Paradox for Windows

Version 1.0

Learning ObjectPAL™

Borland International, Inc. 1800 Green Hills Road
P.O. Box 660001, Scotts Valley, CA 95067-0001, USA

Copyright ©1992 by Borland International, Inc. Portions copyright 1985 by Borland International, Inc. All rights reserved. Borland and Paradox are trademarks of Borland International. Microsoft and MS are trademarks of Microsoft Corporation. Windows, as used in this manual, refers to Microsoft's implementation of a windows system.

10 9 8 7 6 5 4 3 2 1 R1

CONTENTS

Chapter 1
Introduction 1

Before you use ObjectPAL 1
Note to PAL programmers 2
How to use this manual 2
Printing conventions 3

Chapter 2
ObjectPAL basics 5

Programming in ObjectPAL 5
What is ObjectPAL? 6
An extension of Paradox 6
Object-based 7
Event-driven 8
Modular 10
ObjectPAL for programmers 11
Language features 12
Control features 12
Objects in ObjectPAL 13

Chapter 3
The ObjectPAL environment 15

The ObjectPAL Editor 15
The Methods dialog box 15
The ObjectPAL Editor window 17
Editing in an Editor window 17
The Language menu 18
Keywords 19
The Types and Methods dialog box . . 19
The Display Objects and Properties dialog box 21
The Constants dialog box 21
Creating a source code report 22
Delivering forms to users 22
The Properties menu 22
The ObjectPAL Debugger 24
On to programming 24

Chapter 4
Programming a button 25

Built-in methods 26
Changing the default behavior 26
Attaching your own code 28
How it works 29
Summary 29

Chapter 5
Initiating and responding to actions 31

Stages in writing ObjectPAL applications . 31
Creating the form 32
Programming a button to take an action . 34
How it works 35
Responding to an action 36
How it works 38
Actions and properties 39
How it works 41
Summary 42

Chapter 6
Input and output 43

A quick way to get user input 43
How it works 45
Searching for values 46
How it works 48
Inserting a record and generating a unique key value 49
How it works 51
Printing a report 52
How it works 54
Summary 54

Chapter 7
Validating data entry 55
Creating a multi-table form 55
Using built-in validity checks 58
Adding validity checks with ObjectPAL . 59
How it works 59
Supplying values 60
How it works 62
Handling key violations 63
Single-record forms 63
How it works 65
Multi-table forms 66
The closer-is-better principle 67
How it works 68
Summary 68

Chapter 8
Controlling another form 69
Designing a dialog box 69
Managing a dialog box 71
How it works 73
Summary 75

Chapter 9
Working outside the data model 77
What is a TCursor? 77
Using a TCursor 78
How it works 79
Summary 81

Chapter 10
Where do I go from here? 83

Index 85

EXAMPLES

4-1 Hello, world! application 26
4-2 Opening an ObjectPAL Editor window 26
4-3 Attaching your own code 28
5-1 Creating the *NewCust* form 33
5-2 Attaching code to a button 34
5-3 Tab default behavior 36
5-4 Controlling tab order 37
5-5 Responding to an action 40
6-1 Using **view** to display a dialog box . 43
6-2 Searching based on user input . . . 46
6-3 Inserting a new record 50
6-4 Printing a pre-designed report . . . 52
7-1 Creating a multi-table form 55
7-2 Built-in validity checking 58
7-3 ObjectPAL validity checking 59
7-4 Performing calculations 61
7-5 A form as manager 64
7-6 Handling key violations in a multi-table form 67
8-1 Designing a dialog box 69
8-2 Setting a form's properties 70
8-3 Managing a dialog box 72
9-1 Using a TCursor 78

TABLES

1-1 Printing conventions 3
3-1 ObjectPAL Editor actions 18
5-1 ObjectPAL Beginner-level action constants 35

FIGURES

2-1 Using the Object Tree 8
3-1 The Methods dialog box 16
3-2 The ObjectPAL Editor 17
3-3 The Language pop-up menu 19
3-4 The Types and Methods dialog box . 20
3-5 The Display Objects and Properties dialog box 21
3-6 The Constants dialog box 22
3-7 ObjectPAL Level setting 23
4-1 Hello, world! 25
4-2 Inspect the object, select a method, open an Editor window 27
5-1 Properties of the field object *Name* . . 40
7-1 The *Orders* form 61
8-1 *Orders* as the calling form 72
9-1 Attaching code to the *Qty* field object in *LINEITEM* 78
9-2 Record displayed during TCursor **locate** 80
10-1 Using the Help system in the example applications 84

CHAPTER 1

Introduction

ObjectPAL™ is the integrated programming language for Paradox for Windows. You can use ObjectPAL to add new features to a Paradox application—features that you cannot add interactively.

If you are a new programmer, you will find that you can easily spruce up an interactive application with ObjectPAL. For instance,

- You can add buttons that perform frequently repeated actions.
- If your database contains a large volume of data, you can build a dialog box that helps users get to the records they want when something more sophisticated than a standard search is required.
- The success of your application may depend on a robust data-entry module. If so, you can create an editing routine that watches what users enter and responds to incorrect entries.
- On the frivolous side, if you're looking for a way to add a little pizzazz, you can create animation effects with ObjectPAL.

If you've never programmed before and aren't sure you want to learn ObjectPAL, glance through this manual to get an idea of what you can do with ObjectPAL. Allow yourself to be intrigued.

Before you use ObjectPAL

ObjectPAL and Paradox are tightly integrated. Therefore, the more you know about Paradox, the more you can take advantage of it in your ObjectPAL programs.

Important To get the most out of this manual, you should first read the *User's Guide* and get some experience using Paradox interactively. You should understand how to

- Create tables, forms, and reports
- Use the SpeedBar to place design objects

- Work interactively with table frames and multi-record objects
- Name objects
- Inspect objects and set object properties
- Set and change your working directory
- Construct queries using query by example (QBE)
- Assign aliases
- Sort tables

When you're familiar with these actions and concepts, it's much easier to learn ObjectPAL.

If you haven't installed and configured Paradox, see *Getting Started* for instructions.

The best way to understand ObjectPAL is to use it.

This manual doesn't cover all aspects of programming, but it introduces programming concepts as they apply to ObjectPAL. You should work through the examples and try things on your own. The best way to understand ObjectPAL is to use it. ObjectPAL often provides more than one way to accomplish a task; experiment to find the way that works best for you.

Note to PAL programmers

If you're an experienced PAL programmer, you'll find ObjectPAL different in many respects. However, the things you've learned about database programming—things like working with data in tables, records, and fields; using query by example; and controlling access to data—still apply. Appendix A in the *ObjectPAL Developer's Guide* summarizes the most important differences between PAL and ObjectPAL.

How to use this manual

This book is designed to help users who have little or no programming experience get up and running in ObjectPAL with minimal trouble. If you are an experienced programmer, this book serves as a fast-paced introduction to the ObjectPAL language. As you work through the examples, you learn to create Paradox for Windows applications with buttons, dialog boxes, validity checks, and key violation checks.

This manual asumes that you are already familiar with interactive Paradox. In this book, "interactive Paradox" means the set of things you can accomplish in Paradox without ObjectPAL.

You should read *Getting Started* if you are new to Paradox and read the *User's Guide* to fully understand interactive Paradox tools. You should complete the examples in the order they appear. Later you can refer back to this manual for specific code. After mastering the basic concepts in this manual, you can move on to the *ObjectPAL Developer's Guide* and explore the full power of ObjectPAL.

Printing conventions

In text (as opposed to code examples), this manual refers to methods and procedures by name only; it does not give the full syntax. For example, suppose the text mentions the **attach** method defined for the Table type. That's a reference to the method whose complete syntax is

attach (const ***tableName*** String) Logical

To see the complete syntax for every ObjectPAL method and procedure, refer to the *ObjectPAL Reference,* the online help, or the *Quick Reference.*

Table 1-1 lists the printing conventions used in this book.

Table 1-1 Printing conventions

Convention	Applies to	Examples
Bold	Method names and messages displayed by Paradox	**insertRecord**, Paradox displays the message **Index error on key field**
Italic	Names of Paradox objects, glossary terms, variables, emphasized words	*Answer* table, *searchButton*, *searchVal*
ALL CAPS	DOS files and directories, reserved words, operators, types of queries	PARADOX.EXE, CREATE, C:\WINDOWS
Initial Caps	Applications, fields, menu commands	Sample application, Price field, Form | View Data command
Keycap font	Keys on your computer's keyboard	*F1*, *Enter*
`Monospaced font`	Code examples, ObjectPAL code	`myTable.open("sites.db")`
Type-in font	Text that you type in	**Jan - Jun, 7/20/92**

CHAPTER 2

ObjectPAL basics

This chapter introduces ObjectPAL by giving you a conceptual overview of the benefits and structure of the language.

Programming in ObjectPAL

Programming in ObjectPAL is in some ways similar to programming in other languages and in other ways different. ObjectPAL is similar to traditional languages because it uses variables, provides control structures like **if...then...else**, **for** loops, and **while** loops, performs calculations, and gives you a way to create functions (in ObjectPAL, they're called *methods* and *procedures*).

ObjectPAL differs from traditional languages because it is object-based. When you use a traditional language, programming is an all-or-nothing proposition: either you take control of the application from beginning to end, or you don't program at all. With ObjectPAL, however, you need not face such a daunting task. Because ObjectPAL centers on objects, you can program as many or as few objects as you want.

The objects you write ObjectPAL code for are the objects you've been working with all along. Do you need to have Paradox check a value that was just entered in a field and beep if that value is wrong? Programming this function is simple; you change the built-in code that runs when the field's value changes. The operation takes only a little time to learn, and it's easy to use in other situations once you learn how. Naturally, not everything you want to do with ObjectPAL is so simple, but you get to decide how much or how little programming you do.

What is ObjectPAL?

Formally, ObjectPAL is a high-level, event-driven, object-based, visual programming language. You can use ObjectPAL to create a completely customized application, one with entirely new buttons, menus, dialog boxes, prompts, warnings, and help. You can create a user interface for a database application, or you can use ObjectPAL to create an application that has nothing to do with databases.

An extension of Paradox

Formal definitions and ambitious goals aside, a good way to get to know ObjectPAL is to think of it as a tool that extends the power of interactive Paradox. If you think of ObjectPAL as an extension of Paradox, you can think of ways to use ObjectPAL to perform tasks that would be awkward, difficult, time-consuming, or impossible to perform without it.

Automate repetitive tasks.

Suppose that you want to create a unique but sequenced ID number every time a user in a network setting opens a new invoice record. If the last invoice created has the ID number 1203, for example, you would want the next invoice number to be 1204. Without ObjectPAL, you could create a single-record, single-field table in a shared data directory and store the most recently used invoice ID number in that field. Users could be instructed to open that table, start editing, lock the record, change the ID number to the next number in the sequence, unlock the record, leave Edit mode, close the table, return to the invoice table, and enter the new ID number. With ObjectPAL, you could have your application perform these steps automatically whenever a user creates and posts a new invoice. (An example in Chapter 6 shows how to do this.)

Correct field format.

Performing detailed changes on fields can be difficult when you use queries interactively. For example, if a phone-number field was entered or imported to a table without parentheses around the area code (or a dash that separated the area code from the rest of the number), correcting the format with a query would be impossible. Your alternative would be to fix one record at a time. With ObjectPAL, however, you could write a routine that examines the Phone field of each record and changes any fields that weren't entered correctly.

Protect data.

At times, you'll want to warn users when they are about to do something potentially damaging to the database (such as changing a key field). Although the structure of your database (how you link the tables, how you enforce referential integrity, and the type of field validation you define) can go a long way toward protecting the integrity and validity of data, sometimes you'll need more specialized protection, which is impossible to do without ObjectPAL.

The power of Paradox

Keep in mind, however, that many of the things you need to do with a database you can do with Paradox interactively. If you are turning to ObjectPAL only as a means of solving a particularly thorny data-handling problem, you should first make sure that you cannot use interactive Paradox to solve that problem. Even many advanced users of interactive Paradox have only begun to tap the power of queries and calculations. Remember, too, that data validation, table lookups, choice lists, and many other powerful user-interface features are available in interactive Paradox.

Object-based

ObjectPAL works with objects—the things you create and work with when you design forms and reports, including fields, lines, ellipses, boxes, and table frames.

Properties

The first thing to remember about objects is something that you already know: *objects have properties*. When you create an object, you create it with properties that define the appearance and behavior of the object. The properties of a box, for example, include size, position, color, and frame. Using ObjectPAL, you can create or change all the properties that you use in interactive Paradox. For example, you can create a big blue box interactively and then use ObjectPAL to change it to a small red box.

Context

The next thing to remember about objects is *objects exist in a context*.

The context of a given object is defined by the objects that contain it. This feature of ObjectPAL gives advanced programmers great flexibility and power. As a beginning ObjectPAL programmer, all you have to remember is that the form contains all other objects. When you place objects in a form, you are giving those objects a context.

Visual programming

This process of placing objects is called *visual programming*, because the form lets you *see* the user interface of your application as you program. To create an ObjectPAL application, you place objects—for example, fields, choice lists, drop-down edit lists, buttons, and icons—in a form and set their properties. Once you're happy with the look and feel of the form, you use ObjectPAL to change the behavior of only those objects whose default behavior does not suit your needs.

This work is very different from the work you would have to do to create an application in a non-visual environment. In many other languages, you create the user interface by writing code in some kind of text editor. To run the application, you must compile the code, debug it, run it again, and so on until it works. Every time you blindly change the position of an element, you must access the code, a procedure that potentially creates more bugs.

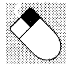
Object Trees

You can see the relationships of the objects to one another not only in the form but also in an Object Tree. Object Trees provide a conceptual view of the relationships among objects.

Object Trees also let you access the code for an object. When an Object Tree for a form is open, you can see all the objects in the form and all the objects inside those objects. The basic steps to display an object tree for a form are

1. Select the page by clicking anywhere on the page (outside of an object).
2. Press *Esc* to select the form.

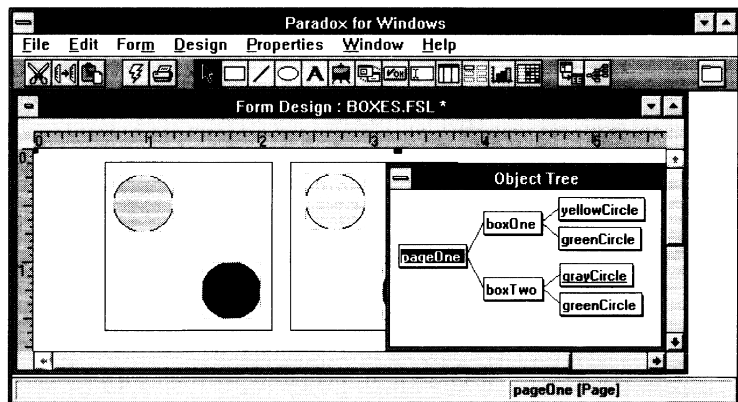
3. Open an Object Tree by clicking the Object Tree button (or choose Form | Object Tree).



To inspect an object in the Object Tree, right-click the object's name in an Object Tree. An underlined object name indicates that the object has custom code attached to it.

For example, suppose a form contains one page, and that page contains two boxes, and each box contains two circles. Figure 2-1 shows the form and its corresponding Object Tree.

Figure 2-1 Using the Object Tree



The Object Tree diagrams the visual, spatial relationships between objects in a form

Event-driven

In Windows, no program has absolute, permanent control. No program can ever assume that it has first-hand knowledge of the specific hardware of a system or presume to know about other programs or what those programs are doing. Any application can be stopped at practically any time—whenever the user clicks on another application.

This interface is possible for two reasons. First, every application is totally dependent on Windows to provide processing time, monitor space, and other resources. Ultimately, Windows controls every Windows application. Second, the nature of Windows encourages (and sometimes forces) Windows applications to be *event-driven.*

The event-driven interface

An event-driven interface is one that responds only to specific system or user actions, such as mouse moves. The application takes control of the system (through Windows) long enough to respond to an event; the application then waits for the next event. To extend this concept further, you can think of every object as being a small application.

Accordingly, a more elaborate description of objects takes into account their event-driven nature: objects have a context that determines their relationship to other objects, a set of properties that determine their characteristics, and built-in methods that determine their behavior in response to events.

Built-in behavior

When you draw a rectangle in a form, you are not merely drawing but also taking the first step in programming. The box is an object with properties that exist in the context of a form. But the box also has a behavior, because Paradox built this box to respond to events.

By default, the box's response to most events is nothing. You use ObjectPAL to tell the box to do something other than, or more than, the default response when a certain event occurs. For example, you can change the color of the box when the user moves inside the borders of the box and change the color back again when the user leaves the box.

As an ObjectPAL programmer, you redefine the response of objects. In other words, you don't tell the box *to* respond, because Paradox built the box to be responsive; you merely tell the box *how* to respond. Furthermore, you don't have to figure out how you want that box to respond to every possible event; you only tell the box how to respond if you need to change the default response to a particular event.

Built-in methods and default responses

Every object has a set of default responses. To modify the default response of an object, you modify one or more of that object's *built-in methods*. You'll spend most of your time in ObjectPAL modifying built-in methods.

Types

ObjectPAL groups all the objects that you draw using the SpeedBar into a category called UIObjects. (Categories of objects are called *types* in ObjectPAL. Some other languages call them classes.) A page of a form is a UIObject, as is the form itself. Forms have behavior that goes beyond their behavior as a UIObject; for the time being, however, think of them as UIObjects.

UIObjects come with their own sets of built-in methods. These built-in methods are triggered automatically in response to events or actions. For example, when you click inside the boundaries of an object, Paradox calls the built-in **mouseClick** method for that object.

The set of methods built into an object varies according to the object. Most UIObjects have the same basic set, with a few notable exceptions; for example, field objects have a built-in **changeValue** method, but boxes don't. The **changeValue** method executes whenever the value in a field object changes. Having a **changeValue** method for a box doesn't make sense, because a box doesn't store data values.

Paradox makes finding and modifying an object's built-in methods easy. You inspect the object to display the Properties menu, choose Methods, and choose the built-in method you want to modify. An ObjectPAL Editor window opens, and Paradox positions the cursor at the point where you should start typing.

Modular

At the beginning of this chapter, you learned that you can use ObjectPAL to do as little or as much programming as you want. ObjectPAL offers this flexibility because objects are inherently modular. In other words, *objects are self-contained*. You can change the behavior of one object in a form without changing the behavior of all the objects in the form.

Objects are self-contained.

The implications of this feature are more far-reaching than you might think. The trivial side is that objects are easy to program. The non-trivial side is that you can use ObjectPAL to build complex systems. In a non-object-based language, a general rule is that the bigger or more complex a system becomes, the more likely it is to become unstable—and not merely because the system is bigger, has more lines of code, and consequently has more bugs.

With traditional languages, systems were built with all the intelligence at or near the top of the system. Processes in a system were seen as linear, and programming was approached in linear fashion. But real-world complex systems (systems such as traffic patterns and the stock market) embody organic principles: nothing travels in a straight line, and little change comes from the top of the system. In a real-world system, control flows not only from the top of the system but from the bottom—from the interaction of all the little pieces, or *subsystems*.

Model real-world behavior

Object-based programming lets you develop systems with a more organic approach. In an object-based system, you build the intelligence into the little pieces (the objects). If you focus on correctly modeling the behavior of the subsystems, the complete system is

likely to feel much more intuitive and much more like a real-world application.

Self-containment decreases errors.

Furthermore, an object-based system supports an incremental development process—a process where you can return to the program again and again to refine it. You can return to an object and make it smarter without jeopardizing the entire system. You can go back over the code for an object and modify that code because the object is relatively self-contained. To seasoned programmers, the implication of this capability is obvious: maintaining and improving an object-based system no longer requires the programmer to know everything about the system. In a well-designed system, one small change is exactly that—one small change.

You don't need to know object-oriented design principles to start programming in ObjectPAL. If you start with small chunks and then work your way up, your system will be better-designed than if you try to control everything from the top. The beauty of ObjectPAL is that the best way to use it is also the easiest:

- Place objects on a form
- Set properties for those objects
- If necessary, attach custom code to some of the built-in methods for those objects

You can write programs the old-fashioned way—starting at the top and working your way down to the bottom—but you'll never appreciate the full power of ObjectPAL if you do. On the other hand, if you keep your application modular, so that the code that affects an object is as close to that object as possible, your application will be well-designed and easy to maintain.

ObjectPAL for programmers

The following sections describe ObjectPAL for users who have programming experience. Some of these terms may be unfamiliar to you, but they are explained in more detail later in this chapter, in the following chapters, or in the *ObjectPAL Developer's Guide.*

Although ObjectPAL is designed to be accessible to non-programmers, serious developers should note that ObjectPAL is a full-featured, high-level, extensible language. It is suitable for demanding programmers writing sophisticated applications.

Language features

ObjectPAL supports the following functions:

- Built-in event handling
- Strong data typing
- User-defined data types
- Powerful data types such as resizeable arrays, associative arrays (called *dynamic arrays*), and records
- Structured program control
- An extensive library of methods and procedures
- User-defined methods and procedures
- Calls to functions and procedures written in other languages, such as Pascal, C, and C++
- Calls to an externally compiled help system (one compiled with a Windows Help Compiler)

Control features

ObjectPAL lets you trap for and change both keystrokes and mouse actions. Usually, however, you don't need to build this kind of low-level control. Instead, you can build code that responds to *events*. An event is a message to an object, generated by some activity (for example, pressing a key or clicking the mouse).

Managing events

You can capture, respond to, change, create, and simulate all mouse events, including position, movement, right-clicks, left-clicks, double-clicks, and clicks in concert with *Shift*, *Alt*, or *Ctrl*. You have similar control over all key presses.

You can open, position, size, minimize, maximize, and otherwise manipulate forms and all other display objects. You can use multiple forms as dialog boxes or as modules for an application.

Setting properties

Any object property that you can set interactively (for example, color), you can set in ObjectPAL. When a form is running, ObjectPAL can control many properties that are not available from the Properties menu, such as an object's position and focus status. For example, you could "follow" a user around a form by drawing a colored frame around the object that has focus.

You can create top-level menus with associated pull-down menus. You can hide the SpeedBar and even reset the main title of the Paradox Desktop.

Manipulating tables

Any table action that is normally available interactively is also available through ObjectPAL. A host of actions that are unavailable interactively are also available through ObjectPAL. For example, you can manipulate tables as Tables, TableViews, TableFrames, and

TCursors. A *Table* is what you use for utility functions, such as Add and Subtract. A *TableView* is a table opened in a window (what Paradox creates when you choose File | Open | Table). A *TableFrame* is a table object placed on a form. A *TCursor* is a table handle that you can use behind the scenes; TCursors are handy for searching and sorting.

Managing the file system

All the file system functions that are available interactively are also available in ObjectPAL. You can call the built-in Browser to let a user choose a file. You can delete or rename files, if you need to, or make directories.

Querying data

You can create queries (.QBE files) interactively and then execute those queries in ObjectPAL. You also can create query statements from scratch in ObjectPAL; these queries can include variables evaluated at run time.

Objects in ObjectPAL

The terminology of objects may be more confusing to experienced programmers than to novices, because the better you know a language or a paradigm, the harder it is to learn new terms for it. But learning about object-based programming is not difficult.

In everyday language, objects are just things—smart things. In programmer's terms, objects are data and code tightly bound together. You create objects either interactively or with ObjectPAL.

To make object-based programming work for you, just remember to start with the objects. In fact, start with the objects you already know best—the objects you place on a form.

Don't start by trying to build complex multi-form applications with low-level keyboard handlers, cascading menus, and flocks of dialog boxes. In fact, you will learn ObjectPAL more effectively if you try not to think too big, at least at first. Instead, for your first project, think about making specific objects on specific forms do what you want them to do. The possibilities for your *next* project will unfold as you go.

CHAPTER 3

The ObjectPAL environment

The ObjectPAL environment for the most part is the same as the one you've been working with all along: you use interactive Paradox to build a form that becomes the basis of your application. Two special tools, however, come in handy when you write ObjectPAL code: the ObjectPAL Editor and the ObjectPAL Debugger.

- The *ObjectPAL Editor* lets you write and check code (and actually does some of the writing for you).
- The *ObjectPAL Debugger* lets you step through code one instruction at a time, watch for certain changes, correct errors, and fine-tune your application.

The following sections describe these two tools.

The ObjectPAL Editor

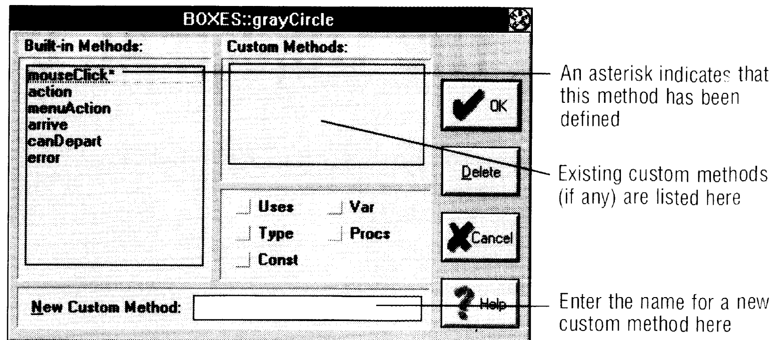
The ObjectPAL Editor is where you write ObjectPAL code. When you choose to modify a method or other code container from the Methods dialog box, Paradox opens an Editor window. Your entry point to the ObjectPAL Editor is the Methods dialog box.

The Methods dialog box

The Methods dialog box opens when you inspect an object and choose Methods from its menu, as shown in Figure 3-1. This dialog box gives you access to code written for a specific object.

Most of the time, you use the Methods dialog box to modify built-in methods. You can also use this dialog box to write other kinds of code, including custom methods, custom procedures, constants, variables, user-defined variable types, and declarations of library methods and functions.

Figure 3-1 The Methods dialog box



Following is a description of each panel in the Methods dialog box:

Built-in Methods

- The Built-in Methods panel of the dialog box lists the built-in methods available for this object. To open a built-in method, double-click the method name, or select the method name and click OK. To open several methods, select as many methods as you want and then choose OK. Paradox opens each method in a separate Editor window.

New Custom Method

- An entry in the New Custom Method field creates a new method attached to the current object. Paradox does not automatically call custom methods in response to events.

  Any custom methods defined for an object appear in the Custom Methods section of the dialog box.

Var box

- The Var box opens a variable window, called a Var window, in which you declare variables that are global to the object. You can also declare variables for a method in the Var section of that method.

  Declaring variables in a Var window makes the variables global to the object. If a variable is global to an object, you can use the variable in any of the methods attached to the object and in any of the objects contained by this object. (Where you can use a method, variable, constant, procedure, or type depends on where that item is defined. The availability of data objects is broadly termed *scoping*.)

Const box

- The Const box opens a constant-declaration window, called a Const window. Constants declared in a Const window are global to the object. You also can declare constants in a Const section of a method, but they will not be global to the object.

Type box

- The Type box opens a window in which you can declare user-defined data types. Data types declared in a Type window are available to all methods attached to this object and to all objects contained by this object. You also can declare user-defined types in the Type section of a method.

Proc box

- The Proc box opens a window that lets you define procedures global to the object. Procedures are like methods except that procedures are not bound to an object type. Procedures defined in a Proc window are available to all methods attached to this object and to all objects contained by this object. You also can declare procedures in Proc sections of a method.

Uses box

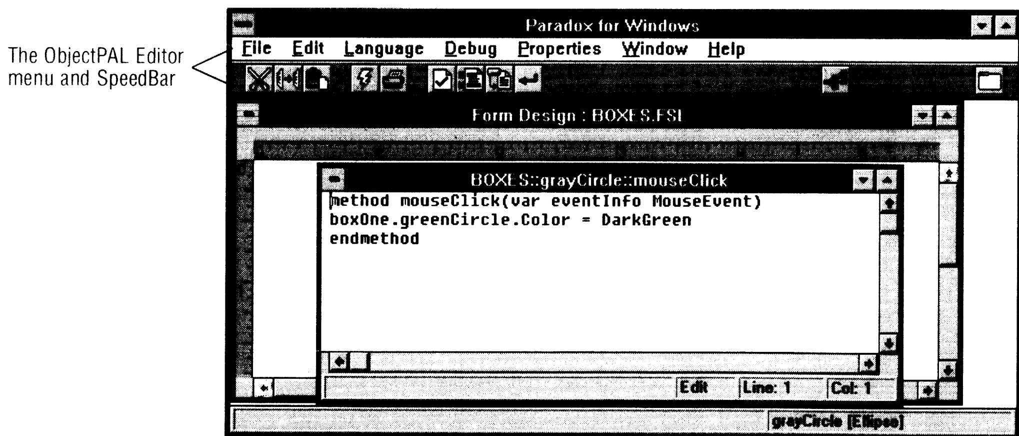
- The Uses box opens a window that lets you declare methods and functions called from an ObjectPAL library or a dynamic link library (DLL). You can also declare library methods and functions in a Uses section of a method.

The ObjectPAL Editor window

An ObjectPAL Editor window opens from the Methods dialog box whenever you select code to modify. When you open an Editor window, the menu changes as shown in Figure 3-2.

Figure 3-2 The ObjectPAL Editor

The ObjectPAL Editor menu and SpeedBar



The ObjectPAL Editor menu is available only when an ObjectPAL Editor window is open. The File, Window, and Help menus are standard menus.

Editing in an Editor window

In terms of text-handling, the ObjectPAL Editor works like most standard Windows editors. For example, to select text, either drag the mouse or hold down the *Shift* key while you press a direction key.

The ObjectPAL Editor window always opens in insert mode. In insert mode, anything you type is inserted at the current insertion point. To move around in the ObjectPAL Editor window, use the direction keys or the scroll bar.

Table 3-1 summarizes common ObjectPAL editing actions.

Table 3-1 ObjectPAL Editor actions

Action	**Key combination or menu command**
Select	*Shift*+direction key extends text selection, or Drag the mouse, or Edit|Select All selects the entire window
Copy	*Ctrl+Ins* copies selected text to the Clipboard, or Edit|Copy
Cut	*Shift+Del* cuts selected text to the Clipboard, or Edit|Cut
Paste	*Shift+Ins* pastes text from the Clipboard, or Edit|Paste
Delete	*Del* erases selected text, or Edit|Delete
Search	Edit|Search and Edit|Search Next find a string
Replace	Edit|Replace and Edit|Replace Next find and replace a string
Move to	Edit|Go To jumps to a specific line number

You can also use another editor, such as the Windows Notepad, to write ObjectPAL code (choose Properties | Alternate Editor from the ObjectPAL Editor menu). If you do, however, syntax checking and other special features of the ObjectPAL Editor menu will not be available.

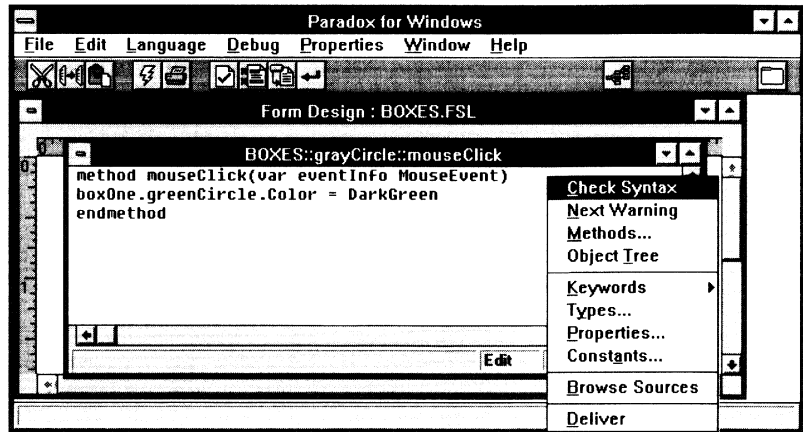
The Language menu

The Language menu helps you write and check ObjectPAL code. To check the code in a window for syntax errors, choose Language | Check Syntax. The compiler examines the code for the form. If the compiler finds an error, Paradox displays an error message in the status line of the open Editor window and positions the cursor at the point of the error.

Some types of errors affect code in other methods and may also affect code for other objects. When this problem occurs, the compiler opens each method that needs to be corrected.

Shortcut The Language menu is also available as a pop-up menu from any ObjectPAL Editor window. Right-click anywhere in the window to display the Language menu, as shown in Figure 3-3.

Figure 3-3 The Language pop-up menu



In the Language menu,

- *Check Syntax* compiles the code on a form.
- *Next Warning* shows the next compiler warning.
- *Methods* opens the Methods dialog box for the current object.
- *Object Tree* displays an Object Tree for the selected object.

The second group of menu items in the Language menu is designed to provide help while you're writing ObjectPAL.

Keywords

Choose Keywords to see a list of keywords, such as **proc** and **if**. When you choose a keyword, the application inserts the keyword into the current ObjectPAL Editor window at the cursor position.

The Types and Methods dialog box

Choose Types from the Language menu to display the Types and Methods dialog box, as shown in Figure 3-4.

Figure 3-4 The Types and Methods dialog box

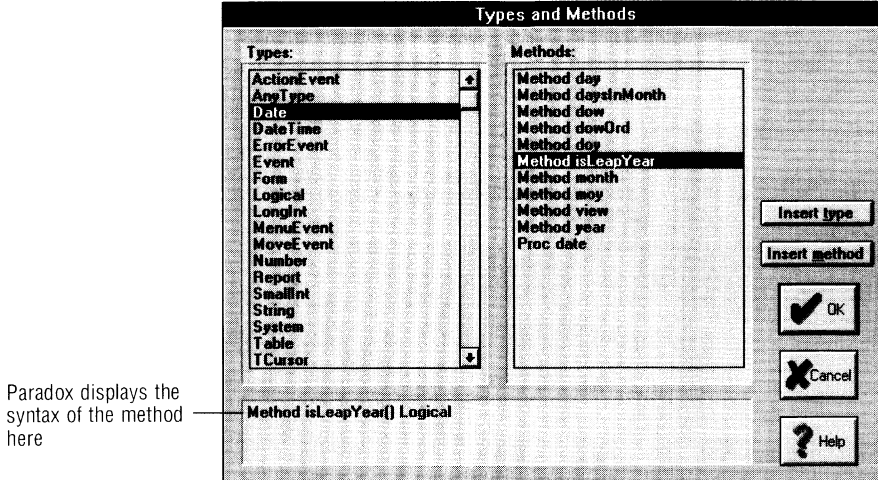


Figure 3-4 lists Beginner-level types and methods.

This dialog box is extremely useful while you're writing code. The Types section of this dialog box lists all Beginner-level ObjectPAL types. When you select a type, the Methods section of the dialog box lists the available methods for that type and the procedures associated with that type.

With the type of object or variable selected, you can use the Insert type button to insert the type name into the current ObjectPAL Editor window. Typically, however, you use an object's name in code. (An object is an instance of a type, not the type itself.) The type name is inserted when you use the Insert type button, even if you click Cancel to close the dialog box.

Displaying syntax

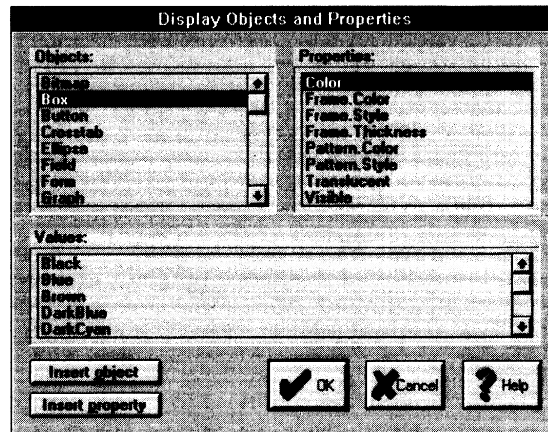
When you select a method or procedure from a type, Paradox displays the syntax (or prototype) of that method in the syntax section of the dialog box (the unlabeled panel at the bottom of the dialog box). With the method name selected, use the Insert method button to insert the method into the current ObjectPAL Editor window. If you insert a method, you'll see the the syntax for that method in the status line of the ObjectPAL Editor window when you return from the dialog box.

The syntax of a method tells you what arguments the method expects and what data type (if any) the method returns. This feature is particularly handy while you're learning the language. If you don't know the exact name of a method or whether a method is available for a certain object, browse through the Types and Methods dialog box. Also, remember that the Paradox for Windows Help system contains entries for every ObjectPAL method and procedure.

The Display Objects and Properties dialog box

Choose Properties from the Language menu to open the Display Objects and Properties dialog box, as shown in Figure 3-5.

Figure 3-5 The Display Objects and Properties dialog box



Like the Types and Methods dialog box, the Display Objects and Properties dialog box lists all the UIObjects (display objects, such as bitmaps and buttons).

When you select an object, all the property names for that object appear in the Properties section of the Display Objects and Properties dialog box. If you select a property, all the possible values for the selected property appear in the Values section of the dialog box.

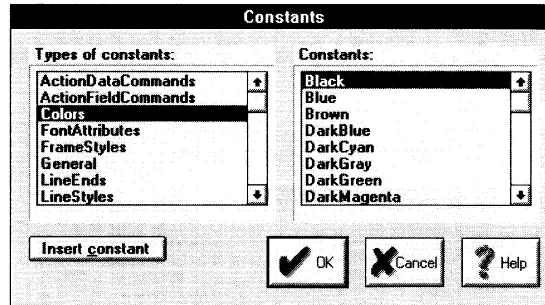
Properties are valuable resources.

The Properties and Values sections are valuable resources while you're learning ObjectPAL, because you can browse through all the properties and values to see what's available.

The Constants dialog box

Choose Constants from the Language menu to open the Constants dialog box, as shown in Figure 3-6. The Constants dialog box lets you view the various types of constants and lets you insert a constant directly into your code.

Figure 3-6 The Constants dialog box



A *constant* is a value of a certain data type that does not change throughout a program. (Variables, on the other hand, are expected to change through a program.) Colors are constants, as are most property settings. Run-time errors are also constants.

Creating a source code report

Typically, when you need to change an application, you open the form, choose the object that you want to change, and then modify a section of code. Sometimes, especially if you have a large application, you might want to see all the code in one place.

Language|Browse Sources

Language | Browse Sources creates a report that shows the ObjectPAL source code for all objects in a form.

The ObjectPAL equivalent of Language | Browse Sources is a method called **enumSource**. This procedure creates a table that contains one record for each defined method for each object in a form. The method definitions are stored in memo fields.

Delivering forms to users

Language | Deliver creates another version of the current form and gives that version an .FDL file-name extension. Users cannot enter a design window in a delivered form and thus cannot change source code. Naturally, you should deliver only complete and debugged forms.

The Properties menu

The Properties menu in the ObjectPAL Editor lets you set properties for the Desktop, choose a default ObjectPAL Editor window size, set up an alternate editor, and toggle the display of compiler warnings.

Properties | Desktop opens the Desktop Properties dialog box so you can set or change characteristics of the Desktop. For example, you can change the title of the desktop from "Paradox for Windows" to a custom title, such as "Inventory Control Application."

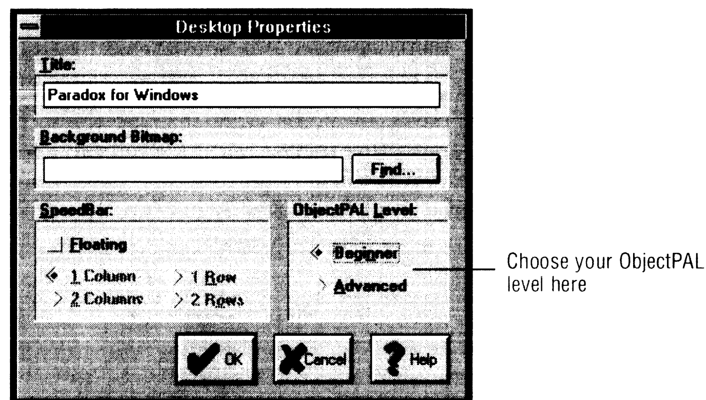
ObjectPAL Level

You also choose your ObjectPAL level from the Properties | Desktop dialog box. The Beginner level is the default; this level presents only

the most basic ObjectPAL methods, types, and constants. This subset of ObjectPAL is powerful enough to build full-featured applications yet small enough to learn in a short time. Make sure the ObjectPAL Level is set to Beginner to complete the tutorial in this manual, as shown in Figure 3-7. (To see all ObjectPAL types and methods, including those for advanced users, change your ObjectPAL level to Advanced.)

Note ObjectPAL code executes the same, regardless of the ObjectPAL Level setting. In other words, an application developed with the level set to Advanced will run on a system with the level set to Beginner.

Figure 3-7 ObjectPAL Level setting



Properties | Window Sizing

Paradox opens an Editor or Debugger window to a default size. To change this size, first open an Editor window and set it to the size you want. Next, choose Properties | Window Sizing. In the Editor Window Size dialog box, choose Use Current Sizing From Now On.

Properties | Alternate Editor

Properties | Alternate Editor lets you link to a different editor. With a different editor linked, whenever you attempt to open an Editor window, Paradox asks whether you want to use the default editor or the alternate editor. This feature is useful because even if you want to use another editor while you're creating code, you certainly will need to use the default editor to debug and fine-tune your code.

Note Because the ObjectPAL Editor provides a great deal of language help, you probably shouldn't use another editor until you're familiar with ObjectPAL.

Properties | Show Compiler Warning

Properties | Show Compiler Warning toggles the display of compiler warnings.

The ObjectPAL Debugger

The ObjectPAL Debugger is an integrated debugger, meaning that the debugger is available to you as you are writing and developing ObjectPAL code. Using the Debugger, you can

- Set breakpoints so you can execute statements up to a certain point, and then stop to see what has happened
- Open a window that lists each line of code as it executes
- Inspect variables to make sure that values are being manipulated as you inteneded
- Execute code one line at a time (called single-stepping)
- Step over methods and procedures that you know are bug-free

It's easier to appreciate the usefulness of the Debugger after you've written some code. Work through the tutorial examples in the chapters that follow, and then refer to Chapter 4 in the *ObjectPAL Developer's Guide* for a complete description of Debugger functions and a short tutorial showing how to use the Debugger.

On to programming

The following chapters present simple lessons you can complete to become familiar with ObjectPAL and its environment. You might want to browse through the *ObjectPAL Developer's Guide* and the *ObjectPAL Reference* as you move through the examples in this book.

After completing this tutorial, you should be able to move ahead with your own programming tasks.

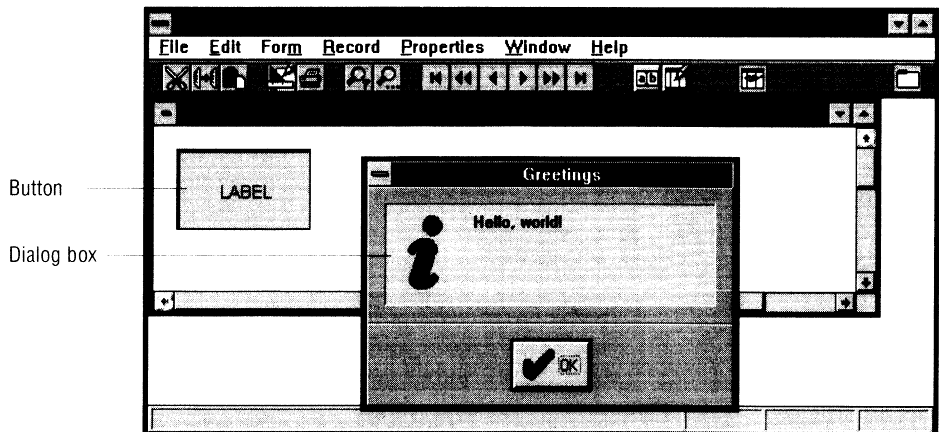
CHAPTER 4

Programming a button

The traditional way to demonstrate how to use a new language is to present a "Hello, world!" program. Such a program usually consists of only enough code to display the message "Hello, world!" on the screen. The classic "Hello, world!" program is not interactive. You run the program, the message appears, and that's it.

In contrast, ObjectPAL is a language for creating interactive, event-driven applications. In the following "Hello, world!" application, the user controls when to display the message and when to put it away. You'll program a button to display a dialog box, but the code executes only when the user clicks the button, and the dialog box stays open until the user closes it. Example 4-1 shows how to do this. First, Figure 4-1 shows the application in action.

Figure 4-1 Hello, world!



By adding the statement **msgInfo("Greetings", "Hello, world!")** to the button's built-in **pushButton** method, you make the dialog box appear when you push the button. The dialog box stays open until you close it.

Note To complete the examples in this manual, change your working directory to PDOXWIN\SAMPLE.

Example 4-1 Hello, world! application

1. Begin by creating a new form. Choose File|New|Form from the Desktop to enter the Form Designer. You'll see a succession of New Form dialog boxes. Accept the defaults in each dialog box to create a blank form. Most of the work in building Paradox applications involves placing objects in forms and writing ObjectPAL code to specify how they respond to events.

2. Click the Button tool to create a button. Place the button anywhere on the form, and don't worry about labeling it for now.

3. Now run the form by clicking the View Data button or by choosing Form|View Data.
4. Click the button you created and watch what happens. Its appearance changes, making it seem to push in and pop out. You didn't write any code, so how did this happen? By default, buttons push in and pop out when clicked. In the following examples, you'll write ObjectPAL code to make this button do more.

5. Click the Design button or choose Form|Design to continue designing your form.

Built-in methods

Every object in a form (as well as the form itself) includes built-in methods that execute in response to events. That is, Paradox interprets the event and calls the appropriate built-in method attached to the target object. For instance, when you clicked the button, it responded; no programming on your part was necessary because every Paradox object comes with default methods built in.

Changing the default behavior

To change the way a button behaves when you click it, you attach your own custom code to the button's built-in **pushButton** method. Example 4-2 shows you how to open the ObjectPAL Editor.

Example 4-2 Opening an ObjectPAL Editor window

1. Inspect the button to view its properties. (To inspect an object, you can right-click it, you can select it and press *F6*, or you can select it and choose Properties|Current Object.)

Built-in methods define how objects respond to events.

2. Choose Methods to display the Methods dialog box, which lists the button's built-in methods.

Shortcut

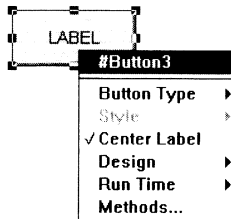
Select the button, and then press ***Ctrl+Spacebar*** to display the Methods dialog box.

3. You want to define what happens when the button is clicked, so select pushButton from the list of methods, then choose OK. An ObjectPAL Editor window opens, containing the following code:

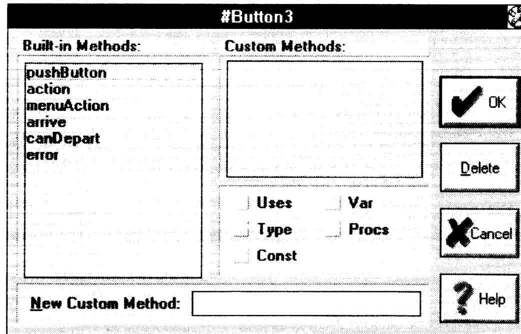
```
method pushButton(var eventInfo Event)

endmethod
```

Figure 4-2 shows each of these steps.

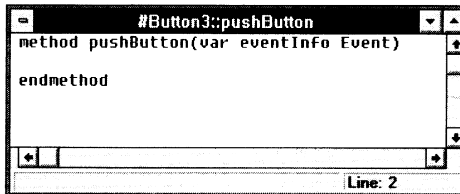
Figure 4-2 Inspect the object, select a method, open an Editor window



First, inspect the button and choose Methods from its menu to display the Methods dialog box



Next, select pushButton to edit the **pushButton** method, and choose OK to open an ObjectPAL Editor window



Finally, use the ObjectPAL Editor to edit the method

As it stands, this method doesn't do much. (If you're wondering about *var eventInfo Event*, it's explained in the "How it works" section later in this chapter. Don't worry about it now.) It's important to understand that for every object you can create, and for every event

that affects an object, Paradox has a default response. In some cases, Paradox's default response is something you can see, such as a button moving when you click it. In other cases, Paradox's default response is not visible, but it's necessary nonetheless.

It's also important to understand that the default response only executes when the object receives an event. For example, the button doesn't move until it's clicked.

Attaching your own code

The next step is to attach your own code to this method. Simply attaching code to a method *does not* disable the default response. Your code executes, and then the default code executes.

Example 4-3 Attaching your own code

1. If it's not already open, open an ObjectPAL Editor window to edit the button's **pushButton** method.
2. Add code to the **pushButton** method so it looks like this:

```
method pushButton(var eventInfo Event)
   msgInfo("Greetings", "Hello, world!")
endmethod
```

3. Click the Check Syntax button (or choose Language|Check Syntax) to check your code for syntax and spelling errors. If there are errors, a message in the status bar informs you. Otherwise, you'll see the message **No syntax errors** in the status bar and your changes are saved to memory.

Shortcut Right-click anywhere in the Editor window to display the Language menu.

4. Click the View Data button to run the form.
5. Click the button you created to display your greeting in a dialog box.
6. Choose OK to close the dialog box.

7. Return to the Form Design window, and choose File|Save to save your changes to disk. Name this form HELLO.FSL.

Copy objects from form to form with the Clipboard.

There you have it—an application that displays a message in a dialog box when (and only when) you click a button and then waits for you to close it. What's more, you can copy this button to the Clipboard and paste it into any Paradox form, and it will work exactly the same way.

How it works

Here's how the code you wrote works:

```
method pushButton(var eventInfo Event)
```

The first line says that this is a method named **pushButton**, that **pushButton** takes one argument, *eventInfo*, of type *Event*, and that *eventInfo* is a variable passed by reference (indicated by the *var* keyword). This statement (which Paradox supplies by default) conveys important information to ObjectPAL. You don't have to worry about it now.

The next line calls the **msgInfo** procedure.

```
msgInfo("Greetings", "Hello, world!")
```

msgInfo is a procedure in the ObjectPAL run-time library, a collection of pre-defined methods and procedures that operate on objects or data of a specific type. **msgInfo** takes two *arguments* (sometimes called *parameters*). The first argument, "Greetings," specifies the text to display in the title bar of the dialog box, and the second argument, "Hello, world!" specifies the text to display in the dialog box itself. This dialog box is *modal*; that is, it stays open until the user closes it, and the user must close it before continuing to work with the form.

The last line marks the end of the method.

```
endmethod
```

By default, the built-in code executes just before this line.

Summary

This lesson gave you an elementary look at what it takes to build a Paradox application. These are the basic steps:

1. Place objects in a form. Every object comes with built-in code, so forms will run even if you don't write any code yourself.
2. Run the form and watch how the objects behave by default.
3. Decide which object(s) should do something different or something more.
4. Decide what each object should do and when it should do it.
5. Attach your own code to the appropriate built-in method(s).
6. Run the form and make sure it does what you expect it to do. Debug it if necessary.

CHAPTER 5

Initiating and responding to actions

The next several lessons are presented in the context of a single application: a form for entering customer data. Lessons in this chapter show

- How code attached to one object can affect another object
- How to use the UIObject **action** method to initiate actions
- How to use the built-in **action** method to respond to action
- How to use the UIObject **locate** method to search for values in a table

Stages in writing ObjectPAL applications

In general, ObjectPAL applications are built in the following stages:

1. Get your data together.

   Build and populate tables. If you're building a multi-table application, determine your data model before building tables.

2. Create the form.

   Although you can write scripts and store code in libraries, the vast majority of ObjectPAL code is attached to objects in forms. The best way to start is to create the form, place the objects, and run the form. Observe how Paradox behaves by default, without any ObjectPAL code attached. You'll find that the built-in code does what you want in most cases. As you watch the default behavior, you can decide which objects should do something different or something more. Then you can move to the next stage.

3. Attach ObjectPAL code to the object's built-in methods.

   Modify an object's behavior by attaching code to the appropriate built-in method. Built-in methods execute in response to events,

so to select the appropriate built-in method, you should determine what the object should do and when it should do it.

4. Test and refine the application.

   Paradox makes it easy to develop an application one piece at a time. You don't have to code the entire application before you can run the form and make sure things are happening as you expect.

Note For the purposes of this tutorial, the first stage has been completed for you: tables have been created and populated with data. Set your working directory to the directory containing your sample files (usually C:\PDOXWIN\SAMPLE) to access these tables and then work through the examples in this tutorial.

If you've changed the sample tables, reinstall them before you work through this tutorial. Otherwise, you might not get the expected results from the examples in this manual. To reinstall the sample tables, run INSTALL and check only the Install Sample Tables option. If you're accessing the sample tables on a network, see your network administrator.

Creating the form

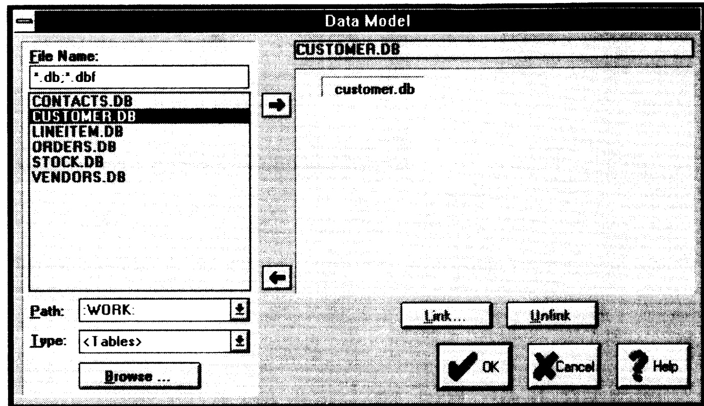
This section begins by explaining how to create a single-record form; that is, a form that displays one record at a time from a table. Subsequent lessons explain how to perform the following tasks:

- Programming a button to take an action
- Responding to an action
- Searching for values

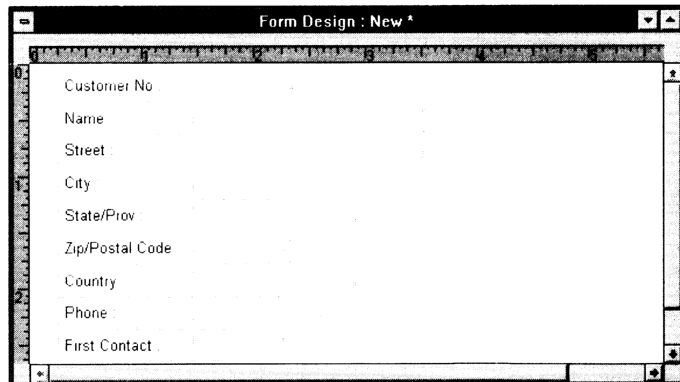
Before you begin these lessons, you need to create the form. Once you create and save this form, you can use it to complete any of the lessons in this chapter.

Example 5-1 Creating the *NewCust* form

1. Choose File|New|Form to display the Data Model dialog box. Make sure you're working in the SAMPLE directory. (For help, see Chapter 4 of *Getting Started*.)
2. Add the *Customer* table to the data model, as shown in the following figure:

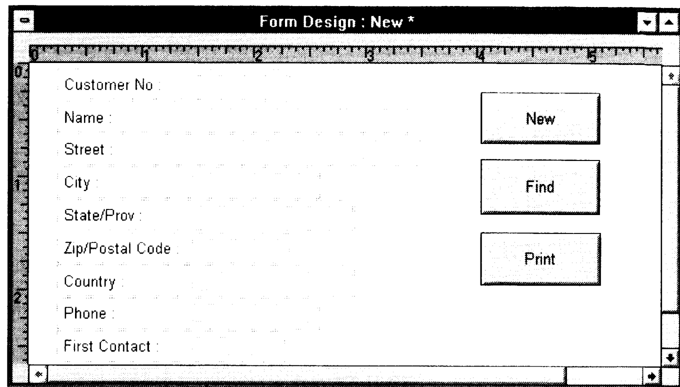


3. Choose OK to accept this data model and display the Design Layout dialog box.
4. Choose OK to accept the default layout and display the new form (shown here) in a design window.



5. Next, use the Button tool to add three buttons. Place them in the form and label them *New*, *Find*, and *Print*, as shown in the following figure:

A button's label is just a text box. To change the text, select the label and change the text.



6. Finally, choose File|Save to save the form. Name it NEWCUST.FSL. Throughout these lessons, this form will be referred to as the *NewCust* form. You can use the *NewCust* form as a starting point for the lessons that follow.

Programming a button to take an action

In this lesson, you'll attach code to a button to initiate a specific action: inserting a new record.

Example 5-2 Attaching code to a button

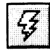
1. Inspect the button labeled *New* and change its name to *newButton*. To change an object's name, inspect the object, and click its default name (it's the first item in the object's menu). You'll see the Object Name dialog box, and that's where you type the new name.

   You don't have to name an object to attach code to it, but a name can remind you what an object is supposed to do, and it provides a convenient way to refer to the object in conversation or text, as well as in your code.

2. Inspect *newButton* again to display its menu, and choose Methods to open the Methods dialog box.
3. Select **pushButton**, and then choose OK to open an Editor window for the built-in **pushButton** method.
4. Add code to the **pushButton** method so it looks like this:

```
method pushButton(var eventInfo Event)
   action(DataInsertRecord)
endmethod
```

5. Check your syntax and correct any errors.



6. Run the form.

7. Click the Edit Data button (or choose Form|Edit Data, or press *F9*) to switch the form into Edit mode.
8. Click *newButton* to insert a new, empty record. (In a later lesson, you'll add more code to this button to make it more useful.)
9. You can enter data into this record, or choose Record|Delete (or press *Ctrl+Del*) to delete it.

10. Return to the Form Design window and save the form.

How it works

When this **pushButton** method executes, the statement **action(DataInsertRecord)** has exactly the same effect as choosing Record | Edit Data or pressing *Ins*. In other words, this button doesn't do anything you couldn't do already. However, it illustrates an important point: a form responds the same way to an ObjectPAL **action** statement as it does to a user action. In fact, as you're learning ObjectPAL, it may be helpful to think of the form "translating" user actions to **action** statements.

The basic **action** statement has two parts:

- The method name, **action**, that refers to the **action** method defined for UIObjects.
- An ObjectPAL action constant (for example, DataInsertRecord) that identifies the action to perform.

Important The **action** statement is the fundamental technique for initiating an action from ObjectPAL: write a statement that combines the **action** method with a constant that specifies what to do.

Table 5-1 lists the action constants available at the Beginner level. Understanding how to use these constants is vital to getting the most out of ObjectPAL.

Table 5-1 ObjectPAL Beginner-level action constants

Constant	Description
DataInsertRecord	Insert a record
DataDeleteRecord	Delete a record
DataLockRecord	Lock a record
DataUnlockRecord	Unlock a record
DataPostRecord	Post (commit) a record to the database
DataCancelRecord	Cancel changes to a record
DataBegin	Move to the first record in a table

Constant	Description
DataInsertRecord	Insert a record
DataEnd	Move to the last record in a table
DataPriorRecord	Move to the previous record in a table
DataNextRecord	Move to the next record in a table
DataBeginEdit	Begin Edit mode
DataEndEdit	End Edit mode
DataArriveRecord	Point to a new or changed record (used only to respond to the action, not to initiate it)
FieldForward	Move to the next object in the tab order
FieldBackward	Move to the previous object in the tab order

Responding to an action

It's not enough to initiate actions: you also need to control how objects respond. As explained in previous lessons, every object in a form has built-in methods that will meet your needs in most cases. Sometimes, though, you want something different or something more. This lesson shows how to control an object's response to a given action.

In this lesson, the task is to control tab order, which specifies the object that the cursor moves to when you press *Tab*. First, get a feel for the way the form behaves by default.

Example 5-3 Tab default behavior

1. Run the *NewCust* form. The cursor (highlight) is in the first field object, *Customer_No*.
2. Press *Tab*. The cursor moves to the *Name* field object.
3. Press *Tab* a few more times, and watch how the cursor moves from field object to field object.

When you run a form and interact with it, you generate actions. Pressing *Tab* is no exception; the result is an action, and the constant that identifies it is FieldForward, as shown in Table 5-1. (When you press *Shift+Tab* to move backward, the resulting action is FieldBackward.)

The default tab order starts with the topmost field object and works down. Suppose, though, that after you enter the customer number and the name, you want to move directly to *Phone*, bypassing the other field objects. Then, after you enter the customer's phone number, you want to move back to *Street* and enter address data.

Using ObjectPAL, this is easy to do, once you understand two key concepts:

- Every object in a form (including the form itself) has a built-in **action** method that executes in response to actions generated by the user, by Paradox, or by ObjectPAL.
- To control how an object responds, attach code to the object's built-in **action** method.

Example 5-4 applies these concepts to control how the *Name* field object responds when you press *Tab*.

Example 5-4 Controlling tab order

1. Click the Design button (or press *F8*) to return to the design window.
2. Inspect *Name* and choose Methods to display the Methods dialog box.
3. Select **action**, and then choose OK to open an Editor window for the built-in **action** method.

Shortcut You can also just double-click a method to open an Editor window.

4. Edit the method so it looks like this:

```
method action(var eventInfo ActionEvent)
   if eventInfo.id() = FieldForward then
      disableDefault
      Phone.moveTo()
   endIf
endmethod
```

5. Close this Editor window, and save the changes when prompted.
6. Inspect *Phone* and choose Methods to display the Methods dialog box.
7. Select **action**, and then choose OK to open an Editor window for the built-in **action** method.
8. Edit the method so it looks like this:

```
method action(var eventInfo ActionEvent)
   if eventInfo.id() = FieldForward then
      disableDefault
      Street.moveTo()
   endIf
endmethod
```

9. Check your syntax, and correct any errors.

10. Run the form, and then press *Tab* twice (or press *Enter* twice, which has the same effect). Verify that you move to *Phone*.
11. Press *Tab* again, and verify that you move to *Street*.

12. Switch to the Form Design window (*F8*) and save the form.

How it works

Each method you customized in this example now contains six lines of code: two lines are supplied by default, four lines are custom code. This section discusses only the custom code, starting with the code attached to the *Name* field object.

Line 2 The line of custom code after the default method header is

```
if eventInfo.id() = FieldForward then
```

Technically, this line tests whether the value returned by **eventInfo.id** is equal to the value of the action constant FieldForward. That's a lot to swallow in one gulp, so here's the same information in bite-size chunks:

- **if** marks the beginning of an **if...endIf** block. This block lets you execute statements only when certain conditions are met.
- The **id** method operates on the variable *eventInfo* and returns a value that identifies the action. Remember, *eventInfo* contains information about the event that triggered the built-in method. In this case, the event is an action, and *eventInfo* contains information that identifies the action. The **id** method retrieves this information.
- The returned value is compared to the value of the action constant FieldForward. All ObjectPAL constants have predefined values.
- If the two values are the same, the next line of code executes. That is, the next line executes only when the returned value is equal to FieldForward. If the returned value is anything else (for example, DataInsertRecord), the rest of this **if...endIf** block does not execute. Instead, the default code executes, and the method is finished.

Line 3 The third line of the method is

```
disableDefault
```

disableDefault prevents the built-in code for this method from executing. In this case, as you have seen, the default behavior is to move the cursor to the *Street* field object. By calling **disableDefault**, you prevent that from happening.

Line 4 The fourth line of the method is

```
Phone.moveTo()
```

As you know, *Phone* is the name of a field object in this form. **moveTo** is a method that moves the cursor to an object. So, this statement moves the cursor to *Phone*.

Line 5 The fifth line of the method is

```
endIf
```

endIf marks the end of an **if...endIf** block.

The code attached to *Phone* is exactly the same except for the following statement, which moves the cursor to the *Street* field:

```
Street.moveTo()
```

Important This simple example presents the framework for responding to actions in ObjectPAL: attach code to an object's built-in **action** method, call **eventInfo.id** to identify the action, and use action constants to test for the action or actions that you want to handle.

You can initiate and respond to actions using the same action constants. For example, the following code uses the action constant DataPostRecord to initiate an action.

```
method pushButton(var eventInfo Event)
   action(DataPostRecord)
endmethod
```

The following code uses the same action constant to respond to an action: if the action is DataPostRecord, the code displays a message in a dialog box, and then allows the default code to execute and post the record. If it's any other action, Paradox skips to the end of the method and executes the default code.

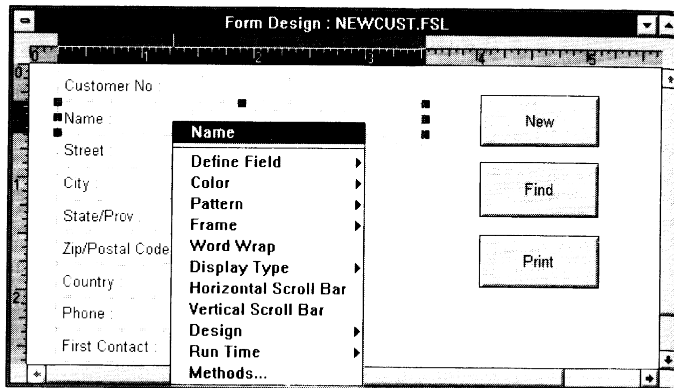
```
method action(var eventInfo ActionEvent)
   if eventInfo.id() = DataPostRecord then
      msgInfo("FYI", "About to post the current record.")
   endIf
endmethod
```

Actions and properties

Like the previous lesson, this lesson shows how to respond to an action. In addition, this lesson shows how to work with object properties and how to manipulate properties of one object based on the properties of another object.

Every design object has specific characteristics and attributes, which in Paradox are called *properties*. When you inspect an object, the object's menu lists many properties, as shown in Figure 5-1. Using ObjectPAL, you have access to all these properties and many more.

Figure 5-1 Properties of the field object *Name*

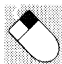


Example 5-5 shows how to use the DataArriveRecord action constant to respond to an action that can be initated in several ways. A DataArriveRecord action occurs whenever the form "arrives at" (displays) a new or different record. For example, moving to the next or previous record initiates a DataArriveRecord action, as does inserting, deleting, or editing a record. DataArriveRecord is a useful general-purpose action. (You can only respond to a DataArriveRecord action; you can't initiate it.)

Suppose that, as you work with customer data, you want to highlight the names of long-time customers—that is, customers you contacted before January 1, 1991. The following example shows how to do this.

To work through this lesson, use the *NewCust* form you created earlier in this chapter (open it in a design window).

Example 5-5 Responding to an action



1. Inspect the *Name* field object, and choose Methods to open the Methods dialog box.

2. Select **action**, and then choose OK to open an Editor window for the built-in **action** method.

Note If you worked through the previous lessons, you may find some custom code already attached. You can either delete the previous code or comment it out for now. To comment code, enclose it in curly braces { }.

3. Edit the method to make it look like this:

```
method action(var eventInfo ActionEvent)
   if eventInfo.id() = DataArriveRecord then
      if First_Contact.Value < Date("1/1/91") then
         Self.Color = Green
      else
         Self.Color = White
      endIf
```

```
    endIf
endmethod
```

4. Check your syntax and correct any errors.

5. Run the form.
6. Move the insertion point to the *Name* field object, then scroll through the records.
7. Notice the color of the *Name* field object. It should be green if the value of the *First_Contact* field object is less than 1/1/91; otherwise, it should be white.

How it works

This method executes whenever the *Name* field object responds to an action (and it only works if you move the insertion point into *Name*). If the action is anything *except* DataArriveRecord, only the built-in code executes, and *Name* behaves like any other field object. But when the action is DataArriveRecord, the custom code executes and makes *Name* do something special.

Line 2

The second line of the method is

```
if eventInfo.id() = DataArriveRecord then
```

This statement identifies the action and tests to see if it is DataArriveRecord, the one you're interested in.

Line 3

The third line is

```
if First_Contact.Value < Date("1/1/91") then
```

This statement reads the Value property of the field object *First_Contact* and tests to see if it is less (earlier) than 1/1/91.

Important

In the *Customer* table, the field name *First Contact* contains a space, but in this form, the name of the field object *First_Contact* contains an underscore. Here's the rule: names of fields in tables can contain spaces; names of objects in a form cannot.

Working with an object's properties

Here is the basic syntax for working with an object's properties:

```
objectName.propertyName
```

Replace *objectName* with the name of an object (*First_Contact*, in this example) and replace *propertyName* with the name of a property (in this example, Value).

The Value property lets you read and write the value of an object. For example, the following statement puts the value 1234 into the *Customer_No* field object, just as if you had typed it yourself.

```
Customer_No.Value = 1234
```

Important ObjectPAL uses special syntax for working with date values. In this example, the quotes make this an expression and tell ObjectPAL to treat 1/1/91 as a date (otherwise, it treats the / symbol as a division operator):

```
Date("1/1/91")
```

ObjectPAL automatically treats the value of the *First_Contact* field object as a date, because it's defined as a Date field in the underlying table (CUSTOMER.DB).

Line 4 The fourth line is

```
Self.Color = Green
```

Using the Self variable This statement uses the basic syntax for working with properties, *objectName.propertyName*. The property name is Color, but what is *Self*? *Self* is a built-in object variable that refers to the object executing the current code.

In this statement, the object executing the code is the field object *Name* (you moved the insertion point into *Name* in step 6 of this exercise), so *Self* refers to *Name*.

Here, *Self* is a very convenient shortcut. In more complex applications, *Self* is an important element in generalized code, because it lets you operate on objects without specifying them by name.

Summary

These lessons have presented some powerful techniques. They've shown you how to

- Use the UIObject **action** method to initiate actions
- Use the built-in **action** method to respond to actions
- Use action constants to initiate and respond to actions
- Prevent default code from executing
- Replace default code with code of your own
- Work with object properties
- Use the object variable *Self*

Subsequent lessons expand on these techniques and show you how to exercise even more control over an application.

CHAPTER 6

Input and output

The lessons in this chapter show how to get information from the user, how to display information to the user, and how to process this information along the way.

This chapter covers the following topics:

- A quick way to get user input
- Searching for values in a table
- Inserting a record and generating a unique key value
- Printing a report

The examples in these lessons explain how to add code to the *NewCust* form you created previously.

A quick way to get user input

This lesson shows how to use the **view** method to display a dialog box where a user can enter a value. It also shows how to declare and use variables in ObjectPAL. In terms of the completed application, this example isn't very useful. However, it presents concepts you can use to do something more practical.

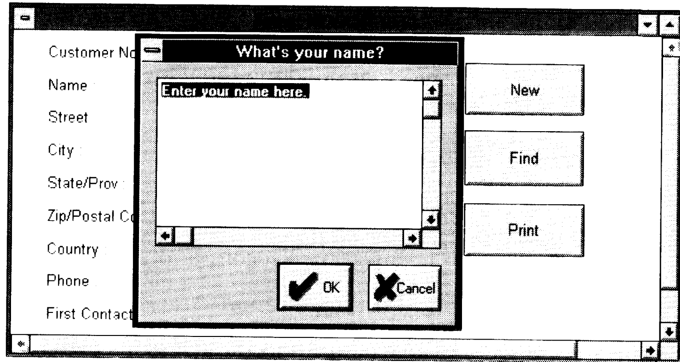
Example 6-1 Using **view** to display a dialog box

1. Open the *NewCust* form in the design window.
2. Inspect the button labeled *Find*, and change its name to *findButton*.
3. Select *findButton*, and press ***Ctrl+Spacebar*** to display the Methods dialog box.
4. Choose **pushButton** by double-clicking it. This opens an Editor window for the **pushButton** method.
5. Edit the method to make it look like this:

```
method pushButton(var eventInfo Event)
   var
      userInput String
   endVar

   userInput = "Enter your name here."
   userInput.view("What's your name?")

   message("Hello ", userInput)
   sleep(1000)
endmethod
```

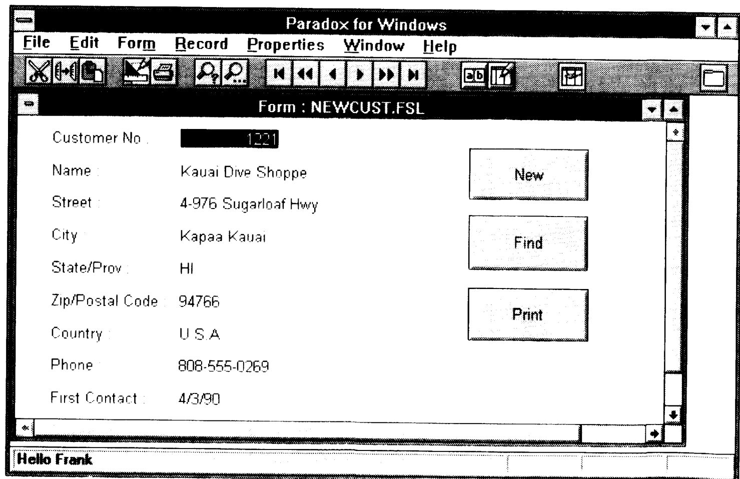
6. Check your syntax, and correct any errors.

7. Run the form, and click the *Find* button. A dialog box appears and prompts you to enter your name, as shown here:



8. Type your name into the dialog box, and press ***Enter*** or click OK.

9. A message appears in the status bar at the lower-left corner of the form's window, as shown here:



The message appears here

10. Return to the Form Design window, and choose File|Save to save the form.

How it works

This example accomplishes a lot with few lines of code. Following is an explanation of each line of custom code.

Line 2 The line of custom code after the built-in method header consists of a single keyword:

```
var
```

var declares the beginning of a block where variables are declared. (Variables are declared by specifying a name and a data type.)

Line 3 The third line is

```
userInput String
```

This code declares a variable whose name is *userInput* and whose data type is String. Now you can use the variable *userInput* in this method to store a character string.

Note ObjectPAL does not require that you declare variables, but you gain advantages if you do: code executes faster, it's less prone to syntax errors, and it's easier to read and maintain.

Line 4 The fourth line is

```
endVar
```

endVar marks the end of the variable declaration block.

Line 6 The sixth line of the method is

```
userInput = "Enter your name here."
```

This statement assigns a value to *userInput*. In other words, it stores the character string "Enter your name here." in the variable. In ObjectPAL, you must assign a value to a variable before you use it in another statement or expression.

Line 7 The seventh line is

```
userInput.view("What's your name?")
```

When this statement executes, the method **view** operates on the variable *userInput*. **view** displays the value of the variable in a dialog box. The string "What's your name?" specifies the text to display in the dialog box's title bar.

view does more than display the value of a variable in a dialog box; when you close the dialog box, **view** assigns the displayed value back to the variable. This new value is used by the next line of custom code.

The **view** dialog box can handle numeric input as well as character strings. For example, the following code prompts you to enter a credit card number:

```
method pushButton(var eventInfo Event)
   var
      cardNumber Number
   endVar

   cardNumber = 0      ; assign a temporary value
   cardNumber.view("Enter the Credit Card number.")
   message(cardNumber) ; display the new value
   sleep(1000)
endmethod
```

Line 9 The ninth line is

```
message("Hello ", userInput)
```

This statement calls the System procedure **message** with two arguments: the literal string, "Hello ", and the variable *userInput*. In this example, the value of *userInput* is whatever name you entered in the **view** dialog box. If you entered the name Dolly, this **message** statement would display **Hello Dolly** in the status bar.

Line 10 The last line of custom code is

```
sleep(1000)
```

A **sleep** statement makes the system "sleep" (actually, it just waits) until a specified number of milliseconds have elapsed; then it resumes. This example specifies 1,000 milliseconds (one second) to give you a chance to read the message.

Searching for values

This lesson shows how to get a value from the user, search a table for a record containing that value, and display that record to the user. You'll use a **view** dialog box to get input from the user (as shown in the previous lesson), use the **locate** method to search for the value, and let the form handle the display work.

Work through the following example using the *NewCust* form you created in previous lessons.

Example 6-2 Searching based on user input

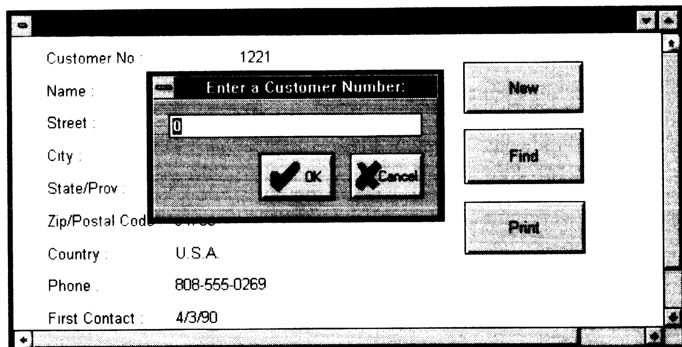
1. The *NewCust* form should be open in a design window.
2. Inspect *Find*, and choose Methods to open the Methods dialog box.
3. Double-click **pushButton** to open an Editor window for the built-in **pushButton** method.

4. Edit the method to make it look like this:

```
method pushButton(var eventInfo Event)
   var
      custNum Number
   endVar

   custNum = 0
   custNum.view("Enter a Customer Number:")

   if custNum <> 0 then
      if not Customer_No.locate("Customer No", custNum) then
         beep()
         message("Couldn't find ", custNum)
         sleep(1000)
      endIf
   endIf
endmethod
```

5. Check your syntax, and correct any errors.

6. Run the form, and click *Find*. A dialog box prompts you to enter a customer number, as shown here:



7. Type **1560** into the dialog box and press *Enter* or click OK to close it.
8. This number is in the *Customer* table, so the form displays that record.
9. Click *Find* again to display a **view** dialog box.
10. Type **1** into the dialog box and press *Enter* or click OK to close it.
11. This number is not in the *Customer* table, so Paradox beeps and the form displays a message in the status bar.

12. Return to the Form Design window, and choose File|Save to save the form.

How it works

Much of this example uses elements explained in the previous lesson. Only the new ones are discussed in detail here.

Lines 2 through 4

Lines two through four are

```
var
   custNum Number
endVar
```

This code declares a variable named *custNum* to be of type Number.

Lines 6 and 7

The next two lines are

```
custNum = 0
custNum.view("Enter a Customer Number:")
```

This code assigns a value of 0 to *custNum*, displays the value in a dialog box, and waits for you to enter a new value and close the dialog box.

Line 9

The ninth line is

```
if custNum <> 0 then
```

This code performs a simple test to see if you entered a value into the dialog box. The previous statement assigned a value of 0 to *custNum*. This statement tests the value of *custNum*, and if it's anything other than 0, the next line of code executes. Otherwise, Paradox skips to the end of the method and executes the default code.

Line 10

The tenth line is

```
if not Customer No.locate("Customer No", custNum) then
```

To understand what's happening in this statement, analyze it piece by piece.

First, look at **Customer_No.locate("Customer No", custNum)**. *Customer_No* is the name of a field object in this form; it is bound to the *Customer* table. **locate** is the name of a method. *Customer No* is the name of a field in the *Customer* table. *custNum* is a variable.

This piece says, in effect, "In the *Customer* table, locate a record in which the value of the *Customer No* field matches the value of the variable *custNum*." In other words, you use the object *Customer_No* to specify the table to search, the method **locate** to specify the search operation, the field name Customer No to specify which field to search, and the variable *custNum* to specify a value to search for.

Important The field name *Customer No* contains a space, but the name of the field object *Customer_No* contains an underscore. Names of fields in tables can contain spaces; names of objects in a form cannot.

The basic **locate** statement consists of the following parts:

- An object name. Typically, this will be the name of a field object bound to a table. That's the easiest way to specify which table to search.
- The method name, **locate**.
- A field name.
- A value to search for.

locate starts searching at the first record in the table and continues until it finds an exact match. If it succeeds (that is, if it finds an exact match), the form displays that record. You don't have to do anything else. If the search fails, the form displays the current record.

The rest of this statement is the familiar **if...then** with a new element: **not**. As you've seen in previous lessons, the basic **if...then** statement tests for a condition to be true. When you add **not**, you're testing for a condition to be false. Taken as a whole, then, this statement says, "If you can't find a value that matches *custNum*, execute the next lines of code. Otherwise, skip to the end of the method and execute the default code."

Lines 11 through 13 The following lines inform you when a search is unsuccessful:

```
beep()
message("Couldn't find ", custNum)
sleep(1000)
```

The **beep** statement plays the system beep sound to alert the user. The **message** and **sleep** statements are explained in Example 6-1.

ObjectPAL provides more powerful versions of **locate**, along with other methods that let you search for a pattern of characters instead of an exact match and methods that search forward and backward in a table.

Important There are no search and replace methods in ObjectPAL; the best tool for that kind of operation is a *query*. Refer to the *User's Guide* for more information.

Inserting a record and generating a unique key value

One of the benefits of using ObjectPAL is being able to automate tasks. For example, suppose you're an order entry clerk. If you just use the form interactively (without using ObjectPAL), you'd have to go through the following steps to enter an order:

1. Choose Form | Edit Data.

2. Choose Record | Insert.

3. Enter a new, unique value into the *Customer_No* field object.

The last step is likely to be the most difficult, since you'd have to keep track of the customer numbers already in the table and come up with a unique number for each new customer.

This lesson shows how to use ObjectPAL to perform these three steps in a single mouse click. Like the previous lessons, it uses the *NewCust* form.

Example 6-3 Inserting a new record

1. Inspect *newButton* (the button labeled *New*), and choose Methods to open the Methods dialog box.

2. Double-click **pushButton** to open an Editor window for the **pushButton** method.

Note If you worked through the previous lessons, you may find some custom code already attached. You can either delete the previous code or comment it out for now. To comment code, enclose it in curly braces { }.

3. Edit the method to make it look like this:

```
method pushButton(var eventInfo Event)
   var
      newCustNum Number
      custTbl Table
   endVar

   action(DataBeginEdit)
   action(DataInsertRecord)
   doDefault

   custTbl.attach("CUSTOMER.DB")
   newCustNum = custTbl.cMax("Customer No") + 1
   Customer_No.Value = newCustNum
   action(DataPostRecord)
endmethod
```

4. Check your syntax and correct any errors.

5. Run the form and click *newButton*. The form enters Edit mode, inserts a record into the table and a new value into the *Customer_No* field object. You have a new record, ready for editing.

6. Return to the Form Design window and choose File|Save to save the form.

How it works

The code in this example is organized into three blocks. The first two blocks use elements explained in previous lessons. The third block introduces new elements, which will be discussed in detail here.

Block 1 The first block of custom code is

```
var
   newCustNum Number
   custTbl Table
endVar
```

This block declares two variables: *newCustNum*, of type Number; and *custTbl*, of type Table. These variables differ slightly in their behavior. A Number variable stores a numeric value, while a Table variable provides a *handle*, something you can use in your code to refer to and manipulate a table.

Block 2 The second block of custom code is

```
action(DataBeginEdit)
action(DataInsertRecord)
doDefault
```

This block initiates two actions and then calls **doDefault** to execute the default code for these actions. The first statement, **action (DataBeginEdit)**, puts the form into Edit mode. If it's already in Edit mode, this statement is effectively ignored. The second statement, **action(DataInsertRecord)**, inserts a new empty record, as explained in the previous chapter. The call to **doDefault** executes the default code for these actions immediately, rather than waiting until the end of the method. It's important to call **doDefault** at this point, because the default code inserts the record and readies its field objects to receive values.

Block 3 The third block of custom code is

```
custTbl.attach("CUSTOMER.DB")
newCustNum = custTbl.cMax("Customer No") + 1
Customer No.Value = newCustNum
action(DataPostRecord)
```

The first statement in this block consists of the following elements: the Table variable *custTbl*, the method name **attach**, and the file name of the *Customer* table, CUSTOMER.DB.

Note By default, ObjectPAL looks for files in your working directory. You can specify a different location by adding the full path name or an alias to the file name.

As mentioned earlier, a Table variable provides a handle to a table. This **attach** statement associates the Table variable *custTbl* with the Paradox table CUSTOMER.DB. Now, when you use *custTbl* in this method, you're referring to the *Customer* table.

The second statement assigns a value to the Number variable *newCustNum*. To get the value, it uses the Table variable *custTbl*, the method **cMax**, and the Customer No field. **cMax** returns the maximum value in a specified column of a table. In this example, **cMax** operates on *custTbl*, which is associated with the *Customer* table. It checks the Customer No field of each record in the table and returns the largest value. By adding one to that value, you're guaranteed to have a unique value. For example, suppose **cMax** returns a value of 4456. That means 4456 is the largest order number currently in the table, so 4456 + 1, or 4457, has to be unique. This value is assigned to *newCustNum*. It's important that this value be unique, because it will be used in the key field *Customer No*, which requires a unique value by definition.

The third statement assigns the value of *newCustNum* to the *Customer_No* field object. It uses the Value property (discussed in Example 5-5) to specify a value to store and display in a field object.

The last statement in this block initiates a DataPostRecord action to post this new record (including the new customer number) to the *Customer* table.

Note The technique presented in this example is for single-user applications. Chapter 12 in the *ObjectPAL Developer's Guide* presents techniques to use in a multi-user application.

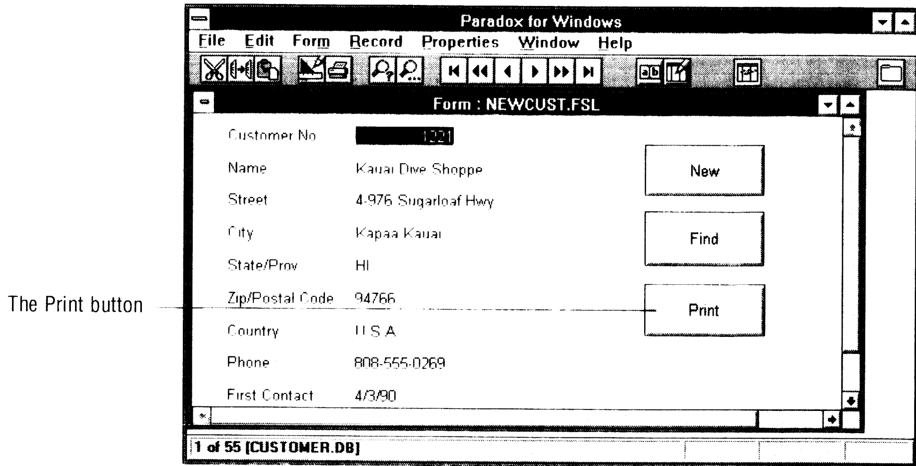
Printing a report

This lesson shows the basic technique for printing a predesigned report; that is, a report that has already been designed and saved to disk. If you've worked through the previous lessons, you can use the *NewCust* form for this lesson.

Example 6-4 Printing a pre-designed report

The code in this example is attached to the button labeled *Print* in the *NewCust* form.

You'll need to create a simple report on the *Customer* table. Name this report CUSTOMER.RSL.



1. Inspect the button labeled *Print*, and change its name to *printButton*.
2. Inspect *printButton* again, and choose Methods to open the Methods dialog box.
3. Double-click **pushButton** to open an Editor window for the **pushButton** method.
4. Edit the **pushButton** method to look like this:

```
method pushButton(var eventInfo Event)
   var
      custRpt Report
   endVar

   if custRpt.open("CUSTOMER") then
      custRpt.print()
   else
      msgInfo("Problem", "Couldn't open the report.")
   endIf
endmethod
```

5. Check your syntax, and correct any errors.

6. Run the form, and click *printButton*. Paradox loads the report from disk and then displays the Print File dialog box. Use this dialog box to set print specifications, such as a range of pages and how to handle overflows, and then click OK to send the report to the printer.
7. Close the report.

8. Return to the Form Design window, and choose File|Save to save the form.

How it works

The custom code in this method is organized into two blocks.

Block 1 The first block declares a Report variable named *custRpt*. Like a Table variable (discussed in Example 6-3), a Report variable acts as a handle to a report file stored on disk.

Block 2 The second block is

```
if custRpt.open("CUSTOMER") then
   custRpt.print()
else
   msgInfo("Problem", "Couldn't open the report.")
endIf
```

The first statement in this block tries to read the report file from disk. Paradox knows to look for a report file because you declared *custRpt* to be a Report variable. By default, Paradox looks first for a file named CUSTOMER.RSL; if no such file is found, it looks for CUSTOMER.RDL. If it finds either file, Paradox tries to open the report. If it succeeds, the variable *custRpt* becomes a handle to the report, and the second statement, **custRpt.print()**, prints the report. If, for any reason, Paradox is unable to open the report file, the code displays a dialog box to inform you of the problem. This is the basic ObjectPAL error-checking technique.

Summary

Lessons in this chapter showed you how to

- Use a **view** dialog box to get user input
- Use **locate** to search for a value in a table
- Use **beep**, **message**, and **sleep** to convey information to the user
- Put a form into Edit mode
- Insert a new, empty record
- Get the largest value in a column of a table
- Assign a value to a field
- Print a report

CHAPTER 7

Validating data entry

The lessons in this chapter present techniques for making sure users enter valid data. They show how to

- Use the validity checks built into Paradox
- Use ObjectPAL to ensure field validity
- Use ObjectPAL to ensure record validity

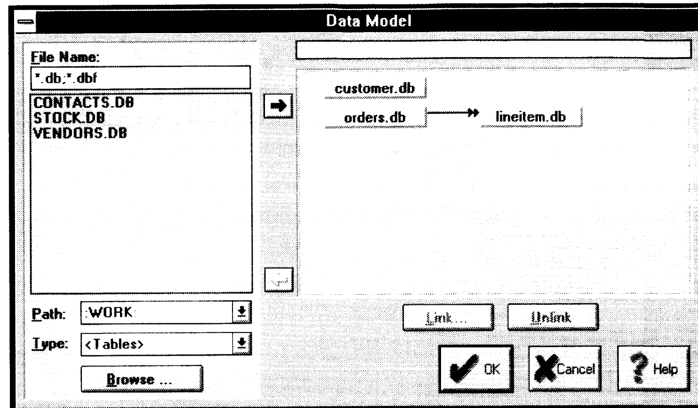
The first few lessons use a new form. Example 7-1 describes how to create it.

Creating a multi-table form

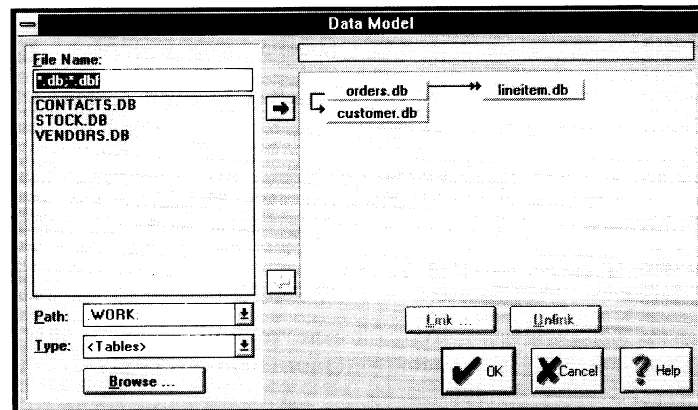
In this lesson you will create a multi-table form (a form that displays data from more than one table).

Example 7-1 Creating a multi-table form

1. Choose File|New|Form to display the Data Model dialog box.
2. Add the *Customer*, *Lineitem*, and *Orders* tables to the data model.
3. In this form, *Orders* is the master table. Link *Orders* to *Lineitem* as shown:



4. Next, link *Orders* to *Customer* as shown in the following figure.

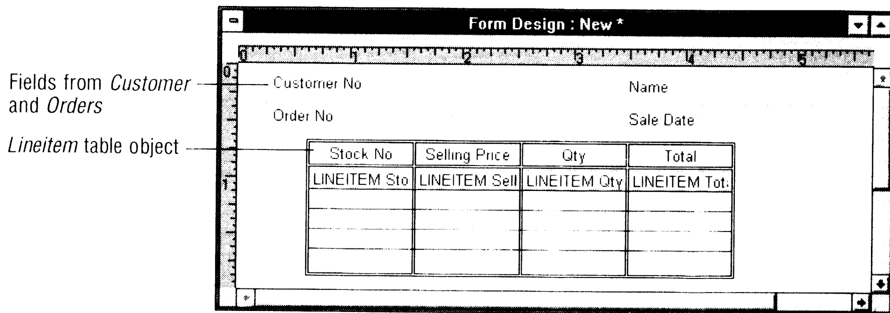


5. Choose OK to accept this data model and display the Design Layout dialog box.

6. In the Design Layout dialog box, uncheck the Fields Before Tables option in the Object Layout panel. This makes it easier to see all the objects in this form.

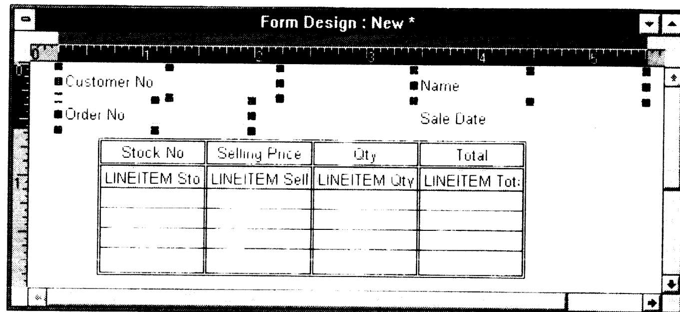
7. Choose OK to accept the layout and display the new form in a design window.

8. This application doesn't use all the fields in these tables. It uses the table frame bound to *Lineitem*, the *Name* field object bound to *Customer*, and the *Order_No*, *Customer_No*, and *Sale_Date* field objects, all bound to the *Orders* table. Delete the other field objects, and arrange the remaining objects to make your form look like this:



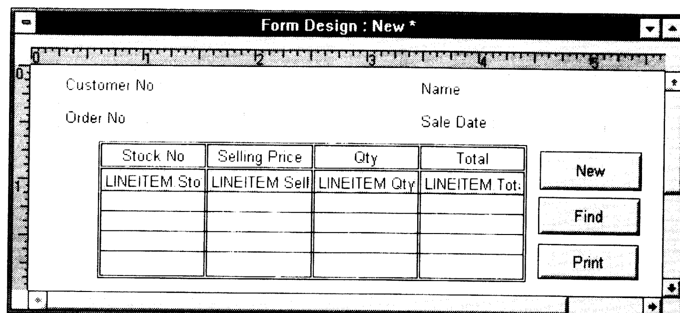
The Tab Stop property

9. ***Shift***+click the following field objects: *Customer_No*, *Order_No*, and *Name*. When you have all three field objects selected (as shown in the next figure), inspect any one of them to view their properties. Choose Run Time|Tab Stop to uncheck the Tab Stop property. By doing this, you set the Tab Stop property for all three fields at once.



Unchecking the Tab Stop property prevents the user from moving the cursor into any of these fields. This gives you more control over the application, because it keeps the user from entering spurious data into these fields. Don't change the Tab Stop property of *Sale_Date*, because you'll need to enter data into it in the next lesson.

10. Next, use the Button tool to add three buttons. Place them in the form and label them *New*, *Find*, and *Print*, as shown next:



11. Choose File|Save to save the form. Name it ORDERS.FSL. Throughout the following lessons, this form will be referred to as the *Orders* form.

Using built-in validity checks

Paradox's built-in validity checking functions are powerful. Learn to use them interactively, as described in the *User's Guide*—you can significantly reduce the amount of code you have to write to build a solid application. Following is a very simple example that just scratches the surface.

Example 7-2 Built-in validity checking

1. When the *Orders* table was created, *Sale_Date* was defined to be a Date field. Therefore, Paradox will reject any value that is not a valid date from January 1, 100 to December 31, 9999. To see this kind of validity checking in action, run the *Orders* form.
2. Press *F9* to enter Edit mode.
3. Move to the *Sale_Date* field object.
4. Enter **abc**.
5. That's not a valid date, so the form displays an error message in the status bar, as shown in the next figure. This happens because Paradox's built-in validity check encountered an error.

Paradox for Windows
File Edit Form Record Properties Window Help
Form : ORDERS.FSL [Data Entry]

Customer No : 1221 Name : Kauai Dive Shoppe
Order No : 1001 Sale Date : abcdefg

Stock No	Selling Price	Qty	Total
1313	$250.00	4	$1,000.00
3340	$395.00	16	$6,320.00

New Find Print

Bad Month Specification Edit Locked

6. Choose Edit|Undo to restore the original (valid) date.

Adding validity checks with ObjectPAL

Powerful as Paradox's built-in validity checks are, there will be times when you need more control. For example, it may not be enough that the value of the *Sale_Date* field object be a valid date.

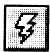
Suppose you want to prevent the user from entering a date in the future; that is, whenever the user enters a date, you want to make sure it's not later than the current date. The following example shows you how to perform more specific validity checks.

Example 7-3 ObjectPAL validity checking

1. Return to the design window.
2. Inspect the *Sale_Date* field object, and choose Methods to display the Methods dialog box.
3. Double-click **canDepart** to open an Editor window for the built-in **canDepart** method.
4. Edit the method to look like this:

```
method canDepart(var eventInfo MoveEvent)
   if Self.Value > today() then
      eventInfo.setErrorCode(CanNotDepart)
      message("Sale Date can't be later than today's date.")
      sleep(1000)
   endIf
endmethod
```

5. Check your syntax, and correct any errors.



6. Run the form and enter Edit mode.
7. Move to *Sale_Date*, type in a future date, and press ***Enter*** .
8. Look for the message in the status bar.
9. Choose Edit|Undo to restore the original date.

How it works

Every object has a built-in method named **canDepart**. In effect, **canDepart** asks for permission to move the cursor off of the object. By attaching custom code to a field object's **canDepart** method, you can prevent the user from leaving the field object until it contains a valid value.

Line 2 The second line of the method is

```
if Self.Value > today() then
```

The main elements in this statement are the object variable *Self*, the Value property, and the run-time library procedure **today**. In this

example, the field object *Sale_Date* is executing the code, so *Self* refers to *Sale_Date*.

today returns the current date, according to your computer's internal clock.

Taken as a whole, this statement compares the value stored in *Sale_Date* with the current date. If *Sale_Date* is greater, the next line of custom code executes; otherwise, execution skips to the end of the method.

Line 3 The third line is

```
eventInfo.setErrorCode(CanNotDepart)
```

This statement uses the **setErrorCode** method defined for the MoveEvent type to store information in the variable *eventInfo*. In this statement, **setErrorCode** uses the ObjectPAL constant CanNotDepart to indicate an error—here, the error is a date in the future, and the CanNotDepart constant stores information in *eventInfo*. When it's stored in *eventInfo*, this information is available to Paradox, and Paradox can respond to it. The response in this case is to prevent the cursor from leaving the field object until the user enters a value that meets the specified criteria.

Important The basic technique for announcing an error condition is to use **setErrorCode** and an error constant. Doing so adds information to the *eventInfo* variable, which Paradox can then respond to. For a complete list of ObjectPAL constants, refer to Appendix C in the *ObjectPAL Reference*.

Lines 4 and 5 The fourth and fifth lines are

```
message("Sale Date can't be later than today's date.")
sleep(1000)
```

The **message** statement displays an error message in the status bar, and the **sleep** statement causes a delay so you have time to read it.

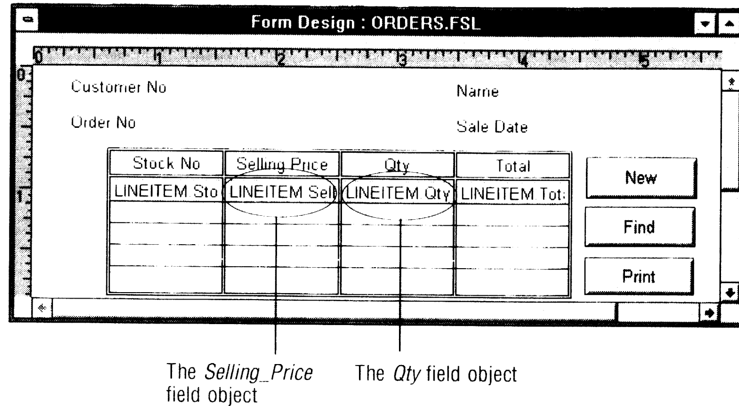
Supplying values

Sometimes, the best way to get valid data is to provide it yourself. In the previous chapter, Example 6-3 showed how to generate a unique order number and put it into a field object, thus relieving the user of that responsibility. Another way to make the user's life easier is to use ObjectPAL to perform calculations whenever possible. This lesson shows one approach.

As shown in Figure 7-1 and Example 7-4, the *Orders* form contains a table frame bound to *Lineitem*. It contains records consisting of the following field objects: *Stock No*, *Selling Price*, *Qty*, and *Total*. For each

line item, the value of *Total* is the product of the values of *Selling_Price* and *Qty*.

Figure 7-1 The *Orders* form



The following steps show how to use ObjectPAL to perform this calculation.

Example 7-4 Performing calculations

1 Return to the design window.

2 Inspect *Selling_Price*, and choose Methods to open the Methods dialog box.

3 Double-click **changeValue** to open an Editor window for the built-in **changeValue** method.

4 Edit the method to look like this:

```
method changeValue(var eventInfo ValueEvent)
   doDefault
   if not Qty.isBlank() then
      Total.Value = Self.Value * Qty.Value
   endIf
endmethod
```

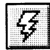
5 Inspect *Qty*, and choose Methods to open the Methods dialog box.

6 Double-click **changeValue** to open an Editor window for the built-in **changeValue** method.

7. Edit the method to look like this:

```
method changeValue(var eventInfo ValueEvent)
   doDefault
   if not Selling_Price.isBlank() then
      Total.Value = Self.Value * Selling_Price.Value
   endIf
endmethod
```

8. Check your syntax, and correct any errors.



9. Run the form, and enter Edit mode.

10. Enter different values into *Selling_Price* and *Qty* to see that ObjectPAL is doing the calculation.

11. Return to the Form Design window, and save the form.

How it works

This example uses code attached to two objects: *Selling_Price* and *Qty*. The code is attached to each object's built-in **changeValue** method.

Field objects have a built-in method named **changeValue**. In effect, **changeValue** asks for permission to post the value of the field object to the underlying table. By attaching custom code to a field object's built-in **changeValue** method, you can specify how to respond when the user changes its value. The code attached to these objects is almost identical. The following discussion analyzes the code attached to *Selling_Price*, then discusses differences in the code attached to *Qty*.

Line 2 The second line of the method is

```
doDefault
```

A call to **doDefault** executes the default code for this built-in method immediately, instead of waiting until the end of the method. In the case of **changeValue**, calling **doDefault** gives you access to the updated value of the field object. For example, if you're entering a selling price for a new record, ObjectPAL doesn't have access to that value until the default **changeValue** code executes. Or suppose you're editing an existing order, and you change the existing selling price from $12.95 to $14.99. Until the default **changeValue** code executes, ObjectPAL will use the old price.

You can watch this happen by attaching the following code to *Selling_Price*'s **changeValue** method:

```
method changeValue(var eventInfo ValueEvent)
   msgInfo("Before calling doDefault", Self.Value)
   doDefault
   msgInfo("After calling doDefault", Self.Value)
endmethod
```

Run the form, switch to Edit mode, and enter a value into *Selling_Price*. Dialog boxes display *Selling_Price*'s value before and

after calling **doDefault**. Enter another value and watch the dialog boxes display the old value first and then the changed value.

Line 3 The third line calls the run-time library method **isBlank** to operate on *Qty*. **isBlank** returns True if the field object is blank (empty); if the field object has a value, **isBlank** returns False. In this example, there's no point in doing the calculation if *Qty* is blank, so execution skips to the end of the method.

Line 4 The fourth line is

```
Total.Value = Self.Value * Qty.Value
```

This code does the calculation. It multiplies the value of *Selling_Price* (represented by the object variable *Self*) by the value of *Qty*, and assigns the result to the Value property of *Total*.

The code attached to *Qty*'s built-in **changeValue** is like a mirror image of the code just discussed, with the following differences:

- It checks to see if *Selling_Price* is blank before performing the calculation.
- The object variable *Self* refers to *Qty*. Remember, *Self* refers to the object to which the currently executing code is attached.

Handling key violations

Previous lessons have shown how to check validity at the field level (that is, how to check the values of individual fields). This lesson presents techniques for record-level validation. Specifically, it shows how to catch key violations.

There are two parts to this lesson. First, you'll work with the *NewCust* form to learn how to catch key violations in a single-record form. Then you'll work with the *Orders* form to do the same thing on a multi-table form.

Single-record forms

In previous lessons, you worked with individual objects contained in a form. This lesson introduces the form as a design object. Here, you'll see how the form manages the objects it contains and how it oversees events and actions.

Example 7-5 A form as manager

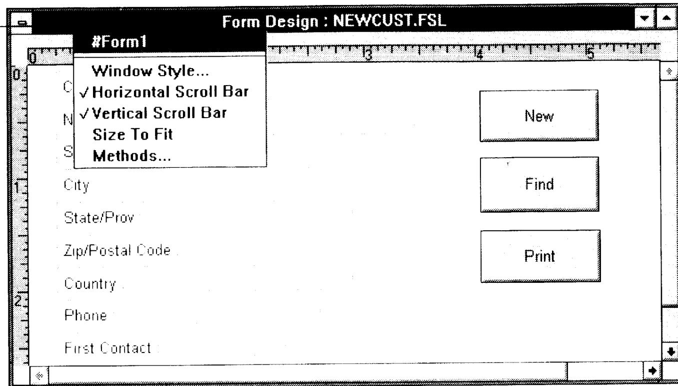
First, open *NewCust* in the design window, and work through the following steps.

1. Choose Properties|Form|Methods to open the Methods dialog box. This dialog box lists the form's built-in methods.

Shortcut

You can inspect a form by right-clicking the form window's title bar or by pressing *Esc* until all other objects are deselected and then pressing *F6*.

You can inspect a form by right-clicking the form window's title bar



2. Double-click **action** to open an Editor window for the form's built-in **action** method.

   Built-in methods at the form level have some additional default code in the window. This code is provided for advanced ObjectPAL programmers and is discussed in the *ObjectPAL Developer's Guide*; don't worry about it now.

3. Edit the method to make it look like this:

```
method action(var eventInfo ActionEvent)
   if eventInfo.isPreFilter() then
      ; code here executes for every object in the form

   else
      ; code here executes just for the form itself
      if eventInfo.id() = DataUnlockRecord or
         eventInfo.id() = DataPostRecord then
         doDefault
         if errorCode() = peKeyViol then
            msgInfo("Problem", "Enter a different Customer No.")
         endIf
      endIf
   endIf
endmethod
```

4. Check your syntax and correct any errors.

5. Run the form, and enter Edit mode (this locks the record).

6. Edit the first record as follows: Change the value of *Customer_No* to **1351**. Because this number is already in the table, using it here will cause a key violation and trigger an error when you do the next step.
7. Press ***F9*** to end Edit mode (and unlock the record). Your code responds to the key violation error: a dialog box opens and displays a message telling you to enter a different customer number. The form does not exit Edit mode.
8. Choose OK to close the dialog box.
9. Press ***Ctrl+F5*** to post the record (without unlocking it).
10. The dialog box opens again. Choose OK to close it.
11. Choose Record|Cancel Changes (or press ***Alt+Backspace***) to restore the field object's original value.

12. Return to the Form Design window, and then choose File|Save to save the form.

How it works

What you've just seen is the form acting as a manager. As you interact with objects in the form, you generate events and initiate actions. These go to the form first, and the form decides what to do with them. In this example, the form tests for two specific actions: DataUnlockRecord and DataPostRecord. When it receives one of these actions, the form checks for a key violation and informs you if one occurs. In the context of this lesson, the interesting code begins with line 7.

Lines 7 and 8

Lines 7 and 8 are

```
if eventInfo.id() = DataUnlockRecord or
   eventInfo.id() = DataPostRecord then
```

This is really just one statement broken into two lines for readability. It calls the **id** method to identify the action. If the action is either DataUnlockRecord or DataPostRecord, subsequent lines execute to test for a key violation.

A DataUnlockRecord occurs when you try to unlock and post a record for any reason: ending edit mode, moving to the next or previous record, inserting a record, deleting a record, and so on.

A DataPostRecord occurs whenever you try to post (commit) a record to the underlying table and still keep the record locked.

Line 9

Line 9 is

```
doDefault
```

As explained in a previous lesson, **doDefault** executes the default code for a built-in method. In this example, it executes the default

code for unlocking a record or posting a record. If it fails for any reason, it returns an error code to tell you what happened.

Line 10 Line 10 is

```
if errorCode() = peKeyViol then
```

This statement uses the run-time library procedure **errorCode** to test the error code returned by **doDefault**. It uses the ObjectPAL error constant peKeyViol to test for a specific error: a key violation. As it does for actions, ObjectPAL provides constants for common error conditions.

Line 11 Line 11 is

```
msgInfo("Problem", "Enter a different Customer No.")
```

This statement displays a dialog box telling you to enter a different customer number. Why? Because the *Customer* table has only one key field, Customer No. So, if there's a key violation, it must be because of a duplicate customer number.

Summary This example showed how to trap for key violations on a single-record form and introduced the form as a design object that manages events and actions for the objects it contains. Everything goes to the form first, so the form can respond to actions on behalf of other objects.

This example also showed you how to

- Use **errorCode** to get information about errors
- Use error constants to identify specific errors

Multi-table forms

Example 7-6 shows how to catch key violations in the detail set of a multi-table form. Use the *Orders* form to work through it. As shown previously in Example 7-4, the *Orders* form contains field objects and a table frame. Example 7-5 showed how to catch key violations on a single-record form; you can use the same technique to catch key violations on the field objects in a multi-table form too.

However, in the *Orders* form, the field objects have their Tab Stop property turned off, which prevents the user from moving the cursor into them. Because you can't move the cursor into these field objects, you can't edit them either, so these field objects can never cause a key violation.

You can edit the field objects in the table frame, though, so this form needs a mechanism to handle key violations at that level.

The closer-is-better principle

You could use the technique presented in the first part of this lesson—that is, you could attach code to the form's built-in **action** method and test for an error after a DataUnlockRecord or a DataPostRecord action. However, as a general principle, it's a good idea to keep code *as close as possible* to the object it operates on. Doing so makes your code modular, object-oriented, and easy to maintain and reuse.

In a single-record form like *NewCust*, the only place to handle key violations is on the form. In a multi-table form like *Orders*, you have a choice: attach the code to the form, or attach it to the table frame. If you think of the form as the manager of all the objects it contains, you can then think of a table frame as a second-level manager: it only manages the field and record objects it contains. The form sees every event and action for every object in the form; the table frame only sees the events and actions for the objects it contains. Applying the closer-is-better principle, the best place to attach the code is to the table frame.

Example 7-6 Handling key violations in a multi-table form

1. Open the *Orders* form in a design window, and inspect the *LINEITEM* table frame.
2. Choose Methods, and then choose **action** to open an Editor window for the table frame's built-in **action** method.
3. Edit the method to make it look like this:

```
method action(var eventInfo ActionEvent)
   if eventInfo.id() = DataUnlockRecord or
      eventInfo.id() = DataPostRecord then
      doDefault
      if errorCode() = peKeyViol then
         msgInfo("Problem", "Enter a different Stock No.")
      endIf
   endIf
endmethod
```

4. Check your syntax, and correct any errors.

5. Run the form, and enter Edit mode (locking the record).
6. Move the cursor to the second record in the table frame. Change the value of *Stock No* to **1313** (making the stock number for the second record the same as the stock number for the first record). This causes a key violation.
7. Press *F11* to move to the first record (and unlock the second record). A dialog box opens and displays a message telling you to enter a different stock number. The cursor does not move to the previous record. (You could have pressed *F9*, as in the previous example, and seen the same results.)
8. Press *Enter* to close the dialog box.

9. Press *Ctrl+F5* to post the record without unlocking it.
10. The dialog box opens again. Press *Enter* to close it.
11. Press *Alt+Backspace* (or choose Record|Cancel Changes) to restore the field object's original value.

12. Return to the Form Design window, and choose File|Save to save the form.

How it works

This code is identical to the code in Example 7-5, with two exceptions:

- It doesn't include the default text provided for form-level built-in methods.
- The dialog box tells you to enter a different stock number rather than a different customer number.

Everything else works exactly the same; it's just happening at a local level because the code is attached to the table frame instead of the form.

Summary

This chapter presented techniques for record-level validity checking. The lessons showed you how to

- Respond to the actions that can cause a key violation
- Catch key violations in single-record forms
- Catch key violations in multi-table forms
- Use the form to manage the objects it contains
- Use a table frame as a second-level manager to manage only the field and record objects it contains
- Apply the closer-is-better principle to decide where to attach your code

CHAPTER 8

Controlling another form

The lessons in this chapter show you how to use ObjectPAL to control one form from another form. The examples present techniques for using a form as a dialog box, but you can apply what you learn to any multi-form application. In these lessons, the *Orders* form is the *calling form;* that is, the *Orders* form calls (opens) a second form (the dialog box)—which in these lessons is the *Cust* form.

Example 8-1 and Example 8-2 show you how to make a form into a dialog box; Example 8-3 shows you how to call and manage a dialog box.

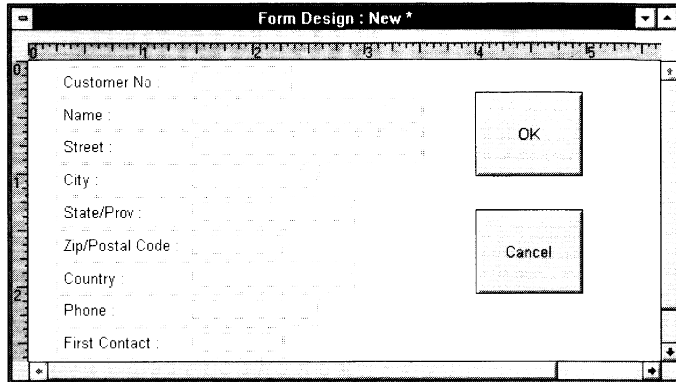
Designing a dialog box

A dialog box is just a form with a few special properties set. To design a dialog box, simply design a form, and then set the appropriate properties, as shown in the following examples.

Example 8-1 Designing a dialog box

1. Choose File|New|Form to open the Data Model dialog box.
2. Choose CUSTOMER.DB from the list of tables, and add it to the data model.
3. Click OK to close the Data Model dialog box. The Design Layout dialog box opens.
4. Click OK to accept the default layout.

5. Use the Button tool to place two buttons, as shown in the following figure:

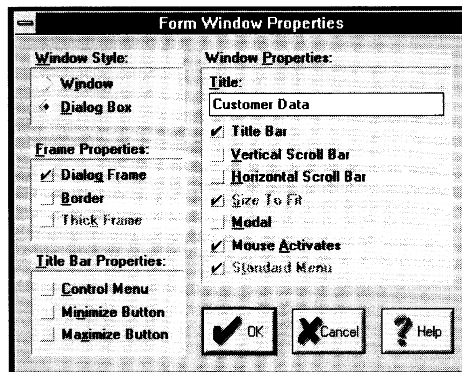


6. Label one button *OK*, and change its name to *okButton*.
7. Label the other button *Cancel*, and change its name to *cancelButton*.
8. Save the form and name it CUST.FSL.

So far, you haven't done anything different than you would to design an ordinary form. Example 8-2 shows you how to set this form's properties to make it look and behave like a dialog box.

Example 8-2 Setting a form's properties

1. Choose Properties|Form|Window Style to open the Form Window Properties dialog box (shown in the following figure).



2. Check and uncheck boxes as necessary to make the dialog box match the one shown above. It's important to set properties exactly as shown here; otherwise, the dialog won't behave as expected.

3. Choose OK to close the dialog box. You won't see any changes to the form; they take effect after you run the form.
4. Choose File|Save to save these changes.

   That's all there is to it. You're finished designing the dialog box. The next step is to attach code to the buttons.
5. Attach the following code to *okButton*'s built-in **pushButton** method.

```
method pushButton(var eventInfo Event)
   formReturn("OK")
endmethod
```

6. Attach the following code to *cancelButton*'s built-in **pushButton** method.

```
method pushButton(var eventInfo Event)
   formReturn("Cancel")
endmethod
```

   Both methods do the same thing: return a value and program control to the calling form. They're discussed in more detail under the "How it works" section in the next lesson.
7. Close the form and save your changes.

Managing a dialog box

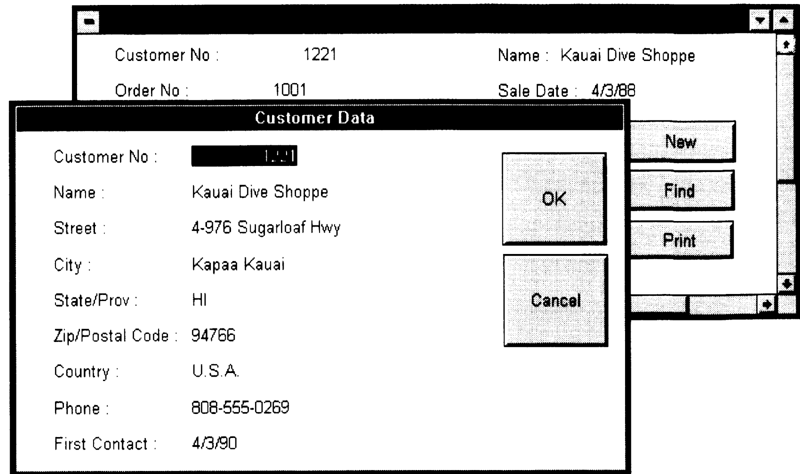
This lesson describes how to manage a dialog box (or any other form you want to control using ObjectPAL). It shows how to

- ❒ Open and display the dialog box
- ❒ Get a value from the dialog box
- ❒ Close the dialog box

This lesson uses the *Orders* form as the calling form; it will call the *Cust* form. The code that calls the dialog box is attached to the button labeled *New*. When you press the *New* button, this code inserts a new, empty record, generates a unique order number, then opens the *Cust* form. This lesson also uses the *Cust* form to enter or retrieve data for the customer and to return values to the calling form.

Figure 8-1 shows the two forms in action.

Figure 8-1 *Orders* as the calling form



The first step is to write the code that calls the dialog box. In this lesson, the code is attached to the *New* button in the *Orders* form. Example 8-3 walks you through the steps.

Example 8-3 Managing a dialog box

1. Open the *Orders* form in the design window.
2. Inspect the button labeled *New*, and change its name to *newButton*.
3. Inspect *newButton*, and choose Methods to display the Methods dialog box.
4. Double-click **pushButton** to open an Editor window for the built-in **pushButton** method.
5. Edit the method to look like this:

```
method pushButton(var eventInfo Event)
   var
      newCustNum Number
      ordersTbl Table
      newCustDlg Form
      dlgVal String
   endVar

   action(DataBeginEdit)
   action(DataInsertRecord)
   ordersTbl.attach("ORDERS.DB")
   Order_No.Value = ordersTbl.cMax("Order No") + 1
   action(DataPostRecord)
```

```
   if newCustDlg.open("cust") then
      dlgVal = newCustDlg.wait()
      if dlgVal = "OK" then
         Customer_No.Value = newCustDlg.Customer_No.Value
      endIf
      newCustDlg.close()
   else
      msgInfo("Problem", "Couldn't open the dialog box.")
   endIf
endmethod
```

6. Save the form and run it.

7. Click *newButton* to insert a new record, generate a new order number, and open the dialog box.

8. Use the dialog box to enter data for a new customer or find data for an existing customer. When you're finished, click the OK button.

Note You have to use the keyboard to scroll through records. Dialog boxes don't respond to Paradox's menus or SpeedBars.

9. After the dialog box closes, verify that *Customer_No* contains a new value.

How it works

The code is organized into three blocks. The first block declares variables, the second block inserts a new record with a unique order number (using the technique described in Example 6-3), and the third block manages the dialog box.

Line 5 In the first block, line 5 is

```
newCustDlg Form
```

This line declares the variable *newCustDlg* to be of type Form. Like the Table and Report variables described in previous lessons, a Form variable provides a handle. Here, *newCustDlg* is a handle to the dialog form whose file name is CUST.FSL.

Line 15 Line 15, the first line in the third code block, reads

```
if newCustDlg.open("cust") then
```

This statement tries to load CUST.FSL (or CUST.FDL, if CUST.FSL is not found) from disk and run the form. If it succeeds, the next line of code executes. If it fails for any reason, execution skips to the **else** clause, and a dialog box tells you about the problem.

Line 16 Line 16, the next line in the third block, reads

```
dlgVal = newCustDlg.wait()
```

This statement says, in effect, "Suspend execution of this method, and wait for *newCustDlg* to return a value. Then assign the value to the variable *dlgVal,* and resume execution."

A **wait** statement gives control to the specified form (in this case, it's the dialog box represented by *newCustDlg*). While the calling form is waiting on the called form, only the called form will respond to events. In other words, the called form is modal.

Important How does the called form return control? The answer is a **formReturn** statement. In Example 8-2, you attached the following code to the built-in **pushButton** method of the OK button in the dialog form.

```
method pushButton(var eventInfo Event)
   formReturn("OK")
endmethod
```

This code returns control *and* a value of "OK" to the calling form.

Lines 17 through 19 Lines 17 through 19, in the third block, read

```
if dlgVal = "OK" then
      Customer_No.Value = newCustDlg.Customer_No.Value
endIf
```

These lines test the value of *dlgVal*, the value returned by the dialog box. If *dlgVal* is "OK," it means the user clicked the OK button in the dialog box. The following statement gets the value of the *Customer_No* field object in the *newCustDlg* form and assigns it to the Value property of *Customer_No*, a field object in the calling form.

Important The following pseudocode shows how to get a value from an object in another form.

```
objVal = formVar.objectName.Value
```

In this example, *objVal* is a variable that stores the value, *formVar* is a Form variable (a handle to the other form), *objectName* represents the name of the object you're intrested in, and Value specifies the Value property.

Line 20 Line 20 is

```
newCustDlg.close()
```

This statement closes the dialog box and removes it from the display. Without a **close** statement, you'd open a new copy of the dialog each time this method executed.

Summary

The lessons in this chapter introduced the basic techniques for managing a multi-form application. They showed you how to

- Create a dialog box by setting special form properties
- Call one form from another form
- Use a **wait** statement to suspend execution in the calling form and wait for the called form to return a value
- Get values from the called form
- Close the called form

CHAPTER 9

Working outside the data model

This chapter is for programmers who want to sample one of the more advanced features of ObjectPAL. It shows how to work with tables that aren't included in a form's data model—without displaying them.

Using the *Orders* form as an example, when you place an order for a number of items, you need to be sure there are enough of those items in stock. If there are enough, then you need to subtract the quantity you ordered from the quantity in stock. You could add the *Stock* table to the form's data model, place the appropriate field objects in the form, and work with them directly. However, you may face situations in which you don't want to do this—perhaps to keep the form simple and uncluttered or to prevent the casual user from gaining access to sensitive data. In such situations, an excellent alternative approach is to use a TCursor.

What is a TCursor?

A TCursor is a pointer to the data in a table, a pointer that enables you to manipulate data at the table level, record level, and field level without having to display the table. When you use a TCursor, you aren't working with a clone or a copy of the table; editing the records in a TCursor changes the underlying table, and any locks on the table affect the TCursor.

Important The SpeedBar has no tool for creating a TCursor as it does for creating a table frame. A TCursor is purely a programming construct; in fact, it is ObjectPAL's principal construct for working with tables.

The relationship of a TCursor to a table is like that of a text cursor to a word-processor document. In a word processor, the text cursor points to one letter at a time, can move anywhere in the document, and specifies where editing takes place. Similarly, when you open a TCursor onto a table, the TCursor points to the current record, can move to any record in the table, and specifies which record to edit. In

addition, you can use a TCursor to perform many table-level operations.

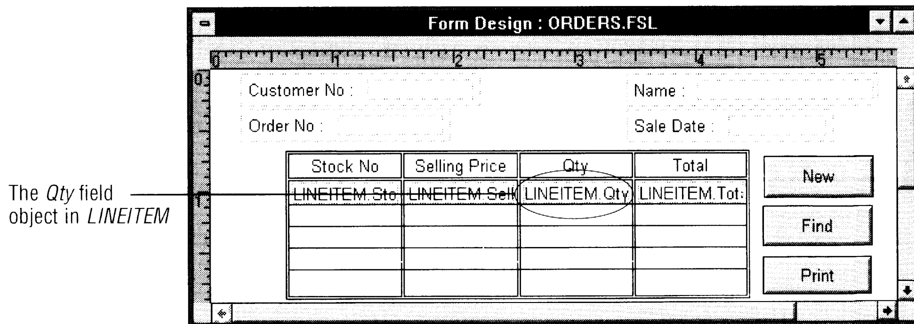
By declaring a TCursor variable and making it point to a table, you can use the TCursor to edit the table without actually displaying the table. Using a TCursor to edit a table is like using a remote control to change channels on a television. When you press a button on the remote control, the television changes channels. When you edit a record in a TCursor, the record in the underlying table changes.

Using a TCursor

The following examples show how to use a TCursor to maintain the *Stock* table from within the *Orders* form. You can use the *Orders* form you created in a previous lesson. This lesson uses the same basic validity checking technique presented in Example 7-3: code attached to a field object's built-in **canDepart** method checks the object's Value property and keeps the cursor on the field until the user enters an acceptable value.

Figure 9-1 shows where to attach the code.

Figure 9-1 Attaching code to the *Qty* field object in *LINEITEM*



The *Qty* field object in *LINEITEM*

Example 9-1 Using a TCursor

1. In the *LINEITEM* table frame, inspect the field object named *Qty*, and choose Methods to open the Methods dialog box.
2. Double-click **canDepart** to open an Editor window for the built-in **canDepart** method.
3. Edit the method to look like this:

```
method canDepart(var eventInfo MoveEvent)
   var
      stockTC TCursor
      qtyOnHand, qtyOrdered Number
   endVar
```

```
    stockTC.open("STOCK.DB")
    stockTC.locate("Stock No", Stock_No.Value)
    qtyOnHand = stockTC.Qty
    qtyOrdered = Self.Value

    if qtyOrdered < qtyOnHand then
       qtyOnHand = qtyOnHand - qtyOrdered
       stockTC.edit()
       stockTC.Qty = qtyOnHand
       stockTC.endEdit()
    else
       msgInfo("Not enough in stock",
               "Only " + String(qtyOnHand) + " on hand.")
       eventInfo.setErrorCode(CanNotDepart)
    endIf
endmethod
```

4. Check your syntax, and correct any errors.

5. Run the form, and then press *F9* to enter Edit mode.
6. Move to the *Qty* field in the *LINEITEM* table frame, and enter a large number for the quantity (a number greater than 100 should do it). The dialog box will open and tell you to enter a smaller quantity.

How it works

The code in this method is organized into three blocks. The first block declares variables, the second block opens a TCursor onto the *Stock* table and reads the value of the *Qty* field, and the third block updates the *Stock* table or displays a message in a dialog box, depending on how many items are in stock.

Line 7 The seventh line is

```
stockTC.open("STOCK.DB")
```

This statement opens the TCursor *stockTC* onto the *Stock* table. Now this method can use *stockTC* to work with data in the *Stock* table. By default, when you open a TCursor, it points to the first record in the underlying table.

Lines 8 and 9 The eighth and ninth lines are

```
stockTC.locate("Stock No", Stock_No.Value)
qtyOnHand = stockTC.Qty
```

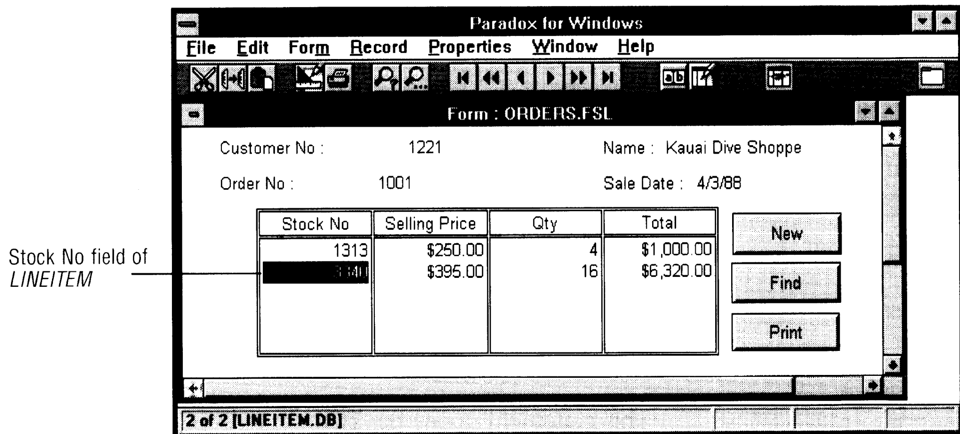
Line 8 uses the **locate** method to search the *Stock* table for the stock number in the current record of the *LINEITEM* table frame. The **locate** method you use on a TCursor is not the same **locate** you use to search a table frame, but they behave the same way. As described in Example 6-2 of Chapter 6, when **locate** searches a table frame and succeeds, the form moves to that record and displays it in the table frame. Similarly, when **locate** searches the TCursor's table and succeeds, the TCursor moves to the record where the value was found.

For example, suppose you're ordering 10 units of the item whose stock number is 3340, as shown in Figure 9-2. In this case, the **locate** method would search the *Stock* table for a value of 3340 in the Stock No field.

When **locate** finds the value, the TCursor moves to point to that record. (If, for example, **locate** finds the value 3340 at record 30 of the *Stock* table, *stockTC* would move to point to record 30.)

Important As the TCursor moves around in the underlying table, it *does not* affect the current record displayed in the form. For example, in Figure 9-2, the form displays record 2 of 2 in the table frame. It will continue to display this record, regardless of which record the TCursor points to.

Figure 9-2 Record displayed during TCursor **locate**



Next, because the **locate** method has succeeded and *stockTC* has moved to the appropriate record, line 9 returns the available quantity of the specified stock number. In other words, building on the previous example, if **locate** finds the stock number 3340 at record 30, line 9 returns the value of the Qty field for record 30 and assigns it to the variable *qtyOnHand*.

Line 10 The 10th line is

```
qtyOrdered = Self.Value
```

Strictly speaking, this line is not necessary; it's included to make the code that follows easier to read. This line gets the value of the *Qty* field object in the current record in the form and stores it in the variable *qtyOrdered*.

Lines 12 and 13

Lines 12 and 13 mark the beginning of the third code block. They are

```
if qtyOrdered < qtyOnHand then
   qtyOnHand = qtyOnHand - qtyOrdered
```

These lines compare the amount ordered with the amount in stock. If there's enough in stock, subsequent lines update the *Stock* table; if not, execution skips to the **else** clause to display a dialog box and to prevent the insertion point from leaving the field object.

Lines 14 through 16

Lines 14 through 16 are

```
stockTC.edit()
stockTC.Qty = qtyOnHand
stockTC.endEdit()
```

Line 14 puts *stockTC* (and, by extension, the *Stock* table) into Edit mode. Line 15 assigns the updated value of *qtyOnHand* to the Qty field of the current record of *stockTC*. Line 16 takes *stockTC* out of Edit mode.

Summary

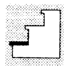
A TCursor is a powerful ObjectPAL construct. Using a TCursor, you can work with the data in a table without displaying the table. This chapter showed you how to

- Manipulate data in a table that isn't included in the form's data model
- Open a TCursor onto a table
- Use a TCursor to search for a value in a table
- Use a TCursor to edit the underlying table

CHAPTER 10

Where do I go from here?

This tutorial has introduced the basic concepts and techniques of ObjectPAL. As you have seen, by using the power of interactive Paradox and these basic techniques, you can accomplish many fundamental database programming tasks. When you need to do more, or if you're just curious about how much ObjectPAL can *really* do, the following resources are available:



- The *ObjectPAL Developer's Guide* provides detailed discussions and examples of all aspects of building ObjectPAL applications. Passages of special interest to beginning programmers are marked with the First Step icon shown in the margin.
- The *ObjectPAL Reference* is a complete reference to the built-in methods, basic language elements, and methods and procedures in the run-time library. Entries for beginning programmers are marked with the word *Beginner*.
- Example applications

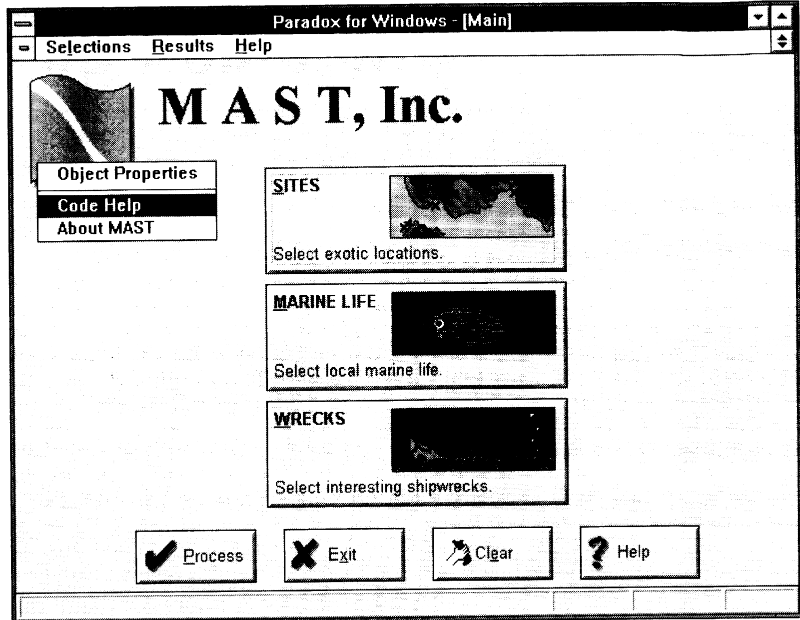
The example files include one full-scale application (the Dive Planner) and several mini-apps. All of the examples include online interactive help that explains how to use them as well as help systems that explain the ObjectPAL code attached to each object. As you run these applications, you can get help on using them by pressing *F1* or choosing an item from the Help menu in the menu bar. You can also get ObjectPAL help by inspecting (right-clicking) an object and choosing Code Help from its pop-up menu. When you choose Code Help, you open a Help window listing the methods and procedures attached to the object you inspected, as shown in Figure 10-1. Then, in the Help window, you can choose an item that interests you and display the actual code and text that explains the code.

Note The Dive Planner application is a learning tool. It demonstrates many approaches and techniques for using ObjectPAL, but it is not intended to demonstrate a real-world application.

Figure 10-1 Using the Help system in the example applications

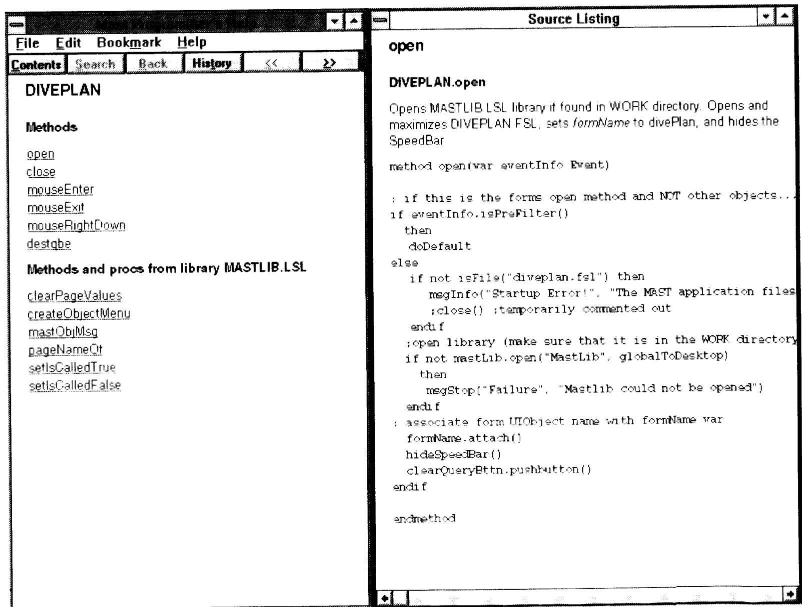
At any time while you're running an example application, you can inspect an object to display a pop-up menu. Choose Code Help to open a Help window showing the ObjectPAL code attached to that object.

(In the Dive Planner, the dive flag represents the form.)



When you inspect an object and choose Code Help, a Help window opens showing all the ObjectPAL code attached to that object. In this figure, the window on the left lists the code attached to the form.

Choose a method from this list to open a second window containing the source code for the method and text that explains it. In this figure, the window on the right explains the form's built-in **open** method.



INDEX

< > (comparison operators) 48, 59, 81
_ (underscore), in object names 41, 48

A

access rights 6
action constants 35, 38, 39, 51
action method
  attaching code to 37, 40
  built-in 37
  inserting records with 51
  simulating actions with 35
  table frames and 67
  UIObject type 35
  validity checking and 64
actions
  See also events
  initiating 31, 35
  responding to 31, 36, 39, 40
Advanced Level 23
aliases 2
Alternate Editor command 18, 23
animation, creating 1
applications
  creating 7, 31
  multi-form 69
  multi-table 55, 66
  sample 32, 83
arguments, defined 29
arrays 12
attach method 51

B

beep procedure 49
Beginner level 20, 22
  constants for 35
blank values 63
boxes 7
Browse Sources command 22
built-in methods 9
  defined 26
  disabling 38
  displaying 27
  executing immediately 62, 65
  modifying 10, 15
Button tool 26, 33, 57, 70
buttons
  attaching code to 26, 34, 71
  clicking 26
  copying to Clipboard 28
  creating 26, 33
  inserting records with 34
  inspecting properties 26
  naming 34
  redefining 26

C

C programming language 12
C++ programming language 12
calculations 61
Cancel Changes command 65
canDepart method
  editing 59, 78
  validity checking and 59
changeValue method 10
  editing 61
Check Syntax command 18, 28
choice lists 7
classes
  See object types
Clipboard
  copying to 18, 28
  pasting from 18
close method, dialog boxes and 74
closer-is-better principle 67
cMax method 52
Code Help command 83
comments in code 50
comparison operators (< >) 48, 59, 81
Const window 16
constants
  action 35, 38, 39, 51
  Beginner level 35
  declaring 16
  defined 22
  Editor 36

error 60, 66
global 16
inserting in code 21
record 35
viewing 21
Constants dialog box 21
containers, code 17
containership hierarchy 7
control structures 5, 12
Copy command 18
cursor
See insertion point
custom methods 16
Cut command 18

D

data
See values
data entry, validity checking 7, 55–68
See also validity checking
data model
adding to 33
creating 55
working outside 77
Data Model dialog box 33, 55, 69
data types
specifying 45
user-defined 12, 17
DataArriveRecord constant 36, 40
DataBegin constant 35
DataBeginEdit constant 36, 51
DataCancelRecord constant 35
DataDeleteRecord constant 35
DataEnd constant 36
DataEndEdit constant 36
DataInsertRecord constant 35, 51
DataLockRecord constant 35
DataNextrecord constant 36
DataPostRecord constant 35, 39, 51, 52, 65
DataPriorRecord constant 36
DataUnlockRecord constant 35, 65
dates
checking validity 58
returning 60
syntax for 42
Debugger 24
delaying execution 60
Delete command 18
Deliver command 22
Design command 26
Design Layout dialog box 33, 69
design objects 63
Desktop
modifying 22
title of 12
Desktop Properties dialog box 22
dialog boxes
associating with Form variable 73
calling 72
closing 74
designing 69
displaying 25
displaying values in 45
entering values in 43
forms as 69
managing 71
modal 29, 74
properties of 70
returning control 74
testing data entry 48
directories
creating 13
specifying 51
working 2
disableDefault keyword 38
Display Objects and Properties dialog box 21
DLL, calling code from 17
doDefault keyword 51, 62, 65

E

Edit Data command 35
Editor 15–23
actions in 18
activating 26
alternate 23
constants for 36
mouse function in 18
navigating in 18
Editor menu 17
Editor window 17
opening 26
size of 23
text handling 17
ellipses 7
endIf keyword 39
endMethod keyword 29
endVar keyword 45
enumSource method 22
errorCode procedure 66

errors
Check Syntax command and 18
constants for 60, 66
displaying messages 23, 60
eventInfo variable and 60
returning 60
testing for 18, 28, 66
validity checking and 58
warning 19, 23
eventInfo variable 29, 38, 60, 65
errors and 60
events
See also actions
handling 12
responding to 9, 12, 26, 36
Windows applications and 9
example applications 32, 83
execution, delaying 46, 60, 73

F

.FDL files 22
FieldBackward constant 36
FieldForward constant 36, 38
fields
assigning values to 41
blank values in 63
changeValue method and 62
displaying values 52
editing format of 6
editing values 10
generating values for 61
maximum value 52
naming 41, 48
queries and 6
selecting multiple 57
tab order 36
validity checking 55, 66
viewing properties of 57
File Browser 13
files
deleting 13
renaming 13
specifying location of 51
for loops 5
Form type, declaring variables 73
Form Window Properties dialog box 70
formReturn method 74
forms
See also dialog boxes
built-in methods for 64
closing 74
controlling 69
creating 26, 33
delivering 22
as dialog boxes 12, 69
editing 51
getting values from 74
inspecting 64
key violations and 65
as manager 65
manipulating 12
modal 74
multiple 12, 69
multi-table 55, 66
objects in 7
pages of 9
properties 69
returning control 74
running 26
tabs in 36
UIObjects and 9
validity checking 63
viewing source code of 22

G

Go To command 18

H

handle, defined 51
Hello world program 25
Help system (Windows) 12
Help window 83

I

id method 38, 41, 65
ID numbers, creating 6
if statements 38
if..endif block 38
if..then block 5, 49
input/output 43
Insert Type button 20
insertion point, controlling 36, 38, 57
inspect, defined 8
INSTALL program 32
isBlank method 63

K

key violations 63
  multi-table forms 67
keystrokes, responding to 12
keywords 19
  disableDefault 38
  endif 39
  endVar 45
  if 38
  not 49
  var 29, 45
Keywords command 19

L

Language menu 18
Level command 22
libraries
  calling code from 17
  run-time 12, 29
lines (drawn) 7
links
  See tables, linking
locate method 48, 79
locks, record 35
lookup tables 7
loops 5, 38, 49

M

menus, creating 12
message procedure 46, 49, 60
messages
  See also errors
  displaying 46
  displaying in dialog box 25
methods
  See also built-in methods
  attaching to objects 26
  calling external 17
  comments in 50
  creating 20
  custom 16
  debugging 24
  defined 5
  delaying execution of 46, 60, 73
  disabling 38
  displaying syntax 20
  editing 15, 28
  editing multiple 16
  syntax 3
  user-defined 12
Methods dialog box 15, 19, 27
modal dialog boxes 29, 74
mouse events 12
mouseClick method 10
MoveEvent type 60
moveTo method 38
msgInfo procedure 29, 54, 66
multi-form applications 69
multi-table forms 55
  key violations 66

N

naming conventions
  objects vs. fields 48
  spaces in names 41, 48
  underscore (_) in names 41
networks, ID numbers for 6
New Custom Method field 16
New Form dialog box 26
Next Warning command 19
not keyword 49
Notepad (Windows) 18
Number type
  declaring 48
  Table type versus 51
numbers
  performing calculations 61
  returning maximum 52

O

Object Name dialog box 34
Object Tree 7
Object Tree command 8, 19
object types
  defined 9
  inserting name in code 20
ObjectPAL Debugger
  See Debugger
ObjectPAL Editor
  See Editor
objects
  See also built-in methods; properties
  built-in methods 26
  declaring global variables 16
  defined 7
  defined for advanced programmers 13
  defining behavior of 9, 26
  design 63
  displaying properties of 21
  focus status of 12

forms and 7
inspecting 26
modular nature of 10
naming 34, 41, 48
position of 12
program design aspects 11
properties of 9
relationship between 7
responding to events 9
Self variable and 42
tabs and 36
unnamed 42
viewing source code 22
visual nature of 7
open method 73, 79
operators, comparison (< >) 48, 59, 81

P

parameters, defined 29
Pascal 12
Paste command 18
peKeyViol constant 66
Print File dialog box 53
print method 54
printing
page ranges 53
reports 52
setting specifications 53
Proc window 17
procedures, user-defined 17
properties
basic syntax 41
browsing 21
defined 7
dialog box 70
displaying 21
inspecting 8, 26
manipulating 39
Self variable and 42
setting 2, 12, 22
setting multiple 57
Tab Stop 57
viewing 57
Properties menu 10, 22
ObjectPAL routines versus 12
protection
See access rights
pushButton method 29
attaching code to 28, 34, 71
editing 26, 46, 50

Q

QBE files, creating 13
queries 49
creating 13
fields and 6
ObjectPAL routines versus 6

R

records 12
See also tables
constants for 35
deleting 35
displaying 40
inserting 34, 35, 50
key violations 63
locking 35, 64, 67
moving between 40
posting 39, 65
searching for 46
unlocking 35, 65, 67
validity checking 55, 63
Replace command 18
Replace Next command 18
Report type, declaring variables 54
reports
print specifications for 53
printing 52
reserved words
See keywords
run-time library 12, 29

S

sample applications 32, 83
Save command 28
scope, defined 16
Search command 18
Search Next command 18
searches 46
strings 18
tables 13, 79
search/replace operations 49
Select All command 18
Select command 18
Self variable 59, 63, 80
defined 42
setErrorCode method 60
Show Compiler Warning command 23
sleep procedure 46, 49, 60
sounds, creating 49
spaces, in object names 41, 48

SpeedBar
  hiding 12
  placing objects with 1
  TCursors and 77
  UIObjects and 9
String type, declaring variables 45
strings
  searching for 18
  storing 45
syntax
  displaying 20
  errors 18
  notation in book 3

T

Tab Stop property 57
table frames 7
  attaching code to 67
  defined 13
  key violations and 67
Table type
  attaching variable to table 51
  defined 13
  Number type versus 51
tables
  See also records
  attaching Table variable to 51
  hidden 77
  inserting records in 34, 35, 50
  key violations 66
  linking 55
  lookup 7
  manipulating 12
  multiple 55, 66
  outside data model 77
  pointers to 77
  posting values to 62
  sample 32
  searching 13, 46, 48, 79
  sorting 2, 13
  TCursors and 77
TableView type, defined 13
tabs, controlling 36, 57
TCursors 77–81
  defined 13, 77
  opening 79
  SpeedBar and 77
text
  deleting 18
  displaying 46
  selecting 18
titles, editing 22
today procedure 59
Type window 17
types
  See data types; object types
Types and Methods dialog box 19

U

UIObjects
  built-in methods 10
  defined 9
  listing 21
underscore (_), in object names 41, 48
Undo command 58
user interface
  creating 6
  event-driven 9
Uses window 17

V

validity checking 55–68, 78
  adding 59
  built-in 58
  canDepart method and 59
  supplying values 60
Value property 41, 52, 74
values
  assigning to fields 41
  assigning to variables 45, 52
  blank 63
  changing 61
  maximum 52
  searching for 46
  testing for 48
var keyword 29, 45
Var window 16
variables
  assigning value to 45
  declaring 16, 43, 45, 73
  Form type 73
  global 16
  as handles 51
  Number type 48, 51
  Report type 54
  scope of 16
  String type 45
  Table type 51
view method 45

W

wait method 73
warning errors 19
  displaying 23
while loops 5
Window Sizing command 23
Window Style command 7
windows, size of 23
Windows (Microsoft) program,
    events and 9
working directory 2

PARADOX
FOR WINDOWS

B O R L A N D

Corporate Headquarters: 1800 Green Hills Road, P.O. Box 660001, Scotts Valley, CA 95067-0001, (408) 438-8400. Offices in: Australia, Belgium, Canada, Denmark, France, Germany, Hong Kong, Italy, Japan, Korea, Malaysia, Netherlands, New Zealand, Singapore, Spain, Sweden, Taiwan and United Kingdom ▪ Part # PDX1110WW21774 ▪ BOR 4901