Standard Dialogs

A *standard dialog* is a dialog that the GUI provides. Its look and feel are common across all applications.

Many menu items result in a standard action, such as opening a file. Visual Objects provides many of the standard dialogs supported by Windows, such as Printer Setup and Save As. Through the OpenDialog class, it extends the file open dialog to support opening index files together with DBF files and to enable the user to select the appropriate RDD.

Typical code to open a file would use the OpenDialog class, as illustrated in the example below:

```
METHOD FileOpen() CLASS MyShellWindow
     LOCAL oOD AS OpenDialog
     LOCAL oDB AS DBServer
     LOCAL oDW AS DataWindow
     // Display File Open dialog
     oOD := OpenDialog{SELF, "*.dbf"}
     oOD:Show()
          // If user did not Cancel...
          IF !Empty(oOD:FileName)
          // Open DBF file in a server
          oDB := DBServer{oOD:FileName}
          // If opened successfully...
          IF oDB:Status = NULL_OBJECT
               // Create a data window
               oDW := DataWindow{SELF}
               // Assign data server to window
               oDW:Use(oDB)
               oDW:Caption := oOD:FileName
               oDW:Show()
          ENDIF
     ENDIF
```

Data Windows

A data window is a special kind of window, in four respects:

- Its visual behavior depends on how it is created.
- It can be displayed in a form view, with individual controls, or in a browse view, as a scrollable spreadsheet-like table.
- It can be nested within another data window, as a sub-data window.
- It can be linked to a data server and provides a number of methods for data management and navigation, as well as other useful features including automatic data validation and propagation.

Different Types of Data Windows

Unlike other windows, a data window can take on many personalities based on the circumstances under which it is instantiated. It automatically adapts itself to the circumstances and provides suitable behavior.

For example, when you construct a data window as a child window, it combines the characteristics of a dialog window and a child window. Like a dialog window, it can be created with a resource defining its controls, and it provides the automatic keyboard interface of a dialog window, but as a whole, the window has the behavior of an MDI child window.

A data window can be used as:

- A child window (typical in MDI applications) or top window (typical in SDI applications) with data controls
- A data-aware dialog window that deals with some special data, outside of the regular flow of the application
- A sub-data window placed on another data window as a custom control

The data window takes on these different personalities depending on its owner and instantiation parameters:

Type of Window	Circumstances
Child window	If the owner is a ShellWindow
Dialog window	If the owner is a TopAppWindow, ChildAppWindow, or DataWindow
Sub-data window	If the owner is a DataWindow and a resource ID is passed as the third instantiation parameter

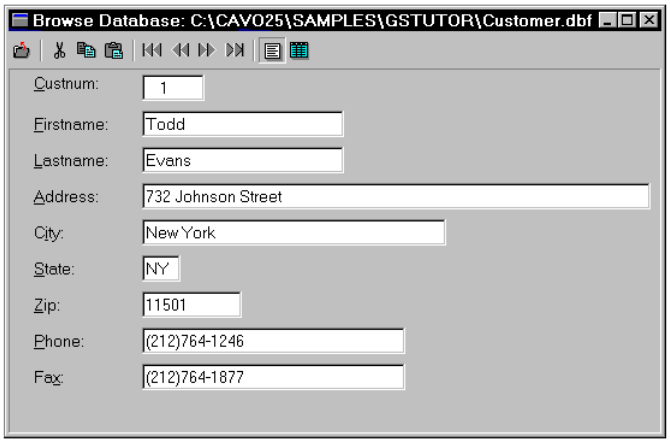
Note: Under standard Windows conventions, a shell window is not used to display data; therefore, a data window is never a shell window.

Form and Browse View

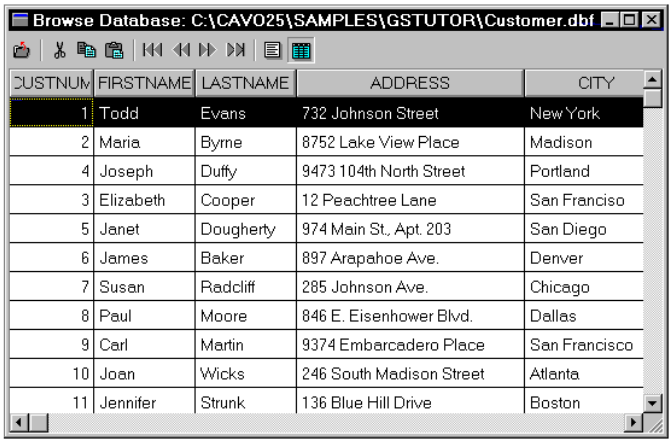
A data window can take on two different view modes, both of which are illustrated below: *form view*, which contains individual controls for the data fields, and *browse view*, which contains a spreadsheet-like *data browser*. The data window can be initially displayed in either mode, and can be switched to the other mode at any time (using the DataWindow:ViewAs() method).

Any data window supports both appearances, although the developer can, of course, choose not to provide a way to select one mode or the other.

Form View



Browse View



CUSTNUM	FIRSTNAME	LASTNAME	ADDRESS	CITY
1	Todd	Evans	732 Johnson Street	New York
2	Maria	Byrne	8752 Lake View Place	Madison
4	Joseph	Duffy	9473 104th North Street	Portland
3	Elizabeth	Cooper	12 Peachtree Lane	San Franciso
5	Janet	Dougherty	974 Main St., Apt. 203	San Diego
6	James	Baker	897 Arapahoe Ave.	Denver
7	Susan	Radcliff	285 Johnson Ave.	Chicago
8	Paul	Moore	846 E. Eisenhower Blvd.	Dallas
9	Carl	Martin	9374 Embarcadero Place	San Francisco
10	Joan	Wicks	246 South Madison Street	Atlanta
11	Jennifer	Strunk	136 Blue Hill Drive	Boston

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

The data window can automatically create a default layout based on the characteristics of the linked data server. The automatic self-configuring behavior is useful when the application is still under development or when the data window needs to look at databases it has not seen before, such as in the Standard Program.

Resource-Driven Instantiation

Just like a dialog window, a data window is normally created based on a resource, identifying the size and location of all the controls on the data window and the data window as a whole. Like all windows, a data window has an owner.

You can construct and activate a data window with statements like this:

```
oDW := DataWindow{SELF,ResourceID{"CustomerWindow"}}
oDW:Show()
```

where CustomerWindow is the name of the resource entity defining the window and each of its controls in terms understood by the Windows dialog manager.

Just as in a dialog window, the controls of a data window are created as objects, each associated with its own resource ID:

```
oDCCustName := SingleLineEdit{SELF, ResourceID{CUSTOMERWINDOW_CUSTNAME}}
```

The code for the CustomerWindow resource entity, which ties the control to the window, might look something like this:

```
RESOURCE CustomerWindow DIALOG  3, 3, 185, 159
     STYLEWS_CHILD
     FONT 8, "MS Sans Serif"
     BEGIN
          CONTROL "", CUSTOMERWINDOW_CUSTNAME, "Edit", ;
               ES_AUTOHSCROLL | ES_LEFT | WS_TABSTOP |;
               WS_CHILD | WS_BORDER, 48, 27, 68, 12
     END
```

Symbolic Names

A data window that is to be linked with a data server requires that each control be given a symbolic name, which is used as the basis for linking controls to data fields (that is, the symbolic name of the control must be the same as its associated data field). You assign a symbolic name to the control by giving it a hyperlabel that holds the symbolic name and other annotations, such as a caption, a description, and a help keyword, which are optional. Thus, the complete creation of an object for the control looks like this:

```
oDCCustName := SingleLineEdit{SELF,;
     ResourceID{CUSTOMERWINDOW_CUSTNAME}}
oDCCustName:HyperLabel := HyperLabel{#CustName}
```

Subclassing DataWindow

The most useful way to create a window with controls is to define a subclass of the generic data window and instantiate all the controls in the Init() method. The most convenient way of producing this kind of code, of course, is to use the Window Editor, which generates the code for you. This is the kind of code it generates:

```
CLASS CustomerWindow INHERIT DataWindow
     PROTECT oDCCustName

METHOD Init(oWindow,iCtlID,oServer) CLASS CustomerWindow
     SELF:PreInit()
     SUPER:Init(oWindow, ResourceID{"CustomerWindow",_GetInst()},iCtlID)
     oDCCustName := SingleLineEdit{SELF, ResourceID{CUSTOMERWINDOW_CUSTNAME, ;
_GetInst()}}
     oDCCustName:HyperLabel := HyperLabel{#CustName}
     SELF:Caption := "Custname"
     SELF:HyperLabel := HyperLabel{#CustomerWindow, "Custname"}
     SELF:PostInit()
```

Given this subclass definition, the customer window can be directly created and shown:

```
oCW := CustomerWindow{SELF}
oCW:Show()
```

Pre/PostInit()

The Window Editor generates a SELF:PreInit() call before the actual control creation and a SELF:PostInit() call after the control creation. Default PreInit() and PostInit() methods are defined in the Window class and are available to all data windows. The default PreInit() and PostInit() methods are empty and don't perform any action.

```
Method PreInit() CLASS CustomerWindow
     MessageBox(0, "Before Init", "", 0)

Method PostInit() CLASS CustomerWindow
      MessageBox(0, "After Init", "", 0)
```

By specifying PreInit() and PostInit() methods for a generated data window you can perform action before and after the initialization of the form. The PreInit() and PostInit() methods are not touched by the Window Editor and therefore your code will not be overwritten when regenerating the Window.

Access to Values

There is a dilemma in referencing controls such as the customer name: you need to refer to the control, to disable (gray) it, for example, and you also need to refer to the value held in the control, the customer name itself.

Since the control is held by a variable of the window, you could get its value by referring to *oDCCustName:Value*, but the control variable is protected and, therefore, not available outside the class. You could also use the FieldGet() and FieldPut() methods of the DataWindow class with the field name, but this is not very elegant. In business logic that lies outside the class, you need a way to reference the control's value in a convenient and straightforward manner. You would like to refer directly to *CustName*.

Virtual Variables

The recommended technique to accomplish this (also used in the code generated by the Window Editor) is to have one internal variable with a name prefixed with oDC (for "object of data control type") for referring to the control, and an external variable without a prefix for referring to the value. The value variable is actually never used. It is a *virtual variable* implemented with a pair of ACCESS and ASSIGN methods:

```
CLASS CustomerWindow INHERIT DataWindow
      PROTECT oDCCustName AS SINGLELINEEDIT
      INSTANCE CustName

ACCESS CustName() CLASS CustomerWindow
      RETURN SELF:FieldGet(#CustName)

ASSIGN CustName(uValue) CLASS CustomerWindow
      SELF:FieldPut(#CustName, uValue)
      RETURN CustName := uValue
```

This structure allows you to distinguish between the control and its value inside methods of the class and to refer to the value in a straightforward manner outside the class:

```
// Inside methods of the data window

oDCCustName:Disable()                   // the control
DoSomethingWith(CustName)               // the value

// Outside the data window

DoSomethingWith(oDW:CustName)  // the value
```

Automatic Generation of Virtual Variables

Even if no virtual variables (pairs of access and assign methods) have been created, the data window still permits reference to controls by their symbolic name, in the same way that data servers present their data fields as virtual variables.

Providing explicit definitions of the virtual variable—the access and assign methods—is useful for two reasons: it makes the window self-documenting and it gives you a convenient way to refer to the data field value outside the class.

Dynamic Instantiation

Like a dialog window, it is also possible to create a data window dynamically, without a resource. In this case the window is created with a size and position (relative to its parent window) instead of a resource, and the controls are created dynamically, each with a size and position specified explicitly instead of using a resource ID.

The data window retains all other aspects of its behavior and can be linked to data servers just like a statically created data window. Of course, the controls must still be given the symbolic names used in data linkage. Once the control exists, there is no difference between the two types of instantiation. However, resource-based instantiation is preferred: it is a good way to manage the layout of windows, and it is faster.

Note: Dynamically created controls and statically created, resource-based controls can be mixed on a window.

Automatic Layout

A data window can also create itself automatically, laying out both its form and browse view based on the structure of the data server it is linked with. If the data window has no controls or columns defined, a default layout that matches the fields of the server is defined when you instantiate the window. This might occur because the window is a generic DataWindow class, or because the program switches it to a new view that has not been explicitly defined.

You also have the option of explicitly laying out one view and having the other view generated automatically. For example, you might choose to replace the automatically generated form view layout with one that you define with the Window Editor; you could leave the browse view for this window undefined, and the system would generate one for you.

When using a data window for which no explicit layout has been specified, however, you should be aware that generated views will contain controls for *all* fields in the database file or table, without regard to fields that have been excluded from the data server via the data server editors.

Linking a Data Window to a Data Server

A data window is linked to a data server with the DataWindow:Use() method, which is called with a data server object. Thus, to illustrate the automatic layout capability of a data window, you could instantiate a generic data server and data window, and link the two together as follows:

```
oDB := DBServer{"customer"}
oDW := DataWindow{SELF}
```

```
oDW:Use(oDB)
```

The window would contain a fixed text and single-line edit control for each field in CUSTOMER.DBF.

Otherwise, if both the data server and data window have been defined, linking the two via DataWindow:Use() sets up linkages between the controls on the data window and the fields of the data server based on common name:

```
oCW := CustomerWindow{SELF}
oDB := Customer{}
oCW:Use(oDB)
oCW:Show()
```

Only those controls and data fields that match are linked. Controls that do not find a match remain unlinked: this is very useful, since it allows the handling of non-database related controls. Data server fields that do not find a matching control also remain unlinked. A database may have more fields than are needed in a particular application, and those can be ignored by simply omitting them from the data window layout.

Note: Each data window can be linked to only one server at a time. Establishing a link with the DataWindow:Use() method removes any previously existing link. To create more complex windows that operate with more than one server, use a sub-data window as described in the Sub-Data Windows section later in this chapter.

Display Options

After a data window is linked to a data server, the data window automatically gains access to the field specifications defined for that data server. This happens, again, by matching the symbolic name of the control to that of a data field.

Because they are linked, the control is able to pick up the FieldSpec and other properties of the data field and use them in the data window. For example, a control automatically displays its data using the FieldSpec:Picture property of the data field it is linked to. Similarly, the field specification's HyperLabel:Description property (or that of the data field itself, if the FieldSpec does not define a Description) is automatically displayed in the status bar when the control has focus.

Of course, you can override the field specification by assigning a different one to the control using its FieldSpec property. This is illustrated in the code generated by the Window Editor if you change the FieldSpec property to something other than <Auto>. You can also override the HyperLabel:Description in the same way, by assigning a description directly to the control's hyperlabel.

Validation

The FieldSpec also defines several field-level validation rules, such as a range of values that the data field must fall between, whether or not the data field is required, and a developer-defined validation rule. Each type of validation has its own hyperlabel to provide helpful information to the user in case the validation fails (for example, a diagnostic message for the status bar and a help keyword that you can tie into your online help system).

Field-Level Validation

The data window handles these field-level validations automatically, without any additional programming on your part—all you do is specify the rule and the diagnostic message, and the data window does the rest. The default mechanism for performing field-level validations is to validate the contents of a data control after you leave the control.

Field-level validation is done based on the EditFocusChange event, which is generated each time the user changes input focus on the control level. Each control has a PerformValidations() method that simply calls the corresponding FieldSpec:PerformValidations() and sets the controls HyperLabel with the FieldSpec:Status. When an EditFocusChange event occurs, PerformValidations() is called only for the control that is losing focus.

If a control fails validation, the windows Status property reflects the necessary information to determine which validation failed. The Status:Description is automatically displayed in the status bar, indicating the diagnostic message associated with the failed validation, but the user cannot be prevented from leaving the control.

Tip: Whenever a failed control has focus, the data window's Status property will reflect the help keyword of the associated validation rule (that is, Status:HelpContext). You can use this information to provide a hook into your online help system. For more information on this subject, see "Using an Online Help System" later in this chapter. To do this, you could define a Help Error menu command or push button. The implementation of the corresponding method might look like this:

```
METHOD HelpError() CLASS EmployeeWindow
       SELF:HelpDisplay:Show(SELF:Status:HelpContext)
```

Conditional Controls

Tied to field-level validation is another level of validation implemented by the DataWindow:Prevalidate() method. This method is invoked after each field-level validation. You can, therefore, use it to perform validations that are not specific to a particular control but that are applicable regardless of the control in question.

DataWindow:Prevalidate() has some built-in behavior that allows you to register controls that are automatically disabled or enabled—via DataWindow:DisableConditionalControls() and DataWindow:EnableConditionalControls()—based on the validation status of the window. For example, you might have an OK push button that the user can press to accept the data and close the window. The following code would register this control so that it was only enabled when the validation status of the window was good:

```
METHOD Init() CLASS EmployeeWindow
     SELF:RegisterConditionalControls(oCCOK)
     ...
```

As always, you can add to or replace the automatic behavior described above using a customized Prevalidate() in your data window subclass. For example, you might add behavior to color-code controls that are invalid or extend the automatic disabling of controls to include menu items. The following example uses Prevalidate() to disable a Help Error menu command when the window status is valid and enable it when the window is invalid:

```
METHOD Prevalidate() METHOD EmployeeWindow
     IF SELF:Status <> NIL               // Error
          SELF:Menu:EnableItem(IDM_MENU_HELP_ERROR)
     ELSE
          SELF:Menu:DisableItem(IDM_MENU_HELP_ERROR)
     ENDIF
     SUPER:Prevalidate()
     RETURN NIL
```

Form-Wide Validation

In addition to field-level validation, you can also define form-wide validation rules for your data window, although you cannot define this level of validation using any of the IDE editors. Instead, you implement it by defining an event named ValidateRecord() for your data window subclass.

Form-wide validation rules usually involve cross-field validations (such as "the bonus can be no greater than half the salary and no greater than 10,000 for grades six and lower") that you do not want to perform until, for example, the user moves to another record. The code for this condition might look something like this:

```
METHOD ValidateRecord() CLASS EmployeeWindow
     DO CASE
     CASE (Grade <= 6) .AND. (Bonus > 10000)
          SELF:StatusMessage("Bonus cannot exceed $10,000",MESSAGEERROR)
          RETURN FALSE
     CASE Bonus > (.5 * Salary)
          SELF:StatusMessage("Bonus cannot exceed half the salary",MESSAGEERROR)
          RETURN FALSE
     OTHERWISE
          RETURN TRUE
     ENDCASE
```

ValidateRecord() is called by DataWindow:StatusOK(), which is called automatically whenever the user performs any operation that affects the database as a whole (such as a skip, go to, or commit operation). StatusOK() does all field-level validations first, before calling ValidateRecord().

Note: It is possible to define cross-field validation rules for an individual field, but this is not the recommended practice. It is not logical to perform this type of validation each time a field loses focus, because the user may already be on the way to satisfying the other conditions and would, therefore, be annoyed by an error message. Therefore, when defining field-level validation rules, avoid conditions involving other fields that are better handled on the form level.

Validation for Delete

One operation that is not automatically tied to form-wide validation is delete because some applications may not require deleted records to be valid, while others may. For example, if the application never recalls deleted records or if the data server does not support recalling deleted records, it makes no sense to waste time performing a validation before the delete is allowed. On the other hand, if the application can recall the deleted record, you probably want to make sure the record is valid, even though it is marked for deletion.

DataWindow provides two different methods, Delete() and DeleteValidated(). Use Delete() if the window does not need to be validated before the delete operation, and DeleteValidated() if it does. DeleteValidated() automatically calls DataWindow:StatusOK() to perform field-level and form-wide validations, allowing the delete operation only if the window passes validation.

Action Methods

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

The data window also provides methods for many standard Windows operations such as Undo(), OK(), Cancel(), Cut(), Copy(), and Paste().

Because of the way push button and menu commands are processed by name, it is very simple to produce a data-aware window that invokes any of these standard action methods without writing any code specific to the window. For example, if you assign the symbolic name SkipNext to the Edit Next menu command, choosing the Edit Next will automatically invoke DataWindow:SkipNext(). An excellent example of this can be seen in the StandardShellMenu of the Standard Program.

Data Propagation

Another behavior that you achieve when you link a data window and data server is automatic data propagation between the window and the server.

Changes to the controls in the data window (whether by the user or by code) are first validated then propagated down to the data server if they pass validation.

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

Concurrency Control

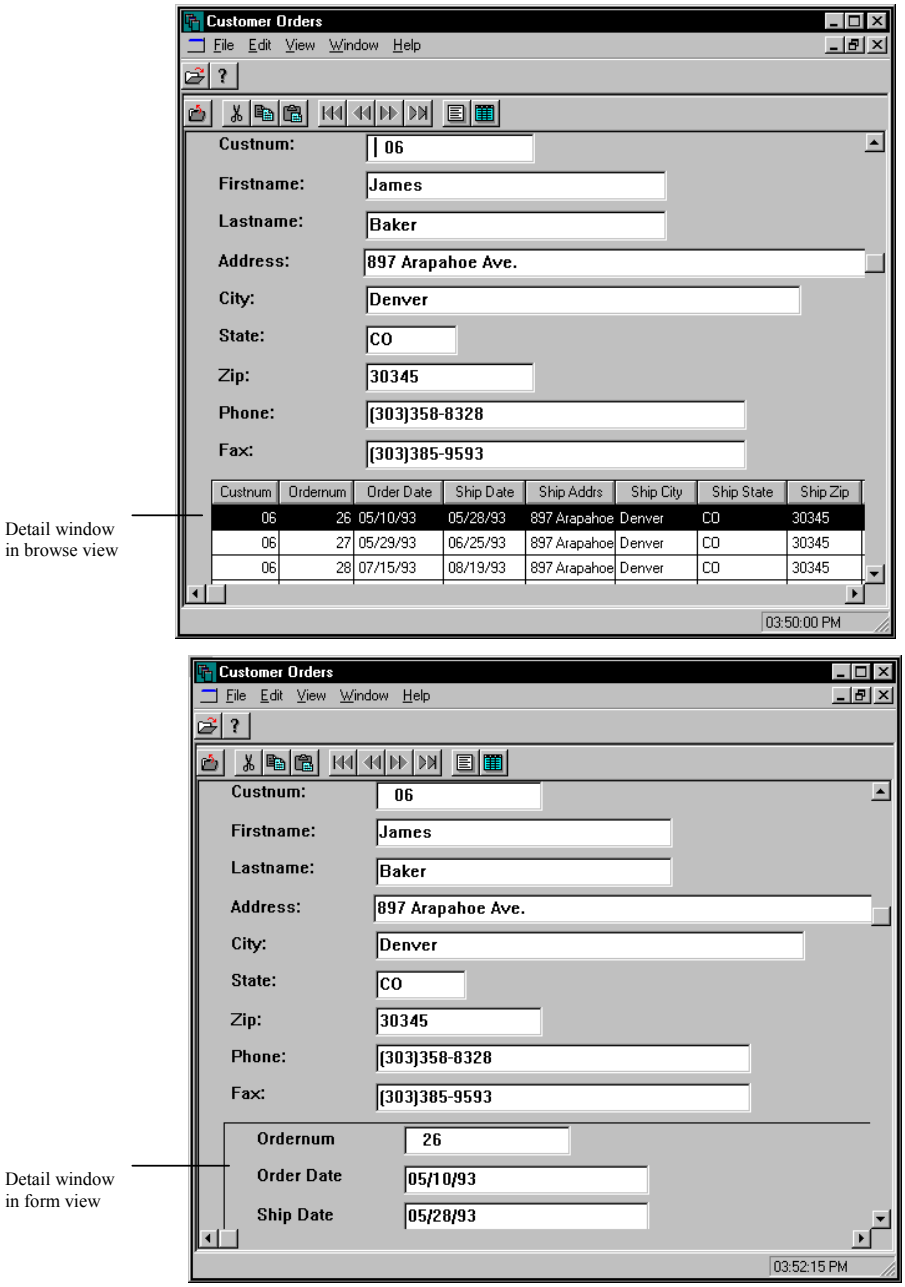
The data window also provides you with several levels of built-in concurrency control so that you do not necessarily have to perform file and record locks manually in your code. Instead, you set the window's ConcurrencyControl:Property by choosing from one of several constant values, depending on the behavior you want (refer to the DataWindow class in the online help system for a complete list of these constants).

Sub-Data Windows

Nesting data windows (that is, placing a data window on another as a sub-data window) is useful for the following reasons:

- When designing complex windows, it is often useful to define a commonly used group of controls as a separate window which is designed independently and which can be reused on other windows as needed. This is no different from the benefits of component reuse already discussed. Remember that a window is not just a visual appearance, it also includes the code that processes commands aimed at the window.
- The common type of window described as a *master-detail* form, such as the ubiquitous customer-order display, often requires that the detail part be presented as a table. This table is not just a visual display of a list control; it requires complex processing of database operations and, therefore, needs to be designed as a data-aware window in its own right. This tabular display is the data browser, or browse view, already mentioned earlier in this section.
- Complex windows, such as the master-detail form illustrated below, need to operate with two databases at the same time. But this makes things a bit more complex: should a delete command be interpreted as deleting the customer, deleting an order, deleting all the orders, or deleting the customer and all the orders? To keep things simple, a data window can be linked with only one data server; to create a master-detail form, you design the detail part as a separate data window, linked to its own server, and place it on the master part, which has its own window. In this case, there is no uncertainty about what is going on: each window gets its own delete command and processes it accordingly.

Note that a sub-data window can be displayed in form or browse view, and both may be useful:



Instantiating a Sub-Data Window as a Control

Controls are normally instantiated with two parameters: an owner, which is a dialog or data window, and a resource ID. (Alternatively, the resource ID can be replaced with a size and position to dynamically instantiate the control.) A dialog or data window is normally instantiated with two parameters: an owner and a resource ID defining the layout of the entire window. This technique of instantiating data windows and controls has already been illustrated in the examples in this section.

When using a data window as a sub-data window, it simultaneously takes on the characteristics of a window and a control. Thus, it needs three parameters: an owner, a resource ID defining the layout of the components of the sub-data window, and a resource ID that refers to the sub-data window as a control in the resource entity of the owner data window.

The code generated by the Window Editor (when you choose the Auto Layout feature and specify a master-detail relationship) performs this instantiation in two steps. First, the sub-data, or detail, window is instantiated in the Init() method of its owner, or master, window with code similar to the following:

```
oSFCustomerWindow_DETAIL := CustomerWindow_DETAIL{SELF, ;
CUSTOMERWINDOW_CUSTOMERWINDOW_DETAIL}
```

CUSTOMERWINDOW_CUSTOMERWINDOW_DETAIL is a constant that refers to the control on the master data window, and oSFCustomerWindow_DETAIL is a protected instance variable used by the master window class to refer to the detail window as a control.

Then, in the Init() method of the detail window's class, the following code completes the instantiation:

```
SUPER:Init(oWindow,ResourceID{"CustomerWindow_DETAIL"},iCtlID)
```

CustomerWindow_DETAIL is the name assigned to both the resource entity and the class defining the detail window.

Note: When a data window in instantiated as a control, its menu and toolbar are not available. Instead, it uses the menu and toolbar associated with the data window in which it is nested.

Relating the Data Servers

Once instantiated, the two underlying data servers must be linked so that movement in the master window is reflected in the detail window. There are two methods in the DataWindow class to accomplish this: SetSelectiveRelation() and SetRelation().

With both of these methods, you link the windows based on a key value in the master data server, causing a lookup in the controlling order of the detail data server.

The difference between the methods is that SetRelation() does not limit the selection in the detail window to records with matching keys. It simply does a lookup and positions the record pointer to the first matching key. You could, for example, switch focus to the detail window and view records that did not match the current record in the master window.

SetSelectiveRelation() implements a more intelligent way to handle the link between a master and detail window, limiting the detail window to displaying only those key values that match the key value currently displayed in the master window. This method sets up a subset rule for the detail window so that methods that move the record pointer, such as GoTop(), GoTo(), GoBottom(), Skip(), and Seek(), are limited to records matching the relationship.

SetSelectiveRelation() is the method used in the code generated by the Window Editor. For example, this code appears in the Init() method of the master data window just after the detail window is instantiated:

```
SELF:SetSelectiveRelation(oSFCustomerWindow_DETAIL, #CUSTNUM, "CUSTNUM")
```

Nesting Sub-Data Windows

The master-detail relationship that has been discussed so far is the most common use of a sub-data window, but it is not the only possible variation. The nested window structure can be assembled for more complex arrangements:

- Master and two details: a customer form has one nested form with orders and another with payments
- Master-detail-detail: a customer form contains a nested order form which contains a nested list of items

SetSelectiveRelation() and SetRelation() support both of these arrangements. You can establish a relation from one master server to two or more detail servers by including several calls to the method in the master window's Init() method—subsequent calls add to the list of relations for the master. You can also establish a chain of relationships by including the proper method call in the Init() methods of the master and detail data windows. This last example is rather complex.

Depending on the application and the user, it may or may not be a good idea to design a window like that—even if it is technically possible, the end user may not be able to manage the window. But what should you do if the information is that complex?

One option is to split the structure up into two windows: instead of one complex window with two levels of nesting, you might have two windows, one of which is nested, or even three simple windows. The relationship between the data can still be maintained, of course, so that moving to a new customer will be automatically reflected in the order and item windows; this is done the same way, regardless of whether the three windows are nested or independent.

Because of the way data windows adopt their behavior at the time of instantiation based on who is their owner, it is very easy to make these changes. After designing the completely nested, single-window version, locate the place where each window is instantiated, change the owner, and eliminate the instantiation parameter identifying the window as a control:

```
// Instantiate data window as sub-data window
CustomerWindow_DETAIL{SELF,CUSTOMERWINDOW_CUSTOMERWINDOW_DETAIL}

// Instantiate data window as separate window
CustomerWindow_DETAIL{oWindow}
```

Second Table for Lookup

Some cases that require linkage to two databases do not actually require a sub-data window. Consider an employee window with a combo box control showing the employee's department: when dropping down the combo box you expect to see a list of departments from the department table. Instead of creating a sub-data window for the combo box, you can link the combo box control to the department table. (The Window Editor provides an automatic way to connect a combo box to a field in a table or a column in an array.) In this case, the second table is used only for lookup purposes; the control is primarily linked to the employee table, and only the employee table is updated.

Using an Online Help System

The help system is a very important aspect of any application because it is often the first thing the end user sees. Users may judge an application on the quality of its help system—especially if the online help is provided in lieu of printed documentation.

Developing an online help system is an in-depth process that requires you to analyze your application very closely. Among other things, you must decide the level of detail to provide and how to organize the hierarchy of help topics. Once the system is planned, you face the tasks of writing and appropriately identifying the help text, then building the help system. The subject of developing an online help system, however, is beyond the scope of this section. Instead, this section focuses mainly on how to interact with the actual help (.HLP) file(s) that make up the online help system.

Tip: The Microsoft Windows Help Compiler and Hotspot Editor are provided as part of your Visual Objects package, along with associated help databases to explain how to use them. These help databases contain useful information about developing an online help system and about how to use the respective systems. Refer to "Installing and Starting Visual Objects" in the *Getting Started* guide for more information on how to install these components.

Specifying Keywords

Even though the actual development of an application's help system may come late in the development cycle, you should always have help in mind when designing the various components of your application.

For this reason, the various IDE tools (such as the Window Editor, the Menu Editor, and the data server editors) allow you to assign a HelpContext property to any window, window control, data server, data field, field specification, menu, or menu command defined in your application. This property, known as a *keyword*, is a unique ID that serves as a hook into the application's help system.

In fact, any object in your system with an associated hyperlabel can have a help keyword. You simply specify it as the fourth argument when you instantiate the HyperLabel object, just as in the code generated by the editors. This results in the creation of the HyperLabel:HelpContext property, which you can access in your code and which is automatically passed along to the default event handler for help requests.

Important! *The keywords that you use must be coded in your help source files using an alternative keyword table identified by the uppercase letter "C". See the HelpDisplay:Show() method in the* online help system *for details.*

Associating Help Files

Thus, you create a link between objects in the application and keywords in a help file via the HyperLabel:HelpContext property using the IDE editors. However, you must also create a link between the application and the help file.

Actually, the link is between a window (rather than the entire application) and a help file, and you establish it by assigning a HelpDisplay object to the window's HelpDisplay property. For example, to assign the help file, MYAPP.HLP, to the shell window of an application, you would include the following statement in the shell window's Init() method:

```
SELF:HelpDisplay := HelpDisplay{"myapp.hlp"}
```

> **Tip:** Using the Window Editor, you can assign the help file name to the window's Help File Name property, and the Window Editor will generate the code necessary to link the help file.

In most applications, this is all that is needed to implement context-sensitive help for the menu commands, controls, and toolbar buttons in your application. It is standard practice to have one help file per application, and linking the help file to the shell window in this manner makes it available to all child windows in the ownership chain. In other words, by default, a child window uses the help file linked to its owner.

Using this architecture, however, it is easy to see how any window in the application can have its own, separate help file. For example, assuming that the customer data window is owned by the shell window in the previous example, this statement in the CustomerWindow:Init() method will use CUSTOMER.HLP as the help file (instead of MYAPP.HLP) when the customer window has focus:

```
SELF:HelpDisplay := HelpDisplay{"customer.hlp"}
```

Built-in Context-Sensitive Help

Without any programming in Visual Objects other than associating the help file, you get context-sensitive help for every control, menu command, and toolbar button in your application, as well as for various regions of the window. You just need to make sure that the keywords in the help file and the HyperLabel:HelpContext property match up and that the appropriate help file is linked to the window.

Context-Sensitive Help

The user can get context-sensitive help in one of two ways. The first way is to press the Help key (F1) for help on the menu command or control with focus. For example, to get help on a menu command, the user could highlight the menu command (without selecting it) and press F1. If no control or menu command has focus when the user presses F1, the default behavior is to display the Contents topic defined in the associated HelpDisplay.

The second way is by pressing Shift+F1. On a datadialog window, this produces a special help cursor (consisting of an arrow and a question marked combined). To get help on a control or toolbar button, the user locates the control or toolbar button with the help cursor and clicks on it. On shell windows, Shift+F1 behaves like F1, displaying either the Contents topic or the help topic for the item with focus.

Help Request Event Processing

In either case, this is what happens:

1. The user's request for help generates a HelpRequestEvent object, which completely describes the context in which the request occurred.
2. The HelpRequestEvent object is passed to the window's HelpRequest() event handler.
3. Window:HelpRequest(), the default event handler, invokes the WinHelp system via the HelpDisplay:Show() method using the appropriate keyword as an argument. The code in Visual Objects that automatically launches WinHelp looks something like the following:

   ```
   METHOD HelpRequest(oHRE) CLASS Window
       SELF:HelpDisplay:Show(oHRE:HelpContext)
   ```

   When the HelpRequestEvent object is created, its HelpContext property is, in most cases, defined as the HyperLabel:HelpContext property of the item for which help is being requested.
4. WinHelp finds the keyword in the .HLP file and displays the associated topic in its window or an error if the keyword cannot be found.

Window Regions

In addition to getting help on controls and menu commands using the built-in behavior of Window:HelpRequest(), the user can also get help on various regions of the window using Shift+F1. If you are planning to implement this level of help, use the following keywords in your .HLP file:

Keyword	Identifies Window Region
Window_Border	Border
Window_Caption	Title bar
Window_MaxBox	Maximize button
Window_MinBox	Minimize button
Window_SysMenuBox	System menu button
Window_WindowCanvas	Canvas area

Note: These keywords are automatically generated by the Window:HelpRequest() event. You do not define them using the Window Editor or by associating a hyperlabel with the window.

If you click on an unknown area of the screen with the help cursor, the default event handler displays the Contents topic defined in the associated HelpDisplay.

Important! *For data windows, these special help keywords are processed only if the HelpContext property for the data window is* ***not*** *defined. Otherwise, requesting help for any of the window regions listed above generates a help request using the keyword defined in the data window's HelpContext property.*

If the built-in help facilities are not enough to suit your needs, you can always override or supplement the HelpRequest() event in your Window subclass. The built-in behavior, however, is fairly standard and comprehensive.

Implementing Additional Help

Two other typical ways to implement help are in response to a menu command selection (as in the typical Help menu found in almost all Windows applications) or a push button click (as in a Help button on a complex dialog window). As discussed earlier in this chapter, both menu command selections and push button clicks generate command events based on their name. So, you can create a method for your window class to process the event and put the help processing logic in there. Using this technique, it is easy to tie the request for help in with the help file using the HelpDisplay:Show() method.

HelpDisplay:Show(), in addition to supporting the ability to look up specified keywords in a help file, is able to process some reserved keywords to give you access to certain standard help features. For example, "HelpIndex" displays the Contents topic defined in your help file, and "HelpOnHelp" displays the Contents topic defined in your Windows help file (this is normally WINHELP.HLP, in which the Contents topic is How to Use Help).

Assuming the menu commands had events named HelpMainIndex and HelpUsingHelp, you could implement these standard help features as follows:

```
METHOD HelpMainIndex() CLASS MyShellWindow
      SELF:HelpDisplay:Show("HelpIndex")

METHOD HelpUsingHelp() CLASS MyShellWindow
      SELF:HelpDisplay:Show("HelpOnHelp")
```

Note: This type of processing can also appear as part of the MenuCommand() or ButtonClick() event handler methods.

Instead of using a method to process the help request, you can choose to display a dialog window or some other type of window. An excellent example of this can be found in the Standard Program's implementation of Help About. This menu command has an event name of HelpAbout, which is linked to a dialog window of the same name. The dialog window (designed using the Window Editor) describes the application in brief and, when the user clicks the OK button, removes itself from view.

These are good techniques for implementing help topics for entire windows, data servers, and so on. Although the editors let you define HelpContext properties for these objects, they are not normally available for context-sensitive help. However, if you intend to link them to help file topics, having the appropriate hyperlabels generated defining the keywords makes your job easier. For example, to link the Help About menu command to a shell window's keyword, you would use the following code:

```
METHOD HelpAbout() CLASS MyShellWindow
    SELF:HelpDisplay:Show(SELF:HyperLabel:HelpContext)
```

Then, you could redefine this for a data window, say CustomerWindow, to display the help text associated with its data server:

```
METHOD HelpAbout() CLASS CustomerWindow
    SELF:HelpDisplay:Show(SELF:Server:HyperLabel:HelpContext)
```

Chapter

12 Other Features of the GUI Classes

The previous chapter focused on some of the more typical arrangements and uses of windows in GUI applications and discussed how to use the GUI classes and visual editors to accomplish them. There are other features of the GUI classes that are not directly tied to any of the visual editors. This chapter discusses several otherwise unrelated uses of the GUI classes that do not fit into the framework of the previous chapter.

Drawing Objects

There are a number of GUI classes representing shapes that you can draw on a window or printer canvas. You can find them using the Repository Explorer because they all inherit from an abstract class named DrawObject.

As an example, consider a drawing program that enables the user to select a shape and size it using the mouse. The main window and one of the drawing methods might look like this:

```
CLASS DrawWindow INHERIT ChildAppWindow
      HIDDEN aShapes AS ARRAY

METHOD ShapeEllipse() CLASS DrawWindow
      AAdd(aShapes, EllipseObject{SELF})
      SELF:Draw(aShapes[ALen(aShapes)])
```

The window maintains an array of the objects that the user has drawn so far. When the user clicks Ellipse on the Shape menu, the application invokes the ShapeEllipse() method. This method adds a new ellipse to the array and then invokes Window:Draw() to draw the ellipse. In the program, there is also code to track the user's mouse moves and drags, so the user can position the ellipse.

Keeping Track

If the user clicks the mouse, how do you determine which drawing object they are referring to?

Each drawing object has a virtual variable named BoundingBox that returns the rectangle that exactly encloses it in the coordinate system of the window's canvas. You use the PointInside() method of BoundingBox when you wish to test if the mouse is stationed over a particular object. If there is an array of objects, then you must loop through the array until you find an object that returns a value of TRUE for PointInside().

```
METHOD MouseButtonDown(oME) CLASS DrawWindow
     LOCAL m, k
     FOR k := 1 UPTO ALen(aShapes)
          IF aShapes[k]:BoundingBox:PointInside(oME:Position)
               m := k
          ENDIF
     NEXT
     // Now, do something to aShapes[m]
```

This code handles the case where the mouse lies in several drawings. The variable *m* ends up referring to the one that was last appended to the array. Since the array reflects the order in which the user created the drawings, the algorithm picks the one on top if there is an overlapping pile of them on the canvas.

The drawing objects in the GUI classes are good enough to support simple block diagrams in two dimensions. Take a look at the Business Graphics sample application on the Samples tab page of the Application Gallery for a working example. For sophisticated business graphics, you will need a real graphics server. One way to produce business graphics is to send the data to a spreadsheet using the GUI classes' DDE feature (described later in this chapter). Another way is to call the C interface to a commercial graphics server.

Working with Controls

The GUI classes also support a complete range of standard controls which you can see using the Class Browser to browse the class hierarchy beneath the class Control.

The largest group of controls derives from TextControl. These controls either present text, capture it, or both. They all have a virtual variable named TextValue that you use to set and retrieve the text.

The meaning of TextValue is obvious for many controls. For certain text controls, such as list boxes and editors, TextValue refers to the entire text content of the control, not just the part that the user has selected. Other virtual variables, such as ListBox:CurrentItem, handle these cases. Similarly, there is a virtual variable called Value that returns the value stored in the control as a USUAL.

Transferring Data Using the Clipboard

The ClipBoard class supports raw text, tab-separated fields, and bitmaps. However, most of the clipboard use does not need the ClipBoard class. If the user cuts a piece of text from a multiline edit control, for example, the cut text automatically goes to the clipboard. Similarly, you can program list boxes to interact with the clipboard or not, as you choose. The only time you really need to use the ClipBoard class is if you wish to manipulate clipped data explicitly.

The purpose of the ClipBoard class is to enable the application to send and receive data directly to and from the clipboard, without going through a control, such as a list box. The ClipBoard class has methods specifically designed for this purpose, such as Insert() to insert an item in the clipboard.

Implementing Drag-and-Drop

The interpretation of drag-and-drop in the GUI classes is the same as that adopted by the Win32 API in which the only drag-and-drop server is the Windows File Manager. When the user drags files from the Windows Explorer and drops them on a shell window, the shell window acquires the list of file names in the drag-and-drop load. Other interpretations of the drag-and-drop metaphor do not exist in the Windows API.

You can, however, implement your own interpretation of drag-and-drop using the MouseDrag and MouseButtonUp events. The Visual Objects dispatcher sends the MouseDrag event to a window as soon as the drag starts and then keeps sending it, updating the mouse position a few times per second. Your code should look at the first drag position and decide what is being dragged. When the user finally lets go of the mouse button, you get the MouseButtonUp event, which you can interpret as a drop.

Additionally, there are various Drag… and Drop… classes, methods, and events available. Please refer to the online help for detailed information about the individual classes, methods, and events.

Using Dynamic Data Exchange

Dynamic Data Exchange (DDE) is a powerful Windows service that allows your Visual Objects applications to exchange data with, and even exert control over, other Windows applications.

For many reasons, programming DDE at the Windows API level can be quite complex. As you will see in this section, Visual Objects handles the vast majority of the technical details for you, allowing you to concentrate instead on the application itself.

Overview of DDE Basics

The Windows Event Model

In the Windows environment, Windows itself controls and coordinates the activities of all running programs. Several Windows applications can each be doing several things at the same time. As the user moves focus from one application to another, Windows sends a flurry of messages to the applications, keeping them up-to-date about what the user is doing.

Most of the communication involves applications asking for information or services from Windows, and Windows sending messages back to the applications containing information or notifications of user or other events. When the user clicks a button or makes a menu choice, it is *not* your application that is dealing the with user. Windows does all the work and passes events to the appropriate applications after the fact.

Due to the event-based architecture that connects applications to Windows, it is possible for Windows to coordinate communication *between applications*. This inter-application communication process is known as DDE.

DDE makes it possible for Windows application to pass messages back and forth in a client/server arrangement, where the *client* application connects to a *server* application and makes requests for information or services.

DDE Basics

DDE is conceptually very simple. DDE functionality can be divided into three general categories:

- Executing commands
- Requesting data explicitly
- Requesting data continuously

Any client application can have one or more of any of these operations active at the same time. Windows does not impose a practical upper limit on the number of DDE conversations. However, individual DDE servers may limit the number of simultaneous conversations they can support.

Command Execution

DDE allows the client to request that the server execute various commands. This is the simplest form of DDE communication because there is little timing involved—it is a one-way conversation. The client does the talking and the server performs the actions. The client does not expect any communication back from the server, other than error notifications (which are very generic).

"Cold" Data Links

If the DDE client wants data back from the server, it gets a bit more complicated. The client can establish a data link that is only activated when the client explicitly requests the data. In DDE jargon, this is known as a *cold* link.

The client notifies the server that it wants a particular kind of data, then waits around for a message back from the server. It is a one-shot deal. For example, a client application might ask a server application for a list of the files it has open. This request is carried out one time. The client can make the request as often as it likes, but the server will respond only once per request.

"Warm" and "Hot" Data Links

DDE also makes it possible for the client to establish a link that is *continuously* updated whenever the data changes. This is known as a *hot* link. Referring back to the previous "list of open files" example, a hot data link would cause the server to send the list of files to the client whenever the list changes. The client does not have to explicitly request the data—once the hot link is established, it functions automatically.
A *warm* link is half way between hot and cold. Instead of automatically sending data whenever it changes, the server will instead advise the client that something has changed, and it is up to the client to explicitly request that the data be sent. Warm links are fairly rare and are not supported by Visual Objects.

Servers, Topics, and Items

DDE communication is divided into three logical parts: servers, topics, and items.

Servers. Any Windows application can be a DDE client; there is nothing special required other than a few function calls. In order to be a DDE server, however, each application must be deliberately programmed to provide DDE services. Not all applications will respond to DDE communication requests. Microsoft Excel is an example of a Windows application that provides DDE services. The Windows NotePad is an example of an application that does not respond to DDE requests.

Topics. Each DDE server divides the services it offers to clients into categories, called *topics*. Unfortunately there are no standards, much less consistency, among DDE servers; therefore, you must rely on documentation to explain the topics supported by each application that offers DDE services. A client establishes communication with a server by requesting a specific topic. Some servers offer only a single topic while others support several. A topic called SYSTEM is *usually,* but not always, supported by DDE servers. Topics are also frequently the names of files that the DDE server has open at the moment. For example, a word processing application might offer LETTER.DOC as a topic.

Items. Within a topic, the DDE server further breaks down services into sub-categories called *items*. Items take many forms, from macros to components of documents. For example, an item in a spreadsheet might be a macro called POST MONTH, or an item in a word processor might be a component called PAGE HEADING. Like topics, items do not follow any particular pattern or standard.

Inter-Process Communication (IPC)

Visual Objects fully supports DDE, primarily via a set of classes in the GUI Classes library that begin with the letters "Ipc." These classes fall into two broad categories, Client and Server, which are summarized in the following tables. Note that these two categories share supporting classes for data and event management.

Client Classes

Class Name	Description
IpcClient	Establishes a DDE communication link with a server application and offers a variety of methods for requesting data and services.
IpcClientErrorEvent	A package of DDE-related error information, which is passed to the IpcClient:ClientError() method.

Server Classes

Class Name	Description
IpcServer	Establishes the application as a DDE server and lists the topics to which the server will respond; offers a variety of methods for handling client requests.
IpcDataRequestEvent	A package of information about a data-related request received from a client.
IpcExecuteRequestEvent	A package of information about a command-related request received from a client.

Shared Classes

Class Name	Description
IpcTopic	A package of information used to define a topic and item; both the server and client classes use this class to list the topics and items of interest.
IpcDataUpdateEvent	A package of information that carries the data associated with a topic and item; both the server and client receive events from this class.

The Client Classes

To attempt communication with a particular DDE server, create an instance of the IpcClient class. In the following example, an attempt is being made to connect to a server called SALES:

```
oIpcSales := IpcClient{"SALES"}
```

An IpcClientErrorEvent occurs if the requested server is not available. In order to receive the error notification, you must create an IpcClient subclass and provide a ClientError() method. (See Error Handling for more information.)

Once successfully connected to a DDE server, the client application has two options. It can begin executing commands, or it can register the topic and item(s) of interest and begin requesting data.

Command execution is accomplished with the IpcClient:Execute() method. This method expects a topic and item along with a command string. For example, the following illustrates the SALES server receiving a command to execute. The topic is REPORTS, the item is CUSTOMERS, and the command is PRINT.

```
oIpcSales:Execute("REPORTS", "CUSTOMERS", "PRINT")
```

Receiving data is a bit more complex. To register topics and items of interest, create an instance of the IpcTopic class and add items. The topic object is then used in subsequent requests for data. In the following example, a MONTHLY topic, with items JANUARY and FEBRUARY, is established.

```
oCustomer := IpcTopic{"MONTHLY"}
oCustomer:AddItem("JANUARY")
oCustomer:AddItem("FEBRUARY")
```

Next you need to tell the SALES server that you want to receive information about the topic and list of items. You have two options: to receive this information once, or automatically whenever the information changes. This is controlled by logical flag, where TRUE means continuous and FALSE means one time only.

```
oIpcSales:RequestData(oCustomer, FALSE)
```

In order to receive data back from the SALES server, you need to supply a DataUpdate() method. The only way to do this is to create a new class and inherit from the IpcClient class, as described previously for error handling. (See Error Handling for more information.)

For purposes of this quick overview, assume the SALES server is an instance of the class called IpcSales. In the example below, IpcSales is indeed a subclass of IpcClient. It declares a DataUpdate() method that receives an IpcDataUpdateEvent object.

```
CLASS IpcSales INHERIT IpcClient

METHOD DataUpdate(oDataUpdateEvent) CLASS IpcSales
```

Within the body of the IpcSales:DataUpdate() method, you can extract the information that is returned from the server as a result of an earlier oIpcSales:RequestData() event.

In the following example, the oDataUpdateEvent:GetData() method is called to extract the data string from the object. You can then take a look at the topic and item properties to determine what you received.

```
cData := oDataUpdateEvent:GetData()
IF oDataUpdateEvent:Topic == "MONTHLY"
     IF oDataUpdateEvent:Item == "JANUARY"
          // cData contains the January data
     ELSEIF oDataUpdateEvent:Item == "FEBRUARY"
          // cData contains the February data
     ENDIF
ENDIF
```

This may seem unnecessary, but due to the unpredictability of an event-driven environment, you cannot be assured of the precise timing of events. The DDE server might not respond in the exact order of your data requests, or error event notifications could be sprinkled in the midst of the data update events.

In the following example, the SALES server is notified of a change to the status of the MARCH item in the MONTHLY topic. The *cData* string contains the new data for MARCH.

```
oIpcSales:ChangeData("MONTHLY", "MARCH", cData)
```

The IpcClient:ChangeData() method is used most often in situations where the DDE server is a database management application and client applications are allowed to make changes to the data.

Finally, the IpcClient:Destroy() method is (despite the name) the polite way to terminate a conversation with a DDE server.

```
oIpcSales:Destroy()
```

The Server Classes

If you want your Visual Objects application to be a DDE server, you must register it as such with Windows. This is accomplished by creating an instance of the IpcServer class and adding a list of topics and items to which the application will respond. Then, your application sits back and waits for any information. Actually, it can be busy doing other things while waiting for DDE requests to service. Your application will receive various event notifications from DDE clients in much the same way it receives user interface events from the end user.

Right from the start, you will have to create a subclass from the IpcServer class. You are responsible for implementing several key methods that will handle commands and data requests. In the following example, a SalesServer class is defined. When an instance of the class is created, it constructs the list of topics and items it supports.

```
CLASS SalesServer INHERIT IpcServer

METHOD Init() CLASS SalesServer
     SELF:LoadTopics()

METHOD LoadTopics() CLASS SalesServer
     LOCAL oTopic AS IpcTopic
     // Add Reports topic and items to server
     oTopic := IpcTopic{"REPORTS"}
     oTopic:AddItem("CUSTOMERS")
     oTopic:AddItem("VENDORS")
     SELF:AddTopic(oTopic)
     // Add Monthly topic and items to server
     oTopic := IpcTopic{"MONTHLY"}
     oTopic:AddItem("JANUARY")
     oTopic:AddItem("FEBRUARY")
     oTopic:AddItem("MARCH")
     SELF:AddTopic(oTopic)
```

It is not strictly necessary to construct the list of topics and items during the Init() process. In fact, one of the slick things about DDE is that you can dynamically adjust the topics and items to match the needs or capabilities of the operating environment. However, if there are some stock topics and items that you always plan on supporting, this is a good time to define them.

This is all you need to do in order to register the SalesServer with Windows. Any Windows application capable of performing DDE operations can now make attempt to communicating with this application. All they have to know is the name of this server (which in the case of Visual Objects is the name of the server application's .EXE file), and the list of topics and items to which it will respond.

Our next task is to implement methods for handing command and data requests: IpcServer:ExecuteRequest() and IpcServer:DataRequest().

With IpcServer:ExecuteRequest(), all you have to do is look at the topic, item, and command string and decide what action to take. The DDE client does not expect any communication back from this method, so you can treat it in many respects like a menu choice. Many DDE servers are arranged around the host application's menu structure for this very reason.

The method receives an IpcExecuteRequestEvent object, which contains the topic, item, and command string sent by the DDE client. In the following example, the SalesServer is shown to be able to respond to the PRINT and PREVIEW commands for the REPORT topic (items CUSTOMERS and VENDORS):

```
METHOD ExecuteRequest(oEvent) CLASS SalesServer
     LOCAL lPrint AS LOGIC
     LOCAL lPreview AS LOGIC
     lPrint := (oEvent:Command == "PRINT")
     lPreview := (oEvent:Command == "PREVIEW")
     IF oEvent:Topic == "REPORTS"
          IF oEvent:Item == "CUSTOMERS"
               ReportCustomers(lPrint, lPreview)
          ELSEIF oEvent:Item == "VENDORS"
               ReportVendors(lPrint, lPreview)
          ENDIF
     ENDIF
```

Finally, the IpcServer:Destroy() method is used to remove the DDE server from the host application.

```
oSalesServer:Destroy()
```

Starting a DDE Conversation

The simplest form of DDE conversation is where the DDE client drives the DDE server via simple commands. The Execute() method of the IpcClient class is used to send a command string to the DDE server. The general form of this process is listed below:

```
oIpcClient := IpcClient{"SERVERNAME"}
oIpcClient:Execute("TOPIC", "ITEM", "COMMAND")
```

The SERVERNAME, TOPIC, and ITEM are standard DDE terms. Things get a little fuzzy when it comes to the COMMAND string. There are no standards whatsoever when it comes to command strings. There are a few conventions followed by some Windows applications, especially those from Microsoft.

- Command strings are usually enclosed in square brackets. This allows you to place several commands in the same string.

  ```
  cCommand := "[command1][command2][command3]"
  ```

- Command strings often take the form of function calls, where the name of an action or service is followed by comma-separated parameters enclosed in parentheses.

  ```
  cCommand := "[ServiceName(Param1, Param2)]"
  ```

- Remember that the commands are strings, so data types are not supported. You almost never have to enclose character string parameters in quotes (there are random exceptions, of course, just to keep it interesting!). In the example below, the DATA.DAT file name does not have to be in quotes, the way it would be in ordinary source code.

  ```
  cCommand := "[Process(data.dat)]"
  ```

  To make the DDE execute commands easier to remember, they are often named to match the server application's menu structure. For example, most Windows applications have a FILE menu, with OPEN, PRINT, CLOSE, and EXIT choices. The corresponding DDE commands are often named along these lines. If the application has a dialog associated with the menu choice, the DDE command usually accepts parameters that match the controls in the dialog.

  The following examples illustrate a typical set of functions to open a data file, print it, then close the file (various quotes and brackets have been omitted to make the syntax more clear):

  ```
  FileOpen(data.dat)
  FilePrint(1,3,PRN,0)
  FileClose()
  ```

  In the FilePrint() example above, the parameters might represent selecting one copy (1), the third option in a group (3), selecting PRN from a list of devices, and not checking a check box (0). It makes sense after you have worked with a few such commands.

Starting Other Applications

More often than not, your application is responsible for checking to see if the desired DDE server application is running, and if not, to launch it.

The main App class in your Visual Objects application has a Run() method that you can use to launch other Windows applications. In the following example, an application called APPNAME is launched. The *nResult* code will contain either a handle to the application, or a Windows error number in the range 0–32 (handle numbers are always greater than 32).

```
nResult := oApp:Run("APPNAME")
IF nResult > 32
    // Launched successfully, nResult = handle
ELSE
```

```
    // Problem, nResult = error number
ENDIF
```

Typical error numbers include: 0–Out of memory; 2–File not found; 3–Path not found. If you do not include a full directory path with the application name, Windows works its way through the following steps as it searches for APPNAME:

- Current directory
- Windows directory
- Windows system directory
- Directories in the order specified in the PATH environment variable
- Directories mapped in a network

Note: In addition to the name of the application, you may also include command line arguments in the string passed to the App:Run() method.

Error Handling

In most DDE conversations, there are many things that can go wrong. Visual Objects handles the numerous low-level timing details, as described earlier. However, there is *still* plenty to worry about. In order to handle DDE-related errors, you have to take a deeper plunge into the Visual Objects class architecture.

The IpcClient class has a ClientError() method that gets called whenever a DDE-related error occurs. The IpcClient:ClientError() method does not actually do any error handling. This is known as a *deferred* method, because Visual Objects assumes you will define a ClientError() method in your subclass.

In the following example, an IpcClient subclass called MyClientClass defines a ClientError() method. Note that the method receives an IpcClientErrorEvent object. In this example, the programmer decided that "item not found" errors can be dealt with elsewhere while all others should halt the communication process:

```
CLASS MyClientClass INHERIT IpcClient

METHOD ClientError(oErrorEvent) CLASS MyClientClass
    /*
    Check the DDE error type, "item not found" is OK
    but any other error is a major problem.
    */
    LOCAL lContinue AS LOGIC
    IF oErrorEvent:ErrorType == IPCITEMNOTFOUND
        lContinue := TRUE
    ELSE
        lContinue := FALSE
        DisplayErrorMessage("Fatal DDE error.")
    ENDIF
    RETURN lContinue
```

The ClientError() method should return a logical value indicating whether or not the DDE communication should continue (TRUE = continue, FALSE = stop).

The IpcClientErrorEvent object's main purpose in life is to deliver a property called ErrorType, described in the table below:

Message	Meaning
IPCITEMNOTFOUND	The server does not have an item of that name.
IPCOUTOFMEMORY	The system is out of memory.
IPCSERVERNOTFOUND	The indicated server is not found.
IPCTOPICNOTFOUND	The server does not have a topic of that name.

OLE Automation can be used as an alternative. This makes the communication with the client much easier since you do not have to take care of starting the other program.

Avoiding the Hourglass

Sometimes, the user asks you to perform some lengthy action, which requires no additional user input, so an hourglass displays until the operation is complete. Visual Objects allows you to avoid the hourglass by doing background processing during this time when the user is thinking.

You accomplish this by passing a special argument to the App:Exec() method, but to call this method you need the name of the owner App object. The ApplicationExec() function provides a convenient way of calling the App:Exec() method from any method, as demonstrated in this example:

```
CLASS SolarWindow INHERIT TopAppWindow
     HIDDEN lStop
     HIDDEN oColor

METHOD Start() CLASS App
     SolarWindow{SELF}
     SELF:Exec()

METHOD Darken() CLASS SolarWindow
     LOCAL oOldBrush AS Brush

     oColor := Color{COLORWHITE}
     DO WHILE !lStop
          ApplicationExec(EXECWHILEEVENT)

          oOldBrush := SELF:Background
          SELF:Background := Brush{oColor}
```

```
            IF oOldBrush ◇ NULL_OBJECT
                oOldBrush:Destroy()
            ENDIF

            --oColor:Red
            --oColor:Green
            --oColor:Blue
            SELF:CanvasErase()
        ENDDO

METHOD Init(oOwner) CLASS SolarWindow
    SUPER:Init(oOwner)
    SELF:Caption := "Click mouse when dark enough"
    SELF:Show()
    SELF:Darken()

METHOD MouseButtonDown() CLASS SolarWindow
    lStop := TRUE
```

You could use any event handler. The ApplicationExec() will run until there are no more events in the queue. Then it will drop through to the next instruction. At that point, you know there are no events waiting for service, so you can afford to do some processing.

If you want to do something lengthy, you should call ApplicationExec() with the EXECWHILEEVENT argument at frequent intervals during the task. Effectively, this means polling the event loop to see if there is anything to do. If an event arrives, the Visual Objects dispatcher queues it. As soon as you call ApplicationExec(), it begins to dispatch events in the normal way. When there is nothing to be done in the background, you call ApplicationExec() with no argument, in which case it just waits patiently for an event.

Custom Events

If you wish to add a single new event type, you can do so by supplying your own Dispatch() method. The Event object that the Visual Objects dispatcher supplies to Dispatch() is a generic event, so you will need to know which Windows message type you want to promote to an event.

Assume, for example, that it is a WM_TIMER message and you want to make an event type named StrobeEvent. You detect the message type by examining the Message access of the event object. This returns an integer returning the message ID.

Once you have detected the right message, you just subclass Event:

```
CLASS StrobeEvent INHERIT Event

METHOD Init(oEvt AS Event) CLASS StrobeEvent
    SUPER:Init(oEvt)
```

Then you construct the event and send it to its handler:

```
METHOD Dispatch(oEvent) CLASS TimerWindow
     local oSE as StrobeEvent
     IF oEvent:Message == WM_TIMER
          oSE := StrobeEvent{oEvent:}
          SELF:Ticker(oSE)
          RETURN TRUE
     ELSE
          RETURN SUPER:Dispatch(oEvent)
     ENDIF
```

If you want to make many events, you can create them in the same way. Your Dispatch() method would have a CASE construct to determine the Windows message type, construct the event, and send it to the correct handler. On return from the handler, Dispatch() must return TRUE, by convention.

If you are only interested in a single message type, you could put the handler code in Dispatch() and not bother to create an event. The technique of creating a separate event that is handled in the Dispatch() method, however, is more in keeping with the architectural principles set out in Chapter 4, "Standard Components-Classes, Objects, and Libraries," of this guide. It also allows you to add specialized functionality into your new Event subclass.

Chapter

13

Printing

There are two principal ways to print in a Visual Objects application:

- Using the ReportQueue class to run the Visual Objects Report Editor
- Using printing capabilities of the GUI Classes library

The technique(s) you use will depend on your needs and style of programming. To print reports in your application, you will use the ReportQueue class. To give users the capability to print data entry forms, you will use the printing facilities in the GUI Classes library. This chapter describes both techniques.

Reports

The Visual Objects Report Editor is integrated to allow you to create state-of-the-art reports from directly within the IDE. This chapter explains how to print, preview, and save reports that you design with the Report Editor. Creating and designing reports, on the other hand, is not the subject of this chapter—these topics are covered in the "Using the Report Editor" chapter in the *IDE User Guide*.

ReportQueue Class

To print a report from a program using the ReportQueue class, following these basic steps:

1. Instantiate a ReportQueue object.

   With this step, establish an ownership relationship between an application window and the ReportQueue object for error reporting and other communication.

2. Connect to a data source.

   Use the ConnectToDB() method to connect to a specific data source.

3. Print, preview, edit or save the report to a file.

   The Print(), Preview(), Edit() and SaveToFile() methods work with a report file. With the Preview() method, you can pass up to ten parameters to the report.

4. Close a report.

   After you are finished working with a particular report, you should Close() it.

Printing a Report

The following example illustrates how to print a single report without parameters. The user selects a report file name from a dialog box, and the program prints the report:

```
CLASS MyShellWindow INHERIT ShellWindow
     HIDDEN oRQ AS ReportQueue
     ...

METHOD Start() CLASS App
     LOCAL oW
     oW := MyShellWindow{}
     oW:Show()
     SELF:Exec()

METHOD Init() CLASS MyShellWindow
     LOCAL oD as OpenDialog
     LOCAL cReportFile AS STRING
     SUPER:Init()

     oRQ := ReportQueue{SELF, "Report Editor" }
     oD := OpenDialog{SELF, "*.RET"}
     oD:Show()
     cReportFile := oD:FileName
     oD:Axit()
     IF cReportFile <> NULL_STRING
          oRQ:Open(cReportFile)
          oRQ:Print()
          oRQ:Close()
     ENDIF
```

Generated Code

The source code generated when you create a report using the Report Editor is as follows:

```
CLASS Employee INHERIT ReportQueue

INSTANCE Employee_file := "C:¥CAV025¥Employee.RET" AS STRING
METHOD Init(oOwner) CLASS Employee
     SUPER:Init(oOwner, "Report Editor")

// The following statement connects to a
// data source.  Remove the data source
// name for data source prompt.
     SELF:ConnectToDB("CA xBase")
     SELF:Open(Employee_file)

// The following statement allows the user to
// preview the report.
     SELF:Preview()
// Use the following statement to print the report.
//   SELF:Print()
// Use the following statement to edit the report.
// SELF:Edit()
```

This code subclasses the ReportQueue class using a class name that is the same as the report entity (in this example, Employee). The Init() method contains code so that the when you instantiate the Employee class, the report is automatically displayed in preview mode. The Init() method can be easily modified to print or edit the report as indicated in the comments.

Customizing the Appearance of the Report Writer

The ReportQueue class has several methods to customize the appearance of the Report Editor while your program uses it, including the ability to hide the window, toggle the window between an icon and full screen, and change the size and location of the window. The easiest way to customize the report window in your programs is to create a new class that inherits from ReportQueue and change the methods of the new class to meet your needs.

In the example below, when a report is previewed, it is automatically maximized:

```
CLASS HideReportQueue INHERIT ReportQueue

METHOD Init(oOwner, cServer) CLASS HideReportQueue
     SUPER:Init(oOwner, cServer) // Start server

METHOD Preview(aParams) CLASS HideReportQueue
     SUPER:Preview(aParams)
     SELF:Show(SHOWZOOMED)
```

Other ReportQueue Methods

The ReportQueue class has several methods that are not mentioned in this chapter, including ACCESS methods designed to give you information about the status of reports. In addition to these, there is a method of the AppWindow class, ReportNotification(), designed specifically for communicating report status between the ReportQueue object and its owner window. See the ReportQueue Class in the online help system for more information about these and other methods of the ReportQueue class.

The GUI Classes

The GUI Classes library provides a full range of print facilities to allow your application to select a printer, set its options, print text and graphics, and handle exceptions as they occur.

The Printer Class

Because it derives from the Window class, the Printer class allows you to treat the printer like the canvas of a window. This inheritance relationship enables you to draw diagrams and text on the printer using all the drawing features of the Window class. For example, the drawing objects described in Chapter 12, "Other Features of the GUI Classes," work on the printer, and you can print using the TextPrint() method at points specified in printer coordinates.

For most business reports, you will find it easier to use the ReportQueue class described earlier in this chapter. The ReportQueue class prints via the Report Editor which provides many of the features you expect of a word processor. The Printer class, by contrast, works in terms of the dots that the printer prints. For example, the method calls:

```
MoveTo(Point{50, 50})
LineTo(Point{100, 100})
```

would draw a diagonal line on a window. You specify the start and end points in canvas coordinates. On a printer, these calls would also draw a diagonal line, but the coordinates are in terms of dots. Therefore, a high-density printer (for example, 1200 dpi) draws a shorter line than a low-density printer (for example, 300 dpi).

Thus, the Printer class is intended for very low-level operations.

Starting the Print Job

To begin a print job, you must first construct a Printer object:

```
oPrinter := Printer{}
```

Then, after using the IsValid() method to test for a valid printer object, you specify a range of pages to print using the Start() method:

```
IF oPrinter:IsValid()
    oPrinter:Start(Range{5, 15})
ENDIF
```

You can print multiple ranges simply by calling Start() a number of times.

If you do not specify a range, the printer will continue to ask the program for new pages until you return FALSE from the PrinterExpose() method (discussed below). The page numbers have no significance except that when the print queue is ready to accept a new page, Visual Objects sends a PrinterExpose event to the Printer object, which has a property called PageNo that you can use to keep track of which page you are currently handling.

When you are finished printing, destroy the printer object as follows:

```
oPrinter:Destroy()
```

Handling PrinterExpose Events

Apart from creating and destroying the Printer object and calling Start() at least once, the only thing you need to do is write the PrinterExpose() event handler for the PrinterExpose event. This event and its handler are closely analogous to the Expose event that a window gets when its canvas area has just become exposed. The only difference is that you get exactly one event per page.

The PrinterExposeEvent class (which derives from the Event class) encapsulates PrinterExpose events generated during a print job. Besides the PageNo property mentioned earlier, it has an ExposedArea property to determine the size of the page to be printed, returning the rectangle for a full page.

You signal Visual Objects that you are finished printing a range of pages by returning FALSE from the PrinterExpose() handler.

Handling PrinterError Events

If there is a problem with the print job, Visual Objects sends a PrinterError event to the PrinterError() event handler.

The PrinterErrorEvent class (which derives from the Event class) encapsulates PrinterError events generated during a print job. Through its property, ErrorType, it indicates the type of error that occurred (for example, insufficient memory or no disk space).

The sending of a PrinterError event does not automatically terminate the print job (unless a fatal error is encountered), but instead prompts the user to abort or retry printing.

Modern printer drivers cope with common problems like the printer going off line or running out of paper, so there is seldom a need for you to write any code for the PrinterError() event handler.

Changing the Default Printer and Settings

Regardless of which printing technique you use, all print jobs are directed to the default Windows or Windows NT printer.

Report Editor

When working with Visual Objects reports using the Report Editor, the user controls the printer with the File Print Setup menu command. This menu command presents a standard dialog box from which the user can select and reconfigure any available printer.

GUI Classes

When printing with the Printer class, you can override the default printer using the PrintingDevice class. For example, you might specify:

```
oPrinter := PrintingDevice{"Postscript printer, PSCRIPT, LPT1"}
```

The PrintingDevice:Setup() method allows you to reconfigure the printer settings using a standard dialog box.

Print Jobs and the Printers Folder

All printing in a Visual Objects application is handled via the Windows Print Manager.

Report Editor

Each ReportQueue Print() method (and Preview() method in which the user elects to print) is treated as a separate print job.

GUI Classes

When printing using the Printer class, each instance of the class represents a single print job. The program connects to the Print Manager when you call the Start() method and breaks the connection when you destroy the Printer object.

Chapter

14 Error and Exception Handling

Exception handling is an important and difficult consideration in any application—but never more so than in GUI applications running in multi-tasking environments. GUI applications are difficult to manage because the flow of control is complicated, because the entire technical framework is rather fragile, and because there may be multiple applications running at the same time, increasing the risk of something going wrong.

CA-Clipper introduced structured exception handling with replaceable error handling routines and nested program structures with integral recovery routines. Visual Objects builds on this technology, removing some limits and integrating it closely into the function and class libraries for database and GUI support.

This chapter discusses the requirements for making an application robust and limiting the impact of exception conditions, and the reasons why traditional thinking needs to be extended to accommodate the complex structures of modern applications. It then describes the solution for exception handling in Visual Objects from two independent perspectives: how the technical underpinnings of the exception handling system work, and how the library functions and classes utilize this technology.

The built-in functions, classes, and methods of Visual Objects provide default exception handling and do a thorough job of protecting the application from programming errors, database corruption, file access conflicts, and resource limitations. The everyday developer can ignore the issue, secure in the knowledge that the built-in recovery mechanisms are robust enough for most circumstances. But any developer who is interested in extending and customizing the exception handling system needs to understand its principles of operation.

Exception Handling in GUI Applications

Before beginning this discussion, it is important that you understand the difference between an exception condition and an error. The distinction is subtle but important: an *exception condition* is something unusual that deserves to be handled outside the regular control flow of the program, while an *error* is something that has gone wrong. An error is an exception that has not been handled correctly.

At a low level, an error condition, such as divide by zero, out of memory, or resource not found, can be raised. A low-level error handler converts this into an exception condition that is propagated and resolved in an orderly manner. Your objective should be to prepare for all exceptions and, by handling them correctly, never allow them to turn into errors that are propagated up into the body of the application.

In traditional, DOS-level Xbase programming the main source of error conditions was the database or other I/O systems. Thus, traditional techniques for exception handling focused on dealing with such error conditions. However, in GUI environments, and especially under Windows, the GUI part is more likely to cause trouble than the database. The environment is quite fragile, and error conditions can easily propagate to harm other applications or even bring down the entire operating system.

In particular, event handling under Windows is quite sensitive. Note that a Windows application has no life except in event handlers: all activity in the application occurs in response to an event passed on from the operating system. Therefore, error conditions in the application code can easily cause serious trouble: if an event is not responded to correctly, the entire Windows structure can be subverted, jamming the message queue and causing the system to freeze. A conscientious developer does not want to be responsible for such calamities.

Objectives

Developers writing directly to the Windows Software Development Kit (SDK), whether using Visual Objects or a lower-level programming language like C, must be very careful with the management of error conditions and event handling. However, when using the GUI Classes library that defines concepts and actions on a higher level of abstraction, the system automatically installs extensive protection layers that reduce the risk that an application error, or other errors such as database sharing conflicts, will crash the operating system.

The GUI Classes library's error handler uses the standard Visual Objects error and exception handling system very meticulously with the aim of meeting these objectives:

- Make the application robust, limiting the propagation of errors lest they bring down the GUI environment or the operating system.
- Place the handling of the errors in the correct place, ensuring that the entity chartered with responding to an error condition has enough information and authority to take whatever action is necessary.
- Display meaningful information whenever the end user is asked to make a decision about how to handle the error.

The GUI Classes library is cited throughout this chapter as a model example of an error and exception handling system built within the Visual Objects framework. These are objectives that you may also want to strive for when designing an error and exception handling system in other class libraries.

The Right Level

The essence of robust exception handling lies in assigning it to the right component. A low-level function does not have enough information to make a reasonable decision nor does it have the authority to abort a major component—such an inversion of authority can wreak havoc with the fragile threads of control in a GUI system. On the other hand, raising the exception level too high is not useful either: at the highest level the application does not know what kind of actions are possible and what structures need to be repaired after the error.

Nor is it reasonable to abdicate responsibility, handing technical problems off to the ultimate authority, the end user. And if you have to involve the user, the message should be expressed in a meaningful way. "FUNCTION _P2OBJECT: General Protection Fault, Ignore, or Abort?" is not a very helpful question—how is a user supposed to know whether the error can be ignored? If an exception is raised because of a record locking conflict during a Delete Record operation, the message should read "Failure During Delete Operation, Record is locked by another user; Try Again, Ignore, or Exit Program?" In old-style programming, a certain amount of roughness was tolerated, but under Windows you talk politely to the user.

Structured Exception Handling

A structured exception handling system allows you to define a code structure that registers a piece of exception handling logic, allowing it to be identified and chartered with handling an exception at the appropriate level.

The SEQUENCE construct is specifically designed for structured exception handling. Described in detail later in this chapter, it lets any program entity, such as a function:

```
FUNCTION DeleteRecord()
     BEGIN SEQUENCE
          DBDelete()
          InfoMessage("Record deleted")
     RECOVER USING oError
          ErrorMessage(oError:ErrorMessage)
     END SEQUENCE
```

or a method of a class:

```
METHOD DeleteRecord() CLASS DataWindow
     BEGIN SEQUENCE
          oDBServer:Delete()
          InfoMessage("Record deleted")
     RECOVER USING oError
          ErrorMessage(oError:ErrorMessage)
     END SEQUENCE
```

be provided with an integral exception handler. It is possible to insert several SEQUENCE constructs within one entity. This allows you to associate individual recovery strategies with different action sequences, providing tailor-made handling of the exceptions that can occur in various circumstances.

Problem Escalation

If the RECOVER clause cannot handle the problem, it is reasonable to escalate the problem to the next higher level:

```
METHOD DeleteRecord() CLASS DataWindow
     BEGIN SEQUENCE
          oDBServer:Delete()
          InfoMessage("Record deleted")
     RECOVER USING oError
          IF oError:Severity = ES_WARNING
               WarningMessage(oError:ErrorMessage)
          ELSE
               EscalateException(oError)
          ENDIF
     END SEQUENCE
```

But this begs the question: what does "EscalateException" mean? To whom do you escalate? This is the essence of proper management of exception handling, and the solutions provided here are aimed at finding the proper authority for every exception.

Frame-Based Exception Handling

Several SEQUENCE constructs can be nested. In particular, when entities (such as methods and functions) call each other in a chain, it is common that each entity provides its own exception handler.

This kind of structure, increasingly common in professional development systems today, is often referred to as a *frame-based* exception handling system. As the different entities call each other, each places a *call-frame* on the stack; when a low-level function discovers a problem it cannot handle on its own and raises an exception, the system searches up the call stack until it finds the closest exception handler.

In a classical, procedural application the call stack represents increasing levels of information and authority: going up the call stack you will eventually find somebody capable of handling the error. Thus, you can escalate simply by issuing a BREAK statement. However, this traditional frame-based exception handling scheme is not well suited to the internal structure of an event-driven GUI application.

Structure of Event-Driven GUI Applications

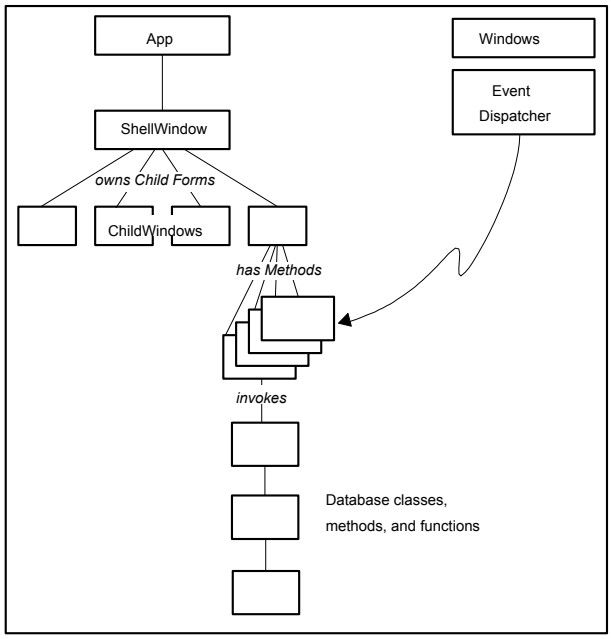
In an event-driven system, the call stack is often quite short and uninteresting. The application's higher levels do not call the lower functions directly—they yield control to the GUI framework, which, in turn, dispatches events to the various action routines (the *event handlers*).

Thus, going up from a low-level database function, for example, you will indeed find the routine that originated the action. However, if that routine fails to handle the exception and wants to escalate up the call stack, all you find is the event dispatcher, which does not know anything about the logic of the application.

It is true that if the exception situation cannot be resolved satisfactorily, you must escalate to the event dispatcher if only to allow it to repair the message queue. Frame-based exception handling remains useful from a technical perspective, in that it prevents an exception from turning into an error that brings down the system.

However, from the perspective of application logic, it is quite useless. You must locate the appropriate authority elsewhere, based on the *logical* relationships of the application rather than the *technical* relationship represented by the call stack.

Escalating to the event dispatcher is a last resort, a request for damage control after the exception has turned into an unmanaged error.



Object-Oriented Exception Handling

In an object-oriented system, it often makes sense to define a general escalation exception handler for an object. While the first line of defense for a method is the RECOVER clause of its SEQUENCE construct, when a method wants help dealing with a difficult problem it is reasonable that it should first escalate to the local exception handler of the object.

Note, however, that this relationship is not reflected in the call stack. The relationship between a method and its object is fundamental and is a good way to organize standardized exception handling for an object. It is different from traditional frame-based exception handling.

Ownership-Based Escalation

In the simple example of the Delete() method above, many errors are non-critical and can be handled adequately by the Delete() method itself. In this context, non-critical does not mean that the exception is not important: if the record cannot be deleted, you may have a serious business problem. Instead, it means that the exception is not structurally critical to the life of the application, and its effects can easily be contained and prevented from damaging other information. In the simple example above, you merely informed the end user of the conflict and went about your business.

Structurally critical exceptions, however, must be handled more gingerly. For example, consider some of the conditions that can render a data window altogether useless: the database is missing or corrupted, the database has been changed and no longer matches the data window, or the resource that defines the layout is missing. In these cases, the data window is not salvageable, but who should take responsibility for humanely destroying it?

The data window was created and instructed to perform certain tasks. Even if it fails at this task, it does not have the authority to unilaterally destroy itself because it does not know what higher-level structures depend on it (more formally, this would be an encapsulation error). The entity that created the data window, its *owner*, is best suited to destroy it and contain the damage that might be caused by the error. Thus, the proper traffic flow in the handling of this exception looks like this:

1. The method is notified of the exception.
2. The method handles the exception, if possible.
3. If not, the method escalates the exception up to the owner of the window.
4. The owner decides to terminate the subordinate, sending it a Close message.
5. The subordinate responds to the Close message in an orderly manner, terminating *its* subordinates and then itself.
6. The owner cleans up any relationships left dangling by the subordinate.

Of course, if the exception was really serious (such as a catastrophic shortage of resources) this orderly shutdown may not be possible and attempting it can generate another exception. In that case, the exception handlers can give up and escalate to the event dispatcher and the application framework, shutting down the application.

This logic is the default exception handling provided in the GUI and RDD class libraries. As always, the developer can customize the standard behavior, giving individual data window subclasses the specific behavior that suits the application needs. Both the decision of when to escalate an exception and when to handle it locally, and the specific actions that need to be taken to recover, can be tailored to the specific needs of the application. At any time, a simple emergency escalation is only a BREAK away.

Cleaning Up

As noted above, when one object escalates an exception to another it is important for the owner to shut down the subordinate in an orderly manner. This includes properly terminating the subordinate (which can be done using a Close() method) and possibly performing some additional clean up tasks. The subordinate object can do some clean up of its own at this point before shutting itself down. But, what if the recovery from a particular exception is handled by breaking instead of escalating? The RECOVER area may need to do some cleaning up in these cases.

RECOVER Area

If your application creates resources that are beyond the ken of the garbage collector, such as opening files or databases, directly manipulating work areas, allocating memory, or grabbing Windows resources (for example, windows, menus, icons, strings), you have to remember to restore these in the RECOVER area. For example, inside the Init() method of the DBServer class the system selects a new work area and restores the current work area after itself. This must be done in the RECOVER section as well:

```
LOCAL wCurrentWorkArea AS WORD

...

// Remember old work area
wCurrentWorkArea := VODBGetSelect()
BEGIN SEQUENCE
     // Get a new work area
     DBUseArea(TRUE, ...)
     wWorkArea := VODBGetSelect()

     ...

     // Restore work area
     VODBSetSelect(wCurrentWorkArea)
RECOVER USING oError
     // Restore work area
     VODBSetSelect(wCurrentWorkArea)
     BREAK oError
END SEQUENCE
```

If you use the standard Visual Object classes, you do not have to worry about this because the classes are built to take care of these problems automatically, as illustrated above for the DBServer class. However, system-level programmers who are building subsystems like these should be aware of the consideration.

Axit() Method

You can also have clean-up code for the deallocation of certain resources as part of an Axit() method that is automatically invoked by the garbage collector (and possibly manually invoked by the object's owner just prior to shut down) when an object is destroyed. See the Axit() Method discussion in Chapter 25, "Objects, Classes, and Methods" later in this guide for more details on allocation and deallocation of resources.

Low-Level Exception Handling

Many exception conditions are raised at the lowest levels of the program. Some, such as addressing violations and zero divides, are raised by the operating system; the Visual Objects runtime system intercepts such events and propagates them into its internal exception system. Others, such as database locking conflicts are detected by the database support library, whether an Xbase RDD or the ODBC system, and reported back into the internal exception system.

In many cases, handling of such exceptions can be attempted at the lowest level. For example, after a lock failure the system might reasonably attempt a few retries before giving up.

Thus, the life cycle of an exception begins with the original condition, raised by the operating system or a low-level support routine, being fed to the low-level exception handler together with information about the original circumstances of the exception. The low-level exception handler may decide to attempt to handle the exception and recover, or it may raise an exception condition and send it up to the structured exception management system discussed above.

Note that when using the GUI classes, the default low-level exception handler does not prompt the end user for Abort or Cancel permission. A low-level exception handler does not have enough information to give the end user meaningful information, and it does not have the authority to abort anything (except in the case of catastrophic errors).

Installable Exception Handlers

At all levels, Visual Objects allows the installation of exception handlers, through the SEQUENCE construct, the Error handling methods, and the ErrorBlock() registration facility. The flexibility and power of the Visual Objects object management system also allows the extension of these systems, and the GUI Classes library's structured exception handling indeed uses these facilities to leverage off the default exception handlers.

Exception Handling Architecture: A Summary

These various approaches have different benefits, and work well together. The relationship is quite simple and logical, and proceeds in several steps:

1. The low-level exception handler intercepts an error condition raised by the operating system or a runtime support routine. If the error can be handled on this level, it will be. A record lock failure, for example, can be handled by several retries before giving up.
2. If not, the exception escalates to the frame-based array of registered exception handlers. Each exception handler can attempt to handle the problem or escalate it to a higher authority. If any exception handler can solve the problem, it will. If the application is organized purely hierarchically and procedurally, eventually you reach the top of the chain.
3. If the application is object-oriented, a frame-based exception handler along the chain will escalate to the exception event handler associated with the object.
4. If the object fails to resolve the exception, it will escalate to its owner, and so on, up the ownership chain. Eventually, you reach the highest level of authority in the ownership chain, the application.
5. If at any time in the object-oriented resolution process a catastrophic error occurs or the exception event handlers fail for any other reason, the exception is thrown back to the frame-based chain as a last resort. This will lead to a technical resolution, but not necessarily to a resolution of the situation from a business viewpoint.
6. In any case, the exception is prevented from propagating up and bringing down the operating environment.

Language Mechanisms

Some of the language mechanisms for error and exception handling have been mentioned in the previous discussion but without much detail. This section describes the specific language constructs and data structures designed to build error and exception handling systems.

In Visual Objects, there are two mechanisms for processing exceptions, the SEQUENCE construct and a posted code block (the *error block*). With these mechanisms as tools, you can build different approaches to exception handling.

The mechanisms differ in the location of exception handling code and how it is called. In the SEQUENCE construct, the code is inline (part of the RAM image of the loaded application) and is called by issuing a BREAK statement. In the error block, it is stored in memory (in the code block) and is triggered automatically or by executing the Eval() function.

In general, use SEQUENCE for handling exceptions (such as database does not match data window, resource file not available, or database does not exist) and the error block for handling generic, low-level errors (such as device not ready, disk full, or stack underflow) that usually deal with the computer and could arise across applications.

The SEQUENCE Construct

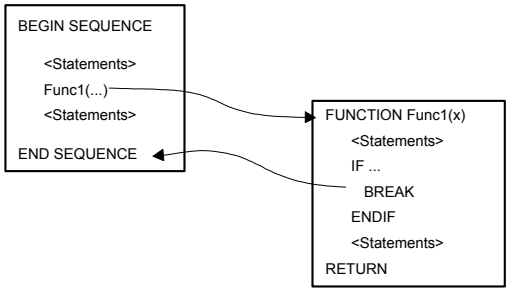
BEGIN SEQUENCE is a control structure not that different from DO WHILE. In the simplest case, this structure:

```
BEGIN SEQUENCE
     <Statements>
     IF !ExceptionFlag
          BREAK
     ENDIF
     <Statements>
END SEQUENCE
```

is essentially identical to:

```
DO WHILE TRUE
     <Statements>
     IF !ExceptionFlag
          EXIT
     ENDIF
     <Statements>
ENDDO
```

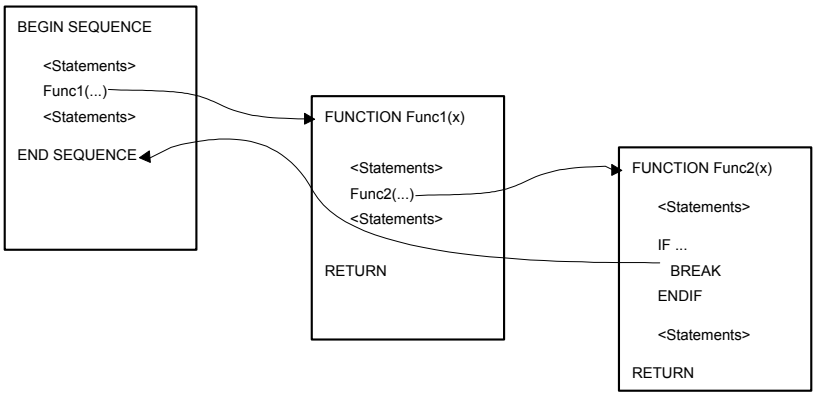
In both cases, control enters the structure and, under normal circumstances, continues through the body and exits at the end. If the BREAK statement (or EXIT in the DO WHILE case) is executed, control jumps out of the structure.



However, there is one important distinction: the WHILE structure is a syntactic construct that is handled by the compiler and must, therefore, be contained all within one entity, but the SEQUENCE structure is a semantic construct that is handled by the runtime system and is allowed to span entities.

Specifically, not only the DO WHILE and ENDDO statements, but also the EXIT statement, must be in the same entity. Thus, if code within a DO WHILE loop calls a function, that function cannot EXIT the loop.

Both BEGIN SEQUENCE and END SEQUENCE must also be in the same entity; however, because the SEQUENCE construct is handled at runtime, BREAK can be anywhere. The code within the SEQUENCE construct can call a function that calls another function, and so on. Then, if deep down in the call tree you discover a problem, you can execute the BREAK statement to jump all the way out to the end of the sequence. The function with the BREAK statement and all intervening functions are terminated, all local and private variables are removed, everything is collapsed, and you end up at the END SEQUENCE statement.



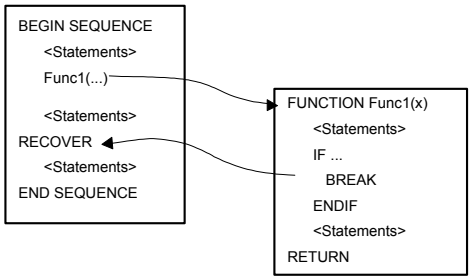
Aside from errors, there can be other exception conditions in which an application wants to terminate processing across several scopes, and the SEQUENCE construct makes this convenient. Without such a runtime construct, the intervening routines would all have to check and propagate return codes, which is cumbersome and also burdens the intervening routines with code that is rarely executed.

RECOVER

In the standard structure, BREAK causes a jump to the END SEQUENCE statement. However, by inserting a RECOVER statement into the structure, you can intercept the BREAK and do some special processing:

```
BEGIN SEQUENCE
     <Statements>
     IF !ExceptionFlag
          BREAK
     ENDIF
     <Statements>
RECOVER
     <Statements>
END SEQUENCE
```

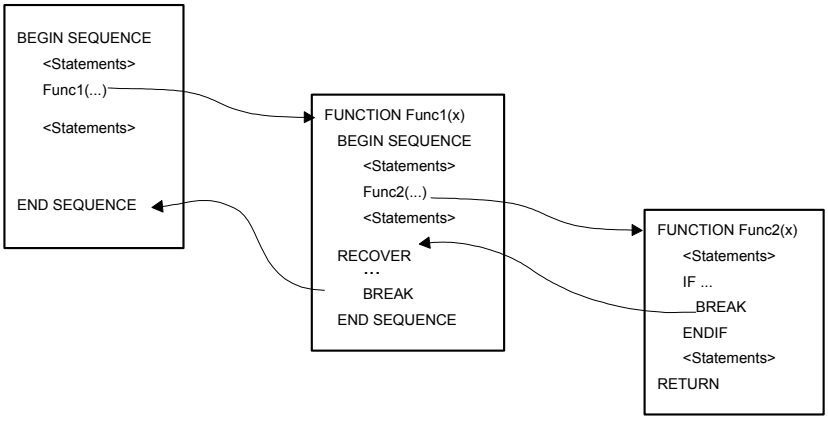
Here the BREAK causes the statements after the RECOVER to be executed. In the case of normal processing (without a BREAK), these statements would not be processed.



The statements after RECOVER are used to handle the situation that caused the BREAK. This is where you put your exception handling code.

Nested SEQUENCE Constructs

You can nest SEQUENCE constructs to accomplish whatever level of exception handling you require in your application. If the RECOVER code determines that it cannot handle a particular situation or if another error occurs, it can issue another BREAK. In this case, control jumps out to the next nested construct—to the RECOVER or END SEQUENCE statement, depending on how it is constructed.



A common practice is to use a CASE construct within the RECOVER code to test for exception conditions and deal with each one appropriately. By carefully coding each CASE that can be handled at the current level, you can delegate to the next level all exceptions that cannot be handled at the current level by putting the BREAK statement as the OTHERWISE case:

```
BEGIN SEQUENCE
     <Statements>
     IF !ExceptionFlag
          BREAK
     ENDIF
     <Statements>
RECOVER
     DO CASE
     CASE <ExceptionOne>
          <Statements>
     CASE <ExceptionTwo>
          <Statements>
     CASE <ExceptionThree>
          <Statements>
     ...
     OTHERWISE
          BREAK
     ENDCASE
END SEQUENCE
```

Using this technique, it is easy to handle all exceptions at the right level.

BREAK Value and RECOVER USING Variable

Since the transfer of control from the BREAK statement to the RECOVER statement can be very far, it is difficult to communicate information about what motivated the break. In the case of errors and other exception situations, for example, it is important that the logic responding to the situation can find out exactly what happened, since different situations require different responses.

The BREAK statement allows the specification of a value to be transferred to the RECOVER statement via the USING clause. The USING clause expects an untyped, USUAL value, so it can be used to transfer anything; however, in the case of error handling, the value that is passed is, by convention, an Error object:

```
LOCAL oError
BEGIN SEQUENCE
     <Statements>
     IF !ExceptionFlag
          oError := Error{}
          BREAK oError
     ENDIF
     <Statements>
RECOVER USING oError
     // Inspect oError and determine action
END SEQUENCE
```

The variable that is used to receive the error information object, *oError*, must be declared as a LOCAL untyped (USUAL) variable.

Abuse of the SEQUENCE Construct

As is obvious from the above discussion, there is nothing inherent in the SEQUENCE construct that limits its use to error situations. However, the construct should not be abused as a cross-entity go to statement. It is otherwise very confusing for a developer looking at some code to understand why the straightforward control flow is not followed:

```
FUNCTION Abuse1()
     Abuse2(100)

FUNCTION Abuse2(x)
     Abuse3()
     ? x
```

When this is executed, it appears from the code that Abuse2() will always print the value of its parameter, in this case 100, but if Abuse3() does a BREAK to a SEQUENCE construct outside of the code shown, the print statement will not be executed. This is terribly confusing and should only occur in rare and troubled circumstances.

The Error Object

As mentioned earlier, the Error object is typically created within a SEQUENCE construct in response to some exception condition. It is then passed to the RECOVER USING statement which inspects the object and decides what to do.

The Error class is specifically designed with several instance variables that are used to describe the situation that generated the exception (see the online help system for information about the Error class and its instance variables). Thus, before you BREAK to the RECOVER code, you would instantiate an Error object and initialize as many of its instance variables as necessary to convey the problem:

```
BEGIN SEQUENCE
     <Statements>
     IF lExceptionFlag
          oError := Error{}
          // Initialize oError instance variables
          BREAK oError
     ENDIF
     <Statements>
RECOVER USING oError
     // Inspect oError contents and determine action
END SEQUENCE
```

The role of the Error object is that of go-between for the code in which the exception is raised (where the BREAK statement is) and the exception-handling code (where the RECOVER statement is). Although it starts out passing information from the BREAK code to the RECOVER code, the Error object can also serve as a two-way dialog if the RECOVER code decides, for example, to retry the operation.

Error objects are also the means of communication in low-level error handling with the error block. Visual Objects creates and sends an Error object to the default error handler when an error condition is raised. See The Error Block section later in this chapter for more information on this subject.

Subclassing the Error Object

It is sometimes useful to subclass the Error class in order to provide special error handling for a particular class of objects:

```
CLASS MyError INHERIT Error
     EXPORT MyReturnCode      AS WORD
     EXPORT MyState                    AS STRING
     EXPORT ErrorMessage               AS STRING
```

Subclassing provides a simple vehicle for an error handler to determine if it wants to deal with the error or not. Instead of checking for specific return codes, it can decide what to do based on the class of the Error object:

```
FUNCTION MyErrorHandler(oError)
     // This error handling function processes your
     // errors only and passes all others back up.

     IF IsInstanceOf(oError, #MyError)
          // Handle this error
     ELSE
          RETURN Eval(cbOldErrorBlock, oError)
     ENDIF
```

Various Visual Objects subsystems subclass the Error class in just this manner. These are, however, declared as STATIC and are, therefore, not available for general usage and do not show up in the IDE.

The Error Block

In Visual Objects, there is a code block (called an *error block*) that is automatically called whenever a library function detects an error condition. There is a default error handler built into the system, but any application or library can register a new one by invoking the ErrorBlock() function. (This was hinted at but not explained in the example above.)

In most cases (including the default case), the code block simply calls a function (the *error handler*) that does the real work, since a code block is limited in the complexity of the code structures it supports. Of course, it is possible to do other types of processing within the error block itself, but the setup in which the error block calls the error handler is the one most often used.

To allow an application entity to clean up after itself and restore the original error block, ErrorBlock() follows the standard convention of returning the current error block before registering the new one. Thus, a function that wants to register an error block temporarily might be coded like this:

```
FUNCTION Func1(x)
     LOCAL cbOldErrorBlock AS CODEBLOCK

     // Register a new error handler,
     // remembering the old one
     cbOldErrorBlock := ErrorBlock({|oError| MyErrorHandler(oError)})

     // Do something that might fail
     <Statements>

     // Restore old error handler
     ErrorBlock(cbOldErrorBlock)

FUNCTION MyErrorHandler(oError)
     // This is the actual error handling function to
     // which control is redirected by the error
     // handling code block.
     // It is passed the Error object and should
     // respond appropriately.
```

The error block is automatically invoked when a system-level error is detected. An Error object (built by the system) containing information about the error is passed as a parameter to the error block which, in turn, passes it along to the error handling function.

Of course, you can also invoke the error block for errors detected within your application by calling ErrorBlock() without a parameter:

```
Eval(ErrorBlock(), oError)
```

This, however, is not the usual or, in most cases, the recommended practice. You will most often handle exceptions at the application level using the SEQUENCE construct and use the error block for handling low-level errors.

Using a Hierarchy of Error Handlers

An error handling routine could inspect the Error object it is passed, decide if this is an error that it is capable of handling, and otherwise pass it on to the default error handler. (Actually, it passes the error back to the previous error handler—there is no way of knowing if it was the default or was installed by a library or another application.)

In order to do this, you must make the previous error handler available to the new error handler so, instead of declaring it as a LOCAL variable you declare it as GLOBAL:

```
GLOBAL cbOldErrorBlock AS CODEBLOCK

FUNCTION Func1(x)
     // Register a new error handler,
     // remembering the old one
     cbOldErrorBlock := ErrorBlock({|oError| MyErrorHandler(oError)})
     // Do something that might fail
     <Statements>
     // Restore old error handler
     ErrorBlock(cbOldErrorBlock)
     RETURN SomeValue

FUNCTION MyErrorHandler(oError)
     // This is the actual error handling function to
     // which control is redirected by the error
     // handling code block.  It is passed the Error
     // object and should respond appropriately.
     IF oError:GenCode = MySpecialCode
          // Handle special case error
     ELSE
          // Pass all others back
          RETURN Eval(cbOldErrorBlock, oError)
     ENDIF
```

As long as every installed error handler does this, you will have a correctly installed hierarchy or *stack* of error handlers. Each one can handle the errors it wants to handle and pass the others back. Eventually, the default error handler processes anything that has leaked through the entire chain of error handlers.

Installing Error Handlers in Libraries

It was mentioned earlier that you could register an error block in a library, but it is not obvious how you would go about this. A library is a group of resources, usually functions or class definitions, that is linked into an application. The application, in turn, simply uses the resources it needs from the library. Thus, it would seem that you would have to inform the users of your library to put some code in their Start() routine to register your error handler, an undesirable situation for you and your library users.

To avoid this situation, create a special library procedure using the _INIT2 keyword to register your error handler. Procedures defined in this manner are automatically executed by the system at startup:

```
GLOBAL cbOldErrorBlock AS CODEBLOCK

PROCEDURE First_Proc() _INIT2
     cbOldErrorBlock := ErrorBlock({|oError| LibErrorHandler(oError)})

FUNCTION LibErrorHandler(oError)
     // This is the actual error handling function to
     // which control is redirected by the error
     // handling code block.
     // It is passed the Error object and should
     // respond appropriately.
     IF oError:GenCode = MySpecialCode
          // Handle special case error
     ELSE
          // Pass all others back
          RETURN Eval(cbOldErrorBlock, oError)
     ENDIF
```

In fact, this is exactly how the default error handler is installed in the System Library with the ErrorSys() procedure.

Return Values

If you decide to write and install your own error handler, there are some rules about what the error handler should return:

- If the function that raised the error condition sets CanRetry to TRUE, indicating that it is prepared to try again, then the error handler should respond with a logical indicating whether the retry should be attempted.
- If the code that raised the error condition sets CanSubstitute and provides a default value, the error handler should normally return this default value.

If you want to study the source code for the default error handler to get some ideas, see the file ERRORSYS.PRG located in your Visual Objects \SAMPLES directory.

Chapter

15 File Handling

Many of the applications that you write will be centered around the use of one or more database and ancillary files. You may also have occasion to deal with creating and reading text files as well as low-level binary files. This chapter deals with the issue of handling files in your applications.

Naming Conventions

Visual Objects takes advantage of the Windows or Windows NT file handling features that allow you to use long file names, universal naming convention (UNC) names and support for mixed case matching.

Long File Names

Support for long file names has been extended to 255 characters from the default DOS 8.3 limitation. If configured correctly, Windows NT Server also supports the use of long file names.

UNC Names

UNC is a standard naming convention that allows you to reference network servers and shared directories. Visual Objects support for UNC names allows you to refer to network servers without having to map to a drive letter. The syntax for UNC is as follows:

```
¥¥server¥sys[¥path]
```

Mixed Case

When doing a file search, Visual Objects first tries to match the case of the file name exactly. If this fails, a search will also be performed for case insensitive matches for the file name.

The Defaults

Anytime you create or open a file, you have to be concerned about its disk drive and directory location. In particular, you need to understand the defaults used by the system to determine where files are created and located and how to change and override these defaults.

- If you do nothing special, the following rules, established by Windows, will be used for *unqualified file names* (that is, file names without explicit path specifications):
  - The application will create and search for files using the current drive and directory. At application startup, this will be the *working*, or *startup*, directory as specified in the Properties dialog box using the Window's startup window (the default is the directory containing the application's main .EXE file).
  - If attempting to locate a file that cannot be found in the current drive and directory, the application will search the DOS PATH setting.
- If you use SetDefault() to explicitly specify a default drive and directory, the application will locate and create unqualified file names using this new location, completely ignoring the Windows rules defined above.

In either of these two situations, you can specify additional places to locate a file, called a *search path*, using SetPath(). If you do this and the file is not found, the application will search each additional directory in the search path before giving up with a file not found condition. SetPath(), however, has no effect on where new files are created.

These rules apply to most files, including database files, index files, label and report form files, and text files. There are, however, some exceptions:

- Memo files and OLE Document files are always created (and must remain) in the same location as their corresponding database files.
- The low-level file operations discussed later in this chapter and certain other operations (all explicitly documented in the appropriate reference guide) ignore the SetDefault(), SetPath(), and DOS PATH settings.
- The search rules for the Windows help files are determined by Windows rather than your application. For unqualified file names, the underlying application (for example, WINHELP.EXE) searches for files using the current drive and directory, then searches the DOS PATH—both SetDefault() and SetPath() are ignored.
- Visual Objects Report Editor files hard code the path for associated database and index files in the .RET file. You can change the path using the Report Editor's Edit Query menu command.

Runtime Configuration

To override these default search rules, you can specify a path as part of the file name:

```
DBServer{"c:¥data¥customer data¥customer"}
```

However, you want to avoid having code that is dependent on a particular configuration because installation and maintenance can become problematic. In the worst case, you could have fully qualified file names hard coded throughout your application, and you would need a different set of source code for each installation configuration (or, even worse, to dictate a particular configuration to all of your clients).

A better solution would be to use unqualified file names in your source code wherever possible and allow the end user to configure the default (for example, using an initialization file or environment variable).

Environment Variables

The following example illustrates how you might set the default drive and directory using an environment variable:

```
METHOD Start() CLASS App
    ...
    SetDefault(GetEnv("VODIR"))
    ...
```

Then, as part of your installation procedure you would place a line similar to the following in the user's AUTOEXEC.BAT file:

```
SET VODIR = c:¥data¥cust¥
```

Initialization Files

The use of initialization (.INI) files is supported for backward compatibility with Windows 3.1. You are encouraged to use the system registry, a central repository used to store the configuration information for Windows or Windows NT. For more information about the registry, see Chapter 17, "Operating Environment" in this guide.

Using Windows Defaults

If you rely on Windows to determine the default drive and directory (that is, you do not use SetDefault() in your application), be aware that the end user of the application can change the current drive and/or directory anytime the application presents a file open dialog box by simply choosing a new directory. Therefore, this might not be the best strategy because it forces you to store the defaults before each file open dialog box is displayed and restore them afterwards.

SetDefault(), on the other hand, provides an explicit default setting that is guaranteed throughout the lifetime of the application. It is not subject to the whims of the user and the operating environment. Thus, you may want to consider retrieving the Windows defaults in the application's Start() routine and using them as the SetDefault() and SetPath() arguments. That way, you can have a user-configurable default that is controlled within the Window's startup window but not subject to change while the application is running.

The Visual Objects language provides a complete set of functions for manipulating the current Windows drive and directory, two of which are illustrated below:

```
METHOD Start() CLASS APP
    ...
    SetDefault(DiskName() + ":¥" + CurDir())
    SetPath(GetEnv("PATH"))
    ...
```

Most functions designed to manipulate the Windows default drive and directory begin with "Dir" or "Disk" and can, therefore, be easily located in the online help system.

Generated Source Code

When you generate source code that deals with file names from within the IDE, the basic technique is to store the path or full path as an instance variable that is used to refer to the file. This technique makes the code easy to adapt using the techniques described in this section. Here are some examples of how you can modify the generated code to force file searching in the path designated by SetDefault().

DBServer Editor

The DBServer Editor takes into account the issue of file locations by storing the path name as an instance variable and using this instance variable name throughout the remaining code that it generates. For example:

```
CLASS Sales INHERIT DBServer
    INSTANCE cDBFPath := "c:¥cavo27¥" AS STRING
```

Simply set the instance variable to NULL_STRING to use the SetDefault() directory instead of using the explicit path name.

Report Editor

The code generated by the Report Editor hard codes the full path name as an instance variable that is used to open the report file:

```
CLASS Cust INHERIT ReportQueue
     INSTANCE CUST_FILE := "c:¥cavo27¥cust.ret" AS STRING

METHOD Init() CLASS Cust
     SUPER:Init(oOwner, "Report Editor")
     SELF:ConnectToDB( "CA xBase" )
          SELF:Preview(CUST_FILE)

     ...
```

It was mentioned earlier that SetDefault() does not apply when searching for report files, but there is another function, GetDefault(), that returns the current SetDefault() setting as a string. Using this function, you can make this code more flexible:

```
CLASS Cust INHERIT ReportQueue
     INSTANCE CUST_FILE := "cust.ret" AS STRING

METHOD Init() CLASS Cust
     SUPER:Init(oOwner, "Report Editor")
     SELF:ConnectToDB( "CA xBase" )
     SELF:Preview(GetDefault() + CUST_FILE)
     ...
```

The FileSpec Class

The purpose of the FileSpec class is to help manage file names and directories. It contains the identification of a disk file (its drive, path, file name, and extension), as well as several useful methods.

Note: The FileSpec class is used to collect the information about a file that you will subsequently open in your application. Instantiating an object of the FileSpec class does not attempt to open the file or verify its existence in any way and, therefore, does not use the SetDefault() or SetPath() settings for unqualified file names as discussed earlier.

In a practical example, you might collect file definitions and directories in one place in your application, defining a FileSpec object for each file:

```
oFSCust    := FileSpec{"c:¥data¥cust¥customer.dbf"}
oFSCustIdx := FileSpec{"c:¥data¥cust¥custnum.ntx"}
oFSOrders  := FileSpec{"c:¥data¥cust¥orders.dbf"}
```

Elsewhere in the application, you could use these file specification objects to create data servers or open databases for data windows. (You may use a FileSpec object in place of a file name with any of the DBServer and DataWindow methods):

```
oDBCust := DBServer{oFSCust}
oDBCust:SetIndex(oFSCustIdx)
oDWOrder:Use(oFSOrders)
```

Programming like this, however, would not take you beyond the maintenance and installation problems described earlier.

The Default Directory

A better alternative is to use the Find() method, which provides a particularly useful way of defining and storing path information. You can use it to determine the location of a known file in the application, and base your application's default on that location. For example, the following code places the FileSpec object of a known file, SOMEFILE.DBF, in a global variable called *oFSConfig*:

```
GLOBAL oFSConfig AS OBJECT

METHOD Start() CLASS App
    ...
    oFSConfig := FileSpec{"somefile.dbf"}
    IF !oFSConfig:Find()
        <BREAK or ask user to specify location>
    ENDIF
    ...
```

For this code to work (assuming no SetDefault() or SetPath() has been issued by the application), SOMEFILE.DBF should be located in the current Windows directory. Then, the components of *oFSConfig* can be used with SetDefault() to make your application's default the same as the Windows default. For example:

```
GLOBAL oFSConfig AS OBJECT

METHOD Start() CLASS App
    ...
    oFSConfig := FileSpec{"somefile.dbf"}
    IF !oFSConfig:Find()
        <BREAK or ask user to specify location>
    ELSE
        SetDefault(oFSConfig:Drive + oFSConfig:Path)
    ENDIF
    ...
```

Alternatively, you could omit using SetDefault() as a global application setting and use the virtual variables *oFSConfig:Drive* and *oFSConfig:Path* to explicitly construct other FileSpec objects in the application. Either way, there is no need to include any explicit directory information in the application code or an initialization file. Installation of the application requires only that the files associated with the application be copied to some suitable place that is specified as the startup directory in the Windows startup window.

The FileSpec class, however, can do much more than simply store the path name. For example, it provides several techniques for manipulating the file, such as Copy, Delete, and Rename with any FileSpec object. You can also obtain information beyond the drive and path name, including the date and time stamp and the file size and attributes.

String Manipulation

It also helps avoid the need for string manipulation to construct path names in situations where the application wants to specify drive, directory, and file name separately:

```
oFSCust := FilesSpec{}          // Contains no info yet!
oFSCust:Drive := "¥¥server¥sys"  // UNC name
oFSCust:Path := "¥data¥customer¥"
oFSCust:FileName := "customer data" //long file name
oFSCust:Extension := "dbf"
```

This is similar to the code generated by the DBServer Editor, except that the path name for the file is stored as an instance variable that is used to assign the Path variable:

```
CLASS Customer
INSTANCE cDBFPath := "c:¥data¥customer¥" AS STRING
...
METHOD Init() CLASS Customer
     oFileSpec := FileSpec{"customer data.dbf"}
     oFileSpec:Path := cDBFPath
     SUPER:Init(oFileSpec,,,)
     ...
     oFileSpec := FileSpec{"custnum.ntx"}
     oFileSpec:Path := cDBFPath
     SELF:SetIndex(oFileSpec)
```

Low-Level File Handling

Visual Objects has a set of functions for low-level handling of binary files, all of which begin with the letter "F" (for example, FOpen(), FRead(), FWrite(), and FClose()). These functions are designed to provide operating system level access to files and, therefore, fall outside of the realm of file handling as discussed so far in this chapter.

In particular, the low-level file functions that allow you to specify a file name ignore the SetDefault(), SetPath(), and DOS PATH settings, always using the current Windows default drive and directory for unqualified file names. To override this default, you must qualify the file name in the function call:

```
LOCAL ptrHandle := FOpen("d:¥osfiles¥binfile")
```

If you know that the binary file is located in the SetDefault() path, you can add the default directory to the file name using the GetDefault() function:

```
LOCAL ptrHandle := FOpen(GetDefault() + "binfile")
```

Once the file is open, you manipulate it using a file handle. To read from the file opened above, for example, you would use *ptrHandle* as an argument instead of the actual file name:

```
FRead(ptrHandle, @cReadBuff, 128)
```

Chapter

16

Hyperlabels

A *hyperlabel* is a descriptive object that attaches to another object, and almost every object in the system has one. It contains summary information about its host object, sort of like an adhesive label that you affix to a machine or a vehicle. (The *hyper* part comes from the fact that the GUI classes use it to drive the hypertext help system.)

This chapter describes how the information in an object's hyperlabel is used by the system at runtime.

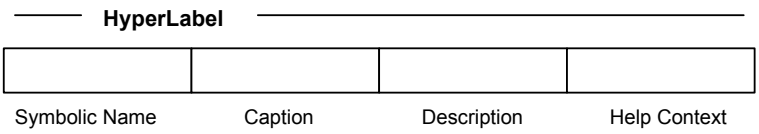
Purposeful Components

As the user learns to drive your application, they inevitably have many questions about the meaning of certain controls and about the model you are using to represent the real world they know. The purpose of hyperlabels is to introduce consistency to the way the user gets information when needed.

If you use the hyperlabel system to its best advantage, you will give each menu item, each control, each window, each database field, and each exception its own hyperlabel.

Hyperlabel Properties

The hyperlabel has four properties, implemented as virtual variables.



Symbolic Name

The first is a *symbolic name*, that the user never sees. The purpose of the symbolic name is to connect the object (to which the hyperlabel is affixed) into the automatic behavior of GUI classes. For example, if you attach a hyperlabel to a menu item, the symbolic name is the name of the method that Visual Objects dispatches when the user clicks that menu item. If you attach a hyperlabel to a control, Visual Objects uses the symbolic name to override the default mapping of control names to field names. Simply put, the symbolic name enables some significant automatic behavior, inside the application.

The other three properties describe the host object to the user, in one word, one line, and one screen, respectively.

Caption

The *Caption* is the one-word name that the user recognizes. For a menu item, the caption is the text of the menu item itself. For a push button, it is the text inside the push button.

Description

The *Description* tells the user the purpose of the hyperlabel's host object—control or menu item, say—in a one-line comment. Whenever the user positions the mouse on the control or menu item, the Visual Objects dispatcher automatically displays the Description on the status bar. What the user wants to know at that moment is "What is the purpose of this control?" Think of the Description as a prompt to the user giving the purpose of the control. If you give every single object in your user interface a *purpose*, the user quickly gets the idea that the application is well thought out, clear, and purposeful.

HelpContext

The *HelpContext* provides a route to the topic in the hypertext system that provides context-sensitive help on the hyperlabel's host object. In the case of Windows, this is the WinHelp system, and the HelpContext is a unique keyword. (Refer to Using an online help system in Chapter 11, "GUI Classes," for more information on implementing context-sensitive help in your application.)

Interaction with Resources

The symbolic name in the hyperlabel doubles as the name of the resource that the hyperlabel's host object links to. If you attach a hyperlabel to a DataWindow, the symbolic name is the same as the name given in the RESOURCE statement that depicts the DataWindow. If you laid out the data window using the Window Editor, the editor generates hyperlabels for you and automatically associates them with the resources that it generates at the same time.

If the hyperlabel attaches to a control, then the symbolic name and the constant that represents the control's resource ID in Windows share some of the same text. For example, the symbolic name might be NEXT, while the constant representing the resource ID might be ORDERSNEXT. That is because the names of the constants must be unique, while the symbolic name is local to the subclass of DataWindow.

The most common use of the symbolic name is to specify a method that Visual Objects dispatches when the user activates the hyperlabel's host object. The Visual Objects code looks something like this:

```
METHOD MenuCommand(oMCE) CLASS DataWindow
    Send(SELF, oMCE:HyperLabel:NameSym)
```

Use by the Status Bar

When the user traverses an item with the mouse, Visual Objects displays the item's purpose on the status bar. The code for a menu item, looks like this:

```
METHOD MenuSelect(oMSE) CLASS ShellWindow
    SELF:StatusBar:Transient := oMSE:HyperLabel:Description
```

Internationalization

There should be only four sources of natural language visible to the end user—databases, reports, resources, and the hyperlabel's Description and Caption properties—thereby limiting the text you must translate in order to internationalize an application.

To translate an application, start by creating a new copy of the application in your repository and new copies of the ancillary files (for example, database, index, report, and help database files) associated with the application. You could either give the ancillary files new names or simply copy them to a new directory using the same names. You would, of course, have to deal with this appropriately, by either changing the applications default search path or changing the file names in the appropriate places in the source code. See Chapter 15, "File Handling," in this guide for more information. After preparing the new application and files, you can begin the translation process.

Source Code

The only information defined in source code that is visible to the user is the Caption and Description associated with each object's hyperlabel and the information in the resource entity that defines the object to the Windows resource compiler. Therefore, these are the only things in the source code that you must translate.

Source code that is generated by the various editors in the IDE uses the Caption and Description properties that you enter to generate the hyperlabels and the resources defining the object . Therefore, the easiest way to translate generated source code is to use the appropriate editor. This insures that making a single change is propagated throughout the generated source code.

Databases

If the application starts out with existing databases containing data that is visible to the end user, the data should be translated. However, if the application only uses the data internally, translation is unnecessary. Similarly, if the application relies on an existing database at the customer's site or if the application starts with an empty database, no translation is necessary. It is never necessary to translate the database structure (for example, the field names), as these are used by the application only and are not visible to the end user.

Reports

Any reports included with the application should also be translated. These changes are limited to the actual report file and are made using the Report Editor.

Help Databases

The text of the help database must be translated, and you must recapture any bitmaps that have changed as a result of the application being translated. Note, however, that the keywords and other internal information (for example, jump links) should not be translated because they are not visible to the user. Leaving the keywords intact is essential, in fact, because the built-in help mechanism in Visual Objects depends on them.

Use by Exceptions

When the standard components of Visual Objects escalate exceptions, they construct an object of the Error class. The Error object includes a hyperlabel that describes the exception that occurred. The Caption, Description, and HelpContext contain the same kind of information as they would for a control or menu item. For example, the HelpContext enables the user to get help on the kind of exception that occurred.

The system uses the hyperlabel if none of the exception handlers in the ownership chain was able to deal with the exception. The symbolic name is the name of a method of the App class that represents the last ditch attempt to handle that type of exception. In most cases it is NIL, but you can provide last ditch exception handlers even for tough exceptions such as index corruption.

Chapter

17 Operating Environment

The purpose of this chapter is to discuss various issues that will affect the delivery of applications developed using Visual Objects. The issues discussed are things that you might not consider during the development phase of your application but will want to think about before making the final delivery to your end users.

Shared Libraries and DLLs

In Visual Objects, there are two ways to share code among applications: shared libraries and dynamic link libraries (DLLs). Each one has its own advantages and circumstances under which it is ideally used, which is the subject of this section.

Shared Libraries

If you have developed DOS applications, you are probably already familiar with .LIB files, which are similar (although not identical) to shared libraries in Visual Objects. Shared libraries provide a convenient way to store code that is used by more than one application. Instead of putting the code directly in each application that uses it, you put it in its own application (identified with an application type of Library) that you include in the search path of every application that needs access to it.

You create a shared library in much the same way as an application, but shared libraries differ from applications in that they do not have a Start() entity and you cannot, therefore, run a shared library as a stand-alone application or generate an executable file from one.

When you create an executable file for an application that depends on one or more shared libraries, the compiled code from each library is statically linked into the resulting .EXE file, similar to the way in which a DOS application has .LIB file code linked into it.

Shared libraries reduce the amount of code you have to maintain, thereby making application development easier. But, they are not an ideal way to distribute library code to end users because they necessitate the distribution of your source code. They are primarily intended for generating stand-alone applications that you want to distribute as executable files.

At any point during the development cycle, you can change a shared library's Application Type to DLL if you find that a DLL better suits your needs; however, there are some differences that may affect your decision. Reading the following sections will help you decide if a DLL is more in keeping with your needs than a shared library.

Dynamic Link Libraries

You may not be familiar with DLLs if you have never developed a Windows application. These libraries reside as disk files that contain fully linked code. A DLL can be thought of as a special kind of binary executable that cannot be called directly but only by .EXE files or other DLLs. DLLs have several advantages over shared libraries.

Library Distribution

First, DLLs are maintained in files that are distinct and independent from any executable file. This makes them ideal for distribution to other developers as extension libraries. The only source code you need to distribute is an Application Export Format (.AEF) file with the _DLL statements that define the interface, or *public protocol*, for the DLL's *exported routines*.

More Efficient Use of Memory

DLLs also allow your applications to run using less memory in certain circumstances. For example, if several applications that share a DLL are running simultaneously, the DLL only needs to be loaded into memory once.

Visual Objects applications are a good example of this benefit. Every application that you create has, as one of its properties, the ability to include runtime support in the .EXE. If you choose this option, the Visual Objects runtime support will be statically linked into the application, and Windows will allocate memory for the runtime support when the application is running. You can see, then, that running several Visual Objects applications that were generated using this option would use more memory than necessary.

If, instead, you do not include runtime support in the .EXE, applications get their runtime support from a DLL (see How to Distribute Your Application later in this chapter for more information). Thus, memory for the runtime support will be allocated only once, no matter how many applications are running. And, since the DLL code does not have to be linked each time you build your application, using DLLs also speeds up the build process.

Version Independence

Applications using a DLL do not require a specific version of the DLL. As long as the interface of the DLL does not change, it may be replaced with a more recent version, giving you dramatically enhanced flexibility in maintaining your applications. Suppose you deliver several Visual Objects applications that use the runtime support DLL. When an updated version of this DLL is released, you do not have to rebuild all your applications. Simply replace the old DLL with the new one, and every application will benefit from the updated DLL.

Language Flexibility

Since each DLL has a clearly defined interface, an application may use DLLs written in different languages. Thus, you can use any DLL available in the marketplace, regardless of how it was created. The Visual Objects language provides you with the necessary data types, such as structures and pointers, so that you are not limited in any way. The Visual Objects system itself is a good example of combining DLLs written in different languages. It combines DLLs written in Visual Objects, C, C++, and Assembly language.

Distributed Development

The clearly defined interface also makes distributed development easier. After the interfaces for the different subsystems are well defined, every developer may work independently, publishing a DLL from time to time that is integrated into the system at large. In fact, the DLL architecture helped a lot to separate tasks in the development of Visual Objects, a very large-scale development project distributed across many different countries.

Using Visual Objects, you can design and build DLLs for use with other applications and use existing DLLs in your own applications.

Using DLLs

To access a routine in an existing DLL file, you simply declare the routine to the compiler (either in the application itself or in one of the shared libraries included in its search path). The basic information needed is the name of the routine and the name of its DLL. Besides that, the compiler also needs to know how to call the routine and how to handle its return value.

You declare DLL routines using the _DLL statement (see the online help system for details) illustrated in the example below. This example declares the Windows API function MessageBeep(), which allows you to play the message tones defined in the control panel. Using the Windows SDK documentation to determine the interface for this function (that is, the data types of its arguments and return value and its calling convention), its declaration can be written as follows:

```
_DLL FUNCTION MessageBeep(uType AS DWORD) AS LOGIC PASCAL:USER32.MessageBeep
```

Tip: All of the Windows API functions and constants are declared in either WIN32 API or the System Library and are available to your application automatically. Thus, this example declaration is for illustration purposes only. Viewing the source code in these libraries (or any other library defined by the system) will give you numerous examples of _DLL declaration statements.

Once declared, you call the DLL function as you would any other function, making sure to respect the data types of the declared parameters and return value:

```
FUNCTION ErrOut(cMsg)
    ? cMsg
    MessageBeep(0)
```

Note: DLL routines may also be declared and accessed from other DLLs, but there are some limitations based on the type of DLL you are working with. See Creating DLLs for more information.

At runtime, any DLL needed by your application is dynamically linked at application startup, with appropriate error messages for missing DLLs and references that cannot be resolved.

Creating DLLs

In addition to using existing DLLs, you can also create your own DLLs in Visual Objects in almost exactly the same way that you create applications. The main difference is that you specify DLL as the Application Type so that the system will know to create a .DLL instead of an .EXE file. Other than that, applications and DLLs are maintained in the same manner.

Guidelines

There are a few guidelines that govern DLLs created using Visual Objects:

- Do not include a Start() entity in a DLL.
- Although not required, it is a good idea to place all entities in one module to minimize the number of far data segments (one for each module containing local static data) that must be managed by the DLL.
- Global variables that must be available to an application should be declared either in the application itself or in a separate, shared library. Globals declared in a DLL are not available to the application.
- During the development and testing phases of a DLL, it is not necessary to generate an actual .DLL file. Instead, while it is under development, include the DLL in the search path of any application that uses it. This causes the applications to draw from the compiled code for the DLL as it is defined in the repository, without making any connection to a .DLL file.

- DLLs must not contain debug information when being linked.

  Then, when the DLL has been tested and is stable, generate the .DLL file by clicking the Generate Executable button when the DLL has focus. Doing so will generate two files: the .DLL file and an .AEF file defining the public protocol for the generated DLL.

  Note: Visual Objects has a DLL debugging feature. While the typical method for debugging DLL code in the past has been to treat the DLL code as a library and include it in the search path of a Visual Objects application, this cannot be done if the host application is a non-Visual Objects application. In such cases (for example, a Web browser running an ActiveX control), you have to use the new DLL debugging feature, which works with the actual binary .DLL file. For more detailed information about DLL debugging, see the online help.

  If you then import this .AEF file, it will be created as a shared library (rather than a DLL) in the repository. This library is your link to the actual .DLL file because it contains the _DLL declarations that point to the .DLL file by name. (It also contains other declarations found in the DLL, such as DEFINE, STRUCTURE, and CLASS declarations).

  Thus, to cause your applications to access the .DLL file (instead of the DLL in the repository), you must replace the DLL in all search paths with the newly created library associated with the .AEF file.

- When distributing an application that uses a DLL that you have created or when distributing a DLL as an extension library, be sure to include both the .DLL file and the definition of its public protocol (for Visual Objects applications, this is contained in the .AEF file mentioned above).

Tip: To keep track of different versions of the DLLs you create, you can use a VERSIONINFO resource.

If you create DLLs which are intended for use with non-Visual Objects applications, you cannot export any features that are specific to Visual Objects.

If you want to use a DLL with a non-Visual Objects generated .EXE, you must adhere to the following rules:

- A public protocol that can include functions, procedures, and methods (including access and assign methods)
- Exported routines (those not declared as STATIC) must be declared using the STRICT, PASCAL, or CALLBACK calling convention
- The parameters and return value of an exported routine must not be typed as ARRAY, OBJECT, STRING, CODEBLOCK, FLOAT, or USUAL

Static Data

Concerning the management of STATIC and GLOBAL data, Visual Objects DLLs follow the WIN32 behavior, where every process accessing a DLL gets its own set of data. Processes cannot overwrite each others data in a DLL and data cannot be shared between processes using DLLs.

Using a DLL

You can use a DLL generated by Visual Objects with any system that allows you to use a standard DLL. There is nothing unusual or special that will limit your usage.

Exactly how you will use the DLL, however, depends on the system itself. For example, to use a DLL in a C system, you need to create the C prototype declarations for its exported routines. Then you have to create an import library for the Visual Objects DLL and specify this import library when linking your C application. For details on creating import libraries for WIN32 DLLs, refer to the Microsoft WIN32 SDK.

Utilizing the Registry

Well-behaved Windows and Windows NT applications make use of the system registry to determine certain runtime settings. The initial settings which are usually stored in the registry when the application is installed are often dynamic.

Visual Objects is no exception. For example, the name of the standard .UDC file and path names for the locations of various file types and are defined. The default settings for the development environment are also stored in the registry under the key:

```
HKEY_CURRENT_USER¥¥Software¥¥ComputerAssociates¥¥
CA-Visual Objects 2.7
```

Visual Objects Registry Entries

The system registry is organized in a tree structure of ordered pairs of keys and their associated values. The Visual Objects runtime library contains functions (e.g., SetRTRegString) that allow you to add and query subkey and the values of that key. The runtime settings for Visual Objects are stored under:

```
HKEY_CURRENT_USER¥¥Software¥¥ComputerAssociates¥¥
CA-Visual Objects Applications
```

For information on specific Visual Objects registry keys, refer to the “Visual Objects Registry Entries” appendix of the *IDE User Guide*.

Accessing the Registry from an Application

For most of the development process, you will probably be using the dynamic execution feature of the IDE to run, test, and debug your application. However, once it is ready to use, you will want to build an .EXE file for distribution to your end users.

The Visual Objects Install Maker will create an install version including an install program to copy the necessary files to the end user's hard disk, managing user-configurable aspects of the application (see Utilizing the Registry section earlier in this chapter), and placing the program icon in the Start menu. For details about creating the install version, refer to the "Using the Install Maker" appendix of the *IDE User* Guide. This section provides specific information on how to create an executable version of your application and exactly what files you need to distribute so that the end user can run your application.

Managing Projects

In Visual Objects , the repository presented by the Repository Explorer consists of two separate parts, the system repository and the user repository, also called project. The system repository comes with Visual Objects and is setup as part of the install process. It contains the system provided standard libraries. The system repository is completely read-only and therefore it never changes. There is no need to backup the system repository when backing up your data, since it can always be reinstalled from the installation CD-ROM. The system repository resides in the SYSTEM directory in your installation root directory.

Default Project

During the installation, one project called Default Project is created in the DATA directory of your installation root directory. Since this project is empty (containing no applications, although there are a few system files in the project directory) you will see only the system libraries, when you start Visual Objects for the first time. As long as you work with only one project, the Repository Explorer gives the impression that you are working with one logical repository.

Multiple Projects

The idea behind using several projects, is to make it easier for you to organize your work and to share your work with others. You might also consider setting up different projects for different applications you are working on and maybe one project for experiments. Each project is supposed to reside in its own directory. To create a new project, you choose New Project from the File menu when the root item is selected in a Repository Explorer. You will be prompted for the project name and the project directory. The directory does not have to exist. Visual Objects will create it. After a successful project creation, the Repository Explorer will show two project branches in the left pane. All projects include the system libraries, but in actuality refer to the same files. The system libraries are shared between all projects.

Sharing Project Components

Although your active projects appear in the same explorer window, they are completely separate. It is not possible to have a connection from an application in one project to an application in another project. In particular, libraries in a project cannot be used by applications in another project. You also cannot have any editor open for entities in different projects. If you change your active project by clicking on an application in another project, all editors currently open will close down.

Drag and drop of application components is also only possible within one project. To transfer application components from one project to another, use Cut and Paste in the Repository Explorer.

Projects are completely self-contained. To back up a specific project, you simply have to back up the directory containing the project.

Important! *Projects cannot be exchanged between different versions of Visual Objects. When switching to another version of Visual Objects, you have to export the contents of your projects as AEF files and import them into the new version.*

The Project Catalog

All your active projects are managed through a project catalog. This catalog is maintained in the PROJECTS directory within your installation root directory. A project can only belong to one project catalog at a time. This means that if you have a project visible in a Repository Explorer, it is in your project catalog and nobody else can access it.

Add/Delete Project

To give others access to a specific project, you have to remove it from your project catalog. This is done by right-clicking on the desired project and choosing Delete from Catalog in the pop-up menu. Deleting a project from your catalog leaves all existing files intact.

Once you have deleted a project, somebody else can access the project and add it to their project catalog. Similarly, you can add projects created by others to your catalog, if the project is no longer in use by anyone else. To add a project to your catalog, choose Add Project from the file menu and specify the desired project name in the project location of the dialog box shown.

The capability of adding or removing projects from your catalog can also be used as an alternative to importing or exporting AEF files to exchange applications. Once a project has been removed from your catalog, you can put it on disk or send it somewhere else. The receiver simply has to add the project to his own catalog.

Important! *This method of exchanging applications only works with the same version of Visual Objects. To transfer data from across versions, AEF files have to be used.*

The following piece of information might prove useful for recovery operations. If you use your project catalog in the PROJECTS directory of the root installation directory, the projects in that catalog still appear in use by Visual Objects. To remove the lock placed on the project, delete the _PROJECT.VO file in the project directory. Do not delete the _PROJECT.VO file without serious consideration!

How to Distribute Your Application

For most of the development process, you will probably be using the dynamic execution feature of the IDE to run, test, and debug your application. However, once it is ready to use, you will want to build an .EXE file for distribution to your end users.

The Visual Objects Install Maker will create an install version including an install program to copy the necessary files to the end user's hard disk, managing user-configurable aspects of the application (see the Utilizing the Registry section earlier in this chapter), and placing the program icon in the Start menu. For details on creating the install version, refer to the "Using the Install Maker" appendix of the *IDE User Guide*. This section gives you specific information on how to create an executable version of your application and exactly what files you need to distribute so that the end user can run your application.

Generating the .DLL and .EXE Files

Prepare the DLLs

The first step is to prepare all user-defined DLLs associated with the application, which is actually a four-step process:

1. Highlight the DLL and verify that its Application Properties are correct. In particular, make sure the path and file name are as you intend and that the Enable Debug check box is not selected.
2. Generate the .DLL file by clicking on the MakeEXE toolbar button. This will generate a corresponding .AEF file defining the interface for the DLL.
3. Import the .AEF file generated in step 2 as a shared library, making sure to build it.

   Note: If the DLL is your end product, you can stop here. All you need to distribute are the .DLL and .AEF files. You should probably document for your end users which other files it depends on so that they will have this information when they distribute applications that use your DLL. See Other Files to Distribute in this chapter for details.
4. Replace the original DLL in the application's search path with the library imported in step 3.

Make the .EXE File

The next step is to highlight the application and verify that its path and file name are correct using the Application Properties dialog box. You may also want to uncheck the Enable Debug check box to minimize the size of the resulting .EXE file.

Then, you can generate the .EXE file by clicking on the MakeEXE toolbar button and then run the application from Windows to make sure all is well.

Note: The dependency management of the repository also includes the generated .EXE. When you touch the application from the repository, the .EXE becomes invalid and will be deleted. Make sure to create or a copy of the .EXE if you want to keep it and continue developing. If the .EXE already exists and you press the link button, the link will not take place.

Other Files to Distribute

System-Defined Files

Obviously, you must distribute to your end users the .EXE file and all .DLL files that you generate, but there are other files that you must also include, depending on the particulars of your application (for example, which system-defined libraries it uses). The following table summarizes the files needed under various circumstances:

Circumstance	Files to Distribute
All applications except those including runtime support in the .EXE file	CAVORT20.DLL CAVONT20.DLL
Applications using RDD Classes library	CAVO2RDD.DLL CAVODBF.RDD
Applications using NTX Driver	DBFNTX.RDD
Applications using MDX Driver	DBFMDX.RDD
Applications using CDX Driver	DBFCDX.RDD _DBFCDX.RDD
Applications using BLOB Files	DBFBLOB.RDD
Applications using Memo Files	DBFMEMO.RDD
Applications using delimeted Files	DELIM.RDD
Applications using SDF File	SDF.RDD
Applications using GUI Classes library	CAVO2GUI.DLL CATO3CNT.DLL CATO3DAT.DLL CATO3MSK.DLL CATO3NBR.DLL CATO3TIM.DLL CATO3SBR.DLL CATO3SPL.DLL MSVCRT2O.DLL
Applications using SQL Classes library (Any ODBC driver that you are using must also be installed.)	CAVO2SQL.DLL
Applications using System Classes library	CAVO2SYS.DLL
Applications using Terminal Lite	CAVO2TRM.DLL
Applications using OLE Classes	CAVO2OLE.DLL CAVOOLE.DLL
Applications using Q&E Files	IVC3.DLL

Circumstance	Files to Distribute
Applications using Q&E Utilities	C3UTL13.DLL C3TRN13.DLL
Applications using Report Classes library	CABL3.DLL CABL3DB1.DLL CABLE.HLP CAVO2REP.DLL CAIM3BMP.DLL CAIM3DBM.DLL CAIM3GIF.DLL CAIM3IO.DLL CAQR3CQM.DLL CAQR3DBA.DLL CAQR3DBC.DLL CAQR3EDT.DLL CAQR3MEM.DLL CAQR3QQ.DLL CAQR3RCC.DLL CAQR3RES.DLL CAQR3RET.DLL CAQR3SPL.DLL CAQR3WBM.DLL CAQR3WQM.DLL CAQR3WRM.DLL CAQR3WUT.DLL CAQRLABL.DAT
(Any ODBC driver that you are using must also be installed.)	CATO3TBR.DLL CAQRWRET.HLP
Install Maker Files	CAIN4SHL.DLL CAINDREG.DLL CALM_W32.DLL CATOCFGE.EXE INSTALL.ISC SETUP.EXE UIPRECAL.EXE

Distribution of the files listed in this table is unlimited and free of charge. You cannot, however, distribute any of the other files provided as part of the Visual Objects package.

Database and Ancillary Files

You must also distribute any database (.DBF), memo (.DBT or other, depending on the RDD), and index (.NTX or other, depending on the RDD) that are required by your application. Make sure that you remove any test data from the files before distribution (for example, using the ZAP command).

All Report Writer (.RET) files, Xbase report (.FRM) and label (.LBL) files, help (.HLP) files, and any other files (for example, .TXT) used by the application must also be distributed.

Location of Files

You can install your application's .EXE file anywhere you want, and in most cases you will want to add this path to the DOS PATH list as defined in the user's AUTOEXEC.BAT file.

Although Windows will search for DLLs in a number of locations (see the _DLL statement in the online help system for details), the easiest configuration is to install them in the same directory as the main .EXE file. This suggestion also applies to the .RDD files mentioned in the table above, as well as any .HLP files that are associated with the application. You may, however, choose to install certain DLLs (those that may be used by applications other than Visual Objects) in the main Windows directory or its SYSTEM subdirectory. You can use the locations chosen by the Visual Objects installation as a guideline.

The database and ancillary files can be installed wherever you like, but the application must be set up to know how to locate them. See Chapter 15, "File Handling," in this guide for more information on the rules for locating and creating files.

Chapter

18 Third-Party Components

This part of the *Programmer's Guide* has been about components. Every developer is faced from time to time with a burning need for a component they do not have. Then they have the difficult decision of whether to design, document, code, and test the component themselves or look elsewhere for it.

This chapter attempts to provide some assistance for people faced with a "develop or buy" decision. The first section looks at the detailed factors that influence the decision, and the following section looks at the long-term benefits of supporting a third-party components industry. Experience shows that although you obviously have total control over components you develop, maintaining them over the long term is a costly and specialized activity.

Selecting Components

Most of the factors that influence your decision to buy a third-party component library center around whether it really helps the task you wish to accomplish. This chapter, however, cannot tell you whether a particular component is suitable for your needs. Instead, it looks at architectural factors.

Such factors appear secondary in the short term; however, as you build or buy many components over time, it is their conformance to architectural standards that determines how well they work together. Your productivity in the long term depends critically on your choice of components.

Guidelines

The following guidelines will help you both when selecting a third-party library and when designing your own component libraries.

First, you should review the architecture set out in Chapter 4, "Standard Components-Classes, Objects, and Libraries," and see how well the components that you are considering follow those principles. As an example, if the dynamic relationships between objects become too complex, you will not be able to debug applications, and you will never be sure that an object is collectable by the garbage collector. One of the functions of the architecture is to keep the dynamic structure simple.

Second, you should review the benefits that the GUI classes deliver. These are set out in Chapter 11, "GUI Classes," in this guide. Try to figure out if the new component library does as well in its application domain as the GUI Classes library does in the GUI domain.

Components as Capsules

Components should be delivered as *capsules*. In the Windows environment, this means that they should be DLLs. It is all right to deliver components in source code form, so long as they can operate as DLLs. If a third party delivers components as source code, this strongly affects their ability to handle technical support. If you change the source code, this makes it harder to support. Many companies solve this problem by handing off technical support to the developers who use the component. Therefore, source code is a mixed blessing. It enables you to be self-supporting, but it also requires you to be self-supporting.

Hypertext

Perhaps the most important aspect of any component is the quality of its hypertext. Do these components support the automatic hypertext system provided by the GUI classes? If they do not, it will be much harder for you to provide help and support to your users.

Third-Party Market

Over time, the effectiveness of the Visual Objects technology will hinge on the speed and accuracy with which you can select and employ appropriate and labor-saving components. Computer Associates encourages the interchange of components.

Computer Associates would like developers to conform to the Visual Objects architecture, as described in the previous section. The reason for this is to help people to learn each new component quickly and to help the components to work together.

As time goes by, the component business is going to grow into a significant part of the software tools business. It is in everyone's interest to contribute to this effort. If you have developed components that other people could use—let them know. If you require components that no one has yet developed—make your requirement known. Everyone will benefit if we trade high-quality components.

Third parties will develop libraries that extend the standard components. The libraries consist of binary code in a shared DLL, plus the class and method prototypes (the signature lines of the class and method declarations). The library's prototype is delivered as an .AEF file.

Chapter 19, "How the Visual Objects Two-level Preprocessor Works" later in this guide, describes some guidelines for evaluating and selecting third-party components that best complement the Visual Objects standard components. The Visual Objects development environment is designed to help you manage and benefit from a wealth of components supplied by both Computer Associates and third parties.

It is also worth pointing out that if someone makes a business out of developing component libraries, they are likely to become very good at it over time. If a library sells well, it can sustain a certain level of training business and a level of awareness and public discussion of its benefits and direction. It makes very good sense to encourage this kind of specialization. The only economy of scale in the software industry occurs through reusing components—there is never any advantage to reinventing the wheel.

Chapter

19

How the Visual Objects Two-Level Preprocessor Works

Visual Objects has a two-level preprocessor which consists of a CA-Clipper compatible preprocessor and the Visual Objects preprocessor. Having both preprocessors offers added benefits to the compilation phase of building an application. You can determine whether both or just the Visual Objects preprocessor is invoked.

This chapter explains how the two level preprocessor works and provides the preprocessor directives necessary to define commands and pseudofunctions.

Compilation

Before the compilation phase takes place, the preprocessor scans CH files attached to the current application and then each individual entity from top to bottom for certain directives and translates them into regular Visual Objects source code that can be compiled.

You can include up to 16 CA-Clipper-compatible .CH files. If one or more .CH files are included on the Clipper Headers tab page of the Application Options dialog box, entities will be preprocessed first by the CA-Clipper preprocessor, which does pure textual replacement. The preprocessed output will then be processed by the Visual Objects preprocessor and then compiled. .

Similarly, an application library can use up to 16 .UDC files. You attach .UDC files to an application or library on the UDCs tab page of the Application Options dialog box.

Note that a .UDC file attached to a library is not automatically visible to an application that includes the library in its search path. You must explicitly attach the .UDC file to the application.

If an application uses .CH files, all the CA-Clipper directives such as #xtranslate and #include can be used inside an entity's code. If an application does not use .CH files, the CA-Clipper preprocessor will not run during compilation therefore having no effect on performance. If an application uses .CH files, all conditional compilation (#ifdef, #ifndef) will be done by the CA-Clipper preprocessor and not seen by the Visual Objects preprocessor. The CA-Clipper preprocessor does not know about the defines in the repository since it uses the #defines in the header files.

Header Files

Header files, also referred to as *include files*, contain preprocessor directives and command definitions. Header files have no default extension and are specified using the Clipper Headers tab in the Application Options dialog box or by using the #include preprocessor directive in another .CH file.

How the CA-Clipper Compatible Preprocessor Works

Before compilation takes place, the preprocessor scans the source file from top to bottom for certain directives and translates them into regular Visual Objects source code that can be compiled. The output of the preprocessor is then used as input to the compiler.

The following table summarizes the preprocessor directives:

Directive	Meaning
#command	Specify a user-defined command or translation directive.
#define	Define a manifest constant or pseudofunction.
#ifdef	Compile a section of code if an identifier is defined.
#ifndef	Compile a section of code if an identifier is undefined.
#include	Include a file into the current source file.
#undef	Remove a #define definition.
#xcommand	Specify a user-defined command or translation directive without abbreviations.

#command | #translate directive

Specify a user-defined command or translation directive.

Syntax

#*command* <matchPattern> => <resultPattern>
#*translate* <matchPattern> => <resultPattern>

Arguments

<*matchPattern*> is the pattern the input text should match.

<*resultPattern*> is the text produced if a portion of input text matches the <*matchPattern*>.

The => symbol between <*matchPattern*> and <*resultPattern*> is, along with #command or #translate, a literal part of the syntax that must be specified in a #command or #translate directive. The symbol consists of an equal sign followed by a greater than symbol with no intervening spaces. Do not confuse the symbol with the >= or the <= comparison operators in the Visual Objects language.

Description

#command and #translate are translation directives that define commands and pseudofunctions. Each directive specifies a translation rule. The rule consists of two portions: a match pattern and a result pattern. The match pattern matches a command specified in the code entity and saves portions of the command text (usually command arguments) for the result pattern to use. The result pattern then defines what will be written to the result text and how it will be written using the saved portions of the matching input text.

#command and #translate are similar, but differ in the circumstance under which their match patterns match input text. A #command directive matches only if the input text is a complete statement, while #translate matches input text that is not a complete statement. #command defines a complete command and #translate defines clauses and pseudofunctions that may not form a complete statement. In general, use #command for most definitions and #translate for special cases.

#command and #translate are similar to but more powerful than the #define directive. #define, generally, defines identifiers that control conditional compilation and manifest constants for commonly used constant values such as INKEY() codes.

#command and #translate directives have the same scope as the #define directive. The definition is valid only for the current code entity file unless defined in one of the header files included through the Clipper Headers tab in the Application Options dialog box. If defined elsewhere, the definition is valid from the line where it is specified to the end of the entity. Unlike #define, a #translate or #command definition cannot be explicitly undefined. The #undef directive has no effect on a #command or #translate definition.

As the preprocessor encounters each source line preprocessor, it scans for definitions in the following order of precedence: #define, #translate, and #command. When there is a match, the substitution is made to the result text and the entire line is reprocessed until there are no matches for any of the three types of definitions. #command and #translate rules are processed in stack-order (i.e., last in-first out, with the most recently specified rule processed first).

In general, a command definition provides a way to specify an English language statement that is, in fact, a complicated expression or function call, thereby improving the readability of source code. You can use a command in place of an expression or function call to impose order of keywords, required arguments, combinations of arguments that must be specified together, and mutually exclusive arguments at compile time rather than at runtime. This can be important since procedures and user-defined functions can now be called with any number of arguments, forcing any argument checking to occur at runtime. With command definitions, the preprocessor handles some of this.

When defining a command, there are several prerequisites to properly specifying the command definition. Many preprocessor commands require more than one #command directive because mutually exclusive clauses contain a keyword or argument.

This also occurs when a result pattern contains different expressions, functions, or parameter structures for different clauses specified for the same command. Each formulation of the command is translated into a different expression. Because directives are processed in stack order, when defining more than one rule for a command, place the most general case first, followed by the more specific ones. This ensures that the proper rule will match the command specified in the entity.

Match Pattern

The <*matchPattern*> portion of a translation directive is the pattern the input text must match. A match pattern is made from one or more of the following components, which the preprocessor tries to match against input text in a specific way:

Literal values are actual characters that appear in the match pattern. These characters must appear in the input text, exactly as specified to activate the translation directive.

Words are keywords and valid identifiers that are compared according to the dBASE convention (case-insensitive, first four letters mandatory, etc.). The match pattern must start with a Word.

#xcommand and #xtranslate can recognize keywords of more than four significant letters.

Match markers are label and optional symbols delimited by angle brackets (<>) that provide a substitute (idMarker) to be used in the <resultPattern> and identify the clause for which it is a substitute. Marker names are identifiers and must, therefore, follow the Visual Objects identifier naming conventions. In short, the name must start with an alphabetic or underscore character, which may be followed by alphanumeric or underscore characters.

This table describes all match marker forms:

Match Marker	Name
<idMarker>	Regular match marker
<idMarker,...>	List match marker
<idMarker:word list>	Restricted match marker
<*idMarker*>	Wild match marker
<(idMarker)>	Extended Expression match marker

Regular match marker: Matches the next legal expression in the input text. The regular match marker, a simple label, is the most general and, therefore, the most likely match marker to use for a command argument. Because of its generality, it is used with the regular result marker, all of the stringify result markers, and the blockify result marker.

List match marker: Matches a comma-separated list of legal expressions. If no input text matches the match marker, the specified marker name contains nothing. You must take care in making list specifications because extra commas will cause unpredictable and unexpected results.

The list match marker defines command clauses that have lists as arguments. Typically these are FIELDS clauses or expression lists used by database commands. When there is a match for a list match marker, the list is usually written to the result text using either the normal or smart stringify result marker. Often, lists are written as literal arrays by enclosing the result marker in curly ({ }) braces.

Restricted match marker: Matches input text to one of the words in a comma-separated list. If the input text does not match at least one of the words, the match fails and the marker name contains nothing.

A restricted match marker is generally used with the logify result marker to write a logical value into the result text. If there is a match for the restricted match marker, the corresponding logify result marker writes true (.T.) to the result text; otherwise, it writes false (.F.). This is particularly useful when defining optional clauses that consist of a command keyword with no accompanying argument.

Wild match marker: Matches any input text from the current position to the end of a statement. Wild match markers generally match input that may not be a legal expression, gather the input text to the end of the statement, and write it to the result text using one of the stringify result markers.

Extended expression match marker: Matches a regular or extended expression, including a file name or path specification. It is used with the smart stringify result marker to ensure that extended expressions will not get stringified, while normal, unquoted string file specifications will.

Optional match clauses are portions of the match pattern enclosed in square brackets ([ ]). They specify a portion of the match pattern that may be absent from the input text. An optional clause may contain any of the components allowed within a <*matchPattern*>, including other optional clauses.

Optional match clauses may appear anywhere and in any order in the match pattern and still match input text. Each match clause may appear only once in the input text. There are two types of optional match clauses: one is a keyword followed by match marker, and the other is a keyword by itself. These two types of optional match clauses can match all of the traditional command clauses typical of the Xbase command set.

Optional match clauses are defined with a regular or list match marker to match input text if the clause consists of an argument or a keyword followed by an argument. If the optional match clause consists of a keyword by itself, it is matched with a restricted match marker.

In any match pattern, you may not specify adjacent optional match clauses consisting solely of match markers, without generating a compiler error. You may repeat an optional clause any number of times in the input text, as long as it is not adjacent to any other optional clause. To write a repeated match clause to the result text, use repeating result clauses in the <*resultPattern*> definition.

Result Pattern

The <*resultPattern*> portion of a translation directive is the text the preprocessor will produce if a piece of input text matches the <*matchPattern*>. <*resultPattern*> is made from one or more of the following components:

Literal tokens are actual characters that are written directly to the result text.

Words are Visual Objects keywords and identifiers that are written directly to the result text.

Result markers: refer directly to a match marker name. Input text matched by the match marker is written to the result text via the result marker.

This table lists the Result marker forms:

Result Marker	Name
<idMarker>	Regular result marker
#<idMarker>	Dumb stringify result marker
<"idMarker">	Normal stringify result marker
<(idMarker)>	Smart stringify result marker
<{idMarker}>	Blockify result marker
<.idMarker.>	Logify result marker

Regular result marker: Writes the matched input text to the result text, or nothing if no input text is matched. Use this, the most general result marker, unless you have special requirements. You can use it with any of the match markers, but it almost always is used with the regular match marker.

Dumb stringify result marker: Stringifies the matched input text and writes it to the result text. If no input text is matched, it writes a null string (""). If the matched input text is a list matched by a list match marker, this result marker stringifies the entire list and writes it to the result text.

This result marker writes output to result text where a string is always required. This is generally the case for commands where a command or clause argument is specified as a literal value but the result text must always be written as a string even if the argument is not specified.

Normal stringify result marker: Stringifies the matched input text and writes it to the result text. If no input text is matched, it writes nothing to the result text. If the matched input text is a list matched by a list match marker, this result marker stringifies each element in the list and writes it to the result text.

The normal stringify result marker is most often used with the blockify result marker to compile an expression while saving a text image of the expression.

Smart stringify result marker: Stringifies matched input text only if source text is enclosed in parentheses. If no input text matched, it writes nothing to the result text. If the matched input text is a list matched by a list match marker, this result marker stringifies each element in the list (using the same stringify rule) and writes it to the result text.

The smart stringify result marker is designed specifically to support extended expressions for commands other than SETs with <*xlToggle*> arguments. Extended expressions are command syntax elements that can be specified as literal text or as an expression if enclosed in parentheses. The <*xcDatabase*> argument of the USE command is a typical example. For instance, if the matched input for the <*xcDatabase*> argument is the word Customer, it is written to the result text as the string "Customer," but the expression (cPath + cDatafile) would be written to the result text unchanged (i.e., without quotes).

Blockify result marker: Writes matched input text as a code block without any arguments to the result text. For example, the input text x + 3 would be written to the result text as {| | x + 3}. If no input text is matched, it writes nothing to the result text. If the matched input text is a list matched by a list match marker, this result marker blockifies each element in the list.

The blockify result marker used with the regular and list match markers matches various kinds of expressions and writes them as code blocks to the result text. Remember that a code block is a piece of compiled code to execute sometime later. This is important when defining commands that evaluate expressions more than once per invocation. When defining a command, you can use code blocks to pass an expression to a function and procedure as data rather than as the result of an evaluation. This allows the target routine to evaluate the expression whenever necessary.

Logify result marker: Writes true (.T.) to the result text if any input text is matched; otherwise, it writes false (.F.) to the result text. This result marker does not write the input text itself to the result text.

The logify result marker is generally used with the restricted match marker to write true (.T.) to the result text if an optional clause is specified with no argument; otherwise, it writes false (.F.).

Repeating result clauses are portions of the <*resultPattern*> enclosed by square brackets ([ ]). The text within a repeating clause is written to the result text as many times as it has input text for any or all result markers within the clause. If there is no matching input text, the repeating clause is not written to the result text. Repeating clauses, however, cannot be nested. If you need to nest repeating clauses, you probably need an additional #command rule for the current command.

Repeating clauses are the result pattern part of the #command facility that create optional clauses which have arguments. You can match input text with any match marker other than the restricted match marker and write to the result text with any of the corresponding result markers.

Notes

Less than operator: If you specify the less than operator (<) in the <*resultPattern*> expression, you must precede it with the escape character (\).

Multistatement lines: You can specify more than one statement as a part of the result pattern by separating each statement with a semicolon. If you specify adjacent statements on two separate lines, the first statement must be followed by two semicolons.

Examples

These examples encompass many of the basic techniques you can use when defining commands with the #command and #translate directives. Note that the functions mentioned below are fictitious functions specified for illustration only.

- This example defines the @...BOX command using regular match markers with regular result markers:

```
#command  @ <top>, <left>, <bottom>, <right> BOX <boxstring>;
=>;
CmdBox( <top>, <left>, <bottom>, <right>,<boxstring> )
```

- This example uses a list match marker with a regular result marker to define the ? command:

```
#command ? [<list,...>] => QOUT(<list>)
```

- This example uses a restricted match marker with a logify result marker to implement an optional clause for a command definition. In this example, if the ADDITIVE clause is specified, the logify result marker writes true (.T.) to the result text; otherwise, it writes false (.F.):

```
#command RESTORE FROM <file> [<add: ADDITIVE>];

=>;
     CmdRestore( <(file)>, <.add.> )
```

- This example uses a list match marker with a smart stringify result marker to write to the result text the list of fields specified as the argument of a FIELDS clause. In this example, the field list is written as an array with each field name as an element of the array:

```
#command COPY TO <file> [FIELDS <fields,...>];

=>;
     CmdCopyAll( <(file)>, { <(fields)> } )
```

- These examples use the wild match marker to define a command that writes nothing to the result text. Do this when attempting to compile unmodified code developed in another dialect:

```
#command SET ECHO <*text*>    =>

#command SET TALK <*text*>    =>
```

- These examples use wild match markers with dumb stringify result markers to match command arguments specified as literals, then write them to the result text as strings in all cases:

```
#command SET PATH TO <*path*>  =>  ;
     SET( _SET_PATH, #<path> )
#command SET COLOR TO <*spec*> =>  SETCOLOR( #<spec> )
```

- These examples use a normal result marker with the blockify result marker to both compile an expression and save the text version of it for later use:

```
#command SET FILTER TO <xpr>;

=>;
     CmdSetFilter( <{xpr}>, <"xpr"> )

#command INDEX ON <key> TO <file>;
=>;
     CmdCreateIndex( <(file)>, <"key">, <{key}> )
```

- This example demonstrates how the smart stringify result marker implements a portion of the USE command for those arguments that can be specified as extended expressions:

```
#command USE <db> [ALIAS <a>];

=>;
     CmdOpenDbf( <(db)>, <(a)> )
```

- This example illustrates the importance of the blockify result marker for defining a database command. Here, the FOR and WHILE conditions matched in the input text are written to the result text as code blocks:

```
#command COUNT [TO <var>];
     [FOR <for>] [WHILE <while>];
     [NEXT <next>] [RECORD <rec>] [<rest:REST>] [ALL];
     =>;
     <var> := 0,;
     DBEVAL( {|| <var>++}, <{for}>, <{while}>,;
     <next>, <rec>, <.rest.> )
```

- In this example the USE command again demonstrates the types of optional clauses with keywords in the match pattern. One clause is a keyword followed by a command argument, and the second is solely a keyword:

```
#command USE <db> [<new: NEW>] [ALIAS <a>] ;
     [INDEX <index,...>][<ex: EXCLUSIVE>] ;
     [<sh: SHARED>] [<ro: READONLY>];
     =>;
     CmdOpenDbf(<(db)>, <(a)>, <.new.>,;
               IF(<.sh.> .OR. <.ex.>, !<.ex.>, NIL),;
               <.ro.>, {<(index)>})
```

- This example uses the STORE command definition to illustrate the relationship between an optional match clause and a repeating result clause:

```
#command STORE <value> TO <var1> [, <varN> ];
=>;

    <var1> := [ <varN> := ] <value>
```

- This example uses #translate to define a pseudofunction:

```
#translate AllTrim(<cString>);

 => ;

LTRIM(RTRIM(<cString>))
```

#define directive

Define a manifest constant or pseudofunction.

Syntax

#*define* <idConstant> *[*<resultText>*]*

#*define* <idFunction>*([*<arg list>*]) [*<exp>*]*

Arguments

<idConstant> is the name of an identifier to define.

<resultText> is the optional replacement text to substitute whenever a valid <*idConstant*> is encountered.

<idFunction> is a pseudofunction definition with an optional argument list (<*arg list*>). If you include <*arg list*>, it is delimited by parentheses (()) immediately following <*idFunction*>.

<exp> is the replacement expression to substitute when the pseudofunction is encountered. Enclose this expression in parentheses to guarantee precedence of evaluation when the pseudofunction is expanded.

Note: #define identifiers are case-sensitive, where #command and #translate identifiers are not.

Description

The #define directive defines an identifier and, optionally, associates a text replacement string. If specified, replacement text operates much like the search and replace operation of a text editor. As each source line from a program file is processed by the preprocessor, the line is scanned for identifiers. If a currently defined identifier is encountered, the replacement text is substituted in its place.

Defined identifiers can contain any combination of alphabetic and numeric characters, including underscores. Defined identifiers, however, differ from other identifiers by being case-sensitive. As a convention, defined identifiers are specified in uppercase to distinguish them from other identifiers used within a program. Additionally, identifiers are specified with a one or two letter prefix to group similar identifiers together and guarantee uniqueness.

When specified, each definition must occur on a line by itself. Unlike statements, more than one directive cannot be specified on the same source line. You may continue a definition on a subsequent line by employing a semicolon (;). Each #define directive is specified followed by one or more white space characters (spaces or tabs), a unique identifier, and optional replacement text. Definitions can be nested, allowing one identifier to define another.

A defined identifier has lexical scope like a static local variable. The definition is valid only for the current code entity file unless defined in one of the header files included through the Clipper Headers tab in the Application Options dialog box. If defined elsewhere, the definition is valid from the line where it is specified to the end of the entity. A #define can be explicitly undefined.

You can redefine or undefine existing identifiers. To redefine an identifier, specify a new #define directive with the identifier and the new replacement text as its arguments. The current definition is then overwritten with the new definition, and a compiler warning is issued in case the redefinition is inadvertent. To undefine an identifier, specify an #undef directive with the identifier as its argument. #define directives have three basic purposes:

- To define a control identifier for #ifdef and #ifndef
- To define a manifest constant—an identifier defined to represent a constant value
- To define a compiler pseudofunction

The following discussion expands these three purposes of the #define directive in your program.

Preprocessor Identifiers

The most basic #define directive defines an identifier with no replacement text. You can use this type of identifier when you need to test for the existence of an identifier with either the #ifdef or #ifndef directives. This is useful to either exclude or include code for conditional compilation.

Manifest Constants

The second form of the #define directive assigns a name to a constant value. This form of identifier is referred to as a manifest constant. For example, you can define a manifest constant for the INKEY() code associated with a key press:

```
#define K_ESC  27
IF LASTKEY() = K_ESC
     .
     . <statements>
     .
ENDIF
```

Whenever the preprocessor encounters a manifest constant while scanning a source line, it replaces it with the specified replacement text.

Although you can accomplish this by defining a variable, there are several advantages to using a manifest constant: the compiler generates faster and more compact code for constants than for variables; and variables have memory overhead where manifest constants have no runtime overhead, thus saving memory and increasing execution speed. Furthermore, using a variable to represent a constant value is conceptually inconsistent. A variable by nature changes and a constant does not.

Use a manifest constant instead of a constant for several reasons. First, it increases readability. In the example above, the manifest constant indicates more clearly the key being represented than does the INKEY() code itself. Second, manifest constants localize the definition of constant values, thereby making changes easier to make, and increasing reliability. Third, and a side effect of the second reason, is that manifest constants isolate implementation or environment specifics when they are represented by constant values.

To further isolate the effects of change, manifest constants and other identifiers can be grouped together into header files allowing you to share identifiers between entities, applications, and groups of programmers. Using this methodology, definitions can be standardized for use throughout a development organization. Merge header files into the current application by specifying them in the Clipper Headers tab in the Application Options dialog box.

Compiler Pseudo-functions

In addition to defining constants as values, the #define directive can also define pseudofunctions that are resolved at compile time. A pseudofunction definition is an identifier immediately followed by an argument list, delimited by parentheses, and the replacement expression. For example:

```
#define AREA(nLength, nWidth)  (nLength * nWidth)
#define SETVAR(x, y)           (x := y)
#define MAX(x, y)              (IF(x > y, x, y))
```

Pseudofunctions differ from manifest constants by supporting arguments. Whenever the preprocessor scans a source line and encounters a function call that matches the pseudofunction definition, it substitutes the function call with the replacement expression. The arguments of the function call are transported into the replacement expression by the names specified in the argument list of the identifier definition. When the replacement expression is substituted for the pseudofunction, names in the replacement expression are replaced with argument text. For example, the following invocations,

```
? AREA(10, 12)
SETVAR(nValue, 10)
? MAX(10, 9)
```

are replaced by :

```
? (10 * 12)
nValue := 10
? (IF(10 > 9, 10, 9)
```

It is important when defining pseudofunctions, that you enclose the result expression in parentheses to enforce the proper order of evaluation. This is particularly important for numeric expressions. In pseudofunctions, you must specify all arguments. If the arguments are not specified, the function call is not expanded as a pseudofunction and exits the preprocessor to the compiler as encountered.

Pseudofunctions do not entail the overhead of a function call and are, therefore, slightly faster. Pseudofunctions, however, are more difficult to debug within the debugger, have a scope different from declared functions and procedures, do not allow skipped arguments, and are case-sensitive.

You can avoid some of these deficiencies by defining a pseudofunction using the #translate directive. #translate pseudofunctions are not case-sensitive, allow optional arguments, and obey the dBASE four-letter rule. See the #translate directive reference in this chapter for more information.

Examples

- In this example a manifest constant conditionally controls the compilation of debugging code:

```
#define DEBUG
.
. <statements>
.
#ifdef DEBUG
     Assert(FILE("System.dbf"))
#endif
```

- This example defines a manifest constant and substitutes it for an INKEY() value:

```
#define K_ESC            27
.
. <statements>
.
IF INKEY() != K_ESC
     DoIt()
ELSE
     StopIt()
ENDIF
```

- This example defines pseudofunctions for the MAX() and ALLTRIM() functions:

```
#define MAX(arg1, arg2)              (IF(arg1 > arg2, ;
     arg1, arg2))
#define ALLTRIM(cString)  (RTRIM(LTRIM(cString)))
.
. <statements>
.
? MAX(1, 2)
? ALLTRIM("  Hello  ")
```

#ifdef directive

Compile a section of code if an identifier is defined.

Syntax

#*ifdef* <identifier>
        <statements>...
[#else]
        <statements>...
#*endif*

Arguments

<*identifier*> is the name of a definition whose existence is being verified.

Description

#ifdef...#endif lets you perform a conditional compilation. It does this by identifying a section of source code to be compiled if the specified <*identifier*> is defined. The <*identifier*> can be defined using the #define directive.

The #else directive specifies the code to compile if <*identifier*> is undefined. The #endif terminates the conditional compilation block.

Conditional compilation is particularly useful when maintaining many different versions of the same program. For example, the demo code and full system code could be included in the same program file and controlled by a single #define statement.

Examples

- This code fragment is a general skeleton for conditional compilation with #ifdef:

```
#define DEMO
.
. <statements>
.
#ifdef DEMO
     <demo specific statements>
#endif
```

- This example defines a manifest constant to one value if it does not exist and redefines it to another if it exists:

```
#ifdef M_MARGIN
     #undef  M_MARGIN
     #define M_MARGIN    15
#else
     #define M_MARGIN    10
#endif
```

#ifndef directive

Compile a section of code if an identifier is undefined.

Syntax

#*ifndef* <identifier>
        <statements>...
[#else]
        <statements>...
#*endif*

Arguments

<*identifier*> is the name of a definition whose absence is being verified.

Description

#ifndef...#endif lets you perform conditional compilation by identifying a section of source code to compile if the specified <*identifier*> is undefined.

The #else directive specifies the code to compile if <*identifier*> is defined. The #endif terminates the conditional compilation block.

Examples

- This code fragment is a general skeleton for conditional compilation with #ifndef:

```
#define DEBUG
.
. <statements>
.
#ifndef DEBUG
     <optimized version of code>
#else
     <debugging version of code>
#endif
```

#include directive

Include a file into the current header file or entity.

Syntax

#*include* "<headerFileSpec>"

Arguments

<headerFileSpec> specifies the name of another source file to include in the current source file. As indicated in the syntax, the name must be enclosed in double quotation marks.

<*headerFileSpec*> may contain an explicit path and file name as well as a file extension. If, however, no path is specified, the preprocessor searches the following places:

- Directories specified in the INCLUDE environment variable

#include directives may be nested up to 15 levels deep – that is, a file that has been included may contain #include directives, up to 15 levels.

Description

#include inserts the contents of the specified file in place of the #include directive in the header file or entity. By convention, the file inserted is referred to as a header file. Header files should contain only preprocessor directives and external declarations. By convention CA-Clipper header files have a .CH extension.

Header files overcome the one major drawback of defining constants or inline functions—the #define directive only affects the file in which it is contained. To have the directives globally available to an application, they should be written to a header file and that header file should either be included on the Clipper Headers tab page of the Application Options dialog box, or #included in another header file that is included in the application's properties.

Another advantage of using the #include directive is that all the #define statements are contained in one file. If any modifications to these statements are necessary, only the #include file need be altered; the program itself remains untouched.

#undef directive

Remove a #define macro definition.

Syntax

#*undef* <*identifier*>

Arguments

<*identifier*> is the name of the manifest constant or pseudofunction to remove.

Description

#undef removes an identifier defined with the #define directive. After an #undef, the specified identifier becomes undefined. Use #undef to remove an identifier before you redefine it with #define, preventing the compiler warning that occurs when an existing identifier is redefined. Also, use #undef to make conditional compilation specific to certain sections of a program.

Examples

- To define and then undefine a manifest constant and a pseudofunction:

```
#define K_ESC  27
#define MAX(x, y)         IF(x > y, x, y)
.
. <statements>
.
#undef K_ESC
#undef MAX
```

- To use #undef to undefine an identifier before redefining it:

```
#define DEBUG
.
. <statements>
.#undef DEBUG
#define DEBUG  .T.
```

- To undefine an identifier if it exists, and otherwise define it for later portions of the program file:

```
#ifdef TEST

     #undef TEST
#else
     #define TEST
#endif
```

#xcommand | #xtranslate directive

Specify a user-defined command or translation directive.

Syntax

#*xcommand* <matchPattern> => <resultPattern>
#*xtranslate* <matchPattern> => <resultPattern>

Arguments

<matchPattern> is the pattern to match in the input text.

<resultPattern> is the text produced if a piece of input text matches the <*matchPattern*>.

Description

The #xcommand and #xtranslate directives work like #command and #translate except that they overcome the dBASE keyword length limitation. They are significant beyond the first four letters, limited only by available memory. All other rules apply.

How the Visual Objects Preprocessor Works

In Visual Objects, all commands are defined in a text file that is linked into your application or library and processed via the User Defined Command (UDC) system during the compilation process.

The standard commands are defined in a file called STD.UDC which is used by default, but you can create additional .UDC files and attach them to your application or library as needed.

Having command syntax and semantics defined in this manner provides a flexible system that can be extended in many ways, including the ability to define your own commands. This chapter tells you how to define commands and use the UDC system to make them available to your application.

Why Commands?

Most of the commands in the standard Visual Objects command set can be reduced to a single expression (usually a function call). Some take more than that, but commands can always be reduced to equivalent expressions, so why have them? There are a couple of reasons:

1. Visual Objects is a new product with lots of history, and a big part of that history is based in the use of commands for database processing. Commands have been part of the Xbase language since its inception.
2. Command syntax is often easier to read and remember than a functional equivalent, especially when there are many parameters involved.

The second reason may also compel you to employ the UDC system to provide a command-like interface for some of your more complicated functions. Functions that require many arguments are often difficult to use and understand since it is only the position of the argument in the argument list (and perhaps its name) that gives it meaning. By replacing such a function with a UDC, you can attach keywords to the parameters and list them in any order you want, making your code not only easier to write but also easier to read.

Creating a .UDC File

Unlike other parts of your application which are maintained in the repository without the need for external files, UDCs must be maintained in external text files. To create a .UDC file (using your favorite editor or the source code editor in the IDE), simply type in the translation rules (see the Translation Rules section below) for the commands you want to define and save the text file with a .UDC extension.

To help maintain .UDC files, the IDE has a Tools UDC Tester command that lets you test a translation rule before writing it to a .UDC file. For more information on using these IDE features, see the *IDE User Guide*.

Order Significance

UDC translation rules are processed in sequential order, which may affect the order in which you place them in the .UDC file.

For example, in STD.UDC, the DELETE command is broken into two translations rules: one for deleting the current record and the other for deleting multiple records. (The reason for having separate rules is that the two cases translate into different function calls.)

In this case, the version to delete a single record is the simpler of the two and appears before the more complicated case. Otherwise, the UDC system would translate a single record DELETE as a multiple record DELETE without options.

Attaching a .UDC File

An application library can use up to 16 .UDC files. You attach .UDC files to an application or library in the Application Options dialog box, similar to the way you add libraries to the search path. (See the *IDE User Guide* for more information.)

Note that a .UDC file attached to a library is not automatically visible to an application that includes the library in its search path. You must explicitly attach the UDC file to the application.

Compilation

You define a UDC using one or more *translation rules* that you specify in the form of a *match pattern* and a *result pattern*. As stated earlier, you store these UDC translation rules in a text file that you attach to your application.

When you compile an application that contains commands, the compiler searches all attached .UDC files in sequential order until it finds a translation rule for which the command *input text* matches the match pattern. When the compiler finds a matching translation rule, it makes the appropriate substitution according to the result pattern and produces *result text* which is compiled as part of the application.

If you specify a command for which the compiler can find no matching translation rule, the result is a compiler error.

Translation Rules

As with other components of the Visual Objects language, UDC translation rules have their own syntax and semantics that you must adhere to and understand. This section serves as the reference for defining UDC translation rules and is structured similarly to the alphabetical command and statement entries in this guide. It describes how to structure match patterns and result patterns to achieve the results you want from the compiler.

Syntax

<*MatchPattern*> => <*ResultPattern*>

Arguments

<*MatchPattern*>

The pattern the input text must match to trigger the translation rule. The match pattern specifies the syntax of the command.

<*ResultPattern*>

The text produced when the translation rule is triggered. The result pattern specifies the semantics of the command.

Note: The => symbol between <*MatchPattern*> and <*ResultPattern*> is a literal part of the syntax. The symbol consists of an equal sign followed by a greater than symbol with no intervening spaces. Do not confuse this symbol with the >= or the <= comparison operators in the Visual Objects language.

Match Pattern

The <*MatchPattern*> portion of a translation rule defines the pattern that a command must match in order to trigger the translation rule. A match pattern consists of tokens and match markers that the compiler tries to match against input text in a specific way.

Tokens

Tokens are keywords and other characters, such as commas and parentheses, that appear in the match pattern. Keywords are compared according to Xbase conventions (case-insensitive and first four letters mandatory), and other characters must appear in the input text exactly as specified. All match patterns must start with a token.

Important! *When matching keywords that have more than four letters, the minimum that must be specified is four letters. If there are conflicting rules defined for the application (for example, SET PROCLINE and SET PROCNAME in STD.UDC), you should specify enough letters to uniquely identify each keyword; otherwise, the first rule that matches the abbreviation will be triggered. The possibility of more than one .UDC file per application increases the likelihood of a conflict; therefore, it is better to spell out each command completely rather than rely on abbreviations.*

Match Markers

Match markers are delimited by angle brackets (<>). They define the circumstances under which text is considered to match and create an identifier to hold the input text for use within the result pattern.

Match Marker	Name
<*idMarker*>	Regular match marker
<*idMarker*,...>	List match marker
<*idMarker:WordList*>	Restricted match marker
<**idMarker***>	Wild match marker
</*idMarker*/>	File match marker

Regular match marker, <idMarker>: Matches the next legal expression or literal identifier in the input text. The regular match marker is the most general and, therefore, the most likely match marker to use for a command argument. Because of its generality, you can use it with the regular result marker, all of the stringify result markers, and the blockify result marker.

List match marker, <idMarker,...>: Matches a comma-separated list of legal expressions. If no input text matches the match marker, the specified marker name contains nothing; otherwise, it contains the list.

Use the list match marker to define command clauses that have argument lists (for example, FIELDS clauses). When there is a match for a list match marker, you can write the list to the result text using the normal or smart stringify result marker. You can turn the list into an array by enclosing the result marker in curly braces ({ }).

Restricted match marker, <idMarker:WordList>: Matches input text to one of the words in a comma-separated list. If the input text does not match at least one of the words, the marker name contains nothing; otherwise, it contains the matched word.

You can use the restricted match marker with the logify result marker to write a logical value into the result text. If there is a match for the restricted match marker, the corresponding logify result marker writes TRUE to the result text; otherwise it writes FALSE. This is particularly useful when defining optional clauses that consist of a command keyword with no accompanying argument. STD.UDC implements the REST clause of database commands using this form.

Wild match marker, <*idMarker*>: Matches any input text from the current position to the end of a statement. Wild match markers generally match input that may not be a legal expression, such as NOTE <*x*> in STD.UDC.

Choices are either to ignore the text (by not writing it to the result text) or to write it using one of the stringify result markers.

File match marker, </idMarker/>: Matches a file name expressed either as a literal identifier or a string expression. If the input text is not a file name, the marker name contains nothing; otherwise, it contains the file name. You will almost always use the smart stringify result marker to write the file name to the result text.

Optional Clauses

Optional clauses are portions of the match pattern enclosed in square brackets ([ ]) that can be absent from the input text. An optional clause can contain any of the components allowed within a match pattern, including other optional clauses.

Optional clauses can appear anywhere and in any order in the match pattern. When the input text is matched, the order is not significant, but each optional clause can appear only once.

There are two types of optional clauses in the Visual Objects command set: one is a keyword by itself, and the other is a keyword followed by one or more arguments. Lone keywords are usually represented using a restricted match marker and a corresponding logify result marker:

```
... [<ro:READONLY>] ... => ... <.ro.> ...
```

A keyword followed by arguments can be represented using any match marker/result marker combination:

```
... [FIELDS <list,...>] ... => ... { <(list)> } ...
```

Repeating Clauses

Repeating clauses are portions of the match pattern that are optional, but can be repeated. Sometimes, a command or clause can have a list of arguments, all of which are optional except the first (for example, USE...INDEX and STORE). These clauses are usually in the form of a keyword followed by a list in which the items can be repeated any number of times. You specify these as a single optional clause and use a repeating result clause in the result pattern:

```
STORE <value> TO <var1> [, <varN> ] =>              ;
    <var1> := [ <varN> := ] <value>
```

When you specify an optional clause like this, adjacent to something that is not optional, the clause can be repeated any number of times (or not at all) in the input text. The repeating result clause (indicated by square brackets in the result pattern) repeats the result text as many times as needed to match the input text.

The USE command in STD.UDC provides several examples of optional and repeating clauses.

Result Pattern

The <*ResultPattern*> portion of a translation rule defines what the compiler will produce when the translation rule is triggered. Similar to match patterns, result patterns consist of tokens and result markers.

Tokens

Tokens are characters that are written to the result text, as is. Anything that appears outside the angle brackets of a result marker, excluding the square bracket symbols ([ ]), is considered to be a literal token. Examples of literal tokens are function names, commas, parentheses, operators, and literal values.

Important! *Command invocations are not allowed as literal tokens in a result pattern. In other words, you cannot define one UDC in terms of another UDC.*

Result Markers

Result markers refer directly to a match marker identifier. Input text matched by the match marker is written to the result text via the result marker.

Result Marker	Name
<*idMarker*>	Regular result marker
<%*idMarker*%>	Function result marker
<#*idMarker*#>	Dumb stringify result marker
<$*idMarker*$>	Normal stringify result marker
<(*idMarker*)>	Smart stringify result marker
<{*idMarker*}>	Blockify result marker
<.*idMarker*.>	Logify result marker

Regular result marker, <idMarker>: Writes the matched input text (or nothing if no input text matches) to the result text, as is. Use this, the most general result marker, unless you have special requirements. You can use it with any of the match markers, but its most common use is with the regular match marker.

Function result marker, <%idMarker%>: Writes the matched input text (or nothing if no input text matches) to the result text as a function call. Use this result marker whenever you want to translate input text into a function call—using a regular result marker will not work because the UDC system generates different token types for regular symbols and symbols used as function names. The function result marker is most often used with the regular match marker.

Dumb stringify result marker, <#idMarker#>: *Stringifies* the matched input text (encloses it in string delimiters) and writes it to the result text. The manner in which lists are stringified depends on the match marker. When used with a list match marker (<*idMarker*,...>), lists are stringified one element at a time (for example, a, b, c becomes "a", "b", "c"). If no input text matches, the result is a NULL_STRING.

Use this result marker when a string is required. This is generally the case for commands where a clause or argument is specified as a literal value and the result text must be written as a string, even if the argument is not specified (like SET PATH in STD.UDC).

Normal stringify result marker, <$idMarker$>: Stringifies the matched input text and writes it to the result text. When used with a list match marker (<*idMarker*,...>), lists are stringified one element at a time (for example, a, b, c becomes "a", "b", "c"). If no input text matches, no result text is written.

You will often use the normal stringify and blockify result markers together to compile an expression while saving a text image of it. (See SET FILTER in STD.UDC.)

Smart stringify result marker, <(idMarker)>: If the matched input text is *not* enclosed in parentheses, this result marker stringifies it before writing it to the result text; otherwise, matched input text is written to the result text as is. When used with a list match marker (<*idMarker*,...>), lists are stringified one element at a time (for example, a, b, c becomes "a", "b", "c"). If no input text matches, nothing is written to the result text.

The smart stringify result marker is designed specifically to support extended expressions. USE is a typical example:

```
USE Customer
```

becomes:

```
DBUseArea(.F.,,"Customer",...)
```

while:

```
USE (cPath + cDatafile)
```

becomes:

```
DBUseArea(.F.,, (cPath + cDatafile),...)
```

Blockify result marker, <{idMarker}>: *Blockifies* matched input text (encloses it in code block delimiters) and writes it to the result text. For example, the input text x + 3 would be written to the result text as {| | x + 3}. When used with a list match marker (<*idMarker*,...>), lists are blockified one element at a time (for example, a, b, c becomes {| | a}, {| | b}, {| | c}). If no input text matches, nothing is written to the result text.

Use the blockify result marker with regular and list match markers to turn expressions into code blocks. In STD.UDC, the blockify result marker defines database commands where an expression is evaluated for each record (such as FOR and WHILE conditions).

Logify result marker, <.idMarker.>: This result marker writes TRUE to the result text if the match marker contains a value and FALSE otherwise—it does not write the actual input text to the result.

The logify result marker is generally used with the restricted match marker to specify the presence or absence of a keyword as a logical value. This formulation is used in STD.UDC to define the READONLY and NEW clauses of the USE command, while slightly more complicated logic is used to define the SHARED and EXCLUSIVE clauses.

Repeating Clauses

Repeating clauses are portions of the result pattern enclosed in square brackets ([ ]). The text within a repeating clause is written to the result text as many times as it has input text for any or all result markers within the clause. If there is no matching input text, the repeating clause is not written to the result text.

Repeating clauses cannot be nested. If you find the need to nest repeating clauses, try breaking the command into two or more translation rules.

You will most often use repeating result clauses with optional match clauses that have arguments. You can match input text with any match marker other than the restricted match marker and write to the result text with any of the corresponding result markers. Typical examples of this facility are the definitions for STORE and REPLACE in STD.UDC.

Notes

Less than operator: If you specify the less than operator (<) in the result pattern, you must precede it with the escape character (\).

Comments: With the exception of the NOTE command (which is itself defined in STD.UDC), all source code comment indicators are allowed in UDC files. (See Chapter 20, "Overview of Language Elements," in this guide for a table.)

Line continuation: Use a single semicolon to continue a translation rule onto a new line.

Multistatement lines: If the result pattern contains more than one statement, each statement must be separated from the next by two consecutive semicolons, regardless of whether the next statement is on a new line. In other words, a double semicolon generates a semicolon in the result text:

```
SET MESSAGE TO <n> [<cent:CENTER, CENTRE>] =>       ;
     SetMessage(<n>)                                          ;;          ;
     SetMCenter(<.cent.>)

CLOSE ALTERNATE =>                                    ;
     SetAltFile("");; SetAlternate(.F.)
```

In cases where the statement is part of a repeating result clause, the double semicolon should be inside the square brackets:

```
REPLACE <f1> WITH <v1> [, <fN> WITH <vN> ] =>       ;
          _FIELD->&lt;f1&gt; := <v1>                    ;
     [;;  _FIELD->&lt;fN&gt; := <vN> ]
```

Examples

These examples, taken from STD.UDC, illustrate many of the basic techniques you can use when defining commands.

This example defines the @...BOX command using regular match markers with regular result markers:

Translation Rule

```
@ <t>, <l>, <b>, <r> BOX <str> [COLOR <color>] => ;
     DispBox(<t>, <l>, <b>, <r>, <str> [, <color>])
```

Input Text

```
@ 1, 1, 15,75 BOX B_SINGLE_DOUBLE
```

Result Text

```
DispBox(1, 1, 15, 75, B_SINGLE_DOUBLE)
```

The following example uses a restricted match marker with a logify result marker to implement an optional clause for a command definition. In this example, if the ADDITIVE clause is specified, the logify result marker writes .T. (TRUE) to the result text; otherwise, it writes .F. (FALSE):

Translation Rule

```
RESTORE [FROM </file/>] [<add:ADDITIVE>] =>  ;
     _MRestore(<(file)>, <.add.>)
```

Input Text

```
RESTORE FROM myfile ADDITIVE
RESTORE FROM (cPath + cFile)
```

Result Text

```
_MRestore("myfile", .T.)
_MRestore((cPath + cFile), .F.)
```

This next example (abbreviated from the STD.UDC version) uses a list match marker with a smart stringify result marker to implement a standard FIELDS clause. In this example, the field list is written as an array of strings:

Translation Rule
```
COPY [TO </dest/>] [FIELDS <list,...>] =>    ;
    DBCopy(<(dest)>, { <(list)> })
```

Input Text
```
COPY TO destfile FIELDS Name, Address
```

Result Text
```
DBCopy("destfile", {"Name", "Address"})
```

The following example uses a wild match marker with a dumb stringify result marker to match a command argument specified as a literal, then write it to the result text as a string:

Translation Rule
```
SET PATH TO <*path*> => SetPath(<#path#>)
```

Input Text
```
SET PATH TO C:¥, C:¥DATA, D:¥MOREDATA
```

Result Text
```
SetPath("C:¥, C:¥DATA, D:¥MOREDATA")
```

This next example uses a regular match marker with blockify and normal stringify result markers to compile an expression and save the text version of it for later use with VODBFilter():

Translation Rule
```
SET FILTER TO <xpr> =>                    ;
    DBSetFilter(<{xpr}>, <$xpr$>)
```

Input Text
```
SET FILTER TO Name = 'Smith'
```

Result Text
```
DBSetFilter({|| Name = 'Smith'}, "Name = 'Smith'")
```

The following example demonstrates how the smart stringify result marker implements the RENAME command for file names specified as extended expressions:

Translation Rule
```
RENAME </oldFile/> TO </newFile/> =>      ;
    FRename(<(oldFile)>, <(newFile)>)
```

Input Text
```
RENAME cust.dbf TO customer.dbf
RENAME (cFileOld) TO (cFileNew)
```

Result Text
```
FRename("cust.dbf", "customer.dbf")
FRename((cFileOld), (cFileNew))
```

This next example illustrates the importance of the blockify result marker for defining database commands. Here, the FOR and WHILE conditions matched in the input text are written to the result text as code blocks:

Translation Rule

```
COUNT [TO <var>] [FOR <fo>] [WHILE <wh>]     ;
     [NEXT <nx>] [RECORD <rec>]                          ;
     [<rs:REST>] [ALL] =>                                ;
     <var> := 0                                                  ;;
     ;
     DBEval({|| <var> := <var> + 1}, <{fo}>,       ;
          <{wh}>, <nx>, <rec>, <.rs.>)
```

Input Text

```
COUNT TO nCount FOR Name = "Smith"
```

Result Text

```
nCount := 0
DBEval({|| nCount := nCount + 1},                ;
     {|| Name = "Smith"},,,,.F.)
```

The following example uses the STORE command definition to illustrate the relationship between an optional match clause and a repeating result clause:

Translation Rule

```
STORE <value> TO <var1> [, <varN> ] =>               ;
     <var1> := [ <varN> := ] <value>
```

Input Text

```
STORE 10 TO nOne, nTwo, nThree
```

Result Text

```
nOne := nTwo := nThree := 10
```

Chapter 20

Overview of Language Elements

This chapter introduces you to the basic language elements that go together to make up a program. It provides enough detail to help you understand sample program code that you read and introduces you to some terminology that you will be seeing throughout this and other guides in the documentation set.

Detailed information on many of the topics introduced here can be found in other chapters of this guide. For example, there is a chapter that discusses variable declaration in much greater detail ("Variables, Constants, and Declarations") and one that elaborates on using operators, identifiers, and literal values to build expressions ("Operators and Expressions"). Refer to the index or the table of contents to find the information you want.

The Parts of a Program

All Visual Objects applications consist of a series of statements that are defined in one or more modules. The types of statements that you will find are:

- Entity declarations
- Variable declarations
- Instance variable declarations
- Control structures
- Method invocations and instance variable access
- Function invocations
- Command invocations
- Object instantiation statements
- Assignment statements
- Comment lines

Language Elements

Within these components, you will find:

- Keywords, such as command and argument names
- Identifiers, such as entity, variable, and field names

- Operators, such as +, :=, {}, and .NOT.
- Literal values, such as "Frank Louis", 123.456, and 1993.12.30
- Expressions (combinations of identifiers, operators, and literal values), such as x + 1000, Substr(Name, 1, 10)

An Example Program

The following example program is used to illustrate the various types of statements and point out the language elements within the statements:

```
GLOBAL cDoneMessage := "Finished!"

FUNCTION Start()
     // This is the startup routine for the app
     NoMen()
     CLEAR SCREEN
     ? "This application is " + cDoneMessage

FUNCTION NoMen()
     LOCAL oDBEmp AS DBServer
     oDBEmp := DBServer{"employee"}
     oDBEmp:SetIndex("empno")
     DO WHILE .NOT. oDBEmp:EOF
          IF oDBEmp:Sex == "M"
               oDBEmp:Delete()
          ELSE
               oDBEmp:Salary += 500
          ENDIF
          oDBEmp:Skip()
     ENDDO
```

Entity Declarations

An *entity* is the smallest named unit of the *repository*, a special purpose database used to manage all of your Visual Objects applications and libraries, as well as the system-defined libraries. There are three entity declaration statements in the sample program:

```
GLOBAL cDoneMessage := "finished!"

FUNCTION Start()

FUNCTION NoMen()
```

GLOBAL and FUNCTION are predefined *keywords*, reserved words that have special meaning to the compiler.

cDoneMessage, *Start*, and *NoMen* are *identifiers* for the entities being declared.

Note: Identifiers must begin with a letter of the alphabet (A-Z or a-z—identifier names are not case-sensitive) or an underscore and can contain only letters, numbers, and the underscore character. Names are fully significant and must not conflict with reserved words. (The reserved words are listed in Appendix B of this guide).

Besides GLOBAL and FUNCTION, the other entity declaration statement keywords in the language are:

ACCESS	PROCEDURE
ASSIGN	RESOURCE
CLASS	STRUCTURE
DEFINE	TEXTBLOCK
METHOD	

All entity definitions start with one of these declaration statements and end with the next such statement. Some entity declaration statements (such as DEFINE and GLOBAL) completely define the named entity, while others (such as CLASS, METHOD, FUNCTION, and PROCEDURE) serve as an indicator that further definition of the entity follows.

Important! *Every application must have one entity name Start() that is a function, procedure, or method of the App class. The Start() routine is the first one executed when the application is executed. Start() cannot declare any parameters and, under normal circumstances, should not return a value. If you want to use strong typing in the declaration statement, you must specify AS USUAL PASCAL.*

Variable Declarations

Variable declarations are statements that inform the compiler of the identifiers used by an entity for variables and fields. In the example:

```
LOCAL oDBEmp AS DBServer
```

is a variable declaration statement that declares the lifetime and visibility of the identifier *oDBEmp* and indicates its data type with the keyword AS and the class name identifier *DBServer*.

Other variable declaration keywords are:

FIELD
LOCAL
MEMVAR
STATIC [LOCAL]

Variable declaration statements must fall within an ACCESS, ASSIGN, FUNCTION, METHOD, or PROCEDURE entity definition, preceding all other statements in the definition.

Instance Variable Declarations

Instance variable declarations are statements that inform the compiler of the names and visibility of the instance variables associated with a particular class. They optionally inform the compiler of data types and initial values. Although not used in the example, possible instance variable declaration keywords are:

EXPORT [INSTANCE]
HIDDEN [INSTANCE]
INSTANCE
PROTECT [INSTANCE]

These statements are always part of a CLASS entity definition.

Control Structures

Control structures, or *constructs*, are used to alter the flow of control in a program. The following IF...ENDIF construct is illustrated in the example:

```
IF oDBEmp:Sex = "M"
    oDBEmp:Delete()
ELSE
    oDBEmp:Salary += 500
ENDIF
```

IF is the keyword that starts the construct, and ENDIF is the keyword that ends it. ELSE is a special keyword that is valid only within the context of an IF...ENDIF construct. All control structures follow this same basic pattern.

Control structures fall into four categories:

- Conditional compilation (#ifdef...#endif and #ifndef...#endif)
- Conditional (IF...ENDIF and DO CASE...ENDCASE)
- Looping (DO WHILE...ENDDO and FOR...NEXT)
- Transfer (BEGIN SEQUENCE...END SEQUENCE).

With one exception, the entire construct must fall completely within an entity definition. The exception is the BREAK keyword belonging to the BEGIN SEQUENCE...END SEQUENCE construct: although the BEGIN SEQUENCE...END SEQUENCE construct must occur within one entity, BREAK may be placed within any entity in the application.

Nesting

The example shows an IF structure nested within a DO WHILE structure:

```
DO WHILE .NOT. oDBEmp:EOF
     IF oDBEmp:Sex = "M"
          oDBEmp:Delete()
     ELSE
          oDBEmp:Salary += 500
     ENDIF
     oDBEmp:Skip()
ENDDO
```

To properly nest them, one construct must fall completely within the other. There is no limit on the level to which you can nest control structures.

Method Invocations and Instance Variable Access

Method invocations execute the code associated with a particular METHOD declaration. (This can be a class library method or one that you define as part of your application.) In the example, there are two method invocations:

```
oDBEmp:Delete()
```

```
oDBEmp:Skip()
```

oDBEmp is an identifier representing a DBServer object.

Delete and *Skip* are identifiers associated with methods of the DBServer class. The parentheses are required operators in method invocations, even if you are not passing any arguments.

The send operator (:) connects to an object. It is also used to access instance variables (and ACCESS/ASSIGN methods) either for the purpose of access, as in:

```
IF oDBEmp:Sex = "M"
```

or for the purpose of assignment, as in:

```
oDBEmp:Salary += 500
```

See the Assignment Statements section later in this chapter for more information.

Function Invocations

Function invocations execute the code associated with a particular FUNCTION entity declaration. (This can be a library function or one that you define as part of your application.) In the example:

```
NoMen()
```

is a function invocation.

NoMen is the identifier associated with the function. The parentheses are required operators in function invocations, even if you are not passing any arguments.

Command Invocations

Command invocations are used to perform the action associated with a command name. (This can be a command defined in STD.UDC or another UDC file that you create). In the example:

```
CLEAR SCREEN
? "This application is " + cDoneMessage
```

are examples of command invocation statements.

"This application is " is a string literal.

The plus sign (+) is an operator used to concatenate two strings.

cDoneMessage is an identifier for a global variable entity.

Object Instantiation Statements

In many cases, you will instantiate an object like this, as part of an assignment statement:

```
oDBEmp := DBServer{"employee"}
```

But, you can also use object instantiations as stand alone program statements. (There are no cases of this in the example.) At first, it might seem like a strange thing to do, but keep in mind that the Init() method of the object, which is automatically called upon instantiation, can do all the processing necessary for some objects, making the assignment unnecessary. In these cases, instantiation is a valid program statement in and of itself.

Assignment Statements

Assignment statements are used to store values in variables and database fields. In the example:

```
oDBEmp := DBServer{"employee"}
```

is an assignment statement.

oDBEmp is an identifier representing a local variable.

:= is the assignment operator.

DBServer{"employee"} is an *expression* used to instantiate an object of the DBServer class. (*DBServer* is the class identifier, {} are the instantiation operators, and "employee" is a literal string that serves as an argument.)

This statement assigns the value on the right to the identifier on the left. After it is executed, *oDBEmp* contains an object of the DBServer class.

Other operators used to form assignment statements are:

++
--
+= (illustrated in the example with oDBEmp:Salary += 500)
-=
*=
/=
%=
^=

Note: The = operator can also be used for assignments provided that the Old Style Assignments option is checked either on the Compiler tab page in the Application Options dialog box or on the Compiler Defaults tab page in the System Settings dialog box.

Assignments that stand alone as program statements, like the one in the example, are allowed in ACCESS, ASSIGN, FUNCTION, PROCEDURE, and METHOD entity definitions only. You can also make assignments in GLOBAL, DEFINE, LOCAL, and all instance variable declarations (such as INSTANCE and PROTECT INSTANCE).

Predefined Identifiers

The following identifiers are reserved by the compiler and are expanded during compile time:

__AEFDIR__	Visual Objects Application Export directory
__APPLICATION__	Application name
__APPWIZDIR__	Visual Objects Application Wizard/Gallery directory
__CAVOBINDIR__	Visual Objects Binaries directory
__CAVODIR__	Visual Objects directory
__CAVODRIVE__	Visual Objects drive
__CAVOSAMPLESROOTDIR__	Visual Objects Samples directory
__DATE__	ANSI date string ("YYYYMMDD")
__ENTITY__	Entity name
__ENTITYSYM__	Symbolic entity name
__EXECUTABLEDIR__	Visual Objects Executable directory
__LINE__	Current line number
__MDFFILENAME__	Master document file name
__MEFDIR__	Visual Objects Module Export directory
__MODULE__	Module name
__OS__	Windows NT, Windows 95, or Windows 98
__PRGDIR__	Visual Objects Program File directory
__PROJECT__	Project name
__SYSDIR__	Windows system directory
__TIME__	Time string (HH:MM:SS)
__VERSION__	Visual Objects build number as a string (e.g., "603")
__WINDIR__	Windows directory
__WINDRIVE__	Windows drive

Additionally, the identifier __DEBUG__ is automatically defined when an entity is compiled for debugging. __DEBUG__ can only be used for conditional compilation (e.g., #ifdef debug).

Comments

Comments make the code more readable. In the example:

```
// This is the startup routine for the app
```

is a comment. The comment indicators available in Visual Objects are:

Indicator	Usage
//	Single line comments and inline comments at the end of a program statement
/* ... */	Multiple line comments and true, inline comments
*	Single line comments
NOTE	Single line comments
&&	Inline comments at the end of a program statement
TEXTBLOCK	Entity declaration statement in which all lines are ignored by the compiler

With all of the comment indicators except /*...*/, the compiler ignores all text following the indicator until the next carriage return/linefeed pair is encountered. With /*...*/ all text between the /* and the */, including carriage return/linefeed pairs, is ignored. You can, therefore, use /*...*/ to embed a comment within a line of code to create a true, inline comment.

Comment lines are allowed anywhere within an entity definition. To maintain comments for the entire module, create a TEXTBLOCK entity.

Line Continuation

Unless you explicitly specify that a statement is to be continued onto a new line, the compiler expects all statements to be completed before it encounters a carriage return/linefeed pair. To continue a statement onto a new line, use the semicolon (;) as a line continuation character:

```
oFSCustomer:FullPath := ;
      "¥¥ServerName¥SharedName¥PathName"
```

When the compiler encounters a semicolon followed by a carriage return/linefeed pair, it looks to the next line for the remainder of the statement. Using this technique, you can break a statement up onto several lines.

You cannot use the semicolon within a literal string to continue the remainder of the string on the next line. If you need to continue a long string, close the string delimiters and use the + operator before continuing with a new string on the next line:

```
? "This line is going to be very long, so I'll" + ;
  " continue it onto a new line."
```

Note: The line continuation character (;) is ignored when used as part of a comment.

Multistatement Lines

You can also use the semicolon to form a multistatement line. To do this, simply place a semicolon, without a carriage return/linefeed, between the statements. When the compiler encounters such a semicolon, it knows to read the next statement from the same line. For example, the following IF construct in the sample program uses the conventional one statement per line coding technique:

```
IF oDBEmp:Sex = "M"
    oDBEmp:Delete()
ELSE
    oDBEmp:Salary += 500
ENDIF
```

Here is the same IF construct, using multiple statements coded on the same line:

```
IF oDBEmp:Sex = "M" ; oDBEmp:Delete()
ELSE ; oDBEmp:Salary += 500
ENDIF
```

Note: The Debugger does not allow you to treat the individual statements in a multistatement line independently. For example, you cannot step through the individual statements. To debug the code, break the statements up into separate lines.

Chapter

21 Data Types

In Visual Objects programs, you will often find yourself manipulating data of one kind or another. Whether it is a database field that you want to display on a report, or a program variable that you want to use as a loop counter, each data item has a specific *data type* that you determine. Data types are used by the compiler and at runtime to enforce certain rules and to trap potential errors in your programs.

These data types are supported by the Visual Objects language:

- String
- Symbol
- Numeric types
- Date
- Logic
- NIL
- Void
- Array
- Code block
- Object

Each data type has specific operations associated with it. For example, expressing a value as a string lets you perform operations on it, such as combining it with another string or obtaining a substring; however, you cannot multiply two strings to get a product. Furthermore, except in rare cases, these operations are strictly limited to the data type for which they are intended. This means, for example, that you cannot compare a string to a number or concatenate a date with a string.

Choosing a data type, however, does not necessarily prevent you from using operations meant for another data type. For example, representing numbers as strings does not preclude mathematical manipulation, but you must first convert the strings to numbers (more on the subjects of operators, data type conversion, and mixing data types in the "Operators and Expressions" chapter).

To help you analyze your data and choose appropriate data types, this chapter discusses most of the data types listed previously. It will help you understand the concept of data typing and give you a high-level overview of the various data types available in the language.

The intention of this chapter is to lay the groundwork for the next two chapters, "Variables, Constants, and Declarations" and "Operators and Expressions." These chapters will give you more specific information on declaring data types to the compiler and using them to build expressions.

The array, code block, and object data types are more complex than the others and are presented as separate chapters in this guide. Because of their complexity, the discussion of these data types involves concepts that will not be presented until the next two chapters. For this reason, detailed discussion of these data types is delayed.

Finally, there are several system level data types that are not discussed in this chapter. These data types include several numeric data types (mentioned briefly in the Numeric section of this chapter) that are platform-independent, data structures that you can use to define your own data types, and pointers. You will find some information on these data types in the "Variables, Constants, and Declarations" chapter.

The "Operators and Expressions" chapter in this guide defines operators by data type.

String

The string—or character—data type identifies data items you want to manipulate as character strings. Examples of string data are a person's name, address, and state.

Often, you will find it more advantageous to represent numbers as strings because of the types of operations you intend to use. For example, you will seldom use a person's social security number in mathematical calculations, but often use it as a reporting field along with other strings. Therefore, social security numbers are almost always represented as strings rather than numbers. Other examples of numbers that are typically represented as strings are zip codes and phone numbers.

Note: If you are already familiar with the concept of a database field, you may also be aware of the existence of the memo data type for representing variable-length strings. Memo is a special data type that applies only to database fields and does not extend to the realm of program variables; however, the discussion of the string data type applies equally to the memo data type. See Chapter 8, "Using DBF Files," in this guide for more information on the subject of memo fields.

Character Set and Literals

The string data type uses the ANSI character set.
To form a string literal, or constant, enclose zero or more valid characters within one of the designated *delimiter* pairs:

- two single quotes (for example, 'one to open and one to close')
- two double quotes (for example, "one to open and one to close")
- left and right square brackets (for example, [left to open and right to close])

Since all of the designated delimiter characters are part of the valid character set, the delimiter characters themselves can be part of a string. To include a delimiter character in a string literal, use an alternate character for the delimiter. For example, if a string contains a single quote, enclose it in double quotes:

"I don't want to go."

Similarly, if a string contains double quotes, enclose it in single quotes:

'She said, "I have no idea."'

Note: To express a *null* – or empty – string, use a delimiter pair with no intervening characters – including spaces. For example, " and [] both represent a null string. You can also use the system-defined constant NULL_STRING for this purpose.

Limitations

The maximum string size is limited by the amount of available memory. Note that this value may differ depending on your platform and is defined by the constant MAX_ALLOC in the System Library.

Symbol

The symbol data type gives you a more efficient way to handle string data. Specifically, if you use symbols instead of strings when the string values are known at compile time, your application will execute faster, especially if you do a lot of comparisons.

The drawback of using symbols is that the available operators are limited to assignment and simple comparisons; however, you can convert symbols to strings using the Symbol2String() function, thereby obtaining access to all of the string operations.

Character Set and Literals

Because the function SysAddAtom() converts strings to symbols (atoms), symbols can contain all characters defined in the string character set. However, the literal representation for symbols limits the range of literal symbols that you can define.

To form a symbol literal, precede one or more characters with the hash mark (#). The first character must be an alphabetic character (A-Z, a-z) or an underscore, and the remaining characters can be letters, numbers, and underscores only. In literal symbol representations, lowercase letters are automatically converted to uppercase. These are valid symbol literals:

```
#DOG
#CAT
#CAT_4
```

These symbol literals are not valid:

```
#
#DOG BARKS
#CAT#45
#CAT+DOG
```

Note: To express a null symbol, use the system-defined constant NULL_SYMBOL.

Limitations

The maximum symbol size is 65,535. The maximum number of symbols is limited by the amount of available memory.

Note: Libraries included in the application may also use symbols, which will further reduce the number of symbols directly available to an application.

Numeric

The numeric data type identifies data items that you want to manipulate mathematically. Examples of numeric data items are a person's age, the balance of an account, and a loop counter.

Many data items that are represented as numbers, however, do not lend themselves to the numeric data type. For example, social security numbers, phone numbers, and zip codes are usually considered strings since you seldom, if ever, use them in mathematical calculations.

The Visual Objects language offers support for several numeric data types. This section generalizes all numeric data types without going into the specifics of any one type unless absolutely necessary.

Numeric Type	Description
INT	Signed integer: under Windows, 32 bits
FLOAT	Floating point number: under Windows, 64 bits
SHORTINT	Signed 16-bit integer; identical to ANSI C short
LONGINT	Signed 32-bit integer; identical to ANSI C long
BYTE	Unsigned 8-bit integer (a byte); identical to ANSI C unsigned character
WORD	Unsigned 16-bit integer (a word); identical to ANSI C unsigned short
DWORD	Unsigned 32-bit integer (a double word); identical to ANSI C unsigned long
REAL4	32-bit floating point number; identical to ANSI C float
REAL8	64-bit floating point number; identical to ANSI C double

Note: The data types listed in this table are ones that you can explicitly declare (see the "Variables, Constants, and Declarations" chapter), but you can also have undeclared numeric variables (sometimes called usual numerics) and field variables that are defined in a database file structure as numeric. The representations for these values are somewhat unique, but for the purpose of this discussion, an undeclared numeric value is like an INT or a FLOAT, depending on its value, and a numeric field is like a FLOAT.

Character Set and Literals

The character set for the numeric data type is defined as the digits from zero to nine, the period to represent a decimal point, the plus and minus symbols to represent the sign of a number, and the letters A-F, L, and X, uppercase or lowercase.

This character set is designed to cover all supported numeric types; however, not all characters make sense for all numeric data types. For example, integer values do not allow a decimal point, and decimal numbers do not allow the A-F characters as digits.

There are several ways to form numeric literals using various subsets of this character set, each of which is discussed below.

Important! *In no case are literal numeric values delimited. If you enclose a number – or any other string of characters – in string delimiters, it becomes a string.*

Decimal Notation

The most familiar method for creating numeric literals is the standard base 10, or *decimal*, notation, formed by stringing together one or more of the following:

- one or more digits (zero through nine) to represent the whole portion of the number
- a single, optional decimal point
- one or more digits to represent the fractional part of the number

For example:

- 1234
- 1234.
- 1234.5678
- .5678
- 0.5678

Hexadecimal Notation

To represent numeric literals in base 16, or *hexadecimal*, use this notation:

0x<*HexNumber*>

In this representation:

- *0* and *x* (uppercase or lowercase) are required to indicate this as a literal hexadecimal representation.
- <*HexNumber*> is an unsigned hexadecimal number (consisting of digits zero through nine and letters A-F).
- You cannot use intervening spaces anywhere within the representation.

These are examples of valid hexadecimal numbers:

- 0x5EA
- 0xFFFF
- 0X5E7AFFFF

Binary Notation

To represent numeric literals in base 2, or *binary*, use this notation:

0b<*BinaryNumber*>

In this representation:

- *0* and *b* (uppercase or lowercase) are required to indicate this as a literal binary representation.
- <*BinaryNumber*> is an unsigned binary number (consisting of digits zero and one).
- You cannot use intervening spaces anywhere within the representation.

These are examples of valid binary numbers:

- 0b10100001
- 0B111100001011
- 0b1010000111110011

Scientific Notation

You can also represent numeric literals using scientific notation:

<*DecimalNumber*>e<*Exponent*>

which you interpret as:

<*DecimalNumber*> * 10^<*Exponent*>

In this representation:

- <*DecimalNumber*> is any valid decimal number as described in the Decimal Notation section earlier. The decimal point in this number, however, is required.
- The letter *e* is required and can be either uppercase or lowercase.
- The <*Exponent*> is an integer value (it has no decimal point) with an optional unary sign (plus is assumed as the default).
- You cannot use intervening spaces anywhere within the representation.

These are examples of numeric literals represented using scientific notation:

- 1.5e7
- 1.5E7
- 911.123e-10
- 2.7E-32
- 1.0e+15

Long Integer Notation

You can specify a literal integer as a 32-bit value (LONGINT) using this notation:

<*Number*>l

In this representation:

- <*Number*> is a numeric literal as described above in the Decimal Notation, Binary Notation, or Hexadecimal Notation sections. If you use the decimal notation, the number should be an integer (it should not include a decimal point).
- The letter *l* is required and can be either uppercase or lowercase.
- You cannot use intervening spaces anywhere within the representation.

These are examples of numeric literals represented using the long integer notation:

- 0l
- 4568930L
- 0x5EAl
- 0xFFFFl
- 0b10100001L
- 0B111100001011L

Negative Numbers

You can precede any of the numeric representations mentioned above with a unary plus or minus sign but, strictly speaking, the sign is a unary operator—not part of the literal representation:

- -0b10100001
- -0x6FF
- -1.2e235
- -1.2e-235
- -1234
- -1234.5678
- -.5678

If you do not use a unary sign, a positive value is assumed. These two numbers are equivalent:

- 1234.567
- +1234.567

Note: Since the unary plus and minus signs are operators, spaces are allowed between the operator and the number. For example, these are valid negative numbers:

- - 0b10100001
- - 0x6FF
- - 1.2e235
- - 1.2e-235
- - 1234
- - 1234.5678

- -.5678

Limitations

The numeric data types are divided into two groups: platform-specific and platform-independent. The platform-specific types are the ordinary mathematical numerics that you will use most often in your programs, and the platform-independent types are designed for low-level interfacing with the underlying operating system.

The definitions of the platform-specific types change depending on the operating system in use so that their size is not the same from one system to another. For the Windows platform, the size limitations are:

Numeric Type	Range of Values	Significant Digits
INT	-2,147,483,648 to +2,147,483,647	NA
FLOAT	1.7E-308 to 1.7E+308	15

The platform-independent numeric types are specified with a certain size that is guaranteed to be kept on all platforms:

Numeric Type	Range of Values	Significant Digits
SHORTINT	-32,768 to +32,767	NA
LONGINT	-2,147,483,648 to +2,147,483,647	NA
BYTE	0 to 255	NA
WORD	0 to 65,535	NA
DWORD	0 to 4,294,967,295	NA
REAL4	3.4E-38 to 3.4E+38	7
REAL8	1.7E-308 to 1.7E+308	15

For information regarding declaring a particular numeric data type, see Chapter 22, "Variables, Constants, and Declarations," in this guide.

Date

The date data type identifies data items that represent calendar dates. Examples of date data items are a birthday, the date on which an account comes due, and today's date.

You can manipulate dates in several ways, such as finding the number of days between two dates and determining what the date will be ten days from now.

Note: DATE is a reserved word and cannot, therefore, be used as a field or variable name.

Character Set and Literals

The date character set is defined as the digits from zero to nine and the decimal point to separate the digits within the date.

To form a date literal, you string characters together using the ANSI date format:

```
[cc]yy.mm.dd
```

In this representation:

- *cc*, if specified, represents the century (the default is 19)
- *yy* represents the year
- *mm* represents the month
- *dd* represents the day
- the decimal point is the separator

These are examples of valid date literals:

- 89.01.02 is January 2, 1989
- 92.07.22 is July 22, 1992
- 1693.09.28 is September 28, 1693
- 100.01.01 is January 1, 100
- 0.0.0 (or any other invalid date) is a null date

Note: You can also express a null date using the system-defined constant NULL_DATE.

Limitations

These limitations apply to the individual components that go together to make up a date:

- *cc*, if specified, must be between 1 and 79
- *yy* must be between 0 and 99
- *mm* must be between 1 and 12
- *dd* must be between 1 and 31

In addition to obeying these rules, all dates must have internal integrity. For example, 89.11.31 and CToD("02/29/90") meet these criterion but are not valid dates because November has only 30 days and 1990 was not a leap year.

All valid dates in the range 01/01/0100 to 12/31/7900 as well as a null date are supported.

Logic

The logic data type identifies data items that are binary in nature. Typical logic data items are those with values of true or false, yes or no, or on or off.

For example, you could represent a switch setting that is either on or off using a logic value. A person's marital status and the answer to a true/false question on a quiz are other examples of logic data items.

Character Set and Literals

The logic character set consists of the letters y, Y, t, T, n, N, f, and F.

Though only two values are possible for this data type, there are several ways to represent them. To form a literal, enclose one of the characters in the defined character set between two periods. The periods are delimiters for logic values just as quote marks are for strings.

The reserved words TRUE and FALSE are used to represent the literal logic values.

NIL

NIL is a unique data type having only one value which is represented using the reserved word NIL. It is categorized as a data type because it has certain operations associated with it. When you assign NIL to an existing polymorphic variable, the variable does not retain its previous data type; however, NIL is in a category of its own and does not have the flavor of a real data type. There are several circumstances under which NIL values arise:

- A variable that is created with PRIVATE, declared but not typed (for example, LOCAL x), or declared AS USUAL will be initialized to NIL if you do not specify an initial value.
- A dynamic array will have its elements initialized to NIL when you first specify its dimensions or create it with ArrayCreate().
- A parameter defined as part of an untyped function, method, or procedure is set to NIL if you skip it when you call the routine.
- You can assign NIL to any polymorphic variable and use it in a comparison statement to determine if such a variable contains the NIL value.

Note: You cannot assign NIL to a strongly typed variable other than a USUAL, an OBJECT (or <*idClass*>), or an ARRAY. Except in these cases, the NIL value is limited to undeclared and untyped variables. See "Variables, Constants, and Declarations" for more information on the relationship between NIL and strongly typed arrays and objects.

Character Set and Literals

Using the reserved word NIL, you can create a literal NIL value:

```
x := NIL
```

By definition, any polymorphic variable containing the NIL value is of the NIL data type.

When you display a NIL value, the value is displayed, literally, as NIL:

```
?? x          // Result: NIL
```

Tip: The purpose of NIL is to let you manipulate a variable that is not initialized without generating a runtime error, but there are other uses that you may find convenient. For instance, with undeclared logical variables, you can use NIL to achieve three-state logic: the variable can contain three values instead of just TRUE or FALSE. You would interpret a NIL value as an unknown. You cannot do this with variables declared AS LOGIC.

See Chapter 22, "Variables, Constants, and Declarations" and Chapter 27, "Functions and Procedures," for more specific information on the NIL data type.

VOID

VOID, like NIL, is a unique data type that has a very limited usage. It is used internally, and you can use it to define functions that do not return any value as you would in a C function definition. However, no other operations are defined for the VOID data type—it cannot be assigned as a value and does not even have a literal representation.

NIL, NULL, and VOID—What Is the Difference?

Think of NIL as a value without a data type and VOID as a data type without a value. Since it is a legitimate value, you can actually assign NIL to a variable and test for this value using a comparison operator; the system uses NIL as a special indicator for polymorphic variables that are not initialized and skipped arguments. {xe "Data types:NIL"}{xe "Data types:VOID"}

VOID, on the other hand, is not a value. When a routine returns VOID, you cannot check the return value using a comparison, and you can never assign VOID to a variable.

NULL values are another case altogether. There are several predefined NULL constants that you can use as assignment values and to test for uninitialized typed variables. They are like typed NILs. Some of these constants (such as NULL_DATE and NULL_STRING) were mentioned earlier in this chapter, and others are listed in Chapter 22, "Variables, Constants, and Declarations."

See Chapter 27, "Functions and Procedures," for more specific information on the VOID data type used as a function return value.

Pointers

At runtime a program is made up of various components, such as functions, methods, and variables. All these components are loaded into memory while the program is running. To ensure that they interact properly, they have to reside in memory in an ordered way. This ordering is achieved by assigning memory addresses to each individual component. Normally, the developer does not have to worry about these low level details, since address assignment and resolution is handled by the operating system. However, for low level interfacing with the operation system, these issues become important.

The address of an application component is also referred to as a pointer. A pointer is an address value, pointing to some location in memory. In 32-bit Windows this value may be within the range of 0 to 4GB. What makes working with pointers difficult and a bit dangerous is the fact, that not all possible address values are valid. Only addresses that contain actual application components are valid values for pointers. A valid address depends on the actual application. Also, some addresses may be read-only, in which case, accessing the value will work fine but trying to change it will cause a processor exception.

Obviously there are different kinds of pointers. Pointers may point to the contents of a variable (data pointers) or to the beginning of a function (function pointers). Taking a look at data pointers, there are once again different types of pointers. Data pointers might point to a different data type such as a LONG value, a WORD value, a LOGIC value or even the location of another pointer. To properly handle function pointers, you not only need the address of the function, but also all the details (calling convention, number of parameters, parameter types, return type) about the function being pointed to.

The operation of retrieving the value that a pointer is pointing to or calling the function that a function pointer is pointing to is called dereferencing the pointer.

Visual Objects supports both untyped (sometimes also called anonymous) and typed pointers. Untyped pointers are simply an address without any semantic information attached to them. Looking at an anonymous pointer, it is impossible to tell if the pointer is pointing to data or to a function. Therefore, the use of anonymous pointers is limited since the compiler does not have the necessary information to perform certain operations (e.g. pointer arithmetic). Also, the compiler cannot prevent the programmer from dereferencing a pointer to the wrong data type.

Untyped Pointers

Variables containing pointers are declared using the PTR type specifier:

```
FUNCTION Start()
    LOCAL p AS PTR
```

Pointers are automatically initialized to contain a NULL_PTR. This is an invalid address, and therefore a NULL_PTR must not be dereferenced. This will immediately cause a processor exception as is the case when using any other invalid address value.

The address of the following items can be obtained by using the address operator (@) with them:

- Variables (Global, Local)
- Functions and procedures
- Elements of DIM Arrays
- Structure members

The following example illustrates how to retrieve the address of a local variable and store it in a pointer variable:

```
FUNCTION Start()
    LOCAL l:=5 AS LONG
    LOCAL p AS PTR

    p := @l
    ? p
```

Displaying the value of a pointer will show the actual address value. This information is usually not of much use.

Pointers are dereferenced using a dereference operator consisting of a type name followed by the pointer in parenthesis. For example:

```
FUNCTION Start()
    LOCAL l:=5 AS LONG
    LOCAL p AS PTR

    p := @l
    ? LONG(p)       // Output: 5
```

Note that with untyped pointers the compiler cannot check, if the data type given in the dereference operation, is the correct type of the item the pointer is pointing to.

Similarly, the value also can be changed by retrieving the value that a pointer is pointing to:

```
FUNCTION Start()
    LOCAL l:=5 AS LONG
    LOCAL p AS PTR

    p := @l
    LONG(p) := 10
    ? l             // 10
```

Pointers come in very handy when working with structures and dynamically allocated heap memory (using MemAlloc()). Since the use of untyped pointers does not allow any kind of compile checking, using typed pointers is highly recommended.

Typed Pointers

A pointer was nothing more than an address to the CA-Visual Objects 1.0 compiler. Visual Objects has extended the view of a pointer to include semantic information. The compiler uses this information to identify the type of data held in the memory area the pointer addresses. This information now enables us to do pointer arithmetic.

Pointer typing is applicable to the majority of Visual Objects basic data types. Their usage is, however, limited in the case of the Visual Objects dynamic data types.

Typed pointers can be used without restriction with the following data types:

Data Type	Description
DATE	Date value
LOGIC	Logical value
INT	Signed integer: under Windows, 32 bits
SHORTINT	Signed 16-bit integer; identical to ANSI C short
LONGINT	Signed 32-bit integer; identical to ANSI C long
BYTE	Unsigned 8-bit integer (a byte); identical to ANSI C unsigned character
WORD	Unsigned 16-bit integer (a word); identical to ANSI C unsigned short
DWORD	Unsigned 32-bit integer (a double word); identical to ANSI C unsigned long
REAL4	32-bit floating point number; identical to ANSI C float
REAL8	64-bit floating point number; identical to ANSI C double

Furthermore, a restricted use of typed pointers is possible with the following dynamic data types:

Data Type	Description
FLOAT	Floating point number: under Windows, 80 bits
OBJECT	General object
<idClass>	Object of a specific class
STRING	Dynamic string

<idStructure> and <idUnion> are representative of specific structures and unions and the AS form of these data types can be interpreted as pointers, due to their internal implementations.

All pointer data types (typed pointers, PTR, PSZ, AS Structures, REF — references) are now considered compatible in Visual Objects. It is therefore possible to do a logical comparison or an assignment employing these various types of equivalent pointers.

The compiler, for example, loses its semantic information in assigning a typed pointer variable to a PTR variable.

Caution! *This type of pointer usage is designed for advanced users with an understanding of the Visual Objects internals.*

Declaration of Typed Pointers

Typed pointers can be declared to the compiler by using either the LOCAL or the GLOBAL statement. Either of these declarations are possible:

```
[STATIC] LOCAL <Variable List> AS <Data Type> PTR
[STATIC] GLOBAL <Variable> AS <Data Type> PTR
```

It is possible to initialize the typed pointer by the declaration. All typed pointers that are not initialized by the declaration are assigned the initial value of NULL_PTR. The following piece of code demonstrates the declaration and initialization of a typed pointer:

```
LOCAL DIM fpArr[10] AS REAL4
LOCAL preal4:=@fpArr AS REAL4 PTR
```

In this example we define a REAL4 pointer (preal4) and initialize it to point to the first element of a one dimensional array with a maximum of 10 REAL4 elements.

Dereferencing Typed Pointers

With Visual Objects, it is sensible to dereference a typed pointer because the compiler then knows the type of data the dereferenced pointer addresses.

The syntax used for dereferencing typed pointers is similar to the type conversion syntax that you may already be familiar with from the previous version of Visual Objects: (<idType>(<Value>)). Typed pointer dereferencing is achieved with the following syntax:

```
PTR(<Pointer Variable>)
```

The pointer variable can be typed to any of these data types:

- DATE
- LOGIC
- INT
- SHORTINT
- LONGINT
- BYTE
- WORD
- DWORD
- REAL4
- REAL8

Using this syntax, we can obtain the value of the first element of the array (the element preal4 addresses) in the following manner:

```
PTR(preal4)
```

Pointer Arithmetic

Performing arithmetic operations with typed pointers is alluded to the inherently semantic information they incur.

Pointer arithmetics are restricted to the operations of addition and subtraction.

Typed pointer addition/subtraction is unlike the normal addition/subtraction in the following aspects:

Firstly, though it is a binary operation like the normal addition/subtraction, typed pointer addition/subtraction will only be carried out in the case where the first operand of the addition/subtraction operation is a typed pointer and its second operand represents a numeric value.

The second difference is an immediate implication of the first: typed pointer addition is as opposed to the normal addition not commutative. If we add a typed pointer to a numeric value, the compiler in a first step automatically promotes the type of the numeric value to a pointer and subsequently does a "normal" pointer addition. This is not equivalent to the typed pointer addition.

In typed pointer addition/subtraction, the resulting scaled pointer obtained with the second operand (the numeric value) is added/subtracted to/from the first operand (the typed pointer). The scaling is carried out using the size of the data type being addressed by the first operand.

Pre/Post incrementing or decrementing typed pointers are special cases of the general typed pointer addition or subtraction. In these special cases the compiler automatically sets the value of the second operand to one.

In the example below, the statement ptemp++ does a typed pointer addition. After the statement is carried out, the variable ptemp then addresses the subsequent REAL4 element in the array, assuming that ptemp points to the first array element before the statement is executed. After execution ptemp will be pointing to the second element. In this particular case, the value of the pointer has not only been increased by one (the magnitude of the second operand), but by 1*4 (since the _SizeOf(REAL4) is 4).

```
LOCAL prandom, ptemp AS REAL4 PTR
LOCAL w AS WORD

// allocate memory area for sparse array of 100
// REAL4 elements
prandom := PTR(REAL4, MemAlloc(100 * _SizeOf(REAL4))

// initialize sparse array
ptemp := prandom
FOR w:=1 UPTO 100
     PTR(ptemp) := 0.0
     ptemp++
NEXT

// randomly access/assign values to elements
// in sparse array
prandom[10] := 5.75
prandom[50] := 10.25
```

The subtraction of pointers further poses a particular result. Typed pointer subtraction has a specific interpretation in the case where both operands of the operation represent pointers. The result of the operation in this particular case gives the number of elements (with respect to the size of the data being addressed by the first pointer) between these two elements.

Executing the expression ptemp - prandom after our initialization loop in the above example will yield the value 100.

Chapter

22

Variables, Constants, and Declarations

The previous chapter, "Data Types," introduced you to the concept of categorizing data according to the kind of information it represents and went on to describe how the Visual Objects language provides you with certain predefined data types designed to handle a wide variety of information.

This chapter introduces you to variables and constants, shows you how to create and declare them, and explains how to associate them with a particular data type.

Terminology

There are several terms used throughout this chapter that you may not know. Many of the terms are defined as they are introduced, but this section defines some of the more basic terms while describing the general nature of variables and constants.

The term *variable* is used, quite literally, to describe a value that is subject to change. Similarly, the term *constant* describes a value that always stays the same.

You define variables and constants by specifying names for them and assigning values to the names. Then, when you refer to a variable or constant name that you have defined in your application, its value is returned.

Variable names and *constant names* must be legal *identifiers*. That is, the name must begin with a letter of the alphabet and can contain only letters, numbers, and the underscore character. Names are fully significant up to 64 KB characters and must not conflict with reserved words. (The reserved words are listed in Appendix B of this guide.)

There are two ways to define a variable name. You can create the variable at runtime by assigning a value to an identifier, or you can declare a variable name to the compiler. Once you have created or declared a variable name, you can change its value at any time during your application.

Constants are a similar case. You could create a variable at runtime and never change its value throughout your application and, in a sense, this variable could be called a constant. However, you can also specifically declare constants as compiler entities and thereby gain certain advantages. True compiler constants are the subject of this chapter.

Once you define a variable or constant name, it continues to exist and to possess a value until it is released from memory. Some variables are released automatically, while others must be explicitly released. Some variables are never released. The duration of a variable's life is referred to as its *lifetime*.

Visibility refers to the conditions under which a variable is accessible to the program during execution. Some variables, even though they have been created and assigned a value, may not be visible under certain conditions.

Scope is used to refer collectively to the lifetime and visibility of a variable or constant. This term is also used to refer to the lexical unit to which a declaration applies.

Field Variables

One of the most important and powerful aspects of Visual Objects is its database system that allows you to create and manipulate structured database files within your applications.

You create a database as a disk-based file by defining a structure, including field names, data types, and lengths, in a distinct step during the application development process using DBServer Editor as described in the *IDE User Guide*. Afterwards, you design and code the programs that rely on this structure to use and manipulate the data within the database file.

You can look at a database as a table, containing rows (or *records*) and columns (or *fields*) as its basic components. Once a database file exists on disk, within your application you can open it in a *work area*, move the work area *record pointer* to change the *current record*, and access the data in a particular column of the current record using its field name.

The term *field variable* is a synonym for a database field name. Field variables typically exist and possess values before the application begins execution, and they continue to exist after the application terminates. They are visible to all entities in an application as long as their corresponding database file is open.

Variable Type	Lifetime	Visibility
FIELD	Persistent—while database exists	Application—while database is open

Note: All fields are visible throughout the application, but the FIELD declaration statement, described later in this section, applies only to the entity in which it occurs.

DBServer Field References

If you are using the DBServer (or another DataServer) class, you refer to field names as if they were instance variables, using the database server object and the send operator (:):

```
FUNCTION ListAll()
     LOCAL oDBNames
     oDBNames := DBServer{"names"}
     DO WHILE !oDBNames:EOF
          ? oDBNames:Name, oDBNames:Phone
          oDBNames:Skip()
     ENDDO
```

Using the DBServer class is the preferred method for database access because the code will automatically support multiple instantiations of itself, a very compelling benefit when programming in a GUI environment. With the exception that the data types of the data server objects are not declared (this concept is introduced later in this chapter in the Strongly Typed Variables section), this code is as efficient and unambiguous as possible. See Chapter 7, "Data Server Classes," earlier in this guide for more information on the benefits of using data servers.

Aliased Field References

If you are not using one of the data server classes to access your database file, you need to concern yourself with the possibility of ambiguous variable references (fields and variables with the same name). Consider the following example:

```
FUNCTION ListAll()
     USE names NEW
     DO WHILE !EOF()
          ? Name, Phone
          DBSkip()
     ENDDO
```

When this code is executed, it will display the fields *Name* and *Phone* in the Names database, provided that they exist. If the field names do not exist, the program will look for variables of the same name and display them. If the variables are not found, a runtime error will occur.

But, none of this can be decided at compile time because the compiler has no way of knowing anything about *Name* and *Phone* and must, therefore, generate a good deal of code to handle all the possibilities, causing your program to be slower than necessary.

To eliminate runtime overhead and make your application as efficient as possible, you should qualify all field names (and function calls) using the database alias name:

```
FUNCTION ListAll()
     USE names New
     DO WHILE !Names->(EOF())
          ? Names->Name, Names->Phone
          Names->(DBSkip())
     ENDDO
```

This code is as unambiguous and efficient as possible because every field reference and function call is qualified.

FIELD Declarations and _FIELD Aliases

Instead of using the database alias to qualify each field reference, you can use the _FIELD alias or declare the field names to the compiler using the FIELD statement:

```
FUNCTION ListAll()
     USE names NEW
     DO WHILE !Names->(EOF())
          ? _FIELD->Name, _FIELD->Phone
          Names->(DBSkip())
     ENDDO

FUNCTION ListAll()
     FIELD Name, Phone IN Names
     USE names
     DO WHILE !Names->(EOF())
          ? Name, Phone
          Names->(DBSkip())
     ENDDO
```

Both of these solutions eliminate ambiguity, but neither is as desirable as explicitly qualifying field names with a particular alias.

Note: Like all variable declaration statements, the FIELD statement applies only to the entity in which it occurs, not the entire application. The statement does not, however, limit the visibility of the fields it declares to the current entity. All fields in all open database files have application-wide visibility.

Recap

There are a lot of ways to deal with fields. The following list summarizes them, in reverse order of preference:

- Use the field name, unqualified, and let the compiler generate runtime code to figure out what you mean. This technique has several disadvantages:
  1. The program will be inefficient because of the runtime overhead involved.
  2. The compiler and you may not agree on the runtime decision.
  3. The program will be prone to errors because of the possibility of conflicting field and variable names, which can go undetected.
  4. The program will require that a particular work area be selected when it runs.
  5. The program will not support opening the same database in multiple work areas.
- Use the _FIELD alias or the FIELD declaration statement. This technique solves all the problems listed above except numbers 4 and 5.
- Use an alias qualifier for all field names and function names. This will eliminate number 4 but not 5.
- Use the DBServer class to eliminate number 5.

Dynamically Scoped Variables

Visual Objects provides support for *dynamically scoped* variables that are created and maintained completely at runtime. The term dynamically scoped refers to the fact that the scope of these variables is not limited by the entity in which the variable is created.

Variable Type	Lifetime	Visibility
PRIVATE	Until creator returns or until released	Creator and called routines
PUBLIC	Application or until released	Application

The data type of a dynamically scoped variable changes according to the contents of the variable. For this reason you will often hear this type of variable described as *dynamic*, or *polymorphic*.

Warning! *Dynamically scoped variables are supported only if the Undeclared Variables compiler option is checked. Otherwise, any reference to a dynamically scoped variable will result in a compiler error.*

Dynamically scoped variables are provided mainly for CA-Clipper/Xbase compatibility; however, they are very useful in certain circumstances. For instance, they let you develop rapid prototypes and have certain inheritance properties that you may find hard to resist.

You must be aware, however, that using them comes at a cost. Consider these points:

- Because they are not resolved at compile time, these variables require overhead in the form of runtime code, making your application larger and slower than necessary.
- No compile time checking for type compatibility is possible with these variables.
- Using the inheritance properties of these variables defies one of the basic tenets of modular programming and may lead to maintenance and debugging problems down the line. Furthermore, this practice will make the transition to lexically scoped and typed variables more difficult.

This section explores dynamically scoped variables fully, but Visual Objects has several options for variable declarations that you will want to explore before choosing to use this variable class. The next two sections in this chapter introduce you to Lexically Scoped Variables and Strongly Typed Variables, which you may find useful.

Important! *For the sake of illustration, some of the examples in this section use unorthodox programming practices. Using the inheritance properties of public and private variables instead of passing arguments and returning values is not recommended.*

Private

Private is one of the two types of dynamically scoped variables, and there are several ways to create a private variable:

- Assign a value to a non-existent variable name (for example, x := 10). The variable takes on the data type of its assigned value until you assign a new value to it. (x is numeric, but the assignment x := "Ms. Jones" changes it to a string.)
- List the variable name as part of a PRIVATE statement. If you do not make an assignment at this time, the variable takes on the NIL value and data type; otherwise, it takes on the data type of its assigned value. You can assign a new value (and a new data type) to the variable at any time:

```
PRIVATE x := 10, y
```

creates *x* as a numeric value of 10 and *y* as NIL. Later on, the assignments:

```
x := "Ms. Jones"
y := TRUE
```

change these types to string and logical.

- List the variable name as part of a PARAMETERS statement within a FUNCTION or PROCEDURE definition. The variable takes on the data type of its associated argument when the routine is called, or NIL if the argument is omitted. You can assign a new value (and a new data type) to the variable at any time.

Private variables have these properties:

- You can access them within the creating routine and any routines called by the creator. In other words, private variables are automatically inherited by called routines without having to pass them as arguments.
- You can hide them from a called routine by explicitly creating a private (using PRIVATE or PARAMETERS) or declaring a local (using LOCAL) variable with the same name in the called routine.
- They are automatically released from memory when the creator returns to its calling routine, or you can release them explicitly using RELEASE, CLEAR ALL, or CLEAR MEMORY.

In this example, the function Volume() expects three arguments, or parameters, to be passed. When the function is called, it creates three private variables, *nLength*, *nWidth*, and *nHeight* to accept the arguments. Because they are created with the PARAMETERS statement, any higher-level variables (either public or private) created with these names are temporarily hidden, preventing their values from being overwritten in memory:

```
FUNCTION Volume()
    PARAMETERS nLength, nWidth, nHeight
    RETURN nLength * nWidth * nHeight
```

In the next example, a modified version of Volume() creates a private variable (assuming no other variable name *nVolume* is visible) to store its return value. If the variable *nVolume* exists prior to calling Volume() and is visible to Volume() (for example, *nVolume* may be public or private to the routine that called Volume()), its value is overwritten in memory and will remain changed when the function returns to its calling routine:

```
FUNCTION Volume()
    PARAMETERS nLength, nWidth, nHeight
    nVolume := nLength * nWidth * nHeight
    RETURN nVolume
```

In this version, Volume() specifies the *nVolume* variable as PRIVATE. Doing this temporarily hides any higher-level variable (either public or private) with the same name, preventing its value from being overwritten in memory:

```
FUNCTION Volume()
     PARAMETERS nLength, nWidth, nHeight
     PRIVATE nVolume := nLength * nWidth * nHeight
     RETURN nVolume
```

Public

The second category of undeclared variable is public. Public variables have application-wide lifetime and visibility, and you can define them in only one way:

- List the variable name as part of a PUBLIC statement. If you do not make an assignment at this time, the variable takes on a value of FALSE (or NIL for array elements); otherwise, it takes on the data type of its assigned value. You can assign a new value (and a new data type) to the variable at any time.

Public variables have these properties:

- Once they are created, you can access them anywhere in the application. In other words, public variables are automatically inherited by all routines in the application without having to pass them as arguments or post them as return values.
- You can hide them from a routine by explicitly creating a private (using PRIVATE or PARAMETERS) or declaring a local (using LOCAL) variable with the same name.
- They are not released from memory until you explicitly release them using RELEASE, CLEAR ALL, or CLEAR MEMORY.

In this example, the function Volume() is defined without arguments. Instead, the calling routine, Compute(), creates three public variables, *nLength*, *nWidth*, and *nHeight* that are automatically visible to Volume():

```
PROCEDURE Compute()
     PUBLIC nLength := 5, nWidth := 2, nHeight := 4
     ? Volume()      // Result: 40

FUNCTION Volume()
     RETURN nLength * nWidth * nHeight
```

In the next example, a modified version of Volume() creates a public variable to store the computed volume, getting around having to return a value to the calling routine. Since *nVolume* is public, it is not released from memory when Volume() returns:

```
PROCEDURE Compute()
     PUBLIC nLength := 5, nWidth := 2, nHeight := 4
     Volume()
     ? nVolume       // Result: 40

FUNCTION Volume()
     PUBLIC nVolume
     nVolume := nLength * nWidth * nHeight
```

Variable References

Once a public or private variable is created as demonstrated in the previous two sections, you obtain its value by referring to its name. You might display the value of a variable using a built-in command or function:

```
? nVolume
QOut(nVolume)
```

or use its value as part of an expression:

```
Str(nVolume, 10, 2) + " cubic feet"
```

For dynamically scoped variables, you can use the _MEMVAR alias to qualify a variable reference. In some cases, you may have to do this in order to help the compiler resolve what might otherwise be an ambiguous reference (for example, if you have a field variable with the same name as a memory variable and want to use the memory variable in an expression).

Note: MEMVAR is an abbreviation for *memory variable*, a term that is synonymous with dynamically scoped variable.

Assuming that the database file Measures has fields named *nLength*, *nWidth*, and *nHeight*, this example calls Volume() using the field variable values:

```
FUNCTION Calculate()
     PRIVATE nLength := 5, nWidth := 2, nHeight := 3
     USE measures
     ? Volume(nLength, nWidth, nHeight)
     ...
```

To force the function to use the private variables instead of the field variables, you could use the _MEMVAR-> (or, more simply, M->) alias to qualify the variable names:

```
FUNCTION Calculate()
     PRIVATE nLength := 5, nWidth := 2, nHeight := 3
     USE measures
     ? Volume(_MEMVAR->nLength, _MEMVAR->nWidth, _MEMVAR->nHeight)
     ...
```

Of course, it is better to avoid ambiguous situations like the one described above by taking care to have unique field and variable names, but the point is that the compiler has certain default rules for handling ambiguous references. If you do not want to be at the mercy of those defaults, it is best to qualify variable names in all cases.

MEMVAR Declarations

Although you may hear them referred to as such, the statements mentioned so far in the discussion of dynamically scoped variables are not declarations. The term *declaration* refers to a statement whose purpose is to inform the compiler of something—PRIVATE, PARAMETERS, and PUBLIC are statements that generate memory variables at runtime.

In fact you never have to declare a dynamically scoped variable to the compiler, which is the reason for their inefficiency. Because they are not created using compile-time declaration statements, the compiler has to generate runtime code for handling such issues as type translation, memory management, and resolving ambiguous references to variable names since it is possible for several variables with the same name to be visible at one time.

You can, however, declare dynamically scoped variables with the MEMVAR statement and they will be created as PRIVATE variables:

```
FUNCTION Calculate()
     MEMVAR nLength, nWidth, nHeight
     nLength := 5
     nWidth := 2
     nHeight := 3
     USE measures
     ? Volume(nLength, nWidth, nHeight)
     ...
```

In this case, the MEMVAR statement causes memory variables to take precedence over field variables with the same names, causing Volume() to be called with the private variables.

Using MEMVAR to declare dynamically scoped variable names to the compiler may make your programs slightly more efficient (especially if you have lots of ambiguous references); however, it will not eliminate the runtime overhead of these variables. The next section shows you how to declare variable names to the compiler, and thus avoid ambiguous variable references all together.

Lexically Scoped Variables

In the previous section, you were introduced to dynamically scoped variables and were informed of their inherent drawbacks. To help you overcome some of these drawbacks, you can declare variables (called *lexically scoped* variables) based on the lexical unit in which they will be used. A *lexical unit* is an executable entity (such as a function, procedure, or code block) or a module.

Like dynamically scoped variables, lexically scoped variables can also be polymorphic (their data type can change during the course of an application). However, lexically scoped variables are resolved at compile time rather than runtime and are, therefore, much more efficient. In addition to the increased program efficiency that they offer, lexically scoped variables also serve to enforce modular programming principles that will make your programs more robust.

This section explores lexical scoping as it applies to polymorphic variables, but Visual Objects also supports strong data typing (discussed in the Strongly Typed Variables section below) which will further increase the efficiency and robustness of your applications.

Local

Local is one of two types of lexically scoped variables. There are two ways to create local variables:

- List the variable name as part of a LOCAL statement. If you do not make an assignment at this time, the variable takes on the NIL value and data type; otherwise, it takes on the data type of its assigned value. You can assign a new value (and a new data type) to the variable at any time. For example:

  ```
  LOCAL x := 10, y
  ```

  creates *x* as a numeric value of 10 and *y* as NIL. Later on, the assignments:

  ```
  x := "Ms. Jones"
  y := TRUE
  ```

  change these types to string and logical.

- List the variable name as a parameter in parentheses as part of a FUNCTION, METHOD, or PROCEDURE statement or in vertical bars as part of a code block definition. The variable takes on the data type of its associated argument when the routine is called, or NIL if the argument is omitted. You can assign a new value (and a new data type) to the variable at any time. For example:

  ```
  FUNCTION Area (x,y)
  RETURN x * y
  cbVar := {|xVAR| xVAR+3}
  ```

Local variables have a lifetime and visibility that is limited to the entity in which they are declared:

- You can access them within the declaring routine only—they are not automatically inherited by called routines like private variables. You must pass them as arguments in order to make them accessible in a called routine.
- They are automatically released from memory when the creator returns to its calling routine. You cannot explicitly release them from memory.

This example declares the variable *nVar* to the compiler using the LOCAL statement:

```
FUNCTION SomeFunc()
    LOCAL nVar := 10
    .
    . <Executable statements>
    .
    NextFunc()
    RETURN TRUE
```

When SomeFunc() is called at runtime, several things happen:

- *nVar* is initialized to a value of 10—any variable that has the same name is temporarily hidden from view.
- When the function NextFunc() is executed, *nVar* still exists but cannot be accessed because it is not visible.
- When the execution of SomeFunc() is complete, the local copy of *nVar* is destroyed, and any variable with the same name in the calling program is once again accessible.

Thus, if you want the value of *nVar* to be visible to a called routine, you must pass it as a parameter (for example, NextFunc(nVar)). Similarly, if you need the value of *nVar* in the calling routine, you must return it (for example, RETURN nVar). In this way, using local variables forces you to adhere to basic modular programming principles.

Note: If a routine is invoked recursively, each activation creates a new set of local variables.

STATIC as a Lifetime Modifier

You may use the STATIC keyword as a lifetime modifier within a LOCAL declaration. Doing this prevents the variable from being released from memory when the creator returns to its calling routine.

Thus, a static local has an application lifetime. The visibility, however, like a regular local, is limited to the creating entity. In other words, static locals have these properties:

- You may access them within the declaring routine only.

- They are retained in memory when the creator returns to its calling routine, but they are no longer visible. You cannot explicitly release them from memory.
- On subsequent calls to the creating routine, they become visible again.

The scoping rules for locals and static locals are different because STATIC changes the lifetime of a local variable:

Variable Type	Lifetime	Visibility
LOCAL	Creator	Creator
STATIC LOCAL	Application	Creator

Note: When an application containing static variable declarations is invoked, the variables are created and initialized (using the initial value specified with STATIC LOCAL or NIL) before the beginning of program execution. Thus, initial values are assigned only once per application run, not each time the creator is called.

This example declares the variable *nCount* to the compiler using the STATIC LOCAL statement:

```
FUNCTION CountMe()
    STATIC LOCAL nCount := 0
    .
    . <Executable statements>
    .
    ? "This is call number", ++nCount
```

Note: Initial values that you define as part of the STATIC LOCAL statement must evaluate to constants at compile time (for example, literals and simple expressions involving only operators, literals, and DEFINE constants.) Then, at runtime, several things happen:

- When the application containing CountMe() is invoked, *nCount* is initialized to zero before the beginning of program execution. This way, *nCount* does not get set back to zero each time CountMe() is called.
- The first time you call CountMe(), *nCount* becomes visible with its initial value of zero—any variable that has the same name is temporarily hidden from view. The function then increments *nCount* to one before displaying "This is call number 1."
- When the execution of CountMe() is complete, the local copy of *nCount* is retained with its new value; however, *nCount* is no longer visible.
- The next time you call this function, *nCount* has its previous value of one which is incremented before the function displays "This is call number 2" and returns.

- This cycle is repeated each time you call CountMe() so that the function displays an accurate count of how many times it has been called.

Tip: You can use STATIC as an abbreviated syntax for STATIC LOCAL. Thus in the previous example, you could substitute STATIC nCount := 0 for STATIC LOCAL nCount := 0.

Global

Global is another category of lexically scoped variable. You can define global variables in only one way:

- List the variable name as part of a GLOBAL statement. If you do not make an assignment at this time, the variable takes on a value of NIL; otherwise, it takes on the data type of its assigned value. You can assign a new value (and a new data type) to the variable at any time.

Note: Global variables are compiler entities. This means, among other things, that you cannot include a GLOBAL declaration within another entity such as a function definition. You can only declare one global variable per GLOBAL statement. Initial values that you define as part of the GLOBAL statement, must evaluate to constants at compile time (for example, literals and simple expressions involving only operators, literals, and DEFINE constants.)

Global variables have application-wide lifetime and visibility:

- You can access them anywhere throughout the application. In other words, global variables are automatically inherited by all routines in the application without having to pass them as arguments or post them as return values.
- You can hide them from a routine by explicitly declaring another variable (for example, using LOCAL or MEMVAR) with the same name.
- You cannot explicitly release them from memory—their lifetime is guaranteed throughout the application.

STATIC as a Visibility Modifier

You can use the STATIC keyword as a visibility modifier within a GLOBAL declaration. Doing this restricts access of the variable to the *module* in which it is declared. A module is a means for organizing entities, such as functions and classes, and is considered to be a lexical unit.

Like a regular global, a static global has an application lifetime, but its visibility is limited to the module in which it is declared. In other words, static globals have the following properties:

- You can access them anywhere within the declaring module. In other words, only those routines defined in the same module have access to static globals.
- You can hide them from a routine by explicitly declaring another variable (for example, using LOCAL or MEMVAR) with the same name.
- You cannot explicitly release them from memory—their lifetime is guaranteed throughout the application.

The scoping rules for globals and static globals are different because STATIC changes the visibility of a global:

Variable Type	Lifetime	Visibility
GLOBAL	Application	Application
STATIC GLOBAL	Application	Module

Strongly Typed Variables

In the previous section, you were introduced to compiler declarations as they apply to polymorphic variables. These variables are declared at compile time, giving them certain advantages over undeclared variables, but their data types are still dynamic. For example, there is nothing to prevent you from changing the data type of a variable from numeric to string:

```
FUNCTION SomeFunc()
    LOCAL nVar := 10
    .
    . <Executable statements>
    .
    nVar := "New character value"
    RETURN nVar
```

You can, however, declare the data types of variables (called *strongly typed* variables) within declaration statements. Strongly typed variables, because they are declared, are lexically scoped and adhere to all of the scoping and other rules stated in the Lexically Scoped Variables section.

The ability to specify data types for your program variables is advantageous for several reasons:

- All type compatibility checks can be performed at compile time, allowing you the advantage of debugging your programs more quickly and easily.
- You can greatly increase the integrity of your data and, therefore, the robustness of your application if you use strongly typed variables.

- Because they involve no runtime overhead, using strongly typed variables significantly increases the efficiency of your code.

Tip: The Visual Objects compiler has a type inferencing option. If you use lexically scoped variable declarations in your programs, you can use this option to achieve some of the benefits of strong data typing without actually specifying data types in your existing declarations.

Data Type Declarations

Both the LOCAL and GLOBAL statements provide syntax for specifying the data types of the variables they declare using the AS keyword. The FUNCTION and PROCEDURE definition statements provide a similar syntax and semantic, including the ability to strongly type the return value of a function.

Note: For information about using strongly typed arguments with METHOD, see the online help system; and for information about using them with code block definitions, see Strong Typing under the Creating Code Blocks section in Chapter 26, "Code Blocks," later in this guide.

Once you declare the data type of a variable, attempting to use it in another context will result in a compiler error. The following table lists the data types available.

Data Type	Description
ARRAY	Dynamic array
CODEBLOCK	Compile-time code block
DATE	Date value
LOGIC	Logical value
OBJECT	General object
<*idClass*>	Object of a specific class
STRING	Dynamic string
SYMBOL	Symbol
<*idStructure*>	Specific structure
USUAL	Explicit declaration for a polymorphic value—identical to an untyped variable

VOID	In function definitions only, used to indicate no return value

Data Type	Description
INT	Signed integer: under Windows, 32 bits
FLOAT	Floating point number: under Windows, 80 bits
SHORTINT	Signed 16-bit integer; identical to ANSI C short
LONGINT	Signed 32-bit integer; identical to ANSI C long
BYTE	Unsigned 8-bit integer (a byte); identical to ANSI C unsigned character
WORD	Unsigned 16-bit integer (a word); identical to ANSI C unsigned short
DWORD	Unsigned 32-bit integer (a double word); identical to ANSI C unsigned long
REAL4	32-bit floating point number; identical to ANSI C float
REAL8	64-bit floating point number; identical to ANSI C double
PSZ	Pointer to a zero-terminated character string
PTR	Address value

Thus, in the example given earlier in which a LOCAL numeric variable was declared, you could have stated its type. In this example, *iVar* is strongly typed as an integer (INT), and the statement iVar := "New character value" produces a compiler error:

```
FUNCTION SomeFunc()
    LOCAL iVar := 10 AS INT
    .
    . <Executable statements>
    .
    iVar := "New character value"       // Compiler error!
    RETURN iVar
```

Type declaration is straightforward, for the most part. You simply use the proper keyword in an AS clause as part of the declaration statement (such as GLOBAL, LOCAL, or FUNCTION). Specifying data types for function parameters and return values, as well as some of the more obscure data types listed in the table above, however, deserve further discussion and are presented as topics later in this section.

Initial Values

When you declare variables using data types, you can assign initial values in the LOCAL and GLOBAL declaration statements similar to the way in which you assign initial values for polymorphic variables:

```
GLOBAL iCounter := 0 AS INT
LOCAL fTemp := 98.6 AS FLOAT
```

Alternatively, you can assign initial values after the declaration:

```
FUNCTION CalcTemp(cAnimal)
     LOCAL fTemp AS FLOAT
     DO CASE
     CASE cAnimal == "Human"
          fTemp := 98.6
     .
     . <More cases>
     .
     ENDCASE
```

If you declare a strongly typed variable and do not make an initial assignment, the variable will take on a default value, depending on its data type, as defined in the following table. The NULL_ symbols listed in the table are system-defined constants representing the null value for each data type.

There is no benefit in initializing a LOCAL variable in the declaration. You won't be able to step through the declarations when debugging your application if you use the initialization. It is therefore recommended to do the following:

```
LOCAL x AS INT
x := 5
```

instead of

```
LOCAL x := 5 AS INT
```

Data Type	Default Initial Value
ARRAY	NULL_ARRAY (see Note below)
CODEBLOCK	NULL_CODEBLOCK
DATE	NULL_DATE
LOGIC	FALSE

OBJECT, <*idClass*>	NULL_OBJECT
STRING	NULL_STRING
SYMBOL	NULL_SYMBOL
USUAL	NIL

Data Type	Default Initial Value
SHORTINT, INT, LONGINT, FLOAT, BYTE, WORD, DWORD, REAL4, REAL8	0
PSZ	NULL_PSZ
PTR, <*idStructure*>	NULL_PTR

You can rely on the NULL value initialization without having to assign a NULL value explicitly. For example:

```
LOCAL x := 0 AS INT
```

or

```
LOCAL x AS INT
X :=0
```

will not be caught by the compiler, causing the NULL value assignment to be performed twice, which is ineffective.

Note: NULL_ARRAY is the initial value for an array declared without dimension specifications (for example, LOCAL <*idArray*> AS ARRAY). If you specify array dimensions as part of the declaration you are, in effect, making an assignment to the array and, therefore, NULL_ARRAY will not apply. As an example:

```
LOCAL aValues AS ARRAY
```

yields *aValues* as a NULL_ARRAY, whereas:

```
LOCAL aValues[10] AS ARRAY
```

does not. The latter statement is equivalent to:

```
LOCAL aValues AS ARRAY
aValues := ArrayCreate(10)
```

which initializes each element in the array *aValues* to NIL.

Tip: Instead of using NULL_ARRAY and NULL_OBJECT, you can compare a strongly typed array or object to NIL to determine if it is in an uninitialized state, and you can assign the NIL value to return an array or object to an uninitialized state. In these cases, NIL is identical to the corresponding NULL_ constant and does not, of course, change the data type of the array or object to NIL.

Typing Parameters and Return Values

You can specify the data type of a local function parameter (and the function return value):

```
FUNCTION SomeFunc(iVar AS INT) AS STRING
    .
    .<Executable statements>
    .
    RETURN cString
```

In this example, the (iVar AS INT) clause specifies the function argument as an integer, and the AS STRING clause following the parameter list declares the function return value as a string (the VOID data type is limited to use in this particular context).

When you call SomeFunc(), the variable that you use as a function argument must be strongly typed as INT, and any operation that you perform using the return value must be valid for the string data type. Otherwise, compiler errors will be generated.

Important! *Declaring data types of function parameters and return values is slightly more complicated than indicated here. In fact, doing so imposes certain restrictions on the use of the function, including your ability to skip parameters when calling the function, to use the function in a macro expression, and to freely pass arguments by reference or value. For more information on this subject, including specific details regarding these restrictions, see Chapter 27, "Functions and Procedures," later in this guide.*

Class Names as Data Types

The subject of classes that you define in your applications has not yet been presented. However, Visual Objects does give you this capability, and this subject is discussed in detail in Chapter 25, "Objects, Classes, and Methods," in this guide.

Briefly, you define class entities using the CLASS statement and later use the class name to create *instances* of the class. Each class instance that you create is called an *object*, which is also its data type. Thus, you can define a class and create a declared instance of the class:

```
CLASS Animal                                    // Define Animal class
    .
    . <Instance declarations defining the class>
    .

FUNCTION UseClass()
    LOCAL oJag AS OBJECT                // Declares oJag as an object, any class

    oJag := Animal{}                            // Creates oJag as Animal class
    .
    . <Executable statements>
    .
```

The AS OBJECT clause allows you to assign any instance of any class to the variable *oJag*. For example, if you had a class named Car defined in your application, you could use the statement oJag := Car{} within the UseClass() function without causing an error.

You can further limit the type of object that can be assigned to a variable by naming the class in the declaration statement:

```
FUNCTION UseClass()
    LOCAL oJag AS Animal                // Declares oJag as an instance of the Animal class

    oJag := Animal{}                            // Creates oJag as Animal class
    .
    . <Executable statements>
    .
```

The AS Animal clause means that you can only assign instances of the Animal class (and its subclasses) to *oJag*. If the Class Checking compiler option is checked, including the statement oJag := Car{} within the UseClass() function would cause a compiler error.

As you can probably guess, declaring a variable using a specific class gives your application speed and code size advantages over using the OBJECT declaration, and being more specific about the declaration of a variable leaves less room for programming errors. However, you may have a situation in which you want a name to be used as more than one class of object, in which case the OBJECT declaration still offers you efficiency advantages over not declaring the variable at all.

Strong Typing Instance Variables

There is another level of data typing available within a CLASS definition. Specifically, you can declare AS data types for instance variables, although doing so is optional:

```
CLASS Animal                    // Define Animal class
     EXPORT cGenus AS STRING
     EXPORT cSpecies AS STRING
     INSTANCE iPopulation AS INT
     .
     . <Other class instances>
     .
```

Instance variables are a special category of variable that you can declare only within a CLASS entity. For more information on instance variables, see the "Objects, Classes, and Methods" chapter later in this guide.

Structure Names as Data Types

The subject of data structures that you define in your applications has not yet been presented in this guide. However, Visual Objects does give you this capability.

Briefly, you define structure entities using the STRUCTURE statement and later use the structure name as part of a declaration statement to declare structure variables. Structure variables are complex, the components being variables that you declare within the structure and access using the dot operator (.).

AS vs. IS Typing

You must declare structure variables using either the AS or the IS keyword. The difference between these two declaration methods is that:

- IS automatically allocates the memory needed to hold the structure and deallocates the memory when the declaring entity returns.
- AS requires that you allocate memory using MemAlloc() before initializing structure variables. You must also deallocate the memory used by the structure variable using MemFree() before the declaring entity returns.

Important! *IS typing is much simpler than AS typing, and in most cases should satisfy your requirements for using structures. AS typing is recommended for experienced systems programmers.*

This example illustrates IS structure typing:

```
STRUCTURE SysOne                // Define SysOne data structure
     MEMBER iAlpha AS INT
     MEMBER pszName AS PSZ

FUNCTION UseStruct()
     LOCAL strucVar IS SysOne
     strucVar.iAlpha := 100
```

```
.
. <Statements that access structure members>
.
```

This example illustrates AS structure typing, with its required memory allocation and deallocation:

```
STRUCTURE SysOne                // Define SysOne data structure
     MEMBER iAlpha AS INT
     MEMBER pszName AS PSZ

FUNCTION UseStruct()
     LOCAL strucVar AS SysOne
     strucVar := MemAlloc(_SizeOf(SysOne))
     strucVar.iAlpha := 100
     .
     . <Statements that access structure members>
     .
     MemFree(strucVar)
```

Strong Typing Structure Members

From these examples, you may have noticed that structures involve two levels of data typing: one within the STRUCTURE definition and the other when the structure variable is declared. The latter has been the focus of this discussion so far, but the former requires some explanation as well.

You use the STRUCTURE statement to mark the beginning of the definition of a structure entity, followed by one or more MEMBER statements that define what the structure looks like. You must adhere to the following rules when defining structure members:

- Include an AS or IS data type for each MEMBER—strong typing is required.
- Do not use data types that require garbage collection (such as array, float, object, string, and usual).

Variable Structure Alignment

The STRUCTURE statement has been extended with an optional ALIGNMENT clause. In CA-Visual Objects 1.0, all structure components were aligned to a byte boundary. The components of a structure completely filled up the memory block allocated for the structure. There was no internal memory fragmentation. Today's modern processors demand specific data alignment in order to guarantee minimal access times. Aligning structure components on a byte boundary does not inherently fulfill this demand anymore. You might unfortunately fragment memory internally by not employing a byte boundary alignment. Trading off space for speed has become quite reasonable.

The syntax of the STRUCTURE statement has been modified to:

```
STRUCTURE <Structure_Name>    [ ALIGN   <Alignment>        ]
```

Member_Declaration

Possible alignments (<Alignment>) are 1, 2, 4, and 8. When the optional ALIGNMENT clause is not specified, a default alignment of 4 is assumed.

Applying the _sizeof() operator to the structure (<Structure_Name>), yields the total amount of memory (in bytes) occupied by the structure.

This example illustrates the STRUCTURE statement:

```
STRUCTURE Person    ALIGN 8
     MEMBER PersId  AS INT
     MEMBER Sex     AS BYTE
     MEMBER Age     AS INT
```

The memory layout of this structure is:

```
pid       pid       pid               pid               FREE      FREE      FREE      FREE
sex       FREE FREE FREE      FREE    FREE      FREE    FREE
age       age       age               age               FREE      FREE      FREE      FREE
```

The expression _sizeof(Person) results in 24.

Changing the alignment in the STRUCTURE declaration of Person to one (1), results in a packed structure and the _sizeof(Person) then yields 9.

The prime reason for introducing structure alignment is to exploit the speed advantage of modern processors. However, interfacing of sub-systems aligning data to a non-byte boundary to Visual Objects would be impossible without this feature.

Unions

UNIONs are like STRUCTUREs, except that all members start at offset zero (0). In other words, assigning a value to a union member affects all other union members. You use the UNION statement to mark the beginning of the definition of a union entity, followed by one or more MEMBER statements that define what the union looks like. You must adhere to the following rules when defining union members:

- Include an AS or IS data type for each MEMBER -- strong typing is required.
- Do not use data types that require garbage collection (such as array, float, object, string and usual).

Visual Objects' polymorphic values (USUALs) are a good example of the use of unions. USUALs either contain a string, an object, a float, a long, a date, a logical, a codeblock or NIL. Since a USUAL only contains one value at a time, it would not be very efficient to have a structure with a member for each possible type. Instead USUALs can best be represented by a structure containing a union as follows:

```
UNION     Value
     MEMBER s                AS STRING
     MEMBER f        AS FLOAT
     MEMBER l                AS LONG
     MEMBER d                AS DATE
     MEMBER lo       AS LOGICAL
     MEMBER c                AS CODEBLOCK
STRUCTURE Usual
     MEMBER dwTag            AS DWORD
     MEMBER Val      AS Value
```

The USUAL Data Type

USUAL is different from other type declarations because it is not a data type per se, but an explicit declaration for a polymorphic variable. There are two main reasons to use the usual type declaration:

- An AS USUAL declaration will prevent the compiler from attempting to infer the data type of a variable, regardless of the Allow Type Inference compiler option.
- Within a function definition, AS USUAL lets you use calling conventions that require strongly typed arguments and return values without actually specifying a data type. Calling conventions and data typing within function declarations are discussed in Chapter 27, "Functions and Procedures" in this guide.

Usual values can be of any data type described in Chapter 21, "Data Types," (except, of course, VOID). STRUCTURE names and dimensioned arrays cannot be stored as usuals.

Constants

Constants are provided to let you assign logical names to constant values that you use repeatedly. In many ways, constants are like lexically scoped variables. For example, you declare them, initialize them, and optionally assign data types to them. There are, however, some distinct differences, the major one being that you cannot change the value of a constant once you have initialized it.

Using declared constants instead of literal values makes your code more readable and easier to maintain. For example, if the constant value changes, you are faced with changing a single line of code, the declaration statement, as opposed to searching for and changing each line of code where you use the value. Using a constant instead of a variable whose value you never change offers a slight size and performance improvement in your application.

Declaration and Initialization

You declare constants using the DEFINE statement:

```
DEFINE cMyName := "Lou"
```

Like globals, constants are compiler entities. Thus, you cannot include a DEFINE declaration within another entity, such as a function definition. Constant values that you assign as part of the DEFINE statement must evaluate to constants at compile time (for example, literals and simple expressions involving only operators, literals, and other DEFINE constants.)

Lifetime and Visibility

Constants have application-wide lifetime and visibility.

- You can access them anywhere throughout the application. In other words, constants are automatically inherited by all routines in the application without having to pass them as arguments or post them as return values.
- You can hide them from a routine by explicitly declaring a variable (for example, using LOCAL or MEMVAR) with the same name.
- You cannot explicitly release them from memory—their lifetime is guaranteed throughout the application.

You can use the STATIC keyword as a visibility modifier within a DEFINE declaration. Doing this restricts access of the constant to the module in which it is declared. Thus, like a regular constant, a static constant has an application lifetime, but its visibility is limited to the module in which it is declared.

The scoping rules for constants and static constants are different because STATIC changes the visibility of a constant:

Constant Type	Lifetime	Visibility
DEFINE	Application	Application
STATIC DEFINE	Application	Module

Strong Typing

As with LOCAL and GLOBAL variables, you can assign a data type to a constant when you declare it:

```
DEFINE cName := "Lou" AS STRING
```

The data types available for constant declarations are limited to the following:

- DATE
- LOGIC
- SYMBOL
- STRING
- SHORTINT
- INT
- LONGINT
- FLOAT
- BYTE
- WORD
- DWORD

Specifying a data type for a constant has no effect on the compiler; it is merely for the sake of documentation.

A Summary Table

Because there are so many different types of variables and constants, determining the scope of a particular one may seem confusing.

This table summarizes the scoping rules for all of the variable and constant types presented in this chapter:

Variable/Constant Type	Lifetime	Visibility
FIELD	Persistent—while database exists	Application—while database is open
PRIVATE	Until creator returns or until released	Creator and called routines
PARAMETERS	Until creator returns or until released	Creator and called routines
PUBLIC	Application or until released	Application
Code block parameters*	Creator	Creator
FUNCTION parameters*	Creator	Creator
METHOD parameters*	Creator	Creator
PROCEDURE parameters*	Creator	Creator
LOCAL	Creator	Creator
STATIC LOCAL	Application	Creator
GLOBAL	Application	Application
STATIC GLOBAL	Application	Module
DEFINE	Application	Application
STATIC DEFINE	Application	Module

* These refer to formal parameters declared within parentheses as part of a declaration statement or within the vertical bars as part of a code block definition. In effect, these parameters are LOCAL to the declaring entity.

Chapter

23 Operators and Expressions

As you were shown in Chapter 21, "Data Types," all data items are identified by type, and each data type has specific rules for forming its literal values. Chapter 22, "Variables, Constants, and Declarations," showed you how to initialize and create variables and constants and how to determine their data types. These chapters introduced you to the most basic data items in the Visual Objects language, literals, variables, and constants.

Another basic data item is the *function call*. The Visual Objects language is rich in the variety of functions that it provides, and each one is documented in the online help system. In addition to these library functions, you can call functions that you define (more on this in Chapter 27, "Functions and Procedures") and functions defined in third-party libraries. Regardless of how and where they are defined, the data type of a function call is always determined by its return value.

This chapter defines all of the operators that are available to you and shows you how to use them with the basic data items to build expressions. One of the main reasons for understanding data types is so that you will understand how to build expressions. The operators that you will learn about in this chapter are strictly limited in the data types they will accept.

Terminology

An *expression* is, in its simplest form, a literal value, a variable or constant name, or a function call. You can also form more complicated expressions by stringing together a finite number of these basic items using operators.

All expressions represent values and, thus, have an associated data type. You will use expressions within other program statements such as functions, declarations, and commands, but (with the exception of assignments and some of the other special operators) an expression cannot appear as a separate line of code in a program. The syntax representations in the online help system will tell you what type of expression is expected.

An *operator* is a special symbol or word reserved by Visual Objects. Like functions, operators perform a specific operation and return a value of a particular data type.

All operators in Visual Objects require either one or two arguments, called *operands*. Operators requiring a single operand are called *unary operators*, and those requiring two operands are called *binary operators*.

Most binary operators use *infix* notation, which means that the operator is placed between its operands. The multiplication operator (*) is an example of a binary operator which demonstrates infix notation:

```
12 * 12         // 144
```

Most unary operators use *prefix* notation in which you place the operator before the operand, or *postfix* notation in which you place the operator after the operand.

An example of a unary prefix operator is the negate (!) operator:

```
? !TRUE         // FALSE
```

The postincrement operator (++) is an example of a unary postfix operator:

```
LOCAL iCount := 1 AS INT
iCount++        // iCount is now 2
```

Some operators use a syntax that looks like a function call (and are, for this reason, sometimes referred to as *pseudofunctions*), with the operands enclosed in parentheses following the operator name. An example of a binary operator that uses function-calling syntax is the _And() operator:

```
LOCAL iOne := 1, iTwo := 2, iResult AS INT
iResult := _And(iOne, iTwo)
```

An example of a unary operator that uses function-calling syntax is the Float() operator:

```
LOCAL fValue := 1.5 AS FLOAT, iType := 5 AS INT
fValue := Float(iType)
```

All operators have strict rules regarding their usage, including the data types for which they are valid (for example, an operator may be valid for numbers but not for dates). All of the rules that apply to operator usage are described in this section. Using any operator incorrectly results in a compiler error (if you use strong typing) or a runtime error (if you use polymorphic variables).

String Operators

The string operators are used to form expressions of the string data type (they return a string value). The + and - operators are binary, requiring two string (or memo) type operands. _Chr() is a unary operator that uses the function-calling syntax and expects a numeric operand.

Symbol	Operation
+	Concatenate
-	Concatenate without intervening spaces
_Chr()	Convert a numeric code to a string value

Concatenation

Concatenate means to form a new string by joining two strings together. The - operator moves the trailing spaces of the first string to the end of the resulting string, so that there are no intervening spaces between the two original strings. The + operator leaves spaces intact.

This example function displays the results of several string expressions:

```
DEFINE cFirst := "Mary " AS STRING
GLOBAL cSecond := "Jo" AS STRING

FUNCTION PrintStrings()
     LOCAL cNew AS STRING
     ? "Mary " + "Alice"                                  // Mary Alice
     ? cFirst - "Anne"                                    // MaryAnne
     ? cFirst + cSecond                                   // Mary Jo
     ? cFirst + (cNew := "Beth")          // Mary Beth
     ? Trim(cFirst) + cSecond             // MaryJo
```

- Concatenate two literals:

  ```
  "Mary " + "Alice"
  ```

- Concatenate a constant and a literal:

  ```
  cFirst - "Anne"
  ```

- Concatenate a constant and a variable:

  ```
  cFirst + cSecond
  ```

- Concatenate a constant and a string expression:

  ```
  cFirst + (cNew := "Beth")
  ```

- Concatenate a string function return value and a variable:

  ```
  Trim(cFirst) + cSecond
  ```

_Chr()

_Chr() is identical (syntactically and functionally) to the Chr() function. The only difference is that since it is resolved at compile time, it is more efficient and can be used in a DEFINE, GLOBAL, or STATIC LOCAL statement to specify an initial value. See the Chr() entry in the online help system for more details.

Date Operators

The date operators are used to form expressions of the date data type. Except where indicated, these operators are binary, requiring one date and one numeric operand.

Symbol	Operation
++	Unary increment (prefix or postfix)
--	Unary decrement (prefix or postfix)
+	Add a number of days to a date
-	Subtract a number of days from a date

Note: You can also subtract one date from another using the subtraction operator (-). The result of this type of operation is a numeric value that represents the number of days between the two dates. Thus, when the - operator is used to subtract one date from another, it is a numeric operator. See the Numeric Operators section for more information.

You have already seen + and - described as string operators in the previous section, but here they are called date operators. You will see these operators yet again in the next section on Numeric Operators. These operators are *overloaded*, which means that their function changes depending on the data type of the operands. When you use + or - with a date and a numeric value, it is a date operator and, as such, returns a date value.

The + and – operators are not commutative, meaning that the order of the operands is significant in determining the result of the operation. For example, *dValue* + *iValue* is not the same as *iValue* + *dValue*. *dValue* + *iValue* is a date, whereas *iValue* + *dValue* is a numeric.

This example function displays the results of several date expressions:

```
GLOBAL dToday AS DATE

FUNCTION PrintDates()
     LOCAL dToday AS DATE
     LOCAL iValue := 14 AS INT
     dToday := Today()
     ? 93.04.01 + 5                    // 04/06/1993
     ? dToday + 10                     // Ten days from now
     ? dToday + iValue                 // Two weeks from now
     ? dToday - iValue                 // Two weeks ago
     ? CToD("04/01/93") + 7    // 04/08/1993
```

- Add a literal date to a literal number:

  ```
  93.04.01 + 5
  ```

- Add a date variable to a literal number:

  ```
  dToday + 10
  ```

- Add a numeric variable to a date variable:

  ```
  dToday + iValue
  ```

- Subtract a numeric variable from a date variable:

  ```
  dToday - iValue
  ```

- Add a date function return value to a numeric literal:

```
CToD("04/01/93") + 7
```

The increment operator (++) is a special case of the + operator that increases a date variable by one. Similarly, the decrement operator subtracts one from a date variable. These operators are also overloaded because you can use them with numeric and date variables. See the Increment and Decrement Operators section below for more information and examples.

Numeric Operators

The numeric operators are used to form expressions of the numeric data type. As a general rule, these operators require numeric type operands (see Date Operators above for the only exception). By definition, all of the numeric operators return numeric values. Except where noted in the table, the numeric operators are binary.

Symbol	Operation
++	Unary increment (prefix or postfix)
--	Unary decrement (prefix or postfix)
*	Multiplication
/	Division
Symbol	**Operation**
%	Modulus (integer remainder of division)
^ or **	Exponentiation
+	Addition or unary positive (prefix) (operands can be two numeric values, two date values, or one of each)
-	Subtraction or unary negative (prefix) (operands can be two numeric values, two date values, or one of each)
>>	Bitwise shift right
<<	Bitwise shift left
_And()	Bitwise and
_Or()	Bitwise or

Note: You can also subtract a numeric value from a date using the subtraction operator (-). The result of this type of operation is a date. Thus, when the - operator is used to subtract a number from a date, it is a date operator. See the Date Operators section for more information.

As you have already learned, Visual Objects supports several declared numeric data types (such as INT and FLOAT) as well as an undeclared numeric data type used for polymorphic variables. With the exception of the bitwise operators discussed later in this section, the numeric operators apply to all numeric data types, and mixing the various numeric types within an expression is permitted.

This example function displays the results of several numeric expressions:

```
DEFINE iTen := 10 AS INT

FUNCTION PrintNums()
    LOCAL fValue := 14.5 AS FLOAT
    ? 2 ^ 3                                 // 8
    ? 15 + 5                                // 20
    ? 20 - fValue                   // 5.5
    ? fValue * iTen                 // 145.0
    ? iTen ^ 2                      // 100
    ? SqRt(100) + 2                 // 12.00
    ? 101 / iTen                            // 10.10
    ? 101 % iTen                            // 1
```

- Perform exponentiation using two literal numbers:

  ```
  2 ^ 3
  ```

- Add two literal numbers:

  ```
  15 + 5
  ```

- Subtract a numeric variable from a literal number:

  ```
  20 - fValue
  ```

- Multiply a numeric variable by a numeric constant:

  ```
  fValue * iTen
  ```

- Raise a numeric constant to a power using a literal number:

  ```
  iTen ^ 2
  ```

- Add a numeric function return value to a numeric literal:

  ```
  SqRt(100) + 2
  ```

- Divide a numeric literal by a numeric constant:

  ```
  101 / iTen
  ```

- Calculate modulus of a numeric literal using a numeric constant:

  ```
  101 % iTen
  ```

Increment and Decrement Operators

Increment and decrement are unary operators that you can use with either a numeric or a date operand. Unlike other operators which can operate on more complicated expressions, the operand must be a variable name. (Field variables must be qualified as described in Chapter 22, "Variables, Constants, and Declarations.") The resulting data type is the same as that of the operand.

The ++ operator *increments*, or increases the value of, its operand by one, and the -- operator *decrements*, or decreases the value of, its operand by one. Thus, both operators perform an operation on, as well as an assignment to, the operand. In terms of addition, subtraction, and assignment operators, they might be defined as:

- ++*x* is equivalent to *x* := *x* + *1*
- --*x* is equivalent to *x* := *x* - *1*

Note: Since it is used as part of a calculation, the operand must contain a value prior to using either of these operators.

You can specify both operators as prefix or postfix: the prefix form changes the value of the operand before the rest of the expression is evaluated, whereas the postfix form changes the value afterwards.

Tip: The prefix forms of these operators generate more efficient code than their equivalent counterparts (that is, using the plus or minus operator with an assignment operator). The postfix forms, on the other hand, do not and may, in fact, make the program less efficient.

This code illustrates the prefix increment operator in an assignment statement. Since the increment occurs before the assignment takes place, both variables have the same value:

```
LOCAL iValue := 1, iNewValue AS INT
iNewValue := ++iValue
? iNewValue                          // Result:   2
? iValue                             // Result:   2
```

The next example demonstrates the postfix decrement operator. Because the assignment takes place before the original variable is decremented, the two values are not the same:

```
LOCAL dValue := 93.12.31, dNewValue AS DATE
dNewValue := dValue--
? dNewValue                          // Result: 12/31/1993
? dValue                             // Result: 12/30/1993
```

Note: Since ++ and -- perform assignments, these operators can be used in their simplest form as program statements. For example, *iValue*++ is considered a valid program statement, but *50* + *iValue*++ is not.

Bitwise Operators

The >> (right shift), << (left shift), _And(), and _Or() operators are *bitwise* operators, meaning they manipulate integer values as bit strings. These operators have special rules regarding their usage and may not be as familiar to you as some of the other numeric operators mentioned so far in this section.

Shift

The >> and << operators shift a numeric value right or left a specified number of bits. The first operand, the number to be shifted, must be one of these numeric data types:

- INT
- SHORTINT
- LONGINT
- BYTE
- WORD
- DWORD

The second operand, the number of bits to shift, must be a numeric literal or constant and must be a SHORTINT or WORD data type.

This example illustrates these two operators. Note that non-zero bits are shifted off the end of the result—they do not wrap around:

```
LOCAL bNum := 255 AS BYTE, bResult AS BYTE
                                            // Bit representation
                                            // 255 is 1 1 1 1 1 1 1 1
bResult := bNum >> 2                  //  63 is 0 0 1 1 1 1 1 1
bResult := bNum << 1                  // 254 is 1 1 1 1 1 1 1 0
```

Note: If you checked Overflow Checking in the Runtime group box on the Compiler Defaults tab page in the System Settings dialog box, you will receive a runtime Overflow error on the second bResult assignment.

Logical

The _And(), _Or(), and _XOr() operators perform a logical operation between two numeric values using their bit string representations.

The following truth tables define the _And(), _Or(), and _XOr() operators by showing their results for all possible combinations of bits 1 and 0. The results in the body of each table are obtained by using the values in the left-hand column and the top row as operands.

_And()	1	0
1	1	0
0	0	0

_Or()	1	0
1	1	1
0	1	0

_XOr()	1	0
1	0	1
0	1	0

For _And(), _Or(), and _XOr() both operands must be one of these numeric data types:

- INT
- SHORTINT
- LONGINT
- WORD
- DWORD

These operators use function-calling syntax as illustrated in the following examples:

```
LOCAL iX := 129, iY := 65 AS INT
                                        // Bit representation
                                        // 129 is 1 0 0 0 0 0 0 1
                                        //  65 is 0 1 0 0 0 0 0 1
? _And(iX, iY)                  //   1 is 0 0 0 0 0 0 0 1
? _Or(iX, iY)                   // 193 is 1 1 0 0 0 0 0 1
? _XOr(iX,iY)                   // 192 is 1 1 0 0 0 0 0 0
```

Variable Parameter Lists

In CA-Visual Objects 1.0 the _AND, _OR, and _XOR operators were limited to a maximum of two operands. The programmer had to cascade these operators, in cases where more than two operands were required. This sort of cascading is no longer necessary with Visual Objects.

The following expression pairs can be considered equivalent:

```
_AND(A, (_AND(B, (_AND(C, D)))))          _AND(A, B, C, D)
_OR(A, (_OR(B, (_OR(C, D)))))             _OR(A, B, C, D)
_XOR(A, (_XOR(B, (_XOR(C, D)))))          _XOR(A, B, C, D)
```

Logic Operators

Like all other operators, the logic operators are grouped together because each returns a logic value. The logic operators, however, are broken into two distinct groups based on their operand data types. *Boolean operators* act on logic operands only, performing a strictly defined algebraic function. *Relational operators* compare two values of the same data type and return a logic value indicating the result of the comparison.

Boolean Operators

The Boolean operators are used to form logic expressions. All of these operators require logic operands. Except where indicated in the table, the Boolean operators are binary.

Symbol	Operation
.NOT. or !	Unary negate (prefix)
.AND.	And
.OR.	Or

The quick way to define these operators is to tell when they return TRUE: .AND. returns TRUE if both operands are TRUE; .OR. returns TRUE if either operand is TRUE; and .NOT. returns TRUE if its operand is FALSE.

This example function generates a complete truth table for each of the Boolean operators:

```
FUNCTION PrintTruth()
     ? TRUE .AND. TRUE, TRUE .AND. FALSE
     ? FALSE .AND. TRUE, FALSE .AND. FALSE

     ? TRUE .OR. TRUE, TRUE .OR. FALSE
     ? FALSE .OR. TRUE, FALSE .OR. FALSE

     ? .NOT. TRUE, !FALSE
```

The result of this program, although not as nicely formatted, is:

.AND.	TRUE	FALSE
TRUE	TRUE	FALSE
FALSE	FALSE	FALSE
.OR.	**TRUE**	**FALSE**
TRUE	TRUE	TRUE
FALSE	TRUE	FALSE
.NOT.	**TRUE**	**FALSE**
	FALSE	TRUE

Note: Visual Objects uses a shortcut when evaluating the .AND and .OR. operators. For .AND., the second operand is not evaluated if the first operand evaluates to FALSE. For .OR., the second operand is not evaluated if the first operand evaluates to TRUE.

Relational Operators

The relational operators (also called comparison operators) are used to form logic expressions. All of them are binary operators requiring two operands of the same data type (or one NIL operand in the cases where NIL is an allowed data type).

Symbol	Operation	Operand Data Types
<	Less than	String, date, numeric, logic
<=	Less than or equal	String, date, numeric, logic
>	Greater than	String, date, numeric, logic
>=	Greater than or equal	String, date, numeric, logic
=	Equal	All data types
==	Exactly equal, including trailing spaces	String
==	Same as = operator	All data types other than string
<>, #, or !=	Not exactly equal, the opposite of the == operator	Strongly typed strings (i.e., declared AS STRING or AS PSZ)
<>, #, or !=	Not equal, the opposite of the = operator	All data types other than strongly typed strings (see above)
$	Substring	String

Note: Many of the numeric and logic operators can be converted to methods if used with an object as the first operand. See Chapter 25, "Objects, Classes, and Methods," in this guide for more information on this subject.

The comparison rules for these operators depend on the data type in question:

- ***String:*** Comparison is based on the underlying ANSI code (for example, the code for "A" is 65 and the code for "Z" is 90, making "A" < "Z"). Strings are compared according to the following rules, assuming the expression to test is *cLeft* = *cRight*:

  1. If *cRight* is a NULL_STRING, quit comparing and return TRUE.
  2. If Len(*cRight*) > Len(*cLeft*), quit comparing and return FALSE.
  3. Compare all characters in *cRight* with *cLeft* until *cRight* runs out of characters or until there is a difference. If all characters are the same, return TRUE; if there is a difference, return FALSE.

- ***Symbol:*** Comparison is performed using a numeric value that is uniquely associated with the symbol when it is created. Comparison of symbols is much faster than strings.

  Note: Literal symbols are converted to uppercase letters, making a comparison such as:

  ```
  ? #cat = #CAT
  ```

  result in TRUE. To preserve the casing of letters in symbols, store them using SysAddAtom() instead of literals:

  ```
  symLittleCat := SysAddAtom("cat")
  symLargeCat := SysAddAtom("CAT")
  ? symLittleCat = symLargeCat        // FALSE
  ```

- ***Date:*** Dates are compared chronologically.
- ***Numeric:*** Numbers are compared based on magnitude.
- ***Logic:*** TRUE is greater than FALSE.
- ***NIL:*** NIL is not equal to anything except a NIL, NULL_ARRAY, or NULL_OBJECT value.
- ***Array:*** Comparison using the = operator determines if two arrays are actually references to the same array. If they are, this operator returns TRUE; otherwise, it returns FALSE.
- ***Object:*** Comparison using the = operator determines if two objects are actually references to the same object. If they are, this operator returns TRUE; otherwise, it returns FALSE.

This example function displays the results of several logic expressions involving relational operators:

```
GLOBAL cName := "Lou" AS STRING

FUNCTION PrintYesNo()
     ? 15 < 5                                  // FALSE
     ? 15 = 5                                  // FALSE
     ? 15 > 5                                  // TRUE
     ? SqRt(25) + 5 = 10               // TRUE
     ? Today() > 59.12.20              // TRUE
     ? cName = "Frank"                 // FALSE
     ? "Lo" $ cName                    // TRUE
```

- Compare two literal numbers:

  ```
  15 < 5
  15 = 5
  15 > 5
  ```

- Compare a numeric function return value to a literal number:

  ```
  SqRt(25) + 5 = 10
  ```

- Compare a date function return value to a literal date:

  ```
  Today() > 59.12.20
  ```

- Compare a string variable to a literal string:

  ```
  cName = "Frank"
  ```

- Determine if a string literal is a substring of a string variable:

  ```
  "Lo" $ cName
  ```

Exactly Equal

Most of the relational operators are fairly straightforward, and you are probably already familiar with them. You may not, however, be familiar with the exactly equal operator (==).

You can use this operator to compare strings for exact equality in length and content, including trailing spaces:

```
? "String One    " == "String One"       // FALSE
? "String One" == "String One    "       // FALSE
? "String One" == "String One"           // TRUE
```

For all other data types, this operator is equivalent to the = operator.

Assignment Operators

The assignment operators are summarized separately to avoid repeating them under each applicable data type. These operators assign a value to a variable and return the assigned value as a result, letting you include assignments in expressions.

Symbol	Operation	Operand Data Types
:=	Assign	String, symbol, date, numeric, logic, NIL, array, code block, object
+=	Add (or concatenate) and assign	String, numeric, mix date and numeric
-=	Subtract (or concatenate) and assign	String, date, numeric, mix date and numeric
^=	Exponentiate and assign	Numeric
*=	Multiply and assign	Numeric
/=	Divide and assign	Numeric
%=	Modulus and assign	Numeric

The assignment operators are unique among the operators in that they require a variable name (including a class instance variable) as the first operand—the second operand can be any data type that is appropriate for the operator in question. The data type of an assignment operation is determined by the value assigned (the second operand).

If you declare the variable you are assigning using strong typing, the data type of the second operand must fit the declared data type or your program will produce errors when compiled. For example, the following assignment is illegal because it attempts to put a string into an integer variable:

```
LOCAL iValue AS INT
...
iValue := "String Value Not Allowed!"
```

Important! *You can perform simple assignments using the equal sign (=) instead of one of the assignment operators, but this is not recommended. Making assignments in this manner is limiting in that the assignment statement must stand alone and cannot be part of an expression. Unless the Old Style Assignments compiler option is checked, the use of = as an assignment operator will not be allowed.*

Note: If you want to assign a value to a field using any of the assignment operators, you must qualify the field name as described in Chapter 22, "Variables, Constants, and Declarations."

Assignments as Program Statements

You can use assignment operators to form program statements just as you would use a command, function, or method invocation. For example, the following lines of code may appear in a program:

```
iValue := 25
iNewValue := SqRt(iValue) ** 5
iValue += iNewValue
```

You can also perform multiple (or *chained*) assignments within the same program statement. When you assign values in this manner, the assignments are executed from right to left. This feature is particularly useful when you need to store the same value to many different fields, possibly in different database files:

```
oDBCust:CustID := oDBTrans:TransNo := cIDNumber
```

Assignments as Expressions

As stated earlier, each assignment operator returns a value whose data type depends on the second operand. Therefore, you can use assignments alone or as part of other expressions anywhere in the language where an expression is allowed:

```
IF (dDate := (Today() - 1000)) = CToD("12/20/79")
? SqRt(iValue := (iValue ** 2))
? cString += " Add this to the end"
```

Compound Assignments

The compound assignment operators perform an operation between the two operands and assign the result to the first operand.

Operator	Example	Definition
+=	a += b	a := a + b
-=	a -= b	a := a - b
*=	a *= b	a := a * b
/=	a /= b	a := a / b
%=	a %= b	a := a % b
^=	a ^= b	a := a ^ b

Unlike the assignment operator (:=) which you can use to initialize a variable, the compound operators require the first operand to have an initial value because it is used as part of a calculation.

This example code illustrates the += operator using string, date, and numeric data types:

```
// String (concatenation)
LOCAL cString := "Hello" AS STRING
cString += " there"
? cString                               // "Hello there"

// Date (addition)
LOCAL dDate := 90.12.12 AS DATE
dDate += 12
? dDate                                         // 12/24/90

// Numeric (addition)
LOCAL iValue := 10 AS INT
? SqRt(iValue += 15)                    // 5
? iValue                                        // 25
```

Mixing Data Types

As you have seen from reviewing all of the operators in this chapter, operations between different data types are almost never permitted. There are a few exceptions, however, that have already been pointed out, such as the case involving calculations between dates and numbers, and the case in which various data types can be compared to NIL.

Automatic Type Conversion

There are instances in which Visual Objects automatically performs a type conversion. In general, there are many cases in the language where you can use a data item other than the type called for, and the compiler will generate code to perform the necessary conversion for you. Internally, this is accomplished using conversion operators of the form:

<idType>(<Value>)

<idType> can be any of these data types:

ARRAY	INT	PTR	SYMBOL
CODEBLOCK	LOGIC	REAL4	WORD
DATE	LONGINT	REAL8	
DWORD	OBJECT	SHORTINT	
FLOAT	PSZ	STRING	

For example, there are no restrictions to prevent you from creating complex numeric expressions involving many numeric data types. The compiler supplies the appropriate conversion operators to change the data items within the expression to the data type with the largest storage capacity:

```
LOCAL iX := 100 AS INT, fY := 25.5 AS FLOAT
? iX * fY
```

Since *iX* and *fY* have different data types, and since FLOATs have a greater storage capacity than INTs, the compiler would generate something like this:

```
? Float(iX) * fY
```

You can also use any numeric expression, regardless of its actual type, as an argument whenever a strong numeric type is required by the function syntax. Again, the compiler supplies the necessary conversion operator to obtain the required type:

```
LOCAL fY := 25.5 AS FLOAT
? I2Bin(fY)
```

I2Bin() requires a SHORTINT as its argument. Thus, the compiler generates something like:

```
? I2Bin(ShortInt(fY))
```

The conversion operators are available as part of the language and, as such, you can use them to perform type conversions. You will find, however, that in most cases their use is unnecessary because of the automatic type conversion described above.

One case in which you will find the conversion operators essential is to *prevent* automatic type conversion when it is not what you want. Using these operators, you can gain explicit control over how numeric expressions are evaluated. For example, to prevent the automatic conversion of *iX* to a FLOAT value:

```
LOCAL iX := 100 AS INT, fY := 25.5 AS FLOAT
? iX * Int(fY)
```

Important! *Many conversions are illegal using these operators. For example, you cannot convert a number to a string as in String(100). Any conversion you attempt that is not possible will result in a compiler error. To perform conversions of this nature, use the functions discussed in the next section, Manual Type Conversion.*

Note: The conversion operators are intended mainly for use by the compiler and are provided to remain in keeping with the open architecture philosophy of this product. There are, however, certain cases in which the advanced user may need them to convert a pointer reference to a particular data type.

Manual Type Conversion

There are instances in which an automatic type conversion is not possible. For example, if a function requires a string argument, you cannot use a number and expect it to work. Doing so will always result in an error, whether it be a compiler or a runtime error.

The compiler cannot handle this type of conversion nor is it permitted using the conversion operators. In these cases, you will use functions that are specifically designed to convert a value from one type to another at runtime.

Suppose that you want to form a string expression that displays a date value along with some text on a report. This expression is not valid because the + operator is not defined for adding strings and dates:

```
"Today's date is " + Today()
```

To get around this problem, you could convert the date value to a string using the DToC() (Date To Character) function:

```
"Today's date is " + DToC(Today())
```

In this expression, the functions Today() and DToC() are evaluated first, converting today's date to a string value. The resulting string is then concatenated to the string literal, "Today's date is ".

Similarly, you can convert numbers to strings using the Str() function:

```
"The total amount is " + Str(fTotal, 9, 2)
```

Another common conversion that you may need to perform is string to numeric, in which case you would use the Val() function:

```
nNextSSN := Val(cSSN) + 1
```

The examples given here illustrate only a few of the more commonly used conversion functions. There are many more, all of which are detailed in the online help system.

Converting Typed Pointers

With the introduction of typed pointers, it now becomes necessary to be able to do an explicit conversion between the different pointer types. Automatic pointer conversions are carried out by the compiler (various compiler warnings inform the user of these automatic conversions). However, as a general rule, it is not recommended to rely on these automatic conversions. The applied conversion rules will occasionally not comply with the programmer's intention, resulting in programs, which are semantically incorrect. Such errors are usually of very subtle nature but become very problematic when debugging the program. Implicit pointer conversion should be avoided whenever possible.

Visual Objects provides the following syntax for pointer conversions. This syntax is similar to the explicit type casting syntax of the language — <idType>(_cast, <Variable>):

```
PTR(<idType>, <Pointer Variable>)
```

The *idType* can be one of the following data types:

- DATE
- LOGIC
- INT
- SHORTINT
- LONGINT
- BYTE
- WORD
- DWORD
- REAL4
- REAL8

The "*Pointer Variable*" can be of any pointer type (typed pointer, PTR, PSZ, AS Structure).

The anonymous pointer (pointer of type PTR) supplied by MemAlloc() is explicitly converted to a typed pointer (of type REAL4). For more information on pointers, refer to the Pointers section in Chapter 21, "Data Types" in this guide.

Type Casting

There are also several operators that let you perform type casting. These operators resemble the type conversion operators discussed earlier, but are designed for use at a much lower level of programming. Their general format is:

```
<idType>(_CAST, <Value>)
```

where <*idType*> is one of the data types listed in the Automatic Type Conversion section above.

Unlike the type conversion operators, the type casting operators change only the data type of an expression, not its value:

```
Long(_CAST, Today())
```

means take the date value Today() and use it as a long integer.

The compilation of these operators will always succeed, even though what you are trying to do may have unpredictable results, as in this example which attempts to use the number 5 as an object pointer:

```
Object(_CAST, 5)                // Nonsense example!
```

Caution! *Type casting is a very low-level and dangerous feature designed for writing and porting Assembler and C-level routines.*

Special Operators

There are several symbols that have special meaning in the Visual Objects language. These are special operators that often appear in expressions.

Operator	Valid Data Types
()	Argument list or grouping within an expression
{}	Literal array, code block, or class instance
[]	Array element
:	Send, used to access instance variables and to send messages to an object
.	Structure member access, macro terminator
->	Alias identifier
&	Compile and execute (macro)
@	Reference

Parentheses

You can use parentheses in expressions to force a particular evaluation order (more on this subject in the Expression Evaluation section) or to make a complicated expression more readable. When using parentheses for grouping within an expression, the item that falls within the parentheses must be a valid expression.

For example, you might group this expression:

```
x + y * 15
```

to cause the addition to occur before the multiplication, thus changing the result of the expression:

```
(x + y) * 15
```

You might also group this expression:

```
x < 14 .AND. x > 21
```

to make it more readable without changing the order of evaluation:

```
(x < 14) .AND. (x > 21)
```

You could not, however, group this expression as follows because the entity within the parentheses is not a valid expression:

```
x < (14 .AND. x) > 21
```

You must use parentheses to indicate a function or method call—arguments, if any, are expected to occur between the parentheses. For example:

```
LOCAL cString := "Frank Louis", cNickName AS STRING
cNickName := Substr3(cString, 7, 3)      // "Lou"
```

Curly Braces

Curly braces ({}) identify literal arrays and code blocks. The only difference between the two representations is that the code block literal must also contain a parameter list (even if it is empty) delimited with vertical bars (||):

```
LOCAL aOne AS ARRAY
LOCAL cbDisplay, cbToday AS CODEBLOCK
aOne := {1, 2, 3, 4, 5, 6}               // Literal array
...
cbDisplay := {|x| QOut(x)}               // Code block with arguments
...
cbToday := {|| QOut(Today())}            // Code block with no arguments
```

You will also use curly braces ({}) to create an instance of a class. An example of this use of the curly braces appears under the Message Send heading below.

Subscript

The subscript operator ([]) references a single array element using its numeric subscript: (subscript)"] (subscript)]

```
LOCAL aOne AS ARRAY
aOne := {1, 2, 3, 4, 5, 6}               // Literal array
...
? aOne[1]
```

Separate subscripts for multidimensional arrays using commas or several sets of square brackets. These two array element references are equivalent:

```
LOCAL aMulti AS ARRAY
aMulti := {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}
? aMulti[3, 2]      // Result: 8
? aMulti[3][2]      // Result: 8
```

When using the subscript operator to access array elements, you are limited to eight dimensions. To access array elements beyond eight dimensions, use the ArrayGet() and ArrayPut() functions, nested to the appropriate level.

Message Send

The send operator (:) sends messages to a class instance, or object, and accesses its instance variables. In the following sketchy example, a class is defined with a single instance variable and one method. The function UseClass() creates an instance of the class (using {} operators), and initializes the instance variable and invokes the method using the send operator (:):

```
CLASS Animal                    // Define Animal class
     EXPORT Genus
     .
     . <Other instance declarations>
     .

METHOD Population() CLASS Animal
     .
     . <Statements defining the method>
     .

FUNCTION UseClass()
     LOCAL oJag AS OBJECT                // Declares oJag as an object, any class

     oJag := Animal{}                              // Creates oJag as Animal class
     oJag:Genus := "Felus"
     ? oJag:Population()
     .
     . <Executable statements>
     .
```

You can also have an array as the first operand when using the message send operator to invoke a method. In this case, the array should contain objects as its elements (or other arrays that eventually lead to objects), and the method will be invoked for each element in the array.

Method invocations are like function calls in that they perform an operation and return a value. They can appear as stand alone program statements; however, instance variables, like all other variables, can only appear as part of another statement, such as an assignment.

Dot

The dot operator (.) accesses a member of a structure. Its syntax is:

<*idStructVar*>.<*idMember*>

To use this operator, you must first define a structure and declare a variable to hold it:

```
STRUCTURE SysOne                // Define SysOne data structure
     MEMBER iAlpha AS INT
     MEMBER pszName AS PSZ

FUNCTION UseStruct()
     // Declare variable to hold SysOne structure
     LOCAL strucVar IS SysOne
     strucVar.iAlpha := 100
     .
     . <Statements that access structure members>
     .
```

The dot operator also serves as the macro terminator, which is discussed further in The Macro Operator section later in this chapter.

Alias Identifier

The alias identifier qualifies a variable reference. For example, _FIELD-><*idField*> explicitly specifies a field variable, and _MEMVAR-><*idVariable*> (or M-><*idVariable*>) explicitly specifies a polymorphic variable.

For a field variable, you can use the specific database alias (defined when you open the database file) as in <*idAlias*>-><*idField*>.

You can also use the alias identifier with an expression by enclosing the expression in parentheses, as in <*idAlias*>->(<*Expression*>). If the expression is a single function or procedure call, the parentheses are optional (for example, <*idAlias*>->EOF()). When you use the alias identifier in this manner, the work area associated with the specified alias is selected before the expression is evaluated, and the current work area is selected afterwards.

Macro

The macro symbol (&) is the compile-and-run operator. It is a unary prefix operator whose only valid operand is a character variable. The macro operator is discussed in more detail later in this chapter in The Macro Operator section.

Reference

The reference operator (@) is valid in the argument lists of method, function, and procedure calls—sometimes it is a required part of the syntax, as with FRead(). It is also valid for assigning the address of a value to a pointer.

@ is a unary prefix operator whose operand can be any variable name (field variables are not allowed), array element, method, or function call. It works by returning the address of, or reference to, its operand.

When you use the @ operator to pass a value, the value is passed by reference. This means that the called routine can modify the value directly.

For functions defined using the CLIPPER calling convention and methods, you can use the reference operator (@) for any argument. For functions defined using the STRICT or other calling convention, the arguments defined using the REF keyword require that you use @ when calling the function, whereas the arguments defined using the AS keyword do not allow @.

You can also use @ to obtain the address of a variable or function return value. The address is returned as a pointer (PTR).

Expression Evaluation

When evaluating expressions with two or more operations that are not explicitly grouped together with parentheses, Visual Objects uses an established set of rules to determine the order in which the various operations are evaluated. These rules, called precedence rules, define the hierarchy of all of the operators discussed so far in this chapter and are strictly enforced to insure that expressions are evaluated in a consistent manner.

For the most part, expressions are operations that manipulate a single data type. For example, an expression might concatenate several character strings or perform a series of mathematical operations on several numbers.

There are, however, expressions in which the evaluation of several different operations are necessary. For example, a complex logic expression can involve several related operations, on different data types, that are connected with Boolean operators:

```
cString1 $ cString2 .AND. nVal1++ > nVal2 * 10
```

Precedence Levels

When more than one type of operator appears in an expression, the subexpressions are evaluated in a particular order according to the operators involved. The established order is:

1. Unary signs (+, -)
2. Exponentiation (^, **)
3. Multiplication, division, and modulus (*, /,%)
4. Addition, subtraction, and concatenation (+ and -)
5. Comparison (<, <=, >, >=, =, ==, <>, #, !=,. $)
6. Boolean negate (.NOT., !)
7. Boolean and (.AND.)
8. Boolean or (.OR.)
9. Assignment (:=, *=, /=, %=, +=, -=)

Note: All function calls and operators that are not mentioned in this list are evaluated before any other operators. Expressions within these entities (such as a function argument that requires evaluation) are evaluated according to the precedence defined above.

Except for assignments, all operations at the same level of precedence are performed in order from left to right. Assignments are performed in order from right to left. For the compound operators, the nonassignment portion (such as addition or concatenation) of the operation is performed first, followed immediately by the assignment.

Note: The increment (++) and decrement (--) operators are not included with the precedence levels listed above. Depending on whether you use the prefix or the postfix form, these operations occur just before or after the most immediate operation in which they are involved.

In the example given earlier:

```
cString1 $ cString2 .AND. nVal1++ > nVal2 * 10
```

The order of evaluation of the subexpressions is:

1. nVal2 * 10
2. cString1 $ cString2
3. nVal1 > (result of Step 1)
4. nVal1++
5. (result of Step 2) .AND. (result of Step 3)

The next example shows how the established precedence rules guarantee that like expressions produce like results. Consider these two, algebraically equivalent numeric expressions:

```
5 * 10 + 6 / 2        // Expression a
6 / 2 + 5 * 10        // Expression b
```

Expression *a* is evaluated as:

1. 5 * 10 equals 50
2. 6 / 2 equals 3
3. 50 + 3 equals 53

Expression *b* is evaluated as:

1. 6 / 2 equals 3
2. 5 * 10 equals 50
3. 3 + 50 equals 53

Without precedence rules, these two expressions would be evaluated in order from left to right, and expressions *a* and *b* would produce different results:

a. 5 * 10 equals 50; 50 + 6 equals 56; 56 / 2 equals 28

b. 6 / 2 equals 3; 3 + 5 equals 8; 8 * 10 equals 80

Neither result is correct, and the two results are not the same.

Parentheses

As stated earlier, you can override the order in which an expression is evaluated using parentheses. When you use parentheses in an expression, all subexpressions within parentheses are evaluated first using the precedence rules described in this section. If you nest the parentheses, the evaluation is done starting with the innermost pair and proceeding outward.

Continuing with the first numeric expression used in the example above, you could use parentheses to force the addition operation to be evaluated first:

```
5 * (10 + 6) / 2
```

Using parentheses in this manner changes the result of the expression by evaluating it in this order:

1. 10 + 6 equals 16
2. 5 * 16 equals 80
3. 80 / 2 equals 40

Tip: Although the Visual Objects language provides a specific order of precedence for evaluating expressions, it is better programming practice to explicitly group operations for readability and to be certain that what executes meets your expectations.

The Macro Operator

The macro operator in Visual Objects is a special operator that allows runtime compilation of expressions and text substitution within strings. Whenever the macro operator (&) is encountered, the operand is submitted to a special runtime compiler referred to as the macro compiler that can compile expressions but not statements or commands.

Text Substitution

You can use the macro operator with a string variable (called a *macro variable*) as the operand:

```
&<cMacroVar>
```

When you include a macro variable within a string, the contents of the variable are substituted for the variable reference. This use of the macro operator is known as *text substitution*:

```
cMacro := "there"
? "Hello &cMacro.!"            // Result: Hello there!
```

The period (.) is the macro terminator and it is optional. You need it to indicate the end of the macro variable in order to distinguish it from adjacent text in the statement as illustrated in the above example.

When you use the macro operator for text substitution, the operand must be a polymorphic variable whose data type is string, the macro operator must immediately precede the variable, and the string within which you use the macro variable must be known at compile time. If you fail to follow these guidelines, the macro variable will not be expanded.

Thus, a macro variable will **not** be expanded under these circumstances:

- The variable is a field variable, a declared variable, or an array element.
- The macro variable is enclosed in parentheses.
- The string within which the macro variable appears is not known at compile time (it may, for example, be entered by the user at runtime). (The StrEvaluate() function described in the Related Functions section below offers you this capability.)

In all of these cases, the macro operator is assumed to be literal text:

```
cMacro := "there"
? "Hello &(cMacro)"            // Result: Hello &(cMacro)
```

Compile and Execute

When you use the macro operator in a place where an expression is expected (such as a function or command argument) the macro symbol behaves as the compile and execute operator, resulting in runtime compilation and execution of the expression.

Thus, you can use the macro operator to compile and execute a macro variable containing an expression stored as a string value. When you use the macro operator in this manner, the expression is compiled by the macro compiler and executed. The compiled code is then discarded:

```
cMacro := "DToC(Today())"
? &cMacro                      // Displays today's date
```

If you specify a string expression enclosed in parentheses and prefaced by the macro operator (&), the expression is evaluated and the contents of the resulting string is compiled and executed:

```
cNum1 := "15 "
cNum2 := "+ 30"
? &(cNum1 + cNum2)             // Result: 45
```

When you enclose the macro operand within parentheses, you are creating a *macro expression* which, unlike a macro variable, can contain array elements and fields. This example illustrates using the macro operator with an array element:

```
LOCAL aStruct AS ARRAY
LOCAL iField AS INT
USE customer
aStruct := DBStruct()
DO WHILE .NOT. EOF()
     // Displays contents of each field
     FOR iField := 1 UPTO ALen(aStruct)
          ? &(aStruct[iField, DBS_NAME])
     NEXT
     DBSkip()
ENDDO
DBCloseArea()
```

Using Declared Variables

When you use the macro operator to compile and execute an expression stored in a macro variable, the variable can be a declared variable; however, the expression contained within the macro variable cannot reference declared variables. Similarly, variables occurring within a macro expression can be declared, but the expressions contained in the variables cannot reference declared variables.

Some examples of the correct and incorrect usage of declared variables with the macro operator are:

- The following example correctly uses a declared variable as the macro operator operand:

```
LOCAL cNum := "5 + 10"
? &cNum                              // Result: 15
```

- This example correctly uses declared variables in a macro expression:

```
LOCAL cNum1 := "5 + 10 ", cNum2 := "+ 15"
? &(cNum1 + cNum2)          // Result: 30
```

- The next example incorrectly makes a reference to a declared variable within a macro variable:

```
LOCAL cNum := "5 + 10 + nNum", nNum := 30
? &cNum                              // nNum not known at runtime
```

- The next example incorrectly makes a reference to a declared variable within a macro expression:

```
LOCAL cNum1 := "5 ", cNum2 := "+ 10 + nNum",nNum := 30
? &(cNum1 + cNum2)          // nNum not known at runtime
```

Using Operators

Certain operators, listed below, are not allowed in macro expressions:

- Type casting operators
- Conversion operators
- ++, --
- >>, <<
- _And(), _Or()
- _Chr()
- _TypeOf()
- _SizeOf()
- _MakePtr()

All other operators, including compound assignments and message sends, are fully supported.

Using Functions

As a general rule, you can call a function within a macro expression only if it is defined with the CLIPPER calling convention. In addition, the following functions may be called even though they use the PASCAL calling convention:

AAdd()	DoW()	Lower()	Str()
Asc()	DToS()	Max()	SqRt()
Abs()	DToC()	LTrim()	Substr()
At()	Eval()	Min()	Today()
CDoW()	Exp()	Month()	Trim()
Chr()	Integer()	Replicate()	Upper()
CMonth()	Left()	Right()	Val()
CToD()	Len()	Round()	Year()
Day()	Log()	Space()	

Nesting Macros

You can nest the macro operator as deep as necessary, provided the resulting expression is not too complex. For example, after assigning a macro variable to another macro variable, the original macro variable can be expanded, resulting in the expansion of the second macro variable and evaluation of its contents:

```
cOne = "&cTwo"
cTwo = "cThree"
cThree = "hello"

? &cOne                         // Result: "hello"
```

Related Functions

The Visual Objects language has several functions that give you access to the macro system.

Function	Description
Evaluate()	Compiles a string value containing an expression and executes the result (i.e., MExec(MCompile()))
MAssign()	Performs an assignment to a macro variable or expression
MCompile()	Compiles a string value containing an expression
MCShort()	Controls logic shortcutting in macro expression evaluation
MExec()	Executes a compiled string

StrEvaluate()	Allows text substitution within runtime strings

MCompile() and MExec() break the compile and execute operations performed by the macro operator into distinct steps. These functions let you compile an expression, save the compiled code in a variable, and execute the compiled code repeatedly:

```
PROCEDURE Start()
     LOCAL cPCode AS STRING
     USE exp_file
     DO WHILE .NOT. EOF()
          // Compile ExpField
          cPCode := MCompile(ExpField)
          ...
          MExec(cPCode)
          ...
          MExec(cPCode)
          ...
          MExec(cPCode)
          ...
          DBSkip()
     ENDDO
     DBCloseArea()
```

Since the compile operation is the most time consuming, using MCompile() and MExec() instead of the macro operator (&) could potentially save you a lot of time.

Important! *Strings that you compile with MCompile() are valid during the current run only. For this reason, you cannot store them (in a database or memory file, for example) and attempt to execute them with MExec() in subsequent runs. Doing so will produce unpredictable results.*

Evaluate() is the functional equivalent of the macro (compile and execute) operator (for example, Evaluate(<*cExp*>) is the same as &(<*cExp*>)). There are no advantages to using this function over the macro operator except for the possibility that you might find the function less cryptic when reading code.

MCShort() lets you set a flag that controls logic shortcutting in macro expression evaluation. When TRUE (the default), macro expressions involving .AND. and .OR. operators are performed using the shortcut described in the Logic Operators section earlier in this chapter. When FALSE, these expressions are evaluated in their entirety—no shortcuts are taken.

StrEvaluate() lets you perform text substitution in strings that are entered at runtime:

```
MEMVAR cOne
LOCAL cTwo AS STRING
cOne := "world"
ACCEPT "Enter a macro string " TO cTwo
// If you type "Hello &cOne" (without quotes)
? cTwo                                  // Result: "Hello &cOne"
? StrEvaluate(cTwo)             // Result: "Hello world"
```

It is possible to perform assignments to a variable, array element, or object instance variable by storing the value on the left-hand side of the assignment in one or more variables and using the macro operator. For instance, this example stores the value 5 to a variable named *iValue*:

```
MEMVAR iValue
LOCAL cVar AS STRING
cVar := "iValue"
&cVar := 5
? iValue                                  // Result: 5
```

The next example is similar, but uses a macro expression instead of a simple variable to make an assignment to an array element:

```
MEMVAR aOne
LOCAL cArray, cIndex, cExp AS STRING
cArray := "aOne"
cIndex := "[5]"
aOne := ArrayNew(10)
cExp := cArray + cIndex
&cExp := 100
? aOne[5]                         // Result: 100
```

You can also use the MAssign() function to perform assignments in this manner. The two previous examples are repeated below to illustrate the use of this function:

```
MEMVAR iValue
LOCAL cVar AS STRING
cVar := "iValue"
MAssign(cVar, 5)
? iValue                                  // Result: 5
```

```
MEMVAR aOne
LOCAL cArray, cIndex AS STRING
cArray := "aOne"
cIndex := "[5]"
aOne := ArrayNew(10)
MAssign(cArray + cIndex, 100)
? aOne[5]                         // Result: 100
```

All of these functions are documented in the online help where you can look to find more information.

Macros and Code Blocks

The macro compiler supports runtime compilation of code blocks, allowing you to evaluate code blocks that are stored (as strings) in database fields or entered by the user at runtime.

In this example, the field *BlockField* contains a code block stored as a string. This field is used in a macro expression that returns the code block. The code block is saved in the variable *cbBlock*, which is later evaluated with Eval():

```
PROCEDURE Start()
     LOCAL cbBlock
     USE exp_file
     DO WHILE .NOT. EOF()
          // Compile code block
          cbBlock := &(BlockField)
          ...
          Eval(cbBlock)
          DBSkip()
     ENDDO
     DBCloseArea()
```

Note: Because runtime code blocks are implemented as instances of the system-defined _CODEBLOCK class, their data type is object; therefore, you cannot save them to variables declared as code blocks (for example, LOCAL cbBlock AS CODEBLOCK in the above example will not work.) You can store a runtime code block either as a polymorphic variable (as in the above example) or AS OBJECT (or, more specifically, AS _CODEBLOCK) if you require strong typing.

Tip: You can also use the macro operator within a code block definition. See Chapter 26, "Code Blocks," in this guide for a discussion of macro expansion within code blocks.

When Not to Use the Macro Operator

The following list summarizes situations in which using the macro operator is either not allowed or not recommended:

- You cannot use the macro operator as part of a declaration statement.
- You cannot use a field or an array element with the macro operator for the purpose of text substitution, although these items are allowed in macro expressions.
- You cannot use a declared variable as part of an expression that you intend to compile and execute with the macro operator.
- You cannot access structures in macro expressions.
- You cannot use SELF and SUPER (see Chapter 25, "Objects, Classes, and Methods") in macro expressions.
- Although you cannot use the macro operator to substitute command keywords, you can use it to substitute all or part of a command argument (for example, with the COPY command, you can use a macro variable for the target file name but not to substitute the COPY or TO keywords). In syntax representations, command keywords are denoted using uppercase letters, whereas arguments are denoted using metasymbols (for example, COPY TO <*xcFile*>).

  The practice of using the macro operator to substitute command arguments, however, is not recommended. In most cases where you would want to do this, the syntax allows you to use an extended expression which is preferable (for example, use COPY TO (cFileName) instead of COPY TO &cFileName).
- You can use the macro operator to substitute individual items in a command argument list but not to substitute the entire list. Again, this is not a recommended practice and in most cases you can use extended expressions instead.
- When creating an array with PRIVATE or PUBLIC, the square brackets are required to be a literal part of the syntax—as with command keywords, you cannot substitute them using a macro variable.

Chapter

24 Arrays

An *array* is a collection of related data items that share the same name. Each value in an array, referred to as an *element*, can be uniquely identified using one or more integer values called *subscripts*.

An array is said to have one or more *dimensions*, depending on how many subscripts are required to identify a particular element. The maximum number of elements in each dimension gives the *size* of an array. For example, a two-dimensional array requires two subscripts (a row and a column number) to identify an element. If the array has three rows and five columns, its size is three by five (3x5).

Some examples of arrays are a list of state names and abbreviations, a list of disk file names and other attributes, and a series of numeric values.

Visual Objects supports two distinct kinds of array, dynamic and dimensioned, both of which are discussed in this chapter.

Dynamic Arrays

Visual Objects supports an array data type that you can use to represent *dynamic* arrays. The term dynamic is used to describe the changing nature of these arrays.

Elements within the same array do not have to be of the same data type. For example, it is possible to create an array in which the first element is a string, the second a numeric value, the third a code block, and so on. You can store other arrays as elements in a dynamic array and, thus, create arrays of non-uniform dimensions. You can also change the size of this type of array throughout your application.

With dynamic arrays, not only can you manipulate the array elements, but you can also manipulate the array itself. For example, you can use dynamic arrays as arguments and return values. For this reason, dynamic arrays represent a true data type.

Literal Arrays

As a data type, dynamic arrays have a literal representation. To create a literal array, enclose zero or more expressions as a comma-separated list within the designated array delimiter pair, {}:

```
{[<uExp> [, <uExp> ... ]]}
```

The expressions represent the elements of the array and can be of any usual data type (see Chapter 22, "Variables, Constants, and Declarations"). For example, the following represents a one-dimensional array containing the numeric values 1 through 3:

```
{1, 2, 3}
```

Another one-dimensional array containing a string, a numeric value, and a date might look like:

```
{"Hello", SqRt(x), Today()}
```

Since an array element can contain another array, you can create a two-dimensional literal array as in this example:

```
{{1, 2, 3}, {"a", "b", "c"}, {.T., .T., .F.}}
```

Note: To return an array to its uninitialized null state, use the system-defined constant NULL_ARRAY (or NIL). To create an empty array (one with no elements, zero length), use the array delimiters with no element list (for example, {}).

Limitations

There is no limit to the number of dimensions per array, but the subscript operator ([]) has a limit of eight dimensions. If more dimensions are required, functional access with ArrayGet() and ArrayPut() must be used.

Creating Arrays

You can declare dynamic arrays to the compiler using the LOCAL or GLOBAL statements (see Chapter 22, "Variables, Constants, and Declarations," for more information). To do this, you specify an identifier name followed by the array dimensions in square brackets using either of the following syntax conventions:

```
<idArray>[<nElements>, <nElements>, <nElements>]
<idArray>[<nElements>][<nElements>][<nElements>]
```

Unlike other syntax representations, the square brackets (shown in bold) are a literal part of the array definition and you must include them—they do not represent an optional part of the syntax. All dimensions other than the first are optional:

```
LOCAL aMyArray[2][10]
STATIC LOCAL aYourArray[100]
GLOBAL aAnother[2, 3, 5]
```

You can also create dynamic arrays at runtime using the PUBLIC or PRIVATE statements. The syntax for creating arrays with PUBLIC and PRIVATE is the same as for GLOBAL and LOCAL.

Another way to create an array is to assign an array expression to a polymorphic variable. This technique also works with GLOBAL, LOCAL, PUBLIC, and PRIVATE, letting you create and initialize an array in the same statement:

```
aFirst := {1, 2, 3, 4, 5}
LOCAL aMyArray := ArrayCreate(100)
GLOBAL aYourArray := {"One", "Two", "Three"}
PRIVATE aAnother := Directory("*.dbf")
```

Important! *You cannot specify multidimensional literal arrays as part of a GLOBAL statement.*

Finally, you can declare a simple variable name and, later in the program, assign an array value to it. These three examples are equivalent:

```
// Example 1
LOCAL aMyArray[100]

// Example 2
LOCAL aMyArray := ArrayCreate(100)

// Example 3
LOCAL aMyArray
aMyArray := ArrayCreate(100)
```

Strong Typing

Dynamic arrays that you declare to the compiler or that you create at runtime are treated as polymorphic variables, which means that you can change their data type. Consider this example:

```
LOCAL aNums[10]
AFill(aNums, 100)
? aNums[1]          // Displays 100
...
aNums := "New character value"
? aNums[1]          // Runtime error!
```

The LOCAL statement creates *aNums* as a ten element array. AFill() assigns a value of 100 to each element so that when you query the first element, 100 is displayed. Then, the assignment statement changes the variable named *aNums* to a string value, destroying the array that was previously stored. Thus, when you query the first array element again, a runtime error occurs because *aNums* is no longer an array.

Note: The ability to use polymorphic variables to store dynamic arrays may be important in your application, but you should know that it comes at a cost. Applications that use polymorphic variables require runtime overhead and are prone to errors.

If you do not want to allow a variable name that you declare as an array to change data type in your application, you can specify the ARRAY data type (called *strong typing*) as part of the declaration statement. Doing this will trap as a compiler error any instance in which you misuse the variable name, including an attempt to change its data type or to use it where another data type is expected.

When the previous example is repeated with the variable *aNums* strongly typed as an array, a compiler error occurs when you attempt to assign a string value to *aNums*:

```
LOCAL aNums[10] AS ARRAY
AFill(aNums, 100)
? aNums[1]  // Displays 100
...
aNums := "New character value"  // Compiler error!
? aNums[1]
```

Declaring an array with the ARRAY data type does not change its dynamic nature. The data types of the individual elements can still vary along with the size and structure of the array—array elements are always untyped, usual values. See Chapter 22, "Variables, Constants, and Declarations," in this guide for more information on the advantages of declaring and strongly typing variables.

Addressing Array Elements

After you create an array, you access its elements using an integer index called a subscript. The syntax you use to address elements is the same as the syntax to specify the array dimensions.

To address an element of a one-dimensional array, place the subscript of the element in square brackets following the array name. For example, if *aNums* is a one-dimensional array you would use this syntax to address its first element:

```
aNums[1]
```

Note that subscript numbering begins with one (it is *one-based*).

To address an element of a multidimensional array, you can either enclose each subscript in a set of square brackets or separate the subscripts with commas and enclose the list in square brackets. If *aNums* is a two-dimensional array, both of these statements address the element stored in the second column of the tenth row:

```
aNums[10][2]
aNums[10, 2]
```

You can use an array element reference as a variable anywhere in the language that is appropriate for its data type.

Assigning Values to Array Elements

Array elements are assigned an initial value that differs depending on how you create the array. For instance, if you create the array by defining its dimensions using a LOCAL, GLOBAL, PUBLIC, or PRIVATE statement, all of the elements are initialized to NIL.

If you create the array using a function, the initial value of the elements depends on the function return value. With ArrayCreate(), the initial value for all elements is NIL, but with a function like Directory(), the initial value of the elements depends on the contents of the disk drive and directory that you specify.

If you create a literal array, you specify the initial value of the elements in the literal representation.

Once you create an array, you can change the value of individual elements using any assignment operator (defined in Chapter 23, "Operators and Expressions,") that is appropriate for the data type of the element:

```
// Declare aOne and initialize to NIL
LOCAL aOne[30] AS ARRAY, siCount AS SHORTINT

// Assign each element a numeric value
FOR siCount := 1 TO ALen(aOne)
     aOne[siCount] := siCount
NEXT

// Increment 1st element from 1 to 2
++aOne[1]

// Multiply 2nd element by 3 and store result 6
aOne[2] *= 3

// Divide 3rd element by 3 and store result 1
aOne[3] /= 3
```

Note: This example illustrates the traditional method for traversing a one-dimensional array with the FOR...NEXT programming construct. See the FOR entry in the online help system for more information. Also see the AEval() entry for another, less traditional method of array traversal.

The AFill() function gives you a quick way to assign the same value to all elements of an array. This example assigns a string value of "First" to each element in *aOne*:

```
LOCAL aOne[30] AS ARRAY
AFill(aOne, "First")
```

The values that you assign to array elements can be of any usual data type (see Chapter 22, "Variables, Constants, and Declarations").

Multidimensional Arrays

To create a multidimensional array, you can either declare the array with more than one dimension parameter or assign another array to one of the elements of a one-dimensional array.

Using either of these techniques, you can create and maintain traditional multidimensional arrays that have a fixed number of elements in each dimension (for example, to represent the rows and columns of a table). To declare a two-dimensional array with ten rows and two columns:

```
LOCAL aTable[10][2] AS ARRAY
```

Arrays that you create in this manner are usually expected to adhere to certain rules. For example, each column should contain the same type of information for each row in the array (for example, column one might be a string and column two a number). However, since there are no rules regarding what can be stored in an array, you must implement this level of control programmatically.

The fact that you can assign an array to an existing array element allows you to dynamically change the structure of an array. For example, there is nothing to prevent you from doing something like this:

```
aTable[1][2] := {1, 2, 3, 4, 5}
```

This assignment changes *aTable* to a point where it can no longer be thought of as a two-dimensional array because one of its elements is an array reference. In particular, a reference to *aTable*[1][2][1] is now valid (it returns 1) where it was not before. References to *aTable*[1][1][1] and *aTable*[2][1][1], however, result in runtime errors.

This feature of assigning an array reference to an array element is handy in many applications—it lets you create *ragged* arrays that have non-uniform dimensions; however, you must exercise whatever control you think is necessary for storing and addressing array elements. You should not make any assumptions about the structure of a dynamic array unless you enforce the structure programmatically.

Arrays as References

Dynamic arrays are treated as *references*. This means that a variable to which you assign an array (or which you declare as an array) does not actually contain the array. Instead, it contains a pointer to a location in memory where the actual array contents are stored.

Thus, it is possible to have multiple references to the same array in your application. When you have more than one reference to the same array, changes to one are automatically reflected in the others:

```
LOCAL aFirst, aSecond AS ARRAY
aFirst  := {1, 2, 3}
aSecond := ASize(aFirst, 5)
aSecond[4] := 4
aSecond[5] := 5
? aFirst[4]                                    // Displays 4
? aFirst[5]                                    // Displays 5
```

Since ASize() returns a reference to its argument, *aFirst* and *aSecond* are actually references to the same array; therefore, any changes made to one are automatically seen in the other.

Equal Operator

You can use the equal operator (=) to determine if two variables refer to the same array:

```
LOCAL aFirst, aSecond, aThird AS ARRAY
aFirst  := {1, 2, 3}
aSecond := {1, 2, 3}
aThird := aFirst
? aFirst = aSecond                    // FALSE
? aFirst = aThird                     // TRUE
```

In this example, even though *aFirst* and *aSecond* contain exactly the same elements, they are created independently and, therefore, are two distinct array references. In other words, they point to different locations in memory and comparing them with = returns FALSE.

However, since *aThird* is created using *aFirst*, these two variables represent the same array—they point to the same memory location—and comparing them with = returns TRUE.

Arrays as Parameters

When you pass an array as an argument, it is passed by value unless you use the reference operator (@). However, unlike parameters of other data types, any changes that you make to array *elements* in the called routine are automatically reflected in the original array (and in all references to it):

```
FUNCTION Start()
     LOCAL aNums[10] AS ARRAY
     AFill(aNums, 0) // Elements set to 0
     MyFill(aNums)            // Elements incremented by 1

     ? aNums[1]               // Result: 1

FUNCTION MyFill(aTemp AS ARRAY)
     LOCAL siCount AS SHORTINT
     FOR siCount = 1 TO ALen(aTemp)
          ++aTemp[siCount]
     NEXT
```

This is another implication of treating arrays as references. Even though the array is passed by value, the value passed is a reference and, therefore, behaves in the indicated manner.

The behavior described above holds true if you pass the array by reference using the @ operator, but there are some subtle differences between passing an array by reference and by value. Most notably, it is possible to destroy an array reference that you pass by reference:

```
FUNCTION Start()
     LOCAL aNums[5]
     AFill(aNums, 0)          // Elements set to 0
     MyFill(@aNums)           // Pass by reference

     ? aNums[1]               // Result: 1

FUNCTION MyFill(aTemp)
     aTemp := {1, 2, 3, 4, 5}
```

In this example, the function MyFill() creates a new array reference. Since *aNums* is passed by reference, this change is reflected when MyFill() returns, and *aNums*[1] contains a value of 1.

When you pass by value, this is not possible. The original array reference will always remain intact when the function returns:

```
FUNCTION Start()
     LOCAL aNums[5]
     AFill(aNums, 0)          // Elements set to 0
     MyFill(aNums)            // Pass by value

     ? aNums[1]               // Result: 0

FUNCTION MyFill(aTemp)
     aTemp := {1, 2, 3, 4, 5}
```

Arrays as Return Values

In addition to the ability to pass arrays as parameters, you can also use them as return values. This lets you create an array reference using a function or method call:

```
FUNCTION Start()
     aNew := MakeArray(10, "New value")

FUNCTION MakeArray(nElements, uFillValue)
     LOCAL aTemp[nElements]
     AFill(aTemp, uFillValue)
     RETURN aTemp
```

Dimensioned Arrays

Declaring Dimensioned Arrays

In addition to dynamic arrays, you can also have arrays whose dimensions are fixed at compile time. These are called *dimensioned arrays* and you declare them using the DIM keyword of the LOCAL and GLOBAL statements. For example:

```
LOCAL DIM aMyArray[10]
LOCAL DIM acNames[2][3] AS STRING
GLOBAL DIM aiGroups[100] AS INT
GLOBAL DIM aoCars[20] AS OBJECT
LOCAL DIM aoAnimals[15] AS AnimalClass
```

Dimensioned arrays can contain up to eight (8) dimensions, except in the case of dimensioned arrays of function pointers (see below). Strongly typing the elements of a dimensioned array is optional.

The dimensions that you specify when you declare a dimensioned array remain in effect throughout your application—you *cannot* change the size of the array at runtime as you can with dynamic arrays. When you strongly type the array elements, the compiler performs the necessary type checking to prevent you from misusing the array (for example, storing a string value to an integer array element).

Dimensioned arrays are much faster to access and more compact than dynamic arrays but, from a language standpoint, not nearly as flexible in their usage. They do not fall into the array data type category, which means that you cannot manipulate them with functions designed to accept arrays as arguments. They have other limitations as well. For example, a DIM array of FLOAT values is not initialized to 0.0.

In fact, the dimensioned array itself does not have a data type of its own and, therefore, does not represent an expression. This means that you cannot use dimensioned arrays as arguments or as return values.

In general, you cannot manipulate the array as a whole the way you can with dynamic arrays. The array elements, however, are typed and you can use them in expressions. As with dynamic arrays, dimensioned arrays are one-based.

Dimensioned arrays are compatible with C and are designed for interfacing at the operating system level.

Using Dimensioned Arrays with Function Pointers

There are some restrictions to using dimensioned arrays with function pointers. The syntax for a dimensioned array of function pointers is:

(LOCAL | GLOBAL) DIM <vName> "[" dim1 [,dim2 … dim4] "]";
<FunctionName> PTR

For example:

```
LOCAL DIM aFcnPtrs [2,3] MyFunction PTR
```

There are restrictions to consider when using dimensioned arrays with function pointers. First, the maximum number of dimensions that can be used are four, as opposed to eight for regular dimensioned arrays. Also, default initialization of function pointer variables is limited to only the first element. For a single-dimensional array, that would only be element <vName>[1]. For a four-dimensional array it would be element <vName>[1,1,1,1]. Therefore, it is the programmer's responsibility to initialize the remaining elements of the DIM array before trying to use the array.

Note: See Chapter 27, "Functions and Procedures," for further information about function pointers.

Using the Array Operator on Typed Pointers

A more flexible way of dereferencing typed pointers is provided with Visual Objects. This is achieved by applying the array operator to the typed pointer. This method is more flexible, because unlike the previous method, we are not only restricted to the current element being addressed by the pointer, but to any other element in our memory space.

Using this syntax we can obtain the value of the tenth element of our array in the following manner:

```
preal4[10]
```

At the first glance it might seem superfluous to define a typed pointer to access an array element, which could directly be accessed using the array variable. However, think of the case where you dynamically allocate and fill memory with elements of a particular data type in your application. It then becomes necessary to either access the data sequentially or randomly. In either case, one of the provided methods are readily applicable.

The following program example demonstrates this aspect. In the example we use certain features like typed pointer conversion and pointer arithmetic which are also explained in Chapter 21, "Data Types," of this guide:

```
LOCAL prandom, ptemp AS REAL4 PTR
LOCAL w AS WORD

// allocate memory area for sparse array of 100
// REAL4 elements
prandom := PTR(REAL4, MemAlloc(100 * _SizeOf(REAL4)))

// initialize sparse array
ptemp := prandom
FOR w:=1 UPTO 100
     PTR(ptemp) := 0.0
     ptemp++
NEXT

// randomly access/assign values to elements
// in sparse array
prandom[10] := 5.75
prandom[50] := 10.25
```

Array Operator used Beyond the Third Dimension

With CA-Visual Objects 1.0 you had a limit of three dimensions when using the subscript operator to access the elements of an array. To access array elements beyond three dimensions, you had to use the ArrayGet() and ArrayPut() functions, nested to the appropriate level.

Now with Visual Objects, this limit has been moved from three to eight. You can now directly use the subscript operator up to the eighth dimension to access array elements. For more information on pointers, refer to the Pointers section of Chapter 21, "Data Types," in this guide.

Chapter

25

Objects, Classes, and Methods

The Visual Objects language provides facilities for defining and otherwise manipulating objects as a data type. This chapter discusses the terminology that you will need to understand object-oriented programming and the language components that you will need to create objects and use them in an application.

In this chapter, you will learn how to define a class of objects, how to instantiate objects of a particular class, and how to invoke their methods and determine their properties.

You will see how the Visual Objects language satisfies the basic principles for an object-oriented programming language and how it provides several unique features, such as exported instance variables and access/assign methods, that are not supported in some other object-oriented languages.

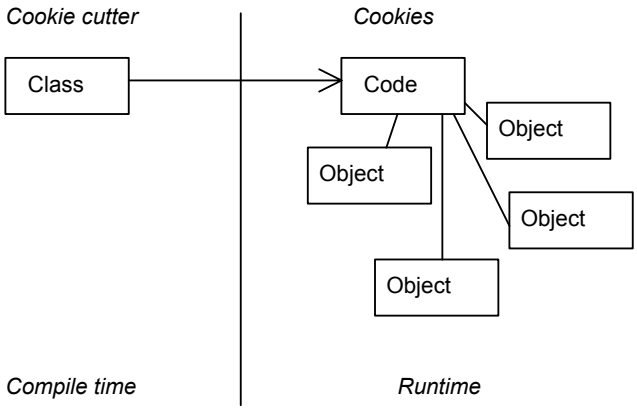
Each of the statements mentioned in this chapter is discussed further in the online help system. There are also numerous support functions for working with objects, instance variables, classes, and methods. In the online help system, these are documented individually and grouped together under Object in the "Functions by Category" chapter.

Note: In addition to providing the syntax and semantics for using your own classes, Visual Objects provides you with several predefined system classes. The system classes are not only described in the online help system, but also found earlier in this guide. You can access them in the same manner as you access classes that you define.

Classes

An *object* is a software component consisting of both code and data. It is created at runtime and exists only while the application is running.

A *class* is the code that tells the application how to implement the behavior of an object. It is a template that you use to define the components of an object (data) and its behavior (code). In Visual Objects, the class exists in code, before the program starts to run. At runtime, many objects share the code of their class. Thus, a class is to objects what a cookie cutter is to cookies.



A class consists of one or more of the following, all of which will be defined and discussed in this chapter:

- Methods
- Instance variables (both internal and exported)
- Virtual variables

The methods define the behavior of the class and the instance and virtual variables define its data. The term *property* refers to all the externally visible components of a class (that is, its virtual variables and exported instance variables).

The CLASS statement declares a class name to the compiler and begins the definition of the data portion of the class.

Methods

A *method* is the code for a single action in the behavior of an object. It is correct, therefore, to use the word "method" to refer to a portion of source code and also to refer to that code at runtime.

Declaring

Methods define what a class of objects is capable of doing. You define methods as separate programming units outside of the class entity definition using the METHOD statement.

For this reason, you must identify a class name for each method you declare. Even though methods are entities, the compiler uniquely identifies method names as part of a particular class, so there is no conflict if you use the same method name in several different classes. (Because of this, the various browsers in the IDE and this documentation refer to methods by their "full names," for example DBServer:Skip().)

A method is a lot like a function—it has parameters, declarations, programming statements, and a return value. It is different, however, in several ways:

- A method is defined for a specific class.
- At runtime, a method knows which specific class it belongs to and can, therefore, access all properties and instance variables defined for and inherited by that class.
- By default, a method returns SELF (a special keyword that refers to the object itself). If you don't want the method to return a value, you must explicitly specify RETURN NIL.

Typing

Visual Objects supports two different types of methods. Methods that do not allow typing of parameters (untyped methods) and methods with typed parameters (typed methods). The behaviors differ in the following ways. Calls to untyped methods are resolved symbolically at runtime (late-bound), incurring some overhead to the method calls. However, they are very flexible. You can check for the existence of methods, send untyped methods to an array, etc.

Typed methods are not as flexible but they provide compile-time type-checking. Also, they have better performance since the method sends are direct calls utilizing a virtual table (vtable) (early-bound).

Visibility

Methods have a visibility both from within the class and from the outside. The visibility defines how a method can be invoked and in what context. The normal, default, visibility is that all methods can be called by all other methods within the class and all subclasses. In addition, all modules outside of the class that have a pointer to an instance of the class can call all methods in the class.

There are two keywords that can be used to further limit the visibility of methods, PROTECT and HIDDEN. Protected methods can only be called by methods of the class declaring the method and all subclasses. Hidden methods can only be called by the class that declares the method.

Protected and hidden methods are declared by prefixing a method declaration with the keywords PROTECT or HIDDEN. For example:

```
PROTECT Method MyProrMethod(x, y, z) CLASS MyClass
```

Both method types can only be called in the context of their class. Thus they must be called through a self call:

```
Method SomeMethod CLASS MyClass
    self:MyProtMethod(1, 2, 3)
```

Currently Protect and Hidden Methods are implemented for late bound (untyped) methods only.

Invoking

When you invoke a method, not only must you specify the method name with its arguments, you must also specify the object for which the method should be invoked. Remember that although the method belongs to the class, it refers to the contents of instance variables and those exist only in the context of a specific object.

Method invocation is done with the *message send* operator, the colon:

```
<oObject>:<idMethod>([<uArgList>]) → uResult
```

Within a method, you must specify <*oObject*> using the keyword SELF to invoke a method in the same class. As with a function call, the parentheses must be specified even if there are no arguments, and the return value can be used or discarded.

Instance Variables

After the CLASS statement, you can declare *instance variables* that serve as the data portion of the class. Thus, instance variables, their names, and types are declared as part of the class entity at compile time.

However, the actual value of each instance variable is handled at runtime by the code in the application, particularly the methods of the class and possibly its subclasses. At runtime, each instance variable contains a value or a reference to an array or another object, and each object created from the class has these instance variables as its components. Instance variable values are kept with the object and vary from one object to another, defining the *state* of the object.

Note: A class definition with no instance variables is perfectly legitimate, of course. For example, a subclass can be used only to override methods of its parent class, without requiring any instance variables of its own.

Declaring

There are four different types of instance variables, each declared with a different keyword and each offering different characteristics:

- INSTANCE declares instance variables that are visible only in methods of their class and subclasses. These regular instance variables are "late bound": you can override them with access and assign methods of the same name.
- PROTECT INSTANCE (or just PROTECT) declares instance variables that are visible only in methods of their class and subclasses. Protected instance variables are "early bound:" you cannot override them with access and assign methods of the same name.
- HIDDEN INSTANCE (or just HIDDEN) declares instance variables that are visible in methods of their class, but not subclasses. They are also "early bound" and cannot be overridden.
- EXPORT INSTANCE (or just EXPORT) declares instance variables that are visible outside the class, as externally properties of objects. Exported variables are scoped like their class: they have module-wide visibility if their class is declared as STATIC; otherwise, they have application-wide visibility. They are "early bound" if possible, and cannot be overridden.

In summary, the different types of instance variables provide these characteristics:

	Binding		Visibility		
	Late	Early	External	Class	Subclass
EXPORT		√	√	√	√
PROTECT		√		√	√
HIDDEN		√		√	
INSTANCE	√			√	√

(The implications of late and early binding and the overriding of instance variables are discussed in the Binding of Instance Variables and Virtual Variables sections later in this chapter.)

The instance variables can be typed or untyped, like any local variable; they can also be assigned initial values, like local variables. These are all valid examples of instance variable declarations:

```
INSTANCE Tare := 800 AS INT
INSTANCE aCar := {} AS ARRAY
PROTECT Engineer
HIDDEN Engine
EXPORT License AS USUAL
```

(See Chapter 22, "Variables, Constants, and Declarations," in this guide for more information on the advantages of strong typing.)

Assigning Initial Values

If not initialized in the declaration or set in the Init() method (see the Instantiation section later in this chapter), instance variables are initialized to the appropriate null value for their type, or to NIL if untyped. (See Chapter 21, "Data Types," in this guide for more information on NIL and initial null values.)

Referencing

When you reference an instance variable outside the class (in the case of exported instance variables), you must specify the object to which the instance variable belongs. Remember that instance variables exist only in the context of a specific object.

Like method invocations, external instance variable references are accomplished using the message send operator:

```
<oObject>:<idVar>
```

SELF

Inside a class and its subclasses, instance variables are normally referenced using only their name, but you can specify <*oObject*> using the keyword SELF to resolve a potential conflict. For example, local variable declarations hide all instance variables with the same name. In such a case, you would need to distinguish the instance variable from the local variable using SELF.

NoIVarGet() and NoIVarPut()

You can prevent a runtime error from occurring when an instance variable name is not found by defining methods called NoIVarGet() and NoIVarPut(). These methods, if present, will be automatically invoked whenever an instance variable cannot be found. They are called with the instance variable name as a parameter, in the form of a symbol and, in the case of NoIVarPut(), with the value to be assigned as a second parameter. This feature is useful in detecting and preventing a runtime error and also for creating virtual variables dynamically at runtime.

The DBServer and DataWindow classes use this technique to be able to create database fields as virtual variables. The dilemma here is that after opening a database with the generic class—say opening EMPLOYEE.DBF in DBServer—you would like to be able to refer to EmpNo as a variable, but of course the DBServer *class* does not know anything about the employee database. The *object* knows about the database once you have opened it, but methods are not defined for objects, they are defined for classes. NoIVarGet() allows the object to pretend that it has a new set of variables.

```
METHOD NoIVarGet(symFieldName) CLASS DBServer
     IF AScan(aFieldNames, symFieldName) > 0
          RETURN FieldGetArea(wWorkArea, symFieldName)
     ELSE                                  // Variable doesn't exist
          SELF:Error(...)      // generate a real error
          RETURN NIL           // and return NIL
     ENDIF
```

Of course, if you use the DBServer Editor in the IDE, it generates a subclass for the particular database, so there is no need for NoIVarGet(). In that case we have a *class* that knows about the field; the NoIVarGet() technique is used for when only the *object* knows about the field.

Instantiation

Objects that you create from a class are called *instances* of that class. You will hear the terms object and instance used interchangeably. To *instantiate a class*, or *create an object*, you name the class followed by the instantiation operators, {}:

<*idClass*>{[<*uArgList*>]}

In most cases the created object is assigned to a variable, in a statement like this:

<*idObject*> := <*idClass*>{[<*uArgList*>]}

but it is also possible and useful to create objects without assigning them to anything. The object can do some useful work on its own before disappearing, it can register itself with another object and thus do the assignment implicitly, or it can be created only for the purpose of being passed as a parameter in a function call.

A function or a method can also create and return an object, using a statement like this:

RETURN <*idClass*>{[<*uArgList*>]}

In that case, you can create an object implicitly using a function or method call:

<*idObject*> := <*idFunction*>([<*uArgList*>])

<*idObject*> := <*idObject*>:<*idMethod*>([<*uArgList*>])

Init() Method

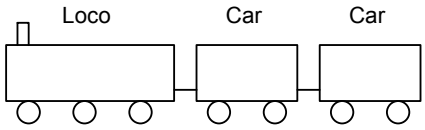
If you define a method named Init(), it is called automatically when you create an instance of the class to which the method belongs. Arguments listed within the instantiation operators are passed as parameters to the Init() method. Common uses for the Init() method are to initialize instance variables, allocate memory needed by the object, register the object, create subsidiary objects, and set up relationships between objects.

Note: This does not correspond exactly to a *constructor* method required in some languages. A constructor is required to create the object and allocate its memory; here, the object is constructed automatically—Init() is used only to do initialization processing that is required by the application logic. Hence the name, Init(), rather than Construct(), Create(), ClassName(), or New().

Important! *The Init() method must be untyped.*

Example

Here is an example that illustrates the concepts and terms discussed so far.



47 Train

```
CLASS Car                        // A railroad car
     EXPORT nTare := 1000 AS USUAL
     EXPORT nPassengers := 0 AS USUAL

METHOD Weight() CLASS Car
     RETURN nTare + nPassengers * 150
     // 150 = average passenger weight

CLASS Loco AS USUAL
     EXPORT nTare := 6000
     METHOD Weight() CLASS Loco
     RETURN nTare + 300
     // 300 = combined weight of driver and guard

CLASS Train
     EXPORT oLoco AS OBJECT
     EXPORT aCar AS ARRAY

METHOD Init() CLASS Train
     oLoco := Loco{}
     aCar := {}      // Empty array, no cars yet

METHOD AddCar() CLASS Train
     AAdd(aCar, Car{})
     RETURN NIL

METHOD Length() CLASS Train
     RETURN ALen(aCar)

METHOD Weight() CLASS Train
     LOCAL nW, k AS USUAL
     nW := oLoco:Weight()
     FOR k := 1 UPTO SELF:Length()
          nW := nW + aCar[k]:Weight()
     NEXT
     RETURN nW
```

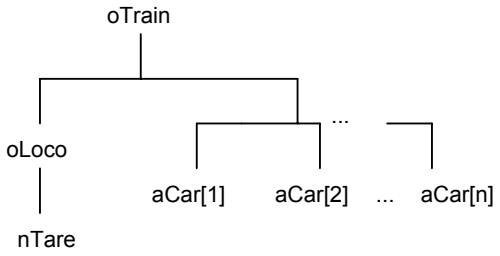
Here is some code that uses these classes:

```
LOCAL oTrain := Train{}
oTrain:AddCar()
oTrain:aCar[1]:nPassengers := 28
oTrain:AddCar()
oTrain:aCar[2]:nPassengers := 22
? oTrain:Length()                                   // Prints: 2
? oTrain:Weight()                                   // Prints: 15800
```

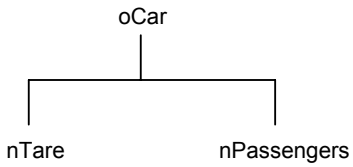
oTrain is an object whose class is Train. Train has four methods and two instance variables. The state of the object *oTrain* consists of the values of its instance variables *oLoco* and *aCar*, and indirectly of the states of the *oLoco* object.

Notice the Init() method of class Train. As explained earlier, Init() is a special method that the system calls when you create an object of that class. Car and Loco do not have Init() methods; instead, they rely on the initial values of their variables, specified in their class declarations.

The state of the object *oTrain* looks like this:



Each Car looks like:



Virtual Variables

Instance variables declared as EXPORT can be referenced from outside the class, but the others, the non-exported, or *internal*, instance variables, are not directly accessible from outside of the class. If you want to access an internal variable, you must use a method. This is a desirable behavior of *encapsulation*: by insulating the data associated with a class through code, you can ensure that only valid values are assigned. But it also has the unpleasant side effect that the interface for accessing data is dependent on the implementation of the class:

```
CLASS Employee
     EXPORT Name
     PROTECT Salary

METHOD SetSalary(x) CLASS Employee
     IF SELF:ValidSalary(x)    // Validate assignment
          RETURN Salary := x
     ELSE
          SELF:Error(...)
          RETURN NIL
     ENDIF

METHOD GetSalary() CLASS Employee
     RETURN Salary
...
oEmp := Employee{}
oEmp:Name := "Jones"
oEmp:SetSalary(10000)                    // Different syntax!
? oEmp:Name
? oEmp:GetSalary()                       // Different syntax!
```

This violates the spirit, at least, of encapsulation.

Access and Assign Methods

Instead of directly referring to the contents of an actual instance variable, you can use a *virtual variable* that simulates an instance variable by means of special methods.

In Visual Objects, the special methods are introduced by the keywords ACCESS and ASSIGN. ACCESS declares a method that is automatically executed each time you access a virtual variable with the name of the method, using the <*idObject*>:<*idVar*> syntax. Similarly, ASSIGN declares a method that is automatically executed each time you use the method name in an assignment statement, using the <*idObject*>:<*idVar*> := <*uValue*> syntax. <*uValue*> is passed as an argument to the assign method. Thus, the syntax for manipulating virtual variables is the same as for exported instance variables even though their implementation is different.

Virtual variables represent an object's *virtual state*, the state the object "pretends" to have. It is the state you see from outside the object and is actually faked by the access and assign methods. The real state is usually inaccessible from the outside.

Note: If all the instance variables are exported and there are no access or assign methods, the virtual state is the same as the real state. This is a legitimate but not very cautious programming style.

Overriding Instance Variables

Access and assign methods can be used to intercept references to regular instance variables (those declared with INSTANCE) of the same name. Thus, if you change the declaration for Salary and replace the methods in the example above with access and assign methods, you turn Salary into a virtual variable and eliminate the special syntax needed to manipulate it:

```
CLASS Employee
     EXPORT Name
     INSTANCE Salary

ASSIGN Salary(x) CLASS Employee
     IF SELF:ValidSalary(x)    // Validate assignment
          RETURN Salary := x
     ELSE
          SELF:Error(...)
          RETURN NIL
     ENDIF

ACCESS Salary() CLASS Employee
     RETURN Salary

...
oEmp := Employee{}
oEmp:Name := "Jones"
oEmp:Salary := 10000 // Same syntax!
? oEmp:Name
? oEmp:Salary                              // Same syntax!
```

Note: You cannot override exported, protected, or hidden instance variables in this manner. See Binding of Instance Variables later in this chapter for more information.

Calculated Variables

Virtual variables have other applications besides providing a consistent interface for instance variables of the same name. For example, you can create virtual variables that are calculated based on the values of other instance variables. As with internal instance variables, you could use a regular method to compute virtual variables, but this would mean using a different syntax for accessing them. Access methods extend the syntax used for accessing instance variables to computed virtual variables.

To illustrate this point, the Train example introduced earlier is repeated, this time using virtual variables:

```
CLASS Car                       // A railroad car
     HIDDEN nTare := 1000 AS USUAL
     HIDDEN nPassengers := 0 AS USUAL

ASSIGN Passengers(nPeople) CLASS Car
     RETURN nPassengers := nPeople

ACCESS Passengers() CLASS Car
     RETURN nPassengers

ACCESS Weight() CLASS Car
     RETURN nTare + nPassengers * 150
     // Average passenger weighs 150 pounds

CLASS Loco
     HIDDEN nTare := 6000 AS USUAL

ACCESS Weight() CLASS Loco
     RETURN nTare + 300
     // + driver + guard

CLASS Train
     HIDDEN oLoco AS OBJECT
     HIDDEN aCar AS ARRAY

METHOD Init() CLASS Train
     oLoco := Loco{}
     aCar := {}                 // Empty array, no cars yet

METHOD AddCar() CLASS Train
     AAdd(aCar, Car{})          // Hitch up a new Car
     RETURN NIL

ACCESS LastCar() CLASS Train
     RETURN aCar[ALen(aCar)]

ACCESS Length() CLASS Train
     RETURN ALen(aCar)

ACCESS Weight() CLASS Train
     LOCAL nW, k AS USUAL
     nW := oLoco:Weight
     FOR k := 1 UPTO SELF:Length
          nW := nW + aCar[k]:Weight
     NEXT
     RETURN nW
```

Note: When referring to a virtual variable in a method of the same class, SELF: is optional if the method overrides a defined INSTANCE variable and required otherwise.

Here is some code that uses these classes:

```
LOCAL oTrain := Train{}
oTrain:AddCar()
oTrain:LastCar:Passengers := 28
oTrain:AddCar()
oTrain:LastCar:Passengers := 22
? oTrain:Length                                         // Prints: 2
? oTrain:Weight                                         // Prints: 15800
```

Train has two methods, three virtual variables, and two internal variables. The virtual variable Length is a good example of something that appears like a variable to the code that uses the class. Inside, it is a call to the ALen() function. The virtual variable Weight works by summing the weights of all the parts of the train. The internal state of the object *oTrain* consists of the values or states of *oLoco* and *aCar*.

Virtual variables can have one of three implementations:

- Read-only, where there is an access but no assign method (LastCar, Length, and Weight in the example)
- Read-write, where both methods are supplied (Passengers in the example)
- Write-only, where there is an assign but no access method (much less common than read-only and read-write)

The purpose of virtual variables is to hide and simplify the internal structure of the object's state. For example, the code that uses the Train no longer knows that an array is used in its implementation. If the train application grew so that you needed to store more information about each piece of rolling stock, you could store the details of each Car in a database, and the code that uses the Train class would still work—without even knowing you had made the change.

Encapsulation

The term *encapsulation* was introduced at the beginning of this section to refer to the insulation of the inside of a class from changes you make outside it, and vice versa.

In the second Train example above, all of the classes are well-encapsulated. Code outside each class cannot see the internal state of an object of that class. It can only see the virtual state.

This encapsulation was achieved by declaring the instance variables inside the class as *hidden* instead of *exported*; the only way that code outside the class can touch hidden and other internal variables is through methods.

Access and assign methods enable you to construct a complete encapsulation of the class, consisting of methods for the set of actions that the class knows how to perform and virtual variables for the set of data that the class exhibits. You can see this difference by comparing the two train examples presented so far.

While most object-oriented languages support encapsulation of behavior, no other commercial language supports encapsulation of the data interface to a class as well.

You can change the way that a class does something internally, as long as you do not change its interface. You do not even have to recompile any code that uses the class, a big advantage, especially when the class is part of an extension library. This is why encapsulation is considered good practice. It limits change propagation and, therefore, reduces the maintenance burden and improves reliability.

Inheritance

The programming technique by which you adapt the behavior of a class without changing its definition is called *inheritance*. Because you can give the class new behavior without in any way destabilizing it, you achieve incremental enhancement. You progress from a stable status to an improved stable status.

When a class inherits from another class, it is called a *subclass*, and it inherits the following:

- Methods
- Instance variables (except hidden ones)
- Virtual variables

The subclass can then add instance variables and add to or redefine inherited methods and virtual variables. Inheritance is accomplished using the INHERIT keyword of the CLASS statement.

The Class Tree

Inheritance forms a tree (called the *class tree* or, less precisely, the *class hierarchy*): each class can have at most one *ancestor* that it inherits from and any number of *descendants*. However, even though a class has only one direct ancestor, that ancestor may inherit from another class, and so on. Thus, a subclass can potentially inherit from many classes. The term *superclass* refers to any one of the classes from which a subclass ultimately inherits, while the term *parent class* refers to its immediate ancestor.

Example with Inheritance

You could modify the Train example above so that Loco inherits from Car. Here is the new code:

```
CLASS Car       // Now means any rolling stock
     PROTECT nTare := 1000 AS USUAL
     PROTECT nLoad := 0 AS USUAL

ASSIGN Passengers(nPeople) CLASS Car
     RETURN nLoad := nPeople * 150

ACCESS Passengers() CLASS Car
     RETURN nLoad

ACCESS Weight() CLASS Car
     RETURN nTare + nLoad

CLASS Loco INHERIT Car

METHOD Init() CLASS Loco
     nTare := 6000
     nLoad := 300              // Crew, actually
```

You can remove ACCESS Weight from class Loco—it now inherits its Weight from class Car. Because of the good encapsulation in the second version of the Train example, you do not have to change any code in either the Train class or the code that uses it. Note that the virtual variable Passengers now stores the weight of the passengers instead of their number. The benefit of this version with inheritance is that you can now write:

```
CLASS FreightCar INHERIT Car

METHOD Init() CLASS FreightCar
     nTare := 800

ASSIGN Load(nNet) CLASS FreightCar
     RETURN nLoad := nNet
```

and you can add an array of freight cars to your train.

Resolving Method Invocations

Inheritance is also used to determine the correct method to invoke when you send a message to an object at runtime (all method invocations are *dynamically*, or *late*, *bound*). If the method is not defined for the class, the runtime system attempts to locate an inherited method with the same name. Only when all inheritance possibilities have been exhausted will you receive an error message that the method is not defined.

NoMethod()

You can prevent a runtime error from occurring when a method name is not found by defining a method called NoMethod(). This method, if present, will be automatically invoked whenever a method cannot be found. Any arguments passed to the original method will be passed to NoMethod() as well.

To determine the method name that caused the problem, use the *function* NoMethod() inside your *method* NoMethod(). For example:

```
METHOD NoMethod() CLASS Jaguar
     LOCAL symMethodName := NoMethod() AS SYMBOL
     ErrorMessage(AsString(symMethodName) +        ;
          " not implemented for class " +                    ;
          ClassName(SELF))
```

Referring to the Superclass

SUPER is a special keyword that refers to the class that is the nearest ancestor of the method lookup. It passes a message up the inheritance tree to the superclass and is meaningful only if the current object's class inherits from another class.

You can use SUPER with the message send operator to refer directly to a method defined in a superclass. If you redefine a method in a subclass (by creating a method with the same name as one in a superclass), SUPER is the only way you can override the redefined method with the superclass version.

This is commonly done to make a subclass method *add* some specific processing to an inherited method but not *replace* it completely. It is particularly common in Init() methods that add some subclass-specific initialization. In this example, the class for NY employees has specific constraints on the values allowed for two fields:

```
CLASS NYEmployee INHERIT Employee
 ...
METHOD Init() CLASS NYEmployee
     SUPER:Init()                          // Regular initialization
     FieldSpec(#State):SetValidation({||State="NY"})
     FieldSpec(#Salary):SetRange(1000, 2000)
     RETURN SELF
```

Declaring Object Variables

An object can be stored in any type of variable: undeclared, declared (as LOCAL, GLOBAL, or one of the types of INSTANCE variable), or created dynamically (with PUBLIC or PRIVATE). For example:

```
LOCAL oPoint
STATIC LOCAL oHideIt
GLOBAL oJaguar, oLion, oTiger
EXPORT INSTANCE oParent
PROTECT INSTANCE oChild
PRIVATE oSecret
```

With LOCAL and INSTANCE declarations (but not with GLOBAL), you can instantiate the object in the declaration.

```
LOCAL oPoint := Point2D{3, 4}
STATIC LOCAL oHideIt := SecretClass{}
```

Strong Typing

Objects can be assigned to undeclared or untyped, polymorphic variables, but as always, these are not safe because they do not provide for compile-time checking of consistent use:

```
LOCAL oJag
...
oJag := Animal{}                        // Creates oJag from Animal class
oJag := 100                             // Changes oJag to numeric
...
oJag:Display()                  // Runtime error!
```

To protect against inadvertent misuse like this, an object variable can be typed as OBJECT:

```
LOCAL oJag := Animal{} AS OBJECT
...
oJag := 100                             // Compiler error!
```

You can further limit the type of object that can be assigned to a variable by naming the class in the declaration statement:

```
LOCAL oJag := Animal{} AS Animal
...
oJag := Car{}                   // Compiler error!
```

The AS Animal clause means that you can assign only instances of the Animal class (and its subclasses) to *oJag*. If the Class Checking compiler option is checked, the statement oJag := Car{} causes a compiler error because it would change *oJag* from an instance of the Animal class to an instance of the Car class.

See Chapter 22, "Variables, Constants, and Declarations," in this guide for more information on the advantages of declaring and strongly typing variables.

Binding of Instance Variables

References to instance variables can be either *early* or *late bound*, depending on how your references are made and how the variables are declared.

Early binding (also called *compile-time* or *static* binding) means that the compiler knows exactly how to reference the variable *at compile time* and generates code to do so.

Late binding (also called *runtime* or *dynamic* binding) is necessary if the compiler cannot determine from the program source code exactly where the variable is or how to go about referencing it. In these cases, it generates code to look the symbol up in a table at runtime.

Obviously, using early bound instance variables improves performance because there is no need for all that runtime lookup. In addition, early bound references allow the compiler to do compile-time validation of data types.

On the other hand, late bound references give you a lot of flexibility in structuring your applications. For example, if you have several classes that all have an instance variable Label, you can write some code that takes an unknown object and prints or displays its label, regardless of what kind of object it is given. You can simply reference oObject:Label, trusting that the runtime system will resolve the reference if possible. (Of course, you must ensure that only objects that have a Label variable are used in this way.)

Isomorphism vs. Polymorphism

This is a key example of both isomorphism and polymorphism. *Isomorphism* means that different types of objects look the same when viewed from the outside (in effect, they respond to the same *message*, whether it be a method invocation or a reference to a property, in some reasonably consistent way). *Polymorphism*, on the other hand, is when a name (such as a variable name) can represent many different data types (hence the term "polymorphic variable").

Isomorphism and polymorphism work together in object oriented programming (OOP). In this example, both the variable *oObject* and the function or method containing the code that invokes oObject:Label are polymorphic. The objects themselves are isomorphic—because they have the same kind of behavior, you can ignore the differences between them.

Thus, your choice of programming approach depends on the requirements of your application.

Early or Late Bound

Whether a reference to an instance variable is early or late bound depends both on how the instance variable is declared and on how the reference to the object is declared. A reference is early bound if both of these conditions are satisfied:

- The reference to the variable is resolvable at compile time, which in turn requires that one of these conditions be satisfied:
  - The object is held in a variable, which is declared as a class
  - The type inferencing engine of the compiler can determine the class of the object held in an untyped variable
  - The object reference is to SELF, explicitly or implicitly.
- The instance variable is declared as EXPORT, PROTECT, or HIDDEN.

PROTECT and HIDDEN instance variables are referred to only from inside methods of the class, so the object reference is always SELF, whether explicitly stated or implied; hence, the type of the object is always known, and early binding is always possible.

EXPORT variables, on the other hand, can be referenced outside of the class and, therefore, early binding may not be possible. For example, in this code the reference to oNum:Number is early bound:

```
CLASS MyClass
     EXPORT Number AS INT

FUNCTION SomeOtherFunction()
     ...
     RETURN oVal                         // oVal is object of MyClass

FUNCTION UseClass()
     LOCAL oNum AS MyClass
     oNum := SomeOtherFunction()
     oNum:Number := 5                    // This reference is early
                                         // bound since the compiler
                                         // knows what both oNum and
                                         // Number are.
```

However, if you replace the code for UseClass() as shown below, *oNum* is undeclared and, therefore, oNum:Number is late bound:

```
FUNCTION UseClass()
     LOCAL oNum
     oNum := SomeOtherFunction()
     oNum:Number := 5                    // At runtime, the system
                                         // will:
                                         // 1.  Find the class of oNum
                                         // 2.  Search class for the
                                         //     location of Number
                                         // 3.  Check type of Number
                                         // 4.  Store the value 5 into
                                         //     that location
```

Note: Dynamic array elements are untyped, so a reference like:

```
cClientName := aClients[i]:Name
```

is always late bound.

Overloading Instance Variables

With regular instance variables, which are always late bound, it is possible to "overload" the variable name with access and assign methods, intercepting a reference to it and replacing it with a call to the method. You do this by creating an access or assign method with the same name as the INSTANCE variable:

```
CLASS Employee
     EXPORT Name
     INSTANCE Salary

ASSIGN Salary(x) CLASS Employee
     IF SELF:ValidSalary(x)    // Validate assignment
          RETURN Salary := x
     ELSE
          SELF:Error(...)
          RETURN NIL
     ENDIF

ACCESS Salary() CLASS Employee
```

```
    RETURN Salary
```

In this case, the value of Salary can be accessed just like any instance variable (late bound), but when assigning a value to Salary the ASSIGN method intercepts the assignment, checks its validity and passes it on only if it is valid. Thus, any method of the class Employee, *or a subclass of Employee*, can reference the Salary and unknowingly get the assigned value checked.

For example, look at the assignment statement in the third line of ASSIGN Salary:

```
RETURN Salary := x
```

Why does this reference not call the ASSIGN method again, resulting in an infinite loop? The compiler is smart enough to know that references to the variable inside an access or assign method *of the same name* should be passed directly to the variable.

Note that the runtime resolution of a late bound reference takes into account the methods of subclasses. Consider this example:

```
CLASS Employee
    EXPORT Name
    INSTANCE Salary
    INSTANCE City, State

METHOD AssignOffice(cCity, cState) CLASS Employee
    City := cCity
    State := cState
    RETURN NIL

CLASS NYEmployee INHERIT Employee

ASSIGN State(c) CLASS NYEmployee
    IF c == "NY"
        RETURN State := c
    ELSE
        SELF:Error(...)
        RETURN NIL
    ENDIF

FUNCTION DealWithEmployee(oEmp)
    ...
    oEmp:AssignOffice(cCity, cState)
```

The method AssignOffice() is declared at the level of the Employee class, and innocently makes reference to the instance variable State. But at runtime, the variable *oEmp* may have been created as a regular employee or as a NY employee, and in the latter case the reference is intercepted and checked by the assign method *at the lower level*. This is one way for a subclass to redefine the behavior of an inherited method (of course, the other way is to replace the inherited method altogether).

To protect your code against this kind of modification from a lower level, use PROTECT variables which are early bound and cannot be rerouted through an access or assign method, or HIDDEN variables which are not even visible in subclasses. This is the approach used in the built-in classes: for example, the DBServer class has a virtual variable called Alias, implemented as an access. This access method does not reference a late bound INSTANCE called Alias, but rather an early bound PROTECT INSTANCE called symAlias.

Binding of Methods

All method references are late bound: the system determines at runtime which version of the method is to be called, depending on the class of the object.

Again, this is a key aspect of isomorphism: different classes can have a Print() method, and your code can make a reference like oObject:Print() without knowing what kind of object it is dealing with. And again, just as with the access and assign methods, a method invocation at one level can be redirected by a new class implementation at a lower level:

Using a vtable for typed methods ensures that this works with typed methods too.

```
CLASS Employee
     INSTANCE City, State

METHOD AssignOffice(cCity, cState) CLASS Employee
     City := cCity
     State := SELF:LookUpState(cCity, cState)
     RETURN NIL

METHOD LookUpState(cCity, cState) CLASS Employee
     IF cState = NIL
          RETURN FindStateFromCity(cCity)
     ELSE
          RETURN cState
     ENDIF

CLASS NYEmployee INHERIT Employee

METHOD LookUpState(cCity, cState) CLASS NYEmployee
     LOCAL cST := SUPER:LookUpState(cCity, cState)
     IF cST == "NY"
          RETURN cST
     ELSE
          SELF:Error(...)
          RETURN NIL
     ENDIF
```

Here, the subclass NYEmployee redefines the LookUpState() method, and this new version is used for all NYEmployee objects—even when the invocation of the method is made from a method of the parent class Employee, as in the AssignOffice() method.

Typed Early Bound Methods

Visual Objects permits the utilization of strongly typed messages in addition to its current untyped message implementation. The main aim of introducing strong typing of messages in Visual Objects is to provide the application programmer with a mechanism through which very stable code can be obtained. The typed information supplied enables the compiler to perform the necessary type checking providing much greater stability and quality of the code.

Another benefit obtained by utilizing strongly typed messages is that of performance. The Visual Objects implementation of typed messages assumes that the programmer employs strongly typed messages and that the compiler can effectively performs an early binding for the respective message invocation. As a result of this implementation, typed message invocations are somewhat faster than the respective untyped counterparts. However, these advantages are attained at the price of losing the flexibility which untyped messages offer.

Important! *It is therefore important to remember that using both the typed and the untyped versions of a particular message interchangeably in an inheritance chain is neither permissible nor possible.*

Visual Objects allows strong typing of METHODs, ACCESSes and ASSIGNs. The programmer accomplishes the specification of strongly typed messages with Visual Objects in two steps.

Step 1

An optional declaration of the typed message is given in its respective class. Although this declaration is optional, it is in some rare cases compulsory, this being the most adequate means for the programmer to provide the Visual Objects system with a specific ordering of all typed methods introduced with the definition of the respective class.

This ordering is of vital importance and the only guarantee to the programmer that all sub-classes of the defining class will inherit the typed messages in the same order specified in the introducing class. This ordering guarantees the consistency of the *virtual table* that Visual Objects employs for the invocation of typed messages. A re-declaration of a message in a sub-class is permissible; however, it is irrelevant for the ordering specified in the parent class. The following syntax is used in specifying the message declaration:

```
DECLARE ( ACCESS | ASSIGN | METHOD )
    <message_identifier_list>
```

Step 2

Define the strongly typed message. As in the case of strongly typed functions, message typing is achieved by strongly typing at least one of the arguments of the message and/or specifying a valid calling convention. The following calling conventions are valid for typed messages: STRICT, PASCAL or CALLBACK. One of the following syntactical expressions can be used in defining the strongly typed message:

```
ACCESS <xyz_access> [ "(" ")" ] <return_value_spec>;
     (STRICT|PASCAL|CALLBACK) CLASS xyz_class

     ASSIGN <xyz_assign> "("<typed_parameter_spec>")" [<return_value_spec> ] ;
     (STRICT|PASCAL|CALLBACK) CLASS xyz_class

METHOD <xyz_method> "("[<typed_parameter_list_spec>])" <return_value_spec> ;
     (STRICT|PASCAL|CALLBACK)  CLASS xyz_class
```

The visibility of typed methods can also be influenced by using the HIDDEN and PROTECT modifiers analogously to their use with instance variables. SUPER calls of HIDDEN methods in parent classes cannot be done by methods of the sub-classes.

The following example program illustrates the use of strongly typed messages:

```
FUNC Start
     LOCAL o AS MyClass
     LOCAL o2 AS MyClass2
     LOCAL o3 AS OBJECT

     // MyClass
     o:=MyClass{}

//Access instance variable
MessageBox(0,"Typed access o:MyLVValue" + Str(o:MyLVValue), "Test",0)

o:MyLVValue := 255

//Access instance variable
MessageBox(0,"Typed access o:MyLVValue" + Str(o:MyLVValue), "Test",0)

// Calling methods directly (using v-table)
//o:Typed(1)
//o:Typed4(4)

// Calling methods indirectly (using v-table)
o:D()

// MyClass2
o:=MyClass2{}

o2:=o
o3:=o

// Calling methods directly (using v-table)
//o:Typed(1)
//o:Typed2(2)
//o:Typed3(3)


//o:Typed5(5)
// note that this expression produces a late
// bound send because "o" is declared as an
// object of MyClass and there is no Typed5()
```

```
// method in MyClass

//o3:Typed5(5)
// a late bound send is also generated here
// because "o3" is declared as OBJECT

o2:Typed5(5)          // this generates an early bound message call

// Calling methods indirectly (using v-table)
o:D()

RETURN(NIL)

CLASS MyClass
     DECLARE METHOD Typed, Typed2
     PROTECT lValue AS LONG
     DECLARE METHOD Typed3
     DECLARE ACCESS MyLValue
     DECLARE METHOD Typed4
     DECLARE ASSIGN MyLValue

ACCESS MyLValue() AS LONG PASCAL CLASS MyClass
     RETURN lValue

ASSIGN MyLValue(lAssignValue AS LONG);
     AS LONG PASCAL CLASS MyClass
lValue := lAssignValue
RETURN lValue

METHOD Typed(x AS INT) AS MyClass CLASS MyCLass
     MessageBox(0,"MyClass:Typed()", "Test",0)
RETURN(SELF)

METHOD Typed2(x AS INT) AS MyClass CLASS MyCLass
     MessageBox(0,"MyClass:Typed2()", "Test",0)
RETURN(SELF)

METHOD UnTyped(x) CLASS MyCLass
     MessageBox(0,"MyClass:UnTyped()", "Test",0)
RETURN(SELF)

METHOD Typed3(x AS INT) AS MyClass CLASS MyCLass
     MessageBox(0,"MyClass:Typed3()", "Test",0)
RETURN(SELF)

HIDDEN METHOD Typed4(x AS INT) AS MyClass CLASS MyCLass
     MessageBox(0,"MyClass:Typed4()", "Test",0)
RETURN(SELF)

METHOD D CLASS MyClass
     MessageBox(0,"In D:  call Typed(), Typed2(),;
          Typed3()", "Test",0)
SELF:Typed(1)
SELF:Typed2(2)
SELF:Typed3(3)

CLASS MyClass2 INHERIT MyClass
     DECLARE METHOD Typed5
// we could redeclare Type, Typed2, Typed3,
// Typed5 here note that Typed4 is hidden and is
// consequently not seen in this sub-class

METHOD Typed2(x AS INT) AS MyClass2 CLASS MyCLass2
     MessageBox(0,"MyClass2:Typed2()", "Test",0)
     SELF:Typed3(3)
     RETURN(SELF)
```

```
METHOD Typed(x AS INT) AS MyClass2 CLASS MyCLass2
      MessageBox(0,"MyClass2:Typed()", "Test",0)
      SUPER:Typed(1)
RETURN(SELF)

METHOD Typed5(x AS INT) AS MyClass2 CLASS MyCLass2
      MessageBox(0,"MyClass2:Typed5()", "Test",0)
      SUPER:Typed4(x)                    // compiler error, because
                                         // parent method is hidden!!!
RETURN(SELF)

METHOD Typed3(x AS INT) AS MyClass2 CLASS MyCLass2
      MessageBox(0,"MyClass2:Typed3()", "Test",0)
RETURN(SELF)
```

Typed Method Restrictions and Pitfalls

The Init method should not be strongly typed. As a matter of fact, strong typing is not applicable to all methods whose logic highly depends on the Visual Objects runtime sub-system.

Always remember that a major prerequisite for early binding of a message is the manner in which the declaration of the message recipient is specified. SUPER and SELF declarations always guarantee (in the case that all other restrictions are met) early message binding. The "AS class_xyz" declaration will lead to early binding in all cases where the invoked message is typed and also defined for the specified class (class_xyz).

The following mistakes are typically made:

Message declaration is specified in class but respective definition is missing.

- The linker notices this discrepancy and signals the inherent GPF of the linked program.

Message declaration is specified in the class but the respective definition is not strongly typed.

- The compiler generates late message send for all message invocations.
- The linker notices this discrepancy and signals the inherent GPF of the linked program.

The message recipient of the typed/declared message is wrongly declared.

- The compiler generates late message send for all message invocations.
- The linker notices this discrepancy and signals the inherent GPF of the linked program.

Objects as References

In Visual Objects, objects are treated as *references*. This means that a variable to which you assign an object does not actually contain the object. Instead, it contains a pointer to a location in memory where the actual object contents are stored.

Thus, it is possible to have multiple references to the same object in your application. When you have more than one reference to the same object, changes to one are automatically reflected in the others:

```
LOCAL oOne, oTwo AS MyClass
oOne := MyClass{}
oTwo := oOne
oTwo:MyExportVar := 10
? oOne:MyExportVar              // Displays 10
```

Equal Operator

You can use the equal operator (=) to determine if two variables refer to the same object:

```
LOCAL oOne, oTwo, oThree AS MyClass
oOne := MyClass{}
oTwo := MyClass{}
oThree := oOne
? oOne = oTwo                   // FALSE
? oOne = oThree                 // TRUE
```

In this example, even though *oOne* and *oTwo* are instantiated using the same class, they are created independently as separate instances of the same class and, therefore, are two distinct object references. In other words, they point to different locations in memory and comparing them with = returns FALSE.

However, since *oThree* is created using *oOne*, these two variables are actually references to the same object—they point to the same memory location—and comparing them with = returns TRUE.

Objects as Parameters

When you pass an object as an argument, it is passed by value unless you use the reference operator (@). However, unlike parameters of other data types, any changes that you make to the objects exported instance variables in the called routine are automatically reflected in the original object (and in all references to it).

This is another implication of treating objects as references. Even though the object is passed by value, the value passed is a reference and, therefore, behaves in the indicated manner.

The behavior described above holds true if you pass the object by reference using the @ operator, but there are some subtle differences between passing an object by reference and by value. Most notably, it is possible to destroy an object reference that you pass by reference. When you pass by value, this is not possible. The original object reference will always remain intact when the function returns, even though its instance variables may be altered.

Destroying Objects

Just as there is no need to explicitly allocate memory when you create an object, there is usually no need to explicitly deallocate memory or otherwise destroy an object. Simply discard the object when you no longer need it, dropping it where you stand, and the automatic garbage collector will pick it up and dispose of it properly. Or more precisely, the garbage collector destroys objects that do not have any references—since there is no way to refer to them, there is no way to use them or cause them to take any action.

For example, consider a function that creates an object and assigns it to a local variable:

```
FUNCTION UseObjectBriefly()
     LOCAL oObject
     oObject := SomeClass{}
     ...<Some Actions>
     RETURN
```

When the function returns, the local variable disappears: it is deallocated as the stack frame for the function is removed. The object is not deallocated with the function, but when the reference to it disappears it is, for all practical purposes, gone.

However, the function may have presented the object to some other part of the program, which retained a reference to it. The object may even have registered itself with some administration facility—the window classes of the GUI Classes library, for example, keep themselves alive even without any references by the programmer, by registering themselves with the App object that represents the entire application.

In any case, the automatic garbage collector keeps track of all these references: it will not discard the object while it can still have some effect on the world, and it will dispose of it as soon as it cannot. Thus, in general the developer need not consider destruction of objects.

Axit() Method

However, in some cases an object can manage other resources that do need proper disposition. For example, if an object opens a database only for its own use, the object should close the database when it is finished and make the work area available to other uses. But if you do not want to explicitly destroy the object, how can you ensure that the database is not left open?

If you register an object with the RegisterAxit() function, for example in the Init() method, and provide a method named Axit(), this Axit() method will automatically be called by the garbage collector just before the object is destroyed. Thus, in the Axit() method you can close databases, deallocate memory, or close communications links.

Note: There are circumstances under which you will want to close files and deallocate other resources as part of an error recovery procedure. See Chapter 14, "Error and Exception Handling," presented earlier in this guide for a discussion.

However, as is obvious from these examples, this type of resource management is rarely needed. Indeed, the question of closing databases illustrates another benefit of using the object-oriented approach: the DBServer class automatically closes databases that are no longer needed, using their own Axit() method. Because of the extensive automatic resource management by the runtime system and the system classes (all of them have predefined Axit() methods, when needed), you can build fairly sophisticated classes without having to program your own Axit() methods.

There are, however, two additional points:

- For optimal performance, especially in limited memory environments, you may want to explicitly destroy objects when you are finished with them, especially complex objects such as windows. Indeed, the built-in window classes have Destroy() methods that properly dispose of all the resources the window may have claimed.
- If you write classes that manage outside resources, you will probably need to provide your own Axit() methods.

Using Arrays of Objects

Instead of using an object as the first operand when invoking a method with the message send operator, you can use an array. The array must contain objects (or other arrays that eventually lead to objects) as its elements, and the system will automatically invoke the method for each element.

Note: This feature only works for untyped methods. If you want to use typed methods, you might want to create untyped wrapper methods that call the typed version.

As a practical example of using arrays to store objects, a data server object keeps an array of its clients. When the data server needs to send an event notification to all of its clients, it simply uses this array name with the proper notification method. For example:

```
aClients:NotifyRecordChange()
```

would be equivalent to:

```
FOR i := 1 TO ALen(aClients)
     aClients[i]:NotifyRecordChange()
NEXT
```

Tip: If the first operand of the message send operator is neither an object nor an array, the runtime system will attempt to send the message to the object registered with the SysObject() function. The first operand, in this case, is passed as an argument to the indicated method. See the SysObject() entry in the online help for more information.

Operator Methods

If the Operator Methods compiler option is checked, the compiler will convert certain operations to method invocations according to the table below. The conversion requires that the left-hand operand (for example *a* in the table) be of the object data type.

Operation	Method
a + b	a:Add(b)
a - b	a:Sub(b)
a * b	a:Mul(b)
a / b	a:Div(b)
a ^ b a ** b	a:Pow(b)
a % b	a:Mod(b)
a > b	a:Gtr(b)
a < b	a:Less(b)
a >= b	a:GtrEqu(b)
a <= b	a:LessEqu(b)
a++	a:Add(1)
a--	a:Sub(1)
-a	a:Neg()
!a	a:Not()

Note: If the object is stored as a polymorphic variable, the Operator Methods switch has no effect because compile-time checking is not possible for polymorphic variables. In this case, operators are *always* converted to method invocations, regardless of the switch setting. This implementation is designed to give you the most flexibility.

Using this feature, you can define methods for these operators in your own classes. For example, this code defines a class for time and implements the plus and minus operations with Add() and Sub() methods:

```
FUNCTION Start()
     LOCAL oTime AS TimeClass
     oTime := TimeClass{}
     ? oTime + 4
     ? oTime - 4
     ? oTime - Time()

CLASS TimeClass
     INSTANCE cTime

METHOD Init() CLASS TimeClass
     cTime := Time()
     RETURN SELF

METHOD Add(nSeconds) CLASS TimeClass
     RETURN TString(Secs(cTime) + nSeconds)

METHOD Sub(uTimeValue) CLASS TimeClass
     IF UsualType(uTimeValue) == STRING
          RETURN TString(Secs(cTime) - Secs(uTimeValue))
     ELSE
          RETURN TString(Secs(cTime) - uTimeValue)
     ENDIF
```

This example implements simple arithmetic for time strings, causing them to behave similarly to dates (for example, <*dValue*> - <*nValue*> and <*dValue*> + <*nValue*> are predefined by the system).

Note, however, that in this example, addition is not commutative. In other words, you can add a number to a time string, but not vice versa: <*nValue*> + <*tValue*> does not invoke Add() since <*nValue*> is not an object. Since everybody expects addition to work both ways, this type of use of operator methods could be misleading. But the technique works well for adding two objects:

```
FUNCTION Start()
     ...
     oSalary := Remuneration{12000, 3600}
     oCommission := Remuneration{4300, 1290}
     // Addition of Remunerations is commutative-
     // these statements produce the same result:
     oTotalPay := oCommission + oSalary
     oTotalPay := oSalary + oCommission

CLASS Remuneration
     EXPORT Amount AS INT
     EXPORT Tax AS INT
```

```
METHOD Init(nAmount, nTax) CLASS Remuneration
     Amount := nAmount
     Tax := nTax
     RETURN SELF

METHOD Add(oAddend) CLASS Remuneration
     RETURN Remuneration{                                          ;
               SELF:Amount + oAddend:Amount,       ;
               SELF:Tax + oAddend:Tax}
```

Of course, you can combine the two techniques: you can write an Add() method that accepts either an object or a number as an addend, or even objects of different classes—you can make the Add() method as polymorphic as you like.

Chapter

26 Code Blocks

A *code block* is a piece of compiled code that you form using one or more expressions and an optional list of arguments. It provides you with a means for passing executable program code from one place in a system to another. It may help you to think of a code block as an anonymous function.

Code blocks are a true data type, and you can therefore use them as arguments and return values; however, the valid code block operator set is limited to assignment and a special case in which you can use the macro operator to return a code block. This chapter discusses various aspects of working with code blocks, including how to declare, create, and evaluate them.

Literal Code Blocks

As a data type, code blocks have a literal representation. To create a literal code block, use this syntax:

{|[<*idArgumentList*>]| <*uExpList*>}

Both <*idArgumentList*> and <*uExpList*> are comma-separated lists.

<*idArgumentList*> is a list of variable names that you use within the code block expression list to identify parameters passed to the code block.

Important! *Even if you do not specify any arguments, you must include the vertical bars that delimit the argument list to distinguish a code block from a literal array.*

The expressions in <*uExpList*> can be of any usual data type (see Chapter 22, "Variables, Constants, and Declarations,")—no commands or statements such as control structures and declarations are allowed.

These are examples of valid code blocks:

```
{|| "just a string"}
{|nValue| nValue + 1}
{|nOne, nTwo|  SqRt(nOne) + SqRt(nTwo)}
{|a, b, c|  MyFunc(a) , MyFunc(b) , MyFunc(c)}
```

Creating Code Blocks

The simplest way to create a code block is to assign a literal code block to a polymorphic variable. For example:

```
cbString := {|| "just a string"}
cbIncr := {|nValue| nValue + 1}
cbCalc := {|nOne, nTwo|  SqRt(nOne) + SqRt(nTwo)}
```

You can also assign an expression that results in a code block, as in the following example:

```
cbLocate := oMyServer:GetLocate()
```

Since the only operator defined for use with the code block data type is the assignment operator (:=), code block expressions are limited to literals, function calls, and method calls, with one exception. It is possible to return a code block as the result of a macro expression.

In this example, the field *BlockField* contains a code block stored as a string. This field is used in a macro expression that returns the code block which, in turn, is saved in the variable *cbBlock*:

```
PROCEDURE Start()
     LOCAL cbBlock
     USE exp_file
     DO WHILE .NOT. EOF()
          // Compile code block
          cbBlock := &(BlockField)
          ...
          Eval(cbBlock)
          DBSkip()
     ENDDO
     DBCloseArea()
```

See Chapter 23, "Operators and Expressions," for more information.

Declaration

You can declare a variable in which you want to store a code block to the compiler using the LOCAL or GLOBAL statements (see Chapter 22, "Variables, Constants, and Declarations," for more information). To do this, you simply specify the variable name as part of the declaration statement:

```
LOCAL cbIncr
GLOBAL cbDisplay
```

Declaring a variable in this manner, however, does not create the code block. You must still assign a value to the variable with an assignment statement:

```
GLOBAL cbIncr
cbIncr := {|nValue| nValue + 1}
```

or as part of the declaration:

```
GLOBAL cbIncr := {|nValue| nValue + 1}
```

Strong Typing

Code blocks that you declare to the compiler or that you create without first declaring them are polymorphic variables, meaning that you can change their data type. Consider this example:

```
LOCAL cbIncr := {|nValue| nValue + 1}
Eval(cbIncr, 5)      // Evaluates 5 + 1
...
cbIncr := "New character value"
Eval(cbIncr, 5)      // Runtime error!
```

The LOCAL statement creates *cbIncr* as a polymorphic variable and assigns a code block to it. Eval() evaluates the code block, passing an argument of 5 (see the next section, Evaluating a Code Block). Then, the assignment statement changes the variable named *cbIncr* to a string value, destroying the code block that was previously stored. When you attempt to evaluate *cbIncr* a second time, a runtime error occurs because it is no longer a code block.

If you do not want to allow a variable name that you declare as a code block to change data type in your application, you can specify the CODEBLOCK data type (called strong typing) as part of the declaration statement. Doing this will trap as a compiler error any instance in which you misuse the variable name, including an attempt to change its data type or to use it where another data type is expected.

When the example given earlier is repeated with the variable *cbIncr* strongly typed as a code block, a compiler error occurs when you attempt to assign a string value to *cbIncr*:

```
LOCAL cbIncr := {|nValue| nValue + 1} AS CODEBLOCK
Eval(cbIncr, 5)                                    // Evaluates 5 + 1
...
cbIncr := "New character value"          // Compiler error!
Eval(cbIncr, 5)
```

Tip: Use the system defined constant NULL_CODEBLOCK to test for an uninitialized, typed code block. For example, the statement LOCAL cbIncr AS CODEBLOCK leaves *cbIncr* in an uninitialized state, and the test cbIncr = NULL_CODEBLOCK returns TRUE until you assign a value to *cbIncr*. Likewise, you can return a code block to its original, uninitialized state using an assignment statement, such as cbIncr := NULL_CODEBLOCK.

See Chapter 22, "Variables, Constants, and Declarations," in this guide for more information on the advantages of declaring and strongly typing variables.

Evaluating a Code Block

The examples in the previous section briefly introduced to the Eval() function used to evaluate a code block. This function evaluates the code block indicated as its first argument and passes subsequent arguments to the code block as parameters.

When the code block is evaluated, the expressions in the code block definition are evaluated in order from left to right, and the result of the last (or only) expression in the list is used as the code block return value:

```
cbIncr := {|nValue| nValue + 1}
? Eval(cbIncr, 1)                                    // Result: 2
```

There are several other built-in functions designed to execute code blocks:

Operation	Description
AEval()	Evaluate a code block for each element in an array
AEvalA()	Evaluate a code block for each element in an array and assign the result to the array element
AEvalOld()	Evaluate a code block for each element in an array, passing the array element number as a parameter
DBEval()	Evaluate a code block for each record in a work area
Eval()	Evaluate a code block
SEval()	Evaluate a code block for each byte in a string
SEvalA()	Evaluate a code block for each byte in a string and assign the result to the string byte
VODBEval()	Strongly typed DBEval()

For more information on any of these functions, refer to the online help system.

Variable Scoping in Code Blocks

In Chapter 22, "Variables, Constants, and Declarations," you were introduced to the concept of lexically scoped variables that you declare based on the lexical unit in which they will be used. In Visual Objects, a code block is considered to be a lexical unit.

Creating Variables

Even though declaration statements are not allowed in code block definitions, it is possible for you to create a new variable in a code block by making an assignment to a non-existent variable name. If you do this, the variable will be treated as local to the code block, which means its lifetime and visibility are limited to the code block:

```
FUNCTION One()
    LOCAL cbCalc AS CODEBLOCK
    cbCalc := {|nValue| nTemp := nValue + 1 , ;
                        SqRt(nTemp)}
    Eval(cbCalc, 100)
    ? nTemp                                    // Runtime error!
```

When you want to create variables within a code block, keep in mind that global variables and variables that are local to the creating entity are automatically visible to the code block. You can also have public and private variables that are visible to the code block.

Any variable that you want to create as local to the code block must have a name that does not conflict with other variable names. Otherwise, the assignment statement will assign to the existing variable, changing its value:

```
FUNCTION One()
    LOCAL nTemp := 2.3 AS FLOAT, cbCalc AS CODEBLOCK
    cbCalc := {|nValue| nTemp := nValue + 1 , SqRt(nTemp)}
    ? nTemp                                                // Result: 2.3
    Eval(cbCalc, 100)
    ? nTemp                                                // Result: 101
```

Exporting Local Variables

When you create a code block, you can access local variables defined in the creating entity within the code block definition without having to pass them as parameters (in other words, local variables are visible to the code block). This was illustrated with the last example in the previous section in which an assignment was made to a local variable within a code block.

Using this fact along with the fact that you can pass a code block as a parameter, you can export local variables:

```
FUNCTION One() EXPORT LOCAL
     LOCAL nVar := 10 AS INT, cbIncr AS CODEBLOCK
     cbIncr := {|nValue| nValue + nVar}

     ? NextFunc(cbIncr)                  // Result: 210


FUNCTION NextFunc(cbAddEmUp)
     RETURN Eval(cbAddEmUp, 200)
```

When the code block is evaluated in NextFunc(), *nVar*, which is local to function One(), becomes visible even though it is not passed directly as a parameter.

Note: The EXPORT LOCAL clause specified as part of the function definition is not required to export local variables using a code block; however, including this clause will make your application compile significantly faster.

Macros and Code Blocks

Code blocks bear a strong resemblance to macros (see The Macro Operator section in Chapter 23, "Operators and Expressions"), but with a significant difference. Macros are character strings, which are compiled *on the fly* at runtime and immediately executed. Code blocks, on the other hand, are compiled at compile time along with the rest of the application. For this reason code blocks are more efficient than macros, while offering similar flexibility.

The difference between code blocks and macros becomes especially important with declared variables. Variables that you declare at compile time are not visible within runtime macros, whereas you can access them freely in code blocks. This section discusses the interaction between code blocks and the macro operator.

Macro Expansion in Code Blocks

When a code block contains a macro, the macro is expanded each time the code block is evaluated, a technique known as *late evaluation*. Consider this use of DBSetFilter() in which the filter condition is specified as a macro expression:

```
DBSetFilter({|| &(cFilter)})
```

When you set a filter condition, the code block must be evaluated each time you move the record pointer in the active work area. In most cases, you would be safe in assuming that the value of *cFilter* remains constant for the duration of the filter; however, if you change *cFilter* before moving the record pointer, the filter condition will reflect this change.

Note: Runtime code blocks (discussed in the next section) use *early evaluation* if you specify a macro variable (omit the parentheses) and late evaluation if you specify a macro expression. With early evaluation, the macro is expanded at the time the code block is created, and the expanded value remains constant for all subsequent evaluations of the block. Compile-time code blocks (those created without using the macro operator) do not support early evaluation because they are created at compile time, not runtime.

Runtime Code Blocks

The macro compiler supports runtime compilation of code blocks, allowing you to evaluate code blocks that are stored (as strings) in database fields or entered by the user at runtime. Code blocks that you create using the macro operator are called *runtime code blocks*.

In this example, the field *BlockField* contains a code block stored as a string. This field is used in a macro expression that returns the code block. The code block is saved in the variable *cbBlock*, which is later evaluated with Eval():

```
PROCEDURE Start()
     LOCAL cbBlock
     USE exp_file
     DO WHILE .NOT. EOF()
          // Compile code block
          cbBlock := &(BlockField)
          ...
          Eval(cbBlock)
          DBSkip()
     ENDDO
     DBCloseArea()
```

Note: Because runtime code blocks are implemented as instances of the system-defined _CODEBLOCK class, their data type is object; therefore, you cannot save them to variables declared as code blocks (for example, LOCAL cbBlock AS CODEBLOCK in the above example will not work.) You can store a runtime code block either as a polymorphic variable (as in the above example) or AS OBJECT (or, more specifically, AS _CODEBLOCK) if you require strong typing.

Important! *Type() and ValType() return B, not O, when used on a runtime code block. UsualType(), however, properly distinguishes between the data type of a compile-time and a runtime code block. See the online help for more information on these functions.*

Chapter

27 Functions and Procedures

Functions and procedures are the basic elements for procedural programming. Unless you choose a pure OOP style, you will want to know how to define and use these elements in your application, which is the subject of this chapter.

Functions and procedures are nearly identical. The only differences are:

- Functions can return any value, but procedures always return NIL. This effectively excludes the use of procedures in expressions, limiting their use to program statements.
- Procedures have special purpose _INIT clauses for automatic execution at startup.
- Functions are declared to the compiler using the FUNCTION keyword and procedures using PROCEDURE.

For the purpose of brevity, the discussion in this chapter is limited to functions, but you can assume that everything mentioned applies equally to procedures with the exception of the points enumerated above. For additional information, see the FUNCTION and PROCEDURE entries in the online help system.

Note: _INIT1 procedures should be considered reserved by Visual Objects. Users should avoid using them, but use _INIT2 and _INIT3 procedures instead.

Defining

A function is a compiler entity. You define it using a FUNCTION declaration statement followed by variable declarations and the *function body*, statements that define what the function does:

```
FUNCTION Test()
    ? "This is the Test function."
```

Visibility

When you define a function in an application module or as part of a library, the function is available to the entire application or library. Library functions are also available to any application that includes the library in its search path.

You can limit the visibility of a function by using the STATIC keyword when you declare the function:

```
STATIC FUNCTION Test()
    ? "This is the Test function."
    ? "It is not visible to the entire application."
```

For an application, this limits the visibility of the function to the module in which it is declared. For libraries, it limits the visibility to the library, thereby hiding it from the application. Use STATIC FUNCTION to declare service functions not meant for public use.

Parameters and Return Values

Part of the function declaration statement is the list of parameters that the function expects to receive when invoked. You specify the function parameters as a comma-separated list in parentheses following the function identifier:

```
FUNCTION Test(p1, p2, p3)
```

Function parameters are declared as local variables in the function. When you call the function, the arguments that you pass are automatically assigned to the parameters.

An important part of the function body is its return value, specified with the RETURN statement:

```
FUNCTION Test(p1, p2, p3)
    RETURN p1 * p2 / p3
```

The return statement does two things: it passes control back to the calling routine and returns the specified value. You do not have to include a RETURN statement if you are not interested in getting a value back from the function—the next entity declaration is considered an implicit return.

You can specify the data type of the return value to detect improper use of the function at compile time:

```
FUNCTION Test(p1, p2, p3) AS FLOAT
    RETURN p1 * p2 / p3
```

Note: AS VOID is a special return value, indicating that the function does not return a value. If you use it, the function can only be invoked as a program statement, not as part of an expression, and the function can RETURN VOID, but no other value.

You can also specify the data types of the arguments (called strong typing) to enforce type checking of arguments at compile time:

```
FUNCTION Test(p1 AS FLOAT, p2 AS FLOAT, p3 AS INT) AS FLOAT
```

With strongly typed parameters, the AS keyword indicates that the argument must be passed by value when the function is called. Using REF instead of AS forces the argument to be passed by reference using the reference operator. This is also enforced at runtime:

```
FUNCTION Test(p1 REF FLOAT, p2 REF FLOAT, p3 REF INT) AS FLOAT
```

See Chapter 22, "Variables, Constants, and Declarations," in this guide for more information on strong typing and its advantages.

Calling Conventions

The manner in which you declare a function determines the calling convention that will be used when the function is invoked. You can also explicitly specify the calling convention with keywords that are part of the FUNCTION declaration statement.

CLIPPER

Functions that you declare with no data typing in the parameter list use the CLIPPER calling convention, by default. These declarations are equivalent:

```
FUNCTION Test(p1, p2, p3) AS FLOAT
FUNCTION Test(p1, p2, p3) AS FLOAT CLIPPER
```

The CLIPPER convention gives you a lot of flexibility when you call the function:

- You can omit any argument
- You can use PCount() to determine how many arguments are passed
- You can call the function in a macro expression
- You can pass any argument by reference or value

STRICT

Typing arguments forces STRICT calling convention. These two declarations are equivalent:

```
FUNCTION Test(p1 AS FLOAT, p2 AS FLOAT, p3 AS INT) ;
     AS FLOAT

FUNCTION Test(p1 AS FLOAT, p2 AS FLOAT, p3 AS INT) ;
     AS FLOAT STRICT
```

Using strong typing increases the performance and reliability of your application. It also takes away all of the CLIPPER flexibility mentioned above:

- You must specify all arguments
- PCount() is not applicable
- You cannot call the function in a macro expression
- You must pass AS arguments by value and REF arguments by reference

Other

PASCAL and CALLBACK are two additional calling conventions that you can specify for low-level interfacing with Windows.

Declarations

Variable declarations go at the top of a function before the function body. These define what variables are used by the function (besides its parameters), their lifetime, and visibility. The most common declaration statement for functions is LOCAL, which declares variables that are visible only within the function body and that retain their values only until the function returns.

```
FUNCTION Test(p1 AS FLOAT, p2 AS FLOAT, p3 AS INT) ;
     AS FLOAT STRICT
     LOCAL nResult := p1 * p2 / p3
     RETURN nResult
```

STATIC is a variant of the LOCAL declaration that extends the lifetime of a variable to the application.

Function Pointers

Another capability introduced with CA-Visual Objects 2.0 was function pointers. CA-Visual Objects 1.0 provided the possibility of manipulating function pointers in the following manner:

```
p := @MyFunc()
```

However, one restriction was that these pointers were only allowed to be passed to functions that expected pointers as parameters. There was no means of invoking the functions being addressed by these pointers.

The extension of pointers in Visual Objects is similar to that of the general typed pointers introduced in Chapter 21, "Data Types," of this guide. Function pointers are now typed also. They now possess semantic information, which the compiler can exploit. Moreover, these function pointers can also be "dereferenced." You can therefore, access the functions being addressed by these function pointers, for instance, by indirectly calling them.

Nonetheless, there are special cases in which programmers should be capable of indirectly invoking functions whose specifications are completely unknown by their current applications. Therefore, special constructs were introduced so that they could be used in such cases.

The following syntax can be used for declaring a function pointer:

```
[STATIC] [LOCAL|GLOBAL] <Variable> AS <Defined Function>()
```

The function pointer variable <Variable> defined is automatically initialized to address the function <Defined Function>.

Visual Objects provides three different constructs for invoking indirect function calls:

```
CALL( <Function Pointer Expression>,<Argument List> )
CCALL( <Function Pointer Expression>,<Argument List> )
PCALL( <Function Pointer Expression>,<Argument List> )
```

PCALL stands for PASCAL CALL and assumes that the <Function Pointer Expression> evaluates to an address of a function declared with the PASCAL calling convention.

CALL is a synonym for PCALL.

CCALL stands for C CALL and assumes that the <Function Pointer Expression> evaluates to an address of a function declared with the STRICT calling convention.

If the <Function Pointer Expression> evaluates to a typed function pointer, the compiler can exploit the semantic information to do argument and return value type checking as well as other checking. The ADAM system also has sufficient information to perform its automatic dependency management.

The compiler, on the other hand, in cases where <Function Pointer Expression> evaluates to an anonymous pointer, cannot exploit any semantic information about the pointer, and consequently performs no checks with regards to argument types, return values, or calling conventions. The programmer must comply with the following rule in this case. The indirectly called function is assumed to have STRICT calling convention which enables functions to be called with a variable number of parameters - and must return a value of type USUAL. Failing to comply with this rule can result in a phase drift error, which is of sporadic nature making it very difficult to track down in large applications.

The following example program illustrates the use of function pointers:

```
FUNCTION calla(a,b,c AS WORD) AS WORD PASCAL
     c := b*a
     QOut(c)
FUNCTION callc() AS WORD PASCAL
     QOut("Callc (without parameter)has been called")
FUNCTION MyFunc(a AS WORD, b AS WORD) AS WORD PASCAL
     LOCAL c AS WORD
     LOCAL p AS PTR
     LOCAL fp AS callc PTR
     p:=@calla()
     CALL(@calla(),a,b,c)
     CALL(fp)
FUNCTION MyFunc2(a, b  AS  WORD) AS WORD PASCAL
     QOut("We have really called MyFunc2")

FUNCTION MyFunc3(a AS  WORD) AS WORD PASCAL

FUNCTION Start
     LOCAL fp AS MyFunc PTR
     LOCAL fp3 AS MyFunc3 PTR
     LOCAL pp AS PTR
// the following statement results in an error because
// MyFunc and MyFunc3 are not compatible
//fp := @MyFunc3()  // Error !

// Note by assigning fp to pp we lose pointer type
// pp := fp
// pp is an anonymous pointer.  Compiler cannot do
// the necessary checking; however since pp is set
// to fp which points to a PASCAL function, we are
// not conforming to the above rule in the following
// statement.  The statement might result to a phase
// error.
// CALL(pp,7,8)
// Note the automatic initialization of fp
     CALL(fp,7,8)
     fp := @MyFunc2()
     CALL(fp,3,4)
     CALL(fp,6,5)
// CALL(@MyFunc(),3,4)
```

The Function Body

The function body comes after all variable declarations and defines what the function does, including its return value:

```
FUNCTION Test(p1 AS FLOAT, p2 AS FLOAT, p3 AS INT) ;
     AS FLOAT STRICT
     LOCAL nDefaultDivisor
     nDefaultDivisor := 1
     p1 := Abs(p1)
     p2 := Abs(p2)
     p3 := IF(p3 == 0, nDefaultDivisor, p3)
     RETURN p1 * p2 / p3
```

You can include any executable statement within the context of the function body, including conditional and looping constructs, commands, function calls, object instantiations, and method invocations.

Calling

You call a function by using its identifier, followed by a pair of parentheses enclosing its arguments. The parentheses are required when you call a function, regardless of whether you pass arguments.

If you are not concerned with the return value (or if the function returns VOID), you can call the function as a stand alone program statement, but more often you will invoke it as part of an expression that uses its return value. If the function is defined using the CLIPPER calling convention, you can call the function as part of a macro expression, but other calling conventions do not allow this.

In any case, the function must be visible to the module from which it is called, or the compiler will return an error.

Default Parameters

Visual Objects now supports default values for parameters. With this new feature, PASCAL and STRICT functions are now more flexible. The following function declaration is now valid:

```
FUNC MyFunc(a:=5 AS INT, b:=8 AS DWORD) AS DWORD PASCAL
```

When calling MyFunc(), parameters may be skipped or omitted. In such cases the compiler automatically knows the default parameter to be used for the missing parameters and thus prepares the call with the appropriate parameters.

The following example program illustrates the use of default parameters:

```
FUNCTION Start
    ? "Calling x()"
    x()
    ? "Calling x(,)"
    x(,)
    ? "Calling x(2,)"
    x(2,)
    ? "Calling x( , 2, )"
    x(,2,)
    ? "Calling x( , ,2)"
    x( , ,2)

FUNCTION x(a:=3 AS INT, b:=5 AS INT, d:=9 AS INT);
    AS INT PASCAL
    ? "In x: value of a: "
    ? a
    ? "In x: value of b: "
    ? b
```

```
? "In x: value of d: "
? d
```

Functions with Variable Number of Parameters

Visual Objects provides a mechanism for defining functions which can accept a variable number of parameters. Such functions are declared to the compiler by using an ellipses (...) as the last parameter in the parameter list of the function declaration.

These functions must comply with the following rule. They must adhere to the STRICT calling convention enabling functions to be called with a variable number of parameters. The functions also cannot have a return value (i.e. return parameter is of type VOID). These functions must have at least one fixed parameter (a parameter before the ellipses).

Visual Objects provides two special functions for accessing the variable parameters within functions defined to access a variable parameter list. These special functions are: _GetFirstParam() and _GetNextParam(). As their names suggest, they are used for accessing the first and remaining parameters making up the variable parameter list respectively.

_GetFirstParam() is declared as follows:

```
FUNC _GetFirstParam  (         ;
                     pptrStart REF PTR,   ;
                     ptrFirst             AS PTR,    ;
                     dwTypeFirst          AS DWORD,;
                     dwType               AS DWORD   ;
                ) AS USUAL PASCAL
FUNC _GetNextParam   (         ;
                     pptrStart REF PTR,   ;
                     dwType               AS DWORD   ;
                ) AS USUAL PASCAL
```

pptrStart references the anonymous pointer variable local to the calling function which will serve as an address to the list of variable parameters.

ptrFirst is the address of a fixed parameter variable in the parameter list of the calling function.

dwTypeFirst specifies the type of the above fixed parameter.

*dwType s*pecifies the type of the actual parameter to be accessed.

The following example illustrates the implementation of a function accepting a variable number of parameters:

```
FUNCTION CFunc(dummy:=1 AS INT, dwNumber AS INT, ... );
     AS VOID STRICT
LOCAL   xParam AS USUAL
LOCAL   i      AS DWORD
LOCAL   pParam AS PTR
```

```
LOCAL   cString     AS STRING
xParam := _GetFirstParam(@pParam, @dwNumber, INT,;
 STRING)
? dwNumber
? xParam

FOR i := 2 UPTO dwNumber
     xParam := _GetNextParam(@pParam, STRING)
     ? xParam
NEXT
RETURN
FUNCTION Start
     CFunc(,1, "How")
     CFunc(,2, "How ", "are ")
     CFunc(,3, "How ", "are ", "you ")
```

Arguments

On the calling side, arguments correspond to parameters. You specify arguments as expressions in a comma-separated list:

```
x := Test(5, 10, 2)
```

If the function is defined using the CLIPPER calling convention, no argument checking is performed at compile time. You can pass any value to the function and even omit arguments—errors will not show up until the function is executed at runtime.

For other calling conventions, the compiler checks the data types of the arguments to make sure they match the function declaration, checks to make sure no arguments are missing, and checks to make sure that the arguments are passed using the correct convention (either by value or by reference). If errors are detected, the compiler will inform you. You can be assured that unexpected errors will not show up at runtime.

Passing by Value

Passing by value means that the argument is evaluated and its value is copied to the receiving parameter. Changes to the receiving parameter are local to the called function and lost when the function returns.

Any argument corresponding to a parameter that is not strongly typed is passed by value as the default, including arrays and objects.

```
FUNCTION Test(p1, p2, p3)
     p1 := p2 * p3
     RETURN p1

FUNCTION CallTest()
     LOCAL nTestValue, nArg1, nArg2, nArg3

     nArg1 := 0
     nArg2 := 10
     nArg3 := 5
```

```
nTestValue := Test(nArg1, nArg2, nArg3)
? nTestValue                                // Returns 50
? nArg1                                     // Returns 0
```

Arguments corresponding to strongly typed parameters must be passed by value if the data type is declared with the AS keyword. Any attempt to pass AS parameters by reference results in a compiler error.

Passing by Reference

Passing by reference means that a reference to the value of the argument is passed instead of a copy of the value. The receiving parameter refers to the same location in memory as the argument. If the called routine changes the value of the receiving parameter, it also changes the argument passed from the calling routine.

Other than field variables, any argument corresponding to a parameter that is not strongly typed can be passed by reference if prefaced by the reference operator (@).

```
FUNCTION Test(p1, p2, p3)
    p1 := p2 * p3
    RETURN p1

FUNCTION CallTest()
    LOCAL nArg1, nArg2, nArg3

    nArg1 := 0
    nArg2 := 10
    nArg3 := 5

    Test(@nArg1, nArg2, nArg3)
    ? nArg1                             // Returns 50
```

The @ is required for arguments corresponding to strongly typed parameters declared with the REF keyword.

```
FUNCTION Test(p1 REF INT, p2 AS INT, p3 AS INT)
    p1 := p2 * p3
    RETURN p1

FUNCTION CallTest()
    LOCAL nArg1, nArg2, nArg3

    nArg1 := 0
    nArg2 := 10
    nArg3 := 5

    Test(@nArg1, nArg2, nArg3)
    ? nArg1                             // Returns 50

    Test(nArg1, nArg2, nArg3) // Compiler error!
```

Passing Arrays and Objects

Arrays and objects are a little tricky because, although they can be passed by value or reference like other arguments, they are treated as references.

If you pass an array or object by value, you are passing a reference. The local parameter is a reference to the same array or object, so that changes to the components (elements or instance variables) are automatically reflected in the original argument. Wholesale changes, however, to the entire array or object, get lost because they create yet another reference.

When you pass by reference, wholesale changes are not lost. They are reflected upon return. See Chapter 24, "Arrays" for an example.

Recursion

Functions can be *recursive* (a function can call itself). Every recursive function should have some test condition that prevents the function from calling itself indefinitely; otherwise, the function will result in a stack overflow.

This recursive function performs an integer division between two positive numbers:

```
FUNCTION IntegerDivide(a AS INT, b AS INT) AS INT
     STATIC LOCAL iResult := 0 AS INT
     LOCAL iRetValue := 0 AS INT

     IF a < b
          // Stops further recursive calls or returns
          // to main caller if this is the first call
          RETURN iResult
     ELSE
          a -= b
          ++iResult
          iRetValue := IntegerDivide(a, b)
     ENDIF

     iResult := 0                                  // Resets for next call
     RETURN iRetValue                              // Returns to main caller
```

Argument Checking

When parameters are strongly typed, argument checking is done at compile time. This is the fastest, easiest way to do argument checking. You know for a fact at runtime that all arguments are present, typed correctly, and passed in the correct way.

With untyped parameters, you may want to put some type checking into the function code. Things you can do are:

- Use PCount() to detect missing parameters within the list, but not those left of the end.
- Supply a default, when appropriate, for a missing argument by testing the parameter for a NIL value. You can use the Default() function for this purpose:

```
Default(@p1, 1)
```

- Use UsualType() to make sure proper data types are passed:

```
IF UsualType(p1) = INT
    RETURN FALSE
ENDIF
```

This may help to prevent certain improper uses of the function, but there may be errors that you cannot detect and these will not show up until runtime.

Appendix

A

RDD Specifics

Specifications

The following table outlines the specifications for all of the RDDs supplied with Visual Objects. Some of the limitations specified may be subject to further restrictions, such as available RAM or disk space. All numbers are specified in bytes, unless otherwise indicated.

The specifications for DBFCDX and DBFMDX are for the corresponding native products, FoxPro (versions 2.0 and above) and dBASE IV (version 2.0), respectively. In all cases, the supplied drivers create and maintain database and index files that are compatible with the native product. (Refer to "Using DBF Files" in this guide for information on interoperability with applications written using the native product.)

Important! *Some of the limitations may not be strictly enforced by the RDD. All limitations, however, are genuine and should be observed. Otherwise, backward compatibility cannot be guaranteed.*

Specification	DBFNTX	DBFCDX	DBFMDX
Database file extension	.DBF	.DBF	.DBF
Memo file extension	.DBT	.FPT	.DBT
Maximum record length	65,535	65,535	4000
Maximum file sizes:			
.DBF (default)	1 billion	2 billion	2 billion
.DBF (ANSI Character Set)())	2 billion records	n/a	n/a
.DBT	32 MB	n/a	32 MB
.FPT or .DBV (using DBFBLOB driver)	4.2 billion	4.2 billion	4.2 billion
Maximum records (default)	500,000	1 billion	1 billion
Maximum records (NewIndexLock() or IndexHPLock())	2 billion	n/a	n/a
Formula for computing	(Max file size - Header() - 1)/RecSize()		
Maximum number of fields	1024	1024	255
Maximum length of:			
date field	8	8	8
float field	n/a	n/a	19
logic field	1	1	1
memo field	64 KB	no limit	64 KB
numeric field	19	19	19
string field	64 KB	64 KB	254
Default Index file extension	.NTX	.CDX	.MDX
Number of orders per index file	1	no limit	47
Number of index files per work area:			
Visual Objects application	15	15	15
Native product application	15	15	15

Specification	DBFNTX	DBFCDX	DBFMDX
INDEX/DBSetOrderCondition() features supported:			
for condition	Yes	Yes	Yes
scope and while condition	Yes	Yes	No
unique keys	Yes	Yes	Yes
descending keys	Yes	Yes	Yes
evaluation of code block at intervals	Yes	Yes	No
create order without clearing order list	No	Yes	No
create order using current controlling order	No	Yes	No
custom built order	No	Yes	No
optimization	No	Yes	No
Maximum key expression length	256	256	220
Maximum length of key expression result	256	240	100
Maximum for condition length	256	256	220
Data types allowed for key expressions:			
date	Yes	Yes	Yes
float	n/a	n/a	Yes
logic	Yes	Yes	No
memo	No	No	No
numeric	Yes	Yes	Yes
string	Yes	Yes	Yes
Production/structural index file supported	No	Yes	Yes
Implicit record unlocking in single lock mode	Yes	Yes	Yes
Multiple record locks supported	Yes	Yes	Yes
Filter optimization supported	No	Yes	No

Record Locking Offsets

A note regarding the locking offset used for the DBFNTX driver is in order, because it limits the .DBF file size that you can accurately maintain if your application relies on record or file locking.

The DBFNTX driver supplied with Visual Objects is designed to be compatible with CA-Clipper and must, therefore, respect the locking offset defined in the .NTX header record. The actual value will depend on whether or not your CA-Clipper application was linked with NTXLOCK2.OBJ.

By default, CA-Clipper applications are not linked with this file and use a locking offset of 1 billion. For record locking, the locking address is 1 billion + the record number (locked for a length of 1 byte). For file locking, the locking address is 1 billion (locked for a length of 1 billion bytes). This locking scheme effectively limits the maximum size of .DBF files to 1 billion bytes. Anything larger cannot be locked appropriately using this locking offset.

Linking NTXLOCK2.OBJ changes the locking offset to a much larger value (4 GB), which removes .DBF file size limitations imposed by the previously discussed locking scheme. The record locking address for NTXLOCK2.OBJ is 4 GB - the record number (locked for a length of 1 byte), and the file locking address is 4 GB (locked for a length of 2 billion bytes).

The limitations for database file sizes created using the Visual Objects DBFNTX driver will depend on the locking offset settings as defined by the NewIndexLock() function. By default, the 1 billion byte offset will be used, imposing the 1 billion byte file size limitation explained earlier. To remove this limitation on database file size, you can set both flags to TRUE.

The DBFBLOB Driver

The DBFBLOB RDD supplied with Visual Objects is designed to give you an alternative mechanism for storing and retrieving memo fields and to give you direct control over managing the file used to store the data (called a BLOB file, for binary large object). The driver provides an efficient and flexible mechanism for managing the data, thereby superseding .DBT/.FPT file managers that are the industry standard.

The DBFBLOB driver features:

- A 4.2 GB file size limitation
- A 1-byte minimum block size limitation
- An efficient technique for recycling file space
- The ability of memo fields to store any usual data type (other than object and code block)
- Extensions to the Visual Objects language (BLOB functions) for file and field management

Using DBFBLOB as an Inherited Driver

If you are currently using a driver that supports .DBT style memo fields (such as DBFNTX or DBFMDX), you can inherit from the DBFBLOB driver. For example:

```
USE customer VIA "DBFMDX" INHERIT FROM {"DBFBLOB"}
```

Using this technique, you can have the standard database and index operations controlled by the main RDD and the memo file operations controlled by the DBFBLOB RDD.

Note: When the DBFBLOB driver is inherited, the block size defaults to 1 (the most efficient setting) and the memo file extension defaults to .DBV. The block size and memo file extension settings can be controlled using the RDDInfo() _SET_MEMOBLOCKSIZE and _SET_MEMOEXT constants, respectively.

Converting Your Data

If you have data that is currently in the .DBF/.DBT file format and want to convert the .DBT file to a .DBV file for use with DBFBLOB as an inherited driver, copy the file using the INHERIT FROM "DBFBLOB" clause, as in:

```
USE <cOriginalDBF> VIA "DBFNTX"  // or other driver
COPY TO <cNewDBF> VIA "DBFNTX" ;
     INHERIT FROM {"DBFBLOB"}
```

In addition to copying the .DBF file, this will copy the data in the associated .DBT memo file to a .DBV BLOB file. Then, to use the .DBF/.DBV file combination:

```
USE <cNewDBF> VIA "DBFNTX" INHERIT FROM {"DBFBLOB"}
```

Using DBFBLOB Via DBFCDX

If you are using the DBFCDX RDD, you will get automatic access to the DBFBLOB driver because DBFCDX inherits from it. Using the DBFBLOB driver in this manner lets you maintain .FPT files that are 100% compatible with FoxPro for a string data types. Furthermore, all report writers and file viewers that recognize the .FPT file format will be able to read from and write to the file. A Visual Objects application concurrently accessing a file with any of these other applications will continue recycling space even though the other application may not.

Note: To maintain this level of compatibility and interoperability, the default memo file extension must be .FPT and the default block size must be 32 (or greater). These are the default settings for the RDDInfo() _SET_MEMOEXT and _SET_MEMOBLOCKSIZE constants when you use the DBFCDX driver.

Important! *Because the DBFCDX driver automatically inherits from the DBFBLOB driver, do not try to use the DBFBLOB driver as an inherited driver in conjunction with the DBFCDX driver. It is unnecessary and could produce unexpected results.*

Converting Your Data

If you have data that is currently in the .DBF/.DBT file format and want to convert it for use with the DBFCDX driver, simply copy the file VIA "DBFCDX," as in:

```
USE <cOriginalDBF> VIA "DBFNTX"  // or other driver
COPY TO <cNewDBF> VIA "DBFCDX"
```

In addition to copying the .DBF file, this will copy the data in the associated .DBT memo file to an .FPT BLOB file. Then, to use the .DBF/.FPT file combination:

```
USE <cNewDBF> VIA "DBFCDX"
```

Appendix

B

Reserved Words

This section lists all the reserved words in Visual Objects; they are presented in alphabetical order.

__ENTITY__
__INLINE
__LINE__
__MODULE__
_CAST
_CO
_DLL
_INIT1
_INIT2
_INIT3
_NC
_WINCALL
AADD
ABS
ACCESS
ALLTRIM
ARRAY
AS
ASC
ASPEN
ASSIGN
AT
BEGIN
BIN2I
BIN2L
BIN2W
BOF
BREAK
BUFFER
BYTE
CALLBACK
CASE
CDECL
CDOW
CHR
CLASS
CLIPPER
CMONTH
CODEBLOCK
CTOD
DATE
DAY
DESCEND
DECLARE
DEFINE
DIM
DO
DOW
DOWNTO
DTOC
DTOS
DWORD
ELSE
ELSEIF
EMPTY
END
ENDCASE
ENDDO
ENDIF
ENDTEXT
EOF
EVAL
EXIT
EXP
EXPORT
EXTERN
FALSE
FIELD
FIELDNAME
FIELDPOS
FLOAT
FOR
FUNCTION
GLOBAL
HARDCR
HIDDEN
I2BIN
IF
IFDEF
IFNDEF
IN
INHERIT
INSTANCE
INT
INTEGER
IS
ISALPHA
ISDIGIT
ISLOWER
ISUPPER
L2BIN
LEFT
LEN
LOCAL
LOG
LOGIC
LONGINT
LOOP
LOWER
LTRIM
MAX
MEMBER
MEMVAR
METHOD
MIN
MONTH
NAME
NEXT
NIL
OBJECT
OF
OTHERWISE
PARAMETERS
PASCAL
PRIVATE
PROCEDURE
PROTECT
PSZ
PTR
PUBLIC
RAT
REAL4
REAL8
RECNO
RECOVER
REF
REPLICATE
RESOURCE
RETURN
RIGHT
ROUND
SECONDS
SELF
SEQUENCE
SHORTINT
SOUNDEX
SPACE
SQRT
STATIC
STATUS

STEP	SYMBOL	TRUE	VOID
STR	TEXT	TYPE	WHILE
STRICT	TEXTBLOCK	UPPER	WITH
STRING	TIME	UPTO	WORD
STRUCTURE	TO	USING	YEAR
STUFF	TODAY	USUAL	
SUBSTR	TONE	VAL	
SUPER	TRIM	VALTYPE	

Index

.

/

:

;

@

[

^

_

{

|

+

<

=

>

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

Z