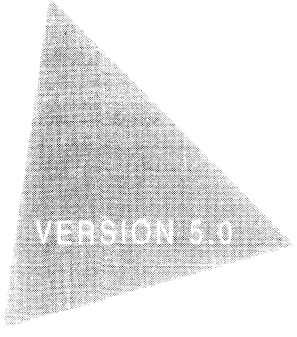
Guide to ObjectPAL™



Borland®
Paradox® for Windows

Borland International, Inc., 100 Borland Way
P.O. Box 660001, Scotts Valley, CA 95067-0001

1E0R694
9495969798-987654321
H1

Contents

Tables

Figures

Examples

Introduction

ObjectPAL™ is the integrated programming language for Paradox for Windows. You can use ObjectPAL to develop full-featured Windows database applications, or simply to add new features to a Paradox for Windows 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 the ObjectPAL tutorial in Chapter 2 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 Toolbar 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

It's much easier to learn ObjectPALwhen you're familiar with these actions and concepts.

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 summarizes the most important differences between PAL and ObjectPAL.

How to use this manual

This manual consists of a tutorial followed by detailed information and examples for intermediate and advanced users.

The tutorial is designed to help users who have little or no programming experience get up and running in ObjectPAL. Experienced programmers can use the tutorial 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. You should complete the examples in the order they appear. Later you can refer back to the tutorial for specific code.

Important The tutorial section of this manual assumes an ObjectPAL level of Beginner. Set the ObjectPAL level in the ObjectPAL Preferences dialog box (see Chapter 8).

The rest of the manual presents information and concepts you'll need to develop Paradox applications. It assumes that you are already familiar with interactive Paradox. In this book, "interactive Paradox" means the set of tasks you can accomplish in Paradox without ObjectPAL.

Printing conventions

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

Table Intro.1 Printing conventions

Convention	Applies to	Examples
Bold	Method names	**insertRecord**
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, text that you type in, and messages displayed onscreen	`myTable.open("sites.db")` `Jan - Jun, 7/20/92`
Important	Important information	
	Information of special interest to beginning programmers	

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 online ObjectPAL Help or the *ObjectPAL Quick Reference*.

The following table summarizes ObjectPAL syntax notation:

Table Intro.2 Syntax printing conventions

Convention	Element	Examples	Meaning
Normal font	Keyword	setPosition	Type exactly as shown.
Italic	Fill-in	*tableVar*	Replace with an expression.
{|} (braces and bar)	Choice	{ Yes|No }	You *must* choose one of the elements separated by the vertical bar.
[] (brackets)	Optional	[, *tableVar2*] [ELSE]	You *can* choose whether or not to include this.
* (asterisk)	Repeat	[, *tableVar2*] *	You can repeat this argument.

How to use the online ObjectPAL Help

The online ObjectPAL Help provides comprehensive information about ObjectPAL, including

- Conceptual material

- A complete language reference (basic language elements, methods, and procedures in the ObjectPAL run-time library, and ObjectPAL constants)
- Code examples you can paste into your own application

To get ObjectPAL help at any time, do the following:

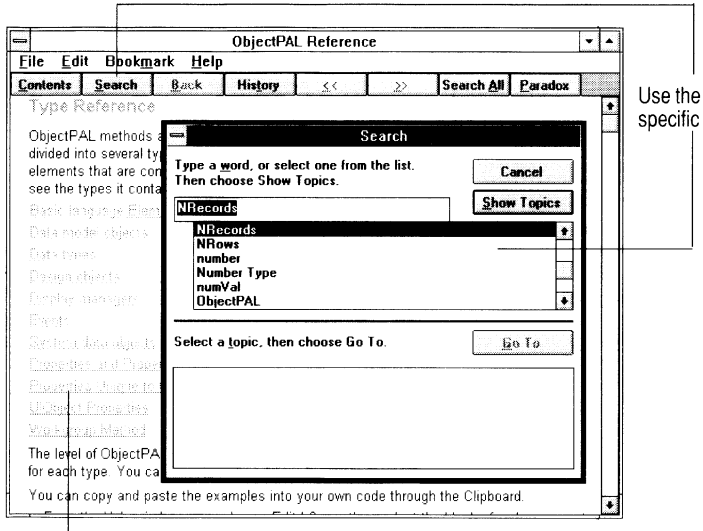
1 Press *F1* to open the Help application.

2 In the ObjectPAL Editor, press *F1* to get Help for a specific ObjectPAL keyword when a word is selected or the insertion point is in word.

3 Click the ObjectPAL button to open the ObjectPAL Help file.

If you know what you're looking for, you can use the Search function to find it, as shown in Figure Intro.1. Otherwise, you can use Help functions (for example, clicking an underlined word) to browse through the file. You can copy code from Help examples to the Clipboard and paste it into your own applications to save typing and reduce the likelihood of errors.

Figure Intro.1 Using the ObjectPAL Help System

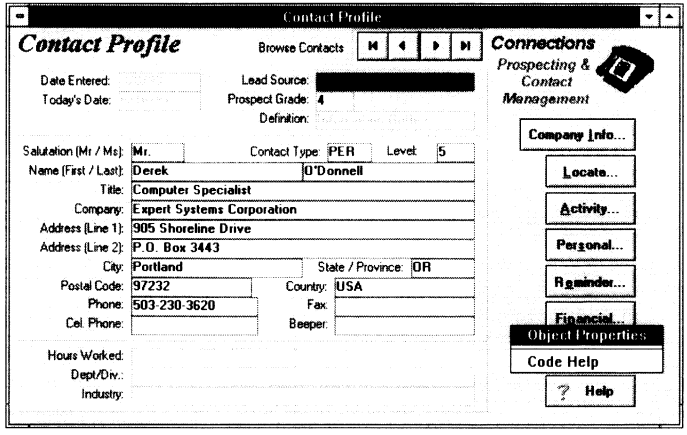


How to use the example files

The example files include one full-scale application (Connections) and several small applications (also called mini-apps). All of the example applications 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. In Connections, you can also get ObjectPAL help by inspecting (right-clicking) an object

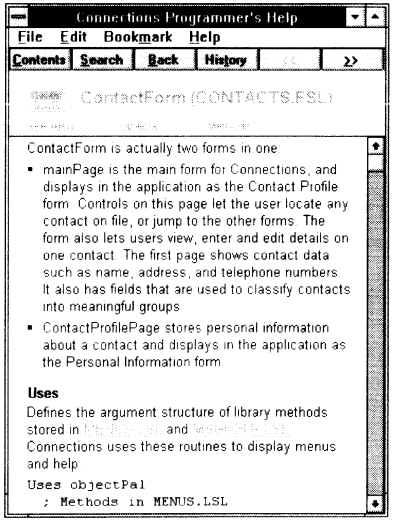
and choosing Code Help from its pop-up menu. When you choose Code Help, you open a Help window listing all the ObjectPAL code attached to the object you inspected, as shown in Figure Intro.3.

Figure Intro.2 Using the Help system in the example applications



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

Figure Intro.3 Code Help



When you right-click an object in the example applications and choose Code Help, a Help window opens listing all the ObjectPAL code attached to that object.

Part

Getting started with ObjectPAL

This part of the manual will help get you up and running with ObjectPAL as quickly as possible. It includes the following chapters:

- Chapter 1, "Introducing ObjectPAL," gives a brief introduction to ObjectPAL. It gives you a conceptual overview of the benefits and structure of the ObjectPAL language.
- Chapter 2, "Learning ObjectPAL—A tutorial," provides step-by-step instructions that show you how to use ObjectPAL to perform some common database programming tasks. The tutorial includes the following lessons:
  - "Lesson 1: Programming a button," shows how to attach ObjectPAL code to a button so that the code executes when you click the button.
  - "Lesson 2: Initiating and responding to actions," presents the fundamental concepts of ObjectPAL programming.
  - "Lesson 3: Preventing actions," demonstrates some techniques for blocking a user from performing certain actions, like inserting new records or editing a field.
  - "Lesson 4: Input and output," shows how to get information from the user, how to display information to the user, and how to process this information along the way.
  - "Lesson 5: Validating data entry," presents techniques for making sure users enter valid data.
  - "Lesson 6: Dialogs—controlling another form," shows how to use ObjectPAL to control one form from another form, and presents techniques for using a form as a dialog box.
  - "Lesson 7: TCursors—working with tables behind the scenes," is for programmers who want to sample one of ObjectPAL's more advanced features. This chapter shows how to work with tables that aren't part of the form's data model.

- "Lesson 8: Creating drop-down lists,"shows how to use tables and TCursors to program drop-down edit fields that present the user with a list of values to select from.
- "Lesson 9: Menus," shows how to build a menu and process menu choices.
- "Lesson 10: System procedures," is an introduction to ObjectPAL's System type procedures. This chapter shows how to use System procedures to display messages, invoke Paradox command dialog boxes, manipulate the Browser, and to get information about the system running Paradox.

Introducing ObjectPAL

This chapter is a brief introduction to the basic concepts of ObjectPAL. If you are new to ObjectPAL, you should read this chapter and then work through the next chapter, "Learning ObjectPAL—A tutorial."

If you will not be working through the tutorial but want a conceptual overview of ObjectPAL, read this chapter and then proceed to the next part of the manual, "ObjectPAL basics."

What is ObjectPAL?

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

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. For example, you could use ObjectPAL to do these tasks:

Automating 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, enter Edit mode, 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.

Correcting 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.

Protecting data At times, you'll want to warn users when they are about to do something potentially damaging to the data (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.

Look to interactive Paradox first

Keep in mind 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. A formal definition says that an object consists of data and code. In ObjectPAL terms, objects have properties (like color, position, and line width) and methods (code that defines how the object behaves). Properties are data. Methods are code.

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.

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

The process of placing objects in a form 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.

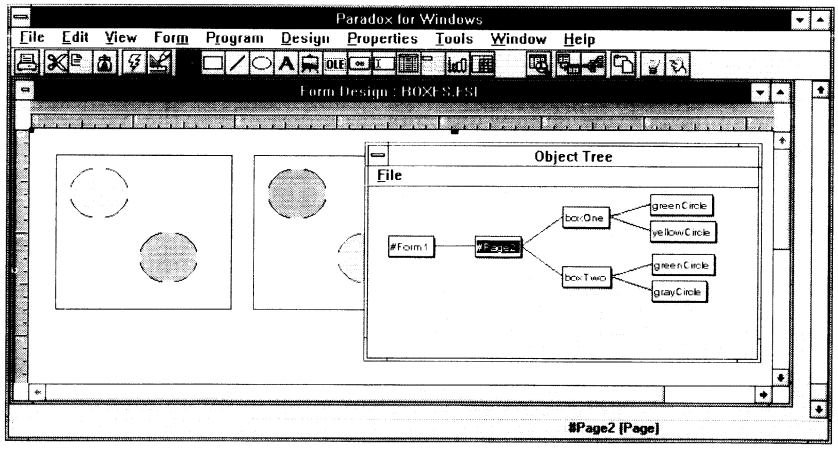
This work is very different from the work you would have to do to create an application in a non-visual environment. In many other languages, you create the user interface by writing code in some kind of text editor. To run the application, you must compile the code, debug it, run it again, and so on 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. For example, suppose a form contains one page, and that page contains two boxes, and each box contains two circles. Figure 1.1 shows the form and its corresponding Object Tree.

Figure 1.1 Using the Object Tree



Event-driven

In Windows, no program has absolute, permanent control. No program can ever assume that it has firsthand 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 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. Here are some examples of user actions that generate events: clicking the mouse button, releasing the mouse button, moving the pointer over an object, pressing a key, moving the insertion point into a field, moving the insertion point out of a field, selecting an item from a menu—in short, anything you do in Paradox generates an event.

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 determines 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 mouse moves inside the borders of the box and change the color back again when the mouse 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 tell the box how to respond only 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.

Modular

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.

This makes objects easy to program. It also means 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 are built with all the intelligence at or near the top of the system. Processes in a system are seen as linear, and programming was approached in linear fashion. But in real-world complex systems (systems such as traffic patterns and the stock market) 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*.

Advantages of modular programming

Object-based programming lets you develop systems that model real-world behavior. 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. 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.

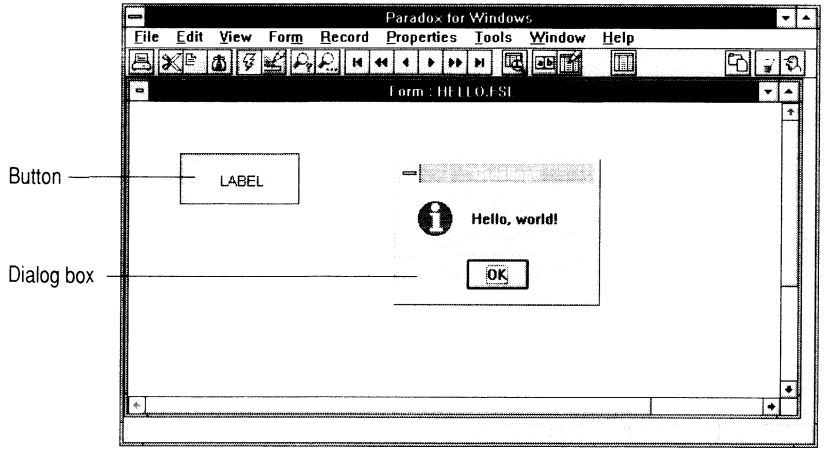
Learning ObjectPAL—A tutorial

Lesson 1: 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 2.1 shows how to do this. First, Figure 2.1 shows the application in action.

Figure 2.1 Hello, world!



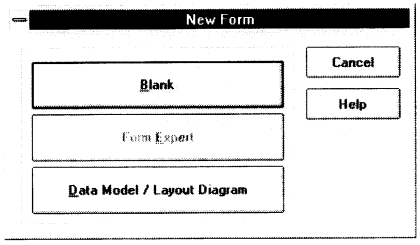
By adding the statement **msgInfo("Greetings", "Hello, world!")** to the button's built-in **pushButton** method, you make the dialog box appear when you click the button. The dialog box stays open until you close it.

Note To complete the examples in this manual, change your working directory to PDOXWIN\SAMPLE. See the *User's Guide* for instructions on changing your working directory.

Example 2.1 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 the New Form dialog box.

Figure 2.2 The New Form dialog box



Note If this dialog does not appear, choose Properties | Desktop, click the Form and Report Preferences button, and choose No Default in the New Forms/Reports panel. Then start the exercise again.

2 Click the Blank button.

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

4 Now run the form by clicking the View Data button or by choosing Form | View Data.

5 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.

6 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 2.2 shows you how to open the ObjectPAL Editor.

Example 2.2 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.)

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

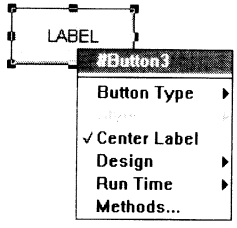
Shortcut Select the button, and then press *Ctrl+Spacebar* to open the Method Inspector.

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:

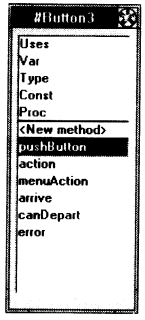
```
method pushButton(var eventInfo Event)

endMethod
```

Figure 2.3 shows each of these steps.

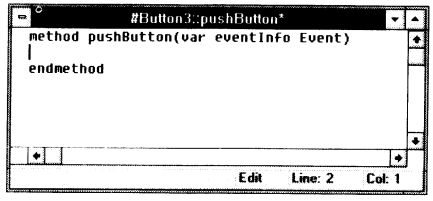
Figure 2.3 Inspect the object, select a method, open an Editor window



First, inspect the button and choose Methods from its menu to display the Method Inspector.



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 executes only 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 2.3 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 Program | 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.

Note You can also right-click anywhere in the Editor window to display the combination of Program menu and Tools 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 Click the Design button to return to the Form Design window, and choose File | Save to save your changes to disk. Name this form HELLO.FSL.

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.

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

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

msgInfo is a procedure of the System type in the ObjectPAL run-time library (RTL), a collection of predefined methods and procedures that operate on objects or data of a specific type. See Chapter 4 for more information about methods in the ObjectPAL run-time library. **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.

Lesson 2: Initiating and responding to actions

This lesson has several parts presented in the context of a single application: a form for entering customer data. This lesson shows

- 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

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 examples explain how to perform the following tasks:

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

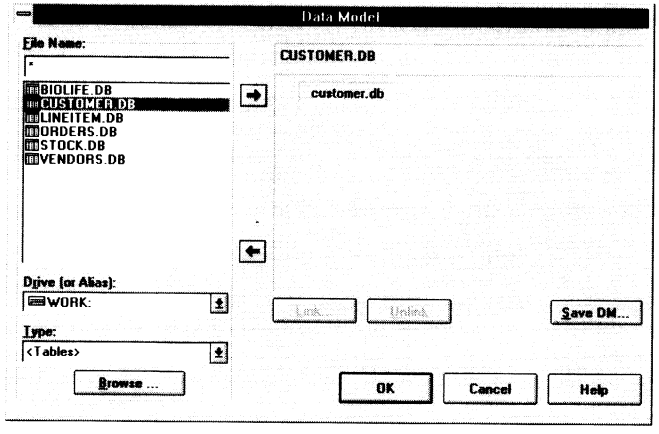
Before you begin this lesson, you need to create the form. Once you create and save this form, you can use it to complete any of the examples in this lesson.

Example 2.4 Creating the NewCust form

1 Choose File | New | Form to display the New Form dialog box. Make sure you're working in the SAMPLE directory. (For help, see the *User's Guide*.)

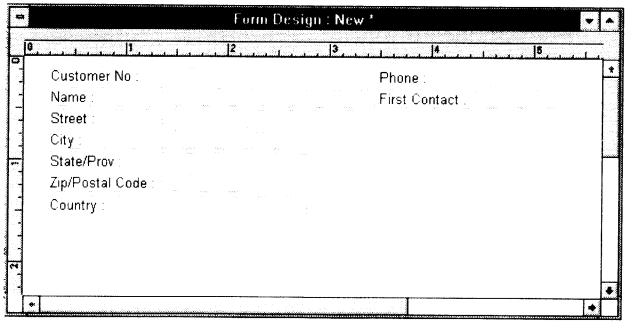
2 Click the Data Model/Layout Diagram button on the New Form dialog box.

3 Add the *Customer* table to the data model, as shown in the following figure:



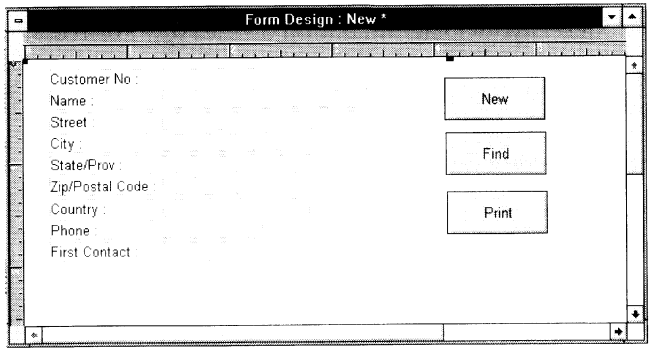
4 Choose OK to accept this data model and display the Design Layout dialog box.

5 Choose OK to accept the default layout and display the new form (shown here) in a design window.



6 Select the *Phone* and *First Contact* field objects and move them beneath the other field objects on the left, if they're not already there.

7 Next, use the Button tool to add three buttons (hold Shift down while you use the Button tool to keep it active while you're creating these 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.

8 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

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

Example 2.5 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 Method Inspector.

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 by clicking on the Check Syntax button and correct any errors.

6 Run the form.

7 Click the Edit Data button (or choose View | 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 | Insert 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 might 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 2.1 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 2.1 ObjectPAL Beginner-level action constants

Constant	Description
DataArriveRecord	Point to a new or changed record (used only to respond to the action, not to initiate it).
DataBegin	Move to the first record in a table.
DataBeginEdit	Begin Edit mode.
DataCancelRecord	Cancel changes to a record.
DataDeleteRecord	Delete a record.
DataEnd	Move to the last record in a table.
DataEndEdit	End Edit mode.
DataInsertRecord	Insert a record.
DataLockRecord	Lock a record.
DataNextRecord	Move to the next record in a table.
DataPostRecord	Post (commit) a record to the database and keep it locked.
DataPriorRecord	Move to the previous record in a table.
DataUnlockRecord	Unlock a record.

Table 2.1 ObjectPAL Beginner-level action constants (continued)

Constant	Description
FieldBackward	Move to the previous object in the tab order.
FieldForward	Move to the next 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 previously, 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. The following example controls the tab order, which specifies the object that gets focus when you press *Tab*.

Note In most cases, you should set a field's tab order interactively in Form Design with the Next Tab Stop Run Time field property (see the *User's Guide* for more information). The purpose of this example is to demonstrate how to control an object's response to a given action.

First, get a feel for the way the form behaves by default.

Example 2.6 Tab default behavior

1. Run the *NewCust* form. The focus (highlight) is in the first field object, *Customer_No*.
2. Press *Tab*. The focus moves to the *Name* field object.
3. Press *Tab* a few more times, and watch how the focus 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 2.1. (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 right and 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 2.7 applies these concepts to control how the *Name* field object responds when you press *Tab*.

Example 2.7 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 Method Inspector.

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 Method Inspector.

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 by clicking the Check Syntax button, 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

```
eventInfo.setErrorCode(UserError)
```

Important **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 focus 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 focus to an object. So, this statement moves the focus 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 focus 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.

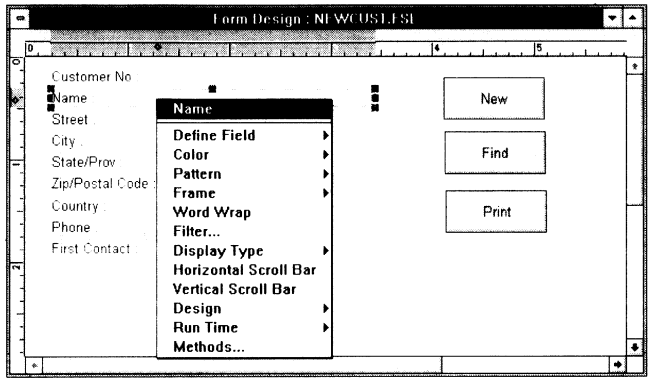
```
method action(var eventInfo ActionEvent)
   if eventInfo.id()=DataPostRecord then
      msgInfo("FYI", "About to post the current record.")
   endIf
endMethod
```

Actions and properties

Like the previous example, this example shows how to respond to an action. In addition, it 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 2.4. Using ObjectPAL, you have access to all these properties and many more.

Figure 2.4 Properties of the field object *Name*



Example 2.8 shows how to use the DataArriveRecord action constant to respond to an action that can be initiated 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 finishing an edit of 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 example, use the *NewCust* form you created earlier in this lesson (open it in a design window).

Example 2.8 Responding to an action

1 Inspect the *Name* field object, and choose Methods to open the Method Inspector.

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 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 (earlier) than 1/1/91; otherwise, it should be white.

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

How it works

This method executes whenever the *Name* field object responds to an action (and it works only 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
```

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

This lesson presented some powerful techniques. It showed 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.

Lesson 3: Preventing actions

There are times when you want to prevent a user from performing certain actions in a form. Say, for example, you don't want the user to change existing data or insert new records in a table. In most cases, you should set Paradox table or form properties to

prevent these kinds of actions. However, there are occasions where it's necessary or more appropriate to use ObjectPAL to block these actions at run time.

This lesson explains how to use Paradox or ObjectPAL to block the user from

- Editing a table or a field
- Inserting new records in a table
- Arriving on a field

This lesson does not include any examples for you to work through, but rather brief illustrations of how to use ObjectPAL to block certain user actions.

Blocking edits

The simplest way to prevent a user from changing data in a table is to assign auxiliary passwords for the table to define what kinds of table operations each user can perform. You can also assign rights to individual fields that give users

- *All* rights to the data in a field (within the limits of the table rights you specify)
- *ReadOnly* rights to view—but not to change—the data in a field
- No rights, *None*, to view or change the data in a field (Paradox hides the values in the field)

If you want to prevent the data in a field from being changed *only* when a form is run, inspect the field object and choose Run Time | Read Only from the field object's list of properties.

For more information about setting a field object's Run Time properties or assigning specific types of rights to a table or individual fields, see the *User's Guide*.

If you do want to use ObjectPAL to set a field to be read-only at run time, attach the following code to the field object's **open** method:

```
method open(var eventInfo Event)
   DoDefault
   self.Readonly = yes
endMethod
```

Important A call to **doDefault** executes the built-in code immediately so the field object is completely opened and initialized before your code executes.

Blocking arrival

Another way to prevent users from editing a field is to block them from tabbing to the field. When you turn a field's Tab Stop property off, the user won't be able to use the keyboard or mouse to move to it. The field is simply bypassed in the tab order.

To turn Tab Stop on or off with Paradox, inspect the field object in the Form Design window, and choose Run Time | Tab Stop from the field object's list of properties.

If you want to use ObjectPAL to prevent a user from tabbing to a field, attach the following code to a field object's **open** built-in method.

```
method open(var eventInfo Event)
   doDefault
   self.TabStop = off
endMethod
```

You can prevent users from moving to any field in a table frame, MRO, or other container objects (boxes, pages, and so on) by attaching the following code to the object's **canArrive** method:

```
method canArrive(var eventInfo MoveEvent)
   if eventInfo.reason() = UserMove then
      eventInfo.setErrorCode(CanNotArrive)
   endIf
endMethod
```

See Chapter 10 for information about how this block of code works.

Blocking new records

By default, users can add new blank records to a table in a form simply by moving beyond the last record in Edit mode and typing or by pressing *Ins*. You can prevent users from adding new records to a table in this way by turning off the Auto-Append property. You turn off the Auto-Append property with Paradox by inspecting the table in the data model panel of the New Form dialog box and unchecking the Auto-Append property.

You can also turn off an object's Auto-Append property at run time with ObjectPAL. The following example is attached to the built-in open method of a table frame bound to the Customer table. It calls **doDefault** to execute the built-in code, then sets the table frame's AutoAppend property to False, which prevents Paradox from automatically inserting a blank record when the user tries to move past the end of the table.

```
method open(var eventInfo Event)
   doDefault
   self.AutoAppend = False
endMethod
```

When AutoAppend is set to False, the user can insert a record by pressing *Ins* or choosing Record | Insert to insert a blank record.

Attaching the following code to the page or a UIObject to block all attempts to insert a record:

```
method action(var eventInfo ActionEvent)
if eventInfo.id() = DataInsertRecord then
   eventInfo.setErrorCode(UserError)
endIf
endMethod
```

On the form level, this code would be attached:

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

```
      endIf
    else
      ; code here executes just for the form itself
    endIf
  endMethod
```

Summary

As you create applications and customize forms with ObjectPAL, you should always ask yourself if the things you need to do with a table or a form can better be accomplished interactively with Paradox.

This lesson has shown that most user actions can be blocked by setting Paradox table properties or form object properties. However, ObjectPAL does provide the appropriate methods to prevent these actions if you prefer to use them, or if you must use them because your application creates or modifies tables or forms at run time.

Lesson 4: Input and output

This lesson shows how to get information from the user, how to display information to the user, and how to process this information along the way. It 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 this lesson explain how to add code to the *NewCust* form you created previously.

A quick way to get user input

The following example 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, and how to detect if the user has clicked OK in the dialog box. 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 2.9 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 Method Inspector.

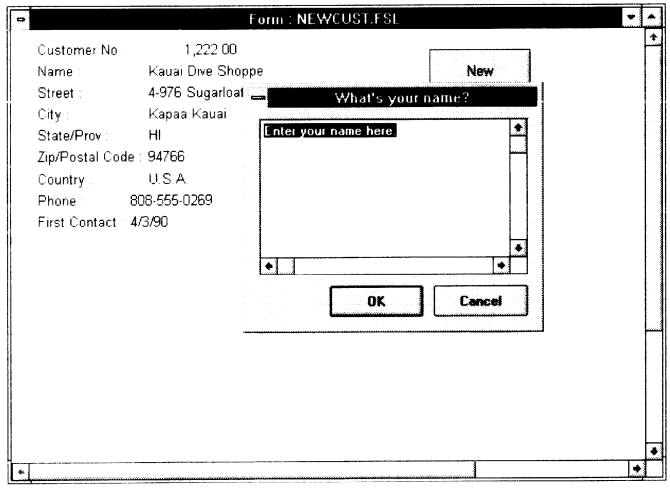
4 Choose **pushButton** by double-clicking it. This opens an Editor window for the **pushButton** method.

5 Edit the method to make it look like this:

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

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

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

   if userInput <> promptString then
      ; User entered value and clicked OK.
      message("Hello, ", userInput)
      sleep(1000)
   else
      ; User clicked Cancel, pressed Esc, or closed dialog box without entering a value.
      message("You closed the dialog box.")
      sleep(1000)
   endIf
endMethod
```

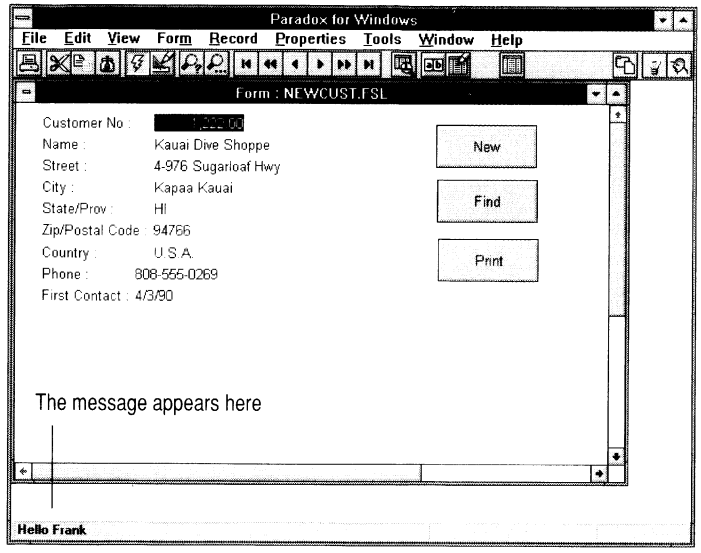
6 Check your syntax, and correct any errors.

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 window, as shown 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, promptStringString
```

This code declares the variables *userInput* and *promptString* of the String data type. Now you can use these variables in this method to store character strings.

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.

Lines 6 and 7 These two lines are

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

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

Line 9 The ninth 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 *userInput*, which now holds the value stored in the variable *promptString*, in a dialog box. The string "What's your name?" specifies the text to display in the dialog box title bar.

view does more than display the value of a variable in a dialog box; when you close the dialog box by clicking OK or pressing *Enter*, **view** assigns the displayed value back to the variable. So whatever value you enter in the dialog box is stored in the variable *userInput*. This new value is used by the next block 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 11 through 18 These lines are

```
if userInput <> promptString then
   ; User entered value and clicked OK.
   message("Hello, ", userInput)
   sleep(1000)
else
   ; User clicked Cancel, pressed Esc, or closed dialog box without entering a value.
   message("You closed the dialog box.")
   sleep(1000)
endIf
```

This block of code tests how you responded to the question in the **view** dialog box. The value of the variable *userInput*, as returned from the dialog box, is compared with the value of the variable *promptString*, the original string displayed in the view dialog box.

If you entered anything in the dialog box, the value of *userInput* is not equal to the value of *promptString*. This means the first condition of the **if** statement is true and the first sequence of statements is executed.

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

This first 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.

A **sleep** statement makes the system "sleep" (actually, it yields control to other windows so you can do other things) 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.

If you didn't enter anything in the dialog box, the value of *userInput* equals the value of *promptString*, and the first condition of the **if** statement is False. The second sequence of statements then executes, displaying the literal string "You closed the dialog box" in the status bar.

Searching for values

The next example 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 example), 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've already created.

Example 2.10 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 Method Inspector.

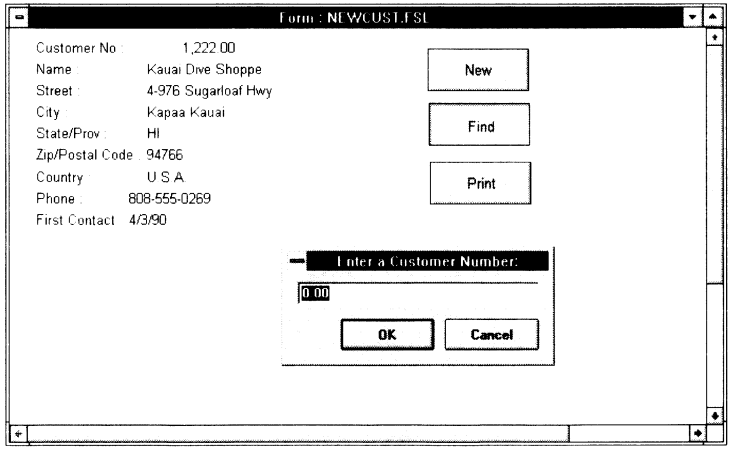
3 Double-click **pushButton** to open an Editor window for the built-in **pushButton** method.

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` in 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` in 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 earlier in this 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 2.9.

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 View | 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.

Note You can define a field in Paradox as an autoincrement field. Paradox autoincrement fields contain long integer, read-only (non-editable) values that begin with the number 1 and add one number for each record in the table. See the *User's Guide* for more information.

The next example shows how to use ObjectPAL to perform these three steps in a single mouse click. Like the previous examples, it uses the *NewCust* form.

Example 2.11 Inserting a new record

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

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 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)

   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*. A handle is a variable you can use in your code to refer to and manipulate objects.

Block 2 The second block of custom code is

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

This block initiates two 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.

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 customer 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 2.8) 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 32 presents techniques to use in a multi-user application.

Printing a report

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

Example 2.12 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 Method Inspector.
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.attach("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 2.12), 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

This lesson 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

Lesson 5: Validating data entry

This lesson presents techniques for making sure users enter valid data. It shows how to

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

The first few examples use a new form. Example 2.13 describes how to create it.

Creating a multi-table form

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

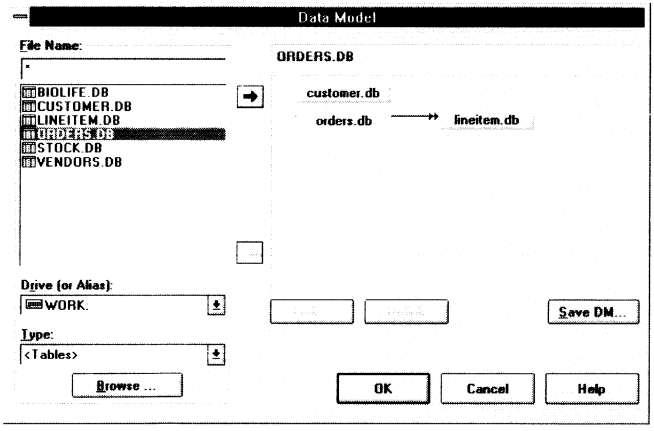
Example 2.13 Creating a multi-table form

1 Choose File | New | Form to display the New Form dialog box.

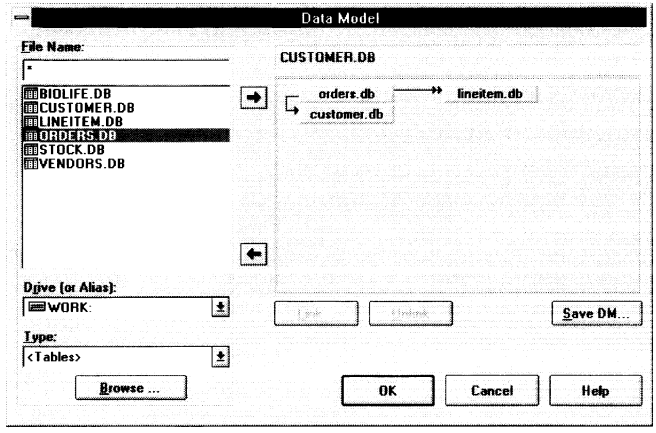
2 Click Data Model/Layout Diagram.

3 Add the *Customer*, *Lineitem*, and *Orders* tables to the data model.

4 In this form, *Orders* is the master table. Link *Orders* to *Lineitem* as shown:



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

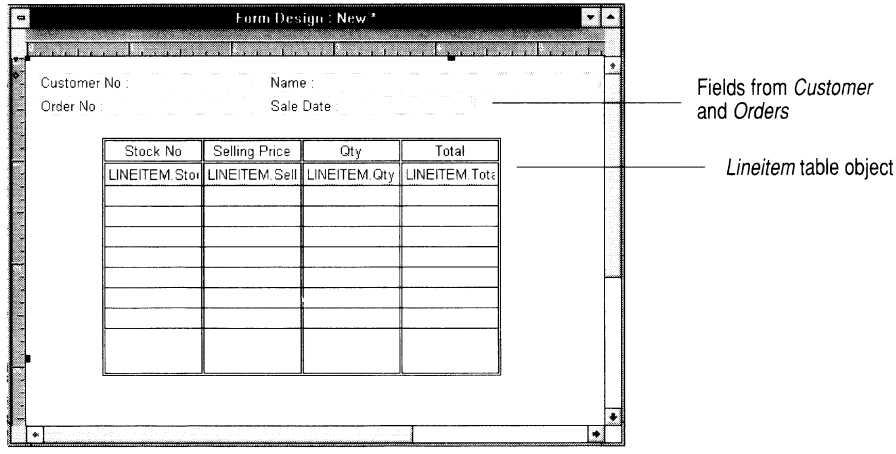


6 Choose OK to accept this data model and display the Design Layout dialog box.

7 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.

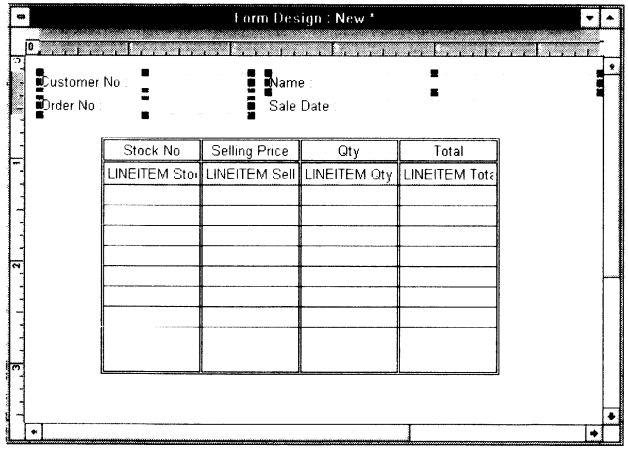
8 Choose OK to accept the layout and display the new form in a design window.

9 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:



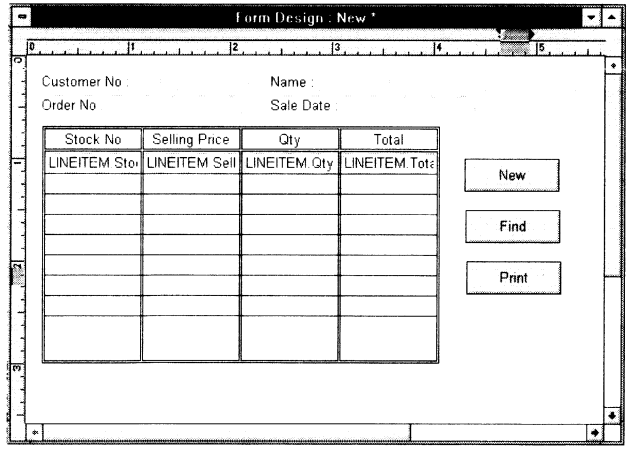
10 *Shift*+click the following field objects: *Customer_No*, *Order_No*, and *Name*. When you have all three field objects selected (as shown in the next figure), inspect any one of

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 insertion point 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.

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



12 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 2.14 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` and press *Tab*.

5 That's not a valid date, so the form displays an error message in the status bar. This happens because Paradox's built-in validity check encountered an error.

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 might 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 2.15 ObjectPAL validity checking

1 Return to the design window.

2 Inspect the *Sale_Date* field object, and choose Methods to display the Method Inspector.

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

4 Edit the method to look like this:

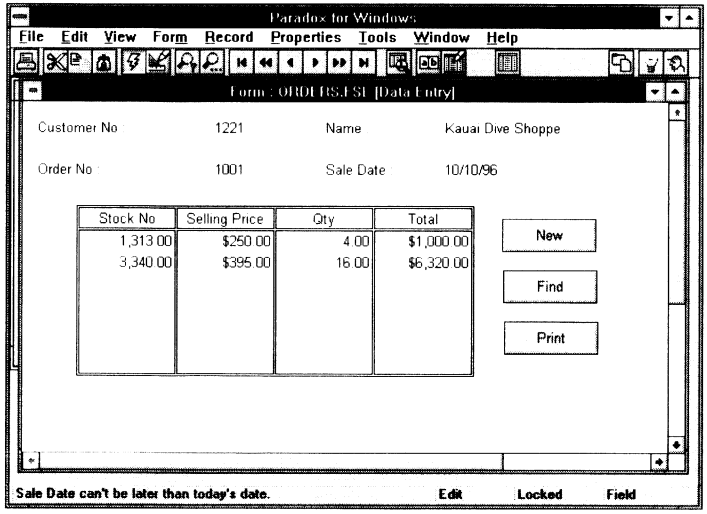
```
method changeValue(var eventInfo ValueEvent)
   if eventInfo.newValue() > Today() then
      eventInfo.setErrorCode(CanNotDepart)
      message("Sale Date can't be later than today's date.")
      sleep(1000)
   endIf
endMethod
```

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 *Tab*.

8 Look for the message in the status bar, as shown in the next figure.



9 Choose Edit | Undo to restore the original date.

How it works

ValueEvent methods control what happens when the value of a field changes. Defined only for field objects, **changeValue** asks for permission to change the value of a field. It is called *before* the value is stored, so you can check the value and decide whether you really want to post it.

Line 2 The second line of the method is

```
if eventInfo.newValue() > Today() then
```

newValue returns the new value to be assigned to a field for a ValueEvent. **today** is a run-time library procedure that returns the current date, according to your computer's internal clock.

Taken as a whole, this statement compares the value in *Sale_Date* with today's date. If the value is later than today's date, the user is prevented from leaving the field object until it contains a valid value. 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 insertion point 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 the online ObjectPAL Help.

Lines 4 and 5 The fourth and fifth lines are

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

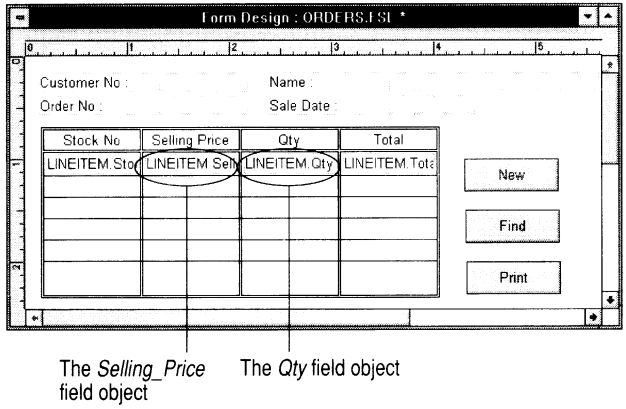
The **message** statement displays an error message in the status bar, and the **sleep** statement causes a delay so you have time to read it.

Supplying values

Sometimes, the best way to get valid data is to provide it yourself. Another way to make the user's life easier is to use ObjectPAL to perform calculations whenever possible. The following example shows one approach.

As shown in Figure 2.5 and Example 2.16, 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 2.5 The Orders form



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

Example 2.16 Performing calculations

1. Return to the design window.
2. Inspect *Selling_Price*, and choose Methods to open the Method Inspector.
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 Method Inspector.

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 the *Selling_Price* **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** method 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 examples have shown how to check validity at the field level (that is, how to check the values of individual fields). This next example presents techniques for record-level validation. Specifically, it shows how to catch key violations.

There are two parts to this exercise. 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

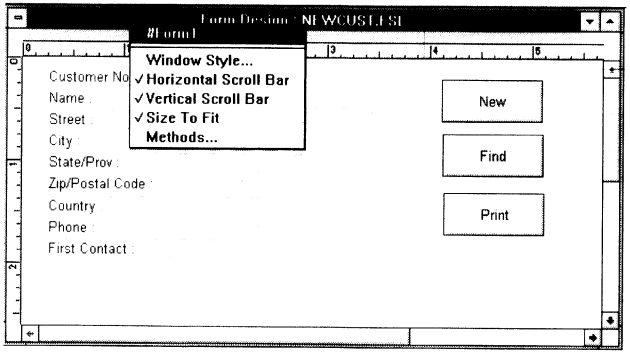
In previous examples, you worked with individual objects contained in a form. Here, you're introduced to the form as a design object. You'll see how the form manages the objects it contains and how it oversees events and actions.

Example 2.17 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 Method Inspector. This window lists the form's built-in methods.

Shortcut You can inspect a form by right-clicking the form window 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 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 Chapter 10; 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 eventInfo.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, such as by choosing Record | Post/Keep 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 eventInfo.errorCode() = peKeyViol then
```

This statement uses **errorCode** to test the error code stored in the variable *eventInfo*. 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

Handling key violations in multi-table forms

Example 2.18 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 2.16, the *Orders* form contains field objects and a table frame. Example 2.17 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 manages only the field and record objects it contains. The form sees every event and action for every object in the form; the table frame sees only 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 2.18 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 eventInfo.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 2.17, 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 lesson presented techniques for record-level validity checking. The examples 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

Lesson 6: Dialogs—controlling another form

This lesson shows 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 this lesson, the *Orders* form is the *calling form*; that is, the *Orders* form calls (opens) a second form (the dialog box)—which in this lesson is the *Cust* form.

Example 2.19 and Example 2.20 show you how to make a form into a dialog box; Example 2.21 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 2.19 Designing a dialog box

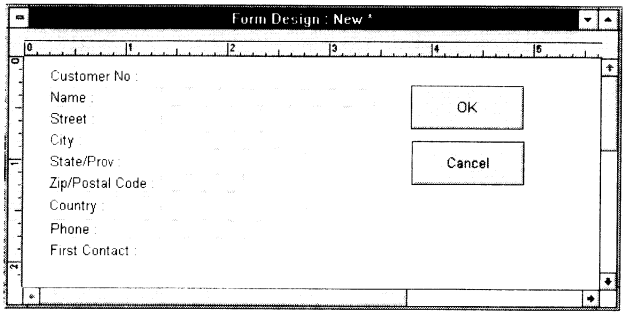
1 Choose File | New | Form to open the New Form dialog box.

2 Click the Data Model/Layout Diagram button.

3 Choose CUSTOMER.DB from the list of tables, and click OK to close the Design Layout dialog box.

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 Choose Form | Page | Layout.

9 In the Custom Size panel of the Page Layout dialog box, set the Width to 5.5 inches and the Height to 2.5 inches.

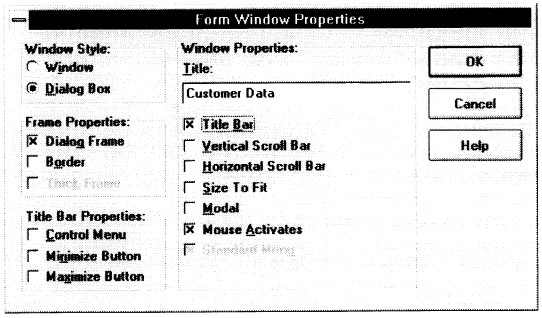
10 Choose OK to close the Page Layout dialog box.

11 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 2.20 shows you how to set this form's properties to make it look and behave like a dialog box.

Example 2.20 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 box 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 Save and close the form.

Managing a dialog box

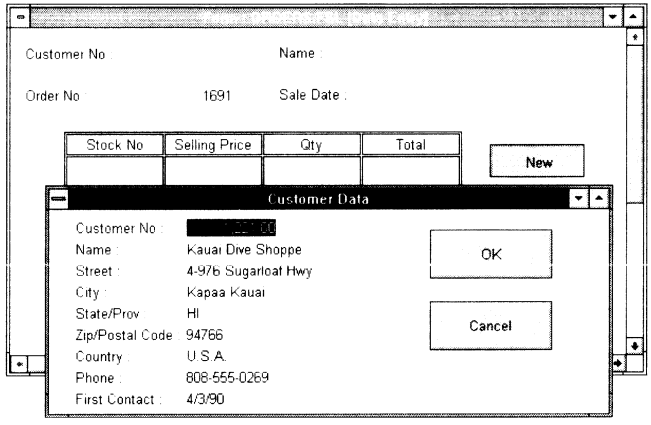
The next example describes how to manage a dialog box (or any other form you want to control using ObjectPAL). It shows how to

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

The example 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. The example also uses the *Cust* form to enter or retrieve data for the customer and to return values to the calling form.

Figure 2.6 shows the two forms in action.

Figure 2.6 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 2.21 walks you through the steps.

Example 2.21 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 Method Inspector.
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
```

```
    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 Toolbars.

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 2.11), 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 2.20, 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 interested 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 box each time this method executed.

Summary

This lesson introduced the basic techniques for managing a multi-form application. It 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

Lesson 7: TCursors—working with tables behind the scenes

This lesson is for programmers who want to sample one of the more advanced features of ObjectPAL. It shows how to work with tables (that is, insert and delete new records, change field values) 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 might 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 Toolbar 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.

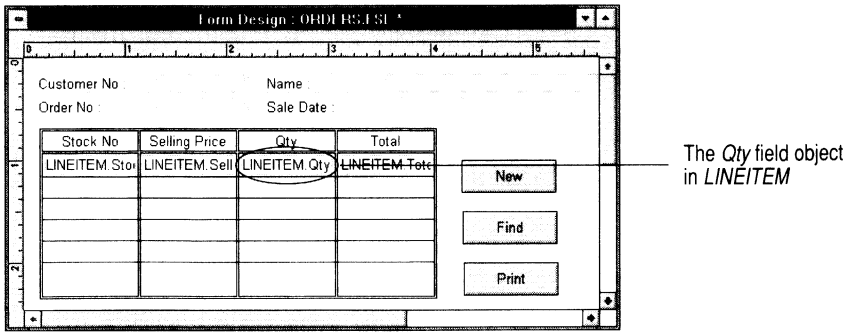
Using a TCursor

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 2.14: code attached to a field object's built-in **changeValue** method checks the object's Value property and keeps the cursor on the field until the user enters an acceptable value.

Figure 2.7 shows where to attach the code.

Figure 2.7 Attaching code to the Qty field object in LINEITEM



Example 2.22 Using a TCursor

1 In the *LINEITEM* table frame, inspect the field object named *Qty*, and choose Methods to open the Method Inspector.

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

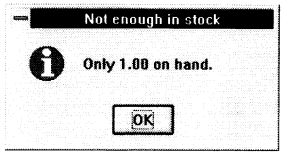
3 Edit the method to look like this:

```
method changeValue(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, 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 500 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
```

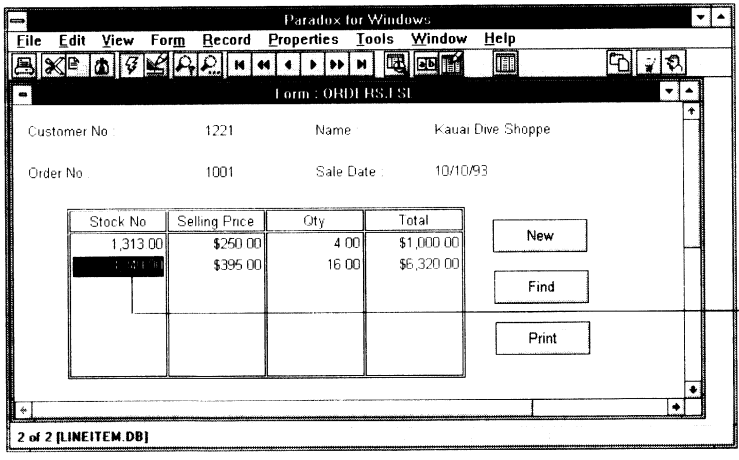
Line 8 uses the **locate** method to search the *Stock* table for the stock number in the current record of the *LINEITEM* table frame. The **locate** method you use on a TCursor is not the same **locate** you use to search a table frame, but they behave the same way. As described in Example 2.10 earlier in this tutorial, 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 2.8. 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 2.8, 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 2.8 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 lesson 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

Lesson 8: Creating drop-down lists

Forms commonly include fields that offer users a specific set of values to choose from. An order-entry form, for example, might include a field for payment method or shipping method. By setting a field's display type to Drop-Down Edit and programming its list, you provide a quick and easy way for the user to enter a valid value.

This lesson presents two techniques for programming drop-down lists like the one shown in Figure 2.9. The first technique is quick and easy; the second technique is slightly more involved, but it gives you more control over the list.

Note The techniques presented here work equally well for drop-down lists and other lists.

Using the tables to program lists

You can enter values you want displayed in a drop-down edit list in the Paradox Define List dialog box. See the *User's Guide* for more information.

The quickest, easiest way to assign values to a drop-down list with ObjectPAL is through the DataSource property. For example, to create a list that displays the phone numbers stored in the *Customer* table, follow these steps:

Example 2.23 Assigning values to a drop-down list using the DataSource property

1. Create a blank, unbound form.
2. Place a field object, and set its display type to Drop-Down Edit (or List, according to your preference). When the Define List dialog box appears, choose OK to close it.
3. With the field object selected, choose Tools | Object Tree to display the tree diagram for the list. A list is a compound object that has two parts: the field object and the list object. Attach code that affects the values displayed in the list to the list object; attach code that affects values displayed in the field to the field object.
4. In the Object Tree, inspect the list object and choose Methods to display the Method Inspector.

5 The usual place to attach list-building code is the list object's **open** method, but you can attach the code to other methods and even to other objects, as needed. Attach the following code to the list object's built-in **open** method:

```
method open (var eventInfo Event)
   doDefault
   self.DataSource = "Customer.phone" ; build list from Phone field of Customer table
endMethod
```

6 Run the form.

7 Close the form without saving it.

Note Embed the table name in quotes if you want to use a .DB or .DBF file extension when you describe your table name. (The backslashes preceding the interior double quotes are required.)

```
self.DataSource = "\"customer.db\".phone"
; backslashes are required to nest double quotes in a string
```

How it works

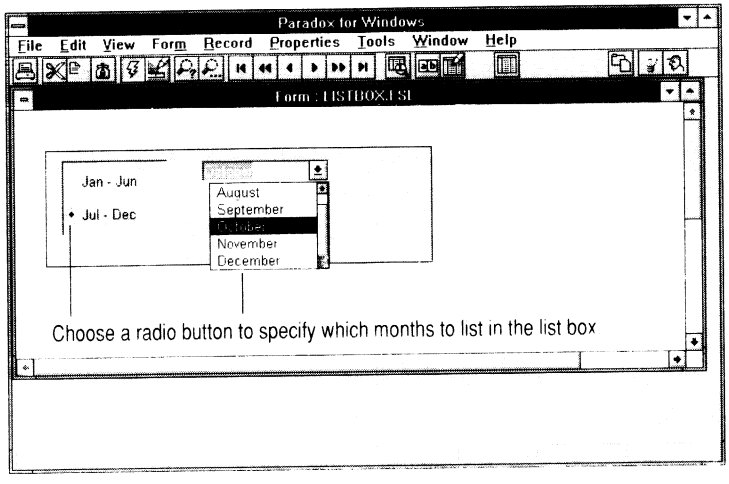
In Edit mode, the actual value of the field is changed to the value the user selects from the list. The DataSource property uses the data from a specified field (column) of a table as the source of items to display in the list.

Using TCursors to program lists

The following example shows how to program drop-down edit lists like the one in Figure 2.9. You can use these same techniques to create regular lists.

As you work through this example, you'll create a table of the months of the year and a form containing two fields: one defined as a pair of radio buttons and the other defined as a drop-down edit list. The drop-down edit list displays the names of six months (read from the table) as specified by the radio buttons.

Figure 2.9 A drop-down edit list



Example 2.24 Creating the table

1 Create a Paradox table containing one unkeyed alpha field 16 characters wide (A16). Name the field object *monthName*, and name the table *Months*.

2 Enter the months of the year, in order, into the *Months* table. (The table is unkeyed so the months appear in calendar order, not alphabetical order.)

3 Close the table.

Next, create a blank, unbound form.

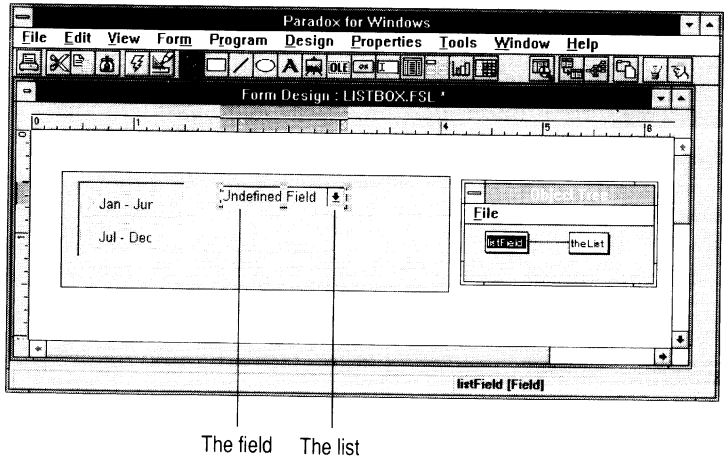
Example 2.25 Creating the form

1 Choose File | New | Form, and click the Blank button in the New Form dialog box to create a blank, unbound form.

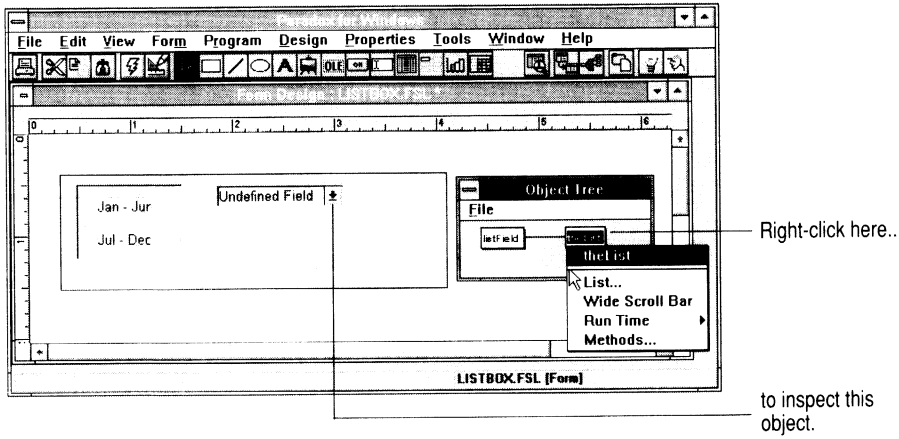
2 Choose the Field tool from the Toolbar, and place a field object near the center of the form. Inspect the field object, and change its name to *listField*.

3 Inspect the *listField* object, and change its DisplayType property to Drop-Down Edit. A dialog box appears. Don't enter any values; just choose OK.

4 Choose Tools | Object Tree. As the Object Tree diagram in the following figure shows, a drop-down edit field is a compound object—that is, it's made up of more than one object. A drop-down edit field consists of the field object itself, which displays the field object's value, and the list object, a button you click to display a list of values. (Programming the list is explained later in this chapter.)



5 Right-click the list object in the Object Tree to inspect it, as shown in the following figure. Change its name to *theList*. Close the Object Tree window.



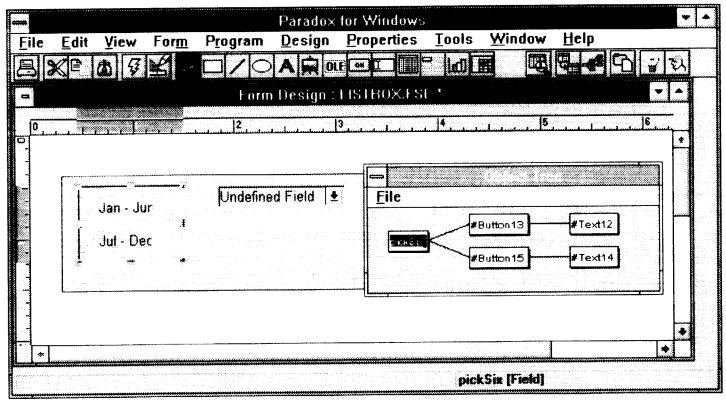
6 Place another field object to the left of *listField*. Inspect this field object, and change its name to *pickSix*.

7 Change *pickSix*'s DisplayType property to Radio Buttons. In the dialog box that appears, enter these values:

```
Jan - Jun
Jul - Dec
```

8 Choose OK to close the dialog box.

9 Choose Tools | Object Tree again. As the next figure shows, radio buttons are compound objects, too.



Example 2.26 Attaching the code

All the code in this example is attached to *pickSix*.

1 Close the Object Tree window.

2 Inspect *pickSix* and choose Methods.

3 From the Method Inspector, open the ObjectPAL Editor windows for **Var**, **open**, **close**, and **newValue**. (You can double-click each item, or use *Ctrl*+click each item, then right-click to display the Method Inspector menu and choose Open. Edit the text in each window as shown:

Var window

```
Var
   monthsTC  TCursor
   i         SmallInt
endVar
```

open method

```
method open(var eventInfo Event)
   monthsTC.open("months.db")
   self.value = "Jan - Jun"     ; assign an initial value
; the previous statement triggers the newValue method
endMethod
```

close method

```
method close(var eventInfo Event)
  monthsTC.close()
endMethod
```

newValue method

```
method newValue(var eventInfo Event)
if eventInfo.reason() = EditValue or eventInfo.reason() = StartUpValue then
      listField.theList.list.count = 0                    ; reset the list
      switch
         case self = "Jan - Jun" : monthsTC.moveToRecord(1) ; start with January
         case self = "Jul - Dec" : monthsTC.moveToRecord(7) ; start with July
      endSwitch

      for i from 1 to 6                                   ; build the list
         listField.theList.list.selection = i
         listField.theList.list.value = monthsTC.monthName
         monthsTC.nextRecord()
      endFor
endIf
endMethod
```

4 Check your syntax, and correct any errors.

5 Choose File | Save to save your work.

6 Choose Form | View Data to run the form.

7 Click the radio button, and then click the drop-down list to verify that the correct months are listed.

How it works

If a built-in method is triggered by an Event, you can use **reason** to find out why. Using **reason** and ObjectPAL constants, you can specify which, if any, of these conditions to respond to.

In the code attached to the **newValue** method, the following statement ensures that the list-building code executes only when the **newValue** method is triggered in two

instances: when you first run the form and when you press a radio button. If **newValue** is triggered for any other reason, the list-building code doesn't execute.

Line 2 `if eventInfo.reason() = StartUpValue or eventInfo.reason() = EditValue then`

You work with a drop-down list by manipulating these properties: List.Count, List.Selection, and List.Value.

Line 3 The List.Count property holds the number of items in the list; specifying List.Count = 0 empties a list.

Line 9 The List.Selection property holds the indexes for the list items. The first item in a list has List.Selection = 1, the second item has List.Selection = 2, and so on.

Line 10 The List.Value property holds the string for a given selection.

Summary

Lists are a quick and easy way to present specific values to a user, while saving the user from having to remember and type the correct values. This lesson showed you how to

- Use a table to assign values to a drop-down list through the DataSource property
- Use a TCursor to program a list of values displayed in a drop-down list

Lesson 9: Menus

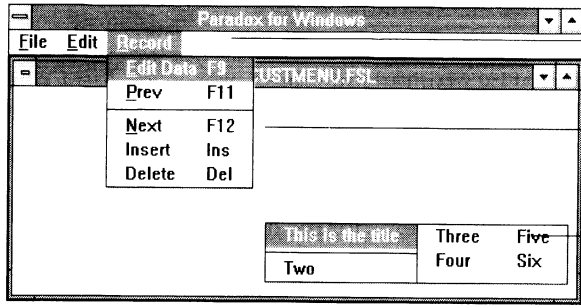
This lesson shows how to build, display, and respond to a menu. For more examples showing how to work with menus, run the Connections example application and the Slot Machine game application. Both come with online Help and explanations of the code. After learning the basic techniques for working with menus in this lesson, be sure to see Chapter 14 for help on building more full-featured menus.

About Paradox and ObjectPAL menus

A menu is a list of items that appears horizontally across the application menu bar. Menus you build using ObjectPAL replace Paradox's built-in menus (but you can get Paradox menus back using **removeMenu**). When you choose an item from a menu (one of Paradox's or one of yours), the text of that item is returned to the built-in method **menuAction**, which is the method to use to handle menu choices. Unlike the menus you program yourself, you cannot add your own items to the Paradox menus.

A typical application combines menus and pop-up menus (shown in Figure 2.10).

Figure 2.10 A menu and two pop-up menus



This is a menu. It contains three items: File, Edit, and Record.

This is a pop-up menu associated with the Record menu. It appears when you choose Record from the menu.

This is another pop-up menu. When not associated with a menu, a pop-up menu can appear anywhere in the form.

Working with menus

Working with menus involves the following basic tasks:

- Building the menu
- Displaying the menu
- Processing menu choices

When you want to go beyond the basics, you can

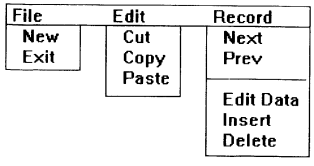
- Specify menu item ID numbers
- Specify the display attributes of menu items
- Provide keyboard access to menu items
- Inspect items in a menu

Unlike the menus built into Paradox, menus created using ObjectPAL don't persist from form to form in an application. If you create a custom menu for a form, the menu appears for that form only. If you then open a second form, the second form uses the built-in menus by default, *not* the menu you created for the first form.

Note You can make two (or more) forms to use the same custom menu structure. Techniques are presented in Chapter 14.

The following example shows how to build, display, and process choices from the menu shown in Figure 2.11. These basic techniques are presented in the context of a one-form, one-page application.

Figure 2.11 The finished menu for this part of the lesson



This figure shows the pop-up menus associated with each menu option. Normally, you can display only one pop-up menu at a time.

Building a menu

This part of the lesson shows how to write code to build and display the menu shown in Figure 2.11. The first step is to create a single-record form bound to the *Customer* table.

Example 2.27 Creating the form

1 Choose File | New | Form to open the New Form dialog box.

2 Add CUSTOMER.DB to the form's data model, then choose OK to close the New Form dialog box. The Design Layout dialog box opens.

3 Click OK to accept the default layout and close the dialog box.

4 Paradox displays the new form in a design window.

Example 2.28 Attaching code

This is a one-form, one-page application, so attach the menu-building code to the page. As a general rule, this is the best place to put menu-building code. (In a multi-page form, if you want all pages to share the same menu, attach code to the form.)

1 Inspect the page and choose Methods to open the Method Inspector.

2 Choose **arrive** to open an Editor window for the page's built-in **arrive** method. The **arrive** method is recommended over the **open** method because of the flexibility it allows. For example, suppose you later want to add a page to the form, and that page uses a different menu. Code attached to **arrive** executes each time you move to that page, where code attached to **open** executes only once, when the page opens.

3 Edit the method to make it look like this:

```
method arrive(var eventInfo MoveEvent)
   var
      mainMenu Menu
      filePop, editPop, recordPop PopUpMenu
   endVar

; build the File menu
   filePop.addText("New")
   filePop.addText("Exit")
   mainMenu.addPopUp("File", filePop)

; build the Edit menu
   editPop.addText("Cut")
   editPop.addText("Copy")
   editPop.addText("Paste")
   mainMenu.addPopUp("Edit", editPop)

; build the Record menu
   recordPop.addText("Next")
   recordPop.addText("Prev")
   recordPop.addSeparator()
   recordPop.addText("Edit Data")
   recordPop.addText("Insert")
   recordPop.addText("Delete")
```

```
    mainMenu.addPopUp("Record", recordPop)

  ; display the menu
    mainMenu.show()
  endMethod
```

As stated earlier, an application menu consists of items listed horizontally in the menu bar and pop-up menus associated with the items. This example code begins by declaring variables for three pop-up menus and one variable for the main menu. Next, the code uses **addText** statements to add items to the first pop-up menu. Items appear in the menu in the order they're added. In this example, the item "New" is added first, so it appears first, and the item "Exit" appears below it.

After all the items are added to a pop-up menu, the next step is to associate it with an item in the menu bar. The following statement calls **addPopUp** to associate the pop-up menu represented by *filePop* with the item "File" in the main menu.

```
mainMenu.addPopUp("File", filePop)
```

The rest of the menu is built the same way: call **addText** and (optionally) **addSeparator** to build a pop-up menu, then add the pop-up menu to the main menu.

The call to **addSeparator** adds a horizontal line to the menu to separate the items before and after it.

Finally, the call to **show** displays the menu.

Note When you create a menu, it replaces Paradox's built-in menu. You *cannot* add items to Paradox's built-in menus.

Displaying the menu

As shown in the previous example, a call to **show** displays a menu. The new menu remains in place until you call **show** to display another menu, call **removeMenu** (which restores Paradox's built-in menus), or close the form (which also restores the built-in menu).

Processing menu choices

When you choose an item from a menu, it triggers the built-in **menuAction** method of the active object. By default, the active object bubbles the event to its container, and so on, until the event reaches the form, and the form's default code handles it. See Chapter 10, "Understanding the event model," for a detailed discussion of this concept.

You could attach all your menu-handling code to the form, but there's a tradeoff in terms of modularity and flexibility. By attaching code to lower-level objects (in this case, to the page), you can add and delete, cut, copy and paste objects within and between forms without having to maintain large blocks of code at the form level.

Example 2.29 Attaching code to process the menu choices

The following code is attached to the page's built-in **menuAction** method to handle menu choices. When you choose an item from the menu built in the previous example, Paradox returns a string containing the item you chose. For example, when you choose

File | New, Paradox returns "New". The *eventInfo* variable for **menuAction** contains this information, and you extract it using **menuChoice**.

Important **menuChoice** returns the item string exactly as specified in the **addText** statement, including upper- and lowercase letters, spaces, punctuation, and special characters.

```
method menuAction(var eventInfo MenuEvent)
   var
      theChoice String
      formVar Form
   endVar

   theChoice = eventInfo.menuChoice()

   switch
   ; File menu
      case theChoice = "New"  : formVar.create() ; create a new form
      case theChoice = "Exit" : close()          ; close this form

   ; Edit menu
      case theChoice = "Cut"   : active.action(EditCutSelection)
      case theChoice = "Copy"  : active.action(EditCopySelection)
      case theChoice = "Paste" : active.action(EditPaste)

   ; Record menu
      case theChoice = "Next"      : active.action(DataNextRecord)
      case theChoice = "Prev"      : active.action(DataPriorRecord)
      case theChoice = "Edit Data" : active.action(DataToggleEdit)
      case theChoice = "Insert"    : active.action(DataInsertRecord)
      case theChoice = "Delete"    : active.action(DataDeleteRecord)
   endSwitch
endMethod
```

How it works

Here's the basic technique for processing menu choices:

1. Attach code to the page's built-in **menuAction** method.
2. Call **menuChoice** to find out which item was chosen.
3. Call the UIObject type **action** method with an ObjectPAL constant to specify a response.

For example, when you choose Edit | Cut, the following statement triggers a response from the current object:

```
active.action(EditCutSelection)
```

The action constant EditCutSelection says, in effect, "Cut the selected text (if any) from the active object to the Clipboard."

You don't have to use **action** and action constants to respond to menu choices. You can use methods and procedures from the run-time library, you can call methods attached to other objects, and you can use custom methods and custom procedures. For example, in the previous code, when you choose File | New, the following statement executes to create a new form:

```
formVar.create() ; create a new form
```

Summary

This lesson introduced you to the basics of programming custom menus in ObjectPAL. It showed you how to

- Use **addText** and **addPopUp** to build the menu, attaching code to a page's built-in **arrive** method
- Use **show** to display the menu
- Process the user's menu choices by attaching code to the page's built-in **menuAction** method

Lesson 10: System procedures

ObjectPAL's System type procedures display messages, find out about the user's system, manipulate the File Browser, work with the Help system, and more. This lesson shows how to employ some of the commonly used procedures of the System type.

The System type: a catch-all type

The System type is a catch-all for procedures that don't fit into other categories. Some of the System procedures are

- **beep** and **sleep**
- **message,** which displays a message in the status line
- **execute**, which executes DOS commands and Windows applications; for example, `execute("clock.exe")` opens the Windows clock application
- **msgInfo** and **msgYesNoCancel** for displaying ObjectPAL's built-in dialog boxes
- **helpShowTopic, helpShowContext,** and other help procedures (beginning with the letters **help**) to access your Help system once it's built and compiled
- **enumFontsToTable, readEnvironmentString, readProfileString,** and **sysInfo** that report about the user's system the application is running on

Message dialogs

The System type provides several message procedures that display built-in Paradox dialog boxes. The most basic message procedure is **msgInfo**, which displays a dialog box containing the information icon, a message, and an OK button.

The following example shows how to use the **msgQuestion** procedure to prompt the user to respond to a Yes/No choice.

Example 2.30 Prompting the user for a confirmation

1 Create a blank, unbound form.

2 Place a button anywhere on the form.

3 Inspect the button, and choose Methods to open the Method Inspector.

4 Double-click **pushButton** to open an Editor window for the built-in **pushButton** method.

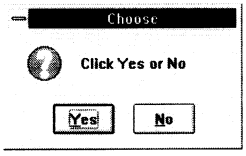
5 Edit the method to look like this:

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

userChoice = msgQuestion("Choose", "Click Yes or No")
switch
   case userChoice = "Yes" :
      message("You chose yes.")
   case userChoice = "No" :
      message("You chose No")
   otherwise: message("You closed the dialog box.")
endSwitch
endMethod
```

6 Check your syntax, and correct any errors.

7 Click the View Data button or choose Program | Run to run the form.

8 Click the button. The **msgQuestion** dialog box appears.



9 Click Yes button. The message `You chose Yes` appears in the status bar.

10 Click the button again.

11 Click No button. The message `You chose No` appears in the status bar.

12 Close the form.

How it works

The code in this method is organized into three blocks. The first block declares the variable for the **msgQuestion** dialog box. The next line of code

```
userChoice = msgQuestion("Confirm", "Are you sure you want to delete this record?")
```

displays a two-button dialog box. The text in the first set of quotes is the caption displayed in the title bar, and the text in the second set of quotes appears in the box itself. The return value corresponds to the button the user clicks to close the dialog box: Yes or No.

The next block of code is

```
switch
   case userChoice = "Yes" :
      message("You chose yes.")
   case userChoice = "No" :
      message("You chose No")
   otherwise: message("You closed the dialog box.")
endSwitch
```

switch...endSwitch checks the incoming return value that corresponds to the button the user clicks. If the user clicks the Yes button, the statement that corresponds to the first case condition executes and displays the specified message in the status bar. If the user clicks the No button, the statement corresponding to the second case condition executes and displays its corresponding message in the status line. If the user presses *Esc* or clicks the Close box (or presses *Ctrl+F4*), neither case condition is True, and the (optional) otherwise clause is executed and displays the specified message.

Interactive dialogs

The ObjectPAL System class includes a range of procedures for invoking Paradox interactive dialog boxes for performing common file operations. These procedures begin with the letters **dlg**, for example **dlgAdd** and **dlgCopy**. See the online ObjectPAL Help for a complete list of dialog procedures.

Note Once you display a **dlg** dialog box, you cannot control it through ObjectPAL. It is up to the user to enter any necessary data and close the dialog box.

This example shows how to let the user sort, copy, and empty a table by calling the **dlgSort, dlgCopy, dlgEmpty,** and **dlgDelete** dialog procedures from a form.

Example 2.31 Invoking Paradox dialog boxes

1 Create a blank, unbound form.

2 Use the Button tool to create a button. Place the button at the top left of the form, and change its label to *Go*.

3 Select the button and press *Ctrl+Spacebar* to display the Method Inspector.

4 Open the Editor for the **pushButton** method by double-clicking it.

5 Edit the method to make it look like the following.

```
const
   kOneInch = 1440
endConst

proc tfSetup(stTbName String)
var
   tf UIObject
endVar
tf.create(TableFrameTool, kOneInch, kOneInch, 2 * kOneInch, 2 * kOneInch)
tf.TableName = stTbName
tf.Visible = Yes
msgInfo("Continue", "Click OK to continue.")
```

```
tf.Visible = No
tf.TableName = ""
dmRemoveTable(stTbName)
tf.delete()
endProc

method pushButton(var eventInfo Event)
var
   contactTb Table
endVar

tfSetup("CONTACTS")
dlgSort("CONTACTS")

tfSetup("CONTACTS")
dlgCopy("CONTACTS")

tfSetup("CONTACT2")
dlgEmpty("CONTACT2")

tfSetup("CONTACT2")

contactTb.attach("CONTACTS")
sort contactTb
endSort
dmRemoveTable ("CONTACTS")
msgInfo("Finished", "Click OK to continue.")
endMethod
```

6 Check your syntax, and correct any errors.

7 Save the form and name it CONTACTS.

Now have a look at what this code does when you run the form.

Example 2.32 Calling Paradox dialog boxes from a form

1 Run the form and click the Go button. The form displays CONTACTS.DB in a Table frame. Note that the table is sorted by Last Name.

2 Click OK. The Table Sort dialog box appears.

3 Double-click Company in the FIELDS: list box and click OK. The form displays CONTACTS.DB again in a Table frame. Note that the table is now sorted by Company.

4 Click OK. The Copy dialog box appears.

5 Enter CONTACT2 in the TO: text box for the new table to create and click OK. The form displays the new table *Contact2* in a Table frame.

6 Click OK. The Empty dialog box appears.

7 Click OK and confirm that you want to empty the *Contact2* table. The form displays the now-empty *Contact2* table.

8 Click OK. The Delete dialog box appears.

9 Click OK and confirm that you want to delete CONTACT2.DB.

10 Close the form.

How it works

The first block of code defines the constant *kOneInch* with the value of 1440, the number of twips in an inch. Twips, which are one twentieth of a printer's point, are used to measure the *x* and *y* coordinates of a point on the screen.

The second block of code declares the custom procedure **tfSetup**. This procedure begins by declaring a UIObject variable for a Table frame. The next three lines of this block create and display a table frame containing all the fields from the *Contacts* table. The coordinates in the **create** statement specify the table frame's position and size. The next line uses the System procedure **msgInfo** to display a one button dialog box. When you click OK to continue, the table frame is deleted and the table it displayed is removed from the form's data model. The last line in the procedure deletes the table frame.

The **pushButton** method declares a Table variable and then calls the procedure **tfSetup** to display CONTACTS.DB in a table frame. The next statement calls the Paradox Table Sort dialog box, just as if you had chosen Table | Sort. ObjectPAL code suspends execution until you close this dialog box.

These two statements are repeated to call Paradox Copy, Empty, and Delete dialog boxes. After you copy the *Contacts* table, the new table is displayed in a table frame. And after you empty the new table, the empty table is displayed in a table frame. The Paradox Delete dialog box is then called so you can delete the new table.

The last block of code attaches to the *Contacts* table and restores it to its original order.

More System procedures

The System type includes several miscellaneous procedures that you'll find handy to use occasionally. The next three examples show how to use the following System type procedures:

- **fileBrowser** displays the Paradox built-in Browser and returns the names of one or more files selected by the user.
- **enumRTLMethods** creates a table listing the methods in ObjectPAL.
- **sysInfo** creates a dynamic array of information about the system running Paradox.

To work through this example, use the *Contacts* form you already created.

Calling the Browser

In this lesson you will use the **fileBrowser** procedure to invoke the Browser, and then use the name of the file the user has selected to display that file in a table frame. Keep in mind that you can also declare a resizeable array variable to return multiple file names (selected by the user *Shift*+clicking). See the online ObjectPAL Help for more information about **fileBrowser**.

Example 2.33 Using the fileBrowser procedure

1 Open the *Contacts* form in the Form Design window.

2 Select the Go button and then choose Edit | Copy.

3 Paste the copied button beside the Go button and change its label to *Browse*.

4 Edit the Browse button's **pushButton** method to make it look like the following:

```
method pushButton(var eventInfo Event)
var
   selectedFile   String
   TF1            UIObject
   fbi            FileBrowserInfo
endVar
   fbi.Alias = "WORK"
   fbi.AllowableTypes = fbTable
   if fileBrowser(selectedFile, fbi) then
      tf1.create(TableFrameTool, 1440, 1440 , 1440*2, 1440*2)
      tf1.tableName = selectedFile
      tf1.visible = yes
      msgInfo("Continue", "Click OK to continue")
      tf1.visible = no
      tf1.tableName = ""
      dmRemoveTable(selectedFile)
      tf1.delete()
   else
      message("You selected cancel")
   endIf
endMethod
```

5 Check your syntax, and correct any errors.

6 Save the form and run it.

7 Click the Browse button. The Paradox Browse window appears. ObjectPAL execution suspends until the user closes the Browser.

8 Select a table and click OK. The table you selected is displayed in a table frame.

9 Click OK.

How it works

The first block of this code declares a String variable for the user's selection in the Browser window, a UIObject variable for a table frame to display the selected table, and a variable of the special data type FileBrowserInfo. This data type, used only with the **fileBrowser** procedure, lets you pass a record as an argument to **fileBrowser** to specify various criteria about what you want the Browser to display. FileBrowserInfo is a predeclared record with fields that let you determine the size and style of the Browser window, and the type, location, and filespec of the files to display. In this example, values are passed to the *Alias* and *AllowableTypes* fields of the FileBrowserInfo variable:

```
fbi.Alias = "WORK"
fbi.AllowableTypes = fbTable
```

The value assigned to the *Alias* field determines that the current working directory will be searched, and the value assigned to the *AllowableTypes* field specifies that the Browser will search only for Paradox tables.

The next line begins an `if` block that uses the name of the file you selected to display that file in a table frame:

```
if fileBrowser(selectedFile, ext) then
```

Creating a table of ObjectPAL methods

In this example you will use the **enumRTLMethods** procedure to create a table listing all the methods defined in the ObjectPAL run-time library. ObjectPAL provides several other procedures to keep track of the objects and methods in a form. See the online ObjectPAL Help for information about other **enum** procedures in the System, Form, Table, TCursor, and UIObject classes.

Example 2.34 Using enumRTLMethods

1. Return to the Design window.
2. Select the *Browse* button and then choose Edit | Copy.
3. Paste the copied button beside the *Browse* button and change its label to *Methods*.
4. Edit the *Methods* button's **pushButton** method to make it look like the following:

```
method pushButton(var eventInfo Event)
var
   TFl          UIObject
endVar
   enumRTLMethods("rtlmeth.db")
   tfl.create(tableFrameBox, 1440, 1440 , 1440*2, 1440*2)
   tfl.tableName = "rtlmeth"
   tfl.visible = yes
   msgInfo("Continue", "Click OK to continue")
   tfl.visible = no
   tfl.tableName = ""
   deleteTable("rtlmeth")
   tfl.delete()
endMethod
```

5. Check your syntax, and correct any errors.

6. Save the form and run it.
7. Click the *Methods* button. A table frame displays the list of run-time methods.
8. Click OK.

By default, this table is created in the working directory (:WORK:). The fields in the created table note the class to which the method belongs, its type (method or procedure), and its name, arguments, and the type of value it returns.

To create a smaller table listing only the Beginner level methods defined in the ObjectPAL run-time library, include the following statement before the **enumRTLMethods** procedure:

```
setUserLevel("Beginner")
```

Getting system information

In this example you will use the **sysInfo** procedure to create and display an array of information about your system.

Example 2.35 Using the sysInfo procedure

1 Return to the Design window.

2 Select the *Methods* button and then choose Edit | Copy.

3 Paste the copied button beside the *Methods* button and change its label to *System*.

4 Edit the *System* button's **pushButton** method to make it look like the following:

```
method pushButton(var eventInfo Event)
var
   userSys DynArray[] AnyType
endVar
sysInfo(userSys)
userSys.view()
endMethod
```

5 Check your syntax, and correct any errors.

6 Save the form and run it.

7 Click the *System* button. The system information appears in a **view** dialog box.

8 Click OK to close the **view** dialog box.

You must declare the DynArray variable before calling **sysInfo**. The dynamic array contains indexes for system attributes and their values. The DynArray variable declared in this example is of AnyType, a catch-all data type that lets you design methods for a variety of data types when you can't predict the data type of the actual value until the method executes. See Chapter 5 for more information about dynamic arrays.

To open a window in the center of the display, use the information **sysInfo** returns about the display resolution for a system to calculate the correct screen coordinates. See the online reference information in the ObjectPAL Help for the **pixelsToTwips** System type procedure for an example of this.

Summary

The System type includes a wide variety of procedures that you will find useful for both developing and running your applications. Be sure to review the other System type procedures in the online ObjectPAL Help. This lesson showed you how to use

- The **msgQuestion** procedure to prompt the user to respond to a Yes/No choice
- The **dlgSort, dlgCopy, dlgEmpty,** and **dlgDelete** procedures to invoke Paradox interactive dialog boxes from a form
- The **fileBrowser** procedure to invoke the Paradox Browser, letting the user select a file from the Browser and returning that name the selected file

- The **enumRTLMethods** procedure to create a table listing all the methods defined in the ObjectPAL run-time library
- The **sysInfo** procedure to create an array of information about your system, and display this information in a **view** dialog box

Part

ObjectPAL basics

This part of the manual presents the Integrated Development Environment (IDE) and gives an overview of how to work with the language.

It includes the following chapters:

- Chapter 3, "ObjectPAL overview," gives an overview of objects, events, methods, and object-based programming.
- Chapter 4, "Language structure," discusses the general structure and syntax of ObjectPAL code, including rules for naming and using variables and constants.
- Chapter 5, "Data types," presents the ObjectPAL data types: AnyType, Array, Binary, Date, DateTime, DynArray, Graphic, Logical, LongInt, Memo, Money, Number, OLE, Point, Record, SmallInt, String, and Time.
- Chapter 6, "Variables, constants, and containership," describes how to define and reference ObjectPAL variables and constants, and explains how a design object's behavior is defined by its containership.
- Chapter 7, "Where to put code," discusses one the most critical aspects of ObjectPAL: deciding where to place code. This chapter offers guidelines, tips, and techniques on where and when to code.
- Chapter 8, "The ObjectPAL Editor," explains how to use the built-in Paradox Editor to create and modify ObjectPAL code.
- Chapter 9, "The ObjectPAL Debugger," explains how to use the built-in debugging tool to test your code.

ObjectPAL overview

This chapter gives you a conceptual overview of the benefits and structure of the ObjectPAL language, explains the terminology of objects, presents a strategy for object-based programming, and points you to the parts of this manual where you can find the information about common programming tasks.

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 or in the following chapters.

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.

Programming in ObjectPAL

ObjectPAL is similar to traditional languages because it uses variables; provides control structures like **if...then...else** constructs, **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.

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 most 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 Toolbar and even reset the main title of the Paradox Desktop.

Manipulating tables

Any table action that is normally available interactively is 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 can also create query statements from scratch in ObjectPAL; these queries can include variables evaluated at run time.

Object-based programming

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 programmers' 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.

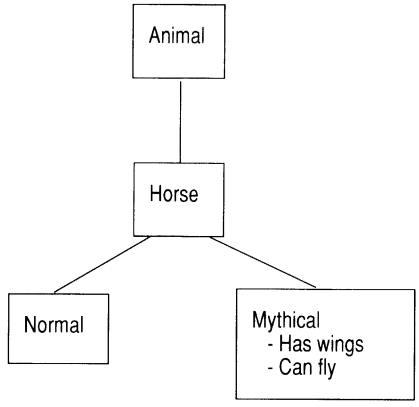
The language of objects

You already know a great deal about objects and object types, whether you realize it or not. People already think and speak in terms of types of things—for example, animal, vegetable, or mineral. Types have *hierarchies*; for instance, a horse is a type of animal.

Objects are classified by *type*; that is, Pegasus is a kind of horse. Pegasus is thus an object of type horse of type animal.

In conversation you'd say that Pegasus is a mythical horse that can fly. In object-based terms you'd say Pegasus is an object of type mythical horse (of type horse, of type animal) that inherits all the *properties* and functions of a horse, and adds some unique functions, like flying, as diagrammed in Figure 3.1. The object-based paradigm lets you organize things in a program in much the same way they're organized in conversation.

Figure 3.1 A hierarchy of types and objects



In object-based terms, if the designer of the type horse wrote the type correctly, you could tell the object Pegasus to jump with the statement

```
Pegasus.jump()
```

without knowing anything about what it took to make that horse jump. All you need to know is that Pegasus is a horse, and you want Pegasus to jump.

The procedural approach

In procedural programming, functions dictate program organization. For example, suppose you're working with two types of animal: horses and dolphins. The process of getting a horse to jump is quite different from getting a dolphin to jump. In a procedural language, the function **jump** would be coded only once, something like this:

```
jump(theAnimal)
   if theAnimal = "Horse" then
      runDirection = forward
      frontLegs(lift)
      backLegs(push)
   else if theAnimal = "Dolphin" then
      swimDirection = up
      swimSpeed = fast
      tail(kick)
      else
         message("Wrong data passed to this function.")
      endIf
   endIf
```

In this example, one function must handle a variety of different types of data. Decision making is handled inside the function, and its effectiveness is largely a matter of the programmer's ability to foresee how many different kinds of data the function will have to sort out. As programs evolve, the **jump** function will be used for a greater variety of animals, some of which the programmer may have foreseen, some not. In a procedural language, you write one function to do many different things.

The object-based approach

In an object-based language, you let the language processor decide which *method* to use based on the type of object that calls the method. The underlying program code is written specifically for objects. Expressed in object-based programming terms, the general method would be called **jump**, but there would be one method for horses and another for dolphins. So, you would specify which animal you want to jump. For example,

```
horse.jump()
```

to make a horse jump, or

```
dolphin.jump()
```

to make a dolphin jump. It's that simple. You just have to remember what animal you're working with (horse or dolphin), and what you want it to do (jump). The language takes care of the rest.

Objects

Like the animals in these examples, ObjectPAL objects are organized into types. Objects of a given type have the same properties and methods. For example, all text files have properties in common, and all tables have properties in common, but the properties of tables and text files are different. Therefore, text files and tables are objects of different types, and you use one set of methods to operate on text files, and another set to operate on tables.

Methods for different object types can have the same names. Just as our example used **jump** for both horses and dolphins, ObjectPAL uses one **open** method to open a table and another to open a text file. ObjectPAL syntax resembles human language. For example, you might think, "I want to open a table," or "I want to open a text file." The underlying code that opens a table is different from the code that opens a text file, but the verb "open" remains the same and it's all you need to remember. ObjectPAL runs the **open** method appropriate for the type of the object. You don't have to remember two different commands, something like OpenTable and OpenTextFile, as you would with a procedural language.

Also, code that operates on an object doesn't have to be attached to the object. For example, the code to open a table could be attached to a button, a field object, or any other object you can place in a form. When the code executes, it opens the table. Where you attach the code depends on when you want the code to execute.

It's worth noting, too, that not every method is appropriate for every object type. For example, it wouldn't make sense to do **slug.jump()**.

The same is true for Paradox and ObjectPAL: objects of different types respond to events differently, and not every method is appropriate for every object type. The effects of an ObjectPAL statement like the following will vary depending on the type of *thisObject*.

```
thisObject.open("orders")
```

As you develop applications, you'll use one set of methods to operate on tables and another set to operate on text files, because tables and text files are objects of different types.

When you use ObjectPAL, ask, "What *object* am I working with?" and "What do I want it to *do*?" When you answer these questions, you'll know which object type to use, and which methods.

Types

ObjectPAL groups all the objects that you draw using the Toolbar 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.

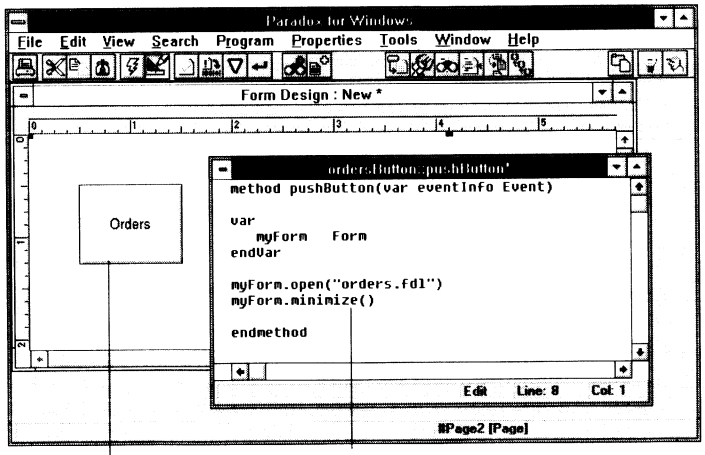
Paradox makes finding and modifying an object's built-in methods easy. You inspect the object to display its menu, choose Methods, and choose the built-in method you want to modify. An ObjectPAL Editor window opens, and Paradox positions the cursor at the point where you should start typing.

An object-based strategy

Creating Paradox applications is largely a process of placing *objects* in forms and writing ObjectPAL *methods* to define how those objects behave. Such applications are sometimes called "Hey, you! Do this" applications. (If this sounds too bossy, think in terms of objects and actions, or nouns and verbs.) First, identify an object, then specify an operation, as shown in Figure 3.2.

Note To see a complete list of ObjectPAL types and their methods, choose Tools | Types to open the Types and Methods dialog box. Select an object type to see a list of methods and procedures for that type.

Figure 3.2 Specifying objects and actions (Hey you, do this)



Clicking this button tells the button to execute its built-in **pushButton** method.

Within the method, this line of code specifies an object (*myForm*) and an action (**minimize**).

In a traditional, non-interactive procedural program, the program code controls what happens and when. Program execution is usually linear. For example, a program might display a table, then do a calculation, then display the results, and so on, all under program control.

ObjectPAL is different. An ObjectPAL application puts users in control: users interact with objects—buttons, tables, menus, and more—in any order they choose. The procedural approach no longer applies; you need to adopt an object-based strategy.

Planning is important

It has been said of object-based applications development that the hardest line of code to write is the first. That's true of ObjectPAL, not because the language is inherently difficult, but because up-front planning is such an important phase of the development process. Before you write a single line of code, you need a clear idea of what you want the application to do, what objects you'll use, and how the user will interact with those objects. The more complicated your application, the more important this phase becomes. To develop a Paradox application, follow these steps:

1 Set a goal.

2 Build tables.

3 Design the form.

4 Attach code.

Set a goal

The first step in building an application is to set a goal. Think about the users: what do they want to do? Start with a general answer; for example, a user might want to process

information about customers and orders. The general goal of the application, then, would be to enable the user to enter, edit, and retrieve relevant data.

When you're building a large application, you may not be able to express the goal so succinctly. A good approach is to analyze what you want to accomplish and define several subgoals. A subgoal may warrant its own form, or its own page in a multi-page form. Paradox and ObjectPAL are ideal tools for creating this kind of modular application.

Build tables

The next step is to build tables to store the data, unless (of course) you already have the data you need. Often you'll find that both situations are true. For example, if you're building an order-entry application, you may already have tables of information about customers, but will need to build tables to store the order data. You can use data stored in a variety of formats—for example, as a text file or as spreadsheet data—but you can take full advantage of the power of Paradox by storing the data in tables.

If your application uses more than one table, you'll need to set up a data model; that is, determine how the tables are related, and which fields you'll use to link them.

Design the form

Once you've set a goal and built tables, you're ready to design a form (or forms) for the user to interact with. Decide which objects to provide to accomplish the goal—buttons, fields, table frames, multi-record objects—whatever the user needs to get the job done. The more you know about the properties and capabilities of these objects, the better-equipped you'll be to make design decisions.

Important After you've placed all the objects, run the form and observe how the objects behave by default, before you attach any code. In most cases, the default behavior will be what you want. You have to attach code to an object only if you want it to do something different.

Attach code

Every design object comes with built-in code, so when you place objects in a form, you're literally programming. Only when you want an object to do something different (or something more) do you have to attach custom ObjectPAL code. At this stage, more planning is necessary because you'll be faced with this important question:

- Where should I put my code?

To answer it, you need to answer three other questions:

1. When should the code execute?
2. How many objects will use this code?
3. Which method should I use?

See Chapter 7, "Where to put code," for a discussion of the issues involved with these questions and some guidelines and tips on where to attach your code.

Object categories

ObjectPAL groups types into the categories listed in Table 3.1.

Table 3.1 Categories and object types

Category	Types
Events	ActionEvent, ErrorEvent, Event, KeyEvent, MenuEvent, MouseEvent, MoveEvent, StatusEvent, TimerEvent, ValueEvent
Design objects	Menu, PopUpMenu, UIObject
Display managers	Application, Form, Report, TableView
Data types	AnyType, Array, Binary, Money, Date, DateTime, DynArray, Graphic, Logical, LongInt, Memo, Number, OLE, Point, Record, SmallInt, String, Time
Data model objects	Database, Query, SQL, Table, TCursor
System data objects	DDE, FileSystem, Library, Session, System, TextStream

Figure 3.3 also shows how the object types are grouped into categories, and how the categories are related in an ObjectPAL application.

Events

The object types that handle *events* are shown at the top of Figure 3.3, because events set ObjectPAL in motion. A user interacting with an application generates events, and methods attached to objects execute in response to events.

If you're wondering how something as abstract as an event can be an object, it's because an event consists of data and code, and so fits the definition of an object. The data include the kind of event (for example, mouse click or key press), what happened (for example, which mouse button or which key was pressed), where it happened (which object was the target), and why it happened (for example, did the user do something, or was the event generated from within ObjectPAL). The code is the ObjectPAL method for extracting the data.

Design objects

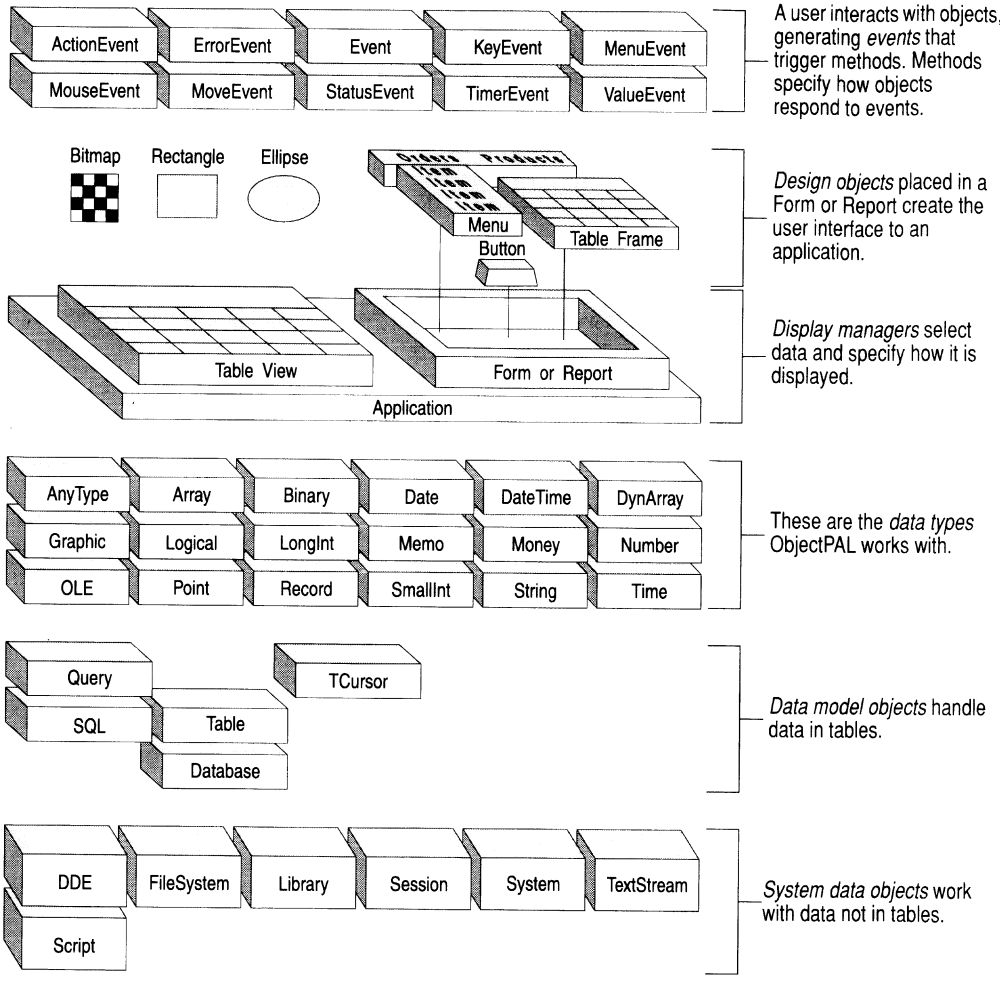
At the next level of Figure 3.3 are the design objects, because *design objects* contain the code that executes in response to events. Design objects include the menus, pages, buttons, boxes, table frames, and other objects in a form—including the form itself—that the user interacts with (Figure 3.3 doesn't show them all). The types in this category are UIObject, Menu, and PopUpMenu. You can attach code to UIObjects only.

Display managers

Next in Figure 3.3 come the *display managers*, because display managers contain design objects. Display managers are windows, such as forms and reports, that display data. The difference between forms and reports is that you can't attach code to reports. Methods for the TableView type let you control the Table window, and methods for the Application type let you control the Paradox Desktop.

Figure 3.3 Components of an application

ObjectPAL methods can address components at every level, binding them together to create graphical, event-driven applications.



Data types

At the next level, Figure 3.3 shows the basic ObjectPAL *data types*. Using these data types, you declare variables to store and manipulate data in tables. Of course, you can also use these basic data types to perform calculations and operations without getting the data from tables or displaying the results to the user.

Data model objects

Data model objects come next in Figure 3.3, because they handle data stored in tables. The Table, Query, and TCursor types access and manipulate the data; the Database type handles collections of tables.

System data objects

The *System data objects* are at the bottom of Figure 3.3. These objects store and access data about Windows, DOS, user counts, and the user's computer system, but *not* data stored in tables.

Summary

When you use a Paradox application, you generate events by interacting with design objects, which are contained in display managers. Together, design objects and display managers create the interface to an application. You can use data type objects to declare variables to perform calculations and manipulate data in tables, and system data objects to provide system-level data. ObjectPAL can work with objects at any level.

Where do I go from here?

Although we strongly recommend reading the chapters in this manual in order, especially if you are new to event-driven programming, experienced programmers may want to skip ahead. This section points you toward those parts of the documentation that explain how to accomplish common programming tasks (you should also check the example files, the index, and the online Help).

Tables

Before you use ObjectPAL to manipulate tables, you should know about the Table, TCursor, and TableView types, and you should know about the following UIObjects: table frame, multi-record object, and field object. Table 3.2 summarizes the functions of these objects.

Table 3.2 Objects for working with tables

Type	Description
Table	A description of a table: location, structure, type, and so on.
TCursor	A pointer to a table, stored and manipulated in memory.
TableView	A table displayed in its own window.
UIObject	A table frame displays a table in rows and columns. A multi-record object displays fields in one or more records of a table. A field object displays data from one field of a table.

For information about these object types, see the following sections:

- For information about creating and restructuring a table and to learn how to place a table frame on a form and bind it to a table, see the *User's Guide*.

- Tables and TCursors are described in Chapters 17 and 18.
- Table frames, multi-record objects, and field objects are UIObjects and are described in Chapter 13.
- TableViews are described in Chapter 20.
- The following example applications:
  - Connections
  - Address Book
  - Checkbook
  - Slot Machine

Queries

You can use ObjectPAL to create and run both queries you code from scratch and queries created using Paradox interactively.

- The methods for working with queries are described in Chapter 22.
- The address book application shows one way to show query results in a form.

Messages and dialog boxes

Messages and built-in dialog boxes give you a way to interact with a user. See the following:

- Examples of **msgInfo** and **msgQuestion** in Lesson 10 of the ObjectPAL tutorial in Chapter 2
- The entries for the System type procedures **msgAbortRetryIgnore, msgRetryCancel, msgStop,** and **msgYesNoCancel** in the online ObjectPAL Help
- Chapter 26

Keyboard events

You can respond to any key press with ObjectPAL, which means you can easily develop shortcut keys for your application. See the following sections:

- The discussion of the event model in Chapter 10
- **keyPhysical** and **keyChar** in Chapter 12
- The Checkbook example application

Mouse events

You can use ObjectPAL to handle the full range of mouse actions: clicks, double-clicks, movements, and more. See the following:

- The "MouseEvent" section of Chapter 10
- The following example applications:
  - Connections
  - Address Book
  - Paradox Paint

Menus

Using ObjectPAL, you can define menus and pop-up menus to display choices to users. See the following chapters and sections:

- The "MenuEvent" section of Chapter 10
- Lesson 9 of the ObjectPAL tutorial in Chapter 2
- Chapter 14
- The following example applications:
  - Connections
  - Slot Machine

Lists

You can use list boxes and drop-down edit boxes to let a user choose from a group of items. Place a field object and set its Display Type to List or Dropdown Edit. To find out how to manipulate these lists, see

- Lesson 8 of the ObjectPAL tutorial in Chapter 2

Multi-form applications

To design applications that use more than one form, you'll need to know how to open a form and control it from another form. Forms can be opened as dialog boxes, as well. See the following:

- The "Form" section of Chapter 20, and Chapter 26

Text

You can use the following ObjectPAL types to work with text: String (plain text), Memo (formatted text), and TextStream (text files). See the following sections:

- "String" in Chapter 5
- "Memo" in Chapter 5
- "TextStream" in Chapter 25

The file system

Methods in the FileSystem type provide access to and information about disk files, drives, and directories. ObjectPAL also includes a built-in Browser dialog box. For instructions on using the FileSystem methods, see the following:

- Chapter 24
- The example of the System type procedure **fileBrowser** in Lesson 10 of the ObjectPAL tutorial in Chapter 2

Code libraries

You can store custom ObjectPAL code in libraries, and make libraries available to one or more forms or applications. For more information about libraries, see the following:

- Chapter 28
- The topic "uses" in the online ObjectPAL Help

DLLs

With the **uses** clause, you can declare and subsequently use functions called from dynamic link libraries (DLLs). To find out how to link a DLL, see the following:

- The topic "uses" in the online ObjectPAL Help

Language structure

This chapter discusses the general structure and syntax of ObjectPAL methods, including rules for naming objects, methods, procedures, variables, and arrays. It covers

- The nature of ObjectPAL
- Dot notation
- Structure
- Control structures
- Data types
- Expressions
- Naming rules
- ObjectPAL terms

The nature of ObjectPAL

Experienced programmers may wonder how ObjectPAL applications differ from traditional procedural programs. The key differences are the program's context and its sequence of execution. The context for a traditional program is a single source file, and the sequence of execution is linear. In ObjectPAL, the context is defined by objects, and methods execute in response to events.

In terms of structure and syntax, ObjectPAL methods resemble traditional programs. For example,

- Methods are structured. In addition to their beginning-to-end sequence of execution, you can define an ordered structure of execution because ObjectPAL supports control structures and loops like while...endWhile, if...then...else...endIf, and switch...case...endSwitch.

- Methods can have parameters (also called arguments). For example, in the following statement, DataNextRecord is the parameter, **action** is the method, and *orderTable* is the object:

```
orderTable.action(DataNextRecord)
```

- As in Pascal and C, you can define procedures to perform one or more tasks. Procedures can receive arguments from, and return results to, the method that calls them.
- Also as in C, you can freely use whitespace (tabs, spaces, and blank lines). You can choose to indent subordinate method lines, you can put one or more statements on a line, and you can append a comment to any method line—whitespace has no effect on how statements are executed.

Dot notation

ObjectPAL uses dot notation, or periods, to separate elements in a statement. The following statement is read as "Box one dot color equals blue."

```
boxOne.color = Blue
```

The statement says that you want to work with the object named *boxOne*, and that you want to set the Color property of *boxOne* to blue. The dots separate the object name and the property name. Similarly, the next statement is read as "My form dot set title to 'Order form'."

```
myForm.setTitle("Order form")
```

The dot separates the variable *myForm* and the method name **setTitle**.

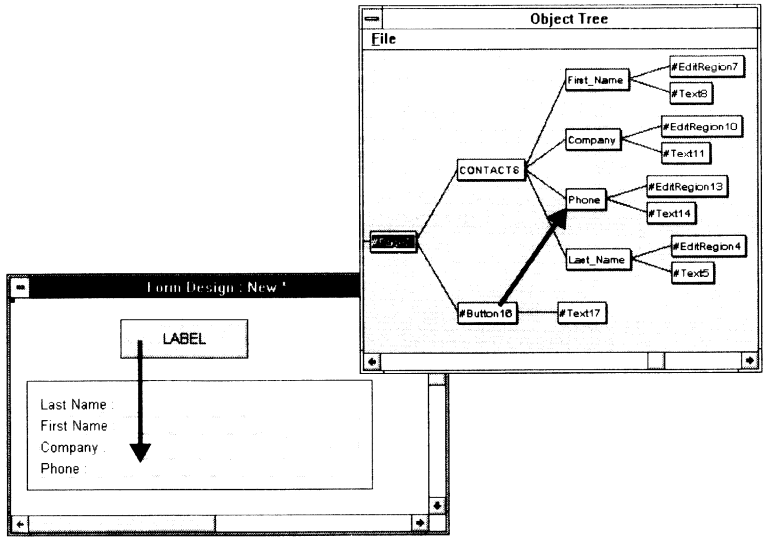
Important In text (as distinguished from code examples), this manual refers to methods and procedures by name only; it does not give the full syntax. For example, the previous paragraph refers to the **setTitle** method defined for the Form type. The **setTitle** method's complete syntax is

setTitle (const ***newTitle*** String)

Refer to the online ObjectPAL Help for the complete syntax for every ObjectPAL method and procedure.

For a more complex example of dot notation, see Figure 4.1, which shows a form containing a button and a table frame bound to the sample *Contacts* table. The figure also shows the Object Tree, which diagrams the relationships between the objects.

Figure 4.1 Addressing an object whose name is unique



In this form, there is only one object named *Phone*. So, the button can use the following statement to set *Phone*'s value:

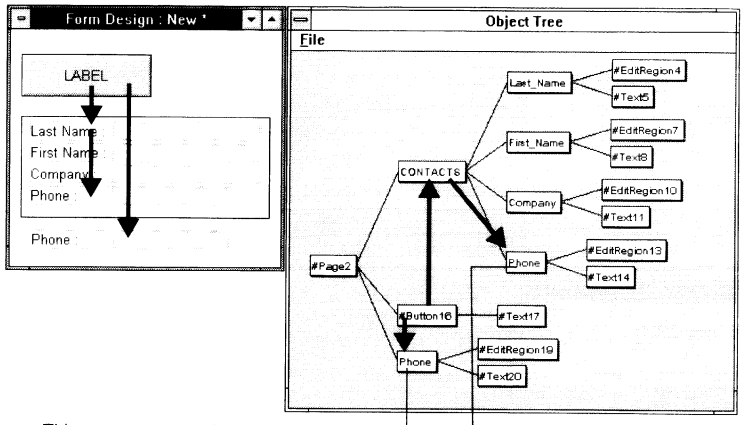
Phone.value = "800-555-0147"

Code attached to the button's **pushButton** method can use the following statement to set the value of the *Phone* field object in the *Contacts* table:

```
method pushButton(var eventinfo Event)
   Phone.value = "800-555-0147"  ; must be in Edit mode
endMethod
```

If an object has a unique name, you can address that object directly, as in the previous example. But, if one or more objects in a form have the same name, you must use dot notation to specify which object to work with. For example, Figure 4.2 shows a form containing the button and table frame from the previous example. Also, another field object, named *Phone*, has been added. Again, the Object Tree diagrams the relationships between the objects.

Figure 4.2 Addressing an object whose name is not unique



This form contains two objects named *Phone*. Use dot notation to indicate which Phone to work with.

This statement sets the value of the unbound field object named *Phone*:
Phone.value = "800-555-0147"

This statement sets the value of the *Phone* field object contained by the CONTACTS table frame:
CONTACTS.Phone.value = "800-555-0147"

Now, to set the value of the *Phone* field object in the *Contacts* table as before, you must use dot notation:

```
method pushButton(var eventinfo Event)
   Contacts.Phone.value = "800-555-0147"
endMethod
```

In this example, the following statement sets the value of the object named *Phone* that is contained by the object named *Contacts*:

```
Contacts.Phone.value = "800-555-0147"
```

To set the value of the other object named *Phone*, the code for the **pushButton** method would be

```
method pushButton(var eventinfo Event)
   Phone.value = "800-555-0147"
endMethod
```

Important Dot notation is recommended for clarity and consistency, but it's not required. As an alternative syntax, you can call the method and use the object name as the first argument, moving other arguments to the right as necessary. For example, the following statements are equivalent:

```
myCosine = theAngle.cos() ; dot notation, no arguments

myCosine = cos(theAngle)  ; alternative syntax, object name is the only argument
```

The following statements are also equivalent:

```
myText.open("notes.txt", "r")   ; dot notation, two arguments
open(myText, "notes.txt", "r")  ; alternative syntax, three arguments
```

Structure

ObjectPAL code has a free-form structure. With few exceptions, you can put as many statements on a single line as you want, each separated by at least one space.

Splitting lines

You can split an instruction between two or more lines, as long as you do not make the split in the middle of a keyword, name, or data value (such as a number or character string). Each line in an ObjectPAL method can be up to 132 characters. For example, these two code examples execute the same way:

```
if tc.locate("CustName", "John Culver") then
   tc.ProductName = "Paradox"
else
   tc.ProductName = "dBASE"
endIf
```

```
if
tc.locate("CustName", "John Culver")
then
tc.ProductName =
"Paradox" else
tc.ProductName =
"dBASE"
endIf
```

Case

You can use uppercase or lowercase letters, or any combination of the two. For example, ObjectPAL recognizes setFieldValue and SETFIELDVALUE and even SeTfIElDvAlUe as the same word (but it's less confusing to use case consistently). The same holds true for the names of tables, fields, variables, arrays, and procedures.

The only place where case matters in your methods is in string (alpha) data values. For example, the string "abc" is not equal to "ABC".

Note You can use **ignoreCaseInStringCompares**, defined for the String type, to specify whether you want case ignored when comparing strings.

Comments and blank lines

ObjectPAL provides two ways to put comments into your code:

- By using a semicolon
- By using braces

Using a semicolon

Unless contained in a string value, anything following a semicolon (;) on a method line is considered a comment and is ignored by ObjectPAL. If you begin a line with a

semicolon, the whole line is treated as a comment. Multiline comments must have a semicolon at the beginning of each line, as in this example:

```
method pushButton (var eventInfo Event)

; first, declare variables
var myNum SmallInt endVar

for myNum from 1 to 5
   myNum = myNum + 1 ; this is a counting loop
endFor

; this comment spans
; two lines

endMethod
```

Use comments to describe what's happening in a method; to identify cryptic messages, variables, or procedures; to provide any other information that might be useful to someone reading or editing the method; or just to remind you about what you've done. Adding comments to a method does not slow execution speed.

Blank lines and whitespace are ignored. You can use them to set off comments and make your methods more readable. You don't have to preface a blank line with a semicolon.

Using braces

You can use braces to comment out large blocks of code. Use a left brace ({) to begin a comment block, and a right brace (}) to end it. The next two examples comment out the same lines.

Example 1

```
var
   i SmallInt
endVar
{
for i from 1 to 10
   i = i + 1
endFor
}
msgInfo("Status", "You are here.")
```

Example 2

```
var
   i SmallInt
endVar

; for i from 1 to 10
;   i = i + 1
; endFor

msgInfo("Status", "You are here.")
```

Quoted strings

A quoted string that spans more than one line in a method spans the same number of lines when displayed. For example, the next statement:

```
msgInfo("2-line string", "This string
spans two lines.")
```

displays a dialog box containing this message:

```
This string
spans two lines.
```

Control structures

ObjectPAL executes statements in the order of their appearance in a method. But ObjectPAL also provides *control structures* you can use to change and rearrange the *flow of control*—the order in which statements execute.

ObjectPAL's basic language elements include control structures to perform the following tasks:

- *Branching* executes a statement or set of statements from among several you specify. The choice is based on a condition that exists when the method executes. The basic language elements for branching are **if**, **iif**, and **switch**.
- *Looping* executes a statement or set of statements repeatedly until some condition is met. The basic language elements for looping are **for**, **forEach**, **loop**, **scan**, and **while**.
- *Terminating* ends a loop. The basic language elements for terminating are **quitLoop** and **return**.

For more information and examples, refer to the online ObjectPAL Help.

Data types

The ObjectPAL data types are listed in Table 4.1. These data types, as well as more complex types, are described in Chapter 5.

Table 4.1 ObjectPAL data types

Type	Contains
Alpha	Up to 32,767 characters for a String variable, up to 255 characters for a quoted string
Money	Money values ranging from $\pm 3.4 * 10^{-4930}$ to $\pm 1.1 * 10^{4930}$, precise to 6 decimal places
Date	Dates ranging from January 1, 100 to December 31, 9999
DateTime	A time and a date combined in the form year-month-day-hour-minute-second-millisecond
Graphic	Bitmap images
Logical	True or false
LongInt	Integer values ranging from –2,147,483,648 to 2,147,483,647 (4 bytes)
Memo	Up to 512MB in Paradox tables
Number	Floating-point values ranging from $\pm 3.4 * 10^{-4930}$ to $\pm 1.1 * 10^{4930}$
SmallInt	Integer values ranging from –32,768 to 32,767 (2 bytes)[1]
Time	Clock data in the form hour-minute-second-millisecond

1. –32,768 cannot be stored n a Paradox table, because to Paradox, –32,768 = Blank.

Table 4.2 shows how to combine and convert values of different types.

Important Not all types can be combined; for example, you can't add a String to a Number. Also, Binary, Graphic, and OLE types cannot be combined at all, so they're not included in the table.

Table 4.2 Combining data types[1]

	M	A	E	T	D	$	N	L	S	&
M (Memo)	M	*[2]	*	*	*	*	*	*	*	*
A (String)	*	A	*	*	*	*	*	*	*	*
E (DateTime)	*	*	E	E	E	*	E	*	E	*
T (Time)	*	*	E	T	T	*	T	*	*	*
D (Date)	*	*	E	T	D	*	D	D	D	*
$ (Money)	*	*	*	*	*	C	C	C	C	*
N (Number)	*	*	E	T	D	C	N	N	N	*
L (LongInt)	*	*	*	*	D	C	N	L	L	*
S (SmallInt)	*	*	E	*	D	C	N	L	S	*
& (Logical)	*	*	*	*	*	*	*	*	*	&

1. Binary, Graphic, and OLE types cannot be combined, and are omitted from this table.
2. * = not allowed

Converting between data types

You can convert explicitly from one data type to another using the casting procedure for each data type, as shown in the following example. The rules for casting are less restrictive than the rules ObjectPAL uses for automatic conversion. You can convert a value to a less-precise data type, but the result might be a truncated version of the original value.

```
var
   myNum Number ; declares myNum as a Number
   theResult   String
endVar

message(1 + 2)              ; Displays 3
sleep(1500)

message(1.0 + 2)            ; Displays 3.00
sleep(1500)

myNum = 3.65                ; Assigns a value of 3 to myNum
myNum = SmallInt(3.65)      ; Changes Number 3.65 to SmallInt, returns 3.00
myNum = SmallInt("12.67")   ; Changes String "12.67" to SmallInt, returns 12.00

theResult = "abc" + String(myNum)    ; theResult = "abc12"
msgInfo("The result is", theResult)
```

Also, consider the following example. Because variables *a* and *b* are declared to be of type SmallInt, ObjectPAL computes the answer as a SmallInt:

```
var
   a, b SmallInt
endVar
a = 1
b = 2
message(a/b)  ; displays 0
              ; a/b = 1/2 = 0.5 which has an integer value of 0
```

To compute the answer as a decimal value, cast *a* and *b* as Numbers:

```
var
   a, b SmallInt
endVar
a = 1
b = 2
message(Number(a)/Number(b))  ; displays 0.50
```

The following statement displays "0.00" because it calls the casting procedure *after* computing the answer using SmallInt values.

```
message(Number(1/2)) ; displays 0.00
```

Another approach is to work with Number values from the beginning. For example, each of the following statements displays "0.50":

```
message(1/2.0) ; each of these statements displays 0.50
message(1.0/2)
message(1.0/2.0)
```

Expressions

An expression is a single value, or a set of one or more elements that evaluates to (or results in) a single value.

Here are some examples of expressions:

```
5                          ; the number 5
"hello"                    ; the quoted string hello
"hello" + " world"         ; adds two strings
myTable.myField.value + 5  ; adds 5 to the value of myField
myTable.cSum("Amount")     ; calculates the sum of values in the Amount field
```

Operators

You can use operators in expressions to combine and manipulate values and data elements. Table 4.3 lists the operators available in ObjectPAL. For more information about using operators with specific complex data types, refer to the "Array," "DynArray," "Point," and "Record" sections of Chapter 5.

Note The (=) symbol also functions as an assignment operator. For more information and examples, refer to the online ObjectPAL Help.

Table 4.3 ObjectPAL operators for data types

Data Type/Category	Operator	Function
Alpha (A)		
	+	Concatenation
Numeric (N,$,S)		
	+	Addition
	–	Subtraction or negation
	*	Multiplication
	/	Division
Date (D), Time (T), and DateTime (E)		
	+	Addition
	–	Subtraction or days apart
Comparison		
	=	Equal to
	<>	Not equal to
	<	Less than
	<=	Less than or equal to
	>	Greater than
	>=	Greater than or equal to
Logical		
	AND	Logical AND
	OR	Logical OR
	NOT	Logical NOT

In the examples in this section, assume these constants have been assigned:

```
const
   Age = 50.2
   NewYear = Date("1/1/1993")
   HighNoon = DateTime("12:00:00 am 12/12/12")
   Deathly = "Tax"
endConst
```

The + operator

The action of the (+) operator depends upon the type of expression it's operating on:

- *Number* + *Number* adds the two numbers.

```
message(45 + 50.2 + 5)        ; 100.20 (addition)
message(45 + Age + 5)         ; 100.20 (addition)
```

- *String* + *String* combines the strings (called *concatenation*); string concatenation is legal on alpha fields and memo fields.

```
message("Income" + " Tax")    ; "Income Tax" (concatenation)
message("Income" + Deathly)   ; "IncomeTax" (concatenation)
```

- *Date* + *Number* (or *Number* + *Date*) adds a number of days to the date (called *date addition*).

```
message(7 + Date("12/17/91"))   ; 12/24/91 (date addition)
message(NewYear + 1)            ; 1/2/93 (date addition)
```

- *Time* + *Number* (or *Number* + *Time*) adds a number of milliseconds to a Time value.

```
message(Time("1:00:00 am") + 10000) ; 1:00:10 AM
```

- *DateTime* + *Date* adds a Date value to a DateTime value. The result is a DateTime value.

```
message(HighNoon + Date("01/01/01")); add 1901 years to get 12:00:00 AM, 12/12/3813
```

- *DateTime* + *Time* adds a Time value to a DateTime value. The result is a DateTime value.

```
message(HighNoon + Time("01:01:01 PM")); add 13 hours, 1 minute, and 1 second to get
                                       ; 01:01:01 PM, 12/12/12
```

- *DateTime* + *Number* (or *Number* + *DateTime*) adds a number of milliseconds to a DateTime value.

```
message(HighNoon + 5000) ; 12:00:05 AM, 12/12/12
```

No other orderings of arguments in expressions combining different data types are permitted with the (+) operator.

The – operator

Like the (+) operator, the (–) operator has multiple uses:

- *Number* – *Number* subtracts the second number from the first (called *subtraction*).

```
message(76.5 - Age)     ; 26.3 (subtraction)
```

- –*Number* reverses the sign of the number (called *unary negation*).

```
message(-Age)           ; -50.2 (negation)
```

- *Date* – *Number* subtracts a number of days from a date (called *date subtraction*).

```
message(NewYear - 3)    ; 29-Dec-1992 (date subtraction)
```

- *Date* – *Date* returns a Date value representing the number of days between two dates (called *days apart*). To represent days apart as a number, cast the result as a Number value, as shown in the following example:

```
var
   d1, d2 Date
   daysApart Number
endVar

d1 = Date("12/21/95")
d2 = Date("12/11/95")
daysApart = Number(d1 - d2)
message(daysApart) ; displays 10
```

```
message(Number(Today() - NewYear))     ; displays number of days since
                                       ; Jan. 1, 1993
```

No other combinations of argument types are permitted with the (–) operator. In particular, you can't use (–) with string arguments. (However, you can use the String method **substr** to extract substrings.)

The * and / operators

You can use the multiplication (*) and division (/) operators only with numeric arguments. Division by 0 results in a run-time error.

Comparison operators

You can use the comparison operators in Table 4.4 to compare values of any of the data types, including logical. The result is always a logical value, True or False. The action of operators like (<) and (>) depends on the types of arguments being evaluated.

Table 4.4 Comparison operators

Data type	Basis	Ordering
Numeric	Numeric order	Lower < Higher
Alpha	Alphanumeric order	Lower < Higher
Date	Chronological order	Earlier < Later
Logical		False < True

Alpha comparisons depend on sort order. In the ASCII sort sequence, A < Z < a < z, but this is not true in the other sequences, which support a true dictionary sorting order (that is, characters are sorted in alphabetic sequence irrespective of case).

You can compare alpha (A) fields with memo (M) fields. If you compare values of different types, (<>) returns a value of True, while all other operators return False, as in this example:

```
message(5 + 1 < 6 * 2)             ; True (see also "Order of evaluation")
message(Age > 21)                  ; True
message(Date(7/4/1776) = NewYear)  ; False, since the dates are not the same
message("Abc" = "ABC")             ; False, since the strings are not the same
message("Abc" > "ABC")             ; True in ASCII order, False in other orders
message(Age > Age + 5)             ; False
```

Note The (=) sign is used *both* as a comparison operator within expressions and as part of the assignment statement. When only a variable name appears to the left of the (=) sign, it gets the value of the expression to the right; otherwise, the two expressions are compared.

```
a = 3       ; assigns the value of 3 to a
x = 4 = a   ; the first = is an assignment, the second a comparison
```

The previous statement assigns to the variable *x* the value of the expression 4 = a (False). Enclose expressions in parentheses when there's any chance of confusion:

```
x = (4 = a)
```

Logical operators (AND, OR, NOT)

You can combine expressions with logical values using the logical operators in Table 4.3:

- *x* AND *y* returns True if both arguments (*x* and *y*) are true.
- *x* OR *y* returns True if either argument (or both) is true.
- NOT *x* returns True if the argument is false.

In ObjectPAL, expressions on both sides of an AND or an OR statement are evaluated, unlike programming languages that use short-circuit evaluation.

Here are some examples:

```
message(3 = 4 AND 7 = 8)        ; False, since neither argument is true
message(3 = 4 OR 7 = 8)         ; False, since both arguments are false
message(3 = 3 OR 7 = 8)         ; True, one argument is true
message(NOT (3 = 4 AND 7 = 8))  ; True, since the argument is false
message(NOT False)              ; True
message(3 <> 4)                 ; True
```

Note There is a difference between the comparison operator (<>) (not equal to) and the logical operator NOT. The (<>) compares any two data values, but the NOT operator negates a logical value.

ObjectPAL also provides methods for performing bitwise operations. The LongInt and SmallInt types include the methods **bitAND**, **bitOR**, and **bitXOR**. Refer to the online ObjectPAL Help for more information.

Note To ObjectPAL, the following statements are equivalent:

```
if x = True then y endIf
if x then y endIf
```

By the same token, these two statements are equivalent:

```
if x = False then y endIf
if not x then y endIf
```

Order of evaluation

In expressions containing more than one operator, the operations are evaluated in the order of precedence shown in Table 4.5.

Important ObjectPAL does *not* use short-circuit Boolean evaluation; it evaluates both sides of a logical (Boolean) expression before execution continues.

Any expression contained in parentheses is evaluated first, and inner levels of parentheses are evaluated before outer levels. When two or more operators of equal precedence are in a single expression, they are evaluated from left to right, as shown in the following code:

```
3 + 4 * 5            ; returns 23 (* has precedence over +)
(3 + 4) * 5          ; returns 35 (parentheses first)
((3 + 4) * 5) / 2    ; returns 17.5 (inner parentheses first)
```

Table 4.5 Operator precedence

Precedence	Operators
1	()
2	* /
3	+–
4	= <> < <= > >=
5	NOT
6	AND
7	OR

Table 4.6 lists other symbols used in ObjectPAL expressions.

Table 4.6 Symbols used in ObjectPAL

Character	Function
~	Tilde (Query variable)
_	Leading underbar (Query example element)
;	Semicolon (comment)
{	Left brace (begin comment block)
}	Right brace (end comment block)

Table 4.7 lists pattern-matching symbols for working with character strings.

Table 4.7 Pattern-matching symbols used with character strings

Symbol	Function
..	Match any characters
@	Match any single character

Naming rules

This section discusses the rules for naming objects.

Naming objects

Examples in the previous section used objects named explicitly when the form was designed. For instance, a box was created and named *boxOne*. You might have noticed, though, that Paradox gives an object a default name when you create it (default names always begin with a pound sign; for example, *#box5*). If you create an object and don't change its default name, that object's name is not required in the containership path.

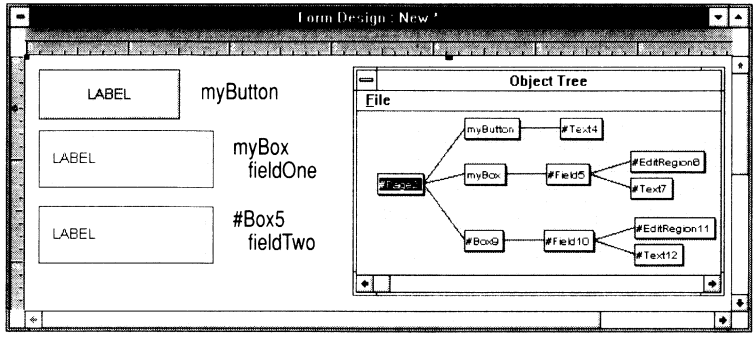
Note The object is part of the containership hierarchy for custom code and variables, but the object's name is optional if it begins with a pound sign.

See the *User's Guide* for naming rules for Paradox objects.

Default names

Figure 4.3 shows a page containing one button (named *myButton*) and two boxes. One box was named *myBox*, and the other box was left with its default name, *#Box5*. A field object was placed in each box (named *fieldOne* and *fieldTwo*, respectively).

Figure 4.3 Default names and the containership hierarchy



Methods attached to *myButton* must specify the path through *myBox* to address *fieldOne*.

Methods attached to *myButton* can bypass *#Box5* and address *fieldTwo* directly, because *#Box5* is a default name.

Statements attached to *myButton* must specify the path through *myBox* to get to *fieldOne* if *fieldOne* is not a unique name, for example,

```
myBox.fieldOne.value = 123
```

They can bypass *#Box5* and address *fieldTwo* directly, like this:

```
fieldTwo.value = 755
```

Because *fieldTwo* is a unique object name on this form, the container name is not required in the containership path. This feature is convenient when you want to place objects in a form for purely cosmetic reasons, without worrying about naming them and addressing them to get to the objects that do the real work.

Note You can address an object by its default name, as in

```
#box5.color = Red
```

Objects bound to tables

If you place an object that gets its default name from a table, that name is part of the containership path. Only default names that begin with a pound sign can be omitted from the path.

For example, suppose you place a table frame in a form and define it to display data from CUSTOMER.DB. In Paradox terms, the table frame is "bound" to the *Customer* table. The table frame gets the name *CUSTOMER* by default, and the field objects in the table frame get the names of the fields in the *Customer* table (without pound signs).

Important Object names cannot begin with a number, even if the object is bound to a table. For example, a table can have a field named *3Dimension*, but an object bound to that field cannot. Paradox prevents you from giving an object an illegal name in a design window.

If an ObjectPAL statement assigns an illegal name to an object, Paradox replaces invalid characters with underbars.

If a statement in another object (say, a button) wants to set the value of one of those fields, it must specify the path through the table frame to the field object. For example, the following statement sets the value of the LastName field to "Meisner".

```
CUSTOMER.LastName.value = "Meisner"
```

Special characters in object names

If an object gets its name from a field in a table, and if that name contains any character *except* a letter a number, or a pound sign (#), Paradox replaces that character with an underscore. The following table lists some examples.

Table 4.8 Special characters in object names

Name of field in table	Name of field object in a form
Order No	Order_No
Size:	Size_
Field: 1	Field__1
Total cost($)	Total_cost___

Note Paradox does not change names in the underlying table.

Methods, procedures, variables, and arrays

Here are the rules for naming custom methods and the ObjectPAL variables, arrays, and procedures you use in methods:

- Names can be up to 32 characters.
- The first character must be a letter A–Z or a–z.
- Subsequent characters can be letters, digits, or one of these three characters: $! _. Extended characters (ANSI 161–255) are allowed.
- No spaces or tabs are allowed.
- No distinction is recognized between uppercase and lowercase letters; ObjectPAL is case-insensitive.
- Names should not duplicate keywords, basic language elements, built-in object variables, or names of object types, events, methods, or properties.

Table 4.9 shows examples of valid and invalid variable names.

Table 4.9 Some valid and invalid variable names

Name	Valid?	Explanation
anyThing	Yes	No distinction between uppercase and lowercase
a.name	No	Has "." embedded
!claim	No	Doesn't begin with a letter
claim!	Yes	Begins with a letter

Table 4.9 Some valid and invalid variable names (continued)

Name	Valid?	Explanation
voilà	Yes	Extended characters are allowed
L0123	Yes	All caps OK and begins with a letter
5A6	No	Doesn't begin with a letter
abc xyz	No	Contains a space
abc_xyz	Yes	Contains an underbar for a space
record	No	Keyword
myRecord	Yes	Not a keyword

These rules govern only the names (also called *identifiers*) of ObjectPAL variables, arrays, custom methods, and custom procedures.

In theory, you should not use object type names, names of basic language elements, keywords, property names, or method names to name objects or variables. In practice, this is not always possible. For example, an application for a paint store might have a table with a field named Color. To avoid confusion with the Color property, use one single quote instead of a dot before the property name. This example sets the Color property of a field named price:

```
price.Color = Red ; uses a dot
```

The next example sets the Color property of the field object named *Color.* (The field name comes before the quote, and the property name comes after.)

```
Color'Color = Red ; uses a single quote
```

You can also use two single quotes (*not* one double quote) instead of the keyword *self* before a property name. For example, these two statements are equivalent:

```
''Color = Red
self.color = Red
```

You can give an object a name beginning with "#," but because the default names Paradox gives design objects always begin with "#" (for example, *#button1320*), you'll want to be able to distinguish the objects you've named from those that you haven't.

ObjectPAL terms

The terms *method, procedure,* and *basic language element* have distinct meanings in ObjectPAL, as explained in the following sections. Table 4.12 at the end of the chapter summarizes the differences.

Methods

A method is code that defines the behavior of an object. Methods define how an object responds to events—events generated by users, by Paradox, and even by other methods. ObjectPAL methods fall into one of three categories:

- Built-in methods included with every Paradox object

- Methods in the ObjectPAL run-time library (RTL)
- Custom methods you create

The characteristics of each method category are listed in Table 4.10.

Table 4.10 Characteristics of methods

Built-in	RTL	Custom
Built into objects in a form	Used for writing custom code	A user-defined routine
Specify default behavior of objects	Operate on objects of a specific type	Attached to an object
Execute automatically in response to events	Executes within an ObjectPAL statement	Executes when called by an ObjectPAL statement
Can be modified by attaching custom code	Can be used in code attached to any object	Public: can be called by other objects using dot notation
	Dot notation specifies the object to operate on	

ObjectPAL methods give the language symmetry and consistency: within a type, methods often come in pairs. For example, if a type has an **open** method, you can expect it to have a **close** method, too. If you can read information from an object, you can write to it; if you can get a value, you can set it. ObjectPAL methods are consistent across types because methods with similar names do similar things. For example, **open** makes an object available for manipulation, whether the object is a table or a text file, and **close** puts it away. The underlying code might differ, but conceptually, the results are the same.

Built-in methods

Every Paradox object comes with built-in methods (for example, **open**, **close**, and **mouseUp**) for each event it can respond to. These built-in methods specify an object's default behavior in response to a given event. You build Paradox applications by attaching ObjectPAL code to the built-in methods of objects placed in forms.

You attach your own code to built-in methods using the ObjectPAL Editor. To edit a built-in method for an object,

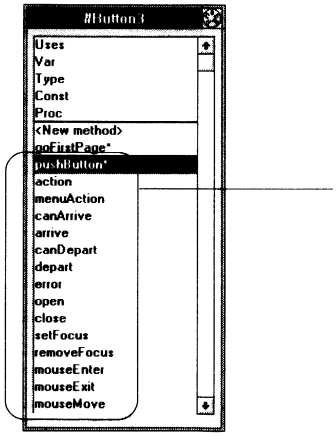
1 Inspect the object and select Methods from the its menu, or press *Ctrl+Spacebar*.

The Method Inspector (shown in Figure 4.4) lists an object's built-in and custom methods and procedures. You can edit these. You can also create and delete methods and procedures, and declare external routines (Uses), variables (Var), data types (Type), constants (Const), and procedures (Proc).

Note Each built-in method is described in Chapter 12.

2 Double-click a method or procedure to edit it. You can type the text for a method directly in the ObjectPAL Editor, or use the Clipboard to copy, cut, and paste methods and parts of methods from other objects or from a file. See Chapter 8 for more information about using the Method Inspector and Editor.

Figure 4.4 Selecting built-in methods to edit



Methods in the run-time library

The ObjectPAL run-time library (RTL) is a collection of predefined routines. It includes methods you can use to perform a wide range of tasks, from reading and editing data in tables to creating and displaying menus. Each of these methods is associated with an object type: methods for working on forms are in the Form type, methods for working with text files are in the TextStream type, and so on. These methods are presented in the online ObjectPAL Help.

Methods in the run-time library require you to use dot notation to specify an object to operate on. For example,

```
var
   orders, salesF Form
   salesTV TableView
endVar
orders.open("orders")
salesF.open("sales")
salesF.setTitle("Sales info") ; set the title of the sales form
orders.maximize()             ; maximize orders form, do nothing to sales
salesTV.open("sales.db")      ; open a table window of the sales form
```

In this example, dot notation separates the objects (*salesF*, *salesTV*, and *orders*) from the methods (**open, setTitle**, and **maximize**).

Custom methods

Custom methods are auxiliary routines you create. They're convenient for making frequently used routines available to several objects. Custom methods are public; that is, custom methods attached to an object can be called by any other object. (In contrast, custom procedures are private.) Suppose a form contains two boxes: *box1* and *box2*. If *box1* has a custom method named **doSomething**, code attached to *box2* could use the following statement to call it:

```
box1.doSomething()
```

This statement says, "Execute the method named **doSomething** attached to the object named *box1*."

A custom method can take one or more arguments, and it can return a value (but it doesn't have to). To create a custom method, inspect an object, and choose Methods from its menu, or press Ctrl+Spacebar. The Method Inspector appears. Choose <new method> to open an ObjectPAL Editor window. You can type or paste text into custom methods just as you can for built-in methods.

After you save a custom method, its name is listed in the Method Inspector. To make changes, choose the name to open an ObjectPAL Editor window, just as you would to edit a built-in method.

To share custom code among two or more forms, you must put the code in a library. See Chapter 28 for more information.

Procedures

There are two kinds of procedures in ObjectPAL:

- Procedures in the ObjectPAL run-time library (RTL)
- Custom procedures you create

The characteristics of each kind of procedure are listed in Table 4.11.

Table 4.11 Characteristics of ObjectPAL procedures

RTL	Custom
Used for writing custom code	A user-defined routine
Operates on objects of a specific type	Defined in the object's Proc window
Can be used in code attached to any object	
Does not specify the object to operate on; the object is implied	Private: cannot be called by other objects using dot notation
Executes within an ObjectPAL statement	Executes when called by an ObjectPAL statement

RTL procedures

The procedures in the ObjectPAL run-time library (RTL) are just like methods, with one exception: procedures never explicitly specify an object. Code attached to any object can call any RTL procedure, and the procedure will know what to do. Like RTL methods, RTL procedures are associated with object types, and they execute in response to events. It may be helpful to think of RTL procedures as RTL methods with the object implied.

For example, the statement `close()` calls the **close** procedure for the Form type, which shuts down the current form. Dot notation isn't needed to specify the current form, because it's implied by the procedure.

Other procedures set system-wide flags. For example, the Session type procedure **blankAsZero** specifies how to handle blank values.

The System type includes procedures for displaying dialog boxes, and you use them without specifying an object. For example,

```
msgStop("Alert!", "This file already exists.")
```

The System type also includes the procedures **beep** and **sleep**, and several procedures for getting and setting the mouse position and shape. The **message** procedure is also defined for the System type, along with several procedures that write information about ObjectPAL out to Paradox tables. The following example uses the **beep**, **sleep**, **message**, and **enumSource** procedures.

```
method pushButton(var eventInfo Event)
   beep()                              ; plays the system beep sound
   sleep(2000)                         ; waits for 2 seconds
   beep()
   message("Did you hear two beeps?") ; displays a message in the status line
   sleep(2000)
   enumSource("mySource.db")           ; creates a table of all code in this form
endMethod
```

Custom procedures

Custom procedures in ObjectPAL resemble procedures in many other programming languages. A custom procedure is a routine you write yourself and use like a subroutine. Custom procedures are private—their availability to other objects is determined by containership (discussed in the Chapter 6).

Important Paradox can call a custom procedure faster than it can call a custom method. The code executes at the same speed, but because of the way a custom procedure is stored, Paradox can "find" it faster than it can "find" a custom method.

You can declare procedures in two places:

- Within a method
- In an object's Proc window

The syntax for declaring procedures is presented in the topic "proc" in the online ObjectPAL Help.

Procedures declared in methods

A procedure declared in a method is private: it can be called only by the method in which it is defined.

Here's an example of a custom procedure:

```
proc inc (x SmallInt) SmallInt
   return x+1 ; increments a number
endProc
```

The following example shows how to declare and call that procedure (and another one) from within a method. (In this example, it's the **pushButton** method, but it could be any method). The procedures are declared first, before the body of the method.

```
proc inc (x SmallInt) SmallInt
   return x+1
endProc
proc showMe (x SmallInt)
   msgInfo("myNum = ", x)
```

```
    endProc
  method pushButton (var eventInfo Event)
    var
      myNum SmallInt
    endVar
    myNum = 3
    showMe(myNum)
    myNum = inc(myNum)
    showMe(myNum)
  endMethod
```

Procedures declared in an object's Proc window

A procedure declared in an object's Proc window has the same syntax as a procedure declared in a method, but it has a different scope. A procedure declared in an object's Proc window is visible to all methods attached to that object, and to all methods in objects *that* object contains. However, unlike a custom method, you can't call a procedure from outside the procedure's containership hierarchy.

To make a custom procedure available to every object in a form, declare it in the form's Proc window.

Basic language elements

Basic language elements include control structures like if...then...else...endIf and switch...endSwitch, commands like quitLoop, and keywords for creating structures like var...endVar. They do not use dot notation. For example, the syntax for the for...endFor structure is

for *VarName* **[from** *startVal***]** **[to** *endVal***]** **[step** *stepVal***]**
  Statements
endFor

Basic language elements are presented in the online ObjectPAL Help.

Summary of differences

Table 4.12 summarizes the differences between RTL and custom methods, RTL and custom procedures, and basic language elements. It also shows prototypes (items in square brackets are optional) and indicates whether a construct is public. In the prototypes, *obj* represents an object name or containership path, and *argList* represents a list of one or more arguments and return types.

Table 4.12 Summary of differences

Construct	Prototype	Public?
RTL method	obj.method([*argList*]) [*returnType*]	Yes
Custom method	[obj].method([*argList*]) [*returnType*]	Yes
RTL procedure	procName([*argList*]) [*returnType*]	Yes
Custom procedure	procName([*argList*]) [*returnType*]	No
Basic language element	keyword (return), or structure (for...endFor)	N/A

Data types

When creating tables, you use different field types to represent different types of data and the operations you want to perform. This is especially important when designing ObjectPAL applications, in which you need different types of operations and data types to accomplish specific tasks. ObjectPAL supports the data formats available to your tables and offers special data types tailored for application development, as shown in Table 5.1.

Table 5.1 Data types

Type	Description
AnyType	A catch-all for basic data types
Array	An indexed collection of data
Binary	Machine-readable data
Date	Calendar data
DateTime	Calendar and clock data combined
DynArray	A dynamic array
Graphic	A bitmap image
Logical	True or False
LongInt	Used to represent relatively large integer values
Memo	Holds lots of text
Money	Used to manipulate currency values
Number	Floating-point values
OLE	A link to another application
Point	Information about a location on the screen
Record	A user-defined structure
SmallInt	Used to represent relatively small integer values
String	Letters
Time	Clock data

This chapter describes each data type, explains how it works, and shows examples of how to use it. (The methods and procedures for each type are listed in the online ObjectPAL Help.) Understanding the nature and use of these data types is a key to learning ObjectPAL.

AnyType: The catch-all data type

AnyType is a catch-all data type. It lets you design methods for a variety of data types when you can't predict the data type of the actual target value until the method executes. An AnyType value can be any one of the following data types:

Binary	Logical	Point
Money	LongInt	SmallInt
Date	Memo	String
DateTime	Number	Time
Graphic	OLE	

An AnyType value can never be a more complex type (such as TCursor or TextStream). It inherits characteristics from the value assigned to it. That is, it behaves like a String when assigned a String value, like a Number when assigned a Number value, and so on.

Using undeclared AnyType variables

AnyType data objects are included in ObjectPAL so you can use variables for basic data types without declaring them first. (Remember, though, that it's better to declare variables whenever possible.) As an example of the convenience of using AnyType variables, suppose you want to create a simple for...endFor structure. You can write the following code without having to explicitly declare a data type for *x*:

```
for x from 1 to 9
   message(x)
endFor
```

When ObjectPAL encounters an undeclared variable, it assumes the variable is an AnyType, and handles conversion internally.

Using declared AnyType variables

You can also explicitly declare a variable as an AnyType. This technique is useful when working with values whose data type is unknown or subject to change. For instance, every design object has properties (described in Chapter 13), and different properties return values of different data types: for example, the Name property returns a String, and the Size property returns a Point.

Why declare variables?

Code executes faster when you declare variables. To demonstrate this, attach the following code to a button's built-in **pushButton** method:

```
method pushButton(var eventInfo Event)
   beep()
   for i from 1 to 1500 endFor
   beep()
endMethod
```

This code uses the undeclared variable *i* as an index in the for...endFor loop. When you run the form and click the button, the computer will beep, remain silent for a few seconds (the duration depends on the computer's speed), and then beep again.

Next, execute the same code, but this time with the variable *i* declared. Edit the button's **pushButton** method, and add one line of code:

```
method pushButton(var eventInfo Event)
var i SmallInt endVar  ; add this line of code to declare the variable i
   beep()
   for i from 1 to 1500 endFor
   beep()
endMethod
```

Now when you click the button, the interval between the beeps is much shorter, because you've declared the variable *i*. Paradox doesn't have to test it each time through the loop, so the code executes faster.

Using AnyType variables with property values

The following example shows how AnyType variables can be used to work with property values. It creates an array named *propAr* where each item in the array is a property. Then it uses the UIObject type method **getProperty** to inspect the field object named *theField* and assign the value of each property in the array to the variable *propVal*. Because *propVal* is an AnyType, it can store the property values, even though each value is of a different data type.

```
var
   propAr  Array[3] String
   propVal AnyType
endVar

; store property names in the array
propAr[1] = "name"        ; The Name property returns a String
propAr[2] = "size"        ; The Size property returns a Point
propAr[3] = "CursorPos"   : The CursorPos property returns a LongInt
theField.action(EditEnterFieldView) ; must be in Field View to get CursorPos
for i from 1 to 3
   propVal=theField.getProperty(propAr[i]) ; inspect theField, get values
   propVal.view()                          ; display the value in a dialog box
endFor
```

Using operators with AnyType values

When using binary operators (operators like + and –, which only work on two values at a time) on AnyType values, ObjectPAL tries to convert the values to a common data type. For example, when adding a SmallInt to a Number, the SmallInt is converted to a

Number before the addition is carried out. Values are always converted to the more precise type: given a SmallInt and a Number, the SmallInt is converted because Number is the more precise type. A value is never converted to a type that cannot hold all the information of the original type; for example, a Number will never be converted to a SmallInt.

Not all types can be combined: you can't add a String to a Number, for example. The rules ObjectPAL uses for automatically combining and converting data types are presented in Chapter 4.

Using the Value property with UIObjects

UIObjects (for example, field objects) have a property called *Value* that evaluates to an AnyType. For example, the following statements assign to *x* the value of the field object *someField*, no matter what type of data *someField* contains:

```
var x AnyType endVar

x = someField.value
```

ObjectPAL lets you omit the *value* keyword in expressions, so the following two statements are equivalent:

```
x = someField.value

x = someField
```

However, you can use this shortcut only when working with an AnyType value or a value that can be converted to an AnyType. In the previous examples, the variable *x* is declared to be of type AnyType. In contrast, consider the following custom procedure, named *getObjectSpecs*. It takes one argument, a UIObject variable named *theObject*.

```
proc getObjectSpecs(theObject UIObject)
   msgInfo("Name", theObject.name)
   msgInfo("Size", theObject.size)
   msgInfo("Position", theObject.position)
endProc
```

When a statement calls *getObjectSpecs*, it passes the UIObject *someField*, not the value of the field. For example,

```
getObjectSpecs(someField)
```

Therefore, although you can in some cases omit the *Value* keyword, your code will be more readable if you use it.

Advanced topic: Working with values in field objects

This section is for advanced programmers. When you edit a field object or switch to Field View, Paradox creates a temporary text object on top of the field object. Keystrokes, mouse clicks, and values assigned by ObjectPAL statements all appear in the temporary text object first. Then, when you move off the field object, Paradox copies the contents of the text object to the field object and deletes the text object. In the vast majority of situations, this operation is transparent, both to the user and the ObjectPAL

programmer. However, note that although the temporary text object exists, the data type of the field object's value is Memo. At all other times, the data type matches the native type of the data. For example, if a field object contains a SmallInt value, its native data type is SmallInt.

If a field object is bound to a table in the form's data model, the data type of the field object's value is the same as the data type of the field in the underlying table. If a field object is unbound, the data type of its value can be one of the following:

- When the field object is empty, the data type value is String.
- When in Field View or when you use the keyboard to enter a value, the data type value is Memo.
- When you use an ObjectPAL statement to assign a value, the data type value is the native type of that value. For example, if you use ObjectPAL to assign a String value to a field object, its data type is String.
- When you paste a value from the Clipboard, the data type value is the native type of that value.

In most cases, the data type of a field object's value doesn't matter, because ObjectPAL automatically converts values between the basic data types. In certain situations, though, you need to guide ObjectPAL to get the desired result. For example, you would expect the following statement to display 3.1, and you would be right:

```
message(2.1 + 1) ; displays 3.1
```

However, suppose a form contains an unbound field object named *theField*. If you type the value 2.1 into *theField*, the following statement displays 3, because the data type of *theField* is Memo (when you use the keyboard to enter data into an unbound field, the data type is always Memo), and the data type of 1 is SmallInt:

```
message(theField.value + 1) ; displays 3
```

To perform the addition, ObjectPAL converts the Memo value to a SmallInt and, in the process, truncates values to the right of the decimal point.

There are two ways to avoid this loss of data: cast the field object's value to the desired data type, or specify the desired precision in the known value. For example, both of the following statements display 3.10:

```
message(Number(theField.value) + 1)  ; cast the field object's value as a Number

message(theField.value + 1.00)       ; specify precision in the known value
                                     ; (use 1.00 instead of 1)
```

Either statement gives ObjectPAL the information it needs to display the expected result.

Array: Pigeon holes for data

An Array holds values (called *items* or *elements*) in *cells* similar to the way mail slots hold mail. An ObjectPAL array is one-dimensional, like a single row of slots where each slot holds one item.

Note In ObjectPAL, array items are counted beginning with 1, not with 0, as in some other languages.

To use arrays in methods, you must declare them by specifying a name, size (number of items), and data type for the items.

- Names. Arrays are named like other variables, according to the conventions listed in Chapter 4.
- Size. The maximum number of items an array can hold depends on the data type and system memory. Arrays can be *static* (fixed size) or *resizeable* (variable size).

Note ObjectPAL also supports dynamic arrays. See the description of the DynArray type in this chapter.

- Data types. An array can store data of any type except Array, DynArray, and Record.

Declaring an array

The syntax for declaring an Array is

```
var
  arrayName Array[arraySize] dataType
endVar
```

A static array is specified by placing the length of the array in square brackets (for example, [100] specifies 100 items).

A resizeable array is specified by empty square brackets: []. Before you can assign an item to a resizeable array, use **setSize** or **grow** to specify an initial size.

The following declarations would go in a Var window or in a method's variable declaration section:

```
var
   stringArray Array[300] String   ;  300 String items, static array
   anyTypeArray Array[300] AnyType ;  300 AnyType items, static array
endVar
```

You can also declare an array in a method's type declaration section or in a design object's Type window. Declarations of user-defined types are placed in a design object's Type window or before the variable declaration section in a method, like this:

```
type
   myArrayType = Array[100] SmallInt
;  Declares an Array type called myArrayType.
;  MyArrayType is static (100 SmallInt items).
;  Now, you can declare other arrays of type myArrayType (see below).

   myOtherArrayType = Array[] LongInt
   ; Declares another Array type: a resizeable array of LongInt items.
endType
var
   myArray1  myArrayType
;  100 SmallInt items, like myArrayType
```

```
   myArray2  myOtherArrayType
;  resizeable array of LongInt items, like myOtherArrayType
endVar
```

To use an array as an argument for a custom method or procedure, you must declare it in the Type declaration box. For more information, see "Passing arrays as arguments" later in this chapter.

Assigning items

Assigning array items is like putting mail into mail slots. In an ObjectPAL array, the slots are called cells, the content of a cell is called an item, and each cell has an index number. The first cell is number 1, not 0. The syntax for assigning items to cells is

arrayName [*cellNumber*] = *expression*

Here are some examples:

```
var
   myStringArray Array[4] String ; array of 4 Strings
   myNumberArray Array[]  Number ; resizeable array of Numbers
   n SmallInt
endVar

myStringArray[1] = "hello" ; assigns String hello to cell 1
myStringArray[4] = "good-bye"
; Puts String good-bye into cell 4. Cells 2 and 3 are empty.

myNumberArray.setSize(8) ; makes cells for 8 items

myNumberArray[1] = 231   ; Assigns Number 231 to item 1

for n from 2 to 8  ; Assigns Numbers 2 through 8 to cells 2 through 8
   myNumberArray[n] = n
endFor
```

Items you assign must be of a type acceptable to the array. For example, assigning a numeric value to an array of strings will cause an error. To use mixed data types in an array, declare the array to be of type AnyType. For example,

```
var
   myArray Array[] AnyType   ; declares a resizeable array of AnyType items
endVar
myArray.setSize(2)           ; makes cells for 2 items
myArray[1] = "cat"           ; stores a String value
myArray[2] = 5               ; stores a SmallInt value
myArray.grow(1)              ; adds another cell
myArray[3] = Date("11/9/59") ; stores a Date value
myArray.view()               ; displays the array in a dialog box
```

Adding data to a resizeable array

The methods listed in Table 5.2 add data to a resizeable array. Each method will automatically expand the array to make room for the item or items being added.

Table 5.2 Methods for adding data to a resizeable array

Method	Description
append	Adds the contents of one array to the end of another array
insert	Inserts one or more empty cells into an array
addLast	Adds one item after the last item of an array
insertFirst	Inserts one item at the beginning of an array
insertAfter	Inserts an item into an array after a specified item
insertBefore	Inserts an item before a specified item
copyToArray	Copies a record from a table to an array (or to a DynArray)

Accessing array items

Accessing array items is the reverse of assigning them. The syntax is

expression = *arrayName*[*cellNumber*]

For example,

```
var
   myArray Array[5] SmallInt ; first, declare the array

   n, x SmallInt
endVar

for n from 1 to 5
   myArray[n] = 2 * n ; assign values to the items
endFor

; access the first and third items
x = myArray[1] ; returns 2
x = myArray[3] ; returns 6
```

A useful method is **contains**, which reports whether an array contains a specified item. For example,

```
if theArray.contains("foo") then
   x = theArray.indexOf("foo")
else
   msgInfo("theArray", "Item foo not found.")
endIf
```

Removing data from an array

The following three methods remove data from a resizeable array:

- **remove** removes one or more items from an array, by index number.
- **removeItem** deletes the first occurrence of a value.

- **removeAllItems** removes all occurrences of a value.

Array operators

Using array operators (+, –, *, /, >, <, >=, <=, <>, and =), you can perform arithmetic operations on array values. Following are some examples:

```
var x    SmallInt
   ar1 Array[10] AnyType
   ar2 Array[10] AnyType
   ar3 Array[2] AnyType
   ar4 Array[2] AnyType
   ar5 Array[2] AnyType
endVar
for x from 1 to 10
   ar1[x] = x
endfor
ar2 = ar1 ; copies ar1 to ar2
; Arrays must be the same size. If items are different types,
; ObjectPAL tries to convert one to the other.

ar3[1] = 21
ar3[2] = 15

ar4[1] = 18
ar4[2] = 75

ar5 = ar3 + ar4
ar5.view() ; ar5 = 39, 90
; ar5[1] = ar3[1] + ar4[1] and ar5[2] = ar3[2] + ar4[2]

; expressions can be complex:
ar3 = (ar1 * ar2) + (ar2 / ar1) - ar1
```

You can also use array operators to compare arrays. Two arrays are equal if they have the same number of items and the items at each index match exactly. One array is greater than (or less than) another if they have the same number of items and every item in the first array is greater than (or less than) the item at the corresponding index of the second array. For example,

```
var ar1, ar2 Array[2] String endVar
ar1[1] = "a"
ar1[2] = "b"

ar2[1] = "a"
ar2[2] = "b"
message(ar2 = ar1) ; Displays True.
                   ; All items in ar2 = corresponding items in ar1
sleep(1500)

ar2[1] = "x"
ar2[2] = "z"
message(ar2 > ar1) ; Displays True.
                   ; All items in ar2 > corresponding items in ar1
```

```
ar2[1] = "a"
message(ar2 > ar1) ; Displays False. ar2[1] = ar1[1]
sleep(1500)

message(ar2 >= ar1) ; Displays True.
                    ; All items in ar1 >= corresponding items in ar2
```

The <> operator compares two arrays of the same size. It returns False only if all corresponding items in both arrays match; otherwise, it returns True. For example,

```
var
   ar1 Array[2] String
   ar2 Array[2] String
   ar3 Array[3] String
   ar4 Array[2] String
endVar

ar1[1] = "aaa"
ar1[2] = "bbb"

ar2[1] = "aaa"
ar2[2] = "bbb"

ar3[1] = "aaa"
ar3[2] = "bbb"

ar4[1] = "aaa"
ar4[2] = "bb"

message(ar1<>ar2) ; displays False
sleep(1000)

message(ar1<>ar3) ; Causes a run-time error, because
sleep(1000)       ; arrays are different sizes

message(ar1<>ar4) ; displays True
sleep(1000)
```

The (=) operator can be used both to compare two arrays and to copy one array to another. For example, the following statement compares two arrays:

```
if ar1 = ar2 then beep() endIf
```

The following statements copy the array *ar1* to the array *ar2*:

```
var
   ar1 Array[2] String
   ar2 Array[2] String ; You could also use a resizeable array,
                       ;   i.e., ar2 Array[] String
endVar

ar1[1] = "aaa"
ar1[2] = "bbb"

ar2 = ar1  ; copy ar1 into ar2
ar2.view() ; display ar2 in a dialog box.
```

Passing arrays as arguments

You can pass an array as an argument to a custom method or a custom procedure, but you must first declare a custom data type, as shown in the following example.

All the code in this example is attached to a button's built-in **pushButton** method. The first block declares a custom data type *PassAr* (the name is insignificant). This custom data type represents the array you want to pass. In this example, it's an array of three strings. The next block declares a very simple custom procedure that takes an argument of type *PassAr*; in other words, it takes an array of three strings. In the body of the method, a var block declares a variable of type *PassAr*. This variable will be passed to the custom procedure **showAr**. The rest of the code in this method initializes the array, and then passes it to **showAr**, which displays it in a **view** dialog box.

```
type ; first, declare a custom data type to represent the array
   PassAr = Array[3] String
endType

proc showAr(ar PassAr) ; this custom proc displays the passed array
   ar.view("Now showing ...")
endProc

method pushButton(var eventInfo Event)
   var
      ar PassAr        ; declare a variable of type PassAr
   endVar

   ar[1] = "Paradox"   ; initialize the array
   ar[2] = "for"
   ar[3] = "Windows"

   showAr(ar)          ; pass the array to the custom proc
endMethod
```

Binary: Machine-readable data

A binary object (sometimes called a binary large object or BLOB) contains data that a computer can read and interpret. An example of a binary object is a sound file: a human can't read or interpret the file in its raw form, but a computer can.

Table 5.3 lists the methods for working with Binary variables and binary objects.

Table 5.3 Binary type methods

Method	Description
readFromFile	Reads data from a file and stores it in a Binary variable
size	Reports the number of bytes in a binary object
writeToFile	Writes the contents of a Binary variable to a disk file

When you declare a Binary variable, you simultaneously create a handle to a binary object. A handle is a variable you can refer to in your code to shuttle binary data between a disk file and a table or from a disk file or a table to a custom routine.

For example, the following statements declare a Binary variable *theSound*, read binary data from a file into the variable, and then assign the value of the variable to a Binary field in a table. (Assume SOUNDS.DB is a Paradox table with the following structure: SoundName, A32; SoundData, B.)

```
var
   soundsTC TCursor
   theSound Binary
endVar
if theSound.readFromFile("noise.bin") then
   if soundsTC.open("sounds.db") then
      soundsTC.edit()
      soundsTC.insertRecord()
      soundsTC.SoundName = "Noise"
      soundsTC.SoundData = theSound
      soundsTC.endEdit()
      soundsTC.close()
   endIf
endIf
```

The following example reads binary data from SOUNDS.DB into a Binary variable and then passes the variable to a custom routine.

Note Custom routines for processing binary data must be written in a programming language other than ObjectPAL. Also, the language must be able to create a file called a DLL (for dynamic link library). For more information about using DLL files in ObjectPAL, search for the subject "uses" in the online ObjectPAL Help. For more information about creating DLL files, consult the documentation for the programming language.

```
var
   soundsTC TCursor
   theSound Binary
   i Smallint
endVar
if soundsTC.open("sounds.db") then
   for i from 1 to 5
      theSound = soundsTC.SoundData ; read binary data from the SoundData field
      playSound(theSound)           ; pass the data to a custom routine in a DLL
      soundsTC.nextRecord()         ; move to the next record in the table
   endFor
endIf
soundsTC.close()
```

Binary type methods are described in detail in the online ObjectPAL Help. See also the description of the OLE type later in this chapter for techniques for editing and displaying data using the OLE protocol.

Money: currency values

Money values can range from $\pm 3.4 * 10^{-4930}$ to $\pm 1.1 * 10^{4930}$ precise to six decimal places. For example, the following code, attached to a button's **pushButton** method, displays the value 1.1234567 twice: first as a Number value, then as a Currency value. The

Currency value is rounded to six decimal places. (This example assumes that your Windows Control Panel settings for Currency Format specify seven decimal places.)

```
method pushButton(var eventInfo Event)
   var
      nu Number
      cu Currency
   endVar

   nu = 1.1234567
   nu.view()          ; displays 1.1234567

   cu = 1.1234567
   cu.view()          ; displays 1.1234570
endMethod
```

The number of decimal places displayed depends on the user's Control Panel settings, but the value stored in a table does not—a table stores the full six decimal places.

Date: Calendar data

In ObjectPAL, date values can be represented in either month/day/year, day-month-year, or day.month.year format.

Important Dates must be cast (explicitly declared). For example, the following code assigns to *d* the date December 21, 1997.

```
var
   d  Date
endVar
d = Date("12/21/1997")
```

Don't omit the quotes around the date value—if you do, ObjectPAL performs division on the values.

Date values are formatted as specified by settings in the Windows Control Panel, or by ObjectPAL formatting statements.

Although you can use ObjectPAL to perform calculations on any valid date, date values stored in a Paradox table must range from Jan. 1, 100, to Dec. 31, 9999.

Dates in the 20th century can be specified using two digits for the year, as in

```
myDay = Date("11/09/59")
```

Dates in the 2nd through the 10th centuries must include three digits of the year (as in 12/17/243); dates in the 11th through 19th centuries must have four digits (12/17/1043). The year cannot be omitted completely.

A date constant must represent a valid date. Paradox knows which months have 30 and 31 days and in which years February has 29. Paradox also knows about leap centuries, should your dates range that far. You must specify a date completely: you cannot omit the day, month, or year. In other words, `Date("12/99")` is not valid.

You can use the following characters as separators in dates: blank, tab, space, comma (,), period (.), colon (:), semicolon (;), hyphen (-), or slash (/).

Table 5.4 lists commonly used methods and procedures defined for the Date type.

Table 5.4 Methods for the Date type

Name	Description
day	Extracts the day from a Date value
dow	Returns the day of the week of a Date value
month	Extracts the month from a Date value
moy	Returns the month of the year of a Date value
today	Returns the current date
year	Extracts the year from a Date value

Calculations using dates

When you perform calculations between Date values, the result is a Date value. For example, in the following example, *Result* is a Date value:

```
var D1, D2, Result Date endVar

d1 = Date("07/03/99")
d2 = Date("07/03/92")
Result = D2 - D1   ; Result = 12/31/0007
```

This lets you mix data types in complex operations, as shown in the following example:

```
var
   d1, d2 Date
   t1, t2 Time
   Result DateTime
endVar

d1 = Today()                  ; get the current date
d2 = d1 + 14                  ; adds two weeks to the current date
t1 = Time()                   ; get the current time
t2 = t1 + Time("12:00:00 PM") ; adds twelve hours to the current time
Result = d2 + t1
Message("In two weeks and twelve hours, it will be ", Result)
```

Casting date calculations

When performing calculations with Date values, cast the result as different data types to return values like the number of days between two dates, as shown in the following example:

```
var
   d1, d2 Date
   Result LongInt
endVar

d1 = Today()
d2 = Date("07/21/69")
```

```
Result = LongInt(d1 - d2) ; if today is 11/03/92, this is 8506
Message("Men first landed on the moon ", Result, " days ago.")
```

For more information about mixing data types in calculations, see Chapter 4.

DateTime: Calendar data and clock data combined

A DateTime variable stores data in the form hour-minute-second-millisecond year-month-day. DateTime values must be cast (explicitly declared). For example, the following statements assign to the DateTime variable *dt* a time of 10 minutes and 40 seconds past eleven o'clock AM and a date of December 21, 1997:

```
var dt DateTime endVar
dt = DateTime("11:10:40 am, 12/21/97")
```

The quotes around the value are required.

You can use the following characters as separators: blank, tab, space, comma (,), hyphen (-), slash (/), period (.), colon (:), and semicolon (;). DateTime values are formatted as specified by settings in the Windows Control Panel, or by ObjectPAL formatting statements.

You must specify a DateTime value completely; you cannot omit any of the fields, but you can specify a value of 0 for any field.

Table 5.5 lists commonly used methods defined for the DateTime type.

Table 5.5 Methods for the DateTime type

Name	Description
day	Extracts the day from a DateTime value
dow	Returns the day of the week of a DateTime value
hour	Extracts the hours from a DateTime value
milliSec	Extracts the milliseconds from a DateTime value
minute	Extracts the minutes from a DateTime value
month	Extracts the month from a DateTime value
moy	Returns the month of the year of a DateTime value
second	Extracts the seconds from a DateTime value
year	Extracts the year from a DateTime value

DynArray: An indexed list

A DynArray is a flexibly structured dynamic array. A dynamic array is a compact storage structure for any combination of data types. Using a DynArray, you can look up values quickly, even when the dynamic array contains a large number of items.

These arrays are dynamic because you do not specify their size; the dimensions of a DynArray automatically change as items are added to it or released from it. A DynArray's size is limited only by system memory.

Note ObjectPAL also supports fixed-size and resizeable arrays. See the description of the Array type for more information.

Unlike fixed-size arrays, the indexes of dynamic arrays are not integers; dynamic array indexes can be any valid ObjectPAL expression that evaluates to a String. Each index in a dynamic array is associated with a value.

Declaring dynamic arrays

You must declare a dynamic array before you can store values in it. To declare a DynArray, give it a name and a data type, which can be any type *except* Array, DynArray, or Record. The syntax for declaring a DynArray is

```
var
  DynArrayName DynArray[] dataType
endVar
```

DynArrayName specifies the name of the DynArray, and *dataType* specifies the data type of the items. All items of that array must be of the declared type. For example, you could declare a dynamic array of strings called *stringThings* like this:

```
var
   stringThings DynArray[]  String
endVar
```

After a dynamic array has been declared, references to items have the following syntax:

DynArrayName [*Tag*]=*Value*

DynArrayName must be unique. *Tag* can be any valid ObjectPAL expression that evaluates to a String. The data type of *Value* must match the data type declared for the DynArray.

Dynamic array items

To assign values to the items of the dynamic array *GroceryBag*, you could use the following statements:

```
var
   GroceryBag DynArray[] AnyType
   s1, s2 String
endVar
GroceryBag["Type"] = "Paper"
GroceryBag["Size"] = "Large"
GroceryBag["Double"] = True
GroceryBag["Fruit"] = "Apples"
GroceryBag["Soda"] = "Root Beer"
GroceryBag["Total"] = 3.29
GroceryBag["Frozen"] = False
GroceryBag["Date"] = Today()
s1 = "Good"
s2 = "Stuff"
GroceryBag[s1 + s2] = "Ice cream"
```

Items in a dynamic array are not ordered sequentially (as fixed array items are). The value of an item in a dynamic array is retrieved by referencing its *tag* (a label for the item's index), rather than its position within the array.

For example, this statement displays the string "Large" as the value of the item whose tag is *Size*:

```
message(GroceryBag["Size"])
```

Methods for the DynArray type (including **contains**, **size**, and **removeItem**) work like their counterparts in the Array type. See also the description of the basic language element FOREACH, which repeats specified statements for each item in a DynArray, in the online ObjectPAL Help.

DynArray operators

You can use DynArray operators (= and <>) to compare two DynArrays and to copy one DynArray to another. There are no arithmetic operators for DynArrays.

Two DynArrays are equal only if all indexes and all items in the first DynArray match the corresponding indexes and items in the second DynArray; otherwise, they are not equal. For example,

```
var
   da1, da2 DynArray[] String
endVar

da1["Name"] = "Frank Borland"
da1["Title"] = "Guru"

da2["Name"] = "Frank Borland"
da2["Title"] = "Genius"

message(da2 = da1)   ; displays False
sleep(1500)
message(da2 <> da1)  ; displays True
```

You can also use the (=) operator to copy one DynArray to another. For example,

```
var
   da1, da2 DynArray[] String
endVar

da1["Name"] = "Frank Borland"
da1["Title"] = "Philanthropist"

da2 = da1
da2.view() ; da2 is a copy of da1
```

Using copyToArray and copyFromArray

The methods **copyToArray** and **copyFromArray**, both defined for the TCursor and UIObject types, move data between a TCursor and an array or a DynArray. When used with a DynArray, **copyToArray** copies the name of each field in the current record of the

TCursor, along with the corresponding data. The resulting DynArray uses the field names as indexes, and the data as items. For example,

```
var
   tc TCursor
   dyn DynArray[] AnyType
endVar

executeQBEFile("getCust.qbe", tc)
tc.copyToArray(dyn)

msgInfo("Name", dyn["Name"])

dyn.view()
```

A major advantage to using **copyToArray** is that the DynArray created by **copyToArray** contains the field names. A regular array, on the other hand, contains only the field values and requires more work to control.

Graphic: An image

A Graphic variable provides a handle for manipulating a graphic object. In other words, you can use Graphic variables in ObjectPAL code to manipulate graphic objects. Graphic objects contain and display graphics in the following formats: bitmap (BMP), encapsulated Postscript (EPS), graphic interchange format (GIF), Paintbrush (PCX), and tagged information file format (TIF).

Using Graphic type methods **readFromClipboard**, **writeToClipboard**, **readFromFile,** and **writeToFile**, you can use Graphic variables to transfer bitmaps between forms (and reports), tables, the Clipboard, and disk files.

As an example, suppose a form contains a table frame named *staffTF* bound to the table STAFF.DB. The following code reads a graphic image from a file into a Graphic variable and then assigns the value of the variable to a graphic field named *mugShot* in the table frame:

```
var
   staffBMP Graphic ; declare a Graphic variable
endVar

if staffTF.locate("Name", "Frank Borland") then
   if staffBMP.readFromFile("borlandf.bmp") then
      staffTF.mugShot.value = staffBMP
   else
      msgStop("Stop", "Could not read from file.")
   endIf
else
   msgStop("Stop", "Could not find name.")
endIf
```

Logical: True or False

Logical variables often answer questions about other objects and operations, for example:

- Is that table empty?
- Is that form displayed as an icon?
- Did that operation successfully create a text file?

Logical variables occupy one byte of storage. They have two possible values: True or False.

Logical operators

In order of precedence, the logical operators are NOT, AND, and OR.

```
if NOT myTable.isEmpty() then
   myTable.empty()
endIf

myBox.visible = NOT(myBox.visible) ; toggles the visible property

if myBox.color = Red OR myCircle.color = Red then
   msgStop("Red alert!", "Red alert!")
endIf

if x > 5 AND y < 12 OR y > 222 then
; same as if (x > 5 AND y < 12) OR (y > 222), because AND has precedence over OR
   z = 234
else
   z = 1
endIf
```

Note There is a difference between the comparison operator (<>) and the logical operator (NOT). The (<>) operator compares any two data values, and the (NOT) operator negates a logical value.

In ObjectPAL, expressions on both sides of an AND or an OR statement are evaluated, unlike programming languages that use short-circuit evaluation.

Bitwise operations

ObjectPAL also provides methods for performing bitwise operations. The LongInt and SmallInt types include the methods **bitAND**, **bitOR**, and **bitXOR**. See the online ObjectPAL Help for more information.

LongInt: Long integers

LongInt values are small integers; that is, they can be represented by a long series of digits. A LongInt variable occupies 4 bytes of storage. ObjectPAL converts LongInt

values to range from -2,147,483,648 to 2,147,483,647. An attempt to assign a value outside of this range to a LongInt variable causes an error. For example,

```
var
   x, y  LongInt
   z     Number
endVar

x = 2147483647   ; The upper limit value for a LongInt variable.
y = 1
z = x + y        ; This statement causes an error.
```

When ObjectPAL performs an operation on LongInt values, it expects the result to be a LongInt, too. That's why the addition operation in the previous example causes an error: the result is too large to be a LongInt. To work with a boundary value (in either the positive or negative direction), convert it to a type that can accommodate it. For example,

```
var
   x, y  LongInt
   z     Number
endVar

x = 2147483647      ; The upper limit value for a LongInt variable.
y = 1
z = Number(x) + y   ; This statement succeeds.
```

In this example, ObjectPAL converts one LongInt to a Number before doing the addition, and the statement succeeds.

Note Run-time library methods defined for the Number type also work with LongInt variables. The syntax is the same, and the returned value is a number. For example, this code will work even though *sin* is defined for the Number type:

```
var
   abc LongInt
   xyz Number
endVar
abc = 43
xyz = abc.sin()
```

Note ObjectPAL supports an alternate syntax:

methodName (*objVar*, *argument* [,*argument*])

methodName represents the name of the method, *objVar* is the variable representing an object, and *argument* represents one or more arguments. For example, the following statement uses the standard ObjectPAL syntax to return the sine of a number:

```
theNum.sin()
```

The following statement uses the alternate syntax:

```
sin(theNum)
```

For clarity and consistency, it's best to use the standard syntax, although you can use the alternate syntax when it's more convenient to do so.

Memo: A large amount of text

Memo variables store text and formatting data—up to 512MB in Paradox tables. Using a Memo variable and a field object's Value property, you can transfer formatted memos between forms, reports, and tables. You can read and write the text of a memo, without formatting data, to and from a disk file using Memo type methods **readFromFile** and **writeToFile**. You can also transfer memos to and from the Clipboard, but as with disk files, you can only transfer text; formatting data is lost.

You can also use the (=) operator to assign the value of a memo field to a Memo variable or a String variable.

Note There are no arithmetic or comparison operators for Memo variables.

If you assign to a String variable, you get only the memo text without any formatting. If you assign to a Memo variable, you get the text and the formatting. For example, suppose a table named MEMOTEST.DB contains a formatted memo field named MemoField, and a form contains an unbound field object named *formField*. When you run the following method, it reads the contents of MemoField into a String variable, and then into a Memo variable. Next it displays the String variable in *formField* (text only), and then it displays the Memo variable in *formField* (formatted text).

```
var
   s  String
   m  Memo
   tc TCursor
endVar
tc.open("memoTest.db")
s = tc.MemoField
m = tc.MemoField
formField.value = s ; displays the text in formField
sleep(1500)
formField.value = m ; displays the FORMATTED text in formField
sleep(1500)
```

The following example reads the contents of a text file to a memo field in a table. Assume that a table named *PJNotes* exists in the current directory and has the following fields: *ProjDate*, a Date field, and *ProjNotes*, a Memo field. The **pushButton** method for a button named *getFile* opens, edits, and inserts a new record in the *PJNotes* table, then fills the *ProjDate* field with the current date, and fills the *ProjNotes* field with text from a file named NOTES.TXT.

```
; getFile::pushButton
method pushButton(var eventInfo Event)
var
  MemoFile Memo
  pTC      TCursor
endVar

if pTC.open("pjNotes.db") then    ; open project notes table
  if MemoFile.readFromFile("notes.txt") then
    ; if memo file read was successful
    pTC.edit()                    ; edit project notes table
    pTC.insertRecord()            ; insert a new blank record
```

```
    pTC.ProjDate = today()          ; fill the ProjDate field
    pTC.ProjNotes = MemoFile        ; write memo to ProjNotes field
    pTC.endEdit()                   ; end Edit mode
  endIf
  pTC.close()                       ; close the TC
endIf
endMethod
```

Searching for text in memos

There are no methods provided specifically for searching for text in memos. Instead, assign the value of the memo to a String variable and use String methods (for example, **advMatch**, **match**, and **search**) to search for text.

For example, suppose you have a form bound to the *Shipwrck* table, which contains a memo field named Comments. The following example shows how to prompt the user to enter text, and then search the table for a record where the Comments field contains the text the user entered. This code is attached to a button's built-in **pushButton** method.

```
var
   memoAsString, findThis String
   newPos SmallInt
   i, recMarker LongInt
   shipsTC TCursor
endVar

proc searchToEnd() Logical
   for i from recMarker to shipsTC.nRecords()
      if doSearch() then
         return True
      endIf
   endFor
   return False
endProc

proc searchFromStart() Logical
   shipsTC.home()
   for i from 1 to recMarker
      if doSearch() then
         return True
      endIf
   endFor
   return False
endProc

proc doSearch() Logical
   memoAsString = shipsTC.Comments
   newPos = memoAsString.search(findThis)
   if newPos <> 0 then
      Comments.moveToRecord(shipsTC)
      Comments.moveTo()
      Comments.action(EditEnterMemoView) ; Must be in memo view to set
                                         ; CursorPos property
```

```
      Comments.cursorPos = newPos - 1     ; Strings count from 1,
                                          ; field objects count from 0.
      Comments.action(SelectRightWord)    ; Highlight word to right of insertion
                                          ; point.
      return True
   else
      shipsTC.nextRecord()
   endIf
   return False
endProc
method pushButton(var eventInfo Event)
   findThis = "Enter text here."
   findThis.view("Enter text to search for:")

   if findThis <> "" then
      shipsTC.attach(Comments)
      recMarker = shipsTC.recNo()
      switch
         case searchToEnd() : return
         case searchFromStart() : return
         otherwise : msgInfo("Couldn't find", findThis)
      endSwitch
   endIf
endMethod
```

Variables are declared first to make them available to the built-in method and the custom procedures. The procedure *doSearch* handles the searching chore, the procedure *searchToEnd* specifies a search from the current record to the end of the table, and the procedure *searchFromStart* specifies a search from the beginning of the table.

For more information and examples, refer to the online ObjectPAL Help.

Number: Floating-point values

Number variables represent floating-point values consisting of a significand (fractional portion, for example, 3.224) multiplied by a power of 10. The significand can contain up to 18 significant digits, and the power of 10 can range from $\pm 3.4 * 10^{-4930}$ to $\pm 1.1 * 10^{4930}$. An attempt to assign a value outside of this range to a Number variable causes an error.

Note Run-time library methods and procedures defined for the Number type also work with LongInt and SmallInt variables. The syntax is the same, and the returned value is a Number. For example, this code will work even though `sin` does not appear in the list of methods for the LongInt type:

```
var
   abc LongInt
   xyz Number
endVar
abc = 43
xyz = abc.sin()
```

Note ObjectPAL supports an alternate syntax:

methodName (*objVar, argument* [*,argument*])

methodName represents the name of the method, *objVar* is the variable representing an object, and *argument* represents one or more arguments. For example, the following statement uses the standard ObjectPAL syntax to return the sine of a number:

```
theNum.sin()
```

The following statement uses the alternate syntax:

```
sin(theNum)
```

For clarity and consistency, it's best to use the standard syntax, although you can use the alternate syntax when it's more convenient to do so.

Note The display formats of numeric methods might vary depending on the Windows number format of the user's system, but ObjectPAL's internal representation is always the same.

Numeric constants

Numeric constants are written as a sequence of digits, optionally preceded by a minus sign (–) for negative numbers, and optionally containing a decimal point.

As an alternative to entering a number literally, you can use scientific notation to represent numbers, which is especially convenient for very large and very small numbers. Numbers in scientific notation begin with the decimal value and end with the letter E followed by the exponent, which can be positive (+) or negative (–).

Table 5.6 shows some examples of numeric constants:

Table 5.6 Numeric constants

Constant	Definition
25	The number 25
3.1514	A number with a decimal point
–17.00000001	A negative value with a small fraction
5.6E+9	$5.6 * 10^9$, or 5,600,000,000

Numeric constants cannot contain dollar signs or commas. Enclosing them in parentheses does not make them negative.

The numeric type specifies a number with 18 digits of precision. Numeric constants can be used for Number, Money, SmallInt, and LongInt values. Value boundaries are set according to the data type.

OLE: Object Linking and Embedding

OLE is an acronym for Object Linking and Embedding, a protocol that provides access to the function of another application without having to leave Paradox and open that application each time you want to make a change.

For example, suppose you have tables that contain bitmap graphics, and you want to create a Paradox application that enables users to edit those graphics. One approach

would be to create the graphics using a paint program that is an OLE server (defined below), and then use ObjectPAL OLE type methods to make the function of the paint program available to your users (assuming, of course, that your users have the paint program installed on their systems).

Note ObjectPAL and Paradox also support DDE (for Dynamic Data Exchange), another protocol for sharing data. The DDE type is grouped with the system data objects, because DDE data cannot be stored in a table. Refer to Chapter 27 for information on using DDE.

Table 5.7 defines terms commonly used when discussing OLE operations:

Table 5.7 OLE terms and definitions

Terms	Definition
OLE Server	An application that can provide access to its documents via the OLE mechanism. Paradox is an OLE server.
OLE Container	An application that can use the OLE mechanism to access documents created by an OLE server. Paradox is an OLE container.
OLE Object	A document created using an OLE server. It contains the data you want to use in your Paradox application.
OLE Variable	An ObjectPAL variable declared to be of type OLE. An OLE variable provides a handle for manipulating an OLE object. In other words, you can use OLE variables in ObjectPAL code to manipulate OLE objects. The OLE Variable is an OLE container.

Table 5.8 lists the methods for working with OLE objects and OLE variables. These methods let you read, write, and work with an OLE Object.

Note The OLE type also includes methods defined for the AnyType type. These methods are **blank**, **dataType**, **isAssigned**, **isBlank**, and **isFixedType**. See the online ObjectPAL Help for more information about these methods.

Table 5.8 OLE type methods

Method	Description
canLinkFromClipboard	Returns True if an OLE object can be linked from the clipboard into an OLE variable; otherwise, returns False.
canReadFromClipboard	Returns True if the OLE object can be read from the Clipboard into an OLE variable; otherwise, returns False.
edit	Launches the server and lets the user edit the object or take some other action. Paradox does not wait for the server to close; ObjectPAL execution continues without interruption.
enumServerClassNames	Fills a DynArray with the registered OLE servers on your system. The index of the DynArray is a string representing the server's name; the item at a given index is a string value representing the server's class name and can be used as the argument to **insertObject** (syntax 3).
enumVerbs	Fills a DynArray with action commands (called verbs). The index of the DynArray is a string representing the verb's name; the item at a given index is an integer value representing the actual action to perform. You can use these values as the verb argument in the **edit** method.
getServerName	Returns a string that identifies the server. The string is a descriptive name, not necessarily the file name of the server.
insertObject	**Syntax 1:** Brings up the **insertObject** dialog box and allows the user to choose what kind of OLE object to insert.

Table 5.8 OLE type methods (continued)

Method	Description
	Syntax 2: Fills the OLE variable with an OLE object created from the specified file. The Link argument specifies whether the object should maintain a link to the original file or not.
	Syntax 3: Fills the OLE variable with a new blank object for a specified server class name.
linkFromClipboard	Links (paste links) an OLE object from the Clipboard into an OLE variable. Fails if there is no OLE object in the Clipboard. When you use **linkFromClipboard**, changes made to the OLE object while in Paradox *will* affect the underlying file.
isLinked	Reports whether an OLE object is a linked object. Returns True if the OLE variable is a linked object, or False if it is an embedded object. When used with **updateLinkNow** this method provides a convenient way to update the linked OLE fields in a table.
updateLinkNow	Updates a linked OLE object and returns True if successful, or False if the OLE object is an embedded object. When used with **isLinked** this method provides a convenient way to update the linked OLE fields in a table.
readFromClipboard	Reads (pastes) an OLE object from the Clipboard into an OLE variable. Fails if there is no OLE object in the Clipboard. When you use **readFromClipboard**, changes made to the OLE object while in Paradox do not affect the underlying file.
writeToClipboard	Write the contents of an OLE variable to the Clipboard as an OLE object. This method copies the original object as if the server had done the copy.

An overview of OLE

ObjectPAL has six ways to retrieve an OLE object:

- From a table
- From the Clipboard
  - Read from Clipboard
  - Link from Clipboard
- From a file
- From a server
- From the insert object dialog box
- From an OLE UIObject

OLE from a table

To retrieve an OLE object from a table, declare an OLE variable. For example:

```
var
   oleVar OLE
endVar
```

Then, assign to the OLE variable the value of an OLE field. For example,

```
oleVar = oleTable.oleField.value
```

Call **edit**. **edit** takes two arguments: a string and an integer. Paradox passes the string to the OLE server, which can use it to inform the user about what's happening. Paradox passes the integer to the server to specify an action to take. Paradox does not wait for the server to close; ObjectPAL execution continues without interruption.

Example: Using OLE with Sound Recorder

The following example uses an OLE variable to get sound data stored in a table. The code is attached to a button's **pushButton** method, and it assumes that a table named SOUNDS.DB exists and that this table contains two fields, an alpha field named SoundName and an OLE field named SoundData. The call to **enumVerbs** fills the DynArray *oleVerbs* with the verbs (actions) supported by the OLE server (Sound Recorder), then displays the list in a pop-up menu. You can choose an item from the pop-up menu to specify an action to take. This code doesn't change the data in SOUNDS.DB; it just plays a sound.

```
var oleVar OLE endVar ; Declare oleVar outside the body of the method
method pushButton(var eventInfo Event)
   var
      oleTC          TCursor
      oleVerbs       DynArray[] SmallInt
      verbPop        PopUpMenu
      verbChoice, i  String
      chosenVerb     SmallInt
   endVar
   oleTC.open("sounds.db")
   oleVar = oleTC.SoundData

   oleVar.enumVerbs(oleVerbs)
   forEach i in oleVerbs
      verbPop.addText(i)
   endForEach

   verbChoice = verbPop.show()
   if not verbChoice.isBlank() then
      chosenVerb = oleVerbs[verbChoice]
      oleVar.edit("PdoxWin", chosenVerb) ; Paradox does not wait for the server.
   endIf                                 ; ObjectPAL code continues to execute.
   message "ObjectPAL code continues to execute."
endMethod
```

OLE from the Clipboard

The following steps retrieve an OLE object from the Clipboard:

1. Open the OLE server application and use it to open or create a document. This document is the OLE object.
2. Use the OLE server to copy the OLE object to the Clipboard.
3. In ObjectPAL, call **canReadFromClipboard** or **canLinkFromClipboard** to make sure the operation is possible. This step is optional.
4. Call **readFromClipboard** or **linkFromClipboard** to retrieve the OLE object from the Clipboard and assign it to an OLE variable.

5 Call **edit**.

6 Call **writeToClipboard**, if desired, to copy the OLE object to the Clipboard and make it available to another OLE client.

Example: Using OLE with Paintbrush

The following example uses OLE to edit a graphic from within Paradox.

1 Open the Paintbrush application that comes with Windows, and use it to open or create a graphic.

Note The *User's Guide* explains how to use OLE and Paradox interactively.

2 Use Paintbrush to copy the graphic to the Clipboard. (Consult your Windows documentation for information about using Paintbrush.)

Note If the OLE server supports DDE and you know the application-specific DDE commands to drive the OLE server, you could automate these steps. Consult the application's documentation.

3 In Paradox, create two buttons and an OLE object. Name the buttons *editButton* and *updateButton*. Name the OLE object *oleObject*. Attach code to the buttons and the underlying page as follows:

```
; page::Var window
var oleGraphic OLE endVar

; editButton::pushButton
method pushButton(var eventInfo Event)
   var
      oleVerbs      DynArray[] SmallInt
      verbPop       PopUpMenu
      chosenVerb    SmallInt
      verbChoice, i String
   endVar
   if oleGraphic.canReadFromClipboard() then
      oleGraphic.readFromClipboard()
      oleGraphic.enumVerbs(oleVerbs)
      forEach i in oleVerbs
         verbPop.addText(i)
      endForEach
      verbChoice = verbPop.show()
      if not verbChoice.isBlank() then
         chosenVerb = oleVerbs[verbChoice]
         oleGraphic.edit("PdoxWin", chosenVerb)
      endIf
   else
      msgStop("Stop", "Can't read OLE graphic from Clipboard.")
      return
   endIf
endMethod

; updateButton::pushButton
method pushButton(var eventInfo Event)
   oleObject.edit()
   oleObject.value = oleGraphic
```

```
   oleObject.endEdit()
endMethod
```

4 Run the form and click *editButton*. Paintbrush will open and let you edit the graphic, if all goes well, or display a dialog box if there's a problem. Exit Paintbrush and return to Paradox, then click *updateButton* to update *oleObject* with the edited graphic.

Example: Using enumServerClassNames and insertObject (syntax 3)

```
var
   olevar  OLE
endVar
method pushButton(var eventInfo Event)
   var
      serverClasses     DynArray[] String
      serverPop         PopUpMenu
      serverIndex, i    String
      serverClassName   String
   endVar

   ; Fill the dynarray with names of all the registered OLE servers.
   oleVar.enumServerClassNames(serverClasses)

   ; Now put those in a popup menu and let the user choose one.
   forEach i in serverClasses
      serverPop.addText(i)
   endForEach
   serverIndex = serverPop.show()

   ; Now, insert an object of that server class.
   ; This will launch the server with the new object, so the user can edit it.
   if not serverIndex.isBlank() then
      serverClassName = serverClasses[serverIndex]
      oleVar.insertObject(serverClassName)
   endIf
endMethod
```

Example: Using insertObject (syntax 2)

```
var
   olevar  OLE
endVar
method pushButton(var eventInfo Event)
   ; Insert linked object from file into the OLE variable
   oleVar.insertObject("c:\\windows\\argyle.bmp", TRUE)
endMethod

method pushButton(var eventInfo Event)
   ; Insert embedded object from file into the OLE variable
   oleVar.insertObject("c:\\windows\\argyle.bmp", FALSE)
endMethod
```

Point: A location on the screen

A Point variable holds information about a point on the screen. To ObjectPAL, the screen is a two-dimensional grid with the origin at the upper left corner of the design object's container, positive x-values extending to the right, and positive y-values extending down. A Point has an x-value and a y-value, where *x* and *y* are measured in twips (a twip is 1/1440 of an inch; 1/20 of a printer's point).

Point methods get and set information about screen coordinates and relative positions of points. For example, the size and position properties of a design object are specified in points. For a demonstration of how Point values can be used, open the *Paradox Paint* example form in the directory where you installed the ObjectPAL example files (typically, C:\PDOXWIN\EXAMPLES).

Note ObjectPAL calculates point values relative to the container of the design object in question. For example, if a box contains a button, ObjectPAL calculates the button's position relative to the box. If the button sits in an empty page, ObjectPAL calculates its position relative to the page. Methods that take or return Point values as arguments use this relative framework.

Point operators

You can use Point operators (+, –, =, <, >, <=, and >=) to add, subtract, and compare Point variables. These operators operate on the x-coordinates of each point, then on the y-coordinates. For example,

```
var
   p1, p2, p3 Point
endVar

p1 = Point(10, 30)
p2 = Point(10, 30)
p3 = Point(10, 33)

message(p1 + p2)  ; Displays (20, 60), because 10 + 10 = 20, and 30 + 30 = 60.
message(p1 = p2)  ; Displays True. Both x- and y-coordinates are equal.
message(p1 = p3)  ; Displays False. Both coordinates must be equal.
message(p3 > p1)  ; Displays False. Both coordinates must be greater.
message(p3 >= p1) ; Displays True. Both coordinates are either greater or equal.
```

Record: A user-defined data type

ObjectPAL provides the Record type as a programmatic, user-defined collection of information, similar to a **record** in Pascal or a **struct** in C.

Important Records defined as an ObjectPAL data type are separate and distinct from records associated with a table.

The syntax for declaring a Record data type is

```
type
RecordName = Record
```

```
        fieldName fieldType
        [fieldName fieldType]
        *
    endRecord
endType
```

where one or more *fieldNames* identify fields (columns) of the record, and *fieldType* is one of the basic data types. Declare records in a design object's Type window (see Chapter 4 for more information). For example, the following code declares a data type named *PartRec* to be of type Record. The *PartRec* record has three fields: *partName*, *partNumber*, and *quantity*.

```
type
PartRec = record
                partName   String
                partNumber String
                quantity   SmallInt
          endRecord
endType
```

Having declared the type, you can declare other variables to be of type *PartRec*, like this:

```
var
   myPartRec PartRec
endVar
```

Then you can assign values to the fields of the record, like this:

```
myPartRec.partName = "widget"
myPartRec.partNumber = "WW120A"
myPartRec.quantity = 174
```

You can use **view** to display a Record's values in a dialog box. For example,

```
myPartRec.view()
```

Record operators

You can use Record operators (= and <>) to compare and assign Record variables. The records must be of the same type. For example,

```
type
PartRec = Record
                partName   String
                partNumber String
                quantity   SmallInt
          endRecord
endType

var
   recOne, recTwo PartRec
endVar

recOne.partName = "widget"
recOne.partNumber = "WW120A"
recOne.quantity = 174
```

```
recTwo = recOne ; assign (copy) values of recOne to recTwo

recTwo.partName = "gadget" ; assign new value to partName field of recTwo

if recOne <> recTwo then
   beep()   ; beeps because records are not the same
endIf
```

Note ObjectPAL has a number of predefined records. If you have variables of the type ReportPrintInfo, ReportOpenInfo, FormOpenInfo, or BrowserInfo, you will automatically have a record. See the online ObjectPAL Help for the **open** method of the Form type for more information.

SmallInt: Small integers

SmallInt values are small integers; that is, they can be represented by a small (short) series of digits. A SmallInt variable occupies 2 bytes of storage. ObjectPAL converts SmallInt values to range from -32,768 to 32,767. An attempt to assign a value outside of this range to a SmallInt variable causes an error. For example,

```
var
   x, y SmallInt
   z    LongInt
endVar

x = 32767 ; The upper limit value for a SmallInt variable.
y = 1
z = x + y ; This statement causes an error.
```

When ObjectPAL performs an operation on SmallInt values, it expects the result to be a SmallInt, too. That's why the addition operation in the previous example causes an error: the result is too large to be a SmallInt. To work with a boundary value (in either the positive or negative direction), convert it to a type that can accommodate it. For example,

```
var
   x, y SmallInt
   z    LongInt
endVar

x = 32767          ; The upper limit value for a SmallInt variable.
y = 1
z = LongInt(x) + y ; This statement succeeds.
```

In this example, ObjectPAL converts one SmallInt to a LongInt before doing the addition, and the statement succeeds.

Note The value –32,768 cannot be stored in a SmallInt field of a Paradox table because, to Paradox, –32,768 = Blank. However, you can use this value in calculations and you can store it in a dBASE table.

Note Run-time library methods and procedures defined for the Number type also work with LongInt and SmallInt variables. The syntax is the same, and the returned value is a

Number. For example, the following code will work, even though *sin* does not appear in the list of methods for the SmallInt type:

```
var
   abc LongInt
   xyz Number
endVar
abc = 43
xyz = abc.sin()
```

Note ObjectPAL supports an alternate syntax:

methodName (*objVar*, *argument* [,*argument*])

where *methodName* represents the name of the method, *objVar* is the variable representing an object, and *argument* represents one or more arguments. For example, the following statement uses the standard ObjectPAL syntax to return the sine of a number:

```
theNum.sin()
```

The following statement uses the alternate syntax:

```
sin(theNum)
```

For clarity and consistency, it's best to use the standard syntax, although you can use the alternate syntax when it's more convenient to do so.

String: A series of characters

A String variable can contain up to 32,767 characters (use Memo variables for longer text). Use double quotes ("") to represent an empty string. String variables occupy 1 byte of storage per character.

Quoted strings

In a method, enclose character strings in double quotation marks (""). Do this to enter or edit data in a table, to type literals on a form or a report specification, to specify a field, and to set design object properties.

Note A quoted string can contain up to 255 characters, but a String variable can contain up to 32,767 characters. For example,

```
var
   a, b, c, d, e, f, g String
endVar

; each quoted string below contains 50 characters
a = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
b = "bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb"
c = "ccccccccccccccccccccccccccccccccccccccccccccccccc"
d = "ddddddddddddddddddddddddddddddddddddddddddddddddd"
e = "eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee"
f = "fffffffffffffffffffffffffffffffffffffffffffffffff"
```

```
g = a + b + c + d + e + f
g.view()
; g can contain more than 255 characters because
; it's a String variable, not a quoted string.
```

Quoted strings are stored in a Windows resource file (consult your Windows documentation for information) and can be edited using Borland's Resource Workshop—in other words, you don't need to run Paradox to edit them.

Note String type methods **fill**, **pattern**, and **space** cannot work with strings longer than 1,000 characters.

Working with strings

You can use String methods to combine and compare strings, to search for and replace strings, and to convert case.

Combining strings

You can combine strings using the + operator, a process known as concatenation. For example,

```
myName = "Frank " + "Borland" ; stores "Frank Borland" in myName
message("Mr. " + myName) ; displays Mr. Frank Borland
```

Comparing strings

You can also use the comparison operators =, <>, <, >, <=, and >= with strings. Both the = and <> operators check for exact matches, but you can use **ignoreCaseInStringCompares** to make operations case-insensitive. The < and > operators compare strings one character at a time. If they are the same, the next character is checked, and so on until the nonmatching string is found.

```
var
   bb String
endVar

bb = "To be or not to be,"
bb = bb + " that is the question." ; concatenation
; result is "To be or not to be, that is the question."

if bb = "hello" then           ; comparison
   message("They match!")
else
   message("No match here.")
endIf         ; they don't match

message("A" < "B") ; displays True
sleep(2000)
message("a" < "B") ; displays False
sleep(2000)

message("A" < "a") ; displays True
sleep(2000)
```

```
message("A" = "a") ; displays False
sleep(2000)
ignoreCaseInStringCompares(True)
message("A" = "a") ; displays True
sleep(2000)

message("Paradocks" < "Paradox") ; displays True
; at the first nonmatching character, "c" < "x"
```

Searching

Use **advMatch** or **match** to search for one string in another. Both are powerful methods with many options. To return a specified portion of a string, use **substr**. Refer to the String topic of the online ObjectPAL Help for details and examples.

Converting case

To convert a string to lowercase letters, use **lower**. To convert a string to uppercase letters, use **upper**.

```
var s1 String endVar
s1 = "abcd"
message(s1.upper()) ; displays ABCD
sleep(2000)
message(s1.lower()) ; displays abcd
sleep(2000)
```

The empty string

An empty string is denoted by consecutive double quotes (""), *not* a space (ANSI 32).

Other examples

Following are more examples of valid strings:

```
myTable."Net_Profit".font.color = "Green"
; "Net Profit" is a field of myTable
; "Green" is the color of field Net Profit
msgInfo(" ", "Hello, world") ; displays "Hello, world" in a dialog box
msgInfo(" ", "This string
spans two lines.") ; displays a two-line string in a dialog box
```

Backslash codes in strings

Certain frequently used ANSI codes, such as the tab character (ANSI 9) have special backslash sequences similar to those in the C programming language. These sequences, shown in Table 5.9, are preferred in strings because they are more readable than their numeric equivalents.

For example, use a double backslash in a quoted string for a directory path:

```
orders.open("c:\\pdoxwin\\forms\\orders.fsl")
```

Table 5.9 Backslash codes

Character	Function
\a	Bell (^G)
\b	Backspace
\f	Formfeed ^(L)
\n	Newline (^J)
\r	Carriage return (^M)
\t	Tab (^I)
\v	Vertical tab
\"	Double quote (")
\\	Backslash (\)
\xxx	3-digit ASCII code less than 128 (example: \009 for Tab)

Any character not shown in Table 5.9 cannot be preceded by a backslash: it will cause a syntax error when you compile the code.

Time: Clock data

Time variables store times in the form hour-minute-second-millisecond. You can use the following characters as separators: blank, tab, space, comma (,), hyphen (-), slash (/), period (.), colon (:), and semicolon (;).

Time values must be cast (explicitly declared). For example, the following statements assign to the Time variable *ti* a time of 10 minutes and 40 seconds past eleven o'clock in the morning:

```
var ti Time endVar
ti = Time("11:10:40 am")
```

The quotes around the value are required. To set Windows time formats from Paradox, use the procedure **FormatSetTimeDefault** defined for the System type.

Table 5.10 lists commonly used methods defined for the Time type.

Table 5.10 Methods for the Time type

Name	Description
hour	Extracts the hours from a Time value
milliSec	Extracts the milliseconds from a Time value
minute	Extracts the minutes from a Time value
second	Extracts the seconds from a Time value

Variables, constants, and containership

Before you attach ObjectPAL code to an object, it is important to understand how a variable's *scope* affects which objects have access to it, and how *containership* affects the accessibility or availability of variables, constants, and procedures to other objects. This chapter covers

- Containers
- How to use variables
- The scope of a variable
- How to define constants
- Passing arguments

Containership

A design object's behavior is defined by its methods, its properties, and its visual context. The methods are the built-in methods and the custom code you write. The properties are characteristics set interactively at design time or by ObjectPAL statements at run time. Containership—an object's visual, spatial relationship to other objects—supplies the context.

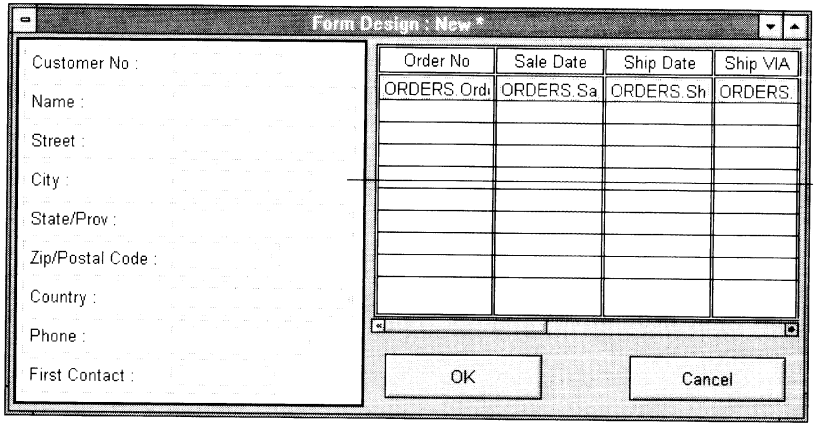
Objects in a form coexist in a hierarchy of containers. For example, when you place a table frame on a page of a form, the form contains the page, and that page contains the table. Position in this hierarchy is important because it defines what an object can get from other objects, including their methods, procedures, properties, and variables. The availability of resources follows what you see on the screen.

Important An object is contained if it is *completely within* the boundaries of the container. (Objects that scroll—for example, field objects in a table frame—are also contained.) But, if you

turn off an object's Contain Objects property, ObjectPAL does not treat that object as a container.

An object has direct access to its own methods, procedures, properties, and variables and to those declared in the objects that contain it. For example, suppose a form contains a group of field objects as shown in Figure 6.1, and you want each one to display a prompt when the user moves the insertion point into it. Instead of attaching code to each field object, you could draw a box around them and attach the code to the box. The field objects have direct access to custom methods and procedures attached to the box, because the box contains the field objects.

Figure 6.1 Using a container to store shared code



For another example, suppose a page in a form contains two buttons, as shown in Figure 6.2. Variables declared in the page's Var window are available to both buttons. So, you could attach code like the following to declare a Number variable and assign it a value, then use the buttons to increase or decrease the variable's value:

page::Var window

```
; page::Var window
Var
   theNumber Number
endVar
```

page::open method

```
; page::open
method open(var eventInfo Event)
   theNumber = 0
endMethod
```

buttonOne::pushButton method

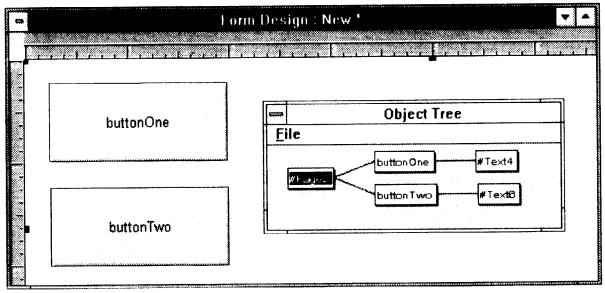
```
method pushButton(var eventInfo Event)
   theNumber = theNumber + 1
   message(theNumber)
endMethod
```

buttonTwo::pushButton method

```
method pushButton(var eventInfo Event)
   theNumber = theNumber - 1
   message(theNumber)
endMethod
```

Because the variable *theNumber* is declared in the page's Var window, every object contained in the page can access *theNumber*.

Figure 6.2 Containership and variables



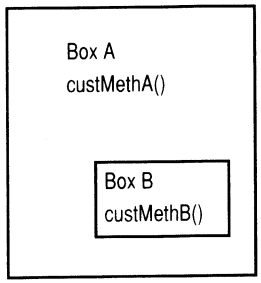
The underlying page contains both buttons, so a variable declared in the page's Var window is available to both buttons.

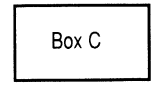
Tip By attaching custom methods, custom procedures, and variables to the form, you make them available to all objects the form contains, because the form is the highest-level object in the containership hierarchy. Forms can't share their custom procedures or variables with other forms because there's no higher-level container you can attach them to. (However, you can store custom code in a library and make the library available to multiple forms. See Chapter 28 for more information.)

Containership and custom methods

Containership affects the availability of custom methods. An object has direct access (no dot notation required) to its own custom methods, and to those in objects that contain it. Also, because custom methods are public, you can use dot notation to call custom methods attached to objects in other containership hierarchies. For example, Figure 6.3 shows three boxes, named *A*, *B*, and *C*. Box *A* contains box *B*.

Figure 6.3 Containership and custom methods





Box A contains box B, and box C is in a separate containership hierarchy.

The custom method **custMethA** is attached to box A, and the custom method **custMethB** is attached to box B.

Using the containership hierarchy shown in Figure 6.3, the following statements are true.

- Box *B* can call the custom method **custMethB** directly because **custMethB** is attached to box *B*:

```
custMethB()
```

- Box *B* can also call the custom method **custMethA** directly because **custMethA** is attached to box *A*, which contains box *B*:

```
custMethA()
```

- Box *C* can use dot notation to call **custMethB**, attached to box *B*:

```
b.custMethB()
```

- Box *C* can use dot notation to call **custMethA**, attached to box *A*:

```
a.custMethA()
```

- Box *A* must use dot notation to call **custMethB**:

```
b.custMethB()
```

- Box *C* can make the following call because of the containership hierarchy:

```
b.custMethA()
```

In the previous case, Paradox searches the code attached to box *B* for **custMethA** and when it doesn't find it, proceeds to search *B*'s container. When Paradox finds **custMethA** attached to box *A*, it executes the code; otherwise, it searches box *A*'s container, and so on up the hierarchy. When calling a custom method, the dot notation does not necessarily specify which object to operate on. Instead, it specifies which object to search for the code.

Note If two or more objects in a form have the same name, you'll have to use dot notation to specify which object to work with.

Using Subject

The object variable *Subject* specifies which object a custom method should operate on. When calling a custom method, dot notation specifies which object is the subject of the method. Using the previous example, suppose the code for the custom method **custMethA**, attached to box *A*, is

Attached to box A

```
method custMethA()
   Subject.color = Red
endMethod
```

When box *C* makes the statement `b.custMethA()`, box *B* turns red because box *B* is the subject.

When box *C* makes the statement `a.custMethA()`, box *A* turns red because box *A* is the subject.

When box *B* makes the statement `custMethA()`, box *B* turns red because Paradox recognizes box *B* as the subject of the method.

Important *Subject* is not the same as *Self*, another object variable. Dot notation specifies the subject; *Self* is always the object the code is attached to. For example, suppose the code for custom method **custMethB** is changed to this:

Attached to box B

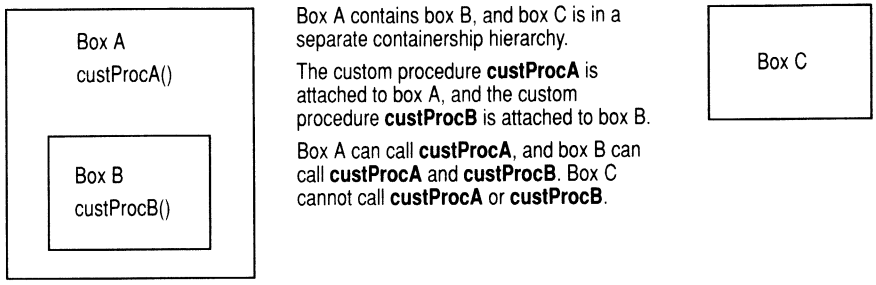
```
method custMethB()
   self.color = Red
endMethod
```

In this case, when an object calls **custMethB**, box *B* always changes color, regardless of dot notation, because **custMethB** is attached to box *B* and *Self* is always the object the code is attached to. For more information about using *Self*, refer to Chapter 13.

Containership and custom procedures

Containership affects the availability of custom procedures. An object has access to its own custom procedures and to those declared in objects that contain it. Custom procedures are private, which means you cannot call custom procedures attached to objects in other containership hierarchies. For example, Figure 6.4 shows three boxes, named *A*, *B*, and *C*. Box *A* contains box *B*.

Figure 6.4 Containership and custom procedures



Using the containership hierarchy shown in Figure 6.4, the following statements are true.

- Box *B* can call the custom procedure **custProcB** because **custProcB** is attached to box *B*:

  ```
  custProcB()
  ```

- Box *B* can also call the custom procedure **custProcA** because **custProcA** is attached to box *A*, which contains box *B*:

  ```
  custProcA()
  ```

- Box *C* cannot call **custProcB**, attached to box *B*.
- Box *C* cannot call **custProcA**, attached to box *A*.
- Box *A* cannot call **custProcB**, attached to box *B*.

Variables

A *variable* is like a slot where you can temporarily store one item of information. The value of a variable can be of any ObjectPAL type.

The simplest way to give a variable a value is to use the assignment operator (=). For example, the following statement assigns the string "abc" to variable x:

```
x = "abc"
```

If *x* had been assigned a value previously, the former value would now be lost.

You must assign values to variables before you use them in expressions. If, for example, the following instruction is executed before *x* is assigned a value, it fails:

```
myMsg = "The value of x is " + string(x)
```

You can use **isAssigned** to test whether a variable has been assigned a value, as in this example:

```
if myVar.isAssigned() then
   myVar = myVar + 1
else
  myVar = 1
endIf
```

You don't need to explicitly indicate a data type for variables: ObjectPAL assumes you want to use one of the data types listed in Table 4.1. For example, this sequence of statements makes *x* a String variable, then a SmallInt, then a Number.

```
x = "abc"       ; x is a String
x = 5           ; x is a SmallInt
x = 3.238       ; x is a Number
```

To use variables of a more complex type (for example, Array or TCursor), you must declare them. The following statement will not compile unless the variable *ordersTC* is declared somewhere within the scope of the statement:

```
ordersTC.open("orders.db")
```

In addition, there are advantages to explicitly declaring which data type to use, as explained in the next section.

Declaring variables

Declaring a variable means specifying its type before using the variable. It's considered good practice to declare variables. Following are some advantages:

- The compiler can catch typing mistakes and inconsistent usage.
- The compiler can optimize your code to make it run faster.
- Code is "self-documenting" and more readable.

You can declare variables using the following structure:

```
var
   varName1                varType1
   varName2a, varName2b    varType2
   ; and so on
endVar
```

var and **endVar** are ObjectPAL keywords. *varName1*, *varName2a*, and *varName2b* represent names you choose for your variables (variables must be named according to the rules presented earlier in Chapter 4); and *varType1* and *varType2* specify the data types. Here's an example:

```
var
   i, j, amount   Number
   FirstName, LastName String
endVar
```

This code declares five variables: *i*, *j*, and *amount* are variables of type Number; *FirstName* and *LastName* are variables of type String.

You can instruct the compiler to warn you about undeclared variables: in an Editor window, choose Properties | Compiler Warnings.

Scope of a variable

The term "scope" means "accessibility." The *scope* of a variable, that is, the range of objects that have access to it, is defined by the object in which the variable is declared, and by the container hierarchy. Objects can access only their own variables and the variables defined in the objects that contain them. Also, the scope of a variable depends on where it is declared. You can declare a variable

- Within a method
- Outside a method
- In the Var window
- Within the containership hierarchy (compile-time binding)

Declared within a method

Variables declared within a method are visible only to that method, and are accessible only while that method executes. They are initialized (reset) each time the method executes. For example, in the following method, *myNum* always equals 1 because it is declared and initialized each time the **pushButton** method executes:

```
method pushButton (var eventInfo Event)
   var
      myNum SmallInt
   endVar
   if not myNum.isAssigned() then
      myNum = 1
   else
      myNum = myNum + 1
   endIf

   message(myNum) ; displays 1
endMethod
```

Declared outside a method

Variables declared in a method window *before* the word **method** are visible only to that method, but are *not* initialized each time the method executes.

In the following example, *myNum* is declared outside the **pushButton** method (but within the same Method window), so its value is incremented by 1 each time the method is called. (To insert code above the first line of a method, move the insertion point to the left of the **m** in **method**, press *Enter* one or more times to insert blank lines, and type as usual.)

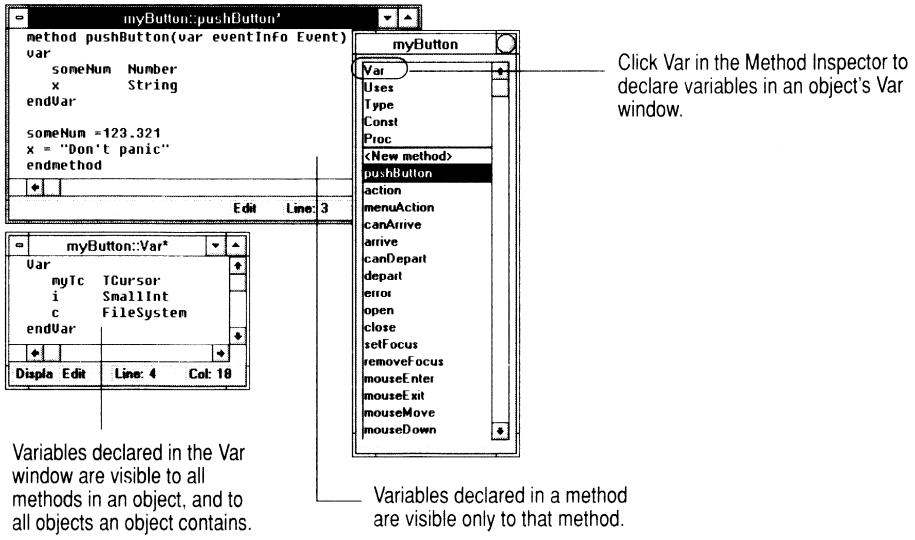
```
   var
      myNum SmallInt
   endVar

method pushButton (var eventInfo Event)
   if not myNum.isAssigned() then
      myNum = 1
   else
      myNum = myNum + 1
   endIf

   message(myNum)
endMethod
```

Declared in the Var window

Variables declared in an object's Var window are visible to all methods attached to that object, and to any objects *that* object contains. A variable declared in an object's Var window is attached to the object and is accessible while the object exists in the form and the form is open.

To declare variables in the Var window, inspect an object, choose Methods, then double click on Var. The Var window is an Editor window, just like the Method window. Figure 6.5 shows variables declared in a Method window and in a Var window.

Figure 6.5 Declaring variables in windows



Scope: An example

The pseudocode in Figure 6.5 shows how the scope of a variable depends on where the variable is declared.

Figure 6.6 Scope of variables

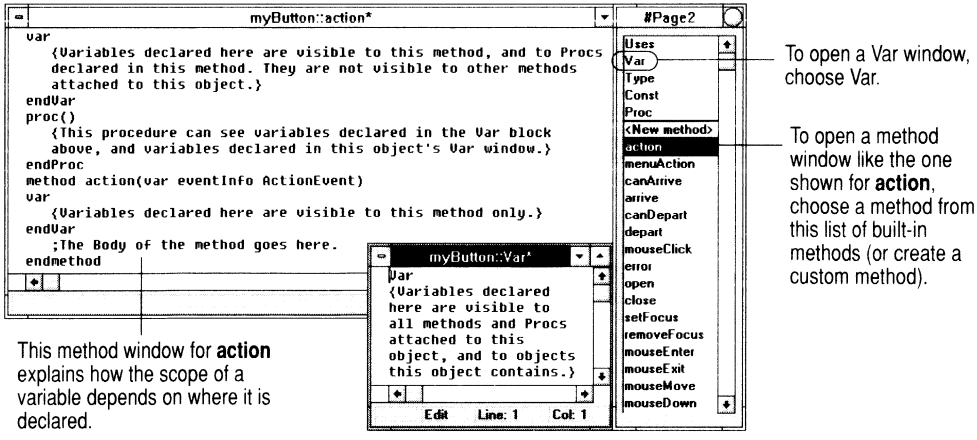
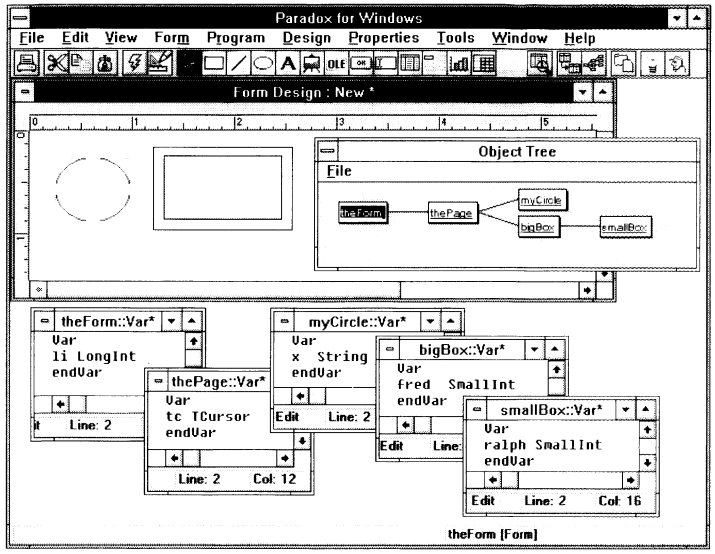


Figure 6.7 shows a form containing an ellipse, *myCircle*, and a rectangle, *bigBox*, which contains another rectangle, *smallBox*.

Figure 6.7 The containership hierarchy



In this figure, Var windows mimic the Object Tree, which diagrams the containership hierarchy.

Objects can see variables declared above them in the same branch, but cannot see variables declared in other branches. For example, *smallBox* can see *fred*, declared in *bigBox*, and *tc*, declared in *thePage*, but it can't see *x*, declared in *myCircle*.

In Figure 6.7, *bigBox* can see the variable *fred* because it was defined in the *bigBox* Var window. Methods attached to *bigBox* can access *fred* directly:

```
method mouseEnter (var eventInfo MouseEvent)
   fred = 123  ; sets value of variable fred to 123
   message (fred)
endMethod
```

An object has direct access to variables defined in objects that contain it. So, *smallBox* can see the variable *fred* too, because *smallBox* is contained in *bigBox*. So *smallBox* can access *fred* directly, as well:

```
method mouseExit (var eventInfo MouseEvent)
   fred = 21 ; sets value of BigBox variable fred to 21
   message (fred)
endMethod
```

Also, *bigBox*, *smallBox*, and *myCircle* all have direct access to *tc*, declared in the Var window for the underlying page, *thePage*.

However, *bigBox* cannot see *ralph*, a variable declared in the *smallBox* Var window, because *smallBox* is lower in the containership hierarchy. *BigBox* can't see variables declared in *myCircle*. Methods in *myCircle* cannot access variables declared in *bigBox* and *smallBox*.

Note When you turn off an object's Contain Objects property, ObjectPAL does not treat that object as a container. Other objects in its borders do not have direct access to its variables.

The form is the highest level in the containership hierarchy. Variables declared in a form's Var window are visible to all objects in the form. Using Figure 6.7 as an example, suppose a method attached to *smallBox* assigns a value of 100 to the variable *li* (declared in the form). Then, suppose a method attached to *myCircle* contains the statement

```
msgInfo("Adding 1 to variable li", li + 1)
```

This statement displays the value 101 in a dialog box because *myCircle* can see the variable *li* and knows that its value is 100.

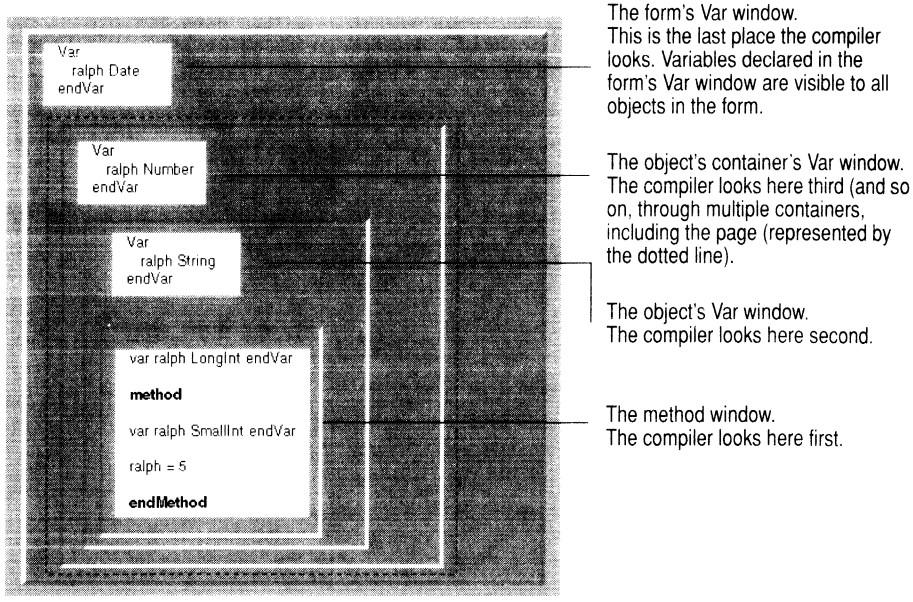
Note The relationship between containership and scope is a key concept in ObjectPAL. It applies not only to variables, but to object properties (discussed in Chapter 13), custom methods and procedures attached to objects (demonstrated in the examples in online ObjectPAL Help), the way objects handle events, and the way ObjectPAL handles errors (discussed in Chapter 30).

Compile-time binding

In programming terms, "binding" a variable is the process of connecting a variable to a data type. The ObjectPAL compiler binds variables when it compiles the source code; there is no run-time binding in ObjectPAL. When the compiler encounters a variable in a statement, it searches the rest of the source code to find out where the variable is declared so it can bind the variable to the declared data type.

Figure 6.8 shows where the compiler looks for variable declarations. It looks first between method and endMethod, then above method, then in the object's Var window, then in the Var window of the object's container, and so on until it reaches the form. The compiler works with the first appropriate declaration it finds, and it stops looking when it finds one. If a variable is not declared, the compiler treats it as an AnyType variable.

Figure 6.8 Compile-time binding



You can declare variables at any level in the container hierarchy, but you don't have to. If all you want are global variables (visible to all objects in a form) and local variables

(visible only to the object in which they're declared), you can declare global variables in the form's Var window and declare local variables in methods and procedures as needed.

Note Global variables are global to the form where they are declared. You can't declare a variable to be global to more than one form.

Lifetime of a variable

Variables you declare in an object's Var window persist as long as the object does. If the object is removed from the form, its variables are no longer accessible. You can't release variables explicitly.

Variables you declare in a method or procedure are accessible only while the method or procedure is running.

User-defined data types

If you select Type from an object's Methods dialog box, you can define synonyms for existing data types using this structure:

```
type
   newTypeName = existingType
endType
```

Data types you declare follow the same scoping rules as variables. Declaring a new type does not create a new data type; instead, it creates a mnemonic synonym for an existing type. This example declares a new data type, Salary, based on the existing type SmallInt.

```
type
   Salary = SmallInt
endType
```

You can use this new type to declare variables. For example, these two declarations are the same:

```
var
   payCheck Salary
endVar

var
   payCheck SmallInt
endVar
```

However, the first is easier to remember, and in a large application, it might be easier to maintain. Suppose you get a substantial raise and a SmallInt is too small. Instead of changing every declaration of *payCheck*, you just change the declaration of Salary:

```
type
   Salary LongInt
endType
```

One useful type is the Record. An ObjectPAL Record is similar to a **record** in Pascal or a **struct** in C. Records defined in an object's Type window are separate and distinct from

records associated with a table; ObjectPAL recognizes them as a separate type. For more information, see Chapter 5.

The following code declares a Record as a data type:

```
type
   EmployeeRec = Record
                    empName String
                    deptNum Number
                    title   String
                 EndRecord
endType
```

After you declare the type, you can declare other variables to be of type EmployeeRec, and you can assign values to the fields of the record:

```
var
   FrankBorland = EmployeeRec
endVar
FrankBorland.empName = "Frank Borland"
FrankBorland.deptNum = 43
FrankBorland.title = "Genius"
```

It can also be convenient to declare a user-defined type for an array:

```
type
   BaseArray Array[1200] String
endType
```

Then you can use the type name to declare other arrays of the same type:

```
var
   myArray, yourArray BaseArray
endVar
```

Blank variables

In Paradox, an empty field is said to be *blank*. When you store blank values in ObjectPAL variables, three states are possible:

- *Blank* (sometimes called Null) means the variable has no value. This is the only state supported in Paradox tables. The empty string ("") is considered to be blank.
- *Err* results from invalid calculations (for example, dividing by 0) in a calculated field. Err takes precedence over Blank, so when both occur as arguments, the result is Err; that is, Err + Blank = Err. When Err is stored in a Paradox table, it's converted to Blank.
- *UnAssigned* results when a variable has no assigned value. It takes precedence over Blank, but not over Err. UnAssigned values cause errors at run time.

The Err and UnAssigned states are provided as an aid to debugging.

To find out if a variable is blank, use the AnyType type method **isBlank**. To find out if a variable has been assigned a value, use the **isAssigned** method (all types).

isBlank is not the opposite of **isAssigned**. **isBlank** reports on a state of the value of a variable (whether it has the "blank" value). **isAssigned** reflects a state of the variable (whether it's initialized for use), not the value of the variable. **isBlank** only works with variables that have an assigned value. In the following example, the variable *testMe* is neither assigned nor blank:

```
Var testMe SmallInt endVar
message(testMe.isAssigned())          ; displays False
message(testMe.isBlank())             ; displays False
```

The following statements assign the variable *testMe* a value of 1, so it is assigned but not blank:

```
Var testMe SmallInt endVar
testMe = 1
message(testMe.isAssigned())          ; displays True
message(testMe.isBlank())             ; displays False
```

The following statements explicitly assign a blank value to *testMe*, making it both assigned and blank:

```
Var testMe SmallInt endVar
testMe = blank()
message(testMe.isAssigned())          ; displays True
message(testMe.isBlank())             ; displays True
```

You can use the Session procedure **blankAsZero** to treat blank values as zeros in calculations, but it's recommended that you explicitly enter the value—eliminating any chance for confusion—rather than leave the field blank.

When working with character strings, ObjectPAL treats an empty string ("") as a blank value. Also, given a blank value, the String procedure **isSpace** returns True. For example,

```
var
   testString String
endVar
testString = "" ; assign the empty string to testString

message(testString.isAssigned()) ; displays True
message(testString.isBlank())    ; displays True
message(testString.isSpace())    ; displays True
```

Defining constants

Constants are like variables, except they're protected from change when the program runs, enabling the compiler to generate more efficient code. You can define constants for a single method, or open a Const window to define constants for all the object's methods. It's good practice to give symbolic names to numbers and strings, so the meaning of a constant is evident. This lets you change a value by modifying one definition, instead of searching sources for "magic numbers."

For example, if you declared the constants *myMAX*, *myMIN*, and *greeting* in an object's Const window, you could refer to them in any method attached to that object and in methods attached to objects that object contains.

```
const
   myMAX = SmallInt(25)
   myMIN = SmallInt(10)
   greeting = "Here's looking at you, kid."
endConst
```

If the boundaries for minimum and maximum values change, or if you prefer a different greeting, you can make the changes in one place, the Const window.

The structure for declaring a constant is

```
const
   constName = value
   ; or
   constName = typeName (value)
   ; to cast the type
endConst
```

Important You can define constants for a single method, or open a Const window to define constants for all the object's methods.

When you declare a constant, Paradox infers the constant's data type from its value as either a Number, a SmallInt, or a String. If you want a constant to have some other data type, you must *cast* (assign) it explicitly, as shown in the following example.

```
const
   x = 123.45              ; Number, inferred
   a = -1000               ; SmallInt, inferred
   s = "11-Nov-92"         ; String, inferred

   m = Money(123.45)       ; Money, cast
   n = Number(-1000)       ; Number, cast
   d = Date("11/11/92")    ; Date, cast
endConst
```

ObjectPAL constants

The ObjectPAL language includes many predefined constants. For example, Blue is defined as a constant with a value of 16711680, which specifies the color blue. The following statements are equivalent:

```
thatBox.color = Blue

thatBox.color = 116711680
```

However, it's easier to remember a word than a long number.

ObjectPAL defines constants for many things besides colors. You can find information about them in the following places:

- In the online ObjectPAL Help
- In a Paradox ObjectPAL Editor window, by choosing Tools | Constants

- In a table you create using the System procedure **enumRTLConstants**

Passing arguments

ObjectPAL supports the following conventions for passing arguments to methods and procedures:

- By reference, that is, a reference or pointer to the original value. The value of an argument you pass by reference can be changed in the called method or procedure. Use the keyword var, followed by the argument name and data type. For example, `var myNum Number`.
- By value, a copy of the original value. (The original value cannot be changed by the called method or procedure, but the copy—the passed value—can be changed.) Use the argument name, followed by its data type. For example, `theWord String`

  You cannot pass DDE, Database, Query, Session, Table, or TCursor variables by value.
- As a constant that can't be modified. (The compiler does not allow any modification.) Passing as a constant passes a pointer to the original value. Use the keyword const, followed by the argument name and data type; for example,

  ```
  const noChange Date
  ```

All ObjectPAL types can be passed by reference (var) or as constants (const). The following types can be passed by value: AnyType, Array, Binary, Money, Date, DateTime, DynArray, Graphic, Logical, LongInt, Memo, Number, OLE, Point, Record, SmallInt, String, Time, and UIObject. These types can also be returned by custom methods and procedures; other types cannot.

Passing an argument by reference (var) or as a constant (const) can be more efficient than passing by value because you're working with pointers to the value, rather than with a copy of the value itself. The following sections present examples that show the effects of passing by reference, passing by value, and passing as a constant. Each example passes the variable *myVal* to the custom procedure *addOne*. When *myVal* is passed by reference, the procedure *addOne* changes the value of *myVal*. When it's passed by value or as a constant, the value of *myVal* does not change.

Passing by reference

The following example uses the var keyword in a custom procedure (these techniques also work in custom methods) to pass an argument by reference. When the **pushButton** method executes, it declares a variable *myVal*, assigns it a value of 0, and passes it to the procedure **addOne**. The first line of **addOne** declares that it takes one argument, *myVal*, of type SmallInt, and returns a value of type SmallInt. The procedure adds 1 to *myVal*, calls **view** to display the new value of *myVal* in a dialog box, and returns the new value of *myVal* to the **pushButton** method. The **pushButton** method then calls **view** to display the value of *myVal* in another dialog box.

```
proc addOne(var myVal SmallInt)
   myVal = myVal + 1
   myVal.view("proc")        ; display the value of myVal (it's 1)
endProc

method pushButton(var eventInfo Event)
var myVal SmallInt endVar    ; declare the variable
   myVal = 0                 ; assign an initial value
   addOne(myVal)             ; call the procedure
   myVal.view("method")      ; display the value of myVal (it's 1)
endMethod
```

Built-in methods pass the argument *eventInfo* by reference, which means you can change its values as well as extract them. For more information about working with *eventInfo*, see Chapter 10.

Passing by value

The next example is just like the first, except the var keyword is omitted from the procedure **addOne**. As a result, *myVal* is passed by value; that is, the value of *myVal* is passed to the procedure and the procedure changes that value, but the value of the *myVal* variable in the method does not change.

```
proc addOne(myVal SmallInt)
   myVal = myVal + 1
   myVal.view("proc")        ; in the procedure, myVal = 1
endProc

method pushButton(var eventInfo Event)
var myVal SmallInt endVar
   myVal = 0
   addOne(myVal)
   myVal.view("method")      ; in the method, myVal = 0 (it's unchanged)
endMethod
```

Passing as a constant

The next example uses the const keyword in a procedure to pass a value as a constant. If you run the syntax checker on this method, you get the error message `Error: Constant variable can't be assigned a value`. The const keyword in the **addOne** procedure declares *myVal* to be a constant, and by definition a constant value cannot change.

So, the following statement causes the error because it tries to change the value of *myVal*:

```
myVal = myVal + 1
```

The **addOne** procedure looks like this:

```
proc addOne(const myVal SmallInt)
   myVal = myVal + 1     ; this line causes a compiler error
   myVal.view("proc")
endProc

method pushButton(var eventInfo Event)
```

```
   var myVal SmallInt endVar
   myVal = 0
   addOne(myVal)
   myVal.view("method")
endMethod
```

The following code uses a place *holder* variable holder to correct the error. The variable *holder* is declared above the procedure and the method to make it visible to both. Now, instead of trying to change the value of *myVal*, 1 is added to *myVal* and the results are stored in *holder*. Then *holder* is returned to the calling method.

```
var holder SmallInt endVar ; declare a place holder variable
                           ; visible to the proc and the method
proc addOne(const myVal SmallInt)
   holder = myVal + 1
endProc

method pushButton(var eventInfo Event)
   var myVal SmallInt endVar
   myVal = 0
   addOne(myVal)
   myVal.view("myVal")         ; displays 0
   holder.view("holder")       ; displays 1
endMethod
```

Where to put code

One of the hardest aspects of learning ObjectPAL is knowing where to attach code. Indeed, much of the information presented in this manual is related to this important issue. As you gain experience with ObjectPAL, you will get better at determining the most appropriate places to put code. This chapter presents some background information and guidelines to help get you started.

This chapter covers

- Code placement
- Levels of code
- Global versus local variables

Placement considerations

Every design object comes with built-in code, so when you place objects in a form you're literally programming. Only when you want an object to do something different (or something more) do you have to attach custom ObjectPAL code. At this stage, you're faced with this important question:

Where should I put my code?

To answer it, you need to answer two other questions:

1 When should the code execute?

2 How many objects will use this code?

3 Which method should I use?

When should the code execute?

To answer the first question, you need to understand when an object's built-in methods execute. Built-in methods execute in response to events. What's an *event*? It's a message

to an object, generated by some activity. Anything you do in Paradox generates an event. For example, opening a form, adding a record, or editing a field. Be sure to read Chapter 10 for a detailed description of the ObjectPAL event model.

Built-in methods execute in response to events.

When you interact with an object, you generate events. Paradox responds to events by calling methods. More specifically, when an action generates an event, Paradox constructs a packet of data about the event, determines which object was the target of the event, and sends the packet to that object. The event packet triggers one of the target object's built-in methods, and code for that method executes.

For example, if you want something to happen when the user clicks a button, attach your code to that button's built-in **pushButton** method. If you want something to happen when the user moves the mouse pointer into a box, attach your code to that box's built-in **mouseEnter** method. If you want something to happen when the user changes a value in a field, attach your code to that field's built-in **changeValue** method. (All the built-in methods, including **pushButton**, **mouseEnter**, and **changeValue** are described in Chapter 12.)

You don't have to write methods for all the events an object can handle. All objects have built-in methods for ObjectPAL events, and an event never goes unrecognized. If you add your own code, you can specify when (and if) the built-in code executes. Using ObjectPAL, you can write methods that handle events, methods that pass events to other objects, and methods that do both.

How many objects will use this code?

To answer the second question, you need to understand containership, as explained in Chapter 6. Table 7.1 lists some general guidelines.

Table 7.1 Guidelines for attaching code to objects

Objects using code	Where to attach code
One object, one method	The object
One object, many methods	A custom method or procedure attached to the object
Many objects in the same form	A custom method or procedure attached to a container object or library
Many objects, many forms	In a library

Code that operates on an object doesn't have to be attached to the object. For example, the code to open a table could be attached to a button, a field object, or any other object you can place in a form. When the code executes, it opens the table. Where you attach the code depends on when you want the code to execute.

The following table summarizes the various factors you should consider when deciding which built-in methods to attach code to.

Table 7.2 Code placement guidelines

What	When	Need code?	Where
Initialize variables	When form opens	yes	Form's **open** method
Validate field values	When user changes a value	maybe	Field's **changeValue** method, or use picture and validity checks
Create unique key values	When a user moves to a field	no	Use auto-increment field type
Display a message	When user moves to a field	yes	Field's **setfocus** method

Which method should I use?

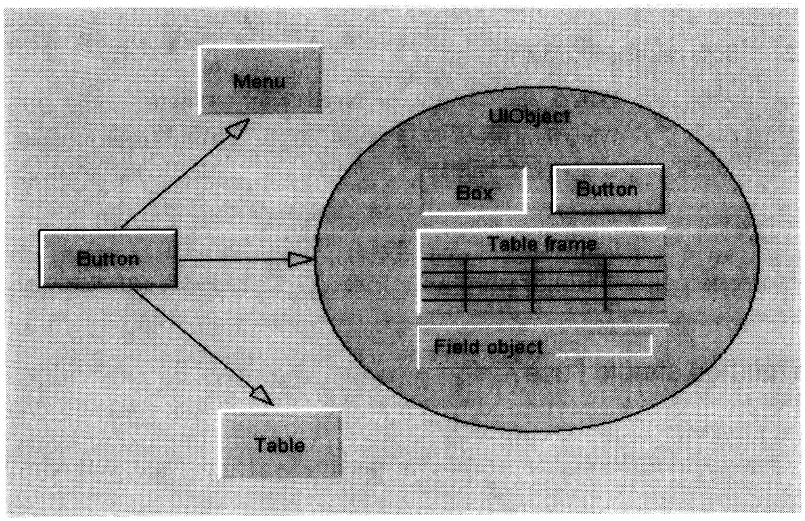
How do you know which type and which method to use? Ask yourself, "What type of *object* am I working with?" Every object has a type—even simple objects like character strings, dates, and numbers have types (respectively, String, Date, and Number).

In most cases, an object and its type have the same name: Table type methods operate on tables, String type methods operate on strings, Menu type methods operate on menus, and so on. The exception is the UIObject type, which includes methods that operate on buttons, boxes, fields, table frames, multi-record objects, and the rest of the objects you can create using tools in the Toolbar.

Note Pages in a form are UIObjects, and the form itself can behave as a UIObject. See Chapter 13 for more information.

When you're deciding which method to use, it's important to distinguish between the object the method is attached to and the object the method operates on. For example, even though a button is a UIObject, you can attach methods to it that operate on objects of any ObjectPAL type. The type of object you're operating on dictates which methods to use. To operate on a table, use Table methods; on a menu, use Menu methods; on a table frame, a multi-record object, a field, a box, or a button, use UIObject methods (those objects are UIObjects).

Figure 7.1 Which method? Which object?



The objects you're operating on dictate which methods to use. For example, code attached to a button can operate on any other object, but it must use methods appropriate for the type of object it operates on.

Code placement

Before you start thinking about where you should be placing your ObjectPAL code, you should build the tables for your application and design a form (or forms) for the user to interact with. The more you know about the properties and capabilities of the objects you've placed on the form, the better-equipped you'll be to make code placement decisions.

When is code needed?

After you've placed all the objects, run the form and observe how the objects behave by default, before you attach any code. In many cases, the default behavior will be what you want. You have to attach code to an object only if you want it to do something more or something different. Understanding the fundamentals of the event model and the default behavior of an object's built-in characteristics can save you a lot of time and trouble experimenting and over-coding. Be sure to read Chapter 10 and Chapter 12.

Take a modular approach

Because 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. This 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.

The modular approach also makes objects portable and reuseable. Forms and applications are easier to maintain because objects can be added, changed, and deleted without affecting other objects.

The closer-is-better principle

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. If you find you later need to use the same code for another object, move the code up the container path to the lowest container both objects can see.

Levels of code

Objects in a form coexist in a hierarchy of containers, or levels. Code can be attached to an object on a page, the object's container, the form, or a library. An object's position in this hierarchy is important because it defines what an object can get from other objects, including their methods, procedures, properties, and variables. This section describes these levels of code.

Script level

Although you will seldom use the script level in an application, scripts are useful if you want to execute code without opening and displaying a form window. A script consists of ObjectPAL code in its own file, not attached to a form. It's an object that doesn't contain any design objects. A script has the built-in methods **run**, **error**, and **status** that you can execute using Paradox interactively, or call from within an ObjectPAL method or procedure. Like any other object, a script also has windows for declaring variables, constants, procedures, types, and external routines. You can also declare custom methods.

Library level

A library is a file that stores custom methods, custom procedures, variables, constants, and user-defined data types. Using libraries, you can store and maintain frequently used routines and share code and variables among several forms (referencing and setting variables shared between forms, for example). Libraries can be thought of as a way to code above the form, the highest container level. Use libraries to improve the performance of multi-form applications, and to reduce the amount of code you need to maintain in at the form-level.

Form level

Because every event goes to the form first, the form has a chance to handle an event before it reaches the intended target, and in the case of external events, after it reaches

the intended target as well. When you attach code at the form level, it's important to know whether the form is handling an event on its own behalf or filtering the event before passing it to another object. ObjectPAL provides a method (**isPreFilter**) that answers both questions at once. See Chapter 10 for more information about the **isPreFilter** method and attaching code to the form.

Consider the following when deciding where to place code in a form.

Table 7.3 Reasons to and reasons not to place code at the form level

Advantages	Disadvantages
High-level control	Can be harder to maintain as objects are added, changed, and deleted.
Centralized code	Harder to handle exceptions
Code is visible to all objects in form	Can lead to large **if...then** blocks and large **switch...case** blocks

Note Forms are limited to a maximum of 64K of code, and a single method cannot be larger than 32K, including comment text. A form's symbol table cannot be larger than 64K. A symbol table is a list of all identifiers encountered when a form is compiled, their locations, and their attributes. The symbol table is kept by the compiler to verify or resolve references to different identifiers.

Page level

Use the page level if you have a multi-page form and you need to distinguish between pages. If you have a multi-page form with different custom menus on each page, use the page's built-in **menuAction** method when you need to trap for the user's menu selection.

You could attach all your menu-handling code to the form, but there's a trade-off in terms of modularity and flexibility. By attaching code to the page, you can add and delete, cut, copy and paste objects within and between forms without having to maintain large blocks of code at the form level.

Container level

An object has direct access to its own methods, procedures, properties, and variables and to those declared in the objects that contain it. For example, suppose a form contains a group of field objects, and you want each one to display a prompt when the user moves the insertion point into it. Instead of attaching code to each field object, you could draw a box around them and attach the code to the box. The field objects have direct access to custom methods and procedures attached to the box, because the box contains the field objects. See the discussion on containership and scoping in Chapter 6.

Object level

Although you can write scripts and store code in libraries, the vast majority of ObjectPAL code is attached to objects in forms. In most cases, you put objects such as

fields, buttons, and table frames within a page or in a container and attach code directly to them.

Elements of objects

The elements of an object are the lowest level of the containership hierarchy. Many objects are compound objects, that is, an object made of two or more objects.

- A field object has an edit region, and when its field's DisplayType is Labeled, it has a text-label object. You can attach code to built-in methods at the field level, the edit region, and the text label level. You'll typically attach code to the field object, not the label or the edit region. You will most frequently attach code to the **newValue**, **keyPhysical**, **action**, **changeValue**, **setFocus** and **removeFocus** built-in methods.

  Use a form's or field object's **keyPhysical** method to trap for all keyboard events. It's called when a key is pressed and each time a key autorepeats. **keyPhysical** goes to the form first, and the form dispatches it to the active object. Then, the active object's built-in code sorts out whether a keystroke represents an action or a character to display in a field, and calls the appropriate **action** method (called when **keyPhysical** maps to some keystroke to an action), or **keyChar** method (for any keyboard event that does not map to an action). Don't use the **keyPhysical** method on objects containing the field object, as the keystroke is processed by the field object before it goes to the container.

- A button has two levels where you can attach code: the button object and the text object it contains. You will most frequently attach code to a button's **pushButton** built-in method, not the text object.

- Table frames and multi-record objects are more complex compound objects. In most cases you will place code at the field level, the record label, and the table frame. You won't normally attach code on the header or on the column labels of a table frame. You will most frequently attach code to the **canArrive**, **canDepart**, **action**, **newValue**, **changeValue**, **setFocus**, **removeFocus**, and **error** built-in methods.

Variables: global or local

When you declare variables, the compiler can optimize your code and improve the performance of your application.You can declare variables at any level in the container hierarchy, but you don't have to. If all you want are global variables (visible to all objects in a form) and local variables (visible only to the object in which they're declared), you can declare global variables in the form's Var window and declare local variables in methods and procedures as needed. Variables declared at the form level are the most global variables. Use library routines to reference and set variables shared between forms. See Chapter 6 for an in-depth explanation of the relationship between containership and scope in ObjectPAL.

The ObjectPAL Editor

The ObjectPAL Editor provides the functions of a Windows text editor, along with special functions (like a compiler that checks your syntax) for editing ObjectPAL methods. In the ObjectPAL Editor, you can access dialog boxes that display the syntax for each method, properties and property values for each object, and constants for things like window attributes and error codes.

The Editor is connected to the ObjectPAL compiler; the compiler translates the ObjectPAL code you write into machine code a computer can execute. When you use the Editor, the compiler can check your code and report any syntax errors so you can correct them before you try to run the application.

The ObjectPAL Editor and Debugger (described in the next chapter), along with the design window, constitute the integrated development environment (IDE). You can edit your methods, debug your code, and inspect objects to display the methods, keywords, actions, and constants supported by ObjectPAL.

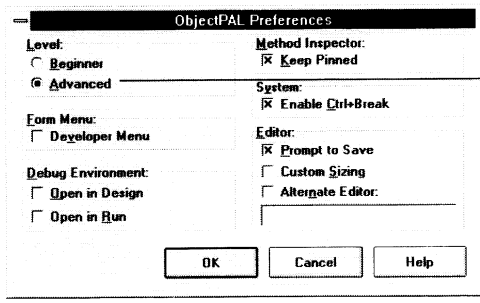
This chapter describes how to

- Start the Editor
- Use the Method Inspector
- Work with the Editor menus, Toolbar, and keyboard shortcuts
- Close the Editor
- Specify an alternate editor

Setting your ObjectPAL Preferences

Many elements of the IDE can be controlled by setting ObjectPAL preferences. To set your ObjectPAL preferences, choose Properties | ObjectPAL to open the ObjectPAL Preferences dialog box, as shown in Figure 8.1.

Figure 8.1 The ObjectPAL Preferences dialog box



The tutorial assumes an ObjectPAL level setting of Beginner; the rest of this manual assumes a setting of Advanced.

- *Level:* The IDE can show you the entire ObjectPAL language or only a subset of the essential elements. Select Beginner if all you want to see is the subset (the default setting), or select Advanced to see everything in the language. This setting affects the IDE only; it does not affect code. Code executes identically at either level, and you can use advanced elements in code even when the level is set to Beginner.

  If you're new to ObjectPAL, keep the ObjectPAL level set at Beginner. That way, you can easily see and select what you need; you don't have to navigate through language elements you aren't likely to use. The tutorial section of this manual assumes an ObjectPAL level setting of Beginner; the rest of the manual assumes a setting of Advanced.

- *Form Menu:* Check Developer Menu to use the extended Developer menu in design mode. The Developer menu includes an extra option—the Program menu—and extra commands on the View, Properties, and Tools menus. These are commands that otherwise appear only in the Editor or Debugger menus.

- *Debug Environment:* Check Open In Design to keep the Debugger environment open in design mode. Check Open In Run to open up the Debugger whenever you run a form. The Debugger environment consists of the Debugger window and the Watches, Breakpoints, Tracer, and Call Stack windows. Choose Properties | Save Debug State in the Debugger to save the current size, location, and state (e.g., minimized) of each of these windows.

  If you don't check either of these boxes, the debug environment will open whenever a breakpoint is hit, and close when you return to the design window.

- *Method Inspector:* Check Keep Pinned to pin the Method Inspector to the Desktop in the design window. The Methods window will automatically open with the design window, and stay open until you leave design mode or close the window. You can also pin the Method Inspector by clicking the box in the top-right of the window (see Figure 8.2).

- *System:* Check Enable Ctrl+Break if you want to be able to halt execution of a form by pressing *Ctrl+Break*. Otherwise, *Ctrl+Break* has no affect.

  If Enable Ctrl+Break is checked and you turn Properties | Compile With Debug on, you can suspend execution and run the Debugger by pressing *Ctrl+Break*, just as if a breakpoint had been encountered. Otherwise, pressing *Ctrl+Break* halts execution of a form.

- *Editor:*
  - Check Prompt To Save if you want the Editor to prompt you to save changes when you close the Editor window or run a form. Contents of an Editor window are saved to the form. To save to disk all the changes made in the Editor windows, save the form.
  - Check Custom Sizing if you want Editor windows to open to the size of the active Editor window (if one is open), or the size of the last Editor window open.
  - Check Alternate Editor to use a different editor. The editor must be able to save text-only files without formatting codes. See "If you choose to use another Editor to write and edit methods, it must be able to save text-only files, without any formatting codes. Specify an alternate editor in the ObjectPAL Preferences dialog box (choose Properties | ObjectPAL). Enter the full path to your editor of choice, for example, C:\APPS\BRIEF\B.EXE." later in this chapter for more information about using an alternate editor.

Starting the Editor

To start the ObjectPAL Editor,

1 Inspect an object in a form, and choose Methods from its menu. (Or, select an object, and choose View | Methods or press *Ctrl+Spacebar.*)

  The Method Inspector opens, as shown in Figure 8.2, with a list of the methods and procedures you can edit. It also includes the items Uses, Var, Type, Const, and Proc. These items are explained in the next section.

2 Select the item(s) you want to edit:
  - To select several contiguous items, use *Shift+*click, or drag over the items.
  - To select several noncontiguous items, use *Ctrl+*click.

3 Press *Enter,* or right-click anywhere in the Method Inspector to display its menu, and choose Open. (If you're selecting only one item in the Method Inspector, you can double-click the item to open an Editor window).

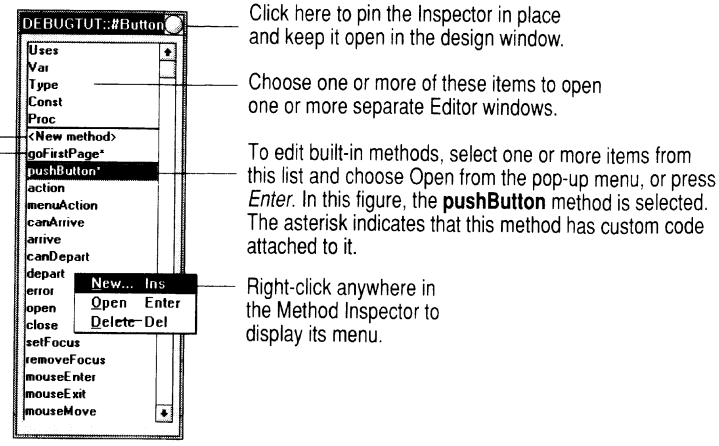
  A separate window opens for each item you selected. You can open as many windows as your system allows, in any order.

Note An asterisk appearing next to an item in the Method Inspector indicates that it has custom code attached to it.

Using the Method Inspector

Use the Method Inspector to open, create, and delete methods and procedures and to declare external routines (Uses), variables, data types, constants, and procedures.

Figure 8.2 The Method Inspector



You must edit each method in its own window; however, you can edit more than one method at a time by opening multiple windows.

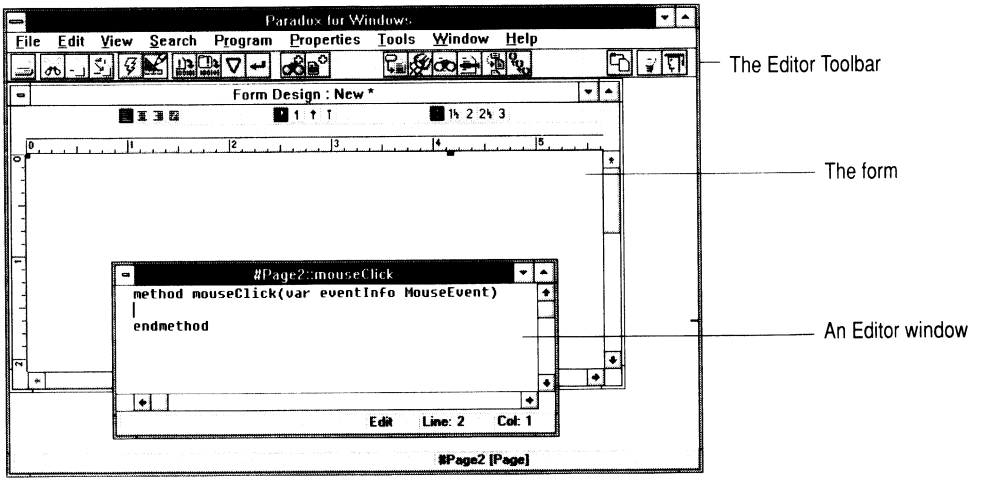
Working with the Editor

When an Editor window opens for the first time, some default text appears. For example, if you're editing the **open** method, the first line reads `method open (var eventInfo Event)`, the second line is blank, and the third line reads `endMethod`. If you accidentally change the default text, you can edit it as you would any other text.

The first time you open an Editor window for a method, the insertion point is positioned on line 2 so you can start typing right away, but you don't *have* to start typing on that line. You can use the mouse or arrow keys to move the insertion point around, and you can insert blank lines by pressing *Enter*.

Note Variables, constants, and procedures declared in a method's Editor window are visible only to that method. To make a variable, constant, or procedure visible to all of an object's methods, create it in a separate window by selecting Uses, Var, Type, Const, or Proc items in the Method Inspector. This is called scoping. For more information about scoping, see Chapter 6.

Figure 8.3 The Editor



When you open an Editor window, the menu bar changes to provide the code-editing functions described in the following sections.

See "Editor Shortcuts" later in this chapter for a description of the Editor pop-up menu, and the Toolbar buttons and keyboard shortcuts available in the Editor and the Debugger.

Using the Editor menus

Items in the File, Window, and Help menus operate as described in the *User's Guide*. You can choose File | Print to print the contents of the active Editor window or ObjectPAL Tracer window.

The rest of the Editor menus are described in the following tables. Searching for text and using commands on the Editor's Tools menu are described in more detail in the following sections.

Table 8.1 The Editor's Edit menu commands

Edit command	Description
Undo	Undoes your last edit.
Undo All Edits	Undoes all edits since last saving or opening active Editor window.
Cut	Copies selected text to the Clipboard and deletes it from the window. Cut is dimmed if nothing is selected.
Copy	Copies selected text to the Clipboard. Copy is dimmed if nothing is selected.
Paste	Copies text from the Clipboard to the location of the insertion point. If text is selected, it is replaced with the Clipboard contents. Paste is dimmed if no text is in the Clipboard.
Copy To	Copies selected text—*not* the contents of the Windows Clipboard—to a text file you specify.

Table 8.1 The Editor's Edit menu commands

Edit command	Description
Paste From	Lets you specify a text file to paste into the current method at the insertion point.
Select All	Selects all text in the window.

Table 8.2 The Editor's View menu commands

View command	Description
Methods	Displays the Method Inspector.
Debug	Opens the Debugger window. The Debugger window and the Watches, Breakpoint, Tracer, and Call Stack windows, are discussed in the next chapter.
Watches	Opens the Watches window.
Breakpoints	Opens the Breakpoints window.
Tracer	Opens the Tracer window.
Call Stack	Opens the Call Stack window.
Data Model	Displays the active form's data model in the Data Model Designer window (not available for libraries and scripts). See the *User's Guide* for more information.

Table 8.3 The Editor's Search menu commands[1]

Search command	Description
Find	Opens the Find dialog box, as shown in Figure 8.9. Use the Find dialog box to search an Editor window or a design document for a specified string.
Find Next	Searches for the next occurrence of the specified text. Search Next is dimmed if you have not searched for anything in this window.
Replace	Opens the Find And Replace dialog box, as shown in Figure 8.9. Use Replace to search for text and replace it with a value you specify.
Replace Next	Replaces the next occurrence of text specified in Replace. Replace Next is dimmed until something is replaced.
Goto Line	Moves to a specific line. Choosing Search | Goto Line displays a dialog box. Enter a line number and choose OK. If the line number doesn't exist, the insertion point won't move.
Next Warning	Displays the next warning found by the compiler.

1. See "Searching for text" later in this chapter for more information.

Table 8.4 The Editor's Program menu commands

Program command	Description
Run	Runs the form.
Check Syntax	Compiles and checks the syntax of the code in the active Editor window. If syntax errors are found, an error message appears in the status bar.
Compile	Compiles and checks the syntax of all the code in the form, library, or script. If syntax errors are found, the first Editor window containing an error is opened and an error message appears in the status bar.
Deliver	Creates a compiled form, library, or script, stripped of its ObjectPAL source code. User cannot edit the design or the source code. For more information about delivering objects, see Chapter 33.
Add Watch	Adds a watch to a variable. You can then track the variable's value in the Watches window while the form or method executes. See the next chapter for more information.
Toggle Breakpoint	Toggles a breakpoint on or off in a method to suspend execution at specified lines. See the next chapter for more information.

Table 8.5 The Editor's Properties menu commands

Properties command	Description
Desktop	Displays a dialog box for setting the properties of the Desktop window.
ObjectPAL	Opens the ObjectPAL Preferences dialog box. See "Setting your ObjectPAL Preferences" earlier in this chapter.
Compiler Warnings	When this item is checked, messages in the status bar warn you about undeclared variables and other conditions that might cause errors at run time. These messages are suppressed when the menu item is not checked. Also, the **errorShow** procedure provides more detailed error information when this item is checked.
Compile with Debug	When this command is checked, debug information is available when you run a form, library, or script. This means that execution is suspended whenever the DEBUG statement is encountered in your code. (Placing a DEBUG statement in a method or procedure has the same effect as setting a breakpoint at that line.) Even without a DEBUG statement in your code, running a form, library, or script with this command checked lets you step though it, instead of over it, when it's called from code in the Debugger. You also receive more detailed error and Tracer information when Compile With Debug is on. Turning on Compile With Debug has the same effect as setting a breakpoint in the Editor.

Table 8.6 The Editor's Tools menu commands

Program command	Description
Project Viewer	Opens the working directory's Project Viewer.[1]
Data Model Designer	Opens the Data Model Designer. Use the Data Model Designer to create a data model without creating a form, report, or query. Data Model Designer is not available when editing libraries or scripts.[1]
Object Tree	Displays a tree diagram showing how objects in the current form are related.[2]
Utilities	Displays the Paradox Utilities menu, which lets you copy, rename, or delete any Paradox object, carry out common file operations, and perform specialized operations on your tables.[1]
Multiuser	Displays the Paradox Multiuser menu, which lets you set preferences for your network use and get information about other users in a multiuser environment.[1]
System Settings	Displays the Paradox System Settings menu, which lets you get or change information about your system and environment.[1]
Types	Displays a dialog box listing all object types and their methods and procedures.[2]
Properties	Displays a dialog box listing objects and their properties.[2]
Constants	Displays a dialog box that lists ObjectPAL constants.[2]
Keywords	Displays a cascading menu of frequently used words that are reserved by Paradox.[2]
Browse Source	Creates and displays a report listing the source code in the current form, library, or script. Paradox stores the data for this report in a table named PAL$SRC.DB in your private directory. This table contains each object's method name and its source. You can then use this information to create a report.

1. See the *User's Guide* for more information.
2. See the next section, "More about the Tools menu," for more information.

More about the Tools menu

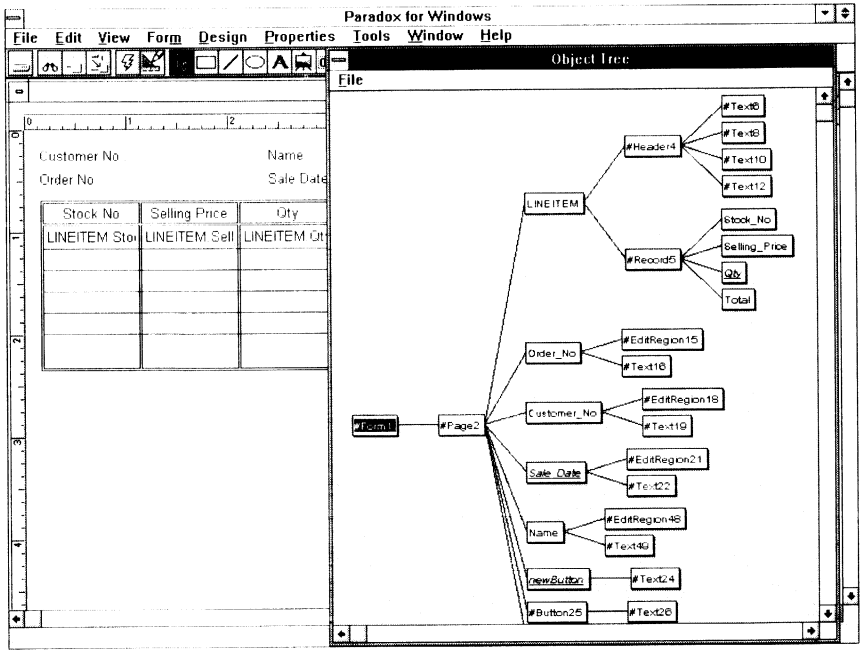
This section describes in more detail some of the items on the Editor's Tools menu.

Using the Object Tree

The Object Tree illustrates the relationships between objects in the current form. The diagram shows the object hierarchy, with the currently selected object at the far left and the tree showing the container hierarchy extending to the right. See Figure 8.4. Object Tree is not available when editing libraries or scripts.

When you place an object in a form, Paradox gives it a default name that begins with a pound sign (#). The Object Tree shows objects you've placed and named, and objects you've placed but haven't named. If you've written methods for an object, its name is underlined. You can inspect an object in the Object Tree and choose Methods to display the Method Inspector. You can use the mouse or arrow keys to move around in the Object Tree.

Figure 8.4 The Object Tree

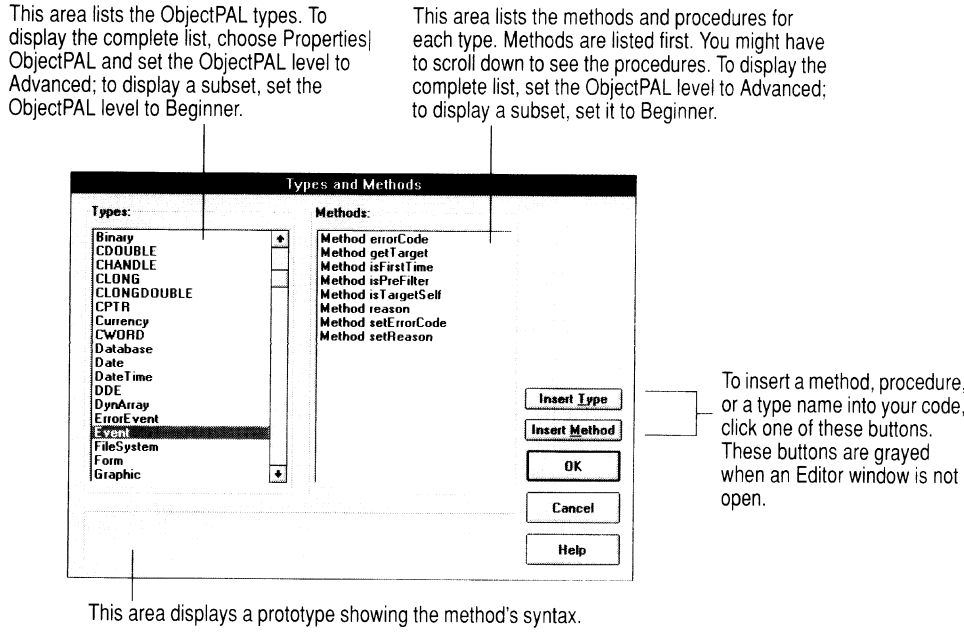


Types

Choose Tools | Types to display a dialog box listing all object types and their methods and procedures. You can choose a type name to display the methods and procedures for that type. The methods are listed first, in alphabetical order, followed by the procedures, also in alphabetical order. You might have to scroll through the list to display the procedures. You can insert a prototype of the method or procedure into your own method at the current insertion point. For example, to display a list of methods and procedures for the Array type, choose Array.

Figure 8.5 shows the dialog box that appears when you choose Tools | Types.

Figure 8.5 The Types dialog box



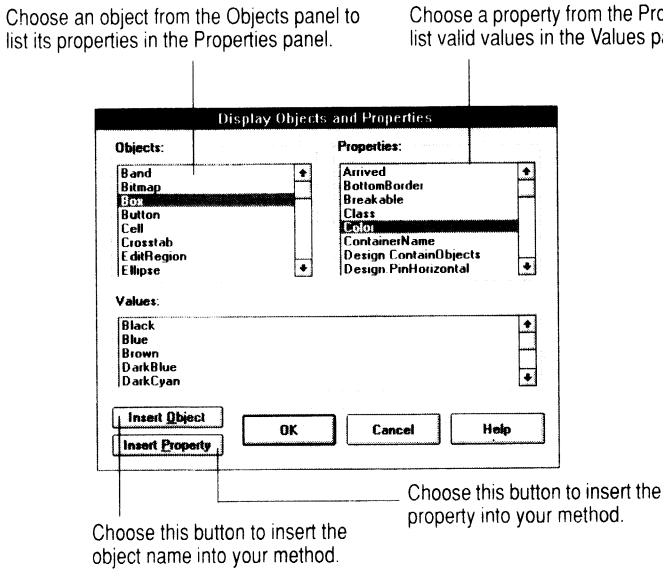
Important The ObjectPAL Level set in the ObjectPAL Preferences dialog box affects which types and methods are listed in this dialog box. The Beginner level lists a subset of the essential elements (the default setting); the Advanced level lists everything in the language. Code executes identically at either level, and you can use advanced elements in code even when the level is set to Beginner.

Properties

Choose Tools | Properties to display a list of objects and their properties. You can choose an object name to display the properties for that object. You can then insert the property into your method. For example, to display a list of Button properties, choose Button.

Figure 8.6 shows the dialog box that appears when you choose Tools | Properties.

Figure 8.6 The Display Objects and Properties dialog box

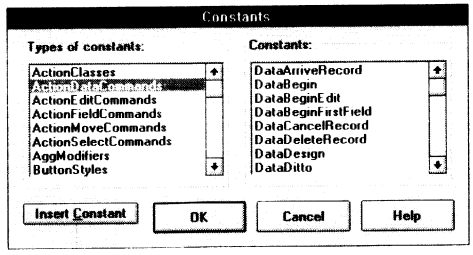


Constants

Choose Tools | Constants to display a list of ObjectPAL constants. ObjectPAL provides many constants so you can easily specify things like colors, mouse shape, menu attributes, and window styles. You can select a constant and choose Insert Constant to insert it into your code.

Figure 8.7 shows the dialog box that appears when you choose Tools | Constants.

Figure 8.7 The Constants dialog box



ObjectPAL constants are grouped into categories according to when and how they are used.

The Types Of Constants panel lists the categories. Choose a category name from this panel to display a list of constants in the Constants panel.

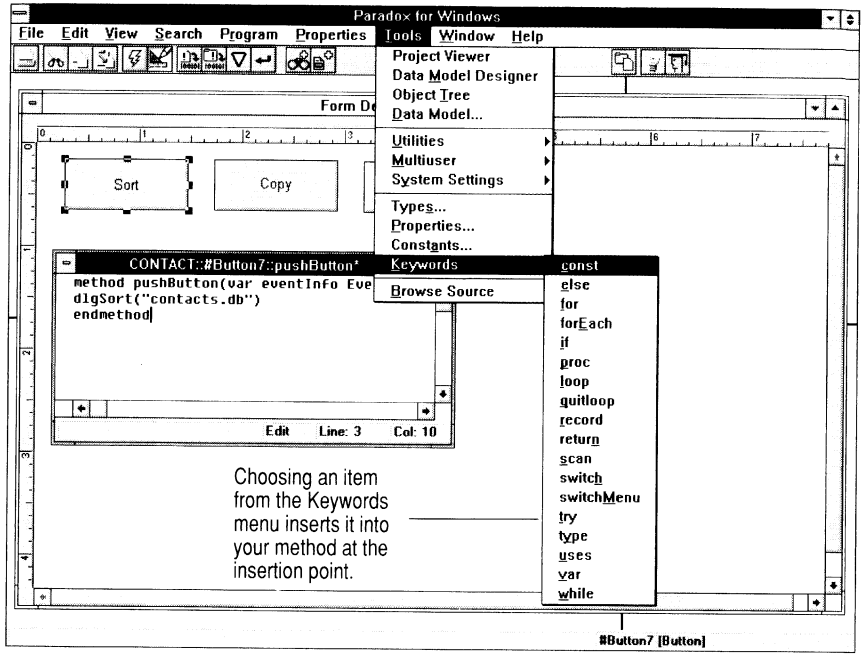
Choose this button to insert the selected constant into your method.

Keywords

Choose Tools | Keywords to display a cascading menu of frequently used words that are reserved by Paradox. Select a keyword from this menu to insert it into a method's Editor window without having to type it. The keywords are basic language elements, described in the online ObjectPAL Help.

Figure 8.8 shows the menu that appears when you choose Tools | Keywords.

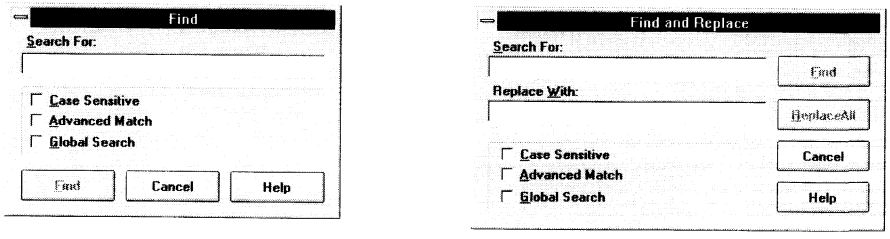
Figure 8.8 The Keywords menu



Searching for text

Choose Search | Find or Search | Replace to search for text in Editor windows. The Find dialog box searches the text from the insertion point forward. The Find And Replace dialog box lets you replace the specified text with a specified value.

Figure 8.9 The Find and the Find And Replace dialog boxes



You can use one of the following options when searching for text:

- Check Case Sensitive if you want to locate only those values that match capitalization of the value you enter in the Search For text box.

- Check Advanced Match if you want to use Paradox's advanced wildcard operators. See the *User's Guide* for a list of the advanced wildcard operators you can use to match patterns.
- Check Global Search to search all methods and procedures for the specified text.

In the Find or Find And Replace dialog boxes, choose Find to begin searching for the specified value. When located, the text is selected. You can then press

- *Ctrl+A* to find the next occurrence of the string.
- *Ctrl+R* to replace the selected text with the replacement value specified in the Find And Replace dialog box. If there isn't a replacement value specified, the located text is deleted.
- *Ctrl+L* to replace all occurrences of the string with the replacement value specified in the Find And Replace dialog box.

Using the keyboard

This section explains how to use the keyboard to move the insertion point and edit code in an Editor window. Other keyboard shortcuts are listed in Table 8.7, later in this chapter.

You can move around in a window one character or line at a time using the arrow keys. Use *Ctrl* ← and *Ctrl* → to move the insertion point one word at a time.

Home moves the insertion point to the beginning of a line. *End* moves the insertion point to the end of a line. *Ctrl+Home* moves to the beginning of the text, and *Ctrl+End* moves to the end.

PgUp and *PgDn* move through text one window at a time.

Backspace deletes the character to the left of the insertion point; *Del* deletes the character to the right of the insertion point.

By itself, *Ins* doesn't do anything. The ObjectPAL Editor is always in insert mode, so as you type, characters are pushed to the right. You cannot overtype characters. *Ctrl+C* or *Ctrl+Ins* copies selected text to the Clipboard, *Ctrl+X* or *Shift+Del* copies selected text to the Clipboard and deletes it from the window, and *Ctrl+V* or *Shift+Ins* pastes text from the Clipboard into your method.

Note The ObjectPAL Editor does not automatically wrap lines of text. A line extends to the right as you type until you press *Enter* to begin a new line.

Tab inserts an invisible tab character (ANSI 9) and pushes text to the right.

Selecting text

You can select a block of text by dragging with the mouse, using the arrow keys with *Shift* held down, or clicking with *Shift* held down to extend the selection.

Select a word by double-clicking it. Select an entire line by clicking to the left of the line. (The mouse is in position to do this when the insertion point changes to an arrow.)

When text is selected, typing a character (or pasting from the Clipboard) replaces the selected text with whatever you type or paste.

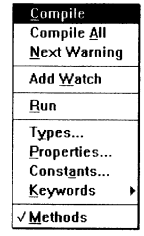
Double-clicking to the left of a line toggles a breakpoint and selects the line. See the next chapter for more information about breakpoints.

Editor Shortcuts

The ObjectPAL Editor provides Toolbar buttons and support for the right mouse button as shortcuts for common actions.

The Editor pop-up menu

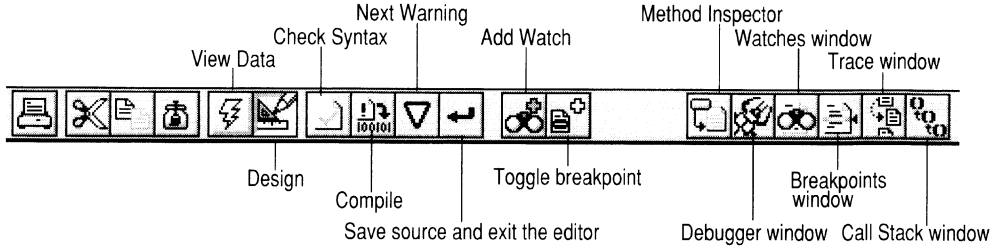
With an Editor window open, right-click anywhere in the Editor, or press *F6* or *Ctrl+M*, to display a pop-up menu. This menu contains commonly used items from the View, Search, Program, and Tools menus.



Toolbar buttons

Figure 8.10 shows the Editor Toolbar buttons.

Figure 8.10 The Editor Toolbar buttons



- *View Data* has the same effect as choosing Program | Run or View | View Data
- *Compile* has the same effect as choosing Program | Compile.
- *Save Source And Exit* has the same effect as choosing Close from the window's Control menu.
- *Add Watch* has the same effect as choosing Program | Add Watch.
- *Toggle Breakpoint* has the same effect as choosing Program | Toggle Breakpoint.

- The *Method Inspector, Debugger, Watches, Breakpoints, Trace,* and *Call Stack* buttons have the same effect as choosing these commands from the View menu.

Editor and Debugger keyboard shortcuts

Table 8.7 lists keyboard shortcuts you can use in the Editor and the Debugger.

Table 8.7 Editor and Debugger keyboard shortcuts

Function-key shortcuts	Keyboard shortcuts	Function
F1		Help
Ctrl+F1		Language Help
F2		Accept and save changes (to the form) without closing the Editor
Shift+F2		Save source and exit editor
	Ctrl+Z	Find
	Ctrl+Shift+Z	Find and replace
	Ctrl+A	Find again
F3	*Ctrl+I*	Inspect variable (Debugger)
Ctrl+F3	*Ctrl+B*	Toggle breakpoint
Shift+F3	*Ctrl+K*	Call stack (Debugger)
F5	*Ctrl+G*	Go to line
Ctrl+F5	*Ctrl+N*	Next warning
F6	*Ctrl+M*	Editor pop-up menu
F7		Step over (Debugger)
Shift+F7		Step into (Debugger)
F8		View data
Shift+F8		Save form to disk and run
Shift+F8		Run to cursor (Debugger)
Ctrl+F8		Run to end of method (Debugger)
F9	*Ctrl+Y*	Check syntax
Shift+F9	*Ctrl+R*	Compile all
	Ctrl+P	Open the Display Objects And Properties dialog box
	Ctrl+W	Add watch (Debugger)
Ctrl+F9	*Ctrl+D*	Deliver form
	Ctrl+C	Copy
	Ctrl+X	Cut
	Ctrl+V	Paste
	Ctrl+Spacebar	Open Method Inspector

Getting quick ObjectPAL help

In the IDE windows and dialog boxes listed below, you can select any element of the ObjectPAL language and press *F1* to get help for the selected word. (In the Editor, you can select a word or just place the insertion point in a word.)

- The Method Inspector
- The Editor window (if a word is not selected or the insertion point isn't in a word, *F1* displays help for the Editor)
- The Types And Methods dialog box
- The Display Objects And Properties dialog box
- The Constants dialog box

If there is only one Help topic for the language element, you are taken directly to that topic. If Help contains multiple topics for the selected word, a Search dialog box appears, listing the topics available for that language element. Select a topic and choose Go To.

Leaving the ObjectPAL Editor

You can exit the Editor in several ways. The one you choose at any given time will depend on what you want to do next.

- To continue designing your form or edit other methods,
  - Double-click the Editor window's Control menu (or choose Close from the Control menu).
  - Click the Save Source And Exit The Editor Toolbar button.
- To run your code immediately and see your form in action, click the View Data button.

Note Any Editor windows that are open when you run a form will open again upon returning to the design window.

Saving changes in the Editor

If the Prompt To Save option in the ObjectPAL Preferences dialog box is not checked, you are not prompted to save your changes when you close an Editor window or run a form with an Editor window open. All your changes are automatically saved. Choose Edit | Undo All Changes to discard changes before closing the Editor window. Changes you make in Editor windows are saved to disk when you save your form.

If the Prompt To Save option in the ObjectPAL Preferences dialog box is checked, a confirmation dialog box lets you save or cancel your changes when you close an Editor window or run a form with an Editor window open.

Using an alternate Editor

If you choose to use another Editor to write and edit methods, it must be able to save text-only files, without any formatting codes. Specify an alternate editor in the ObjectPAL Preferences dialog box (choose Properties | ObjectPAL). Enter the full path to your editor of choice, for example, C:\APPS\BRIEF\B.EXE.

Note ObjectPAL scripts can be edited only in the ObjectPAL Editor.

When you use an alternate editor, you don't have access to the syntax checker or other online information provided in the ObjectPAL Editor. When you open an item from the Method Inspector, Paradox is minimized and the specified editor is invoked. Close the editor to return to Paradox. To switch back to the ObjectPAL Editor, uncheck Alternate Editor in the Editor panel of the ObjectPAL Preferences dialog box.

The ObjectPAL Debugger

This chapter explains how to start the Debugger and work with it. At the end of the chapter, a sample debugging session illustrates debugging techniques.

Note This chapter assumes you're familiar with the ObjectPAL Editor, described in Chapter 8.

Bugs? Me?

It happens. Computers are notorious for doing what you tell them to do, not necessarily what you want them to do. To close this gap, you must rid your code of errors (*bugs*) that cause unexpected results. This process is called *debugging*.

The first step in debugging a method is recognizing that a bug exists. Sometimes it's obvious: the bug rears its ugly head the first time you run the application. Other bugs are insidious and might not surface until a method receives a certain value (often 0 or a negative number), or until you take a close look at the output and find that the results are off by a factor of .2 or the middle initials in a list of names are wrong.

Once you notice unexpected results, the next step is to find the bug and fix it. Normally, the instructions in a method execute much too quickly to follow, so it's hard to pinpoint just where things are going wrong. Using the Debugger, you can

- Set *breakpoints* so you can execute instructions up to a certain point, then stop and see what has happened.
- Open a *tracer* window that lists each line of code as it executes.
- *Inspect* or *watch* variables to make sure that values are being manipulated as you intended.
- Execute a method one line at a time (called *single-stepping*), or *step over* methods and procedures that you know are bug-free.
- List, and optionally view, the methods and procedures on the *call stack*; that is, those called since your form started running.

Using the Debugger

This section describes the Debugger environment and the various things you can do with the Debugger once execution is suspended at a breakpoint.

The Debugger environment

The debug environment includes the Debugger window, and the Watches, Breakpoints, Tracer, and Call Stack windows. You can position and size these windows to your liking and then choose Properties | Save Debug State.

You can set preferences for the Debugger in the ObjectPAL Preferences dialog box. For example, you can choose to have the Debugger open automatically when you're in design mode or when you're running a form. To set your ObjectPAL preferences, choose Properties | ObjectPAL. See Chapter 8 for more information about setting your ObjectPAL preferences.

Starting the Debugger

Although you can open the Debugger window before running code by clicking the Debugger Window button from the Editor Toolbar, you can use the Debugger only when execution is suspended at a breakpoint. To suspend execution and run code in the Debugger, you must either

- Set a breakpoint in the Editor before you run the code
- Use the DEBUG statement in your code and run it with Properties | Compile With Debug turned on
- Press *Ctrl+Break* while your code is running (if the Enable Ctrl+Break option in the ObjectPAL Preferences dialog box is checked, and the Properties | Compile With Debug menu option is turned on).

Once execution is suspended in the Debugger, Properties | Compile With Debug must be turned on if you want to step through, instead of stepping over, code in a form, library, or script called from the code in the Debugger.

To start the Debugger by using breakpoints,

1 Open the Editor window for the method or procedure you want to debug.

2 Set a breakpoint on the line or lines where you want to suspend execution.

You can set a breakpoint in the Editor or Debugger with any of the following methods:

- Move the insertion point to a line and click the Toggle Breakpoint Toolbar button, or choose Program | Toggle Breakpoint.
- Double-click to the left of a line (when the mouse pointer changes to an arrow).
- Move the mouse pointer over the left side of a line and *Ctrl*+click.

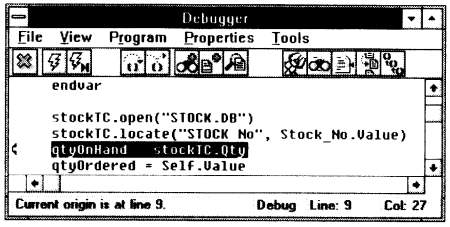
Note Breakpoints must be set at executable lines. You cannot set breakpoints on lines containing variable declaration statements or on the beginning and ending lines of method, procedure, uses, types, const, and case statements.

If you try to set a breakpoint at a non-executable line, it is set at the next available executable line. If an executable line cannot be found, the error message `illegal line number` appears.

3 Run your form.

When the breakpoint is reached, the debugger window opens and displays the method with the breakpoint. A method or built-in procedure executes until it reaches a breakpoint—the line containing the breakpoint does not execute. The gray triangle indicates which line will execute next.

Figure 9.1 The Debugger window



While execution is suspended at a breakpoint, the cursor turns into a stop sign whenever you move it over the form being debugged. The stop sign is a reminder that you are in debug mode for that form and can only proceed under the control of the Debugger.

The Debugger pop-up menu

Right-click anywhere in the Debugger window to display the Debugger pop-up menu. This menu contains commonly used items from the Program and View menus.

Executing code in the Debugger

When execution is suspended at a breakpoint, you can continue execution in debug mode with the following commands.

Run resumes execution from the breakpoint and continues to run until the next breakpoint is encountered. Choose Program | Run, click the Run button on the Debugger Toolbar, or press *F8*.

Run To Cursor continues execution of the method to the insertion point. Move the insertion point to the desired location in your code and choose Program | Run To Cursor, or press *Shift+F8*.

Shortcut Move the pointer over a line, right-click to display the pop-up menu, and select Run To Cursor.

Run To EndMethod continues execution to the end of the current method. Choose Program | Run To EndMethod, click the Run To EndMethod button on the Debugger

Toolbar, or press *Ctrl+F8*. The rest of the method executes and control returns to the Debugger.

Step Over single-steps through a method, treating procedures and custom methods as single steps. Choose Program | Step Over, click the Step Over button, or press *F7*.

Step Into single-steps through every line in a method and every line in the procedures and custom methods the method calls. Choose Program | Step Into, click the Step Into button, or press *Shift+F7*. You must turn on Compile With Debug on the Properties menu to step through code called from the Debugger.

Stop Execution halts execution in the Debugger. A dialog box appears stating that execution is stopped. Choose OK to return to view mode. Click the Design button to return to design mode.

Debugger closes the Debugger and returns control to the running form. You can also choose View | Debug.

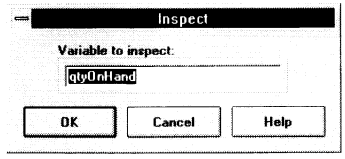
Viewing variables

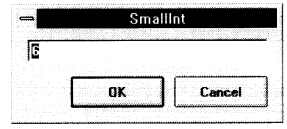
The Debugger provides two tools that let you view the value of a variable: *Inspect* lets you display and optionally change the value of a variable when execution is suspended at a breakpoint, while *Watch* lets you watch variables change as code executes.

Inspect

When execution is suspended at a breakpoint, choose Program | Inspect, click the Inspect button, press *Ctrl+I*, or right-click and choose Inspect. If the insertion point is next to or inside a variable, a dialog box opens displaying the current value of that variable. You can change the displayed value. Then choose OK or Cancel to close the Inspect dialog box and return to the Debugger.

Figure 9.2 The Inspect dialog boxes





If the insertion point is not in a variable, a dialog displays the variable nearest to the breakpoint. Either choose OK or type in the name of the variable you want to inspect. You can also drag the mouse to select an item to inspect.

Watch

Watching a variable is similar to inspecting a variable, except you can watch your variable values change as your code executes. You can watch simple types; more complex types, like arrays, must be inspected. To watch a variable,

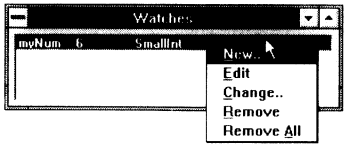
1 In the Debugger window, place the insertion point next to or inside a variable's name.

2 Choose Program | Add Watch, click the Watch button, or press *Ctrl+W*. The Watches window opens.

Using the Watches window

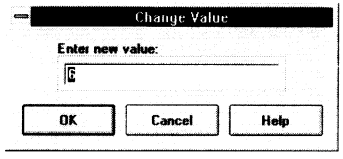
Inspect the Watches window to display its menu. From this menu you can add or remove variables you want to watch, edit the name of a variable being watched, or change the value of a watched variable.

Figure 9.3 The Watch window and its menu



You must use the menu to perform any of these actions. Typing anything in the Watches window opens the Change Value dialog box for the selected variable.

- To add a variable to watch, choose New and type the name of the variable you want to watch.
- To edit the name of a watched variable, choose Edit and change the name of the watched variable.
- To change the value of a variable, choose Change. The Change Value dialog box opens. Type a new value for the variable and press OK.

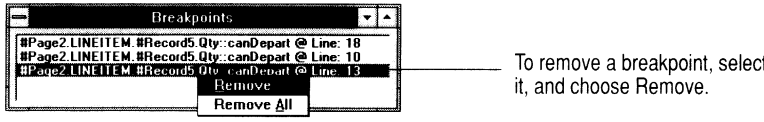


- To remove the selected variable from the Watches window, choose Remove.
- To remove all variables from the Watches window, choose Remove All.

The Breakpoints list

The Breakpoints window lists all the breakpoints in your code for the active form, library, or script. Choose View | Breakpoints, or click the Breakpoints Window button. Right-click in the window to see the Breakpoints window menu. Use the menu to remove one or all breakpoints.

Figure 9.4 The Breakpoints window



The Call Stack window

The Call Stack window lists the methods and procedures containing custom code that were called before the current breakpoint was reached. This list is referred to as the *call stack*. The most recently called routine is listed first, followed by its caller and so on, all the way back to the first method or procedure.

Choose View | Call Stack, or click the Call Stack button, to open the Call Stack window.

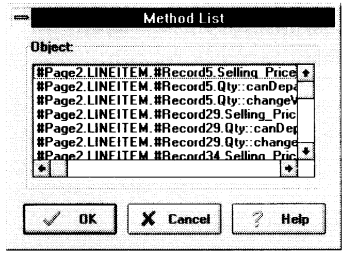
Figure 9.5 The Call Stack window

You can view any method or procedure on the call stack by double-clicking it or selecting it and choosing Inspect Context from the Call Stack window menu.

Navigating in the Debugger

The following describes how to navigate through code in the Debugger without executing the code.

- To view the method currently displayed in the Debugger window, scroll through it.
- To return to a method called earlier, open the Call Stack window, select a method or procedure, right-click, and choose Inspect Context.
- To search the form for a string or regular expression, choose File | Find or File | Find Next.
- To move the insertion point to a specified line number, choose File | Goto Line.
- To load any other of the form's methods that contains custom code, choose View | Source. This is a quick way to get to a specific method for a specific object.



Choose View|Source in the Debugger to see a list of a form's methods.

Select the method or procedure you want to load into the Debugger window and click OK.

- To return to the next line of code to be executed, choose Program | Origin. This displays the method or procedure containing the current breakpoint, and places the insertion point on the line containing the breakpoint. Use Origin to return quickly to

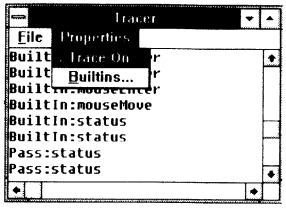
the breakpoint after you have navigated to other parts of your code or to other methods and procedures.

The Tracer

The ObjectPAL Tracer lists each ObjectPAL statement as it executes, providing a record of what happened and when. By default, the Tracer lists only methods and procedures that you have attached code to. However, you can choose to trace any or all of a form's built-in methods, whether or not they have ObjectPAL code attached.

To open the Tracer window, choose View | Tracer, or click the Trace Window button from the Debugger or Editor Toolbars.

Figure 9.6 The Tracer window



The Tracer's File menu provides the following options:

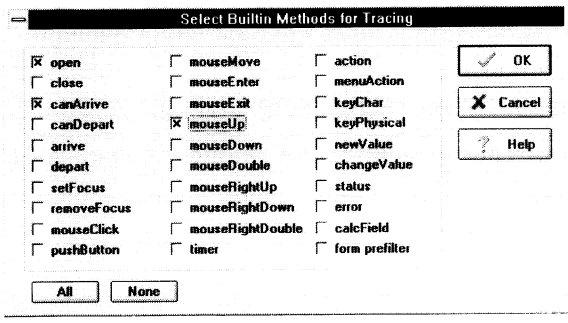
- *Save* or *Save As* saves the contents of the Tracer window.
- *Close* closes the Tracer window.
- *Print* prints the contents of the Tracer window.
- *Printer Setup* lets you select a printer or change other Windows printer settings.
- *Print Layout* lets you set the margins of the printout and choose whether to print the line numbers and page headers with the listing.
- *Clear* deletes all tracer output generated so far.

The Tracer's Properties menu provides the following options:

- *Trace On* turns tracer output off and on without closing the Tracer window. With Properties | Show Code turned on, tracer output consists of each line of code executed in all methods, procedures, and libraries. With Properties | Show Code turned off, tracer output consists only of messages output from **tracerWrite** statements `in code` and any built-in methods checked in the Builtin Methods For Tracing dialog box.
- *Builtins* displays a dialog box listing all the built-in methods (shown in Figure 9.7). Check a built-in method to display information about the method in the Tracer window as it executes.

Note Checking a built-in method indicates that you want that method traced; unchecked methods are not traced. It doesn't matter whether or not the method has ObjectPAL code attached to it—if you check it, it will be traced.

Figure 9.7 The Trace Builtins dialog box



In this figure, the built-in methods **open**, **canArrive**, and **mouseUp** are checked and will be traced.

You can choose All to check all built-in methods, or choose None to uncheck them all.

When the box labeled "form prefilter" is checked, methods are traced as they execute for the form and the intended target object; otherwise, methods are traced only for the target object. Refer to Chapter 10 for information about the ObjectPAL event model and the **isPreFilter** method.

- *Show Code* lets you control whether each line of code is listed in the Tracer as it executes. With Show Code turned off, tracer output consists only of messages output from **tracerWrite** statements and any builtin methods checked in the Builtin Methods For Tracing dialog box.

ObjectPAL Tracer procedures

ObjectPAL also provides procedures for controlling the ObjectPAL Tracer. These procedures include **tracerClear**, **tracerHide, tracerOff, tracerOn, tracerSave, tracerShow, tracerToTop,** and **tracerWrite**. Refer to the "System Type" topic in the online ObjectPAL Help for details and examples.

The Properties menu

The following options are available from the Debugger's Properties menu:

- *ObjectPAL* opens the ObjectPAL Preferences dialog box. See Chapter 8 for more information about the properties you can set in this dialog box.
- *Enable DEBUG* suspends execution whenever the DEBUG statement is encountered. Placing a DEBUG statement in a method has the same effect as setting a breakpoint at that line, with the advantage that it is saved with your source code. The setting for this item is saved with the form.

  In addition, when this item is checked, Paradox provides more detailed error information, so this item can be useful even if you never use a DEBUG statement.

Note Enable DEBUG only works when you run your code with the Compile With Debug option checked in the Editor's Properties menu, or with a breakpoint set. When this item is not checked, DEBUG statements are ignored and performance is improved.

- *Enable Ctrl+Break:* With this item checked, pressing *Ctrl+Break* suspends execution and opens the Debugger with the active method or procedure loaded, just as if a breakpoint had been encountered. This setting works only if Properties | Compile With Debug is turned on before you run your code. If Properties | Compile With Debug is turned off, pressing *Ctrl+Break* halts execution and lets you return to View mode or the design window.

Note *Ctrl+Break* halts execution only of ObjectPAL methods and procedures. Other operations (for example, queries) are not affected.

Note When breakpoints are set, when you trace execution, or when Enable DEBUG or Enable Ctrl+Break are turned on, execution speed is slower than normal.

- *Save Debug State* saves the Debugger environment in its current state. The Debugger environment includes the Debugger window, and the Watches, Breakpoints, Tracer, and Call Stack windows.

  Any of the windows you have open when you choose Save Debug State are automatically opened whenever you bring up the Debugger. The last size and position of these windows are also saved, whether or not they were open when the debug state was saved. The debug state, including minimized windows, is saved to PDOXWIN.INI and is reinstated each time you start Paradox.

A Debugger tutorial

This section provides a brief, hands-on tutorial for the Debugger. In it, you'll learn how to

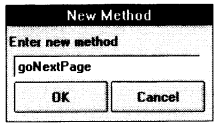
- Set breakpoints
- Inspect and watch variables
- Step through sections of your code
- Trace the execution of your code

Getting started

To begin the tutorial, start with a clean slate and create a form to use the Debugger.

1. Choose File | New | Form, and click Blank in the New Form dialog box to create an unbound form (not attached to any tables).
2. Inspect the page to display its menu, then name the page *myPage*.
3. Inspect *myPage* and, if the Method Inspector is not already open, choose Methods to open it.

4 Double-click New Method. The New Method dialog box appears.



5 Type `addOne`, then click the Check button.

An Editor window opens, containing the following default text:

```
method addOne()

endmethod
```

6 Edit the method so that it looks like this:

```
method addOne(var myNum SmallInt) SmallInt
return myNum + 1
endMethod
```

7 Close the Editor window. If prompted, save the changes.

8 Using the Button tool, place a small button in the upper left corner of the form. Name it *myButton*.

9 Inspect the button and choose Methods to display the Method Inspector.

10 Double-click **pushButton** to open an Editor window.

11 Type the following method:

```
method pushButton(var eventInfo Event)
var
   myNum, dumNum SmallInt
endVar
myNum = 5
myNum = addOne(myNum)
dumNum = 123
myNum.view()
endMethod
```

12 Click the Check Syntax button to check for syntax errors, and make corrections as needed.

13 Close the Editor window and save the new **pushButton** method.

Setting breakpoints

Breakpoints are useful for suspending execution of your code at a specific place so you can see what's happening. For example, if you have a complex method that isn't working and you don't want to trace through each line, set a breakpoint just before your method stops working. This saves time and typing.

This section shows how to set breakpoints.

1 Run the form you just created.

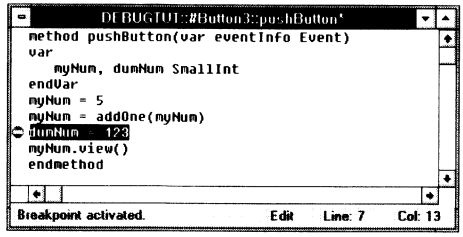
2 Click the button to execute the **pushButton** method. A dialog box appears, telling you that the value of *myNum* is 6. Choose OK.

3 Choose Form | Design to open the design window.

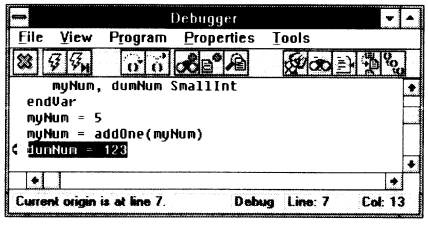
4 Display the **pushButton** method in an Editor window.

5 Use the mouse or the arrow keys to move the insertion point to line 7 of the method (*dumNum = 123*).

6 Click the Toggle Breakpoint button (or double-click to the left of the line when the insertion point changes to an arrow). The status bar indicates that a breakpoint is activated.



7 Run the form and click the button *myButton*. The **pushButton** method executes until it reaches the line with the breakpoint, and then the method is displayed in the Debugger window.



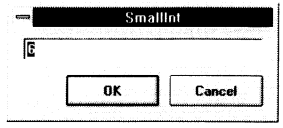
Inspecting a variable

When the Debugger encounters a breakpoint, you can inspect variables to see their values. This is useful for debugging code that is syntactically correct, but still not working properly.

To use your new form (and breakpoint) to inspect a variable,

1 Move the insertion point into the word *myNum* (any occurrence of *myNum* will do).

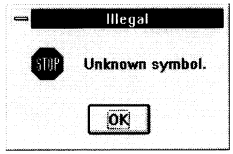
2 Click the Inspect button or press *Ctrl+I*. Another dialog box appears, showing the value of *myNum*. (In this case, it's 6.)



Notice that this value is highlighted, meaning you can change the value of *myNum* by typing a new value in the text box. For now, choose OK to leave it as it is.

3 Move the insertion point in any occurrence of the variable *dumNum*.

4 Again, click the Inspect button. A dialog box informs you that *dumNum* is an unknown symbol and cannot be inspected.



You can't inspect this variable because the variable *dumNum* has not yet been assigned a value at this point in the code. (A method executes *until* it reaches a breakpoint—the line containing the breakpoint does not execute.)

5 Choose OK.

Stepping through code and tracing execution

In this next exercise, you will set a breakpoint at the beginning of the **pushButton** method, and then continue execution by stepping through the code. You will also watch *myNum*'s value change in the Watches window as the form executes, then trace the execution with the Tracer window.

Start by deleting the existing breakpoint and setting a new one.

1 Toggle the breakpoint at line 7 off. (Place the insertion point on line 7 and click the Toggle Breakpoint button or press *Ctrl+B*.)

2 Place the insertion point on line 4 (*myNum* = 5), and set a breakpoint.

3 Click the View Data button to execute the form again.

4 Click the button *myButton*. Execution is suspended at the breakpoint and the Debugger window opens with the **pushButton** method displayed.

5 Place the insertion point in any occurrence of the *myNum* variable, and click the Add Watch button. The Watches window opens.

6 Click the Tracer Window button, or choose View | Tracer. The Tracer window opens.

Stepping into and over code

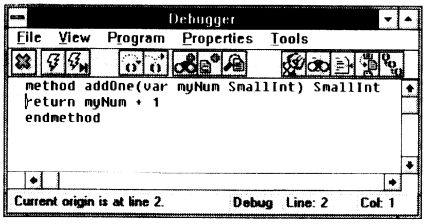
Stepping through each line of code lets you see exactly what's happening when the method runs. This can help you discover why a variable is being set to the wrong value.

For example, the **pushButton** method in the example calls custom method **addOne**. When you're debugging the **pushButton** method, you can choose to trace each command in **addOne** as it executes, or you can treat **addOne** as a single instruction and trace only the commands in the body of the **pushButton** method.

Now, step through the code in the **pushButton** method:

1 Click the Step Into button. The Tracer window lists each line of the **pushButton** method as it executes, and the Watches window shows that *myNum* has been assigned the value of 5. The gray triangle indicates that the following line, the call to **addOne**, will execute next.

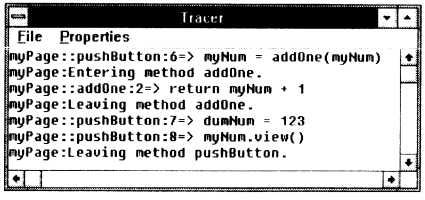
2 Click the Step Into button again. Because you are stepping through the code, the Tracer window indicates that **addOne** is being called. Resize the Tracer window so you can see all the code it lists. Notice that the Debugger now displays the **addOne** method.



3 Click the Step Into button two more times. **addOne** executes, as is reflected in the Tracer window. The value of *myNum*, as shown in the Watches window, has changed as a result of the calculation performed in **addOne**.

4 Click the Step Into button two more times. The form displays the value of *myNum* in a dialog box.

5 Type a new value and choose OK. The Watches window shows the value you assigned to *myNum*. The remaining lines of the **pushButton** method execute.



Now, go through this same procedure, except use Step Over instead of Step Into.

1 First, choose File | Clear in the Tracer window.

2 Now click the Step Over button to execute the **pushButton** method.

Notice that the Debugger now treats the line that calls the **addOne** procedure as a single step and doesn't display the **addOne** procedure. Also notice that the Tracer lists only the line of code that calls **addOne**, not the entire method.

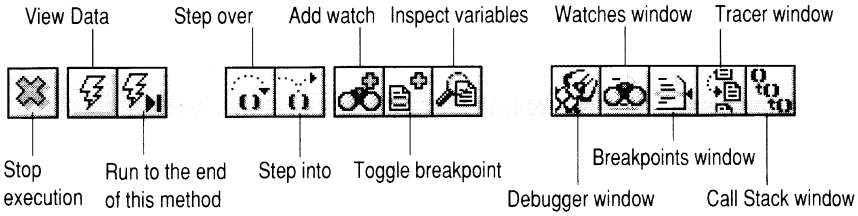
Whether you step into or over the code in the **pushButton** method, the same code executes.

Why would you want to step over sections of your code? Because once a method or procedure works properly, there's no need to debug it every time a method calls it. Stepping over well-tested procedures and methods lets you concentrate on debugging the method at hand.

Debugger shortcuts

The Debugger Toolbar provides shortcuts for commonly performed debug commands.

Figure 9.8 The Debugger Toolbar buttons



- Stop *Execution* has the same effect as choosing Program | Stop Execution.
- *View* has the same effect as choosing Program | Run.
- *Run To EndMethod* has the same effect as choosing Program | Run To EndMethod.
- *Step Over* and *Step Into* have the same effect as choosing Program | Step Over and Program | Step Into.
- *Add Watch* has the same effect as choosing Program | Add Watch.
- *Toggle Breakpoint* has the same effect as choosing Program | Toggle Breakpoint.
- *Inspect* has the same effect as choosing Program | Inspect.
- *Debugger Window* has the same effect as choosing File | Close or double-clicking the Debugger window's Control menu.
- The *Watches*, *Breakpoints*, *Trace*, and *Call Stack Window* buttons have the same effect as choosing these commands from the View menu. These windows remain open in design mode until you close them.

Part

Events, actions, and built-in methods

Every object in a form (as well as the form itself) includes built-in methods, or triggers, that execute in response to events. Paradox interprets the event and calls the appropriate built-in method attached to the target object. Understanding this event model and the default behavior of ObjectPAL's built-in methods is key to understanding ObjectPAL.

This part of the manual contains these chapters:

- Chapter 10, "Understanding the event model," describes the ObjectPAL event model (the rules Paradox uses to process events), and presents objects that handle events—events generated by the user through the user interface, and events generated by ObjectPAL.
- Chapter 11, "Actions and UIObjects," explains how to use ObjectPAL to make UIObjects initiate and respond to actions. It presents some general information about actions.
- Chapter 12, "Default behavior of built-in methods," presents ObjectPAL's built-in methods. Built-in methods specify how objects respond to events.

10

Understanding the event model

When you interact with an ObjectPAL application, you generate events. This chapter presents the object types that handle events, whether the events are generated by the user through the user interface or generated by ObjectPAL. It discusses ObjectPAL's event model, the methods common to all event types, and each event type individually. The ObjectPAL event types are listed in Table 10.1.

Table 10.1 Events

Type	Description
ActionEvent	Information about basic activities
ErrorEvent	Information about errors
Event	Information about events in general
KeyEvent	Information about keyboard events
MenuEvent	Information about user interactions with a menu
MouseEvent	Information about the mouse and mouse actions
MoveEvent	Information about moving the pointer between objects
StatusEvent	Information about messages that display in the status line
TimerEvent	Information about events generated at specified intervals
ValueEvent	Information about changes to a field value

This section describes the event model and discusses

- Events: A first look
- Internal and external events
- The event packet: *eventInfo*
- Methods common to all event types

Events: A first look

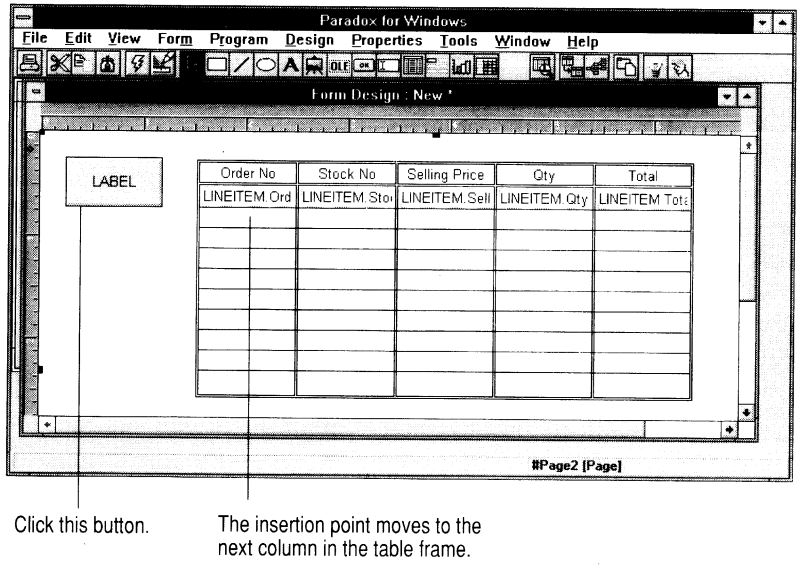
The following example introduces the ObjectPAL *event model*, the rules Paradox uses to process events. The ObjectPAL event model is powerful and gives a programmer great flexibility. Understanding it is a key to getting the most out of ObjectPAL.

As previously stated, you build Paradox applications by placing objects in a form and writing ObjectPAL code to define how the objects respond to events. When you use the application, you interact with these objects and generate events, and the code executes. A single user action triggers a chain reaction of events and built-in methods that execute by default; by attaching your own code to the appropriate built-in methods of the appropriate objects, you can specify precisely when and how objects respond.

Designing the form

The form you'll create in this example contains a button and a table frame. (Table frames are explained in the *User's Guide*.) You'll place these objects in the form, then attach code to the button so that when you click the button, the insertion point moves to the next column in the table frame. Figure 10.1 shows the completed form. The steps for creating it and attaching code follow.

Figure 10.1 The completed form



1 To begin, choose File | New | Form from the Desktop, and click the Data Model/ Layout Diagram in the New Form dialog box.

2 Select LINEITEM.DB in the Data Model dialog box and choose OK.

3 In the Design Layout dialog box, choose the Tabular style and choose OK.

4 Select the table frame and move it to the right, and then use the Button tool to create a button in the upper left corner of the form. Compare your form to Figure 10.1.

5 Name the table frame *TFrame*. To name an object, inspect it and choose the object's name from its menu. A dialog box appears. Type the new name in the dialog box.

Attaching code

Now that the form design is complete, the next step is to attach code to the button so that the code executes when you push the button. When you create your own applications, you can create objects and attach code in any order you like.

1 Inspect the button.

2 Choose Methods.

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

3 Edit the **pushButton** method as shown here.

```
method pushButton(var eventInfo Event)
   TFrame.action(MoveRight)
endMethod
```

4 Close the Editor window.

5 Choose View | View Data.

6 Click the button a few times. The insertion point (highlight) in the table frame moves to the right each time, until it reaches the rightmost field in the table frame.

How it works

In this form, when you click a button, code attached to the button executes, and the insertion point moves in the table frame. Figure 10.2 shows what happens.

Figure 10.2 A chain of events and methods

Every object in a form, including the form itself, has built-in methods that execute in response to events. The code executes by default—you don't have to do anything to make it happen. In this example, the default code for the table frame's built-in **action** method moves the insertion point.

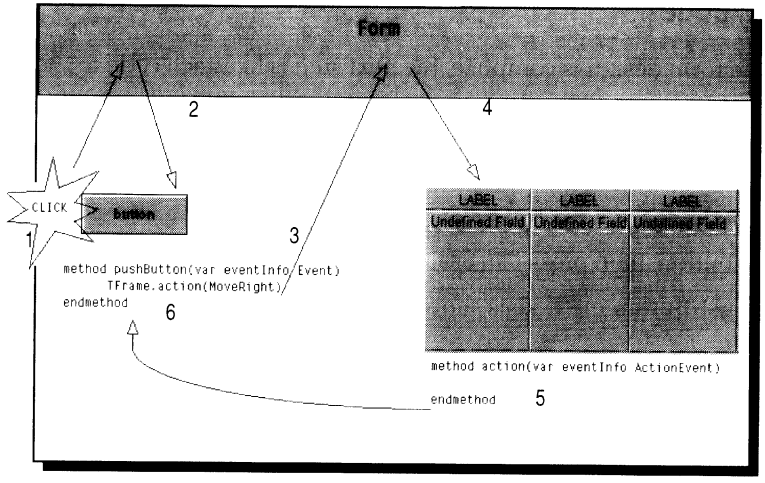


Figure 10.2 diagrams the chain of events described in the numbered steps listed here.

1 First, you click the mouse, which generates an event. Every event goes to the form first. The form interprets the event and decides what to do with it.

2 Because you clicked the mouse when the pointer was over the button, the form's built-in **pushButton** method executes, and by default calls the button's built-in **pushButton** method.

3 Next, the button's **pushButton** method executes. The button appears pushed in. When the statement you attached executes, it generates another event, and that event goes to the form.

```
TFrame.action(MoveRight)
```

4 The form interprets this event. The form's built-in **action** method executes, and calls the built-in **action** method of the object named *TFrame*.

5 The default code for *TFrame*'s built-in **action** method executes, even though you didn't attach any code to it. The insertion point moves to the right.

6 Finally, the default code for the button's built-in **pushButton** method finishes execution, and the button appears to pop out. Processing for this event is complete.

Figure 10.2 presents a very simple overview—the rest of this chapter describes the model in detail.

Internal and external events

ObjectPAL recognizes two kinds of events: internal and external. Internal events are triggered from within Paradox. Examples of internal events include opening and closing an object, arriving at and departing from an object, and timer events. External events are triggered by the user (or from within an ObjectPAL method that simulates a user action). Examples of external events include keypresses, mouse clicks, and menu choices. Tables 10.2 and 10.3 list the built-in methods that respond to internal and external events.

Every design object has built-in default methods that respond to each event. ObjectPAL provides the following keywords to control when (or whether) the default code executes: doDefault, disableDefault, enableDefault, and passEvent. For information about built-in methods, default code, and keywords, see Chapter 12 and the "Basic language elements" topic in the online ObjectPAL Help.

Table 10.2 Built-in methods for internal events

Method	Description
open	Executes for each object when the form runs
close	Executes for each object when the form closes
canArrive	Asks for permission to move to an object
canDepart	Asks for permission to move off an object
arrive	Executes when you move to an object
depart	Executes when you move off an object
setFocus	Executes when an object is ready to receive keyboard input
removeFocus	Executes when an object loses focus
timer	Executes each time a specified timer interval elapses
mouseEnter	Executes when the pointer moves inside the boundaries of an object
mouseExit	Executes when the pointer moves outside the boundaries of an object
pushButton[1]	Executes when you click a button object
newValue[2]	Executes to report that a field object has a new value
changeValue[2]	Executes to post changes to a field value

1. Buttons only
2. Field objects only

Table 10.3 Built-in methods for external events

Method	Description
mouseMove	Executes when the mouse moves
mouseDown	Executes when the left mouse button is pressed
mouseUp	Executes when the left mouse button is released
mouseClick	Executes when the left mouse button is pressed and released while the pointer is inside the boundaries of an object
mouseDouble	Executes when the left mouse button is double-clicked
mouseRightDown	Executes when the right mouse button is pressed
mouseRightUp	Executes when the right mouse button is released

Table 10.3 Built-in methods for external events (continued)

Method	Description
mouseRightDouble	Executes when the right mouse button is double-clicked
keyPhysical	Executes when a key is pressed
keyChar	Executes for any keypress which cannot be mapped into an action event
action	Executes for each of the basic functions on the form (moving, locking records, posting records, etc.) that map to an action
menuAction	Executes when you choose an item from a menu or click a Toolbar button that corresponds to a menu choice
error	Executes when ObjectPAL encounters an error condition
status	Executes when a message is displayed in the status bar

How Paradox handles events

Every event, whether internal or external, goes to the form first. In ObjectPAL terms, the form *filters* every event.

If it's an internal event, Paradox knows which object is the target, so by default the form sends the event packet *eventInfo* (described in detail later in this chapter) to that object. The event packet triggers the appropriate built-in method.

In addition, for external events, Paradox uses a mechanism called *bubbling* to pass these events from object to object in the containership hierarchy. When an object receives an external event that isn't meaningful (for example, a keystroke isn't meaningful to an ellipse), it passes the event information to its container. The information rises, like a bubble in liquid, until an object handles it or it reaches the highest level, the form. So, it's possible for the form to see the same external event twice, as described in Figure 10.3.

Figure 10.3 External events bubble

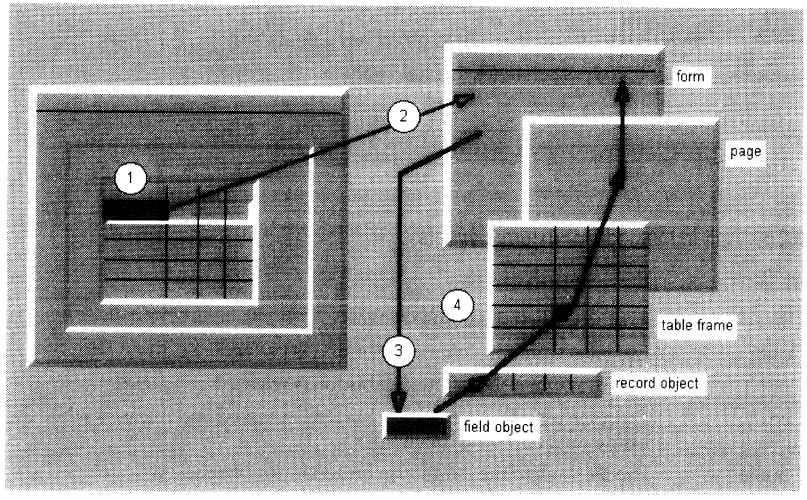


Figure 10.3 diagrams the flow of events that occurs when you type a character or click in a field object.

1 First, click in the field or type a character into the field object.

2 The event goes to the form for the first time. The form knows which object is the target, and acts as the dispatcher for the event.

3 The form dispatches the event to the field object, calling the appropriate built-in method.

4 Then, the event can bubble up the containership hierarchy (field, record, table frame, page, form). As the event bubbles up, *self* changes, but the target is always the field.

The form filters every event, examines it, then dispatches it to the intended target object. From there the event bubbles from object to object in the containership hierarchy until it reaches the form again. For example, as shown in Figure 10.3, when you type characters into a field object, that field object is your *intended* target, but in fact, the event goes to the form first. Then, when the form dispatches the event to the field object, the field object is the actual target, and remains the target until the processing of that event is complete.

The event packet: *eventInfo*

Every event generates a packet of information about itself, and every built-in method has a parameter *eventInfo* that stores the event packet. The following code is an example:

```
method mouseEnter (var eventInfo MouseEvent)
; body of the method goes here
endMethod
```

As the example shows, a method's default declaration also declares the type of *eventInfo* (in the example, *eventInfo* is a MouseEvent). This declaration specifies which methods you can use with *eventInfo* to get information out of the packet and to put information in. If *eventInfo* is a mouse event, use MouseEvent methods; if it's a key event, use KeyEvent methods, and so on. Each *eventInfo* type is an ObjectPAL type, and you can see a list of their methods in the Types and Methods dialog box by choosing Tools | Types.

You can use these methods with *eventInfo* to extract data from the event packet. For example, given a MouseEvent, you can use the following statement in, for example, the **mouseDown** method, to get the x- and y-coordinates of the mouse at the time of the event. (*mouseX* and *mouseY* must first be declared as LongInt).

```
eventInfo.getMousePosition(mouseX, mouseY)
```

Given a KeyEvent, you can use the next statement to find out which character was pressed.

```
eventInfo.char()
```

You can also use methods to set information in the packet. For example, in the following statements in the **keyChar** method, if information in the event packet says "a" was pressed, it's replaced with information saying "z" was pressed.

```
if eventInfo.char() = "a" then
   eventInfo.setChar("z")
endIf
```

As another example, the following method shows how to use the event packet to do some simple value checking.

```
; this method is attached to an unbound field
method changeValue(var eventInfo ValueEvent)
   var
      atNewVal AnyType
   endVar
   atNewVal = eventInfo.newValue()
   if atNewVal > 50 then
      eventInfo.setErrorCode(CanNotDepart) ; CanNotDepart is a nonzero constant
   endIf
endMethod
```

This code prevents the insertion point from leaving the field if it contains a value greater than 50; otherwise, it lets the built-in field validation determine whether the value is legal.

Note The argument *CanNotDepart* is a constant defined by ObjectPAL for use with **setErrorCode**. By definition, it has a nonzero value. Constants are listed online. To display the list, open an ObjectPAL Editor window, choose Tools | Constants, then from the Types of Constants column, choose EventErrorCode. The constants appear in the Constants column.

Using ObjectPAL to handle events

Because every event goes to the form first, the form has a chance to handle an event before it reaches the intended target, and in the case of external events, after it reaches the intended target as well. When you attach code at the form level, it's important to know whether the form is handling an event on its own behalf or filtering the event before passing it to another object. ObjectPAL provides a method (**isPreFilter**) that answers both questions at once.

Using isPreFilter

The method **isPreFilter**, defined for each of the event types, answers two questions:

- Is the form seeing this event for the first time?
- Is the form the target?

isPreFilter is such an important method that it's included in the default text in the Editor window for every form-level, built-in method. For example, the first time you attach code to the form's built-in **open** method, the Editor window contains the following code:

```
method open(var eventInfo Event)
if eventInfo.isPreFilter()
   then
      ;// This code executes for each object on the form:
   else
      ;// This code executes only for the form:
```

```
endIf
endMethod
```

This default code is inserted into form-level, built-in methods as a convenience. We recommend that you use it, but you can delete it, if necessary. (If you delete it, heed the warning at the end of this section.)

isPreFilter returns True in the following cases:

- When the target is some object other than the form, and the form has not already handled this event
- For all internal events
- For external events when they first reach the form

isPreFilter returns False when the external events bubble back to the form.

To build a form that demonstrates the effects of **isPreFilter**, do the following:

1 Create a blank, unbound form.

2 Place two boxes anywhere in the form.

3 Edit the form's built-in **open** method as follows:

```
method open(var eventInfo Event)
if eventInfo.isPreFilter()
   then
   beep()    ;// This code executes for each object on the form:
   sleep(500)
   else
             ;// This code executes only for the form:
endIf
endMethod
```

4 Run the form. Your system should beep three times: once for each box, and once for the underlying page (but *not* for the form).

5 Go back to the Form Design window and edit the form's built-in **open** method as follows:

```
method open(var eventInfo Event)
if eventInfo.isPreFilter()
   then
             ;// This code executes for each object on the form:
   else
   beep()    ;// This code executes only for the form:

endIf
endMethod
```

6 Run the form again. Your system should beep only once, when the form itself opens.

Using getTarget

Use the **getTarget** method, defined for all event types, to find out which object is the intended target of an event. The form knows which object is the intended target (in most cases, it's the active object); that's how it can dispatch the event appropriately. (The

target of an event remains constant in the event packet, regardless of bubbling.) For example, the following statements, attached to a form's built-in **keyChar** method, report the name and the type of the target object, regardless of how many objects the form contains or which object is the target:

```
method keyChar(var eventInfo KeyEvent)
   var
      theTarget UIObject
   endVar

if eventInfo.isPreFilter()
   then
      ;// This code executes for each object on the form:
      eventInfo.getTarget(theTarget)
      msgInfo("Target type:", theTarget.class)
      msgInfo("Target name:", theTarget.name)
   else
      ;// This code executes only for the form:
endIf
endMethod
```

Summary

The following list summarizes the information returned by **isFirstTime**, **isTargetSelf**, and **isPreFilter**:

- **isFirstTime**: Is the form seeing the event for the first time (before dispatching it to the target object)?
- **isTargetSelf**: Is the form the target of the event?
- **isPreFilter**: Is the form seeing the event for the first time *and* is some other object the target?

Using getObjectHit

Mouse events are different from other events because the pointer can roam freely all over the screen, and the intended target is not necessarily the active object. For mouse events, the method **getObjectHit** reports which object is the intended target.

The following example is attached to the built-in **mouseEnter** method for a form. When the pointer enters an object, this method determines if it is a field object. If so, this method displays the field's name in the status bar; if not, no message is displayed.

```
method mouseEnter (var eventInfo MouseEvent)
   var
      theTarget UIObject
   endVar
if eventInfo.isPreFilter()
   then
        ; code here executes for every object on the form
        eventInfo.getObjectHit(theTarget) ; Which object was hit by the mouse?
        if theTarget.class = "Field" then  ; If it's a field object,
           message(theTarget.name)         ; display its name.
        endIf
   else
```

```
        ; code here executes just for the form itself
endIf
endMethod
```

Deleting default code

If you choose to delete the default code in a form-level, built-in method, be careful. Remember: *Every event goes to the form first, and many come back to the form a second time!*

Warning When handling events at the form level, make sure you understand which object is the target and when you want the object to execute. For example, suppose you want to display a message when you open a particular form. You *could* attach the following code to the form's built-in **open** method, and the message *would* display when the form opens. For example,

Don't do this!

```
method open(var eventInfo Event)
   msgInfo("Hello", "Welcome!")
endMethod
```

But the message would also display for *each* **open** method in *each* object in the form! Here's why: when the form opens, the built-in **open** method for each object in the form executes. Each time, the event goes to the form first, and the default code for the form's built-in **open** method dispatches the event to the **open** method of the target object. In this example the ObjectPAL statements execute before the built-in code, so the message displays once for each object in the form.

The correct approach is to test for **isPreFilter** and place code in the **else** clause, as follows:

Do this instead!

```
method open(var eventInfo Event)

   if eventInfo.isPreFilter() then
      ; code here executes for every object in the form
   else
      ; code here executes just for the form itself
      msgInfo("Hello", "Welcome!")
   endIf
endMethod
```

In this example, the **msgInfo** statement executes only once: when the form is executing the **open** method on its own behalf.

Event: The base event type

Event is the base event type. It gets all events not claimed by other event types. Each event has built-in methods. Some of these built-in methods are common to all event types. The next section describes common methods and how to use each method with different events.

Methods common to all event types

Methods common to all event types include:

- **errorCode** reports about the status of the error flag.
- **getTarget** reports which object received the event.
- **reason** reports why an event happened.
- **setErrorCode** sets the error flag.
- **setReason** specifies a reason for an event.

For a complete list of the methods for each event type, refer to the online ObjectPAL Help.

Using setErrorCode and errorCode

Use **setErrorCode** with *eventInfo* to specify the status of an event. For example,

```
method canArrive(var eventInfo MoveEvent)
   eventInfo.setErrorCode(CanNotArrive) ; CanNotArrive is an ObjectPAL constant
endMethod
```

By attaching this method to a field, you can prevent a user from moving the pointer into the field. The constant CanNotArrive has been assigned a nonzero value by ObjectPAL, and any nonzero error value blocks this method's built-in code from executing.

Note For more information about blocking the default code for a built-in method, refer to Chapter 12.

As a rule, when ObjectPAL provides a method for setting information, it provides a method for getting information. That's what **errorCode** is for: it reports about the status of an event. In most cases, you'll know the status already: either it will have the value you set, as in the above example, or it will be 0.

Note The **errorCode** method for the event types is in no way related to the System type **errorCode** procedure or to the **try...onFail...endTry** structure (discussed in Chapter 30).

Using getTarget

Use **getTarget** (provided for all event types) to find out which object received the current event. Using **getTarget** and a **switch...endSwitch** structure, you can create generalized event-handling routines. For example, suppose a form contains two buttons, one for opening an existing file, and one for creating a new file. You could write a **pushButton** method for each button, but this becomes less practical as the number of buttons increases. Instead, create a custom method that responds appropriately no matter which button the user clicks. For example,

```
method doWhat(var eventInfo Event) ; a custom method attached to the form
   var obj UIObject endVar
   eventInfo.getTarget(obj)
   switch
      case obj.name = "increase"  : Qty.value = Qty.value + 1
      case obj.name = "decrease"  : Qty.value = Qty.value - 1
   endSwitch
endMethod
```

You can easily add statements to the **switch...endSwitch** structure as you add more buttons to the form. Next, add a call to the custom method **doWhat** to each button's **pushButton** method.

```
method pushButton(var eventInfo Event)
   doWhat(eventInfo) ; remember to pass eventInfo to the custom method
endMethod
```

In this example, *eventInfo* is passed as an argument to the custom method **doWhat** because the event packet contains information about which object was the target of the event. **doWhat** needs this information to make a decision in the **switch...endSwitch** block.

Using reason

If a built-in method is triggered by an Event, ErrorEvent, MenuEvent, MoveEvent, or a StatusEvent, you can use **reason** to find out why. For example, a field object's **depart** method could be called because the user moved the pointer out of the field object, because of an ObjectPAL statement, or because the application was closed. Using **reason** and ObjectPAL constants, you can specify which, if any, of these conditions to respond to. Table 10.4 lists the constants.

Table 10.4 Reason constants

Method	Constant	Description
Event methods: **newValue**	FieldValue	Value is different by scrolling, by a refresh across a network, by an ObjectPAL statement, or by a user changing the value of a field.
	EditValue	Value was specified by pressing a radio button or choosing an item from a list.
	StartupValue	The form opened.
ErrorEvent methods: **error**	ErrorCritical	Will display a message in a dialog box.
	ErrorWarning	Will display a message in the status area.
MenuEvent methods: **menuAction**	MenuNormal	Toolbar buttons and custom ObjectPAL menus.
	MenuControl	Control menu, Minimize and Maximize buttons.
	MenuDesktop	Paradox built-in menus (e.g., File, Window, Help).
MoveEvent methods: **arrive**, **depart**, **canArrive**, **canDepart**	UserMove	User moved the insertion point to a different object (using mouse or keyboard).
	StartUpMove	The form opened.
	ShutDownMove	The form closed.
	RefreshMove	Move is needed because a record was inserted, deleted, or moved, or refresh occurred due to another user changing data you see on screen.
	PalMove	Triggered by an ObjectPAL statement.
StatusEvent methods: **status**	ModeWindow1	Message sent to the leftmost small area of the status bar.
	ModeWindow2	Message sent to the middle small area of the status bar.

Table 10.4 Reason constants (continued)

Method	Constant	Description
	ModeWindow3	Message sent to the rightmost small area of the status bar.
	StatusWindow	Message sent to the Status area (large area) of the status bar.

Using reason with MoveEvents

Following is an example using **reason** with **depart**.

```
method depart(var eventInfo MoveEvent)
   if eventInfo.reason() = UserMove then
      doSomething() ; do some custom method
   endIf
endMethod
```

In the previous example, the custom method **doSomething** executes only when **depart** is called because of a user action; otherwise, the event is handled normally.

Using reason with StatusEvents

Another set of reasons is associated with the built-in **status** method. Any time a message is sent (by an ObjectPAL method or by Paradox) to one of the four message areas in the status line, a **status** method is triggered. The reasons specify which area the message appears in. Using **status** and the associated reasons, you can intercept any message, change it or replace it, send it to any of the three message areas, assign it to a variable and display it elsewhere, or block it.

```
method status(var eventInfo StatusEvent)
   if eventInfo.reason() = ModeWindow1 then
      eventInfo.setReason(StatusWindow) ; redirect message to the status area
   endIf
endMethod
```

For more information about working with the status line, see the "StatusEvent: Controlling the status bar" section later in this chapter.

Using reason with ErrorEvents

The reasons associated with **error** report whether the error is a critical error or a warning. Every design object in a form (and the form itself) has a built-in **error** method. For every design object except the form, the default behavior is to pass the error to its container. If an error bubbles up to the form, the form's built-in **error** method does one of two things:

- If it's a critical error, **error** displays a message in a dialog box.
- If it's a warning, **error** displays a message in the status line, which triggers normal StatusEvent processing.

But, using **reason**, you can change this default behavior. For example, suppose you want to display a dialog box for every error, critical or not. You could modify the form's **error** method like this:

```
method error(var eventInfo ErrorEvent)
   if eventInfo.reason() = ErrorWarning then
      eventInfo.setReason(ErrorCritical)
   endIf
endMethod
```

As an error bubbles up, any object in the containership hierarchy can intercept it, so two or more objects could respond differently to the same error.

For more information about error handling, see Chapter 30.

Using reason with Events

The reasons associated with the built-in **newValue** method report why the method was triggered. For example, when you press a radio button to specify a value and then move off the field, you trigger **newValue** twice, each time for a different reason: the first time, the reason is EditValue; the second time, it's FieldValue. For more information about using **newValue**, refer to the "ValueEvent: Handling field value changes" section in this chapter, and to Chapter 12.

ActionEvent: Handling table editing and navigation

ActionEvents are generated primarily by editing and navigating in a table. Typically, when you work with ActionEvents, you'll also work with ObjectPAL's action constants. For example, to prevent users from editing a table frame, you could do something like this:

```
; this overrides a table frame's built-in action method
method action(var eventInfo ActionEvent)
   ; if the user tries to switch to Edit mode, display a dialog box
   if eventInfo.id() = DataBeginEdit then   ; DataBeginEdit is a constant
      msgStop("Stop", "You can't edit this form.")
      disableDefault
   else
      enableDefault ; otherwise, behave normally
   endIf
endMethod
```

The methods **id** and **setId** use action constants (like DataBeginEdit) to get and set information about the action. Action constants are listed online. To display the list, open an ObjectPAL Editor window and choose Tools | Constants. Then, from the Types of Constants column, choose an item beginning with Action, for example, ActionDataCommands. The constants display in the Constants column.

User-defined action constants

You can also define your own action constants, as long as you keep them within a specific range. Because this range is subject to change in future versions of Paradox, ObjectPAL provides the constants UserAction and UserActionMax to represent the minimum and maximum values allowed.

For example, suppose you want to define two action constants, ThisAction and ThatAction. In a Const window, define values for your custom constants as follows:

```
Const
   ThisAction = 1
   ThatAction = 2
EndConst
```

Then, to use one of these constants, add it to UserAction:

```
method action(Var eventInfo ActionEvent)
   if eventInfo.id() = ThisAction + UserAction then
      doSomething()
   endIf
endMethod
```

By adding UserAction to your own constant, you guarantee yourself a value above the minimum. To keep the value under the maximum, you'll have to check the value of UserActionMax. One way is to use a **message** statement:

```
message(UserActionMax)
```

In this version of Paradox the difference between UserAction and UserActionMax is 2047. That means the largest value you can use for an action constant is UserAction + 2047.

Important Methods for the ActionEvent type are closely related to the built-in **action** method. For more information, see Chapter 13 and Chapter 12.

ErrorEvent: Information about errors

The ErrorEvent type provides methods you can use to get and set information for processing by the **error** method built into every design object (UIObject type). Like other events, an error goes to the form first, which then dispatches it to the intended target object (which is typically the object whose code triggered the error). With this model you can create a central error-handling routine and attach it to the form. For more information about errors and error handling, see Chapter 30.

User-defined error constants

You can also define your own error constants, as long as you keep them within a specific range. Because this range is subject to change in future versions of Paradox, ObjectPAL provides the constants UserError and UserErrorMax to represent the minimum and maximum values allowed.

For example, suppose you want to define two error constants, ThisError and ThatError. In a Const window, define values for your custom constants as follows:

```
Const
   ThisError = 1
   ThatError = 2
EndConst
```

Then, to use one of these constants, add it to UserError:

```
method error(Var eventInfo ErrorEvent)
   if eventInfo.id() = ThisError + UserError then
      doSomething()
```

```
    endIf
endMethod
```

By adding UserError to your own constant, you guarantee yourself a value above the minimum. To keep the value under the maximum, you'll have to check the value of UserErrorMax. One way is to use a **message** statement:

```
message(UserErrorMax)
```

In this version of Paradox the difference between UserError and UserErrorMax is 2046. That means the largest value you can use for an error constant is UserError + 2046. (The error code 0 is reserved to mean "no error.")

KeyEvent: Handling keyboard actions

The KeyEvent type provides methods for getting and setting information about keystroke events, including

- Characters sent to the program: **char, charAnsiCode, vChar, vCharAnsiCode, setChar, setVChar**
- Status of *Alt, Ctrl,* and *Shift*: **isAltKeyDown, setAltKeyDown, isControlKeyDown, setControlKeyDown, isShiftKeyDown, setShiftKeyDown.**

The following code uses the KeyEvent method **char** in the built-in **keyChar** method so a pop-up menu appears when you type ?. For example, if this code were attached to a field object, the menu would pop up when you typed ? into it.

```
method keyChar (var eventInfo KeyEvent)
   var
      pop PopUpMenu
   endVar

   if eventInfo.char() = "?" then    ; if ? is typed
      disableDefault
      pop.addText("one")             ; build pop-up menu
      pop.addText("two")
      pop.addText("three")
      self = pop.show()
   endIf                             ; otherwise, behave normally
endMethod
```

Note You can't trap certain key combinations because they're processed by Windows before they reach Paradox, as explained in the next section. The key combinations you can't trap include *Ctrl+Break, Ctrl+Alt+Del, F10, Ctrl+Esc,* and *Ctrl+F4.*

Keyboard events and built-in methods

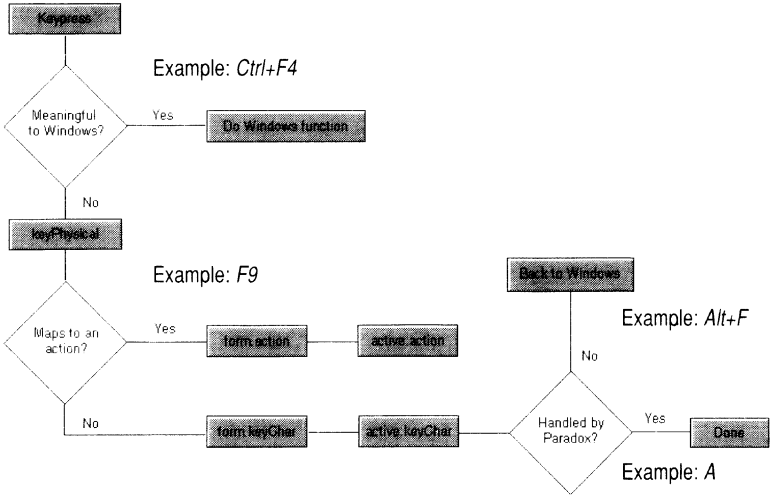
When you press a key in Paradox, one of the following happens:

- Windows intercepts the keystroke and does something.
- Windows passes the keystroke to Paradox, which does something.

- Windows passes the keystroke to Paradox, which passes it to the active object.

When Windows intercepts a keystroke, ObjectPAL does not see it. In the other two cases, you can use the built-in methods **keyPhysical**, **action**, and **keyChar**, to respond to and simulate keyboard events. These built-in methods are closely related, as shown in Figure 10.4.

Figure 10.4 Flow of events and methods for a keypress



When a key is pressed, Windows gets the keystroke from the keyboard driver and stores it in the system message queue. If the keystroke is meaningful to Windows (for example, *Ctrl+F4*, which closes the active window), Windows performs the associated action. Otherwise, Windows sends the keystroke to Paradox. Paradox generates a KeyEvent packet and sends it to the form's **keyPhysical** method. If the keystroke corresponds to a Paradox action (for example, *F9*, which toggles Edit mode on and off), Paradox calls the form's built-in **action** method with the appropriate constant (such as DataToggleEdit).

If the keystroke is not intercepted by Windows or translated to an action by Paradox, the form's **keyPhysical** method passes the event packet to its **keyChar** method, which by default passes it to the **keyChar** method of the active object. The active object handles the keystroke or bubbles it to its container, and so on, up through the containership hierarchy to the form. If the keystroke is a menu shortcut (for example, *Alt+F*, which displays the File menu), the form passes it back to Windows for processing.

Note This model describes keystroke processing within Paradox only. You cannot use KeyEvents to send or intercept characters in other applications. However, you can use the System procedure **sendKeys** to send specified keystrokes to other applications.

The following simple example demonstrates how this model works.

1 First, create a blank, unbound form.

2 Place an unbound field object on the page.

3 Attach the following code to the form's built-in **keyPhysical** method. The code displays a dialog box telling you which key was pressed.

```
method keyPhysical(var eventInfo KeyEvent)
if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
     msgInfo("keyPhysical", eventInfo.vChar())
  else
    ;code here executes just for form itself
  endIf
endMethod
```

4 Attach the following code to the form's built-in **keyChar** method:

```
method keyChar(var eventInfo KeyEvent)
if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
     msgInfo("keyChar", eventInfo.vChar())
  else
    ;code here executes just for form itself
  endIf
endMethod
```

5 Run the form.

6 Press *Ctrl+F4*. This keystroke is intercepted by Windows and a dialog box appears, asking if you want to save this form before closing it. Choose Cancel to close the dialog box. The ObjectPAL code you attached to the form's **keyPhysical** and **keyChar** methods doesn't execute.

7 Press *F9*. Windows passes this keystroke to Paradox, which sends it to the form's **keyPhysical** method. The **keyChar** method doesn't execute.

8 Press *A*. Both **keyPhysical** and **keyChar** execute and the field object displays the letter.

9 Press *Alt*, then *F*. Both **keyPhysical** and **keyChar** execute and the File menu appears.

The code in the following example is attached to the **keyPhysical** method of an unbound field object. It processes most keystrokes normally—for example, typing the letter A puts an *A* in the field. But when the user presses an arrow key, the field object moves! (It's easier to see it move if you set the field object's color to one that's different from the color of the page.) This example uses virtual key codes, explained in the next section.

Note To ObjectPAL, the screen is a two-dimensional grid, with the origin (0, 0) at the upper left corner of an object's container, positive x-values extending to the right, and positive y-values extending down.

```
method keyPhysical(var eventInfo KeyEvent)
   var
      x, y LongInt
      posPt Point
   endVar
```

```
    disableDefault                ; prevent the built-in code from executing

    posPt = self.position         ; position is a property
    x = posPt.x()
    y = posPt.y()

    switch
       case eventInfo.vCharCode() = VK_LEFT  : x = x - 100
       case eventInfo.vCharCode() = VK_RIGHT : x = x + 100
       case eventInfo.vCharCode() = VK_UP    : y = y - 100
       case eventInfo.vCharCode() = VK_DOWN  : y = y + 100
       otherwise : enableDefault  ; enable built-in code
    endSwitch
    self.position = Point(x, y)
endMethod
```

Notice the use of **disableDefault** and **enableDefault** in the example. The call to **disableDefault** at the beginning of the method prevents the built-in code from executing for any keystroke. The **switch...endSwitch** structure checks the incoming keystroke. If it's one of the arrow keys, it sets the value of *x* or *y*, as appropriate, and the built-in code does not execute. If the incoming key is not an arrow key, the OTHERWISE clause calls **enableDefault** to allow the built-in code to execute and process the keystroke normally, displaying it in the field object.

Note For more information about **switch...endSwitch**, **disableDefault**, and **enableDefault**, see the online ObjectPAL Help.

The following example shows how to intercept *F1*. By default, *F1* invokes the Paradox Help system. When you create your own applications, you might also want to provide your own help system and enable the user to invoke it in the usual way. This example assumes that you have created a custom help file named APPHELP.HLP, and that this file is on your path. Attach the code to the form's built-in **keyPhysical** method, as shown.

```
method keyPhysical(var eventInfo KeyEvent)
   if eventInfo.isPreFilter()
      then
         ; code here executes for each object in form
      else
         ; code here executes just for form itself
         if eventInfo.vChar() = "VK_F1" then        ; when user presses F1
            helpShowContext("appHelp.hlp", appHelp) ; invoke help
            disableDefault                          ; block default behavior
         endIf
   endIf
endMethod
```

Keys and virtual key codes

Paradox uses virtual key codes to map keyboard keys to integer values. For example, the virtual key code for *Tab* is 9, and the virtual key code for the letter A is 65. ObjectPAL provides Keyboard constants for virtual key codes, so you don't have to remember numeric values. For example, the Keyboard constant for *Tab* is VK_TAB (VK stands for

virtual keycode). ObjectPAL does not provide Keyboard constants for alphanumeric characters because they're easy to remember and type directly.

Although the Keyboard constants are defined to represent integer values, you can use them as quoted strings with certain methods. For example, the following code (attached to an unbound field object) displays VK_TAB as a string when you press *Tab*.

```
method keyPhysical(var eventInfo KeyEvent)
   x = eventInfo.vChar()
   x.view()
endMethod
```

In general, you can work with virtual key codes as integers or strings, depending on your needs, personal preference, or the syntax of the method you're calling. In some cases, you may need to convert an integer key code to a string, or a string to an integer key code. Table 10.5 lists the procedures ObjectPAL provides for this purpose (they're defined for the String type).

Table 10.5 Procedures for converting between ANSI codes and key names

Method	Description
ChrToKeyName	Returns the key name of the character contained in a string.
VKCodeToKeyName	Returns the key name corresponding to a specified ANSI code.
KeyNameToChr	Returns a string of length 1 containing the ANSI code for the key name.
KeyNameToVKCode	Returns a SmallInt representing the ANSI code for the given key name.

Here are some examples for converting between virtual key codes and key names.

```
ChrToKeyName(Chr(VK_CANCEL)) = "VK_CANCEL"
ChrToKeyName(eventInfo.vChar()) = "VK_CANCEL"

VKCodeToKeyName(VK_CANCEL) = "VK_CANCEL"
VKCodeToKeyName(eventInfo.vCharCode()) = "VK_CANCEL"

KeyNameToChr("VK_CANCEL") = Chr(VK_CANCEL)
KeyNameToChr("VK_CANCEL") = eventInfo.vChar()

KeyNameToVKCode("VK_CANCEL") = VK_CANCEL
KeyNameToVKCode("VK_CANCEL") = eventInfo.vCharCode()
```

MenuEvent: Handling menu choices

The MenuEvent type provides methods for working with menus in the application menu bar, including the Windows system menus. When the user chooses an item from a menu (or clicks a Toolbar button that performs a menu action), the **menuAction** method is triggered. By attaching code to an object's **menuAction** method, you can define how the object responds to choices from custom menus you create, Paradox's built-in menus, and the Windows system menus.

Important For more examples showing how to work with menus, see Lesson 9 in the ObjectPAL tutorial in Chapter 2. Also see Chapter 14.

Custom menus

Here's a simple example that displays the menu choice in an unbound field when you click a button.

The following code is attached to a button's **pushButton** method. It builds and displays a custom menu of three items listed horizontally across the application menu bar.

```
method pushButton(var eventInfo Event)
   var
      m Menu
   endVar
   m.addText("one")
   m.addText("two")
   m.addText("three")
   m.show()
endMethod
```

The following code is attached to an unbound field. It displays the user's menu choice in the field.

```
method menuAction(var eventInfo MenuEvent)
   self = eventInfo.menuChoice() ; displays the menu choice in the field
endMethod
```

Built-in menus

Paradox assigns an ID number to each item in its built-in menus, so you can use the **id** method defined for the MenuEvent type to test choices made from built-in menus.

For example, let's say you want to display a message to users when they close your application. If you use Paradox's built-in menus, you could do the following to modify an object's **menuAction** method:

```
method menuAction(var eventInfo MenuEvent)
   if eventInfo.id() = MenuFileExit then ; MenuFileExit is a constant
      msgInfo("Good-bye", "Thank you.")
   endIf
endMethod
```

In this example, MenuFileExit is a MenuCommand constant defined by ObjectPAL to represent the value returned when you choose File | Exit from Paradox's built-in menu.

User-defined menu constants

You can also define your own menu constants as long as you keep them within a specific range. Since this range is subject to change in future versions of Paradox, ObjectPAL provides the constants UserMenu and MaxUserMenu to represent the minimum and maximum values allowed.

For example, suppose you want to define two menu constants, ThisMenuItem and ThatMenuItem. In a Const window, define values for your custom constants as follows:

```
Const
   ThisMenuItem = 1
```

```
   ThatMenuItem = 2
EndConst
```

Then, to use one of these constants, add it to UserMenu:

```
method MenuAction(Var eventInfo MenuEvent)
   if eventInfo.id() = ThisMenuItem + UserMenuItem then
      doSomething()
   endIf
endMethod
```

By adding UserMenu to your own constant, you guarantee yourself a value above the minimum. To keep the value under the maximum, you'll have to check the value of MaxUserMenu. One way is to use a **message** statement:

```
message(MaxUserMenu)
```

In this version of Paradox, the difference between UserMenu and UserMenuMax is 2047. That means the largest value you can use for a menu constant is UserMenu + 2047.

Control menus

You can use **id** and MenuCommand constants in an object's built-in **menuAction** method to test for choices (like Minimize and Maximize) from the Windows control menu. For example, the following code prevents you from minimizing the current form:

```
method menuAction(var eventInfo MenuEvent)
   if eventInfo.id() = MenuControlMinimize then
                        ; MenuControlMinimize is a constant
      disableDefault ; do not execute the default code
      beep()
      message("Can't minimize this form.")
   endIf
endMethod
```

MouseEvent: Handling mouse actions

Methods in the MouseEvent type answer questions related to a mouse click, including those listed in Table 10.6.

Table 10.6 MouseEvent methods

Question	Methods
Which button is clicked?	isLeftDown, isMiddleDown, isRightDown, setLeftDown, setMiddleDown, setRightDown
Where is the pointer?	getMousePosition, x, y, setX, setY, isInside, setInside, setMousePosition
Is a key held down?	isAltKeyDown, isControlKeyDown, isShiftKeyDown, setAltKeyDown, setControlKeyDown, setShiftKeyDown
Which object got the event?	getObjectHit, getTarget

Responding to mouse actions

UIObjects come with built-in methods you can modify to respond to many common mouse events. In addition, you can use **hasMouse** to find out if the mouse pointer is over an object. You can also use **wasLastClicked** and **wasLastRightClicked** to find out if an object was the last object to receive a mouse click (or a right mouse click).

```
var
   myObject UIObject
endVar
myObject.attach(myTableFrame) ; assume a form contains a TableFrame
if myObject.hasMouse() then
   message("Eek! A mouse!")
else
   message("No mouse here.")
endIf
if myObject.wasLastClicked() then
   message("I got the last left click.")
endIf
if myObject.wasLastRightClicked() then
   message("I got the last right click.")
endIf
```

MoveEvent: Moving between objects

Methods for the MoveEvent type enable you to get and set information about the events that occur as you navigate from one object to another in a form. The following built-in methods are triggered by MoveEvents: **arrive**, **canArrive**, **canDepart**, and **depart**. These methods, along with the rest of the built-in methods, are discussed in Chapter 12. This section presents an overview of how to use MoveEvent type methods.

Setting error codes

Use **setErrorCode** with *eventInfo* to specify the status of a MoveEvent. For example,

```
method canArrive(var eventInfo MoveEvent)
   eventInfo.setErrorCode(CanNotArrive) ; CanNotArrive is an ObjectPAL constant
endMethod
```

By attaching the above method to a field, you can prevent a user from moving the pointer into the field. The constant CanNotArrive has been assigned a nonzero value by ObjectPAL, and any nonzero error value blocks this method's built-in code from executing. Instead, Paradox tries to move to the next object in the tab order.

Note A call to **setErrorCode** does not invoke Paradox's error-handling mechanism (described in Chapter 30). It simply sets information used by the default code.

Using reason with MoveEvents

You can use the MoveEvent type method **reason** to find out why a move is taking place. Following is an example using **reason** with **canDepart**.

```
method canDepart(var eventInfo MoveEvent)
   if eventInfo.reason() = UserMove then
      doSomething() ; do some custom method
   endIf
endMethod
```

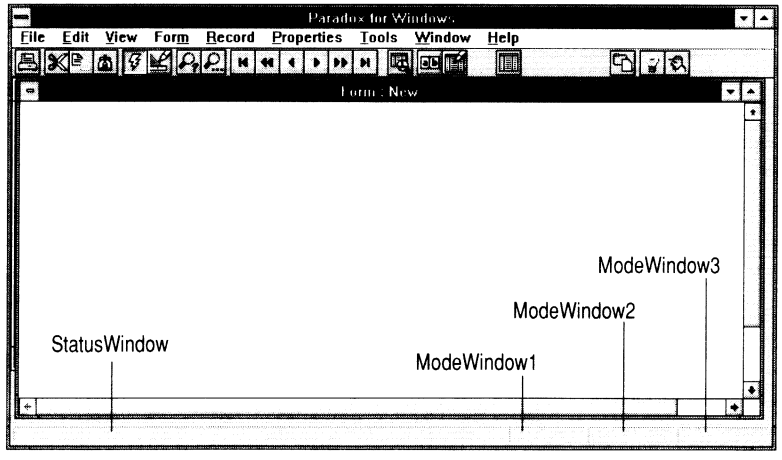
In this example, the custom method **doSomething** executes only when **canDepart** is called because of a user action; otherwise, the event is handled normally. (UserMove is an ObjectPAL constant.)

StatusEvent: Controlling the status bar

StatusEvent type methods control messages that are displayed in the Desktop status bar. Every design object has a built-in **status** method. Using StatusEvent type methods, you can attach code to the built-in method to find out where messages will be displayed and why. You can also block the messages or display them in a different status area, or in another object (for example, in a field object or a text file). You can also specify the text to be displayed in the message.

You can use the StatusReasons constants ModeWindow1, ModeWindow2, ModeWindow3, and StatusWindow to refer to the areas of the status bar shown in Figure 10.5. Paradox and ObjectPAL place no restrictions (other than the size of the area) on the messages you display in these areas. How you use them is up to you, but consistency is recommended.

Figure 10.5 The status bar



The status bar contains four windows. You can address them using **reason** and **setReason** with the StatusReasons constants.

You can use **reason**, **setReason**, and the StatusReasons constants to find out and specify where a message will be displayed (see the discussion of **reason**, earlier in this chapter). For example,

```
if eventInfo.reason() = ModeWindow2 then
   eventInfo.setReason(StatusWindow) ; redirect the message
endIf
```

This example watches for messages going to the second mode area and redirects them to the status area.

Note Within the **status** method, calling **message**—or any other code that displays text in the status bar—does *not* trigger a new StatusEvent. Thus ObjectPAL avoids an infinite loop.

You can use **statusValue** and **setStatusValue** to get and set the text of the message. For example,

```
method status(var eventInfo StatusEvent)
   if eventInfo.statusValue() = "First Record" then
      eventInfo.setStatusValue("Home") ; change the message
   endIf
endMethod
```

This example intercepts and changes the default message that appears when you move the pointer over the First Record button on the Toolbar. By default, moving the pointer over a Toolbar button triggers the built-in **status** method (reason = StatusWindow) and displays a message in the status area; moving the pointer off a Toolbar button also triggers **status** (reason = StatusWindow) to clear the status area.

The following example intercepts any status message and displays it in a dialog box instead of in the status bar.

```
method status(var eventInfo StatusEvent)
   msgInfo("Message:", eventInfo.statusValue())
                     ; display the message in a dialog box
   disableDefault ; block the built-in code from executing
endMethod
```

TimerEvent: Events at specified intervals

TimerEvent type methods process information used by the timer method built into every design object. For example, **isFirstTime** reports whether the form is seeing a particular TimerEvent before dispatching it to the intended target object, and **getTarget** reports which object is the intended target.

Use **setTimer**, defined for the UIObject type, to specify when to send timer events to an object, then to modify the object's built-in **timer** method to control how the object responds. (Use **killTimer**, defined for the UIObject type, to turn off an object's timer.) The following method examples change a button's position every tenth of a second, so it appears to float across the screen.

These methods assume you have already created a form and drawn a button. First, declare variables in the button's Var window to make them visible to all the button's methods:

```
var
   posPt Point
   x, y LongInt
endVar
```

Use the following code to modify the button's **open** method.

```
method open(var eventInfo Event)
   self.setTimer(100)          ; generate a TimerEvent every 100/1000 of a second
   posPt = self.position       ; position is a property
   x = posPt.x()               ; get the x-coordinate of the position
   y = posPt.y()               ; get the y-coordinate of the position
endMethod
```

Use the following code to modify the button's **timer** method.

```
method timer(var eventInfo TimerEvent)
   x = x + 100
   if x > 5000 then
      x = 10
   endIf
   self.position = Point(x, y) ; move to the new position
endMethod
```

ValueEvent: Handling field value changes

ValueEvent methods control what happens when the value of a field changes. The set of built-in methods for field objects includes two methods: **changeValue** and **newValue**. **changeValue** asks permission to change the value of a field—this method gives you a chance to check the value and decide whether you really want to post it. **newValue** reports when a field has received a new value. For example, when you type a value in a field object and then commit the change (typically by moving off the field object), the sequence is

```
changeValue
newValue
```

The sequence is different for a field displayed as radio buttons, a list, or a drop-down edit list. Using these types of fields, you specify a value with a single action—either clicking a radio button or selecting a list item—instead of the series of keystrokes needed to type in a value. So with radio buttons, lists, and drop-down edit lists, when you specify a value, you trigger **newValue**, **changeValue**, and then **newValue** again. However, the first **newValue** and the second **newValue** happen for different reasons.

You can use **reason** with **eventInfo** and ValueReason constants to test the reason for a **newValue** method (but not for **changeValue**) as shown below. The first time, the constant is EditValue because you've edited the field and specified a new value, but the value has not been committed. The second time, the constant is FieldValue to report that the field has a new value. The sequence, in pseudocode, is as follows:

```
newValue, eventInfo.reason() = EditValue     ; new value is specified
; Try to commit the change (for example, by moving to another object)
changeValue                                  ; ask permission to change value
newValue, eventInfo.reason() = FieldValue    ; report about the new value
```

Also, when a form opens, it triggers **newValue** for every field object in the form; the reason constant is StartupValue.

Important A calculated field only displays values, and does not write them to a table. Paradox never writes the data from a calculated field to a table, so it never calls the field's **changeValue** method. However, it does call **newValue** each time the field displays a new value. For more information, see Chapter 12.

In the following example, suppose a form contains a field object bound to the Sale Date field of the *Orders* table, and contains a button called *sendError*. The **pushButton** method for *sendError* creates a ValueEvent and sets the error code for that event to CanNotDepart. The event is then sent off to the **changeValue** method for *Sale_Date*.

Note The name of a field in a table can contain spaces, but the name of an object in a form cannot. By default, Paradox replaces a space with an underscore. So, in this example, Sale Date refers to the field in the table, and *Sale_Date* refers to the field object in the form.

```
method pushButton(var eventInfo Event)
var
  va  ValueEvent              ; the event to send
  ui  UIObject
endVar
va.setErrorCode(CanNotDepart) ; set an error
ui.attach(Sale_Date)
ui.changeValue(va)            ; send off the event
endMethod
```

This code is attached to the **changeValue** method for the *Sale_Date* field object. For the purpose of this example, it merely reports on the error.

```
method changeValue(var eventInfo ValueEvent)
  msgInfo("And the error was...",
          String(eventInfo.errorCode()))
endMethod
```

For the next example, suppose you want to find out whether the user has changed any of the fields in a record. One way to do this is to check the value of the Touched property for the record. However, Touched becomes True when the user edits a field but doesn't set it to a *new* value.

To find out if the old value is different from the new value, you can compare the old and new value in the **changeValue** method on the form, and set a flag only if the values are different. The next example assumes that a form has a multi-record object bound to the ORDERS.DB table, and a button called *undoButton*. When the *UndoFlag* variable is True, pressing the *Undo* button cancels the changes to the record.

```
Var
  UndoFlag Logical
endVar
```

This method is attached to the form's **open** method:

```
method open(var eventInfo Event)
if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
  else
    ;code here executes just for form itself
```

```
    UndoFlag = False
endIf
endMethod
```

This method is attached to the form's **changeValue** method:

```
method changeValue(var eventInfo ValueEvent)
var
  targObj  UIObject              ; get the target of the event
endVar
if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
    eventInfo.getTarget(targObj)       ; get the target of the change
    if targObj.Value <> eventInfo.newValue() then
       UndoFlag = True
       UndoFlag.view()
    endIf
  else
    ;code here executes just for form itself

endIf
endMethod
```

This code is attached to *undoButton*'s **pushButton** method:

```
method pushButton(var eventInfo Event)
if UndoFlag then                   ; check if record has changed
  ORDERS.action(DataCancelRecord)
  UndoFlag = False
else
  msgInfo("Status", "No changes to undo.")
endIf
; note that you can also examine the Touched property of the MRO
endMethod
```

Checking field values

A common task is to inspect the ValueEvent event packet to do some simple value checking. The best place to attach this kind of code is the built-in **changeValue** method, because this method effectively asks for permission to commit the value, giving you a chance to inspect values and make decisions.

For example, suppose the following code is attached to the built-in **changeValue** method of a field object. When you try to commit the field's value, this code executes. It checks the new value displayed in the field object, and if the value is greater than 50, it displays a dialog box and sets an error code to prevent the insertion point from leaving the field (a nonzero error code indicates an error).

```
method changeValue (var eventInfo ValueEvent)
   var
      atNewVal AnyType
   endVar
   atNewVal = eventInfo.newValue()
   if atNewVal > 50 then
```

```
        msgStop("Stop", "Enter a value less than 50.")
        eventInfo.setErrorCode(CanNotDepart) ; CanNotDepart is a nonzero constant
    endIf
endMethod
```

For more information, see the discussion of **changeValue** in the "Special built-in methods" section of Chapter 12.

Demonstration: Events, objects, and containership

This section presents steps for creating a form and writing methods that demonstrate the relationship between objects, containership, and events.

This simple exercise illustrates some key elements of ObjectPAL programming:

- Objects respond to events.
- One event can (and usually does) trigger a sequence of events that execute behind the scenes.
- An object's position in the containership hierarchy determines when it receives events.
- You can intercept these events.
- You can define how objects respond.

Step 1: Getting started

1. To begin, choose File | New | Form from the Desktop, and click the Blank button in the New Form dialog box to create a blank form.
2. Use the Box tool to draw a box with sides about 2 inches long in the upper left corner of the form.
3. Inspect the box and change its name to *leftOutsideBox*.
4. Select the box, then press *Ctrl+Spacebar* to display the Method Inspector.
5. Double-click **canArrive** to open an ObjectPAL Editor window. (**canArrive** is a built-in method that executes when ObjectPAL asks permission to make an object active. It's analogous to knocking on a door before entering a room.)
6. Edit the method so it looks like this:

```
method canArrive(var eventInfo MoveEvent)
self.color = DarkGray
message(self.name, "   canArrive")
sleep(1000)
endMethod
```

   Three spaces were placed between the double quote and the word *canArrive* to make the message readable. You can use as many spaces as you like (including none).
7. Select and copy the three lines you added to the method.

8 Close the window and save the changes if you're prompted to do so.

9 Follow the previous steps (you can use the text you copied to save some typing) to edit the **arrive**, **setFocus**, **canDepart**, **depart**, and **removeFocus** built-in methods for *leftOutsideBox* as shown in the following method examples.

arrive method

An object's built-in **arrive** method executes when the object becomes active.

```
method arrive(var eventInfo MoveEvent)
self.color = Gray
message(self.name, "   arrive")
sleep(1000)
endMethod
```

setFocus method

An object's built-in **setFocus** method executes when the object has *focus*, that is, when it can receive input from the keyboard or the mouse.

```
method setFocus(var eventInfo Event)
self.color = Blue
message(self.name, "   setFocus")
sleep(1000)
endMethod
```

canDepart method

An object's built-in **canDepart** method executes when ObjectPAL asks permission to leave the object.

```
method canDepart(var eventInfo MoveEvent)
self.color = LightBlue
message(self.name, "   canDepart")
sleep(1000)
endMethod
```

removeFocus method

An object's built-in **removeFocus** method executes when the object gives up focus, and can no longer receive keyboard or mouse input.

```
method removeFocus(var eventInfo Event)
self.color = DarkGreen
message(self.name, "   removeFocus")
sleep(1000)
endMethod
```

depart method

An object's built-in **depart** method executes when the object is no longer active.

```
method depart(var eventInfo MoveEvent)
self.color = Green
message(self.name, "   depart")
```

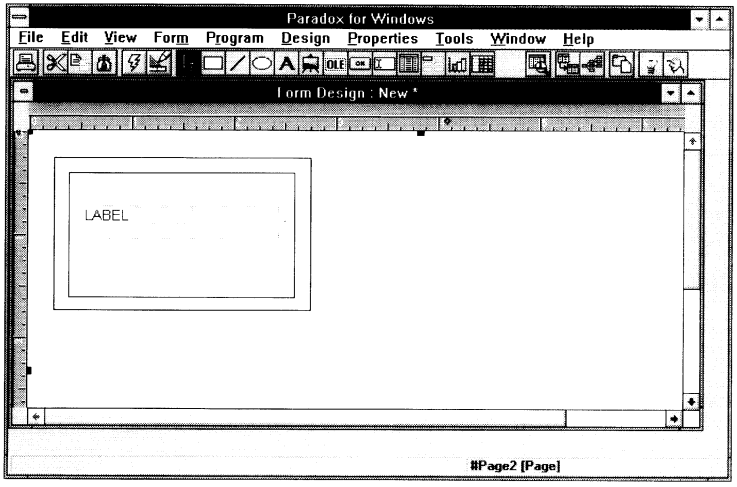
```
sleep(1000)
endMethod
```

Step 2: Copying an object and its methods

The next task is to copy this box and its methods. By copying *leftOutsideBox*, you copy all of its methods.

1. Choose *leftOutsideBox*, then choose Design | Duplicate to copy it.
2. Change the name of this duplicate box to *leftInsideBox*.
3. Resize *leftInsideBox* to make it smaller, then move it completely inside *leftOutsideBox*.
4. Use the Field tool to draw a field object inside *leftInsideBox*, as shown in Figure 10.6. Leave the field object undefined.

Figure 10.6 Draw a field inside *leftInsideBox*



5. Select *leftOutsideBox*, (which includes *leftInsideBox* and the undefined field object, because *leftOutsideBox* contains *leftInsideBox* and the field object), then choose Design | Duplicate to copy these objects and their methods.

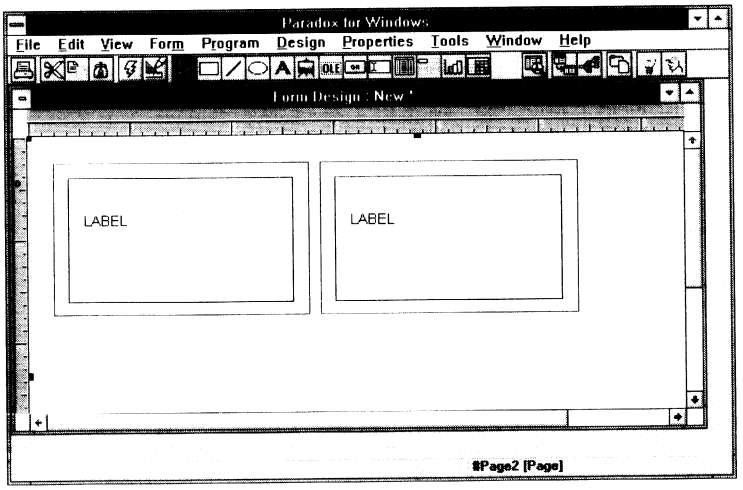
Stop for a moment and consider what you just did: by selecting one object, you selected all the objects that object contains. By duplicating one object, you duplicated all the objects that object contains, along with their methods. You've used the containership model to streamline the design process.

Step 3: Moving and using duplicated objects

The next task is to move the duplicate objects and trigger one event that starts a sequence of events.

1 Move the duplicates to the right as shown in Figure 10.7. Rename the outside box *rightOutsideBox* and rename the inside box *rightInsideBox*.

Figure 10.7 Move the duplicates



2 Save this form, then choose View | View Data or click the View Data button.

3 Press *Tab* and watch what happens. Do it again. By pressing *Tab* you trigger one event that starts a chain reaction. The box colors change as each object responds to the sequence of events from **canArrive** to **depart**.

Note The two field objects are responding to the same sequence of events by executing their built-in methods. No messages appear, though, because you didn't attach any of your own code to the built-in methods.

Using the ObjectPAL Tracer

You can use ObjectPAL's Tracer to list each ObjectPAL statement as it executes. To open the Tracer window,

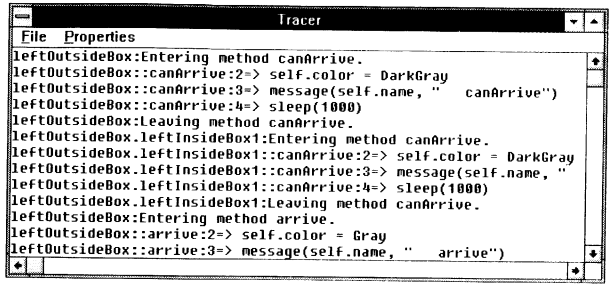
1 Choose View | Tracer in the design window, or click the Trace Window button from the Debugger or Editor Toolbars.

2 Then choose Program | Compile With Debug so the Tracer will list custom methods and code attached to built-in methods.

Now when you switch to the View Data choice, the Tracer lists each line of ObjectPAL code as it executes, providing a record of what happened and when. Figure 10.8 shows the ObjectPAL Tracer listing ObjectPAL statements as they execute.

To display even more information, choose Properties | Builtins in the Tracer window and check the built-in methods you want to trace as they execute, whether or not they have ObjectPAL code attached. Chapter 9 gives more information about using the Tracer.

Figure 10.8 The ObjectPAL Tracer lists ObjectPAL statements as they execute

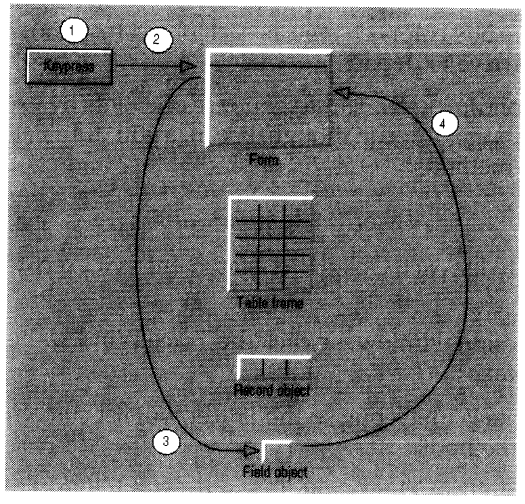


Advanced topic: Sample flow of method calls

Note This section presents technical information for advanced programmers. You don't need to master this material to use ObjectPAL effectively.

Here's a sample set of actions. Suppose you just finished typing into a field object in the third row of a table frame and pressed ↓. Following is the flow of events and methods (diagrammed in Figures 10.9, 10.10, and 10.11):

Figure 10.9 Steps 1 through 4



1. While the field object is active, press the down arrow.
2. form.keyPhysical()
   target = form
3. fieldObject.keyPhysical
   target = fieldObject
4. form.action(MoveDown)
   target = fieldObject

1 Press ↓.

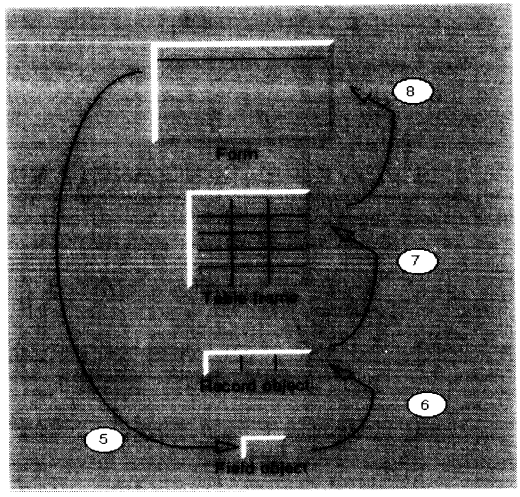
2 Paradox constructs a KeyEvent event packet and calls the form's **keyPhysical** method. The form is the target.

3 In the KeyEvent event packet, the form sets the target to be the field object (because it's active) and dispatches the event to the field object's **keyPhysical** method.

4 The field object's default **keyPhysical** method maps the keystroke to an action. It constructs an ActionEvent event packet with the action constant MoveDown, and

calls the form's **action** method. In other words, it sends **action(MoveDown)** to the form, and the field object is the target.

Figure 10.10 Steps 5 through 8



5. fieldObject.action(MoveDown)
   target = fieldObject

6. recordObject.action(MoveDown)
   target = fieldObject

7. tableFrame.action(MoveDown)
   target = fieldObject

8. form.action(dataNextRecord)
   target = fieldObject

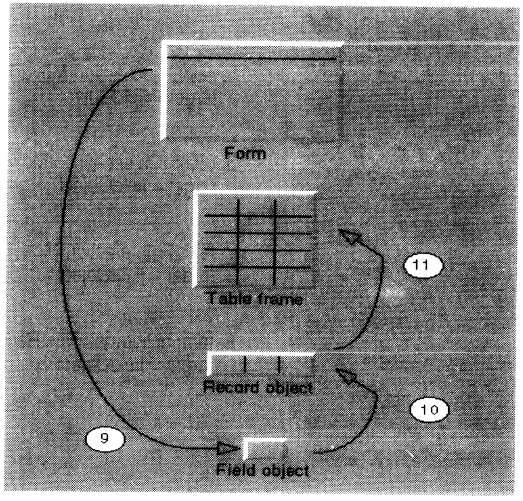
5 The form's **action** method dispatches the event to the field object's **action** method.

6 The field object's **action** method doesn't know what MoveDown means, so it bubbles it up one level to the surrounding record object.

7 The record object's **action** method doesn't know MoveDown, either, so it also bubbles it up one level to the table object.

8 The table object knows what MoveDown means and translates it to **action(DataNextRecord)**, calling the form's **action** method, again with the field object as the target.

Figure 10.11 Steps 9 through 11



9. fieldObject.action(DataNextRecord)
   target = fieldObject

10. recordObject.action(DataNextRecord)
    target = fieldObject

11. tableFrame.action(DataNextRecord)
    target = fieldObject

9 The form dispatches the event to the field object's **action** method with the constant DataNextRecord.

10 The field object's **action** method bubbles the DataNextRecord to the record again.

11 The record bubbles it to the table frame.

What happens next depends on the size of the table frame. If it has a record below the current record, Paradox simply moves to the corresponding field object in that record. If the table frame does not display another record (in other words, if the current record is the last record displayed) and Paradox has to scroll the table frame, the table frame handles the DataNextRecord as follows:

- It generates a **moveTo** the table frame, causing a **canDepart**, a **removeFocus** and a **depart** to happen to the field object and the record object. This causes the field object's value to be posted to the record (but not to the underlying table) and the selection highlight to disappear.
- If the record has been modified, its **depart** method calls the **action** method of the table frame with the constant DataUnlockRecord. The default for this attempts to post the record to the underlying table, and, if the database engine accepts it, returns a success code of zero.
- It asks the database engine to move forward one record.
- If it succeeds, the table frame determines which record is now current onscreen.
- It generates a **moveTo** the next field object, causing a **canArrive**, an **arrive**, and a **setFocus** to be called for the new record and field objects.

This may seem like a lot of steps, but they execute quickly. The benefit to you, the programmer, is a great deal of flexibility and control, because you know that no matter what the user does (mouse-click, tab, or arrow keys), everything translates to a relatively small set of data-specific actions.

Actions and UIObjects

This chapter explains how to use ObjectPAL to make UIObjects initiate and respond to actions. It presents some general information about actions, then it describes

- The **action** method defined for the UIObject type, used to initiate actions
- The built-in **action** method, used to respond to actions

Understanding actions

Understanding actions is a key to getting the most out of ObjectPAL.

Table 11.1 describes the five basic action categories.

Table 11.1 Action categories

Category	Description
Data actions	Data actions are for navigating in a table and for such tasks as record locking and record posting. The Toolbar navigation controls trigger these actions, as do many items in the Form and Record menus. The form itself may use these actions to attempt to lock and post records as needed. This means individual objects can intercept and block (if desired) each of these actions, so they afford a considerable amount of power to ObjectPAL.
Edit actions	Most Edit actions alter data within a field, and work in conjunction with Select actions. For example, you might want to select a word displayed in a field object (a Select action), then copy the word to the Clipboard (an Edit action). In Field View, Edit actions operate on characters within the field object. Outside of Field View, these actions work on the entire field object. Edit actions take effect only when an object is active (selected).
Move actions	Move actions have to do with moving within a field object, or moving between objects. Move actions are typically triggered by the arrow keys. In Field View, they move within the field (for example, to the start of a line). When not in field view, they move to another object as appropriate. Naturally, not all actions make sense to all objects, but those that do are performed appropriately.

Table 11.1 Action categories (continued)

Category	Description
Field actions	Field actions are a special category of Move actions that enable movement between field objects (for example, moving to the next field object in the tab order).
Select actions	Select actions are equivalent to Move actions, with the additional constraint that they extend a selection as they move. You cannot extend a selection across objects—only within a field object.

Action constants

ObjectPAL provides constants you can use to work with actions. When you write ObjectPAL code to work with actions and UIObjects, these constants are very important. Like the actions themselves, these constants are organized into categories:

ActionDataCommands
ActionEditCommands
ActionFieldCommands
ActionMoveCommands
ActionSelectCommands

The constants are listed online; because they're so important, you should browse through them at least once, just to get an idea of what's available. To display the list, open an ObjectPAL Editor window and choose Tools | Constants to display the Constants dialog box. Then, from the Types of Constants panel, choose one of these constants categories. The constants appear in the Constants panel.

To learn what each constant does, refer to the online ObjectPAL Help, which describes them all.

Actions, ObjectPAL, and Paradox

Actions, ObjectPAL, and Paradox are closely related because most things that you can do, either in ObjectPAL or using Paradox interactively, can be thought of as actions. For example, when you use ObjectPAL to change the value of a field, that's a specific action. When you choose an item from a menu, that's another kind of action. Some methods and procedures in the ObjectPAL run-time library are actually requests for an action.

Using ObjectPAL, you can initiate any of these actions, and you can specify how objects respond to them. The next sections explain how.

Initiating actions

To initiate an action—that is, to tell a UIObject to do something—use the **action** method defined for the UIObject type. For example, the following statement moves to the next record in a table frame or multi-record object bound to the *Orders* table:

```
ORDERS.action(DataNextRecord)
```

This statement uses the constant DataNextRecord to specify which action to perform.

The next statement invokes Paradox's built-in Search & Replace dialog box:

```
orders.action(DataSearchReplace)
```

The user can enter values into the dialog box and Paradox will perform the search and replace operation—there's no need to reinvent the wheel.

Note Some methods and procedures in the ObjectPAL run-time library are actually calls to the **action** method. For example, the first statement is actually a call to the second statement.

```
tableFrame.nextRecord()
tableFrame.action(DataNextRecord)
```

The two statements are equivalent, and you can use either in your code.

The action method and the event model

When code attached to a design object calls **action**, it generates an ActionEvent that Paradox handles as defined by the event model (described in Chapter 10): the event goes to the form first, triggering its built-in **action** method. By default, the form dispatches the event to the built-in **action** method of the intended target object, and from there the event can bubble up the containership hierarchy until an object handles it or it reaches the form for the second time.

Previous examples used an object name with **action** to specify a target object, and in most cases that's the correct approach. To move to the last record of the *Orders* table frame, call

```
orders.action(DataEnd)
```

In this case, the event goes to the form, which dispatches it directly to *Orders*.

Calling action with an object

In cases where you want more generalized code, you can call **action** with an object variable, or you can call **action** without specifying an object at all. The resulting sequence of events and objects depends on the containership hierarchy; in particular, it depends on the relationship between the calling object (the object whose code called **action**) and the active object.

Calling action without an object

Calling the UIObject type method **action** without specifying an object has the same effect as calling **action** and specifying *self*. In other words, the following statements are equivalent:

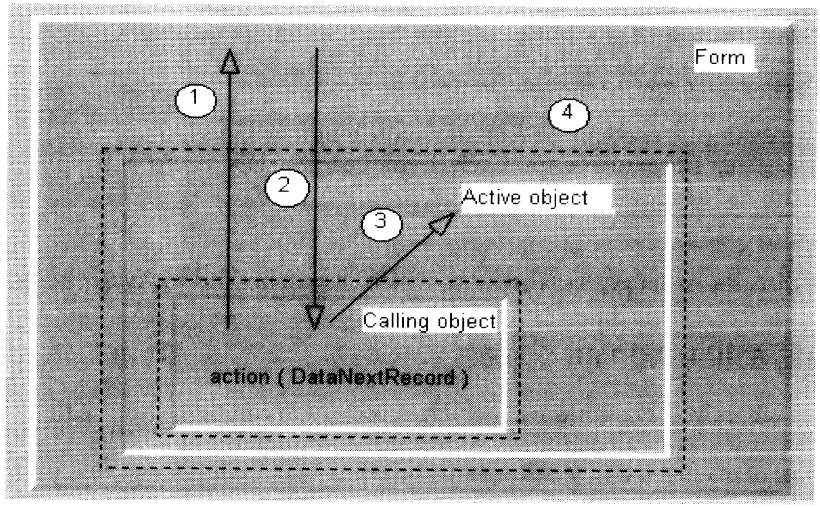
```
action(DataNextRecord)
self.action(DataNextRecord)
```

Figures 11.1 and 11.2 diagram two scenarios for calling **action** without specifying an object. In the first scenario, the active object contains the calling object. In the second scenario, the active object and the calling object are in separate containership hierarchies.

In Figure 11.1, the active object contains the calling object (dashed lines represent possible intermediate containers). When the calling object calls **action** with some constant (for example, DataNextRecord), it generates an ActionEvent that triggers the built-in **action** method for the form. By default, the form dispatches the event to the built-in **action** method of the calling object, which bubbles it to its container, and so on until it reaches the active object. If the action is meaningful to the active object (for example, DataNextRecord is meaningful to a table frame), the active object performs the action. If the action is not meaningful, the active object bubbles it to its container, and so on until it reaches the form for the second time, and the form handles it.

Figure 11.1 Active object contains calling object

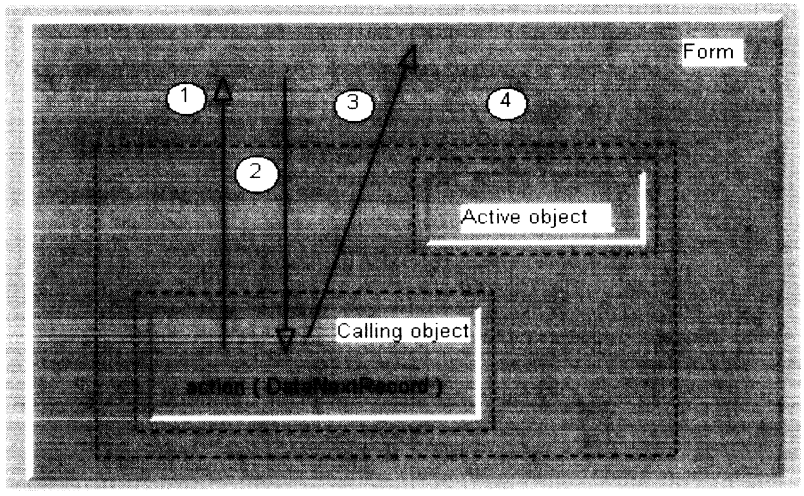
1. The calling object calls action(DataNextRecord), triggering the form's built-in **action** method.
2. The form dispatches the event to the built-in **action** method of the calling object.
3. The calling object bubbles the event to its container, and soon until it reaches the active object.
4. The active object performs the action if it can, or bubbles the event if it can't.



In Figure 11.2, the active object does not contain the calling object; they are in separate containership hierarchies. In this case, the ActionEvent goes to the form, then back to the calling object, then bubbles up the containership hierarchy—which *does not* include the active object—until it reaches the form again. The active object does not see the ActionEvent as it bubbles up to the form, but the form, upon receiving the event for the second time, can then dispatch it to the active object, if appropriate. (This same sequence occurs when the calling object contains the active object.)

Figure 11.2 Separate hierarchies, object not specified

1. The calling object calls action(DataNextRecord), triggering the form's built-in **action** method.
2. The form dispatches the event to the built-in **action** method of the calling object.
3. The calling object bubbles the event to its container, and so on (bypassing the active object) until it reaches the form.
4. The form handles the event, and can dispatch it to the active object, if appropriate.



Calling action with an object variable

When you call **action** with an object variable, the results depend on which variable you use. (The object variables are *self*, *subject*, *container*, *active*, *lastMouseClicked*, and *lastMouseRightClicked*. They are described in Chapter 12.) Calling **action** with *self* as an argument is the same as calling with no argument. The results are as described in the previous section. This section describes the effects of calling **action** with *active* as an argument, and the same principles apply when using the other object variables.

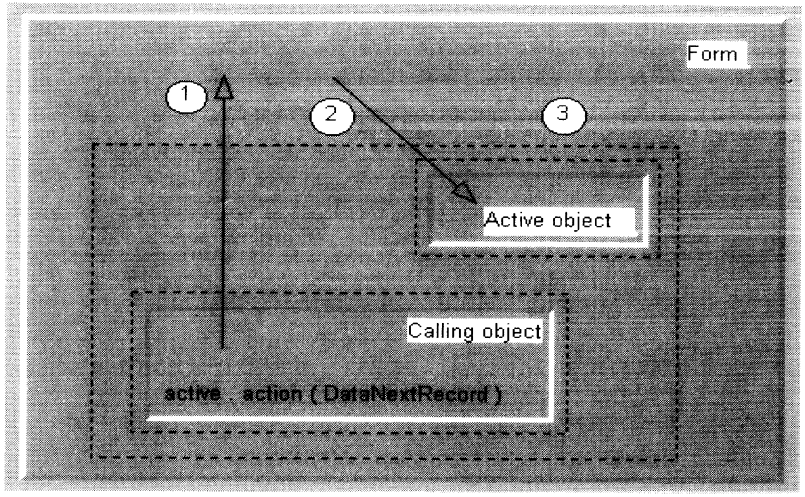
Figure 11.3 diagrams the effect of this statement:

```
active.action(DataNextRecord)
```

This statement generates an ActionEvent that triggers the form's built-in **action** method. The form dispatches the event to the built-in **action** event of the active object. The active object performs the action if it can, or bubbles it if it can't.

Figure 11.3 Separate hierarchies, active object specified

1. The calling object calls active.action(DataNextRecord), triggering the form's built-in **action** method.
2. The form dispatches the event to the built-in **action** method of the active object.
3. The active object performs the action if it can, or bubbles it if it can't.



To summarize, ObjectPAL lets you be as specific or general as you like when calling the UIObject type **action** method. The effects of generalized code depend on the relative positions of the calling object and the active object in the containership hierarchy.

Responding to actions

Every UIObject (including the form) has a built-in method named **action** that specifies how the object responds to actions. You can attach code to an object's built-in **action** method to control how that object responds in specific situations.

The *eventInfo* argument used by an object's built-in **action** method contains information about the kind of action intended; the method **id** defined for the ActionEvent type returns this information. For example, suppose the following code is attached to a record object's built-in **action** method:

```
method action(var eventInfo ActionEvent)
   if eventInfo.id() = DataUnlockRecord then ; if action tries to unlock record
      verifyRecord() ; execute a custom procedure
   endIf
endMethod
```

This code calls **id** to test *eventInfo*. If **id** returns a value equal to the constant DataUnlockRecord, the next statement calls a custom procedure to make sure the record values are valid.

Important An object's built-in **action** method executes in response to any action, whether the action was generated by ObjectPAL, by Paradox, or by the user interacting with the form.

Therefore, the built-in **action** method is an excellent place to put your code when you want to control how an object responds to an action, no matter how the action was generated.

Important Following is a list of actions that people who develop database applications typically want to control, along with strategies for controlling them.

Inserting a record

Inserting a record triggers an action; the constant is DataInsertRecord. In a multi-record form, the best place to handle this action is in the built-in **action** method of the table frame or multi-record object that contains the record. For single-record forms, attach the code to the form. The default code must execute to insert the record; you can prevent the insertion by setting an error code. When the default code executes, it causes a RefreshMove action out to the table frame or multi-record object (triggering the record object's built-in **arrive** and **depart** methods), inserts a new record, repaints the screen, and highlights the appropriate field inside the new record. You can set fields to default values, change the record color—whatever you want.

When a record is inserted—for *any* reason—DataInsertRecord is the only action you ever have to intercept to handle how new records are treated.

The code in the following example tests for an attempt to insert a record. It calls **doDefault** to execute the default code, then sets the value of the field object named *CompanyName* to Borland.

```
method action(var eventInfo ActionEvent)
   if eventInfo.id() = DataInsertRecord then
      doDefault
      CompanyName.value = "Borland"
   endIf
endMethod
```

Unlocking and posting a record

To handle record-level changes to data, respond to both the DataUnlockRecord and DataPostRecord actions. Both can be initiated either by the user interacting with the form, or by an ObjectPAL statement. However, they are different actions.

A DataUnlockRecord action says, in effect, "I'm finished with this record. Unlock it and post changes to the table. If the changes make the record fly away, let it go." In contrast, a DataPostRecord action says, "Post these changes to the table, but I want to stay with the record. Keep it locked, and if it flies away, fly with it." As you can see, these are two distinct actions, so you need to intercept them both. You can block either action by setting an error code before the default code executes.

As with DataInsertRecord, the default code for these actions initiates a RefreshMove action out to the table frame or multi-record object, then does the appropriate database operation (unlock or post), and then moves back into the appropriate field on the new current record.

This 3-stage move (move out to the table frame or multi-record object, do the operation, then move back into the record) is needed to handle records that fly away when a key value is changed. It also triggers the record's built-in **arrive** and **depart** methods, giving you a chance to update the user interface, if you want to.

The code in the following example is attached to a table frame. It responds to DataUnlockRecord and DataPostRecord action by testing the value of the *Qty* field object. If the value of *Qty* is less than the value of *minValue* (a constant defined elsewhere), this code sets the error code to block the action, and displays a message to inform the user.

```
method action(var eventInfo ActionEvent)
   var
      idVal SmallInt
   endVar

   idVal = eventInfo.id() ; idVal is used to save typing

   if idVal = DataUnlockRecord or idVal = DataPostRecord then
      if Qty.value < minValue then
         message("Enter a larger quantity.")
         eventInfo.setErrorCode(UserError)
      endIf
   endIf
endMethod
```

Deleting a record

Deleting a record triggers an action; the constant is DataDeleteRecord. As in previous examples, nothing happens to the record until the default code executes, and an error code can stop it. When it executes, the default code causes a move out to the table frame or multi-record object, asks the database to delete the record, and then causes a move back into the newly current record.

Deleting a record for any reason generates this action, so DataDeleteRecord is the action to intercept to handle record deletions.

The following example responds to DataDeleteRecord by displaying a dialog box where the user can confirm the decision to delete.

```
method action(var eventInfo ActionEvent)
   if eventInfo.id() = DataDeleteRecord then
      if msgQuestion ("Delete?", "Delete this record?") = "Yes" then
         doDefault
      else
         eventInfo.setErrorCode(UserError)
      endIf
   endIf
endMethod
```

Refresh exception

If someone across the network (or in another window) changes some record in the range you see in your form, an action occurs; the constant is DataRefresh. When it executes, the default code causes a move out to the table frame or multi-record object, recomputes the record's current position, and moves back to the field which used to be active (attempting to preserve the edit state).

If a data refresh occurs in a record that's not currently displayed in your form, the action is DataRefreshOutside. DataRefresh and DataRefreshOutside are typically used in applications that run on a network. You can respond to these actions by displaying a message to inform the user, as shown in the following example.

```
method action(var eventInfo ActionEvent)
   if eventInfo.id() = DataRefresh or eventInfo = DataRefreshOutside then
      message("Refreshing data ...")
   endIf
endMethod
```

Arriving at and departing from records

You can use one action constant, DataArriveRecord, to respond to an action that can be initiated 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; so does inserting or deleting a record; so does editing a record. A table frame or MRO containing the current record does not need to be active to receive this action. DataArriveRecord is a useful general-purpose action.

Refer to Lesson 2 in the ObjectPAL tutorial in Chapter 2 for an example of how to use DataArriveRecord.

As explained in the previous descriptions, inserting, deleting, unlocking, and posting a record all trigger the record object's built-in **arrive** and **depart** methods as side-effects of the default code. This means that to do things like set a record's color (for example), you can attach code to its built-in **arrive** method and be sure it will execute for each of those basic actions. So, if there are things you need to do to the user interface whenever you are arriving at or leaving a record (regardless of whether it's due to insertion, deletion, etc.), there is a place to attach your code. For example, when you click another record, it does not generate an action if the record was not locked—you'd have to rely on the **depart** and **arrive** sequence.

Where do I put my code?

To handle actions, the best place to attach code is the built-in **action** method of the lowest-level container of the objects you're interested in. For example, suppose you have a table frame, and you want to do something special for any field object inside. By definition, the table frame contains the field objects, but so does the page, and so does the form. The form is at the "top" of the containership hierarchy, so it's farthest from the

field objects. The table frame is the lowest-level container, so place your code on the table frame.

In single-record forms (which have no record object) the best approach is to attach general code to the page, and object-specific code to the specific object. As an alternative, you could group the field objects and attach code to the group object's built-in **action** method, or draw a box around the field objects and attach code to the built-in **action** method for the box.

Important While it's true that most actions bubble all the way back to the form and could be processed at the form level, the recommended approach is to encapsulate the intelligence with the object concerned.

This approach of encapsulation will show its merits the first time you start cutting and pasting groups of objects between forms. You could put everything at the form level, especially using **isPreFilter**, but you'd rapidly make some very complex form methods.

Filtering actions by category

As previous examples show, you can use **id** with constants to respond to specific actions. Suppose, though, that you want to handle a certain category of action, and leave the rest to Paradox. You can find out which category an action belongs to by calling the ActionEvent type method **actionClass** and comparing the return value to one of the ActionClasses constants: DataAction, EditAction, FieldAction, MoveAction, or SelectAction. For example, suppose you want to handle all DataActions yourself, but let Paradox handle everything else. You could write code like the following:

```
method action(var eventInfo ActionEvent)
   var
      theClass SmallInt
   endVar
   theClass = eventInfo.actionClass()
   if theClass = DataAction then
      disableDefault
      letMeHandleIt(eventInfo)  ; pass the event to a custom procedure
   endIf                        ; otherwise allow the default handling
endMethod
```

This code calls **actionClass** and stores the returned value in a variable named *theClass*. If the value of *theClass* is equal to the value of the constant DataAction, a subsequent statement passes *eventInfo* to a custom procedure named **letMeHandleIt** which processes all actions in that category.

Example: Working with actions and table frames

The UIObject type provides several constants and methods for working with table frame objects. For example, suppose a form contains a table frame bound to the *Orders* table. You can use the **action** method with constants to move the insertion point and edit data in the table frame, as shown in the following statements.

```
Orders.action(DataNextRecord)  ; moves to the next record
Orders.action(DataPriorRecord) ; moves to the previous record
```

```
Orders.action(DataInsertRecord); inserts a record into the table frame
Orders.action(DataBeginEdit)   ; puts the table frame into Edit mode
Orders.action(DataLockRecord)  ; locks the current record
Orders.action(DataEndEdit)     ; ends Edit mode, posts changes to current record
```

There are many more constants you can use with **action**. Refer to the online ObjectPAL Help for details.

Among the methods in the ObjectPAL run-time library are:

- **attach**
- **locate**
- **nRecords**

These and related methods work like their counterparts in the TCursor type, except that the results are displayed in a table frame. You can use table frames and TCursors together, using the TCursor to work with data behind the scenes, and then, when processing is complete, using the table frame to display the results. Refer to Chapter 18 for examples.

Example: Working with actions and records

The Toolbar has no tool expressly designed for creating a record object, but you can work with records in the user interface, using the following techniques:

- Use the action constant DataArriveRecord to detect any action that forces the data model tables to point at a different record. Any time you scroll, insert, delete, or post a record, or use the mouse to move to a record, or when a visible record is modified across a network, it generates a DataArriveRecord action.

  This action differs from other common actions in that you can't block it. DataArriveRecord is for notification: the action has already happened.

  As an example of when to use DataArriveRecord, suppose you have a form for processing order information, and you want to display only the *CreditCardNumber* field object for credit card orders. To do this, check the value of the *PaymentMethod* field for each record: if the value is "Credit Card", display the *CreditCardNumber* field object. One approach (not recommended) is to write a huge switch statement to catch all the possible actions related to records (DataNextRecord, DataPriorRecord, DataPostRecord, DataInsertRecord, DataToggleEdit, etc.). The recommended approach, shown in the following example, is to use DataArriveRecord in the form's built-in **action** method.

```
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() = DataArriveRecord then
      if PaymentMethod.value = "Credit Card" then
         CreditCardNumber.visible = True
```

```
        else
           CreditCardNumber.visible = False
        endIf
     endIf
  endIf
  endMethod
```

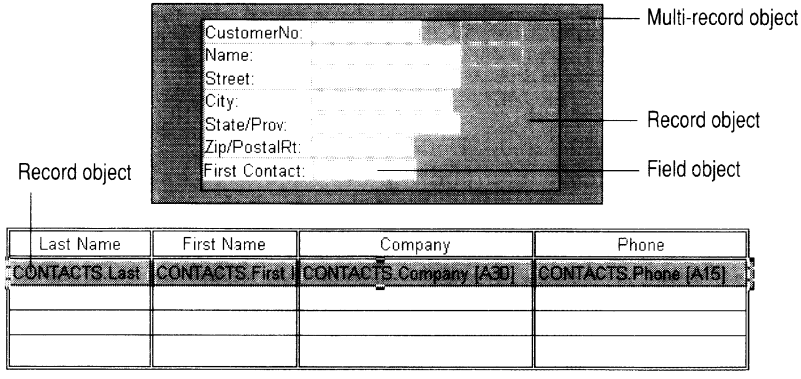
- Create a multi-record object, then define its record layout to display one record across and one record down (also called 1 x 1, pronounced "one by one").
- Use the record object contained in every table frame (the Object Tree provides easy access).

Both techniques enable you to work with a record as you would with any other UIObject. For example, you can inspect a record, set properties, and attach methods. Record properties are listed online. To display the list, open an ObjectPAL Editor window and choose Tools | Properties. Then, from the Objects column, choose Record. The properties appear in the Properties column.

Figure 11.4 shows two compound objects: a 1 x 1 multi-record object bound to the *Customer* table, and a table frame bound to the *Contacts* table. The dark gray region of the multi-record object represents the multi-record object, which contains one record, and the light gray region represents the record itself. The white regions represent field objects contained in the record. In the table frame, the gray region represents the record object.

For more information about record objects, refer to the *User's Guide*.

Figure 11.4 Record objects in a multi-record object and a table frame



Default behavior of built-in methods

This chapter describes the built-in methods for Paradox objects. It discusses the following topics:

- Attaching code to a built-in method and controlling the default behavior of an object.
- Built-in methods for internal events and external events.
- Special built-in methods: describes built-in methods unique to specific objects.
- Built-in object variables: describes ObjectPAL variables for referring to objects in a form.

Important Before you read this chapter, you should be familiar with the ObjectPAL event model, described in Chapter 10.

About built-in methods

Every object in a form (and the form itself) has built-in methods for handling events. Built-in methods have the same names as the events that trigger them. For example, changing a value triggers an object's built-in **changeValue** method, pressing the mouse button triggers the built-in **mouseDown** method, and releasing the mouse button triggers the built-in **mouseUp** method. The behavior of an object is simply the combined effects of its built-in methods.

There are three kinds of built-in methods in ObjectPAL:

- Built-in methods for internal events—events generated internally by ObjectPAL or Paradox.
- Built-in methods for external events—events typically generated by user actions (although they can also be generated by ObjectPAL statements).
- Special built-in methods—methods built into a few specific objects.

This section discusses built-in methods and built-in variables, then presents some simple examples showing how these methods are used in a form while editing data.

Attaching code to built-in methods

You can attach code to any built-in method by opening an ObjectPAL Editor window and typing some code. For example, every design object has a built-in **mouseClick** method that performs some default behavior (described later) when you click that object. To change that behavior, inspect the object, choose Methods from the Properties menu, then choose **mouseClick** from the list of methods to open an ObjectPAL Editor window. Type your code and save it. Now your code executes whenever this object's **mouseClick** method is called.

The built-in code executes too, *after* your code (just before the endMethod keyword).

The built-in code is implicit and executes automatically. But, if you want to change the default behavior—for example, call the built-in code *before* your code, or block it from executing—you can (as described later in this chapter). First, though, you should understand the default behavior for each built-in method.

Descriptions of the built-in methods

Table 12.1 lists the built-in methods for internal and external events, and the special built-in methods. Following the table are descriptions that include when each method is called, the default behavior, and the effects (if any) of an error.

Important This section builds on material presented in Chapters 10 and 11.

Table 12.1 Built-in methods

Internal	External	Special
open	mouseClick	pushButton
close	mouseDown	changeValue
canArrive	mouseUp	newValue
arrive	mouseDouble	
setFocus	mouseRightDown	
canDepart	mouseRightUp	
removeFocus	mouseRightDouble	
depart	mouseMove	
mouseEnter	keyPhysical	
mouseExit	keyChar	
timer	error	
	status	
	action	
	menuAction	

Built-in methods for internal events

The following methods are triggered by internal events—events generated internally by ObjectPAL. Like all events, internal events go to the form first, which dispatches them to the target object. Internal events do not bubble up through the containership hierarchy.

- **open** is called once for every object on the form, starting from the form and working down each container in turn. For every object, the default code for the open method calls the open method for each of its child objects (that is, the objects one level below it in the containership hierarchy). In other words, by default, the form's **open** calls the **open** for each page in the form, and each page's **open** calls **open** for each object on the page, and so on.

Note The form's **open** method opens all tables in the form's data model before any other objects are opened. If aliases are required to open any of the tables, specify them before executing the default code for **open**.

An error from any object prevents a form from opening. Also, explicitly setting an error code in an **open** method's event packet prevents a form from opening. As a general rule, if you attach code to an object's built-in **open** method, you should call **doDefault** before calling any other method or procedure. The call to **doDefault** executes the built-in code so you can be sure the object is completely opened and initialized.

Note Paradox compiles out-of-date forms (forms created in an earlier version of Paradox) before opening them. In this case, tables in the form's data model are opened before any ObjectPAL code executes.

- **close** is called once for every object on a form being closed. By default, a form's **close** method closes all tables attached to the form.
- **canArrive** is called when moving to an object. It asks whether the object (usually a field object) can be made active. Paradox calls this method by working through the object's containers, triggering **canArrive** for each object until it reaches the object itself. At any level of the containership, an error denies permission, blocking the move.

  By default, Paradox blocks **canArrive** for records beyond the ends of a table (except for blank records in Edit mode). The same is true for field objects, which also check the Tab Stop property, blocking moves to field objects that are not tab stops. A crosstab object blocks **canArrive** if the target is not a cell.
- **arrive** is called only after the target and its containers have allowed a **canArrive**. As with **canArrive**, Paradox calls **arrive** on the target object's containers, finishing at the target. Pages, table frames, and multi-record objects all move to the first tab stop object they contain if they are the final destination of the move.

  A successful **arrive** on a record or a field object makes it current (and opens an editing window for the field object, if appropriate).

  arrive can have further effects on a field object, depending on its display type. If it's a drop-down edit list, focus moves to the list. If it's a radio button, focus moves to the first button.

- **setFocus** is called after a successful **arrive** or when focus is returned to the form after going away to another window. This method is called for each of the active object's containers, starting with the outermost container, before it is called for the active object itself.

  On an edit field, the default code for **setFocus** highlights the current selection and starts blinking the insertion point. At this time, also, the object's Focus property is set to True, and the form displays a status message reporting the number of the current record and the total number of records.

  When a button gets focus, a rectangle displays around the label.

- **canDepart** is called when trying to move off any object. This is the place to put code to block a departure: any error blocks the move. Field objects try to post their contents (triggering **changeValue**), and record objects try to commit the current record if changes have been made. If the record is locked, the form calls **action(DataUnlockRecord)** or **action(DataPostRecord)**.

  Switching to another window does not move off an object, it only changes focus.

- **removeFocus** removes the flashing insertion point and highlight (if appropriate) from a field object, and removes the rectangle from a button. The object's Focus property is set to False.

  This method is called for the active object and all its containers, starting with the active object, when the user activates some other window or moves to some other object.

- **depart** is called after all containers of the current object have granted permission to leave the field via **canDepart** and **removeFocus**. Use **canDepart** to block a move, as **depart** is reserved for closing edit regions, repainting, and performing general cleanup.

- **mouseEnter** is called whenever the pointer crosses into an object. It is called only on the transition into the object, not on every move across it.

  By default, field objects set the pointer to the I-beam, and form, page, and button objects set it to an arrow.

  If a button was the last object to receive a click and the mouse button is still down, the button's value toggles between True and False.

- **mouseExit** is called when the pointer leaves an object. An object that sets the shape of the pointer on **mouseEnter** sets it to an arrow in **mouseExit**.

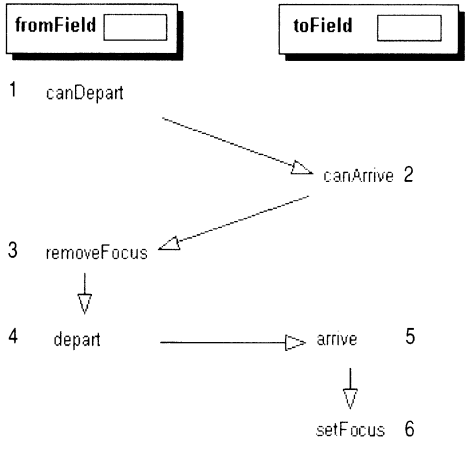
  If a button was the last object to receive a click and the mouse button is still down, its value toggles between True and False.

- **timer** is called each time a timer interval elapses. Use the UIObject method **setTimer** to set timer intervals.

Sequence of execution

Figure 12.1 shows the sequence in which built-in methods execute when you move from one field object to another (for example, by pressing *Tab*). In this example, the field object you're moving from is named *fromField*, the one you're moving to is named *toField*.

Figure 12.1 Moving from one field object to another



This figure shows the sequence of built-in methods that execute when you move from one field object to another field object.

Built-in methods for external events

The following methods are triggered by external events—events typically generated by user actions—although they can also be generated by ObjectPAL statements. Processing for all external events begins with the form, which acts as a dispatcher. An external event bubbles up the containership hierarchy if an object doesn't handle it.

Important For most objects and most built-in methods, the default behavior is to pass the event up the containership hierarchy. If an object does something different, is it noted in the following descriptions.

- **mouseDown** is called when the logical left mouse button is pressed. The event packet for this method contains the mouse coordinates in twips, relative to the last object whose **mouseEnter** method was called.

  An active field object enters Field View, positions the insertion point and begins a drag-selection.

  When the form handles a **mouseDown**, it calls **mouseExit** for all objects no longer under the mouse, and calls **mouseEnter** for all objects now under the mouse. The form then dispatches the **mouseDown** to the object the mouse was pointing at.

  This method toggles a button's value between True and False.

- **mouseUp** is called when the left mouse button is released. It's called for the last object to receive a **mouseDown**, even if the button is released outside the object, so the object always sees the **mouseDown/mouseUp** pair.

  An active field object ends the selection; a field object that is not active performs a **self.moveTo()**.

When the form handles a **mouseUp**, it calls **mouseExit** for all objects no longer under the mouse, and calls **mouseEnter** for all objects now under the mouse. The form then dispatches the **mouseUp** to the object that received the last click.

This method toggles a button's value between True and False. If **mouseUp** is called and the pointer is inside a button, it triggers the button's **pushButton** method.

- **mouseDouble** is called when the left mouse button is double-clicked. In Windows convention, a **mouseDown** and **mouseUp** are delivered first.

  Field objects enter field view on a **mouseDouble**.

  When the form handles a **mouseUp**, it calls **mouseExit** for all objects no longer under the mouse, and calls **mouseEnter** for all objects now under the mouse. The form then dispatches the **mouseDouble** to the object that received the last click.

- **mouseRightDown, mouseRightUp, mouseRightDouble** These methods are duplicates of the three above, but they refer to the right mouse button.

  In addition, the following field objects display a pop-up menu when they get a **rightMouseUp**: formatted memo, graphic, OLE, and unbound (undefined).

- **mouseClick** is called when the logical left mouse button is pressed and released when the pointer is inside the boundaries of an object. **mouseClick** is not called if the user moves the mouse outside the object before releasing the mouse button.

  mouseClick is actually generated from within the **mouseUp** method.

  The mapping from **mouseUp** to **mouseClick** happens at the first container object which does something with **mouseUp**. In other words, **mouseUp** in a box bubbles to its container, and so forth. Only the field object, the button object, the list object, and the form intercept **mouseUp**, so those are the spots where the translation occurs. If you click an object inside a button, that object's **mouseClick** will be called. If that object allows the default (bubbling), then the button will ultimately receive that **mouseClick**, triggering its own **pushButton**. In this way, you can have code execute on objects you click inside the button, but still trigger the button's **pushButton** method. Setting the error code in **mouseUp** will inhibit the **mouseClick**, and setting the error code in **mouseClick** will inhibit a **pushButton**.

- **mouseMove** is called whenever the mouse moves within an object. The event packet for this method contains the coordinates of the pointer (in twips).

  An active edit field checks the state of *Shift*. If *Shift* is down (physically or logically), the selection is extended. An active graphic field scrolls the graphic, if necessary.

  When you press and hold the mouse button inside an object, **mouseMove** is called until you release it, even when the pointer moves outside the object.

  When the form handles a **mouseMove**, it calls **mouseExit** for all objects no longer under the mouse, and calls **mouseEnter** for all objects now under the mouse.

- **keyPhysical** is called when a key is pressed and each time a key autorepeats. It goes to the form first, and the form dispatches it to the active object. Then, the active object's built-in code sorts out whether a keystroke represents an action or a character to display in a field, and calls the appropriate **action** or **keyChar** method (discussed later in this list).

For example, suppose a field object within a table object is active, and the user presses *Enter*. The keystroke triggers **keyPhysical**, which interprets it as a request for an action and maps it to **action(FieldEnter)**, which in turn triggers the built-in **action** method. In contrast, when the user presses *K*, the keystroke triggers **keyPhysical**, which interprets it as a character and triggers the **keyChar** method.

Technically, the event packet for **keyPhysical** contains information from the Windows WM_KEYDOWN message and an optional WM_CHAR. Therefore, it provides both the virtual key code as well as the ANSI character. Although you can attach code to this method, it's best if you don't, except for special character handling. For example, if you want to intercept the F9 key explicitly—rather than handle the eventual **action(DataToggleEdit)**—this is the method to use.

- **keyChar** is called when a **keyPhysical** is not interpreted by Paradox. That is, a **keyChar** gets called for every **keyPhysical** that *does not* map to an action (see **action** in this list). It goes to the form first, and the form dispatches it to the active object.

  When editing a field, the system locks the record before inserting the first character.

  If a button receives a **keyChar** equal to pressing *Spacebar* (for example, **keyChar(VK_SPACE)**), it calls the button's **pushButton** method.

- **menuAction** is called whenever the user chooses an item from a menu (or clicks a Toolbar button that executes a menu action). It goes to the form first, and the form dispatches it to the active object. See Chapter 10 for more information.

- **error** is called when an error occurs. By default, objects (except the form) pass errors to their containers. You can attach code to the default method to make an object handle an error, or pass it, or both. For more information about errors and error handling, refer to Chapter 30.

- **status** is called before a message is displayed in one of the areas in the status bar. Among other things, you can attach code to the built-in **status** method to redirect messages to other areas, or to change the text of the message. See Chapter 10 for more information.

- **action** is called when **keyPhysical** maps some keystroke to an action, when **menuAction** maps a menu choice to an action, or when other methods want some action performed. It goes to the form first, and the form dispatches it to the target object. For example, by default, pressing *F2* in a field triggers **action(EditToggleFieldView)** after its **keyPhysical** method executes, and clicking the Forward navigation button triggers **action(DataNextRecord)** after its **menuAction** method executes.

  action is a very important method. It is discussed in detail in Chapter 13.

Special built-in methods

Some objects have additional built-in methods. They are described in this section.

pushButton

Defined only for button objects and fields displayed as list boxes, **pushButton** is called when the user releases the mouse on a button. This method is actually not called directly by the form, but by the default **mouseUp** method for buttons. You can also call this method directly to accomplish the normal action associated with pressing a button object.

By default, buttons change their appearance when clicked. For example, a push button pushes in and pops out, check boxes check or uncheck, and radio buttons push in or pop out. Focus moves to a button when you click it (unless its Tab Stop property is set to False).

When a button's Tab Stop property is set to True and the button is the active object, there are two ways to trigger its **pushButton** method using the keyboard:

- Press *Spacebar*. The button keeps focus.
- Press *Enter*. Focus moves to the next object in the tab order.

changeValue

Defined only for field objects, **changeValue** asks for permission to change the value of a field. It is called *before* the value is stored, so you can check the value and do something with it (such as performing additional validity checks). It is not called when someone changes a value across a network or through a lookup with fill-all-corresponding.

The following statement triggers *Quant*'s **changeValue** method, even if *Quant* is already 10, and it triggers it immediately, without waiting for the method to finish executing.

```
Quant = 10
```

newValue

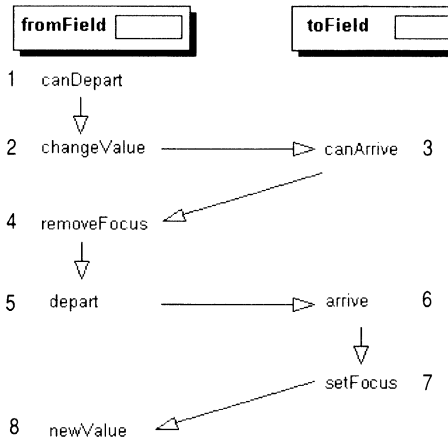
Defined only for field objects, **newValue** is called to report that a field object has a new value. For example, when scrolling through a table, moving to the next record triggers **newValue**. When a field is displayed as radio buttons, **newValue** is called when you click a button. Note that simply typing into a field object does not trigger **newValue** (but does set the Touched property to True). In any case, **changeValue** is not called until you try to move off the field object or otherwise try to commit changes. Also, a form's **open** method triggers **newValue** for each field object in the form.

Labeled and unlabeled field objects

Figure 12.2 shows the sequence in which built-in methods execute when you move from one field object (labeled or unlabeled) to another after editing a value. The field object you're moving from is named *fromField*; the one you're moving to is named *toField*.

Note **newValue** is called when Paradox needs to refresh the value of the field object (in this case, to update the display). Calls to **newValue** are not part of the **canDepart** sequence; that's why **newValue** is listed after all the others in Figure 12.2. However, **newValue** is not necessarily the last method to execute.

Figure 12.2 Moving from field object to field object after editing a value

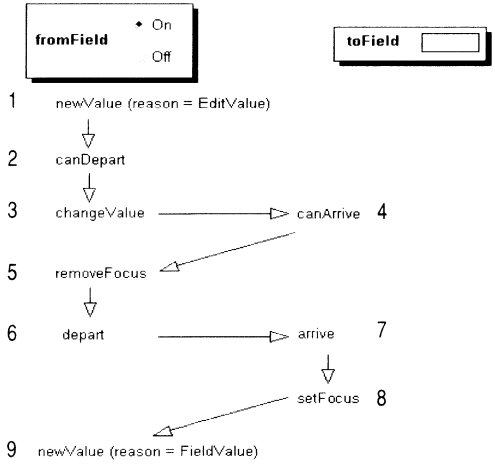


This figure shows the sequence of built-in methods that execute when you move to another field object after editing a value in the first field object.

Radio buttons and lists

Figure 12.3 shows the sequence in which built-in methods execute when you move from one field object (radio buttons, list, or drop-down edit list) to another after editing a value. The sequence is the same as for regular fields, except for an additional **newValue** call when you choose a radio button or a list item. The table also gives the ValueReason constant for each **newValue**. The field object you're moving from is named *fromField*; the one you're moving to is named *toField*.

Figure 12.3 Moving from a radio button or a list to a field object



The first **newValue** method is triggered when you click the radio button (or choose an item from a list). All other methods are triggered when you initiate the move.

Using changeValue with field objects

The built-in code for **changeValue** commits the changes to the value; until the built-in code executes, Paradox uses the old, unchanged value. For example, suppose a field

object has a value of 10, and you move to it and enter a value of 23. Then, when you move off that field object, you trigger its **changeValue** method, to which the following code has been attached:

```
method changeValue(var eventInfo ValueEvent)
   msgInfo("before the change", self.value) ; displays 10
   doDefault
   msgInfo("after the change", self.value) ; displays 23
endMethod
```

When this method executes, the first dialog box displays the old value, 10, because the built-in code has not yet executed. Then, the call to **doDefault** executes the built-in code, which commits the changed value, and the second dialog box displays the changed value.

Within an object's **changeValue** method, you can use the ValueEvent method **newValue** (not to be confused with the built-in **newValue** method, discussed previously) to get the incoming value before the built-in code executes. For example, suppose as before that a field object has a value of 10, and you move to it, enter a value of 23, and trigger its **changeValue** method, to which the following code has been attached:

```
method changeValue(var eventInfo ValueEvent)
   msgInfo("before the change", self.value) ; displays 10
   msgInfo("the new (incoming) value", eventInfo.newValue()) ; displays 23
   doDefault
   msgInfo("after the change", self.value) ; displays 23
endMethod
```

The first dialog box displays the old, unchanged value. The second dialog box calls **eventInfo.newValue** to display the new, incoming value—but that value has not yet been committed. The call to **doDefault** executes the built-in code, which commits the change, and the third dialog box displays the changed value.

You can block an attempted change to a value by calling **eventInfo.setErrorCode** and specifying a nonzero value. You can also alter the incoming value (say, for example, to round up to the nearest dollar amount), using **eventInfo.setNewValue**.

In the following example, assume that a form contains a multi-record object bound to the *Orders* table, and a button named *undoLast*. When a change is made to a field, the **depart** and **changeValue** methods on the form track the field's name and original value. If a change can be "undone," the font color of the label (a text box named *UndoLastLabel*) on *undoLast* changes to black; otherwise, the label is dimmed (dark gray type on a light gray button). When the user clicks the *undoLast* button, the button's **pushButton** method checks if its own label is dimmed; if not, it undoes the most previous change to the field. This code is attached to the Var window for the form:

```
Var
  targObj,                  ; gets handle to current object
  lastTargetField  UIObject ; tracks most recently changed field
  lastValue        AnyType  ; stores last field value
endVar
```

This code is attached to the **depart** method for the form:

```
method depart(var eventInfo MoveEvent)
if eventInfo.isPreFilter()
```

```
  then
    ;code here executes for each object in form
    ; this code tracks the contents and field name of the most
    ; recently changed field in the same record
    eventInfo.getTarget(targObj)              ; targObj declared in Var window
    if targObj.class = "field" then
      if targObj.Touched = True then
        ; code in changeValue method stores last field contents
        lastTargetField.attach(targObj)       ; store handle to field
        UndoLastLabel.Font.Color = Black      ; make button label Black
      endIf
    else    ; if moving to a new record, disable undo
      UndoLastLabel.Font.Color = DarkGray     ; gray button label
    endIf
  else
    ;code here executes just for form itself

endIf
endMethod
```

This code is attached to the **changeValue** method for the form:

```
method changeValue(var eventInfo ValueEvent)
if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
    ; this code picks up the original value of a field about to change
    if targObj.Touched = True then
      lastValue = String(targObj.value)           ; store lastValue
    endIf
  else
    ;code here executes just for form itself

endIf
endMethod
```

This is the code for the **pushButton** method of *undoLast*:

```
method pushButton(var eventInfo Event)
if undoLastLabel.Font.Color = DarkGray then
  ; if button is dimmed, there is no change for this record to undo
else  ; if button isn't dimmed, go ahead and change previous field
  msgInfo("Status", "Changing value for " +
                    lastTargetField.name + " to " +
                    lastValue)        ; tell the user what's going on
  lastTargetField.Value = lastValue   ; restore original value
  undoLastLabel.Font.Color = DarkGray  ; value can't be undone twice!
endIf
endMethod
```

Controlling the default behavior

The following basic language elements control when (and if) the built-in code executes:

- **doDefault** executes the built-in code immediately and prevents the built-in code from executing again at the end of the method.
- **disableDefault** blocks the built-in code from executing.
- **enableDefault** allows the built-in code to execute at the end of a method.
- **passEvent** passes the event to an object's container, whether or not the object has handled the event. It does not affect whether the built-in code executes at the end of the method.

Important You can use these elements only in code attached to built-in methods, not in custom methods or custom procedures.

For example, a new field object's built-in **keyChar** method is

```
method keyChar(var eventInfo KeyEvent)

endMethod
```

Characters appear when you type them into this field object, because the built-in code executes implicitly just before the endMethod keyword.

doDefault

If you attach the following code to the method you might expect each character you type to show up twice: once for your explicit call, and once for the implicit call, but actually, it happens only once.

```
method keyChar(var eventInfo KeyEvent)
   doDefault
endMethod
```

An explicit call to **doDefault** blocks the implicit call to the built-in code.

Important Although the compiler allows multiple calls to **doDefault**, only the first call has any effect; the others are effectively ignored.

disableDefault

You get the same effect—that is, the implicit code doesn't execute—when you use **disableDefault** or **enableDefault**.

For example, the following code calls **disableDefault** to prevent the field object from displaying the character X:

```
method keyChar(var eventInfo KeyEvent)
   if eventInfo.char() = "X" then
      disableDefault
   endIf
endMethod
```

Important A **return** statement in a built-in method executes the built-in code immediately before executing the **return** unless you call **disableDefault** to block it.

When you want to handle specific cases yourself and let ObjectPAL handle the rest, call **disableDefault** before the **switch** block, then call **enableDefault** or **doDefault** as appropriate in the body of the block. For example, the following code allows the default code to execute only when the action ID is DataNextRecord or DataPriorRecord:

```
method action(var eventInfo ActionEvent)
   disableDefault ; Assume we are not going to do the default

   ; But if the Action Id is either DataNextRecord or DataPriorRecord
   ; then do the default.

   switch eventInfo.id()
      case DataNextRecord  : enableDefault
      case DataPriorRecord : enableDefault
   endSwitch
endMethod
```

This technique is useful for handling menu choices. For example,

```
method menuAction(var eventInfo MenuEvent)
   var theChoice String endVar
   disableDefault
   theChoice = eventInfo.menuChoice()
   switch
     case theChoice = "Open" : doOpen() ; do custom method, don't do default
     case theChoice = "New"  : doNew()  ; do custom method, don't do default
     otherwise               : enableDefault
   endSwitch
endMethod
```

Important Do not use **disableDefault** in methods for internal events. Blocking the default code for these methods can cause unexpected results. For example, disabling the default of an **open** method might prevent the tables attached to a form from opening, or disabling the default in a **setFocus** method could suppress the highlighting of a field value. Instead, use **setErrorCode** to set a nonzero error value. The default code checks the error value as part of normal execution; a nonzero error value indicates an error, and the default code takes it into account.

For example, suppose the following code is attached to a field object's built-in **canDepart** method. When this code executes, it compares the date value in the field object with today's date. If the date value in the field object is later than today's date, a dialog box displays a message to inform the user, and the call to **setErrorCode** uses a predefined ObjectPAL constant (CanNotDepart) to specify a non-zero error value. When the default code executes, it "sees" this value and prevents the insertion point from leaving the field object.

```
method canDepart(var eventInfo MoveEvent)
   if Date(Self.value) > today() then
      msgStop("Stop", "That date is in the future.")
      eventInfo.setErrorCode(CanNotDepart)
   endIf
endMethod
```

Processing an error value is *not* the same as disabling the default code. If you were to use **disableDefault** in this example, the move would still happen, because permission is

assumed granted until explicitly denied. For more information and examples, refer to Chapter 10.

enableDefault

enableDefault allows the built-in code to execute at the end of a method. The difference between **doDefault** and **enableDefault** is important: **doDefault** executes the built-in code immediately, and **enableDefault** sets a flag so the built-in code executes at the end of the method. For example, if you attach the following code to a field object, when you type a character into the field, Paradox will wait three seconds, beep, then display the character:

```
method keyChar(var eventInfo KeyEvent)
   enableDefault
   sleep(3000)
   beep()
endMethod
```

In contrast, the following code makes Paradox display the character *before* it sleeps and beeps:

```
method keyChar(var eventInfo KeyEvent)
   doDefault
   sleep(3000)
   beep()
endMethod
```

Important Calling **doDefault** sets a flag that prevents the built-in code from executing at the end of the method.

You can use these keywords anywhere in a built-in method, but keep in mind that an object's behavior may change depending on where the keywords are used. For example, suppose this method is attached to an undefined field:

```
method keyChar(var eventInfo KeyEvent)
   eventInfo.setChar("K")
   doDefault
endMethod
```

When you run this method and type characters into the field, it displays the letter K, no matter what character you type, because the method sets the character to K and *then* calls the built-in code, which displays the character in the field. If you reverse the order of the statements, the results are different:

```
method keyChar(var eventInfo KeyEvent)
   doDefault
   eventInfo.setChar("K")
endMethod
```

In the previous example, the call to **doDefault** executes the built-in code before the character gets set to K, so the field behaves normally.

passEvent

A **passEvent** statement says, in effect, "Suspend execution, and send the information about this event to my container." It has no effect on the built-in code. If the built-in code is enabled, either implicitly or by using **enableDefault**, it executes at the end of the method. If **passEvent** is called and the built-in code is disabled by **disableDefault** or called by **doDefault**, it doesn't execute at the end of the method.

A call to **passEvent** suspends execution of the calling method, and immediately triggers the appropriate method in the calling object's container. By default, the calling method resumes execution when the container's method finishes.

To see how this works, follow these steps to create and run a simple form:

1 Create a new form.

2 Place a large box on the page.

3 Place a field object inside the box, so the box contains the field object.

4 Attach the following code to the field object's built-in **keyChar** method:

```
method keyChar(var eventInfo KeyEvent)
   msgInfo("keyChar", "Calling container's keyChar method.")
   passEvent
   msgInfo("keyChar", "We're back.")
endMethod
```

5 Attach the following code to the box's built-in **keyChar** method:

```
method keyChar(var eventInfo KeyEvent)
   self.color = self.color + 222
   sleep(1000)
endMethod
```

6 Run the form, and type a character into the field object.

Here's what happens when you run this form and type a character into the field object: first, the code attached to the field object's built-in **keyChar** method displays a dialog box. Then, the call to **passEvent** triggers the **keyChar** method of the field object's container, the box. The field object's **keyChar** method suspends execution. When the box's **keyChar** method executes, the box changes color and Paradox sleeps for one second. Then, when the box's **keyChar** method has finished executing, the field object's **keyChar** method resumes execution and displays another dialog box.

For more information, refer to the entries for **doDefault**, **disableDefault**, **enableDefault**, and **passEvent** in the online ObjectPAL Help.

Built-in object variables

Along with built-in methods, ObjectPAL provides built-in object variables:

Self

Self refers to the object to which the currently executing code is attached. For example, when the following statement executes in the **mouseEnter** method attached to *theBox*, *Self* refers to *theBox*.

```
self.color = Red
```

But, suppose a method attached to *theBox* calls a custom method named **changeColor** attached to the page. Suppose the code for **changeColor** is

```
method changeColor()
   self.color = Blue
endMethod
```

When the method attached to *theBox* calls **changeColor**, the page turns blue, not *theBox*. Why? Because *Self* refers to the object to which the code is attached—regardless of which object actually called the code—and in this case, the code is attached to the page. The single exception to this rule is when *Self* appears in a statement in a library. In this case, *Self* refers to the object that called the library routine.

When an event occurs, *Self* and **eventInfo.getTarget** may refer to the same object, but as events bubble up the containership chain, the target remains fixed while *Self* changes to refer to the object executing the method.

Note *Self* always refers to a UIObject, not to the object's value or the object's name.

Container

Container is the object that contains *Self*. For example, suppose a box contains a button, and the button's **pushButton** method is as follows:

```
container.color = Red
```

When this code executes, the box turns red, because the box contains the button, and the button is executing the code.

Active

Active is the currently active object—the last object to receive a **moveTo**. Typically, the active object is highlighted.

When an active object is set to respond to keyboard events, it is said to have *focus*. Even when focus is removed from an object (for example, to activate another form), *Active* still refers to that object. Only when someone moves off that object is *Active* reset to the new object. Don't underestimate the importance of *Active*, since general routines can be written to operate on the active object without really knowing which one it is. For example, suppose a form contains two table frames, each bound to a different table. The following statement automatically operates on the active table frame:

```
active.action(DataNextRecord)
```

Subject

Subject specifies which object a custom method should operate on. For example, suppose a page in a form has a custom method **setColor**, and this is the code for **setColor**:

```
method setColor()
   subject.color = red
endMethod
```

Any object on that page can make the following call, and the object named *someObject* will turn red. When **setColor** executes, it replaces *Subject* with *someObject*.

```
someObject.setColor()
```

For more information and examples, refer to Chapter 13.

LastMouseClicked

LastMouseClicked refers to the last object to receive a **mouseDown**. It is reset when the mouse button is released, but only after the object has been given a chance to do its **mouseUp**.

LastMouseRightClicked

LastMouseRightClicked is the same as *LastMouseClicked*, but for the right mouse button.

Using properties related to built-in methods

This section describes certain methods and properties related to built-in methods. The first part deals with properties common to all UIObjects, the second with field object properties, and the third with record object properties.

Note The properties discussed here represent a small subset of the properties available to ObjectPAL. Appendix B contains a complete list of the properties for each object.

All UIObjects

- *Arrived* is True if the active object's **arrive** method has been called. If Arrived is True for an object, it's True for the object's containers, too. For example, if *fieldOne* is in *pageOne*, and *fieldTwo* is in *pageTwo*, then when Arrived is True for *fieldOne*, it's True for *pageOne*, but not for *pageTwo*. Similarly, when Arrived is True for *fieldTwo*, it's also True for *pageTwo*, but not for *pageOne*. Arrived is a read-only property (you can get its value, but you can't set it).
- *Focus* is set to True when the active object's **setFocus** method is called, and set to False when the active object's **removeFocus** is called. As with Arrived, if Focus is True for an object, it's True for the object's containers, too. Focus is a read-only property.

Field objects

- *Editing* is set to True when a field has an edit window open: When you edit a field object, Paradox creates a text object over the field object, and this text object is where you actually type values. Paradox deletes the text object when the edit is complete. This property is set by the field object's built-in **arrive** method. Editing is a read-only property.
- *Touched* is set to True when the user changes the data in a field. This property determines whether to write the contents of the edit window to the field. When Touched is False, there's no need to write the same data back to the field. Touched is a read-only property, and you can only read it when Editing is True.

Record objects

- *BlankRecord* returns True when a record is blank; otherwise, it returns False.
- *Inserting* returns True when a record has just been inserted into the table; otherwise, it returns False.
- *Locked* returns True if a record is locked; otherwise, it returns False.
- *RowNo* is an integer value representing the row number (starting from 1) of a record displayed in a table frame or multi-record object. For example, the second row displayed in a table frame has a *RowNo* of 2, no matter what record of the table is actually displayed in that row. See also the *RecNo* property for table frame and multi-record objects.

Note When testing *RowNo* for a multi-record object, remember that records can be numbered from top to bottom or left to right.

- *Touched* is True when the record has been changed but not posted (committed). Touched is a read-only property.

Table frames and multi-record objects

- *RowNo* is an integer representing the row number of the active record. It provides an easy way to find out which relative record is active. RowNo returns the row number relative to the table frame, not the underlying table.
- *SeqNo* is an integer representing the record number of the underlying table.

Property lists

To display the list, open an ObjectPAL Editor window, choose Tools | Properties, and select the object and the property of interest.

For example, to display the list of Box properties, open an ObjectPAL Editor window, choose Tools | Properties, and from the Objects column, choose Box. Box properties are listed in the Properties column. To display the valid values for a property, choose the property name (for example, Color). The values display in the Values column.

UIObject properties are listed in Appendix B. Properties are also listed online.

Part

Programming tasks

This part of the manual contains the following chapters:

- Chapter 13, "UIObjects: Creating the user interface," discusses the objects you can place on a form using tools in the Toolbar.
- Chapter 14, "Working with menus," presents advanced techniques for working with menus, pop-up menus, Paradox built-in menus, the Toolbar, and Windows system menus.
- Chapter 15, "Performing calculations," describes how to use ObjectPAL in calculated fields.
- Chapter 16, "Manipulating the data model," explains how to get and set information about which table or tables a form is bound to.
- Chapter 17, "Performing table operations," shows how to perform table-level operations and specify table attributes before you actually open the table.
- Chapter 18, "Using TCursors," explains how to manipulate data at the table level, record level, and field level without having to display the table.
- Chapter 19, "Working with databases," explains how to use Database variables to specify and work with a databases, and how to use aliases.
- Chapter 20, "Displaying output," shows how to use ObjectPAL display managers to present information to a user.
- Chapter 21, "Printing reports," shows how to use Report type variables to control a Report window.
- Chapter 22, "Querying tables," explains how use ObjectPAL to create and execute queries from methods just as if you were using Paradox interactively.

- Chapter 23, "Managing sessions," explains how to use ObjectPAL to open other sessions from within an application.
- Chapter 24, "Working with the file system," shows how to use ObjectPAL's FileSystem type methods for working with disk files, drives, and directories.
- Chapter 25, "TextStream: Working with text files," explains how to read from or write to an ANSI text file.
- Chapter 26, "Developing multi-form applications," discusses the key concepts for building multi-form applications.
- Chapter 27, "Linking with DDE," explains how to use this Windows protocol to share data with other DDE-compliant applications.
- Chapter 28, "Using libraries," explains how to use libraries to store and maintain frequently used routines and share code among several forms.
- Chapter 29, "Creating and playing scripts," explains how to use a script to execute code without opening and displaying a form window.
- Chapter 30, "Handling run-time errors," explains default system behavior for error conditions, and provides information about building customized error-handling into forms.
- Chapter 31, "Ensuring data security and integrity," explains how to use passwords to help you guarantee the security of your data, and how Paradox automatic locks help you provide a balance between the goals of data integrity and concurrent access in interactive use.
- Chapter 32, "Handling other multiuser issues," explains how to design applications that work well in a multiuser (or network) environment.
- Chapter 33, "Packaging and delivering an application," presents the concepts and procedures necessary to package Paradox applications for delivery to end users.

UIObjects: Creating the user interface

Design objects create the user interface for an application; anything you can place in a form is a design object. There are three types of design objects: menus, pop-up menus, and UIObjects. UIObjects include objects created with the design tools on the Toolbar, record objects, the underlying page of a form, and more. UIObjects are the only objects with built-in methods, the only objects you can attach code to, and the only objects that respond to events. Table 13.1 gives a description of each design object.

Note This category also includes the form, even though the Form type itself is a display manager, not a design object. A form has built-in methods, and you can attach code to these built-in methods. Also, a form responds to events.

Table 13.1 Design objects

Type	Description
UIObject	Objects placed in a form to create the user interface. The member objects are bitmap, box, button, crosstab, ellipse, field object, form, graph, line, multi-record object, OLE object, page, record object, table frame, and text box.
Menu	Menus that appear in the application menu bar.
PopUpMenu	Menus that appear in a form, not in the application menu bar.

This chapter discusses UIObjects. Menus and Pop-up menus are covered in Chapter 14.

Building blocks of the user interface

The UIObject type (UI stands for user interface) includes all the objects you can place using tools in the Toolbar (described in the *User's Guide*), the underlying page, the form, and more. A complete list of UIObjects is Tools | Properties. The UIObjects are listed in the Objects column.

This section discusses

- UIObject properties
- Using the Value property to get and set data
- Using properties related to built-in methods
- Properties: special cases
- Compound objects
- Events and UIObjects
- Actions and UIObjects
- Working with actions and table frames
- Working with actions and records
- UIObjects: Shortcuts and special cases

Properties

All objects have *properties* (for example, text, color, pattern, font, name, size, and position) in addition to methods. Properties are like variables, but they are predefined as parts of objects—that is, every object comes with built-in properties. When you inspect an object, you display a list of its properties; you can use ObjectPAL to access those properties. Every property you can set from the menu can also be set using ObjectPAL, and you can set many properties that aren't listed in menus.

Note Setting a property does not generate an event, *except* when you set a field object's Value property (which triggers its **newValue** method).

Most properties affect how an object looks. For example, the following statement sets the Text property of the text box *myText*, making it display the word "Cancel":

```
myText.text = "Cancel"
```

For another example, suppose a form contains a field object named *result*. You could use the Color property to specify a data-dependent color for the characters, like this:

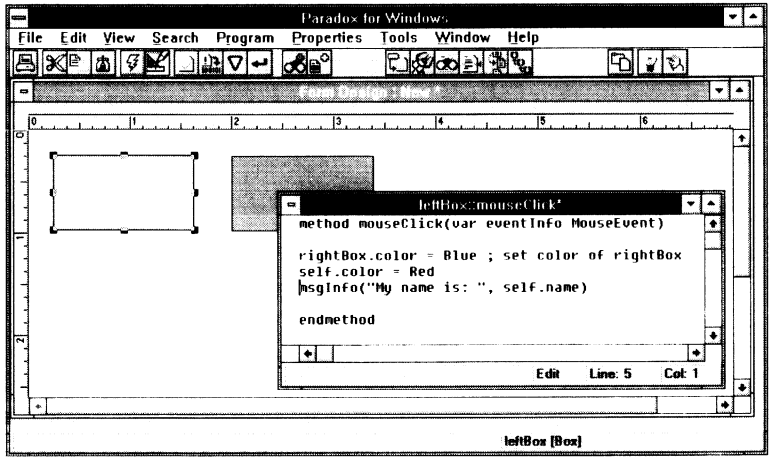
```
if income > outgo then
   result.font.color = Green ; use green characters if there is a profit
else
   result.font.color = Red ; if no profit, use red characters
endIf
```

Objects can set their own properties and the properties of other objects. For example, see Figure 13.1, which shows a form containing two boxes. Attached to the box on the left (*leftBox*) is a method that changes its own Color property and the Color property of the box on the right (*rightBox*), and then displays its name (Name is a property) in a dialog box.

```
method mouseEnter (var eventInfo MouseEvent)
   rightBox.color = Blue      ; sets the color of rightBox
   self.color = Red           ; sets the color of this object (leftBox)
```

```
    msgInfo("My name is: ", self.name) ; displays this object's name (leftBox)
endMethod
```

Figure 13.1 A form containing two boxes



The method in the Editor window executes when you move the pointer into the box on the left. The method sets the color of the right box to blue, sets the color of the left box to red, and displays a dialog box containing the name of the left box.

For more information about object properties, see Appendix B. Also, object properties and property values are listed online.

Using properties

Through ObjectPAL, you have access to many more object properties than you do using Paradox interactively. Use dot notation to access a property directly, as in this example:

```
if myPage.color = Red then ; direct access to color property
   myPage.color = Green
   myBox.visible = No ; makes myBox invisible
endIf

; the next 2 lines both return Strings
propVisible = myBox.visible
propColor = myBox.color
myField.font = "System"  ; sets the font of myField to System
myField.font.color = Red ; sets the font color to Red
thatBox.visible = not(thatBox.visible) ; toggles between visible and invisible
```

Data types of properties

Each property has a native data type. For example, the Color property's native data type is LongInt (long integer). You don't have to remember integer values to set colors, though, because ObjectPAL has predefined constants that represent many (but not all) property values. So, the next statement works because ObjectPAL has defined the word Blue to be a constant whose value is 16,711,680 (the integer value for the color blue).

```
thatBox.color = Blue
thatBox.Frame.Style = Shadow
```

Note You can also assign properties using a property constant in a quoted string, or by assigning the value of a constant to a variable. For example, all of the following statements are valid:

```
var theProp AnyType endVar

thatBox.Frame.Style = ShadowFrame    ; uses a constant
thatBox.Frame.Style = "ShadowFrame"  ; uses a constant as a quoted string
theProp = ShadowFrame
thatBox.Frame.Style = theProp        ; uses a variable assigned to a constant
```

These techniques can be useful when you're getting property values from an outside source, for example, from a table or a text file.

For some properties (for example, Text, LabelText, and TypeFace), the native type is String, so there are no constants. Instead, you just assign a value directly, as in

```
textBox.Text = "Enter the part number."
myButton.LabelText = "Cancel"
myFieldObject.Font.TypeFace = "Times"
```

For a complete list of properties and their data types, see Appendix B.

Note The maximum length of a string returned as a property value is 255 characters, except for the Text property, which can be 32,767 characters.

Using Self

Notice that you can use the object variable *Self* in a method: *Self* refers to the object to which the currently executing code is attached. The following statement executes in the **mouseEnter** method attached to *leftBox*, so *Self* refers to *leftBox*:

```
self.color = Red
```

But, suppose a method attached to *leftBox* calls a custom method named **changeColor** attached to the page, and the code for **changeColor** is

```
method changeColor()
   self.color = Blue
endMethod
```

When the method attached to *leftBox* calls **changeColor**, the page turns blue, but *leftBox* doesn't. Why? Because *Self* refers to the object to which the code is attached—regardless of which object actually called the code—and in this case, the code is attached to the page.

See Chapter 12 for more information about using *Self* and other built-in variables.

Note The single exception to this rule is when calling a method in a library. The code in a library executes on behalf of the object that called it. For example, when a box calls a library method, statements that use *Self* refer to the box, not to the library. Refer to Chapter 28 for more information.

Also, *Self* is not the same as *Subject*. Subject is an object variable that refers to the caller of a custom method. See Chapter 6 for more information.

Using the Value property to get and set values

One of the most important object properties is *Value*. Using the Value property, you can get and set data in a field of a table like this:

```
x = tableID.fieldID.value     ; gets the field's value
tableID.fieldID.value = z     ; sets the field's value
```

The *.value* part is optional. You can include *.value* as needed for clarity, or omit it for faster typing. In other words, the following statements are identical in ObjectPAL:

```
x = tableID.fieldID

x = tableID.fieldID.value
```

Suppose you want to get the first name of a customer named Jones. You can search the Last Name field of the *CUSTOMER* table frame for Jones, then get the value of the *First_Name* field object, like this:

```
var
   firstName String
endVar
if CUSTOMER.locate("Last Name", "Jones") then ; if Jones is in the table then
   firstName = CUSTOMER.First_Name.value      ; put the value of
                                              ; the First_Name field object
endIf                                         ; into the variable firstName
msgInfo("First name", firstName)              ; display the name in a dialog box
```

For text objects, including field labels and button labels, setting the Value property specifies the text to be displayed in the object. You can get the same effect by setting the Text property. For example, the following statements have the same effect:

```
myText.Value = "Hello"

myText.Text = "Hello" ; these statements are equivalent
```

Buttons have a Value property, too. When a button's value is True, it appears to be pushed in (or checked, if it's displayed as a check box or a radio button); a value of False makes it pop out (or unchecks it). These statements make the button named *thatButton* push in and pop out, but *do not* trigger its **pushButton** method:

```
thatButton.value = True                   ; make button push in
sleep(1000)
thatButton.value = False                  ; make button pop out
```

Note Setting a field object's Value property triggers its **newValue** method, and its **changeValue** will be triggered when the new value is posted.

Properties: Special cases

This section describes some alternative ways to work with and set properties of certain design objects. It describes how to use

- Methods (instead of dot notation) to set properties
- A button's LabelText property to change its text label while an application is running
- Properties of field objects

Using methods to set properties

You can use the UIObject type methods **getProperty**, **getPropertyAsString**, and **setProperty** as an alternative to dot notation. These methods are convenient because they bypass the issue of data types, and handle all property values as strings. For example, suppose a form contains a box named *thatBox*. The following code manipulates its Frame.Style property:

```
var
   propName, propVal String
endVar
propName = "frame.style"
propVal = thatBox.getPropertyAsString(propName)
if propVal <> "ShadowFrame" then
   thatBox.setProperty(propName, ShadowFrame)
endIf
```

Properties of button objects

Buttons have a property, LabelText, that lets you specify the text in a button's label without naming the label (it's a text object) and addressing it directly. For example, the following method checks and sets the LabelText property for each of three buttons:

```
var
   buttons Array[3] String
   i SmallInt
endVar
buttons[1] = "btnOne"
buttons[2] = "btnTwo"
buttons[3] = "btnThree"
for i from 1 to buttons.size() ; buttonBox is a box that contains the buttons
   if buttonBox.(buttons[i]).labelText = "On" then
      buttonBox.(buttons[i]).labelText = "Off"
   else
      buttonBox.(buttons[i]).labelText = "On"
   endIf
endFor
```

Properties of field objects

Field objects have many properties. Following are just a few.

- *CursorLine* reports or specifies the vertical position of the insertion point in a field object, relative to the first line. The first line is 1, the second is 2, and so on.
- *CursorCol* returns the insertion point column; that is, the number of characters between the insertion point and the left margin of the field.
- *CursorPos* returns the number of characters between the insertion point and the first character of the field.
- *SelectedText* returns a string representing the currently selected text, or an empty string if no text is selected.

  For example, suppose a field contains this text:

  ```
  "Select the third word of this sentence."
  ```

  Suppose further that this field's **mouseRightUp** method is

  ```
  method mouseRightUp(var eventInfo MouseEvent)
  if self.selectedText <> "" then
     msgInfo("You selected", self.selectedText)
  endIf
  endMethod
  ```

  Now, when you run this form, select the third word of the sentence, and click the right mouse button, a dialog box appears. Its title is "You selected", and inside the dialog box is the word "third."

- *FieldName* reports or specifies which field of which table a field object is bound to. For example, the following statement binds the field object named *myField* to the PartNum field of the *Parts* table (which must be in the form's data model).

  ```
  myField.fieldName = "[Parts.PartNum]"
  ```

  Quotes are required outside the square brackets. When referring to a table with its file extension (for example, CUSTOMER.DBF), place quotes preceded by backslashes around the name of the table, as shown in the following statements:

  ```
  myField.fieldName = "[\"customer.dbf\".Last name]"
  ```

  This is necessary because the period in the file extension is a special character to the ObjectPAL compiler and would cause it to misinterpret your code. In general, use quotes preceded by backslashes where dots or spaces may create ambiguity.

  Spaces in a field name do not require quotes. For example,

  ```
  myField.fieldName = "[customer.Last name]"
  ```

  Also, the DataSource property, which specifies a field as a source of items in a list, uses the same syntax.

- *Value* gets or sets the current value of a field object.

  Field objects have a property called Value that evaluates to an AnyType. For example, the statements

  ```
  var x AnyType endVar

  x = someField.value
  ```

assign to *x* the value of the field object *someField*, no matter what type of data *someField* contains.

For convenience, ObjectPAL lets you omit the VALUE keyword in expressions, so these two statements are equivalent:

```
x = someField.value
x = someField
```

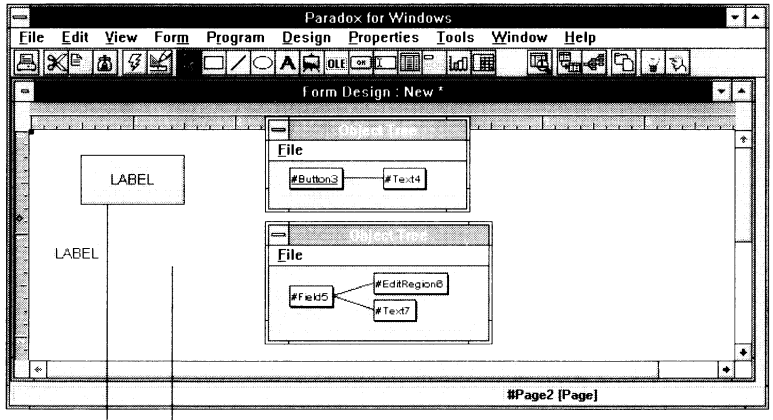
Remember, even when you don't use the VALUE keyword, you're working with the value of the object, not with the object itself.

Compound objects

A compound object is made of two or more objects. A simple example is a button, which consists of the button object and a text box that acts as a label. Another example is a labeled field, which consists of the field object, the label, and the edit region. You can use the Object Tree to see how objects in a compound object are related. Select the object, then choose Tools | Object Tree. Figure 13.2 shows the trees for a button and a labeled field.

Note When working with labeled fields, you'll typically attach methods to the field object, not to the label or the edit region.

Figure 13.2 Object Tree for a button and a labeled field



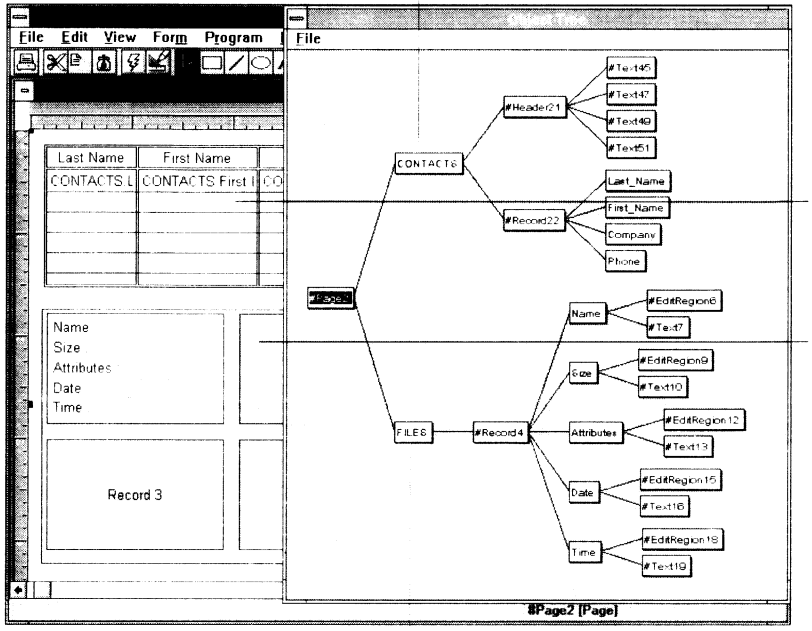
The tree for this button shows the button object and the text box that serves as a label. Typically, you'll attach methods to the button object (highlighted in the tree).

The tree for this labeled field shows the field object, the text box that serves as a label, and the edit region, which is where the data goes. You attach methods to the field object (highlighted in the tree).

Table frames and multi-record objects are more complex compound objects. As shown in Figure 13.3, a table frame contains a record object that contains field objects that might contain text boxes. All these objects are created automatically when you create the table frame.

Figure 13.3 Object Tree for a table frame and a multi-record object

This part of the tree shows the objects in the CONTACTS TableFrame. Notice the record object, which contains the field objects. Anything you do to the record object affects every record in the TableFrame; anything you do to a field object affects that field in every record.



This is a TableFrame bound to CONTACTS.DB

This is a multi-record object bound to FILES.DB. It displays 4 records from the table. The gray boxes are just place holders; you can't inspect them.

This part of the tree shows the objects in the FILES multi-record object. This multi-record object displays 4 records, each with the same structure. The tree shows only the objects contained in one record. Anything you do to that record affects all records, and anything you do to an object in that record affects that object in every record. The other records in the tree are just place holders, counterparts of the gray boxes in the multi-record object.

Although there is no Toolbar tool for creating record objects (as opposed to multi-record objects), record objects belong to the UIObject type. You can work with any object (including record objects) in a compound object just as you would with any other object in the UIObject type. You can inspect it, set properties, and attach methods. Also, the figure shows how multi-record objects use place holder objects to represent multiple records.

By default, an object bound to a table takes its name from that table. For example, when you bind a table frame to the *Customer* table, the table frame's name becomes CUSTOMER. You can change the default name, either interactively by inspecting the object and entering a new name, or by using ObjectPAL to set its Name property. In either case, you can use the TableName property to find out the name of the underlying table. For more information about table frames and multi-record objects (as well as crosstabs and graphs), see the *User's Guide*.

Events and UIObjects

UIObjects respond to all types of events. Different objects can respond differently to the same event. See Chapter 12 for more information.

Keyboard events

You can use the built-in methods **keyChar**, **keyPhysical**, and **action** to respond to keyboard events. See Chapter 10 for more information and examples.

Use the **keyChar** method defined for the UIObject type to send characters to an object in a form. For example, the following code sends the letter *A* to the field object named *hester*. You could attach this code to a button, and each time you click the button, another A will appear in the field object.

```
method pushButton(var eventInfo Event)
   hester.keyChar("A")
endMethod
```

You can also use **keyChar** to send character strings. For example, the following code sends the string "To be, or not to be" to the field object named *hamlet*.

```
method pushButton(var eventInfo Event)
   hamlet.keyChar("To be, or not to be")
endMethod
```

Mouse events

UIObjects come with built-in methods you can modify to respond to many common mouse clicks and movements. Also, you can use **hasMouse** to find out if the pointer is over an object, and use **wasLastClicked** and **wasLastRightClicked** to find out if an object was the last object to receive a mouse click (or a right mouse click). See Chapter 10 and Chapter 12 for more information and examples.

Timer events

Use **setTimer** to specify when to send timer events to an object, then modify the object's **timer** method to control how the object responds. (Use **killTimer** to turn off an object's timer.) The following example displays bitmaps at specified timer intervals to create an animation effect.

These methods assume you have already created a form and placed two bitmap objects (named *flapUp* and *flapDown*), one on top of the other.

Use the following code to modify the form's **open** method:

```
method open(var eventInfo Event)
   self.setTimer(100)          ; generate a TimerEvent every 100/1000 of a second
   flapUp.visible = True       ; display one bitmap
   flapDown.visible = False    ; hide the other bitmap
endMethod
```

Use the following code to modify the form's **timer** method:

```
method timer(var eventInfo TimerEvent)
   flapUp.visible = not(flapUp.visible) ; toggle the Visible property
   flapDown.visible = not(flapDown.visible)
endMethod
```

Value events

Value events happen when the value of a field changes. ObjectPAL can distinguish between scrolling from one record to another and entering data into the field. You can also use the ValueEvent type method **newValue** in a field object's built-in **changeValue** method to inspect a field object's value and decide if you want to commit the value.

Suppose the following code is attached to a field object's built-in **changeValue** method. When you enter a value into this field and try to commit the value (for example, by pressing *Tab* or *Return*), this method inspects the value. If the value is less than 100, this method displays a dialog box prompting you to enter another value; otherwise, it lets the built-in default code process the value normally.

```
method changeValue(var eventInfo ValueEvent)
   if eventInfo.newValue() < 100 then
      msgStop("Stop", "Enter a value greater than 100.")
      disableDefault
   endIf
endMethod
```

Refer to the "ValueEvent" section of Chapter 10 and the online ObjectPAL Help for more information and examples.

UIObjects: Shortcuts and special cases

This section describes how to copy, prototype, and create UIObjects and how to define calculated fields.

Copying objects

When you copy an object, the result is an exact copy of the object, its properties, methods, and procedures. The only thing that changes is the object's name. When you copy an object, be sure to update code that refers to the object by name. If a method in an object uses *Self* and then you copy the object, *Self* refers to the copy, not to the original object, because *Self* always refers to the object to which the code is attached.

There is no link between the original object and the copy. Changing a method attached to one object has no effect on methods in the other object, as demonstrated in the following example.

1 Create a form containing one button, and modify the button's **pushButton** method as follows:

```
method pushButton (var eventInfo Event)
   message("I'm the original button.")
endMethod
```

2 Close the ObjectPAL Editor window, save changes, and name the button *myOriginal*.

3 Choose Design | Duplicate to copy this button. (You get the same effect if you cut and paste or copy and paste the object.)

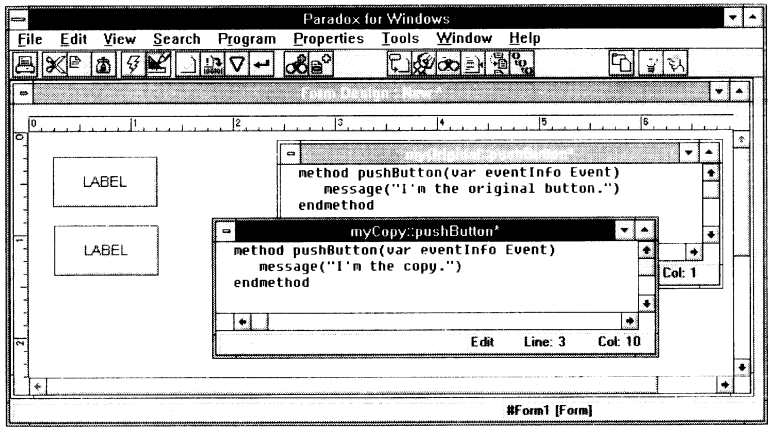
4 Inspect the duplicate button's **pushButton** method. It's the same. Now change it:

```
method pushButton (var eventInfo Event)
   message("I'm the copy.")
endMethod
```

5 Close the ObjectPAL Editor window, save changes, and name the button *myCopy*.

6 The method for the original button does not change. To confirm this, choose Form | View Data, then click each button and compare the messages that appear in the status line. You could also compare the methods themselves by displaying each in its own ObjectPAL Editor window, as shown in Figure 13.4.

Figure 13.4 The original and the copy



Prototyping objects

A fast way to create objects that have the same properties and methods is to create a prototype object. Here's one way to do it:

1 Use Paradox interactively to create an object, set its properties, and write its methods.

2 Select the object.

3 Choose Design | Copy To Toolbar.

Now, every object of that type will have those properties and methods. For example, if you create a field object, set its properties and write its methods, then copy it to the Toolbar, *every* subsequent field will have those properties and methods—this includes fields you create using the Field tool from the Toolbar, fields you create using ObjectPAL,

and fields created as parts of compound objects (for example, the fields in a table frame). For more information about creating and saving prototype objects, see the *User's Guide*.

Creating UIObjects

You can create and manipulate UIObjects from within a method using **create**, **delete**, **methodGet**, **methodSet**, and **methodDelete**. The constants (like BoxTool) for creating objects are listed online. To display the list, open an ObjectPAL Editor window, choose Tools | Constants, then choose Types Of Constants. The constants appear in the Constants column.

Note Objects created from within a method are invisible until you set their Visible property to True.

The following example manipulates objects when you choose View Data or open a design window. It creates a form and some rectangles, assigns methods to the rectangles, and changes their colors. (You can create objects in a design window or in View Data, but you can work with their methods only in a design window.)

```
method pushButton(var eventInfo Event)
var
   f form
   ui Array[10] UIObject
   c array[3] longInt
   i SmallInt
endVar

c[1] = Red
c[2] = Green
c[3] = Blue

f.create() ; Create a new form in a design window

; Create some rectangles
for i from 1 to ui.size()
   ui[i].create(BoxTool,i*100,i*100,1000,1000,f)
   ui[i].color = c[i.mod(3)+1]
   ui[i].visible = True
endFor

; Name the rectangles
for i from 1 to ui.size()
   ui[i].name = "BoxNumber" + String(i)
endFor

f.save("MyForm") ; Save the form

; Create an open() method for each object
for i from 1 to ui.size()
   ui[i].MethodSet("open",
       "method open(var eventInfo Event)
        self.color = Red
        self.visible = True
        Beep()
```

```
        endMethod
        "}
endFor
```

The next example shows how to work with objects in a design window:

```
for i from 1 to ui.size()
   ui[i].setPosition(2000 * rand(), 2000 * rand(),
                     2000 * rand(), 2000 * rand())
   ui[i].color = c[mod(rand()*100,3) + 1]
   ui[i].visible = True
endFor
```

The next example shows how to work with objects at run time:

```
f.run() ; Switch to View Data

for i from 1 to ui.size()
   ui[i].setPosition(6000*rand(), rand()*6000,
                     4000 * rand(), 4000 * rand())
   ui[i].color = c[mod(rand()*100,3) + 1]
   ui[i].visible = True
endFor

f.design() ; switch back to a design window

; Delete the objects
for i from 1 to ui.size()
   ui[i].visible = False
   ui[i].delete()
endFor

endMethod
```

Working with menus

This chapter presents advanced techniques for working with menus, and explains how to

- Work with pop-up menus alone
- Work with Paradox built-in menus and the Toolbar
- Work with Windows system menus
- Respond to choices from Paradox built-in menus

Lesson 9 in the ObjectPAL tutorial in Chapter 2 introduces the basic technique for working with menus: use **addText** and **addPopUp** to build the menu, use **show** to display it, and use **action** to respond to choices. This approach has the advantage of being quick and easy, but it has limitations.

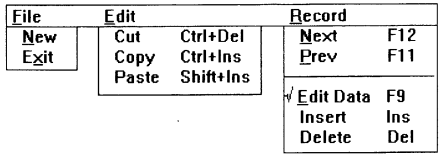
For example, you can't have two items with the same name. Suppose you want to provide two menu choices, Edit | Copy and Table | Copy. Using the basic technique, you can't tell them apart. Or suppose you want a menu choice to be available sometimes, and other times make it unavailable and dimmed (grayed), depending on data. Finally, suppose you want to provide keyboard access—that is, enable users to use the keyboard to choose menu items.

To accomplish these tasks, you'll need to use the following techniques.

- Specify menu item ID numbers
- Specify the display attributes of menu items
- Provide keyboard access to menu items
- Inspect items in a menu

In this chapter, you'll build the full-featured menu shown in Figure 14.1. It's almost identical to the menu in Lesson 9 of the ObjectPAL tutorial, but you'll use advanced features to add more functionality.

Figure 14.1 A full-featured menu



At the end of this chapter is an example showing how to respond to choices from Paradox built-in menus.

Assigning ID numbers to menu items

The first step in creating a menu for a complex application is to assign ID numbers to the menu items. By doing so, you can use integer values (rather than character strings) to identify menu choices. By assigning ID numbers, you can write code that executes consistently regardless of the text in the menu items: you can have duplicate items, you can change items, you can even translate them to another language without affecting your underlying code.

Start by defining some constants. Attach the following code to the page's Const window.

```
Const
   FileNew = 101
   FileExit = 102

   EditCut = 201
   EditCopy = 202
   EditPaste = 203

   RecordNext = 301
   RecordPrev = 302
   RecordEdit = 303
   RecordIns = 304
   RecordDel = 305
endConst
```

This code assigns integer values to constants that represent menu choices; for example, the constant FileNew represents the menu choice File | New. The literal integer values have no significance, except as an aid to memory. The value 101 represents the first item in the first menu, 201 is the first item in the second menu, and so on.

Getting the most out of addText

Now that you have defined the constants, you can use them in **addText** statements to build the pop-up menus and menus. The extended syntax for **addText** takes three arguments: the text of the menu item (as before), an ID number, and another argument that specifies the item's display attribute. For example, if you want to display an item dimmed or checked, you can.

You can also use **addText** to provide keyboard access in one or both of the following ways:

- Using combinations of *Alt* and other keys. Placing an ampersand (&) before a letter in a menu item designates the key to press with *Alt*.
- Using accelerators. An *accelerator* is a function key or combination of keys you can press to access a menu item. Creating an accelerator is a two-step process: first, use **addText** to put the accelerator in the menu item; second, write code to translate the keystroke into an action (described later).

The next example presents code that does all these things, and an explanation follows.

```
method arrive(var eventInfo MoveEvent)
   var
      mainMenu Menu
      filePop, editPop, recordPop PopUpMenu
   endVar

; build the File menu
   filePop.addText("&New", MenuEnabled, UserMenu + FileNew)
   filePop.addText("E&xit", MenuEnabled, UserMenu + FileExit)
   mainMenu.addPopUp("&File", filePop)

; build the Edit menu
   editPop.addText("Cut\tShift+Del", MenuGrayed + MenuDisabled,
                                                   UserMenu + EditCut)
   editPop.addText("Copy\tCtrl+Ins", MenuGrayed + MenuDisabled,
                                                   UserMenu + EditCopy)
   editPop.addText("Paste\tShift +Ins", MenuGrayed + MenuDisabled,
                                                   UserMenu + EditPaste)
   mainMenu.addPopUp("&Edit", editPop)

; build the Record menu
   recordPop.addText("&Next\tF12", MenuEnabled, UserMenu + RecordNext)
   recordPop.addText("&Prev\tF11", MenuEnabled, UserMenu + RecordPrev)
   recordPop.addSeparator()
   recordPop.addText("&Edit Data\tF9", MenuEnabled, UserMenu + RecordEdit)
   recordPop.addText("Insert\tIns", MenuGrayed + MenuDisabled,
                                                   UserMenu + RecordIns)
   recordPop.addText("Delete\tCtrl+Del", MenuGrayed + MenuDisabled,
                                                   UserMenu + RecordDel)
   mainMenu.addPopUp("&Record", recordPop)

; display the menu
   mainMenu.show()
endMethod
```

There's a lot of code here, but don't be daunted: when you understand one line, you'll understand them all. Here's the line:

```
recordPop.addText("&Edit Data\tF9", MenuEnabled, UserMenu + RecordEdit)
```

The first part, `recordPop.addText`, calls **addText** to add an item to the pop-up menu represented by the variable *recordPop*.

The next part, `&Edit Data\tF9`, assigns the string "Edit Data" to the item. The ampersand before the letter E means you can press *Alt+E* to access this item (this functionality is built into Windows, and requires no programming on your part). The characters `\tF9` assign the *F9* key as an accelerator. You may recognize "\t", it's the backslash code for a tab character. You'll have to write code to make this accelerator work, as explained later.

The next part, `MenuEnabled`, is an ObjectPAL constant. It makes the item appear normally, and enables the user to choose it. You can add constants; for example, MenuGrayed + MenuDisabled makes the item gray and unselectable. Table 14.1 lists the ObjectPAL constants for setting menu choice attributes.

The last part, `UserMenu + RecordEdit`, specifies a constant (defined earlier) to identify this menu item. This constant is used in the built-in **menuAction** method (described later).

You can define your own constants to identify menu items, but there's a restriction: you must keep them within a specific range. Because this range is subject to change in future versions of Paradox, ObjectPAL provides the constants UserMenu and MaxUserMenu to represent the minimum and maximum values allowed. By adding UserMenu to your own constant, you guarantee yourself a value above the minimum. To keep the value under the maximum, you'll have to check the value of MaxUserMenu. One way is to use a message statement, as follows.

```
message(MaxUserMenu).
```

In this version of Paradox the difference between UserMenu and MaxUserMenu is 2047. That means the largest value you can use for a menu ID number is UserMenu + 2047, so you shouldn't define a menu constant to have a value greater than 2047.

As you can see, the extended syntax for **addText** lets you control many aspects of a menu's appearance and functionality.

Table 14.1 Menu choice attributes

Constant	Description
MenuChecked	Displays the item preceded by a checkmark
MenuDisabled	Makes the item inactive
MenuEnabled	Makes the item active
MenuGrayed	Displays the item in gray (dimmed) characters
MenuHilited	Displays the item highlighted
MenuNotChecked	Displays the item without a checkmark
MenuNotGrayed	Displays the item normally
MenuNotHilited	Displays the item without a highlight

Processing menu choices by ID number

Lesson 9 of the ObjectPAL tutorial in Chapter 2 shows how Paradox returns the text of an item chosen from a menu. Paradox also returns an integer value representing the ID number of the chosen menu item. This section explains how to work with menu ID numbers.

The following code, attached to the page's built-in **menuAction** method, processes menu choices by ID number. It uses **id**, defined for the MenuEvent type, to get the ID number of the chosen menu item stored in *eventInfo*. Then it uses a large **switch** block to specify an appropriate response to each menu choice.

```
method menuAction(var eventInfo MenuEvent)
   var
      itemID SmallInt
      formVar Form
   endVar

   itemID = eventInfo.id()

   switch
   ; File menu
      case itemID = UserMenu + FileNew  : formVar.create()
      case itemID = UserMenu + FileExit : close()

   ; Edit menu
      case itemID = UserMenu + EditCut   : active.action(EditCutSelection)
      case itemID = UserMenu + EditCopy  : active.action(EditCopySelection)
      case itemID = UserMenu + EditPaste : active.action(EditPaste)

   ; Record menu
      case itemID = UserMenu + RecordNext : active.action(DataNextRecord)
      case itemID = UserMenu + RecordPrev : active.action(DataPriorRecord)
      case itemID = UserMenu + RecordEdit : active.action(DataToggleEdit)
      case itemID = UserMenu + RecordIns  : active.action(DataInsertRecord)
      case itemID = UserMenu + RecordDel  : active.action(DataDeleteRecord)
   endSwitch
endMethod
```

Processing menu choices by ID number is very similar to processing them by item string. You have to write more code to work with ID numbers, but your code will execute regardless of the text of the menu items.

Controlling menu item attributes

The code in the section titled "Getting the most out of addText" showed how to use ObjectPAL constants with **addText** to control an item's appearance. You can use these same constants after the menu is displayed. Usually, you'll want to control an item's appearance based on conditions in the form. For example, when the user presses *F9* or chooses Record | Edit Data from your custom menu to enter or exit Edit mode, you'll want to check or uncheck the Edit Data menu item to reflect the current state of the form. Also, you'll want to enable the Record | Insert and Record | Delete choices when in Edit mode, and disable them otherwise. The following code, attached to the page's built-in **action** method, shows how to do it.

```
method action(var eventInfo ActionEvent)
   var
      RecordEditAttrib LongInt
   endVar
   if eventInfo.id() = DataToggleEdit then
```

```
      RecordEditAttrib = getMenuChoiceAttributeByID(UserMenu + RecordEdit)
      if hasMenuChoiceAttribute(RecordEditAttrib, MenuChecked) then
         setMenuChoiceAttributeByID(UserMenu + RecordEdit, MenuNotChecked)
         setMenuChoiceAttributeByID(UserMenu + RecordIns,
                                    MenuGrayed + MenuDisabled)
         setMenuChoiceAttributeByID(UserMenu + RecordDel,
                                    MenuGrayed + MenuDisabled)
      else
         setMenuChoiceAttributeByID(UserMenu + RecordEdit, MenuChecked)
         setMenuChoiceAttributeByID(UserMenu + RecordIns, MenuEnabled)
         setMenuChoiceAttributeByID(UserMenu + RecordDel, MenuEnabled)
      endIf
   endIf
endMethod
```

You can use this technique to control menu choice attributes from other objects, too. For example, you could attach code to a field object's built-in **changeValue** method to enable or disable a menu choice depending on the field object's value.

Using MenuInit to control menu choice attributes

When you choose an item from the menu bar, before Paradox displays the associated pop-up menu, it initiates an action, represented by the constant MenuInit. For example, when you choose File | New from your custom menu, this is the sequence of events:

1. You click the word "File" in the menu bar.
2. Paradox initiates a MenuInit menu action and a MenuEvent, which triggers the built-in **menuAction** methods of objects in the form, as appropriate.
3. Paradox displays the pop-up menu associated with the File menu item.
4. You choose "New" from the pop-up menu, which generates a MenuEvent.
5. The MenuEvent triggers the built-in **menuAction** methods again.

In other words, MenuInit represents that instant before the pop-up menu appears, when you can initialize attributes of menu items before they're displayed to the user. For example, in the custom menu you created in the first part of this section, the menu choices Edit | Cut and Edit | Copy are dimmed and disabled in the initial **addText** statement. By adding the following code to the page's built-in **menuAction** method, you make them available when the user selects text in a field object. In this example, the code to handle MenuInit precedes the switch block that handles actual menu choices. This order is recommended for readability, and because it illustrates the sequence of events that happens in Paradox.

```
method menuAction(var eventInfo MenuEvent)

   var
      itemID SmallInt
      retVal Logical
   endVar

itemID = eventInfo.id()
```

```
if itemID = MenuInit then
   try
      retVal = active.Editing ; Editing property must be True
   onFail                     ; to access SelectedText property
      return
   endTry

   if retVal = True then
      if active.SelectedText <> "" then
         setMenuChoiceAttributeByID(UserMenu + EditCut, MenuEnabled)
         setMenuChoiceAttributeByID(UserMenu + EditCopy, MenuEnabled)
      else
         setMenuChoiceAttributeByID(UserMenu + EditCut,
                                                MenuGrayed + MenuDisabled)
         setMenuChoiceAttributeByID(UserMenu + EditCopy,
                                                MenuGrayed + MenuDisabled)
      endIf
   endIf
endIf
{ the switch block to handle actual menu choices goes here }

endMethod
```

You can use MenuInit to test the properties of any object in the form, and set menu choice attributes accordingly.

Accelerators

An accelerator is a function key or combination of keys the user can press to access a specific item in a specific menu. For example, the accelerator *Ctrl+Ins* has the same effect as choosing Edit | Copy (but the menu doesn't drop down). Creating an accelerator is a two-step process:

1 Add the accelerator to the menu item.

2 Write a method to translate the keystroke into a menu choice.

The following sections present these steps in the context of a simple menu, not the complex menu used in earlier examples.

Step 1: Adding an accelerator to a menu item

To do the first step, use "\t" to put a tab between an item and its accelerator, for example,

```
recordPop.addText("Next\tF12")
recordPop.addText("Prev\tF11")
```

appears as

```
Next     F12
Prev     F11
```

Pressing *F12* is the same as choosing Next, and pressing *F11* is the same as choosing Previous. As with ampersands, the "\t" is part of the return value. So, to test the user's choice, do something like this:

```
method menuAction(var eventInfo MenuEvent)
   var
      itemID String
   endVar

   itemID = eventInfo.menuChoice()
   switch
      case itemID = "Next\tF12" : active.action(DataNextRecord)
      case itemID = "Prev\tF11"  : active.action(DataPriorRecord)
   endSwitch
endMethod
```

Step 2: Translating the keystroke to a menu choice

The second step involves writing a method to take the appropriate action when the user presses an accelerator. To do this, override the form's built-in **keyPhysical** method.

Note This example just tells you *how*. To find out *why*, see Chapter 10, "Understanding the event model" and Chapter 12, "Default behavior of built-in methods."

1. Right-click the form's title bar to inspect it, and open an ObjectPAL Editor window to edit the **keyPhysical** method.
2. Make **keyPhysical** look like this:

```
method keyPhysical(var eventInfo KeyEvent)
   var
      theKey String
   endVar

if eventInfo.isPreFilter() then
   disableDefault
   theKey = eventInfo.vChar()
   switch
      case theKey = "VK_F12" : active.action(DataNextRecord)
      case theKey = "VK_F11" : active.action(DataPriorRecord)
      otherwise              : doDefault
   endswitch
endIf
endMethod
```

This example uses virtual keycodes (described in the "KeyEvent" section of Chapter 10). ObjectPAL provides constants for Windows virtual keys (like *F11* and *F12*), listed online. To display the list, open an ObjectPAL Editor window and choose Tools | Constants. Then, from the Types of Constants column, choose Keyboard. The constants appear in the Constants column.

Responding to choices from Paradox built-in menus

Previous sections have discussed how to build and respond to choices from custom menus. This section explains how to respond to choices from Paradox built-in menus. The technique shown here can save you from having to program a complete menu system when all you want to do is provide a few special functions. For example, suppose that when the user chooses File | Exit, you want to display a custom dialog box. You can't add your own items to the built-in menus, so to provide this feature, you'd have to program an entire menu system. Or, you could use the following technique to respond when the user chooses File | Exit from the built-in menu. The code in this example is attached to the page's built-in **menuAction** method:

```
method menuAction(var eventInfo MenuEvent)
   var
      customDlg Form
      dlgChoice AnyType
   endVar

   if eventInfo.id() = MenuFileExit then
      customDlg.open("dlg.fdl")
      dlgChoice = customDlg.wait()
      doDefault
   endIf
endMethod
```

This code uses the MenuEvent type method **id** and the MenuCommand constant MenuFileExit to test the value of the chosen menu item. The MenuCommand constants are defined to have the values returned by the corresponding item in a built-in menu. MenuFileExit represents the menu choice File | Exit. Other examples are MenuEditCopy, which represents Edit | Copy, and MenuHelpAbout represents Help | About.

For another example of how to use MenuCommand constants, suppose you have a form full of objects, and the built-in menus do exactly what you want for all objects but one: a memo field named *memoFld*. For this memo field, you want Edit | Copy to write the contents of the memo field to a file instead of to the Clipboard. The following code, attached to the page's built-in **menuAction** method, shows one way to do it.

```
method menuAction(Var eventInfo MenuEvent)
   var
      theMemo Memo
   endVar

   if eventInfo.id() = EditCopy then
      theMemo = memoFld.value
      theMemo.writeToFile("memoData.txt")
   endIf
endMethod
```

Use MenuCommand constants to test for and respond to menu actions. To initiate a menu action, use **action** or other methods or procedures. The System type provides several procedures that display built-in Paradox dialog boxes for such tasks as adding and copying tables. These procedures begin with the letters **dlg**, for example **dlgAdd** and **dlgCopy**. Refer to the online ObjectPAL Help for more information and examples.

PopUpMenu: Lists on demand

A pop-up menu is a vertical list of items that appears in response to an event (usually a mouse click). When the user chooses an item from a pop-up menu, the text of that item is returned to the method.

A pop-up menu is distinct from a menu, a horizontal list of items that appears in the application menu bar.

Note Choosing an item from a pop-up menu *does not* trigger the built-in **menuAction** method.

Using PopUpMenu methods, you can

- Build a pop-up menu
- Display the pop-up menu and return the selected item
- Inspect the items in a pop-up menu
- Provide keyboard access

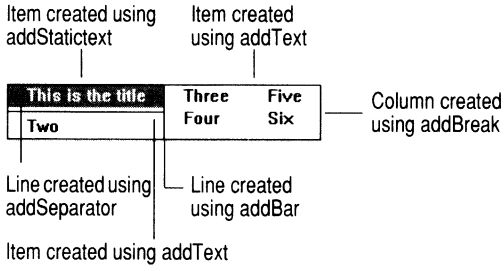
Building a pop-up menu

Use **addArray**, **addText**, **addStaticText**, **addSeparator**, **addBar**, and **addBreak** to build a pop-up menu. Use **show** to display the pop-up menu and return the selected item.

The following example shows how to build and display the pop-up menu shown in Figure 14.2. You probably won't ever build a menu like this one. Its purpose is to show at a glance the effects of the various menu-building methods.

```
var
   p PopUpMenu
   theChoice String
endVar

p.addStaticText("This is the Title")
p.addSeparator()
p.addText("Two")
p.addBar()
p.addText("Three")
p.addText("Four")
p.addBreak()
p.addText("Five")
p.addText("Six")

theChoice = p.show()
; Displays the menu, puts the user's choice into theChoice.
; Subsequent commands and methods will execute based on the
; value of theChoice.
```

Figure 14.2 Sample pop-up menu



Using addArray

You can use **addArray** to get the effect of multiple **addText** statements. For example, the code in example 1 produces the same pop-up menu as the code in example 2:

Example 1

```
var
   ar Array[3] String
   p PopUpMenu
   x String
endVar
ar[1] = "one"
ar[2] = "two"
ar[3] = "three"
p.addArray(ar)
x = p.show()
```

Example 2

```
var
   p PopUpMenu
   x String
endVar
p.addText("one")
p.addText("two")
p.addText("three")
x = p.show()
```

There's not much difference between these two examples, because you create the array and the added items with the method. To get the full benefit of **addArray**, create the array somewhere else. For example, use a **copyToArray** statement to copy the fields of a table to an array. Then you could display them in a pop-up menu.

Using addPopUp

addPopUp adds one pop-up menu to another to create cascading menus. In the following example, *p2* is a pop-up menu containing three items: First, Second, and Third. When you choose Third, a second pop-up menu (*p1*) appears.

```
var
   p1, p2 PopUpMenu
   x String
endVar
p1.addText("one")
p1.addText("two")
```

```
p2.addText("First")
p2.addText("Second")
p2.addPopUp("Third", p1)

x = p2.show()
```

Keyboard access

The techniques for providing keyboard access to a pop-up menu are the same as for a menu. See Lesson 8 in the ObjectPAL tutorial in Chapter 2 for more information.

Inspecting items in a pop-up menu

Use **contains** and **count** to inspect the items in a pop-up menu. These methods are useful when you're building a pop-up menu on the fly. Use **remove** to delete items. Refer to the entries for these methods in the online ObjectPAL Help for more information.

switchMenu: A shortcut

The PopUpMenu type provides the **switchMenu** structure as a shortcut for building and displaying pop-up menus. It combines the functions of **addText** and **show** with a switch...endSwitch block. The case statements to the left of the colon specify the menu items to be displayed, and the statements to the right execute depending on the item you choose.

The next example displays a pop-up menu of two items, *itemOne* and *itemTwo*. You can choose either item using the mouse, and you can choose *itemOne* from the keyboard by pressing *Alt+O*. If you choose an item, a dialog box appears; otherwise, the system beeps.

```
switchMenu
   case "item&One" : msgInfo("You chose:", "itemOne")
   case "itemTwo"  : msgInfo("You chose:", "itemTwo")
   otherwise       : beep()
endSwitchMenu
```

Working with built-in menus and the Toolbar

You can use **menuAction** and **id** to work with Paradox's built-in menus. For example, let's say you want to display a message to users when they close your application. If you use Paradox's built-in menus, you could do this to modify an object's **menuAction** method:

```
method menuAction(var eventInfo MenuEvent)
   if eventInfo.id() = MenuFileExit then
      msgInfo("Good-bye", "Thank you.")
   endIf
endMethod
```

ObjectPAL provides MenuCommand constants (like MenuFileExit) to represent the numeric values of the built-in menu items. Constants are listed online. To display the list, open an ObjectPAL Editor window and choose Tools | Constants. Then, from the

Types of Constants column, choose MenuCommands. The constants appear in the Constants column.

As an alternative to using constants, you can use **menuChoice** to test the text string returned for each menu choice. For example, when you choose Edit | Copy, the returned string is "Copy".

```
method menuAction(var eventInfo MenuEvent)
if eventInfo.menuChoice() = "Copy" then
  msgInfo("Copy", "You chose Copy.")
else
   msgInfo("You chose:", eventInfo.menuChoice())
endIf
endMethod
```

Note If two menu items return the same string (for example, Edit | Copy and File | Utilities | Copy both return "Copy"), **menuChoice** works with the first one, counting from top to bottom, left to right.

Working with the Toolbar

The menu-handling techniques outlined in this chapter also work with Toolbar buttons that represent menu choices, because the buttons are shortcuts to the same end result. For example, clicking the Cut To Clipboard button on the Toolbar is equivalent to choosing Edit | Cut. So, you don't have to write separate handler methods for Toolbar buttons—you can use the same **menuAction** method to handle both events.

In the following example, **eventInfo.id** returns MenuEditCut whether you choose the menu item or click the Toolbar button.

```
method menuAction(var eventInfo MenuEvent)
if eventInfo.id() = MenuEditCut then
   doEditCut() ; do a custom method
endIf
endMethod
```

Note The default behavior for a form's **menuAction** method is to convert the menu action to an action and trigger the **action** method with the corresponding action ID. For example, suppose the form handles a menu action and the ID is MenuEditCut. By default, the form converts it to an action with an ID of EditCutSelection. In other words, when you choose an item from a menu, you trigger the **menuAction** method, which triggers the **action** method.

Working with control menus

You can use **id** and MenuCommand constants in an object's built-in **menuAction** method to test for choices (like Minimize and Maximize) from the Windows control menu. For example, the following code prevents you from minimizing the current form.

```
method menuAction(var eventInfo MenuEvent)
   if eventInfo.id() = MenuControlMinimize then
                     ; MenuControlMinimize is a constant
      disableDefault ; do not execute the default code
```

```
      beep()
      message("Can't minimize this form.")
    endIf
endMethod
```

Advanced example of working with menus

This example records menu actions, and lets the user push a button to replay the last menu command. For this example, assume a form has a field (perhaps a memo field), a button, and a box containing a text box. The following code is attached to the Var window for the form. The variables declared in this Var window are global to every object on the form.

```
Var
  lastMenuId    SmallInt   ; ID of the last menu selected
  lastTarget    UIObject   ; handle for the last target
endVar
```

The following code is attached to the form's **open** method. This code constructs a simple menu to let the user cut, paste, or delete selected text in the field.

```
method open(var eventInfo Event)
var
  menu1    Menu
  popUp1   PopUpMenu
endVar
if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
  else
    ;code here executes just for form itself
    popUp1.addText("&Cut")              ; construct a pop-up menu
    popUp1.addText("&Paste")
    popUp1.addText("&Delete")
    menu1.addPopUp("&Edit", popUp1)     ; attach the pop-up to a menu bar item
    menu1.show()                        ; show the menu
endIf
endMethod
```

The following code is attached to the form's built-in **menuAction** method. This code keeps track of the last menu selected and the object that had focus when that MenuEvent took place. This information is stored so that the button's built-in **pushButton** method knows which menu command to execute, and the object on which to execute the menu command.

```
method menuAction(var eventInfo MenuEvent)
if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
    if eventInfo.isFromUI() then         ; if user selects a menu
      lastMenuId = eventInfo.id()        ; store the ID of the selected menu
      eventInfo.getTarget(lastTarget)    ; store the current target handle to
                                         ; the variable lastTarget
    endIf
  else
```

```
    ;code here executes just for form itself
endIf
endMethod
```

The following code is attached to the button's built-in **pushButton** method. When the user presses this button, the code constructs an event from the last MenuEvent, then calls the built-in **menuAction** method that belongs to the last target. Essentially, this code replays the most recent menu command.

```
method pushButton(var eventInfo Event)
var
  lastMenuEvent MenuEvent                ; this creates a MenuEvent
endVar
 ; if user presses the button before selecting a menu,
 ; tell the user what's going on
if NOT isAssigned(lastMenuId) then
  msgStop("Stop", "You must select a menu before you can replay it.")
else                ; user must have selected a menu at least once
  lastMenuEvent.setId(lastMenuId)
                    ; set the menu event id to the last menu id
  lastTarget.menuAction(lastMenuEvent)
                    ; call menuAction method of the last target
endIf
endMethod
```

The following code is attached to a field's built-in **menuAction** method. This is where menu selections are evaluated and executed. For demonstration purposes, this code also changes the text and color of the box to indicate where the MenuEvent originated.

```
method menuAction(var eventInfo MenuEvent)
var
  choice String
endVar

 ; change the color of the box, and the text within the box to
 ; indicate where the MenuEvent originated from
boxOne.text = "isFromUI is " + strVal(eventInfo.isFromUI())
boxOne.color = iif(eventInfo.isFromUI(), Blue, Red)

 ; store the user's menu selection to choice
choice = eventInfo.menuChoice()

 ; now take action based on the menu selection
switch
  case choice = "&Cut"    : action(EditCutSelection)
  case choice = "&Delete" : action(EditDeleteSelection)
  case choice = "&Paste"  : action(EditPaste)
endswitch

endMethod
```

This simple example demonstrates how MenuEvents and menu actions are related. It's important to understand this relationship as you develop more complex menus for applications.

Performing calculations

This chapter describes how to use ObjectPAL in calculated fields.

The rule for using ObjectPAL in a calculated field is simple: A calculated field can be defined to use any ObjectPAL statement or expression that returns or results in a single value.

Note The *User's Guide* explains how to create and define a calculated field; this chapter focuses on ObjectPAL issues.

Elements in calculated fields

As shown in Table 15.1, a calculated field can use any of the following elements:

- Literal values.
- Variables, provided they are declared within the scope of the calculated field, and have been assigned a value.
- Object properties.
- Basic language elements.
- Custom methods attached to other objects (or to the field itself). You must first declare a UIObject variable within the scope of the calculated field and use an **attach** statement to associate the variable with a UIObject.
- Any method or procedure in the ObjectPAL run-time library (RTL) that returns a value (including a Logical value).
- Special functions (like **Sum** and **Avg**) provided specifically for use in calculated fields.

Also, the DataRecalc action is closely related to calculated fields, as described following the table.

Table 15.1 ObjectPAL elements in calculated fields

Element	Comments
5	Literal value.
"a"	Literal value.
x	Variable. Must be declared within the scope of the calculated field. Must be assigned a value. (See the example following this table.)
x + 5	Simple expression. Rules for working with variables apply.
self.name	Property. Displays the field's name (String).
theBox.color	Property. Displays an integer value representing the object's color.
iif(State.Value = "CA", 0.075, 0)	Basic language element **iif**. Value of calculated field depends on value of State field object.
uio.objCustomMethod()	Custom method attached to another object. The custom method must return a value. (See the example following this table.)
tc.open("orders.db")	RTL method. Field displays True if the **open** succeeds; otherwise, it displays False. TCursor must be declared within the scope of the field.
Avg([DIVEITEM.Sale Price])	Special function. Operates on the Sale price field of the *Diveitem* table. The table must be in the form's data model. Quotes are not used, spaces are allowed.
tc.cAverage("Sale Price")	RTL method. The TCursor must be declared and opened previously. The table does not have to be in the data model. If a field name contains spaces, quotes are required. (See the example following this table.)

Calculated field examples

This section presents the following examples:

- Using a variable
- Calling a custom method attached to another object
- Working outside the data model
- Using the DataRecalc action

Using a variable

The following example shows one way to use a variable in a calculated field. The variable must be declared within the scope of the calculated field; that is, it must be declared in the field's Var window, or in the Var window of an object that contains it. Also, the variable must be assigned a value before it is used by the field. The usual way to do this is in an object's built-in **open** method. In this example, the variable *myVal* is declared in the field's Var window and assigned a value in its **open** method.

The following code is attached to the field's Var window:

```
; This code is attached to the field's Var window
Var
```

```
    myVal SmallInt
endVar
```

The following code, attached to the field's built-in **open** method, assigns a value to the variable *myVal*:

```
method open(var eventInfo Event)
    myVal = 12
endMethod
```

The following code defines the calculated field:

```
myVal
```

When this form runs, the code executes to declare the variable and assign a value, and the value displays in the calculated field.

Calling a custom method attached to another object

The following example shows how to call a custom method attached to another object and display the returned value in a calculated field. You must declare a UIObject variable within the scope of the calculated field and use an **attach** statement to associate the variable with an object.

In this example, suppose a form contains a box named *theBox*, and attached to *theBox* is the following custom method:

```
method custMeth() Number ; method must return a value
    return 1001.22
endMethod
```

The following code, attached to the calculated field's Var window, declares a UIObject variable.

```
Var
    methBox UIObject
endVar
```

The following code is attached to the field's built-in **open** method. It calls **attach** to associate the UIObject variable *methBox* with the actual UIObject *theBox*.

```
method open(var eventInfo Event)
    methBox.attach(theBox)
endMethod
```

The following code defines the calculated field:

```
methBox.custMeth()
```

When the form runs, the calculated field displays 1001.22, the value returned by the custom method **custMeth**.

Working outside the data model

The special calculated field functions operate on tables in the form's data model. This example shows how to use a TCursor to work with tables outside the data model. You must declare a TCursor variable and open it before you can use it in a calculated field.

The following code, attached to the field's Var window, declares the TCursor variable:

```
Var
   itemsTC TCursor
endVar
```

The following code is attached to the field's built-in **open** method. It opens a TCursor onto the *Diveitem* table.

```
method open(var eventInfo Event)
   itemsTC.open("diveitem.db")
endMethod
```

The following code defines the calculated field:

```
itemsTC.cAverage("Sale Price")
```

When the form runs, the calculated field displays the average of the values in the Sale Price field of the *Diveitem* table.

Using the DataRecalc action

DataRecalc is an ObjectPAL action constant. You can use DataRecalc to initiate or respond to the action that makes a calculated field recalculate its value. For example, you could have a calculated field that displays the number of the current record in a table outside the form's data model. Suppose that as you work with the form, you use a TCursor to search for values in this table. A successful search changes the current record of the TCursor, but Paradox will not automatically update the calculated field—you'll have to do it yourself.

Note You cannot enter values directly into calculated fields, either interactively or using an ObjectPAL assignment statement. You must either let the field recalculate its value automatically or initiate a DataRecalc action. In other words, if you have a calculated field named *calcField*, the following statement will fail:

```
calcField.Value = 123 ; this assignment will fail
```

The following example shows how to use a DataRecalc action to update a calculated field. In this example, suppose that one of the objects in a form is a calculated field named *itemRecNo*. This field object displays the number of the current record in the *Diveitem* table, which is not part of the form's data model. Code attached to the form's Var window declares a TCursor variable *itemsTC*; code attached to the form's built-in **open** method associates *itemsTC* with the *Diveitem* table; and code attached to a button's built-in **pushButton** method initiates a search. If the search succeeds, code in the **pushButton** method initiates a DataRecalc action to update the value displayed in *numItemRecs*. The other code of interest in this example is the code that defines *numItemRecs*.

The following code is attached to the form's Var window:

```
Var
   itemsTC TCursor
endVar
```

The following code is attached to the form's built-in **open** method:

```
method open(var eventInfo Event)
   if eventInfo.isPreFilter() then
      ; code here executes for each object in the form
   else
      ; code here executes just for the form itself
      itemsTC.open("diveitem.db")
   endIf
endMethod
```

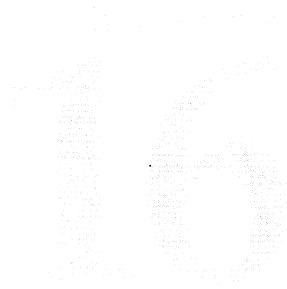
The following code defines the calculated field:

```
itemsTC.recNo()
```

The following code is attached to the button's built-in **pushButton** method:

```
method pushButton(var eventInfo Event)
   var
      itemNo LongInt
   endVar

   itemNo = 0
   itemNo.view("Enter an Item Number")

   if itemNo <> 0 then
      if itemsTC.locate("Item No", itemNo) then
         itemRecNo.action(DataRecalc)
      else
         msgInfo("Search failed", "Couldn't find " + String(itemNo))
      endIf
   endIf
endMethod
```

When you run this form, the calculated field *itemRecNo* displays 1. When you click the button, a **view** dialog box prompts you for an item number, then tries to find it in the *Diveitem* table. If the search is successful, the DataRecalc action updates the value displayed in *itemRecNo*.



Manipulating the data model

You can use ObjectPAL to manipulate a form's or report's data model by getting and setting information about which table or tables the form is bound to. The methods listed in Table 16.1 are described in detail in the online ObjectPAL Help.

Table 16.1 Methods for working with a form's data model

Method	Description
dmAddTable	Adds a table to a form's data model
dmAttach	Associates a TCursor variable with a table in a specified data model
dmBuildQueryString	Builds a query string based on the data model of a form
dmEnumLinkFields	Lists the fields that link two tables
dmGet	Gets a value from a field in a table in the form's data model
dmHasTable	Reports whether a specified table is in a form's data model
dmPut	Assigns a value to a field in a table in the form's data model
dmRemoveTable	Removes a table from a form's data model
dmGetProperty	Returns the value of a specified table property
dmLinkToFields	Links two tables in a form's data model based on lists of field names
dmLinkToIndex	Links two tables in a form's data model based on a list of field names and an index name
dmResync	Resynchronizes a table in a specified data model to a TCursor
dmSetProperty	Sets the value of a specified table property
dmUnlink	Unlinks two tables in a form's data model
enumDataModel	Lists the tables in a specified data model

These methods let you change values in a table in the data model directly, without attaching to a specific design object in the form.

The table name you specify must match either the name of a table in the current data model, including the alias (the name you assign to a directory path), or its *table alias* (a different name you assign to the table). Using a table alias lets you change the table your

code refers to without breaking your code). See the *User's Guide* for information about assigning table aliases.

If you have attached ":MYDB:MYDATA.DB" using the Data Model dialog box then you must specify ":MYDB:MYDATA.DB" as an argument to **dmGet** or **dmPut**. The field name is the physical field name in the table; it is not related to any design object name. For example, the following statements get the value of the Quantity field in the *Answer* table in the user's private directory and assign it to the variable *answerQty*:

```
var
   answerQty AnyType
endVar
dmGet(":PRIV:ANSWER.DB", "Quantity", answerQty)
```

Directory paths and the data model

When you add a table to the form's data model interactively, the table stores its name as follows:

- Tables with absolute paths or aliases (like ":GREG:SALES.DB" or "C:\TABLES\ORDERS.DB") store that path or alias exactly as specified.
- Tables with ":WORK:" as the alias strip that alias and store just the base name of the table.

When you open a form, it finds its data model tables as follows:

- Tables with path names or aliases resolve the complete path name and use it as is.
- Tables with no path or alias (that is, just the base name) use the path or alias where the form was found.

In other words, if you open a form in C:\TEMP (while your working directory is I:\RUN) and that form's data model includes BOOKORD.DB, the path to the table is C:\TEMP\BOOKORD.DB.

If the table has a relative path (such as MYDIR\BOOKORD.DB), the form sees the path as C:\TEMP\MYDIR\BOOKORD.DB. Relative paths also work with aliases, so :ALIAS:MYDIR\BOOKORD.DB is also valid.

Note The form does not search for the tables. If the tables are not where the form expects them to be, Paradox displays an error message.

When the form finds a table, it applies the same algorithm to any lookup tables used by that table. If the lookup table in the validity check file (*tablename*.val) has an absolute path, it is used as is. If the lookup table has no path, then it is assumed to exist wherever its table was found. So, in the previous example, if the lookup table is "MYLOOK.DB", the form expects it to be in C:\TEMP\MYDIR\MYLOOK.DB.

When you save a form to a directory other than the working directory, the system uses the same strategy to assign full names to the tables.

Performing table operations

A Table variable represents a description of a table; it does not open a table or display any data. In fact, you can use a Table variable to describe a table even before you create it. It is distinct from a TCursor, which is a pointer to the data, and from a table view, table frame, or multi-record object, which are objects that display the data.

Table variables and Table type methods let you perform two types of actions:

- You can perform table-level operations: add, copy, create, sort, and index tables, column calculations, get information about a table's structure, and more.

Note You can't use Table variables or Table type methods to edit a table; use a TCursor, tableview, table frame, or multi-record object instead.

- You can specify table attributes, including filters, ranges, indexes, table names, and access rights, before you actually open the table. A Table variable acts as a table control variable, creating a structure that describes or specifies the nature of a table.

Performing table-level operations

Although they are not methods or procedures, the ObjectPAL language elements for creating, indexing, and sorting tables are described in the online ObjectPAL Help; search for "table."

The following simple example shows how to create a table.

```
var
   myParts Table
endVar
myParts = CREATE
            "PARTS.DB"
            WITH "Part Number" : "A20", "Part Name" : "A20", "Quantity" : "S"
            KEY "Part Number"
          ENDCREATE
```

This code creates a Paradox table named PARTS.DB. The table has three fields: Part Number, Part Name, and Quantity. The Part Number field is a key field.

To perform a table-level operation on a table that already exists, first use **attach** to associate the Table variable with that table. (There is no **open** method for the Table type, because you can't open a Table variable for record- or field-level editing.) Then you can use Table methods to manipulate the description.

The following example declares the Table variable *myTable* and uses **attach** to associate *myTable* with ORDERS.DB. Next, **fieldType** tests the second field of the table (counting from left to right, starting at 1, not 0). If field 2 is a Date field, **cCount** counts the records that have entries in field 2, and **fieldName** gets the name of field 2. Finally, the information is displayed in a dialog box.

```
var
   myTable  Table                          ; declare the Table variable
   numDates LongInt
   fName    String
endVar
if myTable.attach("orders.db") then       ; associate myTable with ORDERS,DB
   if myTable.fieldType(2) = "Date" then ; if the second field is a Date field
      numDates = myTable.cCount(2)        ; count the records that have
                                          ; entries in field 2
      fName = myTable.fieldName(2)        ; get the name of field 2
      msgInfo("ORDERS.DB", fName + " has " + String(numDates) + " dates.")
   endIf
endIf
```

In a method, you can associate one Table variable with more than one table (and with tables of different types) but only with one table at a time. The following example declares the Table variable *myTbl* and uses **attach** to associate *myTbl* with the Paradox table ORDERS.DB. Then **cAverage** calculates the average of the values in the Quantity field. Next, **attach** associates *myTbl* with the dBASE table SALES.DBF (the Table variable unattaches automatically), and another call to **attach** associates the Table variable *bakTbl* with SALES.BAK. Finally, **copy** makes a backup copy of SALES.DBF, and **unlock** releases the lock on SALES.BAK.

```
var
   myTbl, bakTbl Table
   avgQty Number
endVar

myTbl.attach("orders.db", "Paradox") ; associate myTbl with ORDERS.DB
avgQty = myTbl.cAverage("Quantity")

myTbl.attach("sales.dbf", "dBASE")   ; associate myTbl SALES.DBF
bakTbl.attach("sales.bak", "dBASE")  ; associate bakTbl with SALES.BAK

myTbl.copy(bakTbl)                   ; copy SALES.DBF to SALES.BAK
```

Specifying table attributes

Other Table type methods specify table attributes, such as which indexes to use and maintain, and whether to display deleted records. (For examples search the online ObjectPAL Help for "table.")

Use these methods after using **attach** and before using the Table variable to open a TCursor. (TCursors are discussed in the next chapter.) For example,

```
var
   ordTbl Table
   ordTC  TCursor
endVar
ordTbl.attach("orders.dbf")    ; associate ordTbl with orders.dbf
ordTbl.setIndex("ordx.ndx")    ; set table attributes
ordTbl.usesIndexes("ord1.ndx", "ord2.ndx")
ordTbl.setReadOnly(Yes)
ordTbl.showDeleted(Yes)

ordTC.open(ordTbl)             ; open the TCursor using the specified attributes
```

Note Methods that perform column operations (for example, **cAverage** and **cCount**) operate on the range of data specified by the table variable. For example, the following statements associate the Table variable *custTbl* with CUSTOMER.DB, then call **cCount** to count the records that have a nonblank value in the first field. A dialog box displays the results. Then, the call to **setRange** tightens the table description, specifying records where values in the first field range from 2,000 to 4,000. Another call to **cCount** returns the number of records that meet this criteria, and another dialog box displays the results.

```
var
   custTbl Table
endVar

custTbl.attach("customer.db")
msgInfo("Before filter", custTbl.cCount(1)) ; displays 55

custTbl.setFilter(2000, 4000) ; specify filtering criteria
msgInfo("After filter", custTbl.cCount(1))  ; displays 20
```

As this example shows, you can use column functions on a Table variable, with or without filtering records. You can also use column functions on TCursor variables.

Using TCursors

A TCursor is a pointer to the data in a table, enabling you to manipulate data at the table level, record level, and field level without having to display the table. It is not 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.

This chapter explains how to manipulate data at the table level, record level, and field level without having to display the table.

About TCursors and tables

The Toolbar 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 an insertion point to a word-processor document. In a word processor, the insertion point 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. Table 18.1 lists some methods for working with TCursors.

Table 18.1 Methods for working with TCursors

Task	Methods
Getting structure information	nRecords, nFields, nKeyFields, fieldName, fieldNo, fieldType, enumTableProperties
Moving around	home, end, moveToRecord, nextRecord, priorRecord, skip
Determining position	atFirst, atLast, bot, eot, recNo
Finding values	locate, locateCase, locateNext, locatePrior, locatePattern, locateNextPattern
Filtering records	setGenFilter, setRange, getGenFilter, getRange,dropGenFilter,

Table 18.1 Methods for working with TCursors (continued)

Task	Methods
Doing column calculations	cAverage, cMax, cMin, cNpv, cStd, cSum
Working with records	copyRecord, currRec, initRecord, insertRecord
Controlling access	lock, unLock, lockRecord, unLockRecord, fieldRights

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

The three ways to make a TCursor point to a table are

- Open the table directly
- Open the table using attributes specified by a Table variable
- Associate a TCursor with a table window or UIObject bound to a table

Note If you attach TCursor to a UIObject that contains linked tables, the TCursor gets the structure and the data of the master table only.

Opening the table directly

To open a TCursor onto a table directly, without setting any attributes, use the TCursor type method **open**. The TCursor points directly to that table, independent of objects (for example, a table frame) in the form. Then you can use TCursor type methods to get information, move around, find values, get values, set values, do calculations, control access, and so on. This example makes *ordersTC* point to the first record of ORDERS.DB:

```
var
   ordersTC TCursor           ; declare the TCursor
endVar
ordersTC.open("orders.db")     ; open an insertion point onto the table
```

Now you can use TCursor methods on *ordersTC* to work with the data in ORDERS.DB.

Within a method, you can make a TCursor variable point to more than one table but only to one table at a time. In the following example, the TCursor variable *tcVar* points first to DIVEITEM.DB, then to DIVECUST.DB. TCursor is closed automatically in between, and hence the table DIVECUST.DB is closed.

```
var
   tcVar TCursor
   x String
endVar

tcVar.open("diveItem.db")
x = tcVar.fieldName(1)
x.view()                ; displays the name of the first field in DIVEITEM.DB

tcVar.open("diveCust.db")
```

```
x = tcVar.fieldName(1)
x.view()                ; displays the name of the first field in DIVECUST.DB
```

Opening the table using attributes

You can use a Table variable to specify table attributes, including filters, ranges, indexes, and access rights, before using a TCursor to open the table. A Table variable acts as a control variable, in effect creating a window through which the TCursor can open the table, as this example shows:

```
var
   ordTbl Table
   ordersTC TCursor
endVar

ordTbl.attach("orders.dbf")              ; associate ordTbl with orders.dbf
ordTbl.setIndex("ordx.ndx")              ; set table attributes
ordTbl.usesIndexes("ord1.ndx", "ord2.ndx")
ordTbl.setReadOnly(Yes)
ordTbl.showDeleted(Yes)

ordersTC.open(ordTbl)   ; open the TCursor using the specified attributes
```

Note In the following example, the first statement is not the same as the second statement:

```
ordTC.open(ordTbl)
ordTC.open("orders.dbf")
```

The first statement opens ORDERS.DBF using the attributes specified in the Table variable *ordTbl*; the second opens ORDERS.DBF directly, without specifying attributes.

Associating a TCursor with a table view or a UIObject

You can use the TCursor type method **attach** to associate a TCursor with the table displayed in a table window (the TableView type is described in Chapter 20). For example, the following statements declare a TCursor variable named *ordTC*, then call **attach** to associate *ordTC* with the *Orders* table displayed in a table window opened with the TableView variable *ordTV*:

```
var
   ordTC TCursor
   ordTV TableView
endVar

if ordTV.open("orders.db") then   ; open a table window
   ordTC.attach(ordTV)            ; associate a TCursor with the table
else
   msgStop("Stop", "Could not open the table.")
endIf
```

If a form contains a UIObject (for example, a field, table frame, or multi-record object) bound to a table, you can use the TCursor type method **attach** to associate a TCursor with the UIObject, and by so doing, associate the TCursor with the underlying table. For

example, suppose a form contains a table frame named *SALES* bound to SALES.DB. The following statements point *salesTC* to SALES.DB:

```
var salesTC TCursor endVar
salesTC.attach(SALES)
```

Note This section assumes you are familiar with table windows, table frames, and multi-record objects, as described in the *User's Guide*. See also Chapters 20 and 13 of this manual.

A TCursor gets its data and structure from the underlying table, not from the displayed view. So, for example, suppose a form contains a single field, bound to the Phone field in the *Customer* table. The next statement associates the TCursor *custTC* with all fields in the *Customer* table, not just with the displayed Phone field:

```
custTC.attach(Phone)
```

If you attach a TCursor to a table window or a UIObject, the TCursor won't know about any data that has not been committed, so it's possible to have one record out of date after the call to **attach**. Rotating columns in a table view does not affect the field order known to the TCursor.

Attaching to unlinked tables

Figure 18.1 shows how you can attach TCursors to UIObjects where the UIObject is bound to a single unlinked table. The figure shows representations of three forms. Each form is bound to CUSTOMER.DB. In each case, attaching a TCursor to a UIObject associates the TCursor with all fields in CUSTOMER.DB.

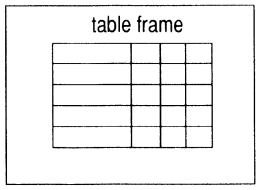
In the first two forms, which contain a table frame and a multi-record object respectively, the following statements make the TCursor *custTC* point to the *Customer* table. (By default, when a table frame or multi-record object is bound to a table, it takes the name of that table. The table frame and the multi-record object in the figure are bound to CUSTOMER.DB, so they take the name CUSTOMER.)

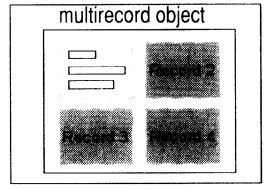
```
var custTC TCursor endVar
custTC.attach(CUSTOMER)    ; associate custTC with the UIObject named CUSTOMER
```

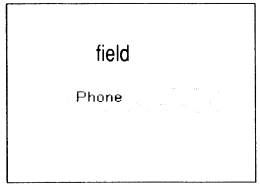
In the third form, which contains a field object bound to the Phone field of the *Customer* table, the next statements accomplish the same thing: they make *custTC* point to the *Customer* table—the whole table, not just the Phone field. (By default, when a field object is bound to a field in a table, it takes the name of the field. In the figure, the field object is bound to the Phone field of the *Customer* table, so it takes the name Phone.)

```
var custTC TCursor endVar
custTC.attach(Phone)       ; associate custTC with the UIObject named Phone
```

Figure 18.1 Attaching a TCursor to a UIObject bound to an unlinked table







The table frame and the multirecord object in the two forms above are bound to CUSTOMER.DB. The following statements associate the TCursor with the underlying table:

```
var custTC TCursor endVar
custTC.attach(CUSTOMER)
```

The field object in the form at left is bound to the Phone field of CUSTOMER.DB. These statements do the same thing: make the TCursor point to CUSTOMER.DB:

```
var custTC TCursor endVar
custTC.attach(Phone)
```

Attaching to linked tables

Figure 18.2 shows how you can attach TCursors to UIObjects where the UIObject is a table frame or multirecord object bound to linked tables.

Note If you attach a TCursor to a table frame or multirecord object bound to linked tables with a single-valued detail table, the TCursor gets the structure and data of the master table only.

Figure 18.2 shows representations of two forms. The data model for the first form, which contains a table frame, specifies a one→one link between EMPLOYEE.DB and DEPT.DB, with EMPLOYEE.DB as the master table. This table frame displays linked records from the *Employee* table and the *Dept* table. The following statements associate the TCursor variable *thisTC* with the UIObject named EMPLOYEE, so *thisTC* points to the *Employee* table *only*—not because the object's name is EMPLOYEE, but because EMPLOYEE.DB is the master table.

```
var thisTC TCursor endVar
thisTC.attach(EMPLOYEE)
```

In the second form, which contains a multi-record object named CUSTOMER and a table frame named ORDERS, the data model specifies a one→many link between CUSTOMER.DB and ORDERS.DB, with CUSTOMER.DB as the master table. As in the previous example, these statements associate *thisTC* with the object named CUSTOMER, so *thisTC* points to the *Customer* table only:

```
var thisTC TCursor endVar
thisTC.attach(CUSTOMER) ; associate thisTC with MRO named CUSTOMER
```

Similarly, the following statements associate the TCursor variable *thisTC* with the object named ORDERS, so *thisTC* points to the *Orders* table:

```
var thisTC TCursor endVar
thisTC.attach(ORDERS) ; associate thisTC with the table frame named ORDERS
```

Note that this **attach** statement associates *thisTC* with the *Orders* table contained in the table frame. It attaches to the detail set for the current master record. For example, suppose the first record is for a customer who has placed three orders. The following statements would display a 3 in the status bar:

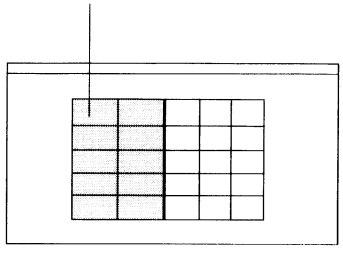
```
var thisTC TCursor endVar
thisTC.attach(ORDERS)
message(thisTC.nRecords()) ; displays 3
```

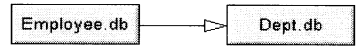
But, when you move to another record, this time for a customer who has placed seven orders, executing those same statements displays a 7 in the status bar, because the TCursor is attached to a different detail set.

Figure 18.2 Attaching a TCursor to a UIObject bound to linked tables

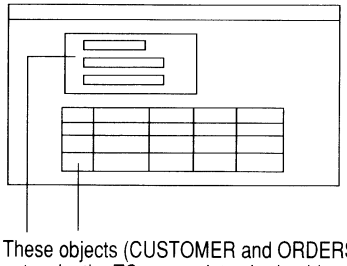
This table frame (EMPLOYEE) displays records from EMPLOYEE.DB and DEPT.DB. But the following statement makes the TCursor point to records in EMPLOYEE.DB only (shaded in the figure), because EMPLOYEE.DB is the master table:

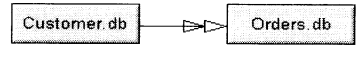
custTC.attach(EMPLOYEE)





EMPLOYEE.DB and DEPT.DB are linked one→one. EMPLOYEE.DB is the master table.





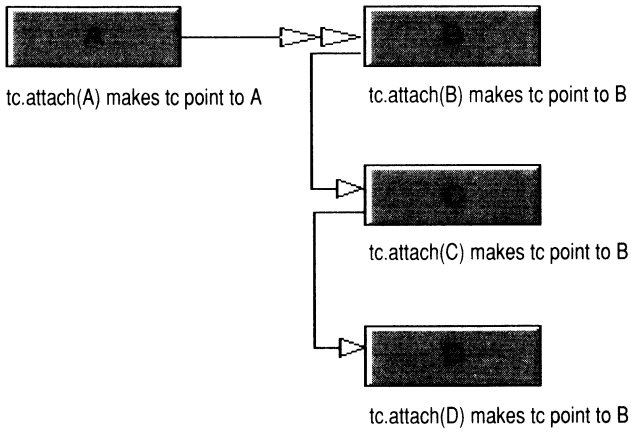
CUSTOMER.DB and ORDERS.DB are linked one→many. CUSTOMER.DB is the master table.

These objects (CUSTOMER and ORDERS) are linked, but attaching a TCursor to one object does not make the TCursor point to both tables—a TCursor can point only to one table at a time. Attaching to ORDERS attaches to the current detail set contained in ORDERS.

Attaching to multiple linked tables

Figure 18.3 shows the data model diagram for tables linked one→many→one→one. The figure also lists four **attach** statements and tells which table the TCursor points to after each call to **attach**.

Figure 18.3 Attaching a TCursor to linked tables



Using TCursors and tables

This section describes how to use TCursors to work with tables. It shows examples of the methods listed in Table 18.1 and describes many common tasks you'll perform with TCursors.

Note Lesson 8 of the ObjectPAL tutorial in Chapter 2 presents two techniques for using tables and TCursors to program drop-down lists.

Editing with TCursors

To change data in a TCursor, set the TCursor in edit mode by using the **edit** method. When your changes are complete, post them to the underlying table by moving off the record or calling **postRecord** or **unLockRecord**. If you don't want to keep your changes, use **cancel**. Changes are processed one record at a time.

When you finish editing, use **endEdit** to switch out of Edit mode. When you're finished working with the TCursor, use **close**.

The following example opens a TCursor onto the *Sales* table and puts the table into Edit mode. Next, it searches the table for a value of "Morland" in the CoName field and changes it to "Borland". Then it takes the table out of Edit mode and closes the TCursor, ending the association between the TCursor and the table.

```
var tc TCursor endVar
tc.open("sales.db")
tc.edit()
if tc.locate("CoName", "Morland") then
   tc.CoName = "Borland"
endIf
tc.endEdit()
tc.close()
```

Specifying fields

ObjectPAL provides several ways to specify fields in a TCursor, Table, table frame, TableView, or multi-record object. If you know the field's name, you can use dot notation. For example, the following code assigns the value of the Quantity field to the variable *myQty*:

```
var
   tc TCursor
   myQty LongInt
endVar
tc.open("orders.db")
myQty = tc.Quantity  ; specify the Quantity field
```

Similarly, to assign a value to the Quantity field, you could do this:

```
tc.Quantity = 120
```

Field names aren't bound by the same restrictions as ObjectPAL variables. To work with a field named Customer Credit Number, which contains spaces, enclose the name in double quotes:

```
myQty = tc."Customer Credit Number"
```

You can also use variables and expressions to specify fields by enclosing them in parentheses, and you can specify a field by number, counting from left to right. For example,

```
var
   tc, otherTC TCursor
   s String
   v AnyType
endVar
tc.open("orders.db")

s = "Customer Credit Number"
v =tc.(s)                                    ; gets Customer Credit Number
v = tc.("Customer " + "Credit " + "Number") ; does the same
v = tc.(2)                ; gets the value of the second field from the left
v = tc.(myMethod())       ; myMethod() must return a value to specify a field

otherTC.open("other.db")
v = tc.(otherTC.someField) ; the value of otherTC.someField specifies a field
                           ; for example, if otherTC.someField = "Quantity"
                           ; then v =tc.Quantity
```

Getting values from a table

The syntax for reading a value from a field is

valueVar = tableVar.fieldID

fieldID contains the field name, or an integer representing the field's position in the underlying table counting from left to right, or an expression that evaluates to the field name or position.

For example, the following code shows both techniques:

```
var
   tc TCursor
   partName String
   qty LongInt
endVar

if tc.open("parts.db") then
   partName = tc."Part name" ; reads data from field "Part name"
   qty = tc.(4)              ; reads from field 4 (fourth from the left)
endIf
```

Changing values

The syntax for changing values in a table is

tableVar.fieldID = newValue

fieldID is the field name or the field number (fields are numbered from left to right, starting with 1). The following code searches the Part name field of PARTS.DB and replaces every occurrence of "sprocket" with "TurboSprocket".

```
var
   tc TCursor
   oldName, newName, tableName, fieldName String
endVar

oldName = "sprocket"
newName = "TurboSprocket"
tableName = "parts.db"
fieldName = "Part name"

if tc.open(tableName) then
   tc.edit()
   scan tc for tc.fieldName = oldName :
      tc.(fieldName) = newName              ; this line changes the field value
   endScan
   tc.endEdit()
else
   msgStop("Stop", "Couldn't open " + tableName)
endIf
```

Adding records to a table

To add records to a table, open the table, put it into Edit mode, insert a record, put values into the record, and end Edit mode. These steps are shown in the following example:

```
var
   tc TCursor
   endVar
if tc.open("parts.db") then    ; open the table
   tc.edit()                   ; put it into Edit mode
   tc.insertRecord()           ; insert a blank record
```

```
    tc."Part name" = "Sprocket" ; put values in
    tc."Quantity" = 348
    tc.endEdit()                ; end Edit mode
endIf
```

Using TCursors and UIObjects

This section presents some examples showing how you can use TCursors and UIObjects. The first example compares the time it takes to move through a table one record at a time using a TCursor and using a table frame. The second example shows how you can use TCursors with UIObjects to improve performance when you search a table.

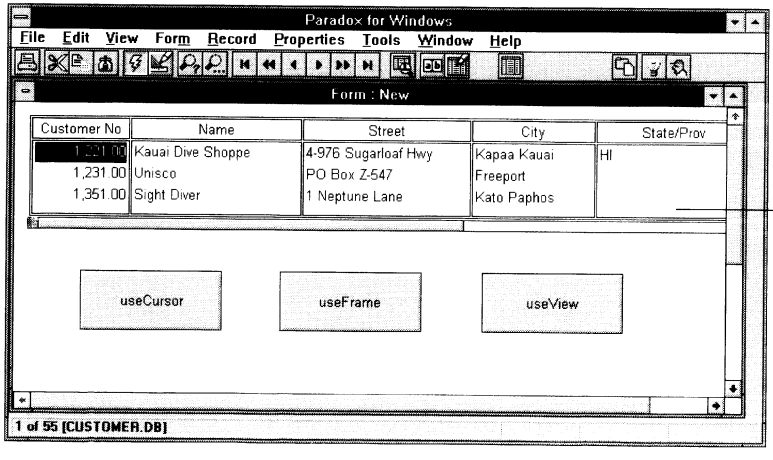
Stepping through records in a table

By using TCursors and table frames together, you can process data behind the scenes before displaying the results to the user. (A table frame is a UIObject.) This technique often improves performance. The key methods are the TCursor type method **attach**, which binds a TCursor to a UIObject, and the UIObject type method **moveTo**, which synchronizes the UIObject to the TCursor.

Figure 18.4 shows a form containing a table frame (*Customer*) and three buttons. The table frame is bound to a table named CUSTOMER.DB. (When a UIObject is bound to a table, the object takes its name from the table by default. Hence, the table frame's name is *Customer*, because it's bound to the *Customer* table.)

All three buttons (named *useCursor*, *useFrame*, and *useView*) accomplish the same thing: they step through the records one at a time and display the last record in the table. However, the methods for *useFrame* and *useView* take longer to complete the task because they update the screen after each move to the next record. The method for *useCursor* updates the screen only once, so it finishes much sooner.

Figure 18.4 Table frame vs. TableView vs. TCursor



Methods attached to buttons

The following methods are attached to the *useCursor*, *useFrame*, and *useView* buttons.

useCursor

The following method is attached to *useCursor*. It reaches the last record more quickly because it updates the screen only once.

```
method pushButton (var eventInfo Event)
   var
      i SmallInt
      tc TCursor
      start, stop, delta LongInt
   endVar
   start = CPUClockTime()
   tc.attach(customer)           ; binds insertion point to customer table frame
   for i from 1 to tc.nRecords()
      tc.nextRecord()            ; steps through the records
   endFor
   customer.moveToRecord(tc)     ; synchronizes insertion point and frame
                                 ; to display the last record
   stop = CPUClockTime()
   delta = stop - start
   msgInfo("TCursor time", delta)
endMethod
```

useFrame

The following method is attached to *useFrame* and is slower than *useCursor* because it updates the screen after each move.

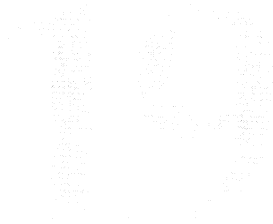
```
method pushButton (var eventInfo Event)
   var
      i SmallInt
      start, stop, delta LongInt
   endVar
   start = CPUClockTime()
   for i from 1 to customer.nRecords() ; frame's name is CUSTOMER by default
      customer.nextRecord()
   endFor
   stop = CPUClockTime()
   delta = stop - start
   msgInfo("Table frame time", delta)
endMethod
```

useView

The following method is attached to *useView* and is also slower than *useCursor* because it updates the screen after each move.

```
method pushButton (var eventInfo Event)
   var
      custTV TableView
      i SmallInt
      start, stop, delta LongInt
   endVar
```

```
    custTV.open("customer.db")
    start = CPUClockTime()
    for i from 1 to custTV.nRecords ; uses TableView property, not a method
       custTV.rowNo = i  ; uses rowNo property to move to row i in table window
    endFor
    stop = CPUClockTime()
    delta = stop - start
    msgInfo("Table window time", delta)
endMethod
```



Working with databases

A Database variable provides a *handle*—a variable you can use to specify and work with a database. For example, some methods take a Database variable as an argument, and other methods operate on the database specified by a Database variable. When you're working with Paradox and dBASE tables, *database* and *directory* are synonymous.

An *alias*, in broad terms, describes a path. When you're working with Paradox and dBASE tables, an alias is a character string that represents a directory path—in other words, when you use an alias, you're specifying a directory. Paradox uses the aliases WORK and PRIV to refer to your working and private directories, respectively.

Although database variables seem similar to aliases, they're used to refer to a collection of tables; aliases are directory shortcuts indicating where files are stored.

Using an alias

Suppose you want to work with some Paradox tables in the database C:\APPS\PARADOX\TABLES\MASTDATA. Rather than type the full path to open the table, you can create an alias by using Paradox interactively (choose File | Aliases) or by using the Session type method **addAlias**. For example, the following statement creates an alias named MAST to represent the C:\APPS\PARADOX\TABLES\ MASTDATA directory. (In this statement, double backslashes are required in the path because it's a quoted string.)

```
addAlias("mast", "Standard", "C:\\APPS\\PARADOX\\TABLES\\MASTDATA")
```

Once you create an alias, you can use it as a prefix for file access. For example, the following statement opens the *Orders* table in the directory C:\APPS\PARADOX\TABLES\MASTDATA, because you've defined the alias MAST to represent that path.

```
ordersTC.open(":mast:Orders.db")
```

The colons around the alias name in the **open** statement are required.

The other advantage to using aliases is portability; instead of hard-coding every directory path for each user's system, you can simply redefine the alias. For example, suppose one user stores data in C:\APPS\PARADOX\TABLES\MASTDATA, and another user stores them in D:\TABLES\MAST. If you use aliases, specify the appropriate paths for them, and the rest of your code will run without modification.

Note If you don't specify an alias, Paradox uses WORK.

Opening a database

When you start a Paradox application, Paradox opens the *default database* (the working directory). The default database stores the path to the current working directory. If you want to work with those tables only, you don't have to explicitly open any other database. To work with tables stored elsewhere, declare a Database variable and open it to create a handle to another database. (You could specify the full path to each table each time you wanted to use it, or use an alias with the name, but code that uses Database variables is easier to maintain.)

Using **open** and an alias, you can specify which database to open, as shown in the following example:

```
var
   custInfo Database
endVar
addAlias("CustomerInfo", "Standard", "D:\\pdoxwin\\tables\\custdata")
custInfo.open("CustomerInfo") ; opens the CustomerInfo database
                              ; CustomerInfo must be a valid alias
```

Paradox now knows about two databases: the default database and CustomerInfo. The variable *custInfo* is a *handle* to the CustomerInfo database—that is, you can use *custInfo* in statements to refer to the CustomerInfo database. For example, suppose you have two files named ORDERS.DB (one in your working directory and one in CustomerInfo), and you want to find out if these files are tables. The following example tests ORDERS.DB in the working directory first, then uses *custInfo* as a handle for the CustomerInfo database and tests ORDERS.DB there:

```
var
   custInfo Database
endVar
addAlias("CustomerInfo", "Standard", "D:\\pdoxwin\\tables\\custdata")
custInfo.open("CustomerInfo")

if isTable("orders.db") then           ; test ORDERS.DB in the default database
   msgInfo("Working directory", "ORDERS.DB is a table.")
endIf

if custInfo.isTable("orders.db") then  ; use myDB as a handle for
                                       ; the CustomerInfo database
   msgInfo("CustomerInfo", "ORDERS.DB is a table.")
endIf
```

If you use **open** but don't specify a database, Paradox assumes you want to work in the default database. For example, the following code gives you a handle for the default database, which you could pass to a custom method that requires a database handle:

```
var defaultDb Database endVar
defaultDb.open() ; opens the default database
```

Using a handle to the default database can also make code more readable, especially when you're working with several databases at once.

Chapter 20

Displaying output

Display managers let you control the windows that present information to a user. Using display manager objects and their methods, you can control the size, shape, position, and appearance of the windows a user works with, including the Desktop and Form, Report, and Table windows.

This chapter describes the ObjectPAL display managers, shown in Table 20.1, and explains how to use their methods.

Table 20.1 Display managers

Type	Description
Application	The Paradox Desktop
Form	A Paradox form displayed in its own window
Report	A Paradox report displayed in its own window
TableView	A table displayed in its own window

This chapter also show how to use **format** to control output strings.

Application: The Desktop window

An Application refers to the Desktop window of the current Paradox application. Although you can have several applications running at the same time, they don't communicate with or operate on each other.

Use Application methods to control the Desktop window. For example, you can minimize a form but keep the Desktop displayed at full size. You can also minimize all open Paradox windows by using an Application variable to minimize the Desktop.

Application type methods are a subset of the methods for the Form type (described in detail in the "Form: A window to data" section of this chapter). You can use them to control the application window's size, position, and appearance. For example,

```
var ap Application endVar
ap.hide() ; makes the Desktop invisible
sleep(1000)
ap.show() ; makes the Desktop visible
sleep(1000)
ap.bringToTop() ; displays the Desktop above all other windows
ap.setPosition(100,100,2000,2100) ; sets the upper left corner (100, 100)
                                  ; the width (2000) and the height (2100)
```

Form: A window to data

A form in Paradox plays two roles:

- Display manager: a window that displays data. You can use Form type methods to open a form, specify display attributes, and close the form.
- Design object: the highest level in the containership hierarchy. A form has built-in methods you can attach code to, and you can attach custom code and variables to make the methods available to all design objects that form contains.

This section describes both roles, but you should also read Chapter 13 for information about working with design objects.

This section also discusses techniques for displaying dialog boxes, including how to use forms as dialog boxes and how to use the dialog boxes built into Paradox. This section covers the following topics:

- Using the form as a display manager
- Using the form as a design object

Using the form as a display manager

To work with a form as a display manager, first declare a variable of type Form and associate it with a Paradox form. Then you can use the Form variable as a handle to the form—in other words, you can use the Form variable in ObjectPAL code to manipulate the actual form displayed onscreen.

The most common way to associate a Form variable with a Paradox form is to use an **open** statement. For example, the following code declares a Form variable named *custForm*, uses an **open** statement to associate the variable with an actual form, and then uses the variable as a handle to manipulate the form:

```
var
   custForm Form              ; declare the variable
endVar

custForm.open("customer") ; associate the variable with an actual form
custForm.minimize()          ; use the variable as a handle to manipulate the
                             ; form
```

Within a method, you can associate one Form variable with more than one form, but only with one form at a time. For example, the following statements open CUSTOMER.FSL and ORDERS.FSL, then display ORDERS.FSL as an icon:

```
var
   formVar Form
endVar

formVar.open("customer")
formVar.open("orders")
formVar.minimize() ; minimize ORDERS.FSL
```

Table 20.2 lists commonly used Form type methods. Use these methods to work with a form as a display manager.

Table 20.2 Commonly used Form type methods

Method	Description
create	Creates a blank form in a design window.
save	Saves a form. You can save forms only in a design window.
load	Loads a form from disk in a design window.
run	Runs a form (equivalent to choosing Form | View Data).
open	Opens and runs a form
design	Switches to a design window (equivalent to choosing Form | Design).
isDesign	Reports whether the form is in a design window.

Show and hide

To control whether a Form is visible to the user, use **show**, **hide**, and **bringToTop**. The following example shows code attached to a form named *form0* that controls two other forms, *form1* and *form2*.

```
var
   form1, form2 Form    ; declare 2 Form variables to use as handles
endVar

form1.open("map.fsl")   ; displays form1
form2.open("sites.fsl") ; displays form2 on top of form1

form1.bringToTop()      ; now form1 is on top

form1.hide()            ; form1 is invisible, form2 is visible
sleep(2000)
form1.show()            ; form1 is visible again

bringToTop()            ; displays form0 on top

form1.close()           ; close form1
form2.close()           ; close form2

close()                 ; close form0 (the form where this code is executing)
```

As the previous code shows, some Form type methods can also be called as procedures. For example, to call **hide** as a method, use a Form variable as a handle to another form, as in

```
var
   otherForm Form
endVar
otherForm.open("customer.fsl") ; use otherForm as a handle to CUSTOMER.FSL
otherForm.hide()
```

To call **hide** as a procedure, you don't use a handle: Form type procedures operate on the current form (the form where the code is executing). For example, this statement hides the current form:

```
hide()
```

Closing a form

When you're finished with a form, use **close**. For example,

```
var formTwo endVar
formTwo.open("trips.fsl")

; do your processing here,
; then, when you're finished do this:

formTwo.close() ; closes formTwo
close() ; closes formOne
```

Setting properties

In a design window, you can use Paradox interactively to set many form properties, including title, frame style, caption properties, and scroll bars. (See the *User's Guide* for more information about setting properties interactively.) In ObjectPAL code, you can use Form type methods to set many properties, and you can use dot notation to access many properties directly. For example, you can use the Form type methods **getTitle** and **setTitle** to manipulate the text in the Form window's title bar:

```
var
   myForm Form
   oldTitle, newTitle String
endVar
myForm.open("map.fsl")
oldTitle = myForm.getTitle()
; puts current window title into oldTitle
newTitle = "I have changed the title."
myForm.setTitle(newTitle)
; puts "I have changed the title." into the window title bar.
```

You can access Form properties directly using dot notation. For example,

```
var
   myForm  Form
   myPos, mySize Point
endVar
if myForm.open("orders.fsl") then
   myPos = myForm.position     ; get the Position property
```

```
    mySize = myForm.size        ; get the Size property
    myForm.title = "Order form" ; set the Title property
  endIf
```

Controlling size

In addition to manipulating the Size and Position properties, you can use Form type methods to manipulate a form's size and position. For example, you can expand a form to fill the entire screen using **maximize**, shrink it to an icon using **minimize**, or specify any position in between using **setPosition**. To get information about a form's size, use **isMaximized**, **isMinimized**. For example, suppose the following code is attached to a form:

```
var
   proteus Form
endVar
proteus.open("map.fsl") ; reads form from disk
proteus.minimize() ; shrinks it to an icon
proteus.maximize() ; shows it full screen
if proteus.isMaximized() then
   proteus.setPosition(20,20,6000,5000)
;  sets upper left corner, width, and height (in twips)
endIf
```

This code declares a Form variable named *proteus* to use as a handle—that is, it uses the variable *proteus* in code to manipulate the form window on the screen. The call to **setPosition** takes arguments that specify the coordinates of the upper left corner of the form, the width, and the height. All values related to the screen are expressed in *twips*. A twip is a unit of measurement equal to 1/1440 of an inch (1/20 of a printer's point).

For more information about working with form properties, see Chapter 26.

Using the form as a design object

The form can function as a design object. As such, it is the highest level in the containership hierarchy: it contains all other objects. Code attached to a form is visible to every object the form contains.

Note In code attached to a form, *self* refers to the form as a UIObject (design object), not as a Form (display manager).

The topics in this section describe how to attach code to a form and how to set the properties of design objects in other forms. For more information about design objects and the containership hierarchy, see Chapter 6.

Editing methods

You can attach and edit methods for a form, just as you can for a button. Here are the steps:

1 In the Form Design window, choose Properties | Form | Methods to display the Method Inspector. (Shortcut: Press *Esc* repeatedly until no objects are selected, then press *Ctrl+Spacebar* to display the Method Inspector.)

2 Select one or more built-in methods—or attach custom code, for example, a custom method or custom procedure—as you would for any other object.

The Form type also has methods you can use to control a form's appearance.

Setting properties of other objects

You can use the containership hierarchy to manipulate properties of objects in other forms. For example, suppose code in *formOne* opens *formTwo*, and suppose *formTwo* contains a text box (*theText*) and an ellipse (*theEllipse*). Methods in *formOne* can set properties of *theText* and *theEllipse* by specifying the containership path. For example,

```
var
   formTwo Form
endVar
formTwo.open("orders.fsl")
formTwo.theText.text = "This text was sent by formOne."
formTwo.theEllipse.color = "Red"
```

TableView: Display rows and columns

A Table window displays a table in its own window. It is distinct from a table frame, which is a UIObject placed in a form, and from a TCursor, which is a programmatic construct, a pointer to the data in a table. See the *User's Guide* for information about working with Table windows interactively.

When you declare a TableView variable, you simultaneously create a handle to the Table window. This handle is a variable you can refer to in your code to manipulate the Table window. For example, the following statements declare a TableView variable *ordersTV*, and then use the variable to set the position of the Table window for the *Orders* table.

```
var
   ordersTV TableView
endVar

if ordersTV.open("orders.db") then
   ordersTV.setPosition(100, 200, 3000, 12000)
else
   msgStop("Stop", "Couldn't open ORDERS.DB")
endIf
```

This section discusses the following topics:

- TableView type methods
- Properties of Table windows
- Table windows and TCursors

TableView type methods

TableView type methods are a subset of the methods defined for the Form type. You can use them to control the Table window's size, position, and appearance. You can also use the TableView type **action** method with action constants to "drive" a Table window. For example, the following statements open a Table window and scroll through the first five records:

```
var
   ordersTV TableView
   i SmallInt
endVar
if ordersTV.open("orders.db") then     ; display a Table window
   for i from 1 to 5
      ordersTV.action(DataNextRecord)  ; scroll through 5 records
      sleep(500)
   endFor
else
   msgStop("Stop", "Could not open the table.")
endIf
```

Using TableView to edit data

You can use ObjectPAL to edit the data in a Table window by using a TableView variable, as shown in the following example:

```
var
   ordersTV TableView
   i SmallInt
endVar
if ordersTV.open("orders.db") then      ; display a Table window
   ordersTV.action(DataBeginEdit)
   for i from 1 to ordersTV.nRecords    ; nRecords is a property, so no ()
      if ordersTV.PartName = "Widget" then
         ordersTV.PartName = "Gadget"
      endIf
      ordersTV.action(DataNextRecord)
   endFor
   ordersTV.action(DataEndEdit)
else
   msgStop("Stop", "Could not open the table.")
endIf
```

Using wait with a Table window

The TableView type method **wait** behaves differently from the Form type method **wait** in the following respects:

- The TableView type method **wait** cannot take a return value. Because you can't attach code to a Table window, there is no **formReturn** method for the TableView type. The only way to return from a TableView **wait** is to close the table interactively.
- After you close the table interactively, you still must call **close** to remove it from the screen. This gives you one last chance to do things to the table before it closes. For example, you could check values.

```
method pushButton(var eventInfo Event)
   var
      custTV TableView
   endVar

   custTV.open("customer.db")    ; display the table
   custTV.action(DataBeginEdit)
   custTV.wait()                 ; wait for the user to close it

   if custTV."Name".value = "" then ; check the value of the Name field
      msgInfo("Name", "Enter a name.")
      custTV.bringToTop()
   else
      custTV.close()
   endIf
endMethod
```

Properties of Table windows

You can use the TableView variable to manipulate Table window properties in three main areas:

- The Table window as a whole—for example, background color, grid style, number of records, and the value of the current record
- The field-level data in the table—for example, font, color, and display format
- The Table window title bar—for example, text, font, color, and alignment

The following code opens a Table window, then sets some properties, moves around in the Table window, and reports the values of certain properties:

```
var
   custTV  TableView
   propVal AnyType
endVar

if custTV.open("customer.db") then

   ; these commands affect the table view as a whole

   custTV.Color = LightBlue                ; change color of the grid
   custTV.GridLines.LineStyle = DottedLine ; change line style of grid
   custTV.CurrentRecordMarker.Show = True  ; display the current record marker
   custTV.CurrentRecordMarker.Color = Red  ; change color of record marker

   ; these commands affect the field-level data

   propVal = custTV.NRecords               ; get number of records, read only
   if propVal > 5 then
      custTV.RowNo = 5                     ; go to row number 5
   endIf
   custTV.fieldNo = 4                      ; go to field 4

   ; this command affects the title bar, by adding the number of records to the
```

```
    ; current message.

    custTV.setTitle(CustTV.GetTitle() + " (" + String(propVal) + " records)")

  else
    beep()
    msgInfo("Oops!", "Couldn't open CUSTOMER.DB")
  endIf
```

For a complete list of Table window properties, refer to Appendix B.

Table windows and TCursors

You can relate Table windows and TCursors in two ways:

- Associate a TCursor with a Table window
- Synchronize a Table window with a TCursor

These relationships are discussed in the following sections.

Associating a TCursor with a Table window

You can use the TCursor type method **attach** to associate a TCursor with the table displayed in a Table window. For example, the following statements declare a TCursor variable named *ordTC*, then call **attach** to associate *ordTC* with the *Orders* table displayed in a Table window opened with the TableView variable *ordTV*:

```
var
   ordTC TCursor
   ordTV TableView
endVar

if ordTV.open("orders.db") then   ; open a Table window
   ordTC.attach(ordTV)            ; associate a TCursor with the table
else
   msgStop("Stop", "Could not open the table.")
endIf
```

Once the TCursor is associated with the Table window, you can use TCursor methods to operate on the data in the table. This often improves performance, because the processing is done in system memory, without any display overhead.

Synchronizing a Table window with a TCursor

The TableView type provides a **moveToRecord** method that synchronizes a Table window to a TCursor. In other words, a **moveToRecord** statement makes a Table window display the data in a TCursor. For example, the following statements attach a TCursor to a Table window, then locate a record in the TCursor, then synchronize the Table window to the TCursor:

```
var
   custTC TCursor
   custTV TableView
endVar
```

```
custTV.open("customer.db")    ; open a Table window
custTC.attach(custTV)         ; associate a TCursor with the Table window

if custTC.locate("Name", "Smith") then   ; Search the TCursor for a value.
   custTV.moveToRecord(custTC)           ; If the value is found,
else                                     ; synchronize Table window to TCursor
   msgInfo("Locate failed.", "Could not find Smith.")
endIf
```

For information about using **attach** and **moveToRecord** with TCursors and UIObjects, refer to Chapter 18.

Using format to control output

You can use **format** to specify how strings are output. You can use **format** to

- Control numeric precision
- Justify and align data
- Change case of letters
- Specify the form in which dates display
- Suppress leading blanks of values

Also, you can combine different formats by separating the specification with commas. For example, "W6,AL" specifies left alignment within a width of 6. You can also combine editing formats following a single *E*: for example, "E$C" for a floating dollar sign with commas every three digits.

The following sections give some examples of how to use **format**. Complete specifications are listed with the entry for **format** for the String type in the online ObjectPAL Help.

Important When you format an unbound field object, that format affects values set by ObjectPAL but not values entered interactively. To make sure all values are formatted correctly, attach code like the following to the field object's built-in **changeValue** method.

```
method changeValue(var eventInfo ValueEvent)
   try
      eventInfo.setNewValue(Date(eventInfo.newValue()))
   onFail
      message("Invalid date")
      eventInfo.setErrorCode(CanNotDepart)
   endTry
endMethod
```

This example assumes you're working with a date format; it casts the new value as a Date and assigns it to the field object. You can use this technique to work with other data types, too. For example, for a numeric format, cast the new value as a Number.

Width

Width (W) specifications control the total number of characters that a displayed or printed value can have. Width values can be between 1 and 255. If you don't specify a width, the entire data value is output.

If the formatted expression is a number, you can also control the number of digits to the right of the decimal point. The total width must include space for the entire value, including

- All digits to the left and right of the decimal point
- The decimal point itself
- The sign (whether shown or not)
- Any other characters, such as whole number separators and dollar signs

The number of decimal places cannot exceed 15.

If the length of a number or date value exceeds the format width, a string of asterisks is output. If the width specified isn't enough for the number of decimal places in the expression, the resulting value is rounded. If the length of an alpha, logical, memo, point, or string value exceeds the format width, the value is truncated.

Here are some examples of width specifications:

```
message(FORMAT("w6","This is a test"))   ; displays This i
message(FORMAT("w6",1234567)             ; displays ******)
message(FORMAT("w1",(5 = 5)))            ; returns True, displays T
message(FORMAT("w9.2",1234.567))         ; displays   1234.57
```

Alignment

Alignment specifications adjust the placement of an output value within its format width. Thus, when specifying alignment, you must specify a width as well; if you don't, or if the width is equal to the length of the value, alignment will have no effect.

If alignment is not specified, strings, memo, point, and logical values are aligned to the left, while numbers, times, and dates are aligned to the right.

Here are some examples of alignment specifications:

```
message(FORMAT("w20,ac","This is"))       ; displays        This is
message(FORMAT("w20,ac","The Title"))     ; displays       The Title
message(FORMAT("w20,ac","Of the Book"))   ; displays     Of the Book
message(FORMAT("w20,al",123456))          ; displays 123456
message(FORMAT("w20,ar",123456))          ; displays               123456
```

Case

Case specifications control the way values are capitalized. For example, the initial capitalization option CC capitalizes the first letter of each word; a word, in this case, is defined as a string of characters up to but not including the next non-letter character.

Here are some examples of case specifications:

```
message(FORMAT("cu","the quick brown fox"))    ; displays THE QUICK BROWN FOX
message(FORMAT("cl","JUMPS OVER THE LAZY"))    ; displays jumps over the lazy
message(FORMAT("cc","dOG."))                   ; displays DOG.
message(FORMAT("cc","widgets'r us " + "too"))  ; displays Widgets'R Us Too
```

Edit

Edit specifications control the way numbers are shown. You can combine several edit specifications following a single E prefix. So, if you wanted both a $ sign and commas to separate whole digits in a number, you would use the format specification "E$C". Although in this situation the input data type could be Money(**$**), Number (**N**), LongInt (**L**), or SmallInt (**S**), you could place the output only in an alpha field because of the dollar sign ($) and comma.

Here are some examples of edit specifications:

```
x = 34567.89
message(FORMAT("w10.2, e$c", x))          ; displays $34,567.89
message(FORMAT("w10.2, e$ci", x))         ; displays $34.567,89
message(FORMAT("w13.2, e$c", x))          ; displays    $34,567.89
message(FORMAT("w14.2, e$cb, al", x))     ; displays $   34,567.89
message(FORMAT("w15.2, e$cz, al", x))     ; displays $000034,567.89
message(FORMAT("w15.2, e$c*, al", x))     ; displays $****34,567.89
```

The last option is often used for writing checks, to protect against unauthorized insertion of digits into a number. When you include an edit specification, use a width specification in the same FORMAT() call.

You might want to use a sign specification (described in the next section) as well as an edit specification to format numbers.

Sign

Sign specifications determine how positive and negative values are distinguished. Sign specifications apply only to numeric values, and you can use only one at a time.

Here are some examples of sign specifications:

```
x = -3456.12
message(FORMAT("w8.2, s+", x))          ; displays -3456.12
message(FORMAT("w11.2, e$c, sc", x))    ; displays $3,456.12CR
message(FORMAT("w14.2, e$c*, sp", x))   ; displays ($***3,456.12)
message(FORMAT("w13.2, e$c*, s+", x))   ; displays $-***3,456.12
message(FORMAT("w14.2, e$c*, sd", x))   ; displays $***3,456.12CR
```

DB (debit) and CR (credit) are used primarily in accounting applications.

You can use sign and edit specifications together. The three preceding examples show the order in which sign and edit items are output:

1. Leading parenthesis, if specified
2. $ sign, if specified

3 + or – sign, if specified
4 Fill characters (blanks, zeros, or *), if necessary and specified
5 The number itself
6 DB or CR, if specified
7 Closing parenthesis, if specified

Date

Date specifications output date values in any of the Paradox date formats. For more information see the online ObjectPAL Help. If you use both a width and date specification, the width should be wide enough to fit the date format; otherwise, Paradox displays a string of asterisks.

Here are some examples of date specifications:

```
da = Date("1/6/92")
message(format("d2", da))       ; displays January 6, 1992
message(format("d7", da))       ; displays 06-Jan-1992
message(format("d11", da))      ; displays 92-01-06
message(format("d7,w5", da))    ; displays *****

message(format("DOLW1",da))     ; displays Mon, 01 06, 92
message(format("DOLW2",da))     ; displays Monday, 01 06, 92
message(format("DOLWL",da))     ; displays Monday, 01 06, 92
message(format("DOLWS",da))     ; displays Mon 01/06/92

message(format("DM1",da))       ; displays 1/06/92
message(format("DM2",da))       ; displays 01/06/92
message(format("DM3",da))       ; displays Jan/06/92
message(format("DM4",da))       ; displays January/06/92
message(format("DML",da))       ; displays January/06/92
message(format("DMS",da))       ; displays 1/06/92

message(format("DO(%M/%D/%Y)",da)) ; displays 01/06/92
message(format("DO(%D/%M/%Y)",da)) ; displays 06/01/92
message(format("DO(%Y/%D/%M)",da)) ; displays 92/06/01
message(format("DO(%D/%Y/%M)",da)) ; displays 06/92/01
message(format("DO(%D%%Y%%M)",da)) ; displays 06%92%01

message(format("DO(%M-%D/%Y)",da)) ; displays 01-06/92
message(format("DO(%D/%M%%Y)",da)) ; displays 06/01%92
message(format("DO(%Y$%D*%M)",da)) ; displays 92$06*01
message(format("DO(%D@%Y!%M)",da)) ; displays 06@92!01
```

Logical

Logical specifications substitute the logical values Yes/No or On/Off for the default values of True/False. Logical specifications apply only to logical values.

For example,

```
message(FORMAT("LY", (5= 5)))           ; displays Yes
message(format("LT(Good)",1= 1))        ; displays Good
```

```
message(format("LT(Good)",1= 2))          ; displays False
message(format("LF(Bad)",34<12))          ; displays Bad
```

Raster operations

The information in this section applies to graphic objects placed in a form, not to Graphic variables.

When you define a graphic object interactively, you identify a *source graphic* (the file you choose) to be placed in a *destination* (your computer's screen). Most often, Paradox assumes you want an unchanged copy of the source placed on the screen.

Suppose, however, you want the source graphic and the screen to interact. You might want to make the source graphic transparent, so the color of the page shows through it, or you might want to invert the color of the source graphic. You can achieve these kinds of effects using Paradox interactively, as described in the *User's Guide*, and you can use ObjectPAL to manipulate the RasterOperation properties of the graphic object.

Raster operations define how Paradox combines the source graphic with the destination—inverting, combining, including, or excluding colors to your specifications. Paradox uses the Boolean AND, OR, and XOR (exclusive OR) comparison operators to combine individual pixels of color during raster operations.

Table 20.3 describes what each raster operation property does.

Table 20.3 Date types

RasterOperation property	Onscreen result
SourceCopy	Copies an unchanged source graphic to the destination
SourcePaint	Combines the source graphic and the destination using the Boolean OR operator
SourceAnd	Combines the source graphic and the destination using the Boolean AND operator
SourceInvert	Combines the source graphic and the destination using the Boolean XOR operator
SourceErase	Inverts the destination and combines it with the source graphic using the Boolean AND operator
NotSourceCopy	Inverts the source graphic and copies it to the destination
NotSourceErase	Combines the source graphic and the destination and inverts the result using the Boolean OR operator
MergePaint	Inverts the source graphic and combines it with the destination using the Boolean OR operator

You can set a graphic object's RasterOperation property by using dot notation to address the object or by associating a UIObject variable with the graphic object. For example, the following statement sets the RasterOperation property of a graphic object named *fishGraphic* to SourcePaint:

```
fishGraphic.RasterOperation = SourcePaint
```

The following statements declare a UIObject variable named *theGraphic*, associate *theGraphic* with a graphic object named *monaLisa*, and set the graphic object's RasterOperation property to SourceInvert.

```
var
   theGraphic UIObject
endVar
theGraphic.attach(monaLisa)
theGraphic.RasterOperation = SourceInvert
```

Demonstration form

The form RASTEROP.FSL demonstrates how to use the RasterOperation property. If you installed the samples when you installed Paradox, you can run this form by changing your working directory to EXAMPLES, choosing File | Open | Form, and choosing RASTEROPS.FSL. If you did not install the sample files, you can do so now. Follow the instructions in the *User's Guide*.

21

Printing reports

Report type variables and methods let you control a Report window. A Report variable is a handle to a report. With it, you can change the Report window's size, position, and appearance, as well as preview and print a report.

Use **load** to load a report file in a design window; use **open** to open the report in the View Data choice, and use **print** to open a report and print it. You cannot attach methods to objects in a report.

Note ObjectPAL features many new methods for printer setup and report handling. For more information and examples, search for "report" in the online ObjectPAL Help.

The following example shows a simple technique for printing a report from disk. The example assumes the report has been designed and saved beforehand.

```
method pushButton(var eventInfo Event)
   var
      custRpt Report
   endVar
   custRpt.open("customer.rdl")
   custRpt.print()
endMethod
```

When this method executes, the **open** statement opens the report, and the **print** statement initiates the printing process. First, the Paradox Print File dialog box opens, giving you a chance to specify a range of pages to print. You must close this dialog box to print the report.

When you want more control over how a report is printed, you can use ReportPrintInfo, a pre-defined record that has the following structure:

```
name           String   ; Run this report if not already open
  masterTable     String   ; Master table name
  queryString     String   ; Run this query (actual query string)
  restartOptions  SmallInt ; What to do if data changes while printing
                           ; Use a ReportPrintRestart constant
  printBackwards  Logical  ; True: Backward, False: Forward, default: False
```

```
makeCopies      Logical  ; Who makes copies: Paradox or the printer?
                         ; If True, Paradox makes copies
panelOptions    SmallInt ; Use a ReportPrintPanel constant
nCopies         SmallInt ; Number of copies, default is 1
startPage       LongInt  ; Starting page, default is 1
endPage         LongInt  ; Ending page def: ending page
pageIncrement   SmallInt ; Use for multi-pass printing. Default is 1
xOffset         LongInt  ; Horizontal page offset
yOffset         LongInt  ; Vertical page offset
orient          SmallInt ; Use a ReportOrientation constant
```

To use ReportPrintInfo, declare a variable of type ReportPrintInfo and assign values to one or more of the fields in the record. You only have to assign values to fields you're interested in; Paradox supplies default values for the others. When you print a report using the ReportPrintInfo structure, the Print File dialog does not open. Paradox assumes you've specified everything you want to specify and sends the report directly to the printer.

The following example shows how to print using a ReportPrintInfo record.

```
method pushButton(var eventInfo Event)
   var
     stockRep  Report
     repInfo   ReportPrintInfo
   endVar
   ; first, set up the repInfo record
   repInfo.nCopies = 2
   repInfo.makeCopies = True
   repInfo.restartOptions = PrintLock
   ; then open the report and print it using repInfo
   stockrep.open("stock.rdl")
   stockRep.print(repInfo)
endMethod
```

This code assigns values to the fields of *repInfo* to print two copies of the report and to lock the underlying tables while the report is printing.

Querying tables

Querying is the process of retrieving information from database tables. This chapter explains how to use ObjectPAL to

- Execute a saved query
- Define and execute a query statement, including use of variables
- Define and execute a query string, using the values of other strings and variables

Some methods for working with queries are defined for the Database type, because in some applications you must specify the database that contains the tables you want to query.

Note Information on using ObjectPAL to create and execute SQL commands from methods is in the online ObjectPAL Help; search for "SQL Type" and "Database Type." Also see ObjectPAL Help for a list of new and changed Query type methods for this version of Paradox.

An ObjectPAL Query variable is a handle to a QBE query. You can use ObjectPAL to create and execute queries from methods just as if you were using Paradox interactively. You can execute a query from a query file, a query statement, or a quoted query string. By default, Paradox stores query results in ANSWER.DB in the user's private directory (represented by the alias :PRIV:), but you can specify a different table and a different directory.

QBE is a powerful mechanism, and it can save you a lot of programming. For example, suppose you have an employee database and you want to change a job title from "Supervisor" to "Manager" and change the salary from $40,000 to $45,500. You could write ObjectPAL code to do this, using scan loops or if...then...else blocks, but you'd be working harder than you need to. Using a simple CHANGETO query, you can accomplish the same task.

Important Before you create and run queries using ObjectPAL, you should be familiar with using queries interactively. See the *User's Guide* for more information.

Using a query file

From a programming perspective, the simplest way to run a query is to execute a query file. Assuming the query file has already been created and saved (for example, using queries interactively), use **readFromFile** to run the query and write the results to a table. The following example runs the query stored in the file named NEWORDER.QBE and writes the results to a table named NEWORDER.DB in the working directory. (If you don't specify a table, results are written to ANSWER.DB in the private directory.)

```
var
   tv TableView
endVar

; the results of the query will be written to neworder.db
if executeQBEFile("neworder.qbe", "neworder.db") then
   tv.open("neworder.db")
else
   msgStop("Stop", "Couldn't run the query.")
endIf
```

You can also use an alias to specify a path to the *Answer* table (or another table that holds the results). For example, the following statement writes the results to NEWORDER.DB in the directory represented by the alias *mast* (assuming *mast* has already been defined elsewhere).

```
executeQBEFile("neworder.qbe", ":mast:neworder.db")
```

Using a query statement

A query statement begins with the keyword **Query** and ends with the keyword **endQuery**. In between, you specify the table or tables to query, along with the fields and selection criteria. To run a query statement, use **executeQBE**. This method stores the results in ANSWER.DB in the Private directory by default, but you can specify another table.

Note You don't have to list all the fields in a table. Instead, you can list only those fields that affect the query, as in this example:

```
myQBE = Query

orders.db | Name  | Qty  |
          | Check | >10  |

endQuery

executeQBE(myQBE) ; stores results in :PRIV:ANSWER.DB by default
```

The previous query retrieves from the *Orders* table the name of any customer who ordered more than 10 items. (The *Orders* table has more than two fields, but only the two fields you're interested in are specified in the example.)

Important The blank line after the **Query** keyword and the blank line before the **endQuery** keyword are required. Also, if you're querying more than one table, you must put a

blank line between the query statement for each table. You don't have to make the columns in a query line up, but if you do, the code is easier to read.

A QBE query statement can contain variables, example elements, and operators, as discussed in the following sections.

Query variables

You can use variables in query statements by preceding them with a tilde (~) character. When Paradox finds a tilde variable in a query statement, it replaces the variable with its current value. In the following example, the variable name is *myItem*. The example assumes the form contains a field object named *userItem*. It retrieves the names of people who ordered the item specified in *myItem* and stores the results in ITEMS.DB in the working directory. If the query fails for any reason, the call to the System procedure **errorShow** displays error information in a modal dialog box.

```
var
   qstmt2 Query
endVar

myItem = userItem.value ; get value from userItem field in form
qstmt2 = Query

orders.db  | Name    | Item             |
           | Check   | Check ~myItem    |      ; tilde variable is myItem

endQuery

if not executeQBE(qstmt2, "items.db") then  ; stores results in :WORK:ITEMS.DB
   errorShow()
endIf
```

The last block of code is an if...then block that tests whether the query executes successfully. If there is an error, the call to the System procedure **errorShow** displays a dialog box showing the error code and the error message. For more information about errors and error handling, refer to Chapter 30.

You can also use expressions in queries. Simply precede the expression with a tilde, as shown in the following example. This code retrieves the names of all employees whose salary is greater than $36,000 (the base value of $35,000 plus another $1,000 added in the tilde expression).

```
method pushButton(var eventInfo Event)
   var
      qStmt Query
      baseValue LongInt
   endVar

   baseValue = 35000

   qStmt = Query

        employee.db | Name  |  Salary                      |
                    | Check | Check >~(baseValue + 1000) |
```

```
            EndQuery

   if not executeQBE(qStmt) then ; stores results in :PRIV:answer.db
      errorShow()
   endIf
endMethod
```

The following example uses a tilde variable to represent the name of the table to query. It also uses the **view** method defined for the String type to get input from the user.

```
var
   qStmt Query
   tblName String
endVar

tblName = "Enter table name here."
tblName.view("Which table?") ; User types table name into dialog box,
                             ; and the variable tblName stores it.

qStmt = Query

        ~tblName | Name  | Phone |
                 | Check | Check |

        endQuery

if not executeQBE(qStmt) then ; stores results in :PRIV:answer.db
   errorShow()
endIf
```

Using aliases

You can use an alias to specify a path to a table to query. For example, the following query string is valid, because it uses the alias *mast* to represent the path to CUSTOMER.DB:

```
var
   qStmt Query
endVar

qStmt = Query

          :mast:customer.db | Cust# | Name  |
                            | _cust | Check |

          endQuery

if not executeQBE(qStmt) then
                       ; puts results in :PRIV:ANSWER.DB by default
   errorShow()
endIf
```

Example elements

To represent an example element, precede the text with an underscore (_). For example, type `_cust` to represent the example element *cust*.

The following example uses example elements and the keyword Check (discussed in the next section) to retrieve the names of customers from the *Customer* table and the items they've ordered from the *Orders* table. The blank lines after the Query keyword, between the query statements for each table, and before the endQuery keyword are required. Because a query statement is not a quoted string, only single backslashes are allowed in directory paths.

```
var q Query endVar
q = Query

c:\tables\customer.db   | Cust# | Name  |
                        | _cust | Check |

c:\tables\orders.db      | Cust# | Item  |
                         | _cust | Check |

endQuery

executeQBE(q) ; stores results in :PRIV:ANSWER.DB
```

Operators

Table 22.1 lists all Paradox Query operators—including reserved symbols and words—and arithmetic, comparison, wildcard, special, summary, and set-comparison operators. For more information about queries and query operators, refer to the *User's Guide*.

The following query uses the Check and CheckPlus keywords:

```
var qstmt1 Query endVar
qstmt1 = Query

C:\tables\customer.db | Cust#               | Name      |
                      | Check >1200, <1500  | Check     |
                      | Check >2000         | CheckPlus |

EndQuery

executeQBE(qstmt1, "cust.db") ; puts results in Cust.db
```

You can use a wildcard operator (.. or @) with a query variable. You must make the wildcard operator part of the query statement, not part of the variable assignment. For example, to find all order numbers ending in 301, assign a value of "301" to a variable named *Ordnum*. Then specify **..~Ordnum** in the Order# field of the query statement.

```
var
   q Query
   Ordnum String
endVar
```

```
Ordnum = "50"

q = Query

c:\tables\orders.db | Order#          |
                    | Check ..~Ordnum |

endQuery

executeQBE(q, "ord301.db") ; store the results in :WORK:ORD301.DB
```

Table 22.1 Query operators

Category	Operator	Meaning
Reserved words	check	Display unique field values in *Answer*
	checkPlus	Display field values including duplicates in *Answer*
	checkDescending	Display field values in descending order
	groupBy	Specify a group for set operations
	INSERT	Insert records with specified values
	DELETE	Remove records with specified values
	CHANGETO	Change specified values in fields
	SET	Define specific records as a set for comparisons
Arithmetic operators	+	Addition or alphanumeric string concatenation
	–	Subtraction
	*	Multiplication
	/	Division
	()	Group operators in a query expression
Comparison operators	=	Equal to (optional)
	>	Greater than
	<	Less than
	>=	Greater than or equal to
	<=	Less than or equal to
Wildcard operators	..	Any series of characters
	@	Any single character
Special operators	LIKE	Similar to
	NOT	Does not match
	BLANK	No value
	TODAY	Today's date
	OR	Specify OR conditions in a field
	,	Specify AND conditions in a field
	AS	Specify the name of a field in *Answer*
	!	Display all values in a field regardless of matches
Summary operators	AVERAGE	Average of values in a field
	COUNT	Number of values in a field

Table 22.1 Query operators (continued)

Category	Operator	Meaning
	MIN	Lowest value in a field
	MAX	Highest value in a field
	SUM	Total of all values in a field
	ALL	Calculate summary based on all values in a group, including duplicates
	UNIQUE	Calculate summary based on unique values in a group
Set comparison operators	ONLY	Display records that match only members of the defined set
	NO	Display records that match no members of the defined set
	EVERY	Display records that match every member of the defined set
	EXACTLY	Display records that match all members of the defined set and no others

Using a query string

A query string is like a query statement, with the following differences:

- The "Query...endQuery" block is enclosed in double quotes.
- A query string can be built from smaller strings. This is useful for defining a query through context or user interaction.
- A query string cannot contain tilde variables, but you can use String variables to get the same effect.

To run a query string, use **readFromString**. This method stores the results in ANSWER.DB by default, but you can specify another table. Unlike a query statement, a query string is a quoted string, so you must precede a special character with a backslash. (However, query strings are not stored in a Windows resource file, because they can be longer than 255 characters.)

The following query string is built by adding quoted strings and the String variable *qs*. Newline characters ("\n") specify the required blank lines. The query retrieves the customer number and the name of every customer whose customer number is greater than 100.

```
var
   qs, qstr String
endVar
qs = "C:\\tables\\customer.db | Cust#      | Name       |"

qstr ="Query" +
       "\n\n" +
        qs + "\n" +
       "| Check >100 | CheckPlus  |" +
       "\n\n"  +
       "endQuery"

executeQBEString(qstr) ; puts results into :PRIV:ANSWER.DB by default
```

You can't use tilde variables in a QBE string, but you can get the same effect by declaring a String variable and using it as part of the query. The following example gets the value of the partName field of the first record in the *Parts* table, stores it in the String variable *qv*, and builds a query string *qs* using quoted strings and the String variable *qv*. The query retrieves the name of everyone who ordered the part specified in *qv*.

```
var
   qs, qv String
   tc TCursor
endVar
tc.open("parts.db")
qv = tc.partName ; get value from partName field of Parts table

qs = "Query" +
"\n" + "\n" +
"orders.db    | Name   | Item              |\n" +
"             | Check  | Check " + qv + " |\n" +
"\n" + "\n" +
"endQuery"

executeQBEString(qs) ; puts results into :PRIV:ANSWER.DB by default
```

Using aliases to specify a path for a table to query

You can use an alias to specify a path to a table to query. For example, the following query string is valid, because it uses the alias *mast* to represent the path to CUSTOMER.DB:

```
var
   qString String
endVar

qString = "Query

          :mast:customer.db | Cust# | Name  |
                            | _cust | Check |
          endQuery"

if not executeQBEString(qString) then
                        ; puts results in :PRIV:ANSWER.DB by default
   errorShow()
endIf
```

You can also use **getAliasPath**, defined for the System type, to return the path an alias represents. **getAliasPath** returns a string that you can assign to a String variable and use in a query string. For example, the following code gets the path for the alias *mast* and assigns it to the variable *pathString*; then it uses *pathString* as part of the query string. The double backslashes before the table name are required.

```
var
   qString, pathString String
endVar

pathString = getAliasPath("mast")

   qString = "Query" +
             "\n\n" +
              pathString + "\\customer.db | Phone |
                                          | Check |\n" +
             "\n\n" +
             "endQuery"

if not executeQBEString(qString, "custphon.db") then
                        ; writes results to :WORK:CUSTPHON.DB
   errorShow()
endIf
```

Managing sessions

A Session variable represents a channel to the database engine. Opening a Paradox application opens one session by default (called the default session), and you can use ObjectPAL to open other sessions from within an application. The number of sessions you can open varies in different environments, but only the default session can be managed using Paradox interactively. You must manage other sessions with programming. A Session variable provides a handle—a variable you can manipulate in ObjectPAL code to manage the actual Paradox session.

You may want to open another session to

- Implement different security schemes
- Set up different alias structures
- Specify different handling of blank values
- Specify different retry periods
- Implement separate locking schemes

Table 23.1 lists some Session tasks and the methods to perform them.

Table 23.1 Session tasks and methods

Task	Methods
Add to a Session	addAlias, addPassword
Use alternate project aliases	addProjectAlias, loadProjectAliases, removeProjectAlias, saveProjectAliases
Session settings	blankAsZero, ignoreCaseInLocate, setRetryPeriod
Get information about a Session	getAliasPath, getNetUserName, retryPeriod
Enumerate information to tables	enumAliasNames, enumDatabaseTables, enumUsers

Important Opening another session is not the same as launching another instance of Paradox. Multiple sessions run under the same Desktop.

The important attributes of a session are:

- *Project aliases:* Each session maintains its own project aliases not seen by other sessions.
- *Session-specific settings*: Each session can specify settings for handling blank values, retry periods, and wildcards and case sensitivity in text searches.
- *Passwords*: Each session maintains its own list of passwords.
- *Locking*: Locks are not shared between sessions. Locks set by ObjectPAL interact as peers with locks set interactively in the same session. Because ObjectPAL can use Session variables to create sessions, you can write methods to lock tables that you couldn't lock using Paradox interactively.
- *User count*: Paradox licensing arrangements are tracked on the basis of user count, which is maintained on the network by the database engine. Each session exhausts one user count. This mechanism might be a convenient method of tracking licenses for your Paradox-based application.

Working with sessions

The first step in working with a session is to open it. As previously stated, Paradox opens one session by default. To create a handle to the default session, declare a Session variable and call the Session type **open** method with no arguments.

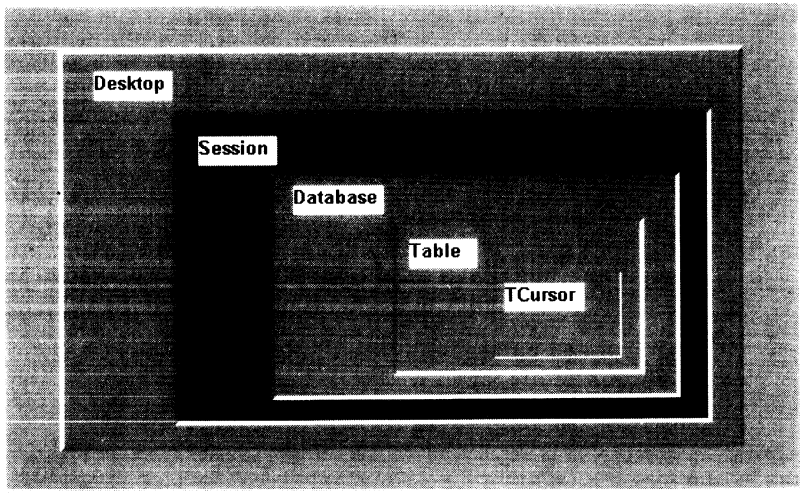
To open another session, declare a Session variable and call **open** with one argument, a string. The text of the string doesn't matter, but you have to supply it; otherwise, you'd get another handle to the default session.

When the following code executes, it creates a handle to the default session (*defaultSes*) and opens a second session (*secondSes*).

```
method pushButton(var eventInfo Event)
   var
      defaultSes, secondSes Session
   endVar

   defaultSes.open() ; creates a handle to the default session
   secondSes.open("Second session") ; opens a second session
endMethod
```

Opening a session creates a handle that you can use to specify session settings. The Desktop maintains settings for each session separately. Once you open a session, you can open a database in that session, and the session settings will affect the database. Similarly, you can attach a Table variable or open a TCursor onto a table in that database, and the session settings will affect the table. Figure 23.1 diagrams this scoping hierarchy.

Figure 23.1 Session scoping hierarchy



You can use ObjectPAL to open multiple sessions in the same Desktop. The Desktop tracks the settings for each setting. You can open a database in a session, and attach a Table variable or open a TCursor onto a table in the database. The session settings affect operations at each level of the hierarchy.

Using session settings

After you open a session, you can use Session type methods and procedures to specify settings for that session. Then you can open a database in that session, and the settings determine how Database type methods such as **delete** and **executeQBE** execute.

The following example shows how session settings affect database operations. Suppose the *Customer* table is protected with "foo" as the password. The following code opens two sessions, *firstSes* and *secondSes*, then adds the password "foo" to the second session. Next, it opens a database in each session. The attempt to delete the table in the first session fails because the password was not added to that session. The attempt to delete the same table succeeds in the second session because the password was added.

```
method pushButton(var eventInfo Event)
   var
      firstSes, secondSes Session
      firstDb, secondDb Database
   endVar

   firstSes.open("First session")   ; open first session
   secondSes.open("Second session") ; open second session

   secondSes.addPassword("foo")     ; add password to second session

   firstDb.open(firstSes)   ; open database in first session
   secondDb.open(secondSes) ; open database in second session

   message(firstDb.delete("customer.db")) ; displays False
```

```
    sleep(2000)
    message(secondDb.delete("customer.db")) ; displays True
endMethod
```

As the previous example shows, you can open a session, then open a database in that session. You can also attach to a table in that database, and settings for the session will affect operations on that table. In the following example, session settings affect databases, which in turn affect the results of an operation using a table variable.

The code opens two sessions and two databases (as in the previous example). Then, calls to **blankAsZero** specify how to handle blank values in each session. As a result, the subsequent two calls to **cCount** return different values, even though they operate on the same table. This example assumes that some records in the *Customer* table have blank values in the State/Prov field. Blank values are counted in the first session but not in the second session.

```
method pushButton(var eventInfo Event)
   var
      firstSes, secondSes Session
      firstDb, secondDb Database
      firstTb, secondTb Table
      firstCount, secondCount Number
   endVar

   firstSes.open("First session")   ; open first session
   firstSes.blankAsZero(True)       ; specify how to handle blank values
   firstDb.open(firstSes)           ; open database in first session
   firstTb.attach("customer.db", firstDb) ; attach in 1st database
   firstCount = firstTb.cCount("State/Prov") ; count values

   secondSes.open("Second session")
   secondSes.blankAsZero(False)
   secondDb.open(secondSes)
   secondTb.attach("customer.db", secondDb)
   secondCount = secondTb.cCount("State/Prov")

   msgInfo("Is firstCount greater?", firstCount > secondCount)
   ; displays True, because firstSes counts blank values
endMethod
```

Working with the file system

This chapter shows how to use methods in the FileSystem type to work with disk files, drives, and directories. Also discussed is **fileBrowser**, the System type procedure that enables the user to choose one or more files using Paradox's built-in Browser.

A FileSystem variable provides a handle—a variable you can use in ObjectPAL statements to work with a directory or a specified group of files in a directory. Table 24.1 lists some tasks and the methods to perform them.

Table 24.1 Common FileSystem tasks and methods

Task	Methods
Working with files	accessRights, copy, delete, findFirst, findNext, fullName, name, rename, size
Working with drives	drives, existDrive, freeDiskSpace, getDrive, isFixed, isRemote, isRemovable, setDrive, totalDiskSpace
Working with directories	deleteDir, getDir, makeDir, setDir, windowsDir, windowsSystemDir

Note Be careful when using ObjectPAL and FileSystem methods to work with tables, because tables may have related files. For example, suppose the *Customer* table is keyed on the Customer No field and includes a memo field. In this case, you would have three related files: the table file, named CUSTOMER.DB; the index files, named CUSTOMER.PX; and a file containing the memo data, named CUSTOMER.MB. For certain operations (for example, copying a table), it is important to get all related files. Refer to the *User's Guide* for more information.

Working with a FileSystem variable

In many cases, the first step in working with a FileSystem variable is to use **findFirst** to see if the directory contains any files. It may be helpful to think of this step as initializing the FileSystem variable.

This method initializes the FileSystem variable *c*. Then it uses *c* as a handle to examine the C:\WINDOWS directory to see if it contains any files. If files exist, this method then

reports about your access rights to this directory; otherwise, it displays a message saying that no files were found, as in the following example:

```
var c FileSystem endVar
if c.findFirst("C:\\windows\\*.*") then ; notice the double backslash
   msgInfo("Access rights", c.accessRights())
else
   msgInfo("Windows directory", "No files found.")
endIf
```

When you work with a FileSystem object, enclose directory paths in quotes and use two backslash characters. Case doesn't matter. For example, suppose the DOS path you're working with is

```
C:\WINDOWS\SYSTEM
```

To specify that path in a method, you could use

```
"c:\\windows\\system"
```

You can use the wildcard characters * and ? with FileSystem objects just as you use them in DOS and Windows. For example, the following code finds files named ORDERS.DB and SALES.DBF.

```
c.findFirst("C:\\tables\\*.db?")
```

Important You can't use ObjectPAL to change the Working (:WORK:) or Private (:PRIV:) directories of an application that's running, because when you change :WORK: or :PRIV:, Paradox closes all open windows, including the one executing the statement to change directories!

Instead, you can use the System procedure **execute** to launch another instance of Paradox, and command-line switches to specify :WORK:, :PRIV:, and a startup document, as shown in the following example.

The following code launches another instance of Paradox, sets :WORK: to C:\PDOXWIN\FORMS, sets :PRIV: to C:\PDOXWIN\PRIVATE, and runs the form ORDERS.FSL.

```
method pushButton(var eventInfo Event)
   execute("pdoxwin.exe -w c:\\pdoxwin\\forms -p c:\\pdoxwin\\private orders.fsl")
endMethod
```

For more information about command-line configuration, refer to online Help.

In addition to the FileSystem methods listed in Table 24.1, ObjectPAL provides two other methods: the FileSystem type method **enumFileList**, and the System type method **fileBrowser**.

Listing file information to a table

Use **enumFileList** to list file information to a Paradox table or to an array. Then you can use ObjectPAL's table-manipulation methods to work with the data. For example, the following code searches the C:\WINDOWS directory for files having an extension .EXE, and writes information about those files into a Paradox table named WINAPPS.DB.

```
var c filesystem endVar
if c.findFirst("C:\\windows\\*.exe") then
   c.enumFileList("C:\\windows\\*.exe", "winapps.db")
endIf
```

Now you can search the table, run a query, edit the data, and so on.

The following example searches the specified directory for file names that represent Paradox forms, stores the names in an array, then displays the names in a pop-up menu.

```
method pushButton(var eventInfo Event)

   var
      fs FileSystem
      formDir String
      formNames Array[] String
      p PopUpMenu
   endVar

   formDir = "C:\\pdoxwin\\forms\\*.f?l"

   if fs.findFirst(formDir) then
      fs.enumFileList(formDir, formNames)
      p.addArray(formNames)
      theForm = p.show() ; displays a pop-up menu of form names
   endIf
endMethod
```

Using the fileBrowser procedure

You can use the System type procedure **fileBrowser** to enable the user to choose one or more files using Paradox's built-in Browser. You can store the user's choice or choices in a String or an Array, as shown in the following example.

```
var
   oneFile String
   manyFiles Array[] String
   tView TableView
endVar
fileBrowser(oneFile) ; Displays the Browser, and waits
                     ; for the user to choose one file.
                     ; Variable oneFile stores the file name chosen.
if isTable(oneFile) then
   tView.open(oneFile)
endIf

; lets the user select multiple files and stores the file names in an array
fileBrowser(manyFiles)

manyFiles.view() ; displays the user's choices
```

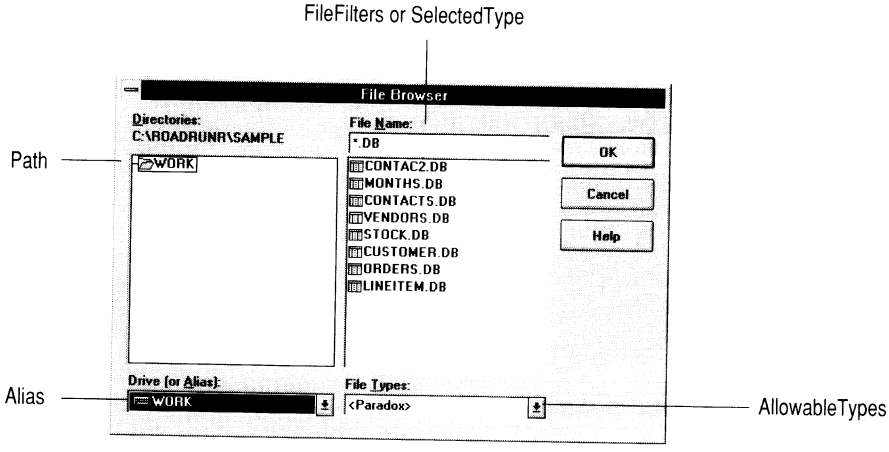
You can also pass a record as an argument to **fileBrowser** to specify what data the Browser displays. For example, you can make the Browser display Paradox tables only, or forms only, or forms and reports.

ObjectPAL provides a special data type, called FileBrowserInfo, that you can use *only* with the **fileBrowser** procedure. FileBrowserInfo is a Record with the following structure:

```
TYPE FileBrowserInfo =
  Record
    x, y, w, h    SmallInt      ; Size of Browser window in twips
    WindowStyle   LongInt       ; Window style. See WindowStyle constants.
    AllowableTypes LongInt      ; See Table 24.2
    SelectedType  LongInt       ; One of the AllowableTypes
    FileFilters   String        ; The filespec in edit box
    Alias         String        ; Alias or drive name
    Path          String        ; Path relative to ALIAS
  endRecord
ENDTYPE
```

This record structure is predefined and built into ObjectPAL, so all you have to do is declare a variable of type FileBrowserInfo and assign values to its fields, as shown in the example at the end of this section—you don't have to declare the type yourself each time you want to use it. Figure 24.1 shows the Browser and the areas affected by the various fields of the FileBrowserInfo record.

Figure 24.1 Browser areas affected by FileBrowserInfo



After the call to **fileBrowser**, the Alias, Path, and FileFilter fields are filled in with the values that were in the Browser dialog box. In other words, you can find out what the user entered in those areas of the Browser.

The AllowableTypes field specifies what appears in the drop-down edit list for the Types panel in the Browser. The SelectedType field indicates which of the AllowableTypes is

currently selected. Table 24.2 lists valid ObjectPAL constants to use in the SelectedType and AllowableTypes fields.

Table 24.2 Constants for the AllowableTypes and SelectedType fields

Constant	Extension	Description
fbAllTables	*.db, *.dbf	All table types supported by Paradox
fbASCII	*.txt	Text files
fbBitmap	*.bmp	Windows Bitmap graphics
fbConfig	*.cfg	Configuration files
fbDBase	*.dbf	dBASE tables
fbDM (5.0)	*.dm	Data model files
fbExcel	*.xls	Excel worksheets
fbFiles	*.*	All files
fbForm	*.fsl, *.fdl	Paradox forms
fbGraphic	*.bmp, *.pcx, *.tif, *.gif, *.eps	Assorted graphics file formats
fbIni	*.ini	Initialization files
fbLibrary	*.lsl, *.ldl	ObjectPAL libraries
fbLotus1	*.wks	Lotus 1-2-3 version 1 worksheets
fbLotus2	*.wk1	Lotus 1-2-3 version 2 worksheets
fbParadox	*.db	Paradox tables
fbPrintStyle (5.0)	*.fp	Printer style sheets
fbQuattro	*.wkq	Quattro worksheets
fbQuattroPro	*.wq1	Quattro Pro for DOS worksheets
fbQuattroProWindows	*.wt1	Quattro Pro for Windows notebooks
fbQuery	*.qbe	QBE files
fbReport	*.rsl, *.rdl	Paradox reports
fbScreenStyle (5.0)	*.ft	Form style sheets
fbScript	*.ssl, *.sdl	ObjectPAL scripts
fbSQL (5.0)	*.sql	SQL files
fbTable	*.db	Paradox tables
fbTableView	*.tv	Paradox table view files
fbText	*.txt	All text files

The **fileBrowser** procedure looks only at the names given in the structure. You can pass a different record structure to it and it will find the fields with the appropriate names and use them. In other words, you can define a simpler record structure with only the items you are interested in.

FileBrowserInfo

Attach the following code to a button's built-in **pushButton** method. When the method executes, it invokes the Browser and waits for you to choose a file. Then, it displays information about your choice in the status area.

```
method pushButton(var eventInfo Event)

var
```

```
    fbi FileBrowserInfo ; Declare a variable that uses the predefined
                        ; FileBrowserInfo record structure
    selectedFile String
endVar

; The following statements assign values to fields in the
; record of file browser information
fbi.Alias = "WORK" ; Search the current working directory
fbi.AllowableTypes = fbTable + fbForm ; Search for tables and forms

; Display the Browser and process the user's selection
if fileBrowser(selectedFile, fbi) then
   message("You selected ", selectedFile," with the path ", fbi.path)
else
   message("You selected cancel")
endIf

endMethod
```

TextStream: Working with text files

This chapter shows how to work with a TextStream—a sequence of characters read from (or written to) a text file. TextStreams contain only ANSI characters; formatting information such as font, alignment, and margins is not included. (To work with formatted text, use Memo objects; see Chapter 5.) However, nonprinting characters, such as carriage returns and line feeds (CR/LF) are included.

Paradox maintains a file position pointer that behaves like an insertion point in a word processor. The pointer tells you how far (how many characters) you are from the beginning of the file.

Working with a TextStream

You can create text files using **create**, using Paradox interactively, or using a word processor. (The word processor must have a "Save as text" function, or something similar.) Table 25.1 lists tasks and the TextStream methods to perform them.

Table 25.1 TextStream tasks and methods

Task	Methods
Open and close file	open, close
Specify where to read from and write to	position, setPosition
Read and write	readLine, readChar, writeLine, writeString

Note The first position in a TextStream is 1 (not 0, as in some other languages).

You can read from and write to a TextStream one or more bytes at a time, one line at a time, or one entire object at a time. **readLine** and **writeLine** work with lines of text delimited by and including a carriage return-linefeed combination (CR/LF). **readChar** and **writeString** work with text up to—but not including—the CR/LF characters.

TextStreams and Strings

TextStreams and Strings are distinct, but related, objects. TextStream methods handle the flow of text (input/output) between your application and a disk file. String methods manipulate text, as shown in the following example:

```
method pushButton(var eventInfo Event)
   var
      txt1 TextStream
      string1, string2, string3 String
   endVar

   string1 = "These are times "
   ; 18 characters, including cr/lf and 1 space after times
   string2 = "that try men's souls."
   ; 23 characters, including cr/lf at end of line

   txt1.create("henry.txt")  ; creates file henry.txt in working dir
   txt1.writeLine(string1)   ; leaves file open for writing
   txt1.writeLine(string2)
   txt1.close()

   if not txt1.open("henry.txt", "W") then ; opens file for Read and Write
      errorShow()
   endIf
   txt1.setPosition(31)
   ; next read or write will begin at the 31st
   ; character in the file, the "m" in men's

   string3 = "women's souls."
   txt1.writeString(string3)
   txt1.close()

endMethod
```

Developing multi-form applications

Multi-form applications are applications that use more than one form, either sequentially or simultaneously. Multi-form applications are constructed from standard forms and from dialog boxes, which are a special type of form. Dialog boxes are typically used to exchange information with the user in a specific sequence, for example, when collecting a password before allowing other activity to proceed. In contrast, multiple standard forms are typically used to subdivide an application into functional modules.

This chapter discusses the key concepts for building multi-form applications. It covers these topics:

- Differences between forms and dialog boxes
- Designing a dialog box
- Handling multi-window interaction
- Using pre-built dialog boxes
- Advanced topics, including **openAsDialog** and the DeskTopForm property.

Differences between forms and dialog boxes

There are a number of characteristic differences between Paradox forms and dialog boxes.

- **Modality** Windows can be either modal or nonmodal. A strictly *modal* window is one that has exclusive rights to focus and events. In contrast, a *nonmodal* window allows the user to move freely from that window to other windows, to access system menus and the Toolbar, and to invoke other Windows applications. Paradox also provides an intermediate state in which only the waiting form is denied focus.

  Paradox provides three types of windows:

- *Standard forms* are always nonmodal. However, a form behaves as though it is suspended when it waits upon another form or dialog box.
- *Nonmodal dialog boxes* behave in much the same way as forms. The form that calls the nonmodal dialog box is suspended as long as it waits, but the user has access to other forms, menus, etc.
- *Modal dialog boxes* behave in a strictly modal fashion as long as they are waited upon, or *if they have been invoked interactively*. Otherwise, they behave like a nonmodal dialog box.

The difference between modal and nonmodal dialog boxes is subtle. When not waited upon, they behave the same. When waited upon, a modal dialog box retains strict modal focus. Typically, a modal dialog box is used when a program sequence needs additional information before it can proceed. The user cannot continue some other operation unless the command is canceled or some additional information is provided.

On the other hand, a nonmodal dialog box, while it is being waited upon, lets the user access the system menus and access other windows. A good example of this style is a text-search command: the dialog box remains displayed while the search is carried out. The user can then return to the dialog box and search for the same or different value again and again.

- **Position** The position coordinates for a dialog box are calculated with respect to the screen, while the position coordinates for a form are calculated with respect to the Paradox Desktop. In both cases, the origin of the grid has coordinates of (0,0).

  Note for experienced Windows developers: The position coordinates of dialog boxes are calculated relative to the screen because dialog boxes are *not* MDI children. See "MDI child windows" later in this chapter.

- **Always on top** Dialog boxes are always displayed on top of other windows. One dialog box can cover another, whereas a regular form can never cover a dialog form. Forms among forms can be moved to the top (see Form type method **bringToTop**).

- **Run-time resizing and cosmetics** Forms can be resized at run time, whereas dialog boxes cannot be resized interactively.

  Both forms and dialog boxes can include vertical and horizontal scroll bars. Forms must contain a title bar, which enables the user to move the window freely about the screen; with dialog boxes this is an option.

- **Movement outside Desktop window** Dialog boxes, both modal and nonmodal, can be repositioned outside of the Desktop window. Regular forms, which are MDI children, must remain within the boundaries of the parent Desktop.

- **Handling menus** Dialog boxes do not interact with the menu; they do not alter the menu bar and do not accept menu selections. Therefore, if the user will need to interact with menus, don't use a dialog box.

  When a modal dialog box is active, the user cannot interact with any other part of Paradox (other forms or Table windows) while the dialog box is waited upon.

- **MDI child** For experienced Windows programmers, the difference between Paradox forms and dialog boxes is summarized as follows: a form is a Windows MDI child; a dialog box is a popup window. The implications of this are explained in the advanced topics section later in this chapter.

Designing a dialog box

The first step in designing a dialog box is to design the form you'll use. Choose File | New | Form to create a form. The next step is to set the form Window Style to *dialog box*. To do this, choose Properties | Form | WindowStyle to display the Form Window Properties window. From here you can set window style, window properties, frame properties, and title bar properties.

In the Window Style panel, choose Dialog Box to give the form the attributes of a dialog box. This makes the Title Bar options (mandatory with standard forms) and the Modal property available.

After setting the properties, save the form (File | Save As). Property settings will be preserved. The next time you open that form, it will behave as a dialog. Refer to the *User's Guide* for information about setting form properties interactively.

Note Use care when designing a dialog box; leave a way to close the dialog box. When a modal dialog box is active, the user must have a way to escape from it.

Setting properties with ObjectPAL

Form and dialog box properties can also be set with ObjectPAL when the window is opened, using the **open** method. It is highly recommended that the dialog box properties be set using the Window Style panel in the Form Design window. It's easier to set the properties interactively when you are designing the form, and then use **open**, rather than figuring out how to manage a correct and consistent set of Window Style constants.

Specifying position and size

Standard forms can be marked with the "size to fit" property, which makes the window fit the surrounding page object. So a form can be sized interactively by setting the size of the page beforehand. Otherwise, the default MDI window size will be used, in which case you will probably need to resize interactively.

Alternatively, you can override Paradox default size by calling **setPosition**, or you can specify a size in the **open** statement. For example, the **open** statement in the following code sets the position of the upper left corner of the form to (100, 100), sets the width to 5,000 twips, and sets the height to 2,000 twips.

```
var
   ordForm Form
endVar
ordForm.open("orders.fsl", WinStyleDefault, 100, 100, 5000, 2000)
```

A situation might arise in which you don't want to specify all four position coordinates. For example, you might want to display the form one inch below its default position below the Toolbar. In such a case, use the constant WinDefaultCoordinate, as follows:

```
var
   ordForm Form
endVar
ordForm.open("orders.fsl", WinStyleDefault, WinDefaultCoordinate, 1440,
WinDefaultCoordinate, WinDefaultCoordinate)
```

The WinDefaultCoordinate constant acts as a place holder, and is replaced by the form's default value (size or position) when you run the form.

Modal dialog boxes

Remember, when you set the Dialog Box property to True, you must explicitly set the Modal property to get a Modal dialog box.

Handling multi-window interaction

There are many different types of multi-window interactions. For example, you can

- Open a modal dialog box, interact with the user, return the user's response (to the calling form), and then close the dialog box
- Use a nonmodal dialog box to let the user enter repetitious information or actions
- Use two (or more) forms in an application

To control variables or objects on another form, use dot notation to access the objects, values, and properties you need. This forms the basis of multi-window programming. Several built-in methods make this coordination easy to handle.

Using wait, formReturn and close

Typically, when you display a dialog box, you want the user to respond. For example, you want the user to enter some information, or simply acknowledge a message. While you're waiting, you want your code to wait, too, because subsequent statements may need one or more values returned by the dialog box. To accomplish this, use a **wait** statement to suspend execution until the called form returns control to the calling form.

The called form or dialog box does this by issuing a call to either **formReturn** or **close**. There are three ways of returning control from a dialog box to the calling form:

- Use the dialog box's menus to close it interactively
- Call **close**
- Call **formReturn**

Closing a dialog box interactively

If a dialog box has a Control menu, you can close it interactively by choosing Close from the Control menu, or by pressing *Ctrl+F4*. Either action returns control to the calling form and returns a Logical value of False.

When the user has finished with a dialog box, get the values you need, and then use **close** to close the dialog box. When you close a dialog box, you can't refer to its objects, their values, or their properties, unless you reopen it.

Calling close

Use the **close** method to close the dialog box (or standard form) and return control to the calling form. You can include an optional argument in the **close** statement to return a value to the calling form. If you call **close** without an argument, it returns a Logical value of False.

Calling formReturn

The **formReturn** method returns control to a calling form which is suspended by a **wait** statement. In contrast to **close**, the dialog box or form which calls **formReturn** remains open. The **formReturn** statement can return a value to the calling form; without an argument, it returns a Logical value of False.

The calling form may set up a **wait** loop to enable the called dialog box to invoke **formReturn** repeatedly. For example, the calling form might check the value returned and call **wait** for that form again until a legitimate value is returned.

Modal dialog boxes

When the called dialog box is marked modal, the user can't interact with anything else as long as the caller (or some other form) is waiting on it. Once this dialog box has invoked **formReturn**, focus can be returned to other windows.

Nesting wait statements

You can nest **wait** statements, but be sure to return control in the opposite order that you called the **wait** statements. For example, a form *formA* can open *formB* and call **wait**; *formB* can then open *formC* and call **wait**—*formA* and *formB* will not respond to events until *formC* returns control. When you return control, keep in mind that it will return first from *formC* to *formB*, and then from *formB* to *formA*.

Example

This example illustrates the use of **close**, **formReturn** and **wait**, as well as direct access of objects on other forms. See also the Connections sample application, which uses several forms and dialog boxes.

This example shows code attached to two forms. The first form is the main form and contains a button that opens a second form. This suspends activity in the first form until the second form returns control. If the user returns control by clicking the OK button, code executes to read the value of *AnswerField*, a field object in the dialog box that presumably contains a value entered by the user. This value is passed as an argument to

the custom procedure **doSomething** (defined elsewhere). If the user returns control by clicking the Cancel button, the dialog box closes itself.

This code is attached to a button in the main form:

```
method pushButton(var eventInfo Event)   ; Main (calling) Form
var
   dlgForm Form
   dlgReturn AnyType
endVar

if dlgForm.openAsDialog("myDlg.fsl", WinStyleDialogFrame, 100, 100, 2880, 2880) then
   dlgReturn = dlgForm.wait()  ; suspend until the dialog box returns control
   switch
      case dlgReturn = "OK" :
         doSomething(dlgForm.AnswerField.Value) ; dialog box is still open
      case dlgReturn = "Cancel" :
         return                                 ; dialog box has closed itself
   endSwitch
   dlgForm.close()                              ; close dialog box
endIf
endMethod
```

The second form, opened as a dialog box, contains two buttons, OK and Cancel. The following code is attached to the OK button. It returns control to the main form, returning a value of "OK" to the **wait** statement in the main form.

```
method pushButton(var eventInfo Event)   ; OK button code
   formReturn("OK")       ; return value, don't close this dialog box
endMethod
```

The following code is attached to the Cancel button of the second form. It uses a **close** statement to close the dialog box and returns "Cancel" to the **wait** statement in the main form.

```
method pushButton(var eventInfo Event)   ; Cancel button code
   close("Cancel")        ; return a value and close this dialog box
endMethod
```

Modal example

When an ObjectPAL statement opens a form whose Modal property is set to True, the method does not suspend execution. For example, the following code opens MODFORM.FSL, a form whose Modal property was set to True interactively. After the **open** statement, a simple counting loop executes and displays a message.

```
var
   modForm Form
endVar
modForm.open("modform.fsl") ; open form
for i from 1 to 5 ; execute loop
   message(i)
endFor
```

The loop executes immediately after the **open** statement; it suspends only upon the **wait**.

Using pre-built dialog boxes

ObjectPAL provides several pre-built modal dialog boxes. Methods for displaying them are defined for the System type, and begin with the letters "msg" (for example, **msgYesNoCancel**). For a complete listing, refer to the online ObjectPAL Help.

```
method pushButton(var eventInfo Event)
   msgYesNoCancel("Are we having fun yet?", "Inquiring minds want to know.")
endMethod
```

Many of these dialog boxes return a value, so you can use them to get user input. For example, the following code declares a variable named *theChoice* that gets assigned a value depending on which button the user clicks to close the dialog box. Then, a switch...endSwitch structure determines what to do next, based on the value of *theChoice*.

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

   theChoice = msgYesNoCancel("Exit", "Do you really want to quit?")
   switch
      case theChoice = "Yes" : doExit() ; do a custom method
      otherwise              : return
   endSwitch
endMethod
```

These dialog boxes are strictly modal, and code execution is suspended until the user responds. In other words, there is an implicit system **wait** statement. For example, in the following code, the counting loop does not execute until the user responds to the dialog box, either by choosing OK or using the control-menu to close it.

```
msgInfo("Modal dialog.", "Choose OK to start the counting loop.")
for i from 1 to 5 ; This loop does not execute until
   message(i)     ; the user responds to the dialog box.
endFor
```

For more information about System type methods, see Lesson 10 of the ObjectPAL tutorial in Chapter 2.

Calling Paradox dialog boxes

You can use ObjectPAL to display many of the dialog boxes built into Paradox. The procedures, all of which begin with the letters "dlg" (for example, **dlgCreate**), are defined for the System type. See the online ObjectPAL Help for details about specific procedures.

For example, this statement displays the Paradox dialog box for creating a table named ORDERS.DB, just as if you had chosen File | New | Table:

```
dlgCreate("orders.db")
```

Note As a programmer, you have no control over these dialog boxes once they are displayed. It's up to the user to use a built-in dialog box correctly.

Advanced topics

This section presents information about forms and dialog boxes that may interest experienced programmers.

MDI child windows

To understand the difference between forms and dialog boxes it is helpful (though not necessary) to understand the Windows concept of *MDI child*. In Windows terminology, Paradox is an *MDI* application. The multiple document interface (MDI) is an interface standard for Windows applications that allows the user to simultaneously work with multiple open documents. You can think of an MDI application as a mini-Windows session, complete with many applications represented by windows or icons.

The Desktop is an *MDI frame window*, which means that it provides behind-the-scenes management of all form and event activity for your applications. A standard form is an MDI child window. An MDI child has the following characteristics:

- It can be maximized to the full size of the frame window, or minimized to an icon within the frame window.
- It never appears outside the borders of the frame window.
- It does not have a menu, so its functions are controlled by the frame window's menu. (But note that ObjectPAL methods let you customize the menu bar and the Toolbar and intercept user menu actions.)
- The caption of each MDI child window is often the same as the name of the open file associated with that window (again, under ObjectPAL optional control).
- It can be tiled, cascaded, or manually resized by the user.

Standard menu

The Form Window Properties dialog contains a Standard menu checkbox, which is available to standard forms only. When checked (the default), this causes a form that is running to display the normal run-time menu, which may cause menus to "flicker" in some applications where an ObjectPAL custom menu replaces the run-time menu.

When Standard Menu is not checked, menus are left alone when a form is run, unless the form contains ObjectPAL code to create menus. This means the form displays the menu that was active when the form was run. This technique can help eliminate menu flicker.

Using openAsDialog

openAsDialog is a companion to the **open** Form method. It is intended for advanced users only (to run delivered forms with specific properties, for example). **openAsDialog** differs from **open** in two important ways:

- You can use any Windows style constant to specify display attributes (you can use only selected constants with **open**)
- A form opened using **openAsDialog** has its dialog property set to True.

open and **openAsDialog** use arguments to indicate the position, size, and display attributes of the form. Table 26.1 lists the Windows style constants you can use with these methods. Constants are described in the online ObjectPAL Help.

Table 26.1 Constants for open and openAsDialog

openAsDialog	open
WinDefaultCoordinate	WinDefaultCoordinate
WinStyleBorder	
WinStyleControlMenu	
WinStyleDefault	WinStyleDefault
WinStyleDialog	
WinStyleDialogFrame	
WinStyleHidden	WinStyleHidden
WinStyleHScroll	WinStyleHScroll
WinStyleMaximize	WinStyleMaximize
WinStyleMaximizeButton	
WinStyleMinimize	WinStyleMinimize
WinStyleMinimizeButton	
WinStyleModal	
WinStyleThickFrame	
WinStyleTitleBar	
WinStyleVScroll	WinStyleVScroll

Note Some properties cannot be set interactively, and therefore must be set using ObjectPAL on form **open** or **openAsDialog**. To open a form hidden, minimized or maximized, use properties WinStyleHidden, WinStyleMinimize, and WinStyleMaximize, respectively.

Note Calling **openAsDialog** does not make a form modal by default—it simply sets the specified display attributes. To make a dialog box with this method, use the WinStyleModal constant.

DesktopForm property

DesktopForm is a read-write property of form objects which enables the application to have a background form available to handle menu events when no other forms are active on the Desktop. Only one form on the Desktop should have this property set at any time.

This property takes effect only when there are no other MDI child windows on the Desktop. In this case, menu events are sent to this form (via its **menuAction** method). Its use is illustrated by this code fragment:

```
method open(var eventInfo Event)
   var
      f Form
   endVar
   f.attach()
   f.DesktopForm = TRUE
   f.hide()
endMethod
```

This code leaves the form active, yet hidden, on the Desktop and ready to respond to menu events if the Desktop becomes blank. If some other form were to open, menu events would again be directed to that new form by default. If you want the new form to use the menu of the Desktop form, set the new form's StandardMenu property to False. You still must attach code to the new form's built-in **menuAction** method.

DesktopForm doesn't appear in the Form Window Properties dialog box; it is available from ObjectPAL only. It is a persistent property, so if a form is saved with this property set to True, it will be True the next time the form is opened, either with ObjectPAL or interactively.

Linking with DDE

Dynamic Data Exchange (DDE) is a Windows protocol that lets Paradox share data with other applications that behave according to DDE protocol. Using DDE methods, you can create and store Paradox data in another application. You can also use DDE methods to send commands and Paradox data to the other application.

Note ObjectPAL and Paradox support OLE (for Object Linking and Embedding), another protocol for sharing data. Because OLE data can be stored in a table, the OLE type is grouped with the data types, in Chapter 5.

DDE conversations

Data sharing through a DDE link is like a conversation. A DDE conversation proceeds in three steps:

- Open the DDE conversation
- Converse
- Close the DDE conversation

The nature of a DDE conversation depends on the application you're conversing with. Consult the application's DDE documentation for information about data exchange, such as the commands it will accept and how it handles errors.

Opening a DDE conversation

Using **open**, you specify

- An application to converse with. The application must be on the user's path. (For information about setting a path, consult your DOS and Windows documentation.)
- A topic of conversation.
- An item within that topic (optional).

For example, the following method opens a conversation with ObjectVision (VISION), which must be on your path. Do not specify a file name extension for the application. The topic is a form named *Address*, and the item is the data in the field *LastName*. Notice that double backslash characters are required when specifying a path.

```
var ddeLink DDE endVar
ddeLink.open("vision", "C:\\vision\\forms\\Address.ovd", "LastName")
```

When you use DDE items with **open**, they have two important aspects: they're optional, and they vary from application to application.

To open the ObjectVision form in the previous example without naming a specific field, you could omit the *item* part of the **open** statement, like this:

```
var ddeLink DDE endVar
ddeLink.open("vision", "C:\\vision\\forms\\Address.ovd")
```

After opening the link, you can use **setItem** (discussed later) to specify an item.

There is no one syntax for specifying items in a DDE conversation, because different applications handle data differently. For example, rows and columns make perfect sense in a spreadsheet, but not in a paint program. To find out how to specify a DDE topic, refer to an application's documentation.

Conversing, DDE-style

Once the conversation is established, you can use the link to get data from the other application. The item value is *not* stored in the DDE variable—use the DDE variable to get the current data for the item or send new data to the item. As long as the DDE link exists, you can use the DDE variable to get and send data.

Getting data

To get data from a DDE link, you assign the value of the DDE variable to another variable. The variable must be of type AnyType, or of a type represented by AnyType. When you're not sure of the data type of the target data, or if the data type might vary, use an AnyType variable. The following example assumes that the data in the LastName field is a character string, and assigns the value of the DDE link to a String variable, *linkName*.

```
var
   myLink DDE
   linkName String
endVar
myLink.open("vision", "C:\\vision\\forms\\Address.ovd", "LastName")
                    ; item is field LastName
linkName = myLink ; sets linkName = value of field LastName
                  ; of the ObjectVision Address form
```

You can get more than one value from an application in two ways:

- Use **setItem** to change the item
- Open more than one DDE link

The following example uses **setItem** twice to get the values of two fields in an ObjectVision form.

```
var
   getNames DDE
   firstName, lastName AnyType
endVar

; link to the ObjectVision form
getNames.open("vision", "C:\\vision\\forms\\Address.ovd")

getNames.setItem("LastName")                    ; item is field LastName
lastName = getNames                             ; sets lastName = field LastName

getNames.setItem("FirstName")                   ; item is field FirstName
firstName = getNames                            ; sets firstName = field FirstName

msgInfo("The name is:", firstName + space(1) + lastName)
getNames.close()                                ; close the link
```

The next example opens two DDE links and stores the values in ObjectPAL variables.

```
var
   d1, d2 DDE
   firstName, lastName AnyType
endVar

d1.open("vision", "C:\\vision\\forms\\Address.ovd", "FirstName")
d2.open("vision", "C:\\vision\\forms\\Address.ovd", "LastName")

firstName = d1
lastName = d2

msgInfo("The name is:", firstName + space(1) + lastName)
d1.close()
d2.close()
```

The following example opens a DDE link to Quattro Pro for Windows, gets a value from a cell in the spreadsheet, and based on that value, sets the value of another cell.

```
method pushButton(var eventInfo Event)
var
   quattroLink DDE
   linkVal AnyType
endVar

quattroLink.open("qpw", "C:\\qpw\\notebk1.wb1", "A:A1")
linkVal = quattroLink

if SmallInt(linkVal) < 100 then  ; cast linkVal as a SmallInt
   quattroLink.setItem("B:B1")   ; move to page B, cell B1
   quattroLink = 999             ; set cell value to 999
else
   quattroLink.setItem("C:A2")   ; move to page C, cell A2
   quattroLink = 101             ; set cell value to 101
endIf
```

```
quattroLink.close()

endMethod
```

Sending data

Send data to another application by assigning a value to the DDE variable. For example, the following statements open a DDE link to the ObjectVision form ADDRESS.OVD, then set the value of the FirstName and LastName fields.

```
var
   ddeVar DDE
endVar

ddeVar.open("vision", "C:\\vision\\forms\\Address.ovd")
ddeVar.setItem("FirstName")

ddeVar = "Frank" ; set the value of the FirstName field in the OV form to Frank

ddeVar.setItem("LastName")
ddeVar = "Borland" ; set the value of the LastName field in the OV form to Borland
```

Sending commands

You can use **execute** to send commands to the other application. The nature of these commands varies from application to application. The following example uses the ObjectVision function @SETTITLE to specify the text to display in the ObjectVision title bar.

```
var d1 DDE endVar
; open a link to the ObjectVision form named Address
d1.open("vision", "C:\\vision\\forms\\Address.ovd")
d1.execute("[@SETTITLE(\"I'm in charge!\")]")
```

Closing the DDE conversation

To end a DDE conversation, use **close**. This method closes the DDE link between Paradox and the other application, but the other application stays open. To close the other application, you can use **execute** with the application-specific command (if available) before you use **close**. The following example gets data from an ObjectVision form, then uses the ObjectVision function @APPEXIT to close ObjectVision.

```
var
   d1 DDE
   firstName AnyType
endVar

; open a link to the ObjectVision form named Address
d1.open("vision", "C:\\vision\\forms\\Address.ovd", "FirstName")
firstName = d1
msgInfo("First name", firstName)

d1.execute("[@APPEXIT]") ; @APPEXIT is an ObjectVision function
d1.close()
```

Using libraries

A library is a file that stores custom methods, custom procedures, variables, constants, and user-defined data types. Using libraries, you can store and maintain frequently used routines and share code among several forms.

Important Forms do not have direct access to variables declared in a library. You must write custom methods or procedures to set and retrieve the values.

In many ways, working with a library is like working with a form. For example, to create a form, choose File | New | Form; to create a library, choose File | New | Library. Like a form, a library has built-in methods. You add code to a library just as you do to a form, using the Method Inspector and the ObjectPAL Editor. As with a form, you can open Editor windows to declare custom methods, procedures, variables, constants, data types, and external routines.

However, there are some important differences:

- At run time, a library does not appear in a window.
- A library cannot contain design objects; it can only contain code.
- In a library method, statements that use *Self* do not refer to the Library—instead, they refer to the object that called the method. Similarly, statements that use *Container* refer to the container of the object that called the library method. This same principle holds for the other built-in object variables: *active*, *lastMouseClicked*, *lastMouseRightClicked*, and *subject*. Refer to Chapter 12 for more information about built-in object variables.
- The scoping rules are different for libraries.

The next sections explain how to use libraries. They cover the following topics:

- Library type methods
- Controlling the scope of a library
- Using library variable as arguments
- Creating a library

- Adding code to a library
- Calling methods from a library

Library methods

The Library type has the run-time methods listed in Table 28.1. You can use these methods in your own code—attached to any object, even another library—to manipulate a library. For more information about these methods, refer to the online ObjectPAL Help. Also, libraries are used in the sample application.

Note New 5.0 methods let you create libraries and manipulate their code from ObjectPAL.

Table 28.1 Library methods

Method	Description
enumSource	Writes library code to a Paradox table.
enumSourceToFile	Writes library code to a text file.
execMethod	Executes a specified library method.
close	Removes the library from memory. You don't have to explicitly close a library—it is removed automatically when all referencing forms or libraries are closed—but using this method makes it easier to read the code and understand what's supposed to happen.
create (5.0)	Creates a library.
deliver (5.0)	Delivers a library.
design (5.0)	Displays an open Library in a Design window.
load (5.0)	Opens a library in the Design window.
methodDelete (5.0)	Deletes a library-level method from a form.
methodGet (5.0)	Gets a library-level method.
methodSet (5.0)	Sets the definition of a method attached to a library.
open	Opens a library and loads it into system memory, making it available to one or more forms and Desktops (see the section on controlling scope later in this chapter).
run	Runs a library.
save (5.0)	Saves a library to disk.

Controlling the scope of a Library

The *scope* of a library refers to its accessibility—that is, which objects have access to the library's code. A library's scope is determined by

- Where the Library variable is declared
- How the library is opened

Declaring a Library variable

A Library variable follows the same scoping rules as any other ObjectPAL variable:

- A variable declared in a method is available as long as that method is executing.

- A variable declared in an object's Var window is available to all methods attached to that object, and to all objects that object contains.

So, to make a library available to all objects in a form for as long as that form is running, declare the Library variable in the form's Var window, and declare the library routines in the form's Uses window.

For example, to declare the Library variable *mathLib*, attach the following code to the form's Var window.

```
var
   mathLib Library
endVar
```

To declare the library routines you want to use, attach the code to the form's Uses window. For example, the following code declares routines named *factorial* and *calcValue*.

```
Uses ObjectPAL
   factorial(const someValue SmallInt) LongInt
   calcValue(const rate Number, const qty Number) Number
endUses
```

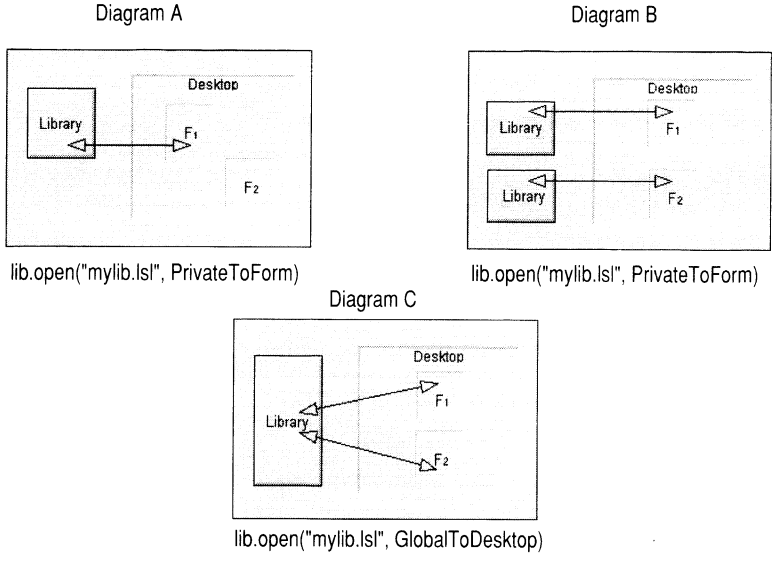
Opening a library

The Library type method **open** takes arguments that specify the scope. A library can be opened in one of the following ways:

- Global to the Desktop: every form in the Desktop (Paradox session) can access the library. This is the default scope.
- Private to the form: only the form that opened the library has access to its code.

Figure 28.1 shows the various **open** options.

Figure 28.1 Using **open** to specify library scope



Global to the Desktop

To open a library and make it available to every form in the current session of Paradox, use the argument GlobalToDesktop. For example, the following statement opens the library MYLIB.LSL.

```
lib.open("myLib.lsl", GlobalToDesktop)
```

As shown in Figure 28.1, diagram C, each form on the Desktop shares the same instance of the library. For example, if the form F1 calls a custom method that changes the value of a library variable, the changes will be seen by F2.

Important For two or more forms to share the same library, each form must open the library global to the Desktop, and each form must have a Uses window that declares which library routines to use. For example, in diagram C, F1 and F2 must open the library global to the Desktop and declare routines in a Uses window.

This level of scope is very useful in multi-form applications, because it lets several forms access the same custom methods and share the same global variables.

Note By default, a library opens global to the Desktop. Thus, the following statements are equivalent:

```
lib.open("myLib.lsl") ; these statements are equivalent
lib.open("myLib.lsl", GlobalToDesktop)
```

Private to the form

To open a library and limit its scope to the calling form, use the argument PrivateToForm as in the following example,

```
lib.open("mylib.lsl", PrivateToForm)
```

As shown in Figure 28.1, diagram A, if the form F1 opens a library it gets access to the library, but the form F2 doesn't. But diagram B shows that F2 can open the same library and get a different instance. In other words, F1 can call custom methods that change values of variables in the library, and so can F2, but the changes made by F1 are not seen by F2; likewise, changes made by F2 are not seen by F1. It's as if the forms are working in two separate libraries.

A form can have only one instance of a library that is private to the form—in other words, you can have multiple statements in the same form, but each successive **open** statement closes the current instance of the library and opens a new one.

```
lib.open("mylib.lsl", PrivateToForm)
```

Multiple instances

A library can be opened private to the form in one form and global to the Desktop in another form, and Paradox will load a new instance of the library, if necessary.

Using Library variables as arguments

You can use a Library variable as an argument in a custom method or custom procedure attached to an object in a form (or to the form itself). By passing a library as an argument, you can change the behavior of a routine (method or procedure) and still maintain the routine's independence. A routine may use a library and routines from the library, but the caller can determine the function of the routines by just changing the library.

For example, the following custom procedure **calcNum** could be declared in a button's Proc window. It takes two arguments: *num*, a Number, and *lib*, a Library.

```
proc calcNum(var lib Library, var num Number) Number

   ; This proc is attached to a button,
   ; it is not stored in a library.

   lib.doCalc(num)
   msgInfo("The result is:", num)
endProc
```

Now, suppose you have two libraries, *libOne* and *libTwo*. Each library contains a custom method named **doCalc**, but the **doCalc** in *libOne* does a different calculation (and returns a different result) than the **doCalc** in *libTwo*. The following code opens one of these libraries, depending on the value of the variable *x* (the code assumes variables are declared elsewhere), then passes the Library variable to the **calcNum** procedure. Then, **calcNum** calls **doCalc** in the specified library, and displays a different result in the dialog box depending on which library is used.

```
method pushButton(var eventInfo Event)
   num = 123.45
   if x = True then
      lib.open("libOne.lsl")
   else
      lib.open("libTwo.lsl")
   endIf
   calcNum(lib, num)
```

```
    lib.close()
endMethod
```

Creating libraries

To create a new library, choose File | New | Library. To edit an existing library, choose File | Open | Library. Then attach your code as explained in the following sections. When you're finished writing code, choose File | Save to save the library—both source code and executable code—to disk. To save only the executable code, choose Program | Deliver from an Editor window. See Chapter 33 for more information about saving and delivering libraries.

When you save a library, you give it a file name, and Paradox appends an extension of .LSL. When you deliver a library, you give it a file name, and Paradox appends an extension of .LDL.

When you create a new library, an Editor window opens. The Editor window represents the library. *You can't place objects in this window.* Instead, use the Method Inspector to attach code to the library. To display the Method Inspector, right-click in the library window and choose Methods from the pop-up menu, or press *Ctrl+Spacebar.* Use the Method Inspector to open, create, and delete methods and procedures and to declare external routines (uses), variables, data types, constants, and procedures.

A library is stored in a separate file on disk so it's easy to copy a library and distribute it with your applications.

Adding code to a library

Using the Method Inspector and ObjectPAL Editor windows, you can add code to a library in the following ways:

- Attach code to the built-in methods.
- Add custom methods.
- Add custom procedures.
- Declare variables, constants, data types, and external routines.

Attaching code to the built-in methods

Every library has the following built-in methods: **open**, **close**, and **error**.

You can attach code to these built-in methods as you would with any other object.

A library's built-in **open** method is called when the library is first opened, **close** is called when the library is being closed, and **error** is called when code in the library generates an error. Typically, you use **open** to initialize global library variables and **close** to "tidy up" after using the library. By default, a library's **error** method calls the **error** method of the form that called the library routine.

Adding custom methods

This section describes how to add custom methods to a library. The custom methods in a library can be called by other methods in the same library, by methods in other forms, and by methods in objects in other forms. This accessibility makes libraries so useful.

The syntax for declaring a custom method is

METHOD *methodName* ([var | const] *argList*) [*returnType*]
  The body of the method goes here
ENDMETHOD

methodName represents the name of the method, and *argList* represents a comma-separated list of argument/data type pairs, optionally preceded by the keyword var or const, as appropriate. (Chapter 6 explains how and when to use these keywords.) The optional argument *returnType* specifies the data type of the value (if any) returned by the method.

For example, the following code declares a custom method **hello** that takes one argument, a String variable named *userName*. This custom method displays a dialog box.

```
method helloUser(userName String)
   msgInfo(userName, "Hello")
endMethod
```

See the entry for **method** in the ObjectPAL online Help for details about declaring methods.

Adding custom procedures

From the Method Inspector, you can choose Procs to open an Editor window where you can declare custom procedures for the library.

Note Unlike custom methods, which can be called from other forms and other objects, custom procedures can be called only from within the library in which they are declared.

The syntax for declaring a custom procedure is

PROC *procName* ([var | const] *argList*) [*returnType*]
  The body of the procedure goes here
ENDPROC

procName represents the name of the procedure, and *argList* represents a comma-separated list of argument/data type pairs, optionally preceded by the keyword **var** or **const**, as appropriate. (Chapter 6 explains how and when to use these keywords.) The optional argument *returnType* specifies the data type of the value (if any) returned by the procedure.

For example, the following code declares a custom procedure **addOne** that takes one argument, *incNum*, and returns a Number value.

```
proc addOne(var incNum Number)  Number
   incNum = incNum + 1.00
   return incNum
endProc
```

See the topic "proc" in the online ObjectPAL Help for details about declaring procedures.

Declaring variables, constants, data types, and external routines

From the Method Inspector, you can declare variables, constants, data types, and external routines by choosing Var, Const, Type, or Uses, respectively, to open the appropriate Editor window. Items declared in these windows are global to the library but cannot be accessed by other forms or objects. However, other forms and objects can call library routines that access these variables. For more information about declaring these items, see Chapters 3 and 6.

Calling methods in a library

To call a method in a library, you must first declare the method in the Uses window of the object doing the calling. For example, suppose you want a button's **pushButton** method to call a custom method from a library. Declare the method in the button's Uses window (or in the Uses window of an object that contains the button), so Paradox knows where to look for the method and knows what arguments it will take. You can declare more than one library method in the same Uses window.

The following pseudocode shows how to declare a library method in a Uses window.

```
Uses ObjectPAL
   methodName ( [var | const] argList) [returnType]
EndUses
```

Note Arguments and data types must be declared in the Uses window exactly as they are declared in the library.

The keyword ObjectPAL is required to indicate that you're calling methods from an ObjectPAL library, rather than from a dynamic link library (DLL). (See the topic "uses" in the online ObjectPAL Help for more information about using a DLL.)

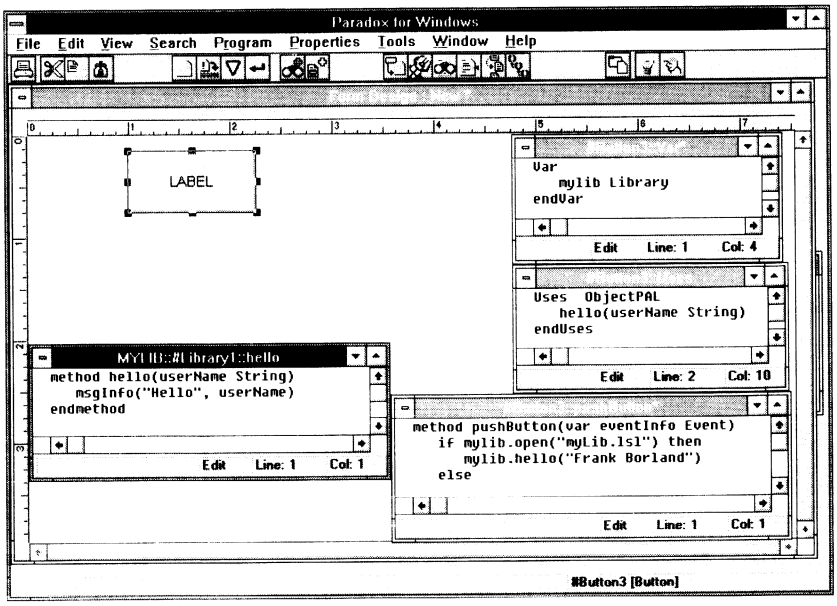
Next, *methodName* represents the name of the method to call, and *argList* represents a comma-separated list of argument/data type pairs, optionally preceded by the keywords var and const, as appropriate. (Chapter 6 explains how and when to use these keywords.)

The optional argument *returnType* specifies the data type of the value (if any) returned by the method.

Note You can write code that calls a library method, and the code will compile in a design window even if the library doesn't exist. However, the library must be present in order to run the form.

Figure 28.2 shows an example of calling a custom method from a library. The example is a form that contains a button. Three Editor windows are open for the button: a Var window, a Uses window, and a window for the **pushButton** method. The code in the Var window declares the variable *lib* to be of type Library and makes it visible to all methods attached to the button. The code in the Uses windows declares that the ObjectPAL custom method **hello** is stored in a library, and it declares the argument to pass to **hello**. The code in the **pushButton** window opens the library and calls the custom method.

Figure 28.2 Example of calling a custom method from a library



In this figure, the **pushButton** method opens a library and calls a custom method. To do this, we must first declare a Library variable (Var window) and declare the method we want to call (Uses window). The custom method is shown in the window at the upper left.

Important The code in a library executes on behalf of the object that called it, and the object sets the context for a library method that the code calls. So, if a box calls a library method, statements that use *Self* refer to the box, and statements that use *Container* refer to the box's container. This same principle holds for the other built-in object variables: *active*, *lastMouseClicked*, *lastMouseRightClicked*, and *subject*. Refer to Chapter 12 for more information about built-in object variables.

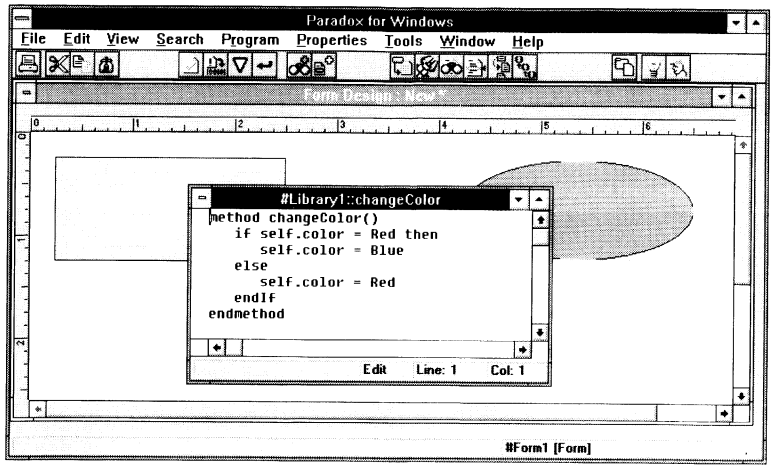
For example, Figure 28.3 shows a form that contains one box named *theBox* and one ellipse named *theEllipse*. The form opens a library that contains a custom method named **changeColor**.

```
method changeColor()
   if self.color = Red then
      self.color = Blue
   else
      self.color = Red
   endIf
endMethod
```

Now, when a method attached to *theBox* executes the following statement, *theBox* changes color, and when a method attached to *theField* executes the statement, *theField* changes color.

```
lib.changeColor()
```

Figure 28.3 Using *Self* in a library routine



In a library routine, *Self* refers to the object that called the routine, just as it does in any other method or procedure. In this figure, when the box calls **changeColor**, *Self* refers to the box; when the ellipse calls **changeColor**, *Self* refers to the ellipse.

Creating and playing scripts

A script consists of ObjectPAL code in its own file, not attached to a form. It's an object and appears on the Desktop as an icon, it doesn't appear in a window, and it doesn't contain any design objects.

The Script type includes built-in methods for manipulating scripts—and the code they contain—from within an ObjectPAL method or procedure. Like any other object, a script also has windows for declaring variables, constants, procedures, types, and external routines. You can also declare custom methods. See the topic "Script Type" in the ObjectPAL Help for more information and examples.

Use a script when you want to execute code without opening and displaying a form window. For example, you can use scripts to execute a scan...endScan loop to capitalize a field's values to uppercase or to enumerate the properties and methods attached to a form. While these tasks can be accomplished by creating a blank form, placing a button in it, and then attaching the appropriate code to the **pushButton** method, a script "sidesteps" the first two steps.

From a script, you have complete access to the ObjectPAL run-time library, so you can control other objects. For example, you can call other scripts, open and work with tables, forms, and reports, and run queries. You can call methods attached to other objects, and get and set their properties.

Creating a script

Following are the steps for creating a script.

1 Choose File | New | Script. An ObjectPAL Editor window opens, containing the following text:

```
method run(var eventInfo Event)

endMethod
```

2 This is a standard ObjectPAL Editor window, so you can edit, compile, and debug the **run** method as you would any other. From a script, as from any other object, you can open and close other forms, create objects, get and set properties and values, display messages, and trigger methods.

3 You can display the Method Inspector by choosing View | Methods. From there, you can declare variables, constants, data types, procedures, custom methods, and DLLs to use. Keep in mind, though, that whatever you declare is visible only to the script's **run** method.

4 When you're finished editing, close the window. A dialog box prompts you to enter a name for this script. Enter a name and choose OK to save the script to disk.

5 Like a form, a script can be saved by choosing File | Save or File | Save As, and it can be delivered by choosing Program | Deliver. Saved scripts can be changed; delivered scripts cannot. See Chapter 33 for more information about delivering forms and scripts.

Playing a script

You can play a script using Paradox interactively or from within a method. In either case, the result is that you execute the script's **run** method.

Using Paradox interactively

Following are the steps for running a script using Paradox interactively.

1 Choose File | Open | Script.

2 A dialog box lists available scripts. Choose one, choose Play, and choose OK. The script executes.

From within a method

Use the System type method **play** to play a script from within a method or procedure. For example,

```
switch
   case theValue = "this" : play("doThis")   ; play script "doThis"
   case theValue = "that" : play("doThat")   ; play script "doThat"
   otherwise              : play("theOther") ; play script "theOther"
endSwitch
```

Handling run-time errors

This chapter presents concepts and techniques that will be useful in handling run-time error conditions in the ObjectPAL environment. It begins by discussing the different types of errors that occur when building and running applications using ObjectPAL. It then defines the types of run-time errors, and explains how ObjectPAL classifies these errors according to severity. It introduces the important concepts of the Paradox *error stack*, and the **try...onFail** block. The chapter then explains default system behavior for error conditions, and concludes with information about building customized error-handling into forms. The topics in this chapter are

- Three categories of errors: compiler, logic, and run-time
- Classifying errors by severity
- The error stack
- The **try...onFail...end try** block
- Default system error handling
- Custom error handling
- Advanced topics

Categories of errors

Three general categories of errors can prevent an application from running:

- Compiler errors
- Logic errors
- Run-time errors

Compiler errors

Compiler errors result from statements that are not well formed, or that ObjectPAL does not recognize. When the compiler detects an error, it stops, opens an Editor window for the proper code module (if necessary), positions a cursor on the statement nearest to the error, and displays an error message in the status line of the Editor window. This leaves Paradox in the method Edit mode so that you can immediately observe and correct the error.

Compiler errors are typically caused by misspelled, missing, or misplaced elements in expressions. The compiler detects, for example, an improper method or procedure calling sequence where the parameter list contains the wrong number or type of arguments. It also detects type mismatches when a variable of one type is assigned an incompatible value, such as String to SmallInt. The following example contains several different types of errors that the compiler detects.

```
proc example (var eye smallInt)
   for i FORM 1 to 10    ; [1] keyword "FROM" misspelled
      eye = i
                         ; [2] FOR loop needs the keyword "ENDFOR"
   eye = custproc()      ; [3] custproc() returns a String not a smallInt
endProc

proc custproc() String
   return("some string" ; [4] missing closing parenthesis
endProc
```

Tip It is good programming practice to explicitly declare each variable in your form. (Remember that a variable used without a declaration is implicitly declared to be of type AnyType.) When you declare variables explicitly, the compiler detects certain errors that it might otherwise miss and generates more efficient code.

Logic errors

Logic errors result from code that is syntactically correct but produces unexpected results. Logic errors often result from faulty algorithms or incorrectly implemented control structures, like endless loops or calculations that return the wrong results because they are based on an incorrect formula. Another type of logic error involves variable scope, where the variable you intend to operate on is, for example, a different variable with the same name.

Logic errors can sometimes produce run-time errors (described next). For example, a method that calls itself recursively can eventually use up all available memory or stack causing a run-time error.

Run-time errors

Run-time errors result from statements that compile (because they are syntactically valid) but for some reason cannot be carried out to completion. The classic example is an expression of variable A divided by variable B, where variable B evaluates to 0 at run time. Other examples of operations that result in run-time errors are trying to open a

table that does not exist, read from a floppy disk when the drive door is open, assign an inappropriate value to a variable, or manipulate a property that doesn't belong to a particular object.

The compiler catches clear violations of the reference to nonexistent properties, but can't detect an invalid object property reference if it is made off of an object variable rather than a specific named object. For example, the following code is syntactically correct, but it causes an error if the object assigned to the UIObject variable *someObject* doesn't exist, or if the object exists but doesn't have the Color property.

```
proc setRed(var someObject UIObject)
   someObject.color = Red
endProc
```

Another class of run-time errors results when Paradox runs out of resources to carry out the current operation. For example, a run-away recursive **while** loop, mentioned in the discussion of logic errors, can cause the system to run out of stack or memory space.

Summary

The more experienced you become with ObjectPAL, the fewer compiler errors your code will contain. You can reduce logic errors by carefully planning, designing, and testing your programs.

In contrast, you should build run-time error handling into your code from the beginning. (Here is an example: what should happen when the table you want to access is found unexpectedly to be locked or missing?) The remainder of this chapter presents tools and techniques for run-time error handling in the ObjectPAL environment.

Understanding run-time errors

This section discusses several key concepts that are important to fully appreciate and use the powerful error-handling capabilities of Paradox. They are

- Run-time error levels
- The error stack
- The try...onFail...end try block

Run-time error levels

ObjectPAL classifies run-time errors as "critical" or "warning" depending on severity.

A *warning error* is best thought of as an unsuccessful return from a method or procedure call, meaning that it could not complete its operation successfully. A typical ObjectPAL method returns the Logical value of True to indicate success, and the Logical value of False to indicate a warning error. However, this error will not stop execution of the program.

A *critical error* occurs when Paradox determines that it cannot complete the requested operation, and that it is not possible or advisable to return with a warning error. Critical errors can be caused by

- An assignment error, such as referring to a nonexistent field in a table
- A method or procedure call with invalid arguments (if the call is not caught by the compiler)
- An invalid return value from a method or procedure
- An attempt to read or write past the end of a table or file

For example, when the following statement executes, if *someObject* doesn't have a Color property, the attempt to assign a value to the Color property causes a critical error.

```
someObject.color = Red
```

This error will stop the program.

Classification of an error as critical or warning

In a complex, multilevel environment such as Paradox, it is sometimes difficult to indicate which error conditions cause critical errors and which cause warning errors. In Paradox, computing tasks are distributed among a number of modules (for example, the database engine, the forms system, ObjectPAL, and so on); these modules interact and can cause errors and warnings in unpredictable ways. The next sections describe the difference between the two levels of run-time errors to help you develop appropriate strategies for dealing with each of them.

The error stack

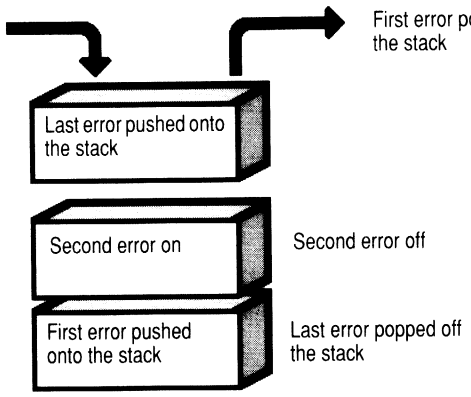
Paradox provides a simple mechanism called the *error stack*, which returns information about the most recently detected run-time error. Each time a run-time error occurs, ObjectPAL stores information about the error on the error stack. You can use a number of methods to examine and even modify this information.

Nearly every ObjectPAL statement that calls a method or procedure in the run-time library affects the error stack as it executes. The exceptions are error-related methods (for example, **errorCode**), message-display methods (for example, **msgInfo, message, beep**), and user-defined custom methods and procedures. When a statement causes a run-time error, information about the error is pushed onto the stack; when a statement executes successfully, it clears the stack. Knowing this, the next logical question would be, "What is this information, and how can I use it?" The next sections provide the answers.

Error stack information and format

As illustrated in Figure 30.1, the error reporting mechanism is implemented as a Last In First Out (LIFO) stack so that the most recent information "pushed" onto the stack is the first information removed ("popped") from the stack.

Figure 30.1 The error stack



Each time a run-time error occurs, Paradox pushes one or more error information records onto the error stack. Each record contains

- *error code*, a numeric value that identifies the error
- *error message*, a text string to display to the user

Note Clicking the browse buttons in the Error Display dialog box, whether or not ObjectPAL code is involved, navigates through this same error stack.

Important The stack is cleared as each method or procedure executes; therefore, information is available only for the very last method executed. This being the case, why is the stack needed?

The reason is that a single ObjectPAL statement can generate several layers of error information as the various Paradox modules encounter the error and push error information onto the stack.

For example, suppose a custom method attempts to open on a TCursor on a nonexistent table. This would cause errors in two different areas: the database engine module and the ObjectPAL run-time module. The database engine would encounter the error first, and would push low-level information about the failure onto the error stack. Then the run-time module will encounter the error and push information from its level onto the stack. In this example, the stack will contain two messages: the topmost stack entry `An error was triggered in the 'open' method on an object of type TCursor`, followed by a second stack item, `No such table. File foo.db`.

Error stack procedures

ObjectPAL provides several methods (defined for the System type) to examine, display, and modify run-time error information. Use these procedures to find out what the error was, to see why it happened, to modify or add information about the error to the stack, and to display this information to the user.

errorCode and **errorMessage** return the error code and error message, respectively, off the top of the stack. Since these procedures leave the stack unchanged, **errorPop** is provided to remove one entry at a time from the stack in order to access the remaining

items. The **errorClear** method clears all entries off the stack. The **errorLog** procedure pushes a new entry (error code plus message string) onto the error stack. Finally, **errorShow** invokes the standard ObjectPAL error dialog presenting the user with a standard dialog form to browse the error stack.

As an example, this simple error alert method displays the error code in the dialog box title bar, and the error message in the box itself.

```
proc myErrorDisplay()
   msgInfo("error code: " + errorCode(), errorMessage())
endProc
```

Note Calls to **errorCode** and **errorMessage** do not affect the contents of the error stack, but other methods and procedures do. Therefore, call these procedures first in your error-handling routines.

Displaying the Error dialog box

To display error information, consider using **errorShow**, which pops up the default Error Display dialog box. This dialog box has a common look and feel with the dialog box used by the forms run-time system and enables the user to browse through the error stack.

Note **errorShow** effectively pops all the entries off the error stack, so be certain you no longer need this information before invoking this procedure.

In a production application you will want to do more than display the error information. You might, for instance, want to handle certain types of errors yourself, and let Paradox take care of the others. The following sections show you how to use the error stack procedures by building an example custom error-handling procedure.

Using errorCode

The example begins by using **errorCode** in a switch...endSwitch structure to filter out and handle selected error codes. Errors are coded using standard Paradox error constants for clarity. **errorCode** returns the error code on the top of the error stack. Use a **switch** statement to exclude codes outside of a specific set of error codes.

```
proc ourErrorProc()
var
   erc smallInt
endVar
   erc = errorCode()
   switch                              ; only handle selected error conditions
      case erc = peObjectNotFound      : noObject() ; execute a custom method
      case erc = pePropertyNotFound : noProperty() ; execute a custom method
      otherwise                        : return
   endSwitch
   ;do more processing...
endProc
```

ObjectPAL provides constants for error codes. To display the list, open an ObjectPAL Editor window and choose Tools | Constants. Then, from the Types of Constants column, choose Errors. The constants appear in the Constants column. Constants are also listed

in the online ObjectPAL Help. You can use these constants in methods as shown in the examples in this chapter.

Note You can use error **HasErrorCode** to search the error stack for a specific error code. **HasErrorCode** searches the error stack for a specified errorCode, where the errorCode is an Errors constant or a user-defined error constant. See the online ObjectPAL Help for more information.

Using errorMessage

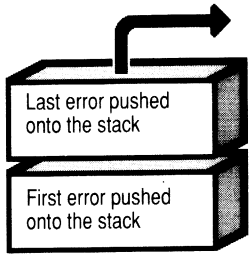
errorMessage returns the text of the error message on the top of the error stack. A typical use for **errorMessage** is to log error messages to a file. Another use is to combine the default error message with a custom message. The following code shows both uses by adding a custom error-logging routine to the previous example.

```
proc ourErrorProc()
var
   erc smallInt
   ers String
   errMsg TextStream
endVar
   erc = errorCode()
   switch                              ; only handle selected error conditions
      case erc = peObjectNotFound      : noObject() ; execute a custom method
      case erc = pePropertyNotFound : noProperty) ; execute a custom method
      otherwise                        : return
   endSwitch
   ers = String(Date())+","+String(errorCode())+","+errorMessage()
   errMsg.open("errorLog.txt","A")
   errMsg.writeLine(ers)
   errMsg.close()
   ;do more custom processing...
endProc
```

Using errorPop

Use **errorPop** to remove one layer at a time from the error stack to make successive levels of error information available. (Figure 30.2). (To remove all layers from the stack, use **errorClear**.)

Figure 30.2 The error stack: pushing and popping information



Calling **errorShow, errorCode, errorMessage**, or **errorLevel** retrieves information from the top of the stack

To retrieve lower-level information, call **errorPop**, then call **errorShow, errorCode, errorMessage**, or **errorLevel**

errorPop returns True when it succeeds and False otherwise (because there are no more layers on the stack). The next example logs the entire error stack.

```
proc ourErrorProc()
var
   erc smallInt
   ers String
   errMsg TextStream
endVar
   erc = errorCode()
   switch                              ; only handle selected error conditions
      case erc = peObjectNotFound      : noObject() ; execute a custom method
      case erc = pePropertyNotFound : noProperty() ; execute a custom method
      otherwise                        : return
   endSwitch
   ers = String(Date())+","+String(errorCode())+","+errorMessage()
   errMsg.open("errorLog.txt","A")
   errMsg.writeLine(ers)
   While errorPop()
      ers = String(Date())+","+String(errorCode())+","+errorMessage()
      errMsg.writeLine(ers)
   endWhile
   errMsg.close()
   ;do more custom processing...
endProc
```

Adding information to the stack

Use **errorLog** to modify or add information to the error stack. **errorLog** takes two arguments, an error code and an error message. This example pushes a custom warning message onto the error stack for any errors that are not dealt with in this routine.

```
proc ourErrorProc()
var
   erc smallInt
   ers String
   errMsg TextStream
endVar
   erc = errorCode()
   switch                          ; only handle selected error conditions
      case erc = peObjectNotFound   : noObject() ; execute a custom method
      case erc = pePropertyNotFound : noProperty() ; execute a custom method
      otherwise                     :
         errorLog(MYCODE, "Can't deal with this one")
         myAdvancedErrorProc()      ; invoke more specialized processing
         return
   endSwitch
   ers = String(Date())+","+String(errorCode())+","+errorMessage
   errMsg.open("errorLog.txt","A")
   errMsg.writeLine(ers)
   While errorPop()
      ers = String(Date())+","+String(errorCode())+","+errorMessage()
      errMsg.writeLine(ers)
   endWhile
   errMsg.close()
   ;do more custom processing...
endProc
```

try...onFail block

The try...onFail block consists of a *transaction block*, which contains one or more ObjectPAL statements (called a *transaction*), and a *recovery block*, also consisting of one or more ObjectPAL statements that specify what to do if the transaction fails. That is, you want the *transaction* to succeed, and if it doesn't, you want control to pass to the *recovery block*. The try...onFail block looks like this:

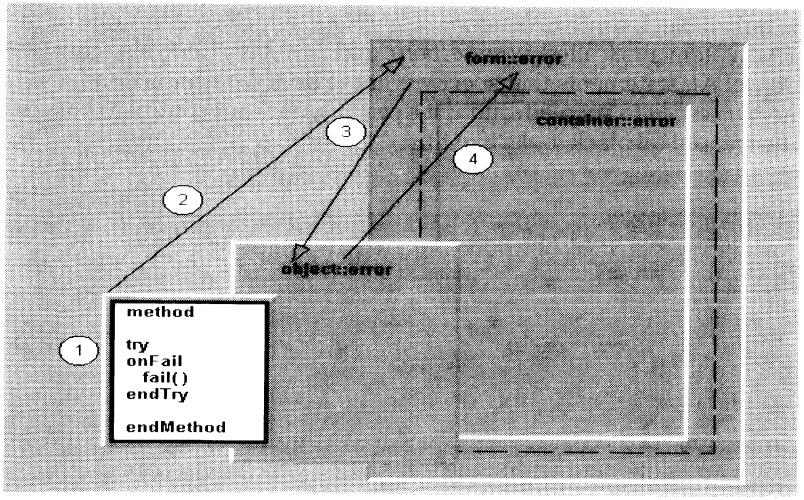
```
try
   [transaction block]
onFail
   [
      recovery block
      [retry]                          ; optional keyword
      [fail ( [ errCode, errMsg] ) ]   ; optional procedure, optional arguments
   ]
endTry
```

If the transaction succeeds, the program skips to the endTry keyword and the error stack is cleared. If any statement within the transaction generates a critical error, control shifts immediately to the recovery block.

For example, if the fourth statement in a five-statement transaction block fails, execution resumes at the first statement of the onFail block. The fifth statement of the transaction does not execute, but the effects of the first three statements, which executed successfully, remain in effect.

Important You can nest one or more explicit try...onFail blocks around critical sections of code. When a transaction block fails, the error jumps to the next-highest block, and so on, until no more blocks are inside the method. At that point, control passes to the recovery clause in the implicit try...onFail block which Paradox places around each method. It is from here that an ErrorEvent is generated and sent to the form, as shown in Figure 30.3.

Figure 30.3 The try...onFail model



1. Within a method, a try...onFail block attempts a transaction. In the recovery block, a call to **fail** generates an ErrorEvent.
2. The error triggers the form's built-in **error** method.
3. By default, the form's **error** method calls the **error** method of the object whose code caused the error.
4. By default, the error bubbles up through the containership hierarchy until it once again reaches the form's **error** method, which displays a dialog box.

retry keyword

Within the recovery block, you can use the retry keyword to cause the transaction block to execute again.

fail procedure

Also, you can force a failure by calling the System type procedure **fail** in the recovery block or within procedures called by the recovery block. **fail** can be called without arguments (but parentheses are still required), or with an error code and an error message, as in

```
fail(peObjectNotFound, "The object doesn't exist!")
```

A **fail** call can be nested in one or more custom method or procedure calls deep from within the onFail block.

Note Be careful using **fail** in these cases. When variables local to all nested methods or procedures are removed from the stack, any special objects (such as large text blocks) are deallocated, and local reference objects (such as TCursors) are closed. Of course, changes to variables outside the scope of these methods or procedures remain intact as do changes to tables successfully committed before the failure occurred.

Default system error handling

This section describes Paradox's default mechanisms for handling critical and warning errors. To understand the differences in the way ObjectPAL handles critical errors and warning errors, keep the following issues in mind as you read this section:

- Information about the error is always pushed onto the error stack.
- When does the system display this information automatically by default?
- Under what conditions is the built-in **error** method invoked?
- Where, if anywhere, will your code continue to execute after an error?
- Can the system default error behavior be modified?

Errors can be further grouped into two categories: those generated as a result of ObjectPAL activity, and those generated as a result of user interaction with Paradox. This section deals with the former; the latter category is covered in the "Advanced error handling topics" section at the end of this chapter.

Warning errors

A warning error is characterized by the following default system behavior:

- Paradox pushes information about the error onto the error stack.
- Program control is returned to the next statement in your code. You are alerted to the error by checking the return value of the method or procedure that failed (a standard programming practice).
- Stack error information is not displayed by default but might be examined using the error methods.
- The built-in **error** method is not invoked by default, but it can be, by calling **fail**.
- You can change the default behavior. As explained in a later section, the system can be instructed to treat warning errors as it does critical errors.

Critical errors

Critical errors are characterized by the following default system behavior:

- Paradox pushes information about the error onto the error stack.
- Control is returned to the onFail section of the implied try...onFail block. (An explanation follows.)
- The built-in **error** method is invoked.
- When the error bubbles to the form-level, stack error information is displayed via the standard Paradox Error Dialog box.

- Default behavior is modified either by creating custom error methods, or by placing an explicit try...onFail block around appropriate statements or blocks of statements with custom error-handling logic.

The implicit try...onFail block

Just as you can place a try...onFail block around a transaction in your own ObjectPAL code, Paradox places an implicit try...onFail block around every built-in method for every design object on a form. In other words, Paradox treats each built-in method as a separate transaction.

When an ObjectPAL statement causes a run-time error, the transaction implied by the built-in method fails. Paradox first checks the code to see if the statement is enclosed in an explicit try...onFail block. If no such block is present, if the block doesn't adequately handle the error, or if the block calls **fail**, control switches immediately to the *implicit* try...onFail block, which in turn generates an ErrorEvent. With this ErrorEvent, the appropriate **error** method is invoked.

The built-in error method

Every UIObject has a built-in method called **error**, which executes when an error isn't handled fully in the onFail section, or when an ObjectPAL statement calls **fail**.

Following standard Paradox behavior, the ErrorEvent goes first to the form. Then the form calls the built-in **error** method of the current object whose code triggered the error. The current object can handle the error, bubble it up through the containership hierarchy, or both.

For example, suppose a form contains a table frame, and some code attached to the table frame causes a critical error. When that code executes and causes the error, it generates an ErrorEvent that triggers the form's built-in **error** method. By default, the form's built-in **error** method dispatches the ErrorEvent to the table frame's built-in **error** method, from which it might bubble up the containership hierarchy until the error reaches the form. Finally, the form's built-in **error** method displays a dialog box to tell you about the error, or displays it on the status bar if it's a warning level error.

Custom error handling

This section presents examples of the basic techniques you can use for handling warning and critical run-time errors.

Handling warning errors

The general technique for handling warning errors is familiar to most programmers: when there is any possibility of failure in the run-time environment, test the return value of methods and procedures for success or failure.

Because warning errors are typically caused by methods that return a Logical value, you can handle them within an if...then block or a **switch** statement. For example, the return values of the Table type methods **attach** and **rename** in the following example are checked for success or failure using an if...then block.

```
proc someProc()
   var
      t Table
   endVar
   if not t.attach("oldname.db") then
      msgInfo("Warning", "Couldn't open the table.") ; report the error
      openError() ; execute custom method to handle the error
      return
   endIf
   if not t.rename("newname.db") then
      renameError() ; execute custom method to handle the error
      msgInfo("Warning", "Couldn't rename table.") ; report the error
      return
   endIf
endProc
```

Use **errorCode** and **errorMessage** with the filtering techniques discussed earlier to obtain or display error information about the warning from the error stack, as in the following example.

```
proc someProc()
var custTC TCursor endVar
      if custTC.close() then  ; equivalent to "if custTC.close() = True then"
      doSomething()           ; do some processing
   else
      if errorCode() = peTableClose then
         msgInfo("Problem", errorMessage()) ; report the error to the user
      else
         ; call a custom procedure
         furtherErrorProcessing(errorCode(),"Requires further processing")
      endIf
   endIf
endProc
```

Handling critical errors

The two basic approaches to handling run-time critical errors are the try...onFail block, and the modified built-in **error** method.

try...onFail block

The first approach makes use of the fact that ObjectPAL returns control after a critical error to the first statement in the **fail** section of the innermost try...onFail block. The following example uses the try...onFail block to trap critical errors having to do with setting the property of a nonexistent object. Assume a form contains *box1*, and that *box1* contains *box2*.

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

box1.box2.color = Blue                    ; this works
s = "box5"                                ; box5 doesn't exist

try
   box1.(s).color = Red                   ; try to set color of box5
```

```
onFail                                    ; handle the error
   msgStop("Error", "Couldn't find " + s)
   s = "box2"                             ; box2 exists
   reTry                                  ; try again
endTry

s="box6"                                  ; box6 doesn't exist
try
   box1.(s).color = Green
onFail
   fail(peObjectNotFound, "The object " + s + " does not exist.")
endTry
endMethod
```

In the following example, an overflow error occurs when an out-of-range number is assigned to a variable of type SmallInt. **retry** and **fail** are used to respond differently to particular error codes.

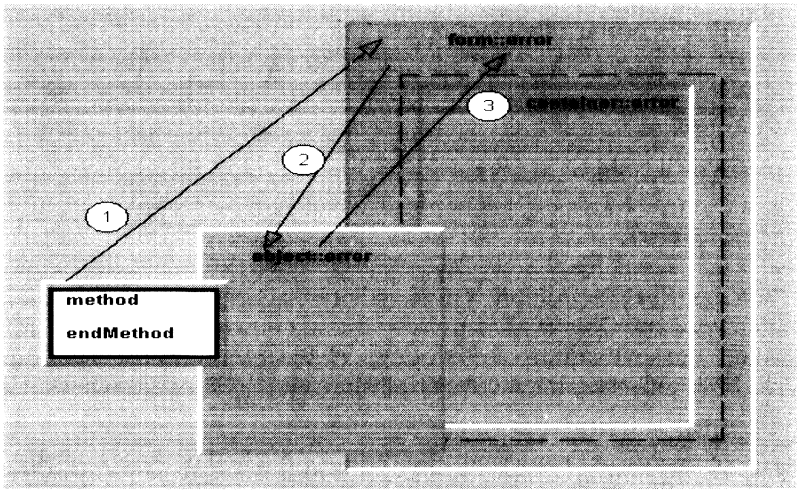
```
method pushButton(var eventInfo Event)
var sm    smallInt
    maxsm smallInt
endVar
   maxsm = 32767
   try
      sm = maxsm + 1
   onFail
      if errorCode() = peOverFlow then
         msgStop("Error", "overflow error in assignment")
         maxsm = -10
         retry
      else
         fail()
      endIf
   endTry
endMethod
```

Note The peBreak error code is generated when a user interrupts program activity by typing *Ctrl+Break*. Therefore, depending on the needs of your application, it might be particularly useful to check for the peBreak error code in the onFail section of critical operations. Use the **disableBreakMessage** procedure to prevent the user from interrupting a running program with *Ctrl+Break*.

error method

As Figure 30.4 illustrates, you can place object-specific error-handling code on the object's built-in **error** method. Alternatively, you can handle errors for groups of objects by placing code on the built-in **error** method of any object in the containership hierarchy. Finally, you can handle errors globally for all objects on the form by attaching the code to the form's built-in **error** method.

Figure 30.4 The event model for ErrorEvents



1. A statement in a method causes an error, which triggers the form's **error** method.
2. By default, the form calls the **error** method of the object whose code caused the error.
3. By default, the error bubbles up through the containership hierarchy back to the form's **error** method, which takes the appropriate action.

When the error reaches the form for the second time, by default it displays a dialog box.

Information about the error level is stored in the event packet *eventInfo*. When writing custom code for the **error** method, use ErrorEvent type methods **reason** and **setReason**, respectively, to read and write this information. To determine the severity of the error, ObjectPAL has defined the constants ErrorCritical and ErrorWarning for use with **reason** and **setReason**.

As an example of the use of these constants, and of the power of the **error** method, suppose you want all errors (critical and warning) to be handled as critical errors. (In fact, the **trapOnWarning** method, discussed later in this chapter, does this automatically, but we like this example.) The following code attached to an object's built-in **error** method creates the desired effect:

```
method error(var eventInfo ErrorEvent)
   if eventInfo.reason() = ErrorWarning then
      eventInfo.setReason(ErrorCritical)
   endIf
endMethod
```

When this object's **error** method executes, it calls **reason** to find out the error level, converts it to a critical error if necessary, and then bubbles the event packet up through the containership hierarchy to the form, which by default displays the critical error dialog box.

Advanced error handling topics

This section discusses the following topics:

- Interactive errors
- Trap on warnings
- Where to attach code

Interactive errors

This section describes the default system activity that occurs for errors generated independently of any ObjectPAL activity as a result of user interaction. For example, when you use the keyboard to scroll through a table frame and try to scroll past the end of the table, that causes an interactive error. Here is what happens when an interactive error occurs:

- Paradox generates an ErrorEvent, and the error method of the active object is invoked. As with any built-in method, the form is called first, and then the event goes to the active object (the "culprit" who generated the error, in this case); if the event is not handled there, it bubbles up the containership hierarchy until it reaches the form again.
- If the ErrorEvent does bubble up to the form a second time, and if the error was an interactive *warning* error, the form's built-in **error** method generates a StatusEvent. This invokes the built-in **status** method, which follows a similar route through the form, the active object, and back to the form again, after which a status message appears in the status message area of the screen.
- On the other hand, if the ErrorEvent reaches the form error method a second time, but the error was an interactive *critical* error, the form's built-in **error** method displays the standard Paradox error dialog form.

Overriding default behavior

Although you can't write ObjectPAL code to replace the default interactive warning error handler, you can intercept the StatusEvent, and place custom code on the **status** method of any object or on the form. You could, for instance, customize the resulting warning messages, or log them. Remember that the StatusEvent goes first to the form's built-in **status** method, which dispatches it to the **status** method of the active object (the object that caused the error). By default, a StatusEvent bubbles up through the containership hierarchy until it reaches the form again. See Chapter 10 for more information about StatusEvents.

Warning errors and try...onFail

By default, a try...onFail block does not trap warning errors. The Session type method **ErrorTrapOnWarnings**, which takes as its argument a Logical value of True or False, can tell ObjectPAL to trap warning errors in a try...onFail block, in the same way that it does

critical errors. Default operation is the same then as for critical errors, except that the end result is a message in the status bar instead of the standard Paradox error dialog box.

You can turn **ErrorTrapOnWarnings** on or off at any time during a session as often as necessary. For example, you might want to handle certain warning errors within try...onFail blocks and others within if...then blocks. To do so, include one of the following statements in the error-handling routine as appropriate.

```
ErrorTrapOnWarnings(Yes)
ErrorTrapOnWarnings(No)
```

Where should code be attached?

The ObjectPAL event model provides tremendous flexibility when working with the error stack and the built-in **error** method. However, there are several issues to consider when you're deciding where to attach your error-handling code. These issues can involve trade-offs, as discussed in the following paragraphs.

Remember that all events—including ErrorEvents—go first to the form. This means that you can achieve centralized error handling by attaching code to the form's built-in **error** method. This makes sense in some applications, but possibly at some cost:

Complexity You might not want to handle every error at the form level for reasons of *complexity*. Handling errors for every object, of every type, on a form of any size can become quite complicated. Suppose, for example, that you have a form containing a number of objects: buttons, boxes, text boxes, bitmaps, and one table frame. In this form, only one object, the table frame, could possibly generate a key violation error. In this case, it might make more sense to localize the error handling, and then attach code to the **error** method for the table frame as shown in the next example.

```
method error (var eventInfo ErrorEvent)
   if errorCode() = peKeyViolation then
      handleIt()   ; call your custom error handler
   endIf
endMethod
```

Portability Another issue to consider about global error handling at the form level is *portability*. It is much easier to cut and paste objects and groups of objects between forms and applications when error handling is localized to the objects themselves.

A useful technique to keep in mind is *convenient grouping*. For example, in the case of a large group of field objects, you can attach code to the built-in **error** method of each field object, which designates an error-handling routine for each field object. Alternatively, you can place these fields in a container (such as a box), and attach the custom **error** method to the box. This way, an error in any of the field objects is handled by the box's **error** method.

When you're deciding where to handle errors, no general rule is valid for all situations. You can use the techniques discussed in this section in any combination and attach code to the form, to the container boxes, and even to one or more fields and other objects.

Ensuring data security and integrity

Passwords help you guarantee the security of your data, and Paradox's automatic locking features help you provide a balance between the goals of data integrity and concurrent access in interactive use. This chapter covers

- Controlling access with passwords
- Managing locks

Controlling access with passwords

Multiuser applications often require that different users be given different degrees of access to data. For example, some users may be allowed to examine but not change data, others may be allowed to change existing records but not delete them or enter new ones, while still others may be allowed to make any modifications they want.

By default, tables are unprotected. Paradox provides five levels of password security as shown in Table 31.1.

Table 31.1 Password levels and their descriptions

Level	Description
Read Only	User can read from the table, but not change it
Modify	User can enter or change data
Insert	User can add new records
InsDel	User can add and delete records
All	User can perform all operations

Password security is generally administered using Paradox interactively. See the *User's Guide* for more information. In addition, ObjectPAL provides the administrative functions described here.

Protecting the table

Before any password-based access control can be used, the table must be protected in one of the following ways:

- Interactively, when either creating or restructuring the table
- From ObjectPAL, when the table is created
- From ObjectPAL, by using **protect** on an existing table

The Table type method **protect** encrypts and assigns an owner password to the table, granting the owner full ("All") access rights to the table, including the right to restructure the table. By encrypting the table, **protect** ensures that users cannot access tables with an outside debugger or text editor. The Table type method **unprotect** reverses this operation.

Assigning access levels

Once the table is protected, levels of password protection can be assigned to the table in one of the following ways:

- Interactively, when either creating or restructuring the table
- From ObjectPAL, when the table is created

Note that ObjectPAL provides no way to modify password restriction on an existing table; this is intended to be an interactive system administration function.

Determining access levels

Once password security is in place, your application may need to determine its access rights at run time. Use the Table type methods **tableRights** and **familyRights**, which return table access information.

Gaining access to protected objects

Once a table is protected and passwords have been assigned, access to the table is obtained in the ObjectPAL environment by using the Session type method **addPassword**. **addPassword**, named for historical reasons, essentially marks the table as accessible to this session at whatever level the previously arranged password permits.

In a typical application, user access to protected tables is accomplished by first prompting the user for an application-level password, and then calling **addPassword** behind the scenes, to gain access to the protected table.

The Session type method **removePassword** reverses this action, removing access to the protected table from this session.

Understanding sessions

Password-based access control works on a per-session basis, providing lots of flexibility for the application developer. Recall that Paradox creates one session automatically

when the application is invoked, and that ObjectPAL provides means by which additional sessions can be created at run time.

For example, suppose the following code is attached to the **pushButton** method of a button in a blank form. It uses a Table variable to protect the *Customer* table and assign a password, "foo." Then, it tries to open a TCursor onto the Table variable without first presenting the password. The attempt fails because the table is now password-protected.

Next, a call to **addPassword** presents the correct password to gain access to the table; that is, it adds "foo" to the list of passwords known to this session. Now, attempts to open the table succeed. The call to **removePassword** removes "foo" from the list of passwords known to this session, so the next time you push this button, you'll get the same results: failure, then success.

```
method pushButton(var eventInfo Event)
   var
      custTbl Table
      custTC  TCursor
   endVar

   custTbl.attach("customer.db")
   custTbl.protect("foo")
   message(custTC.open(custTbl)) ; displays False

   addPassword("foo")
   message(custTC.open(custTbl)) ; displays True

   custTC.close()
   custTbl.unprotect("foo")
   removePassword("foo")
endMethod
```

Managing locks

Paradox's automatic locking features strike a balance between the goals of data integrity and concurrent access in interactive use. They can help you provide the same balance in your applications, and by using these features, you can write a functional multiuser application without ever explicitly locking either records or objects.

Typically, though, you will want to explicitly lock shared resources so your application can take the appropriate actions if a lock fails. With that in mind, this section explains how locks work in Paradox.

How locks work

Locks limit access to a table or a record. Locks can be placed either automatically by Paradox when a particular operation is invoked, or explicitly in a ObjectPAL method.

You can contrast locks to password control in this way: passwords limit access at the outset of operations, before tables are even opened. Passwords restrict access based on user access permissions. Locks, on the other hand, operate on an immediate, real-time

basis, and only after resource password access has been accomplished. With locking, the programmer gains temporary control of data resources on a need and usage basis, rather than on a password access basis. Locking should be used to bracket only the smallest data acquisition and modification code segments, whereas password control can bracket whole sections of an application if not the entire application.

This section includes the following topics:

- Working with automatic locks
- Using explicit locks
- Types of table locks
- Using lock and unlock
- Locking records

Working with automatic locks

In a multiuser environment, Paradox automatically locks objects during operations where there could be contention for a resource. In general, locks placed automatically by Paradox are the weakest possible consistent with the need to maintain data integrity for the duration of the operation.

For example, when you use **copy** to copy a table, Paradox places a write lock on the source table. The write lock prevents other users from modifying the source table during the copy operation, but does not prevent read-only operations such as viewing. Paradox also places a full lock on the destination table involved in the copy; this lock prevents other users from accessing the destination table in any way during the copy operation.

Locking nonexistent resources

In this instance, the destination table often does not exist at the start of the operation. Paradox can lock it nevertheless. This ability to lock a nonexistent resource is essential, because it prevents another user from creating an object during the copy operation, or from deleting a table out from under you between the time you create it and the time you first use it.

Other operations that involve automatic locks are most batch-style operations, such as **add**, **subtract**, **cSum**, **cNpv**, and so on.

Using explicit locks

Even though Paradox provides automatic locking for many operations, in most multiuser applications you should use explicit locking commands to control access to resources, rather than depend on the automatic locks. The explicit lock commands give more control than the automatic locks and also make it easier to handle situations where a user cannot place a lock because of contention for a resource with other users.

You can have explicit and automatic locks active at the same time on the same table. For example, if you use the Session type method **lock** to explicitly place a full lock on the *Orders* table, and then use the Table type method **copy** to copy *Orders* to *Newords*, you

will have placed both an explicit full lock and an automatic write lock on *Orders*. From the perspective of other users, *Orders* will appear to have only a full lock on it, since that is the stronger of the two locks. At the end of the operation, the automatic write lock disappears, leaving your explicit full lock intact.

Other users can also place locks, both explicit and automatic, on objects that you have locked, as long as these locks can legally coexist with existing locks. Attempts to place object and record locks, both automatic and explicit, are honored on a first-come, first-served basis.

Types of table locks

The types of table locks available in Paradox vary by strength and degree of concurrency. These types are explained in detail in later sections. Table 31.2 shows the locks you can place, in order of *decreasing* strength and *increasing* concurrency.

Table 31.2 Types of locks

Lock type	Description
Full lock	No other session can read, write, or restructure the locked table
Write lock	No other session can place a write lock or a read lock on this table
Read lock	No other session can place a write or full lock on this table

You can place *full* locks on Table variables, TCursor variables, or on tables referred to by quoted names. In contrast, you can place *write* locks and *read* locks only on TCursor variables. In addition, there are some specific differences between Paradox and dBASE tables, as shown in Table 31.3.

Table 31.3 Available locks for Paradox and dBASE tables and TCursors[1]

	TCursor var	Table var	Quoted table name
Read	P	P*[2]	P*
Write	P D	P*D*	P*D*
Full	P D	P D*	P D*

1. P = Paradox table; D = dBASE table

2. * = if table exists

Full locks and write locks limit the operations other users can perform on the table. They are appropriate when a table must remain stable over a given period. For example, place a write lock on a table when you need to perform an operation that cannot tolerate changes to the contents of a table by other users, such as modifying the contents of some records.

Use a full lock when you need to reserve exclusive use of a table. For example, use a full lock when an application needs to

- Work on secure information in a temporary table
- Reserve a table name, but the table has not been created

- Perform one or more changes to the structure of a table, such as creates, deletes, sorts, or restructures
- Run several informational methods such as **cMax** and **cNpv** on the table and ensure that the contents don't change

Note that many of these operations automatically place these locks.

Using lock and unlock

Use **lock** and **unlock**, defined for both the Table and the TCursor types, to place explicit locks on one or more tables. Both **lock** and **unlock** take as arguments a comma-separated list of one or more pairs of strings representing table names and lock types. The quoted strings FULL, READ and WRITE are used to request full, read, and write locks, respectively.

The following statements place a full lock on the Orders table and a write lock on the Stock table:

```
var
   ordersTbl Table
   stockTC TCursor
endVar

ordersTbl.attach("orders.db")
stockTC.open("stock.db")
lock(ordersTbl,"FULL",stockTC,"WRITE")
```

When the **lock** method requests locks on a list of tables in a single statement, **lock** either places all the locks or none of them. This feature lets you avoid certain types of table acquisition deadlocks. **lock** returns True when it successfully places all the locks and False if it cannot. Use the error stack System procedures **errorCode**, **errorMessage**, and **errorShow** to determine why the operation failed in the later case.

A good example of an application where records in different tables must be locked simultaneously is an order-processing system. For a given customer, you may need to update inventory and billing tables as part of the same transaction. To keep these tables consistent with one another, you must lock the appropriate record in each table before making any changes to either of them.

Scope of locks

Locks are *not* released when the method that placed them ends. Full locks persist until explicitly unlocked. Write locks and read locks persist according to the scope of the variable that is locked.

Read locks guarantee access to a consistent data set. Although other sessions may view the data, this lock prevents write and full locks from being placed on the table by other sessions. By placing a read lock on the table you want to edit, you ensure that other users cannot change the data out from under you while you are using it. Of course, a read lock would not prevent another user from locking a particular record that you want to edit.

Placing a full lock

A full lock prevents other users from accessing the table in any way—it gives you sole access. If you place a full lock on a table, no other user can choose that table from a Paradox menu or access it from an ObjectPAL method. If someone else has opened or locked the table, this lock will be denied. A full lock says, in effect, "No other user in any other session can read or write to this table."

Note Placing a full lock on a table is *not* the same as calling the **setExclusive** method defined for the Table type. Certain operations (for example, restructuring) cannot be performed unless you call **setExclusive**; placing a full lock does not give sufficient rights.

The full lock corresponds to the full lock provided with earlier versions of Paradox. You cannot place a full lock on a dBASE table.

A full lock can be placed on a Table variable, a TCursor variable, or a quoted table name. This lock persists until you unlock it, or until you end the session in which it was set. The following example uses **lock** with a Table variable and a table name to place full locks on the *Sales* table and the *Orders* table.

```
var salesTbl Table endVar
salesTbl.attach("sales.db")
if not lock(salesTbl, "FULL", "orders.db", "FULL") then
   ; strategy to deal with locked resources...
endIf
```

Placing a write lock

A write lock prevents other users from changing the contents of a table and from placing a write lock or a full lock on a table. It does not limit the ability to view the table, nor does it limit access to objects in the family of the table. If someone else has placed a full lock, write lock, or read lock on the table, this write lock will be denied.

A write lock says, in effect, "I'm writing to this table; no one else is allowed to do this; I need exclusive write access to maintain data integrity."

Placing a read lock

A read lock prevents other users from placing a write lock or a full lock on a table, either automatically or explicitly, but does not prevent other sessions from also placing read locks on the table. If there is already a write lock on the table, this read lock request will fail. A read lock does not prevent other sessions from placing a read lock. For dBASE tables, Paradox upgrades read locks to write locks. In Paradox, this read lock corresponds to a prevent write lock in earlier versions of Paradox.

A read lock says, in effect, "I'm working in this table. Don't let anyone else place a write lock on it, or in any way change this data."

You must use a TCursor to place a read lock. The following example places a read lock on the *Orders* table.

```
var ordersTC TCursor endVar
lock(ordersTC, "READ")
```

Avoiding deadlock

There are situations in which the contention for resources in a multiuser environment can lead to deadlock. For instance, suppose two users, Ken and Dick, both need to lock *Orders* and *Stock*. Ken has successfully locked *Orders* and is attempting to lock *Stock*; meanwhile, Dick has locked *Stock* and is attempting to lock *Orders*. Each of the two users is waiting on a resource held by the other producing what is called a *deadlock*.

Because **lock** enables you to lock more than one table at once, it has built-in deadlock prevention. If Ken executes the following code

```
var
   Orders, Stock Table
endVar
Orders.attach("orders.db")
Stock.attach("stock.db")
lock(Orders, "FULL", Stock, "FULL")
```

at exactly the same time that Dick executes this code

```
var
   Stock, Orders Table
endVar
Stock.attach("stock.db")
Orders.attach("orders.db")
lock(Stock, "FULL", Orders, "FULL")
```

one of the two will secure a lock on both tables, while the other will have to wait.

To avoid deadlock, try to lock all the resources you need for an operation with a single **lock** statement. If you structure your applications by using small modules and lock all the tables you'll need at once, you won't need to worry about deadlock.

Placing multiple locks on one table

You can explicitly place two or more different types of locks on the same table concurrently. For example, the following code attempts to place a write lock and a read lock on the *Orders* table.

```
var ordersTC TCursor endVar
ordersTC.open("orders.db")
lock(OrdersTC, "WRITE", OrdersTC, "READ")
```

When you apply multiple locks to a single table, you must pair each explicit lock with an explicit unlock of the same type. Locks can be stacked, and an object isn't unlocked until you remove as many locks as you placed.

Placing two or more of the same type of lock on a single table does not alter the effect of the lock. However, applying multiple locks of the same type can sometimes simplify the flow of control in programs. For example, you can nest methods or procedures, each of which perform a **lock** followed by an **unlock** without undoing the locks already secured by the calling method or procedure.

Testing for a successful lock

Always test the return value of **lock**, or use **errorCode** after attempting to place explicit locks, unless you are certain the attempt will not fail. It is often convenient to place the lock and the related test or **errorCode** in a **while** loop, as shown in the following example.

```
var
   emp TCursor
endVar
emp.open("employee.db")
while True
   if lock(emp, "FULL") then  ; lock succeeded?
      quitLoop                ; then continue beyond loop
   else
      msgInfo(errorCode(), errorMessage()) ; show user the error message
   endIf
   ...                        ; rest of method
endWhile
```

What you put into the **else** clause in this code depends upon the application. For example, it might be appropriate to ask users to wait for the table to be free. If you have structured the application so it will be tied up for only a very short period of time, you may just want to display a `Waiting...` message, pause for a second using the System procedure **sleep**, then loop back to the top of the **while** clause. Don't loop without pausing first, as this will load the network unnecessarily. You could also use **setRetryPeriod** (Session type, in Chapter 32) to build an automatic retry period into the attempt to get a lock in that session.

As soon as an explicit lock is no longer needed, use **unlock** to release it. For example, suppose the *Orders* and *Stock* tables were locked by the following statements.

```
var
   stockTC TCursor
   ordersTbl Table
endVar
stockTC.open("stock.db")
ordersTbl.attach("orders.db")
lock(ordersTbl, "FULL", stockTC, "WRITE")
```

To undo the locks, call **unlock** and specify the tables and lock types to unlock.

```
var
   stockTC TCursor
   ordersTbl Table
endVar
unlock(ordersTbl, "FULL", stockTC, "WRITE")
```

Using lockStatus

Use **lockStatus** to find out if you have explicitly placed a certain type of lock on a table, and if so, how many times. For example, the following statement returns the number of write locks that have been placed on *Orders*:

```
lockStatus("Orders.db", "WRITE")
```

The string ANY as the second argument lets you determine how many locks of whatever type you have placed on the table.

lockStatus reports only your own locks—not those of other users.

Locking records

In contrast to table locks, placing a record lock prevents other users from modifying or deleting the locked data on a record basis rather than on a table basis, for the duration of the lock. Record locks are similar to table write locks, in that other sessions may continue to view the locked record, but may not modify it or acquire a second record lock on it. When another user moves to a record you have locked, the record has the same value that it had when the lock was placed. Only after the record is unlocked are your changes posted to the table and thereafter available to other users.

From the standpoint of an application, record locking serves two purposes:

- It provides you with the exclusive ability to make changes to a given record without locking the entire table. This has potential performance implications.
- When a record is updated, Paradox refreshes values to reflect these changes in other sessions on the network.

lockRecord and unlockRecord

Like table locks, record locks can be placed either automatically, by performing an action that modifies a record, or explicitly, by using **lockRecord**. Similarly, record locks can be released either automatically by moving the cursor to a different record, or explicitly by using **unlockRecord**. For precise control and performance, explicit locking and unlocking is recommended.

To understand how **lockRecord** and **unlockRecord** work, remember that there are two basic ways to edit a record:

- You can change a record that already exists.
- You can enter a new record that has not yet been posted to the table.

Locking existing records

To examine or modify an existing record under program control in a multiuser environment, make the record current and then execute the **lockRecord** method off of a TCursor or UIObject.

You will generally want to test the return value of **lockRecord** to see if the lock was placed. If the lock fails, use **errorCode** and **errorMessage** to find out why it failed. The most likely reason is that there is already another lock on the record, in which case the error stack messages will indicate what session has placed the lock. Other reasons for failure are that another user has placed a write lock on the table containing that record, or that the record has been deleted and therefore no longer exists.

If the lock attempt is successful, the application can proceed to examine or change the record. When processing the record is complete, use **unlockRecord** to unlock the record and post any changes to the record.

Like **lockRecord**, **unlockRecord** returns True when it succeeds and False when it fails. Failure would be caused by a key field violation.

Entering and posting new records

When you create a new blank record and enter data into it, the new record and its data does not actually exist in the table until it is posted. Also keep in mind that a new record which has not yet been posted cannot be locked. Indeed, if a new record has not been posted, there is no need to lock it because other users can't access it. When you are ready to post the record to the table, use **postRecord** or move the insertion point off the record.

Handling other multiuser issues

This chapter explains how to design applications that work well in a multiuser (or network) environment. Although multiuser applications are not much different than applications run on a local drive, they do require some additional planning and an awareness of what other users may be doing with the tables, forms, and other files involved in the application.

This chapter describes the major issues and explains how to

- Organize multiuser network applications
- Use private directories
- Implement interprocess communication
- Work with (and program) aspects of multiuser applications

Chapter 31 discusses using passwords and locks to control access.

Organizing multiuser network applications

Inherent in all multiuser applications is a basic trade-off between the need for data integrity and the need to let many users access the same data concurrently. You could ensure data integrity by allowing only one user at a time to use a table, but that would be inconvenient to others. At the opposite extreme, you could allow anyone on the network to use any table at any time, but such a free-for-all would quickly lead to inconsistencies in the data. The goal in all multiuser applications is to ensure data integrity while maximizing concurrent access.

A key issue therefore in the design of multiuser applications is how to resolve simultaneous requests for the same resources. One guiding principle is to minimize and efficiently control competition for resources. Here are three general rules:

- Keep critical sections short: don't tie up shared resources longer than needed.

- Apply the weakest locks possible: for example, don't lock up an entire table when locking a record suffices.
- Place locks early in a method or procedure: for example, don't execute excess code to set things up only to find that you can't access the table.

Planning a multiuser application

First, when setting up a multiuser application, decide where the resources used by the application will reside on the network. For instance, will all tables reside on a central server or be distributed across several servers?

Second, determine whether your application will need multiple levels of data access restrictions. Are there different classes of users, each with different needs and security requirements? What data access privileges (read/update/none) are appropriate to each class? Organize data access in a way that logically partitions data access by need, and then create an appropriate password protection scheme based on that.

Third, for each table used by the application, what are its locking requirements? Follow the guiding principle stated earlier.

Next, consider system administration issues. Users will require the appropriate operating system access rights to directories and files used by the application. How these rights are granted will vary depending on the particular network your application runs on. Decide how new users of your application will be given proper access, and how users will have access rights removed when no longer appropriate.

Specifically, users must have read/write/create/delete privileges for the directories where tables are stored, because to lock a table, Paradox creates a special lock file (with a .LCK extension) in the same directory where the table is located.

To resolve these issues, the primary information-sharing tools at the programmer's disposal are:

- Private directories, which prevent interference among temporary tables and the like
- Interprocess communication, needed by some applications
- Password protection, which provides precise access control
- Locking, which facilitates data integrity

The following sections discuss the first two tools. Password protection and locking are discussed in Chapter 31.

Using private directories

In addition to sharing objects such as files and directories, the users of your application will generally need to use one or more private objects, typically files. Each user who starts Paradox gets a private directory automatically. In addition, each Paradox session can specify its own private directory. In each case, Paradox refers to this directory with the alias :PRIV:.

Paradox automatically stores the *Answer* table and other temporary tables it generates for a particular user in that user's private directory. This prevents users from overwriting each other's temporary files.

Do not build any assumptions about the name of this private directory into your application; you can expect that different users will have private directories in different places with different path names. To access a table in a user's private directory, always use the alias :PRIV:.

Use the private directory to store temporary or dummy tables used in your application. For example, the following code opens a window of the *Answer* table in the user's private directory and creates an additional scratch copy:

```
var
   ansTV TableView
   tmpTbl Table
   ansTbl Table
endVar
   ansTbl.attach(":PRIV:Answer.db")
   tmpTbl.attach(":PRIV:TMPAnsw.db")
   tmpTbl.empty()
   ansTbl.add(tmpTbl)
   ansTbl.unattach()
   tmpTbl.unattach()

   ansTV.open(":PRIV:Answer.db")
```

For more information about using aliases in ObjectPAL code, see Chapters 19 and 22. For information about using aliases interactively, see the *User's Guide*.

Using interprocess communication

In some multiuser applications, the developer needs to coordinate activity between processes or applications on the network. In a typical client-server architecture, for example, several *client* processes can post updates simultaneously to a central master table. The first client activated will typically initialize the entire environment for itself and for all future clients. As each client comes on line, it needs to determine whether or not the initialization sequence has already been completed, or if another client process is currently running the initialization sequence.

To solve this type of problem, you can use Paradox's ability to guarantee the locking integrity of tables as the basis for a solution. For example, you can

- Use the contents of a shared table to pass information between processes.
- Use the locks placed on a shared table as flags or semaphores to multi-process activity.

To employ such a strategy, set up a special table that will function internally as an information pipe between instances of your application. Transfer information between processes in any appropriate manner. For example, one type of process might write information into a table, while one or more other types of processes might be the

readers, using locks on this table, as well as perhaps a separate semaphore table, for synchronization.

As another example, your application can determine how many client processes are currently active if each client writes a record of its existence to a central table upon startup, deleting its record as it completes. Other processes can determine the total number of active clients at any time by using **cCount** to get the number of records in the table.

Although no ObjectPAL mechanism supports explicit parameter passing between processes, these techniques achieve the same effect.

Database programming aspects of multiuser applications

This section describes a variety of database programming concepts that are especially important when developing multiuser applications. If you understand these concepts, you'll be able to govern the behavior of your application and give your users exactly what they need.

This section describes

- How to handle key conflicts
- Record flyaway and relative ordering issues
- The effects of **postRecord** and **unlockRecord** on locks
- The effects of table locks on the data model
- Strategies Paradox takes when unlocking, committing, or canceling a record
- How to use **setRetryPeriod** to build automatic retries
- The difference between **setExclusive** and **setReadOnly**

Note While these concepts are especially important when developing multiuser applications, they are also useful for developing single user applications.

Handling key conflicts

A key conflict arises when you attempt to

- Post a new record with the same key as an existing one
- Post a modification to an existing record that would give it the same key as an existing record

In either case, the post operation fails. At this point Paradox retains the lock on the existing record with the conflicting key, and the error stack may be examined to determine the reason for failure. Two tools are available to aid in resolving key violations:

- The TCursor method **updateRecord** modifies the existing record after a key violation by updating it with values from the new record, effectively replacing the old record with the new one. Whether the record that was to have been posted was a new record

or an existing record, it is deleted. This method is valid for Paradox tables only. This method takes an optional Logical argument to indicate whether or not to **moveTo** (follow) the record on flyaway.

- The TCursor method **attachToKeyViol** takes as its argument the TCursor that reported the key violation. This is a quick way to access the existing record with the intended key by setting a second TCursor to point to it.

Flyaway and relative ordering issues

If you've added records to keyed tables, you may have noticed how your new records "jumped" to their correct positions when you posted them. This movement is called *flyaway* and can be confusing when you begin working with keyed tables interactively. Flyaway can cause subtle changes in behavior, especially in multiuser environments (where other users may be adding or changing records and key values in the middle of an operation your application is working through).

Flyaway is normal and occurs when the order of a table's records changes (usually when a record is added or the index is modified). Like other aspects of multiuser application development, it requires a little planning to work with effectively.

To help you, Paradox provides a read-only property called FlyAway for fields, records, table frames, multi-record objects, and forms. FlyAway is True if the most recent record unlock (or record post) caused the record to move from its current position; if the record stayed where it was, FlyAway is False. This property is only valid (and useful) immediately after a DataUnlockRecord or a DataPostRecord action, since that's when the property is set. Its state remains unchanged until the next such action.

Important Both DataUnlockRecord and DataPostRecord attempt to write the current record to the underlying table, but with the following difference:

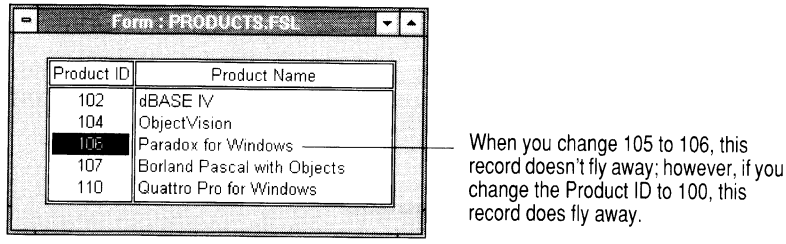
- DataUnlockRecord stays in the current general area of the records, allowing the record to fly away if it belongs in a new location (and then unlocks it).
- DataPostRecord flies with the record to its new location and keeps it locked.

The following example shows code attached to the built-in action method of a table frame (it would also work for a multirecord object). It tests for a DataUnlockRecord and forces a DataPostRecord, invoking the built-in code that makes the record fly away. The table frame follows the record as it flies away.

```
method action(var eventInfo ActionEvent)
   if eventInfo.id() = DataUnlockRecord then
      self.action(DataPostRecord)
   endIf
endMethod
```

As an example of how fly away works, suppose a form contains a table frame (keyed on the Product ID field) as shown in Figure 32.1. If you select record 3 and change its key value to 106, the FlyAway property returns False, because the records stays in the same position in the table. If you change the key value to 100, the record moves, and the FlyAway property returns True.

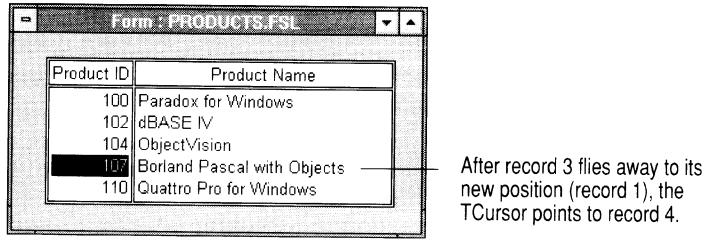
Figure 32.1 Using the FlyAway property



In ObjectPAL, flyaway can cause confusion about where a TCursor is pointing after a record is posted. The general rule is that if the record moves, the TCursor points to the next record, as shown in Figure 32.2. After its key value is changed from 105 to 100, record 3 flies away to position 1. The TCursor must continue to point to a valid record, so it effectively (and implicitly) performs a **nextRecord** and points to record 4.

Important When working with keyed tables, when you call **postRecord**, the TCursor follows the posted record if it flies away. When you call **unlockRecord**, the TCursor does not follow the record.

Figure 32.2 More about the FlyAway property



Flyaway can occur in scan loops

Pay particular attention to flyaway in scan loops where the relative order of records within the table is being changed as you scan. For example, if you scan through a table and operations sometimes change the relative order of records, what happens when the record you are on does a flyaway?

```
Proc someProc(var tc TCursor)
   scan tc:
      if someCondition = True then   ; Under some conditions,
         tc.delete()                 ; delete this record.
      endIf
      tc.nextRecord()                ; Now, what record does tc point to?
   endScan
endProc
```

Ordinarily, when the relative ordering of records changes, you don't know for certain where the TCursor is after the record is unlocked. Two methods are provided to add flexibility in this regard:

- The TCursor method **setFlyAwayControl** has two effects: first, if set to True, it indicates that the TCursor will stay on the record after an **unlockRecord**, even if the record should move out of its relative position in the table, as long as it doesn't move outside of its immediate neighborhood (otherwise, the FlyAway property will be set to False). If FlyAway is True, it implies that the record did move out of relative position. The second effect of this method is to enable the method **didFlyAway**.
- The TCursor method **didFlyAway** can be queried after an **unlockRecord**, to determine whether the TCursor did, in fact, fly away.

These two methods can make scan loops easier to understand, but the trade-off is performance. Because **setFlyAwayControl** entails a lot of internal checking on every cursor operation, it slows things down and should be used only with caution.

Effect of postRecord on locks

When working with keyed tables, TCursor method **postRecord** posts changes to the record without unlocking it and keeps the record as the current record. In contrast, **unlockRecord** unlocks the record and makes it fly away to its new sorted position in the table, if necessary.

When developing multiuser applications, use the method appropriate for the action you want to occur when records are committed. Although **postRecord** is the one you'll use most often, there are times (and tasks) where **unlockRecord** is more appropriate.

Table locks on the data model

Whenever you lock a record, whether interactively or with ObjectPAL, Paradox will automatically lock all related tables in the data model in order to maintain referential integrity. Paradox starts locking from the topmost master in the data model and walks its way down to the table you intend to lock, locking every immediate master. This is necessary to prevent some other session from altering the master and causing details to fly away (locked!). This may be confusing to the uninitiated, because it will appear to another session that it cannot lock a master table when the first session has only acquired a lock on a detail record.

Sibling tables which are not themselves masters will be unaffected. This means that if there are two one-to-many tables off of one master, locking one detail set will not affect the other.

Unlock, commit, and cancel

When a record is unlocked, posted, or canceled, Paradox will take one of three strategies:

- If this is from a keystroke or menu choice (that is, interactive), the entire data model will be affected, as described in "Managing locks" in Chapter 31.
- If this is an internal event or the result of an ObjectPAL **action** method, it will affect only the table associated with the target of the event.

- An unlock/post/cancel record action sent directly to the form object by ObjectPAL will affect the entire data model, as in the interactive case.

The consequence of this is that ObjectPAL has the ability to unlock/post/cancel any specific table in the data model without affecting the others, whereas the interactive user automatically affects the entire data model.

Note to dBASE developers

It is not necessary for Paradox to lock a record explicitly before deleting it. (Of course, if the record is locked from another machine or session, the delete will fail as expected.) As a result, a second machine may not modify an unlocked detail record if its master was locked, for reasons stated above, even though it can delete any unlocked detail record. Also note that if you lock a record and then delete it, the delete operation removes the lock. (And if ShowDeleted is True, the record will remain visible though unlocked.)

Building automatic retries with setRetryPeriod

As previously mentioned, test the return value of **lock** after each attempt to place a lock unless you are sure the lock will succeed. In some cases, you might know from the flow of control in the application that a lock will eventually succeed if you try to place it repeatedly over a long enough interval of time. For example, in applications designed to run in a batch mode overnight, it may be acceptable to wait several minutes for a table to become available. In such cases, you can often simplify the flow of control by using **setRetryPeriod** (Session type).

This method builds an automatic retry period into every operation, implicit and explicit, that could fail because of a resource conflict. It is much simpler than adding retry code to your own methods. After the following statement, any operation that could fail because of a locked resource is automatically retried for 100 seconds:

```
setRetryPeriod(100)
```

Note Use **setRetryPeriod** with caution. Your application must still handle the possibility that the retry period will elapse without success. Also, because the retry period is not reset when the method ends, be sure to disable the automatic retry feature when you no longer need it by executing

```
setRetryPeriod(0)
```

If your application leaves it set to a large number, a user who later attempts unsuccessfully to lock a table or record will be confused by the delay—indeed, Paradox will appear to have frozen. You can determine the length of the current retry period setting by using **retryPeriod**.

SetExclusive and setReadOnly

Placing a full lock on a table can be contrasted with the Table method **setExclusive**. If you invoke **setExclusive** on a Table variable before opening a TCursor on the table, **setExclusive** will fail if anyone else has the table open; when it succeeds it will prevent anyone else from opening the table thereafter.

A full lock is more powerful, in the sense that it allows changes to be made to the table's characteristics. In contrast, **setExclusive** affects only the tabular data described by the Table variable.

setExclusive is useful for applications that require all or nothing access to an object. The user acquires exclusive use of the file for the length of time the table is open. In contrast, setting a full lock with **lock** is preferable in most cases, because the application can keep a table open a long time (all the time), non-exclusively, turning locks on and off only briefly as needed.

The Table method **setReadOnly**, again called on a Table variable before a TCursor is opened on it, enables you to make the table read-only for the user of your application. The telephone information data system used by telephone information operators is a good example of an application where it is particularly useful for the user to have information access without permission to modify this information.

Automatic refresh

When Paradox detects a change to a table (across the network, or just because some other window or TCursor in your application modifies some data), a "Refresh" is generated. This refresh occurs only when the data currently onscreen has been modified. Modifications to data not onscreen do not generate a refresh.

When this happens, the form initiates a DataRefresh action. The form's default code carries out the necessary processing giving ObjectPAL the ability to pre- and post-process the action. Note that this action happens after the change has occurred so there is little that ObjectPAL can do other than repair any internal calculations.

The read-only property Refresh for fields, records, table frames, multi-record objects, and forms is True between the time Paradox first tells you a refresh is happening and the time you have finished all your processing.

Performance tips

You can use full locks and write locks to improve performance as well as to limit access. For example, if you write lock a shared table, many operations are faster because Paradox does not need to check whether other users have modified it. If you place a full lock on a shared table, operations are even faster because Paradox can buffer in memory the changes you make to a table, rather than having to write each one out as it is made. Thus, write locks and full locks can be beneficial even if you do not expect others to access the table.

Packaging and delivering an application

This chapter presents the concepts and procedures necessary to package Paradox applications for delivery to end users. This process consists of collecting all the files required by the application, including forms, reports, tables, index files, and queries, as appropriate. Some additional issues involve integrating the Windows Help system, localizing an application for international users, and documenting the application code.

Topics covered in this chapter are

- Components of a Paradox application
- Saving and delivering forms
- Adding a Help system
- Localization and character set issues
- Documenting the application

Collecting the application components

The main components of a Paradox application are

- Desktop
- Data model objects
- Forms

Desktop

The Desktop is the framework in which the Paradox application runs. It provides an environment that includes the parent window, menus, Toolbar, a default session, and

password lists. Properties control how a Desktop looks, and also some aspects of its relation to the system. Properties include

Desktop properties

- Application title bar
- Default font
- Colors for various standard control and window components
- Wallpaper
- Database
- Private directory
- Child window size
- Toolbar position
- Ruler style

The Paradox development Desktop may have additional properties such as run/design mode and debugging and editing preferences.

Note For more information about Desktop properties and other settings, see the *User's Guide*. See also the SETTINGS.TXT file in your Paradox system directory.

The Desktop maintains a list of passwords entered and canceled by the user during the current session.

Each time you invoke Paradox, you invoke a new instance of the Desktop. Each instance of the Paradox Desktop can run multiple applications at the same time. You can also run multiple instances of Paradox concurrently on the same Windows platform.

Each instance of Paradox on the same workstation runs independently, unrelated to other instances of itself. Each instance competes against the others for resources and locks as if they were on different workstations.

Note See Chapter 23 for information about using the Session variable to open multiple sessions with ObjectPAL.

Data model objects

The data model object components are Table and Query. Data model objects are not included automatically when you "deliver" your forms, so you must explicitly include them in your product packaging considerations. Each table in the data model is represented by one or more files, with file name extensions as described in Table 33.1.

Table 33.1 File name extensions for files related to tables

Feature	Paradox	dBASE
Table	.DB	.DBF
Primary index	.PX	N/A
Secondary index	.Xnn and .Ynn	.NDX or .MDX

Table 33.1 File name extensions for files related to tables (continued)

Feature	Paradox	dBASE
Validity checks	.VAL	N/A
Memo fields	.MB	.DBT

Forms

In this chapter, the term "form" means "form, report, library, and script" unless specifically noted. Forms are delivered in a "stripped" version by a process called "delivery," discussed in the next section.

Saving and delivering forms

Paradox provides two ways to store forms: saved and delivered. Ordinarily, you *save* forms during applications development, and *deliver* forms as part of packaging the application for delivery and installation to end users. Both of these commands are available only when working in a design window.

Use the Save option while you're developing the application. The objects and code in saved forms can be edited only with the development version of Paradox. Paradox assigns file name extensions to saved form objects as shown in Table 33.2.

Use the deliver option when your form is complete and you're ready to package it for end users. The delivered form contains no source code, and its objects and code can't be edited. Paradox assigns file name extensions in delivered forms as shown in Table 33.2.

When you save or deliver a form, Paradox creates a special Windows file called a DLL (for dynamic link library), which contains one or more compiled objects and compiled ObjectPAL code. When you save a form, ObjectPAL source code is retained; when you deliver a form it is not.

Table 33.2 File name extensions for saved and delivered objects

Object	Saved	Delivered
Form	.FSL	.FDL
Library	.LSL	.LDL
Report	.RSL	.RDL
Script	.SSL	.SDL

Saving forms

To save a form, report, library, or script, choose File | Save (or File | Save As). Name the object, if prompted.

Delivering forms

The steps for delivering objects are similar, but different enough to list separately.

Form

Follow these steps to deliver a form:

1 Open the form in a design window.

2 Choose Program | Deliver.

Report

Follow these steps to deliver a report:

1 Open the report in a design window.

2 Choose Report | Deliver.

Library or script

Follow these steps to deliver a library or a script:

1 Open the library or script in a design window.

2 Open an ObjectPAL Editor window.

3 Choose Program | Deliver.

Adding a Help system

ObjectPAL provides a way for you to call the standard Windows Help application, which is a standalone Windows application. To use Windows Help, first create a file of Help information and context strings, and then compile it using the Microsoft Help compiler (included with the Paradox Developer's Edition or Borland C++). The documentation that comes with the Help compiler explains in detail how to build and compile a Windows Help system.

Once you have compiled your Help system, you can use the following System type methods to access it:

- **helpOnHelp** provides help about using the Help application.
- **helpQuit** tells the Help application that a particular Help file is not being used.
- **helpSetIndex** tells the Help application which index to use.
- **helpShowContext** displays the Help associated with the specified context ID.
- **helpShowIndex** displays the index of a particular Help file.
- **helpShowTopic** displays the Help associated with the specified topic key.
- **helpShowTopicInKeywordTable** displays Help associated with a keyword stored in an alternate keyword table.

Important For examples showing how to access a Windows Help system, refer to the Connections application and the examples included online. Also, refer to the entries for these methods in the online ObjectPAL Help.

Localizing for international users

This section discusses how to modify applications for international users.

Using international formats for numeric constants

Most ObjectPAL expressions take numeric constants in U.S. formats.

In certain non-U.S. countries, numbers and currency can be input and displayed using an international number format. This format can be specified using **formatSetCurrencyDefault** and other System format procedures. Your application uses your Windows International setup by default.

Note System type methods **readProfileString** and **writeProfileString** provide access to the user's WIN.INI file.

Localizing quoted strings

Quoted strings associated with a form are automatically stored as resources—you don't need a separate Windows resource (.RC) file. Because the form is a DLL, you can use Borland's Resource Workshop to change these strings (for example, to translate them) without recompiling and without having to have the source code for the application available.

For example, suppose you want to deliver a form to protect the source code, but you also want to make messages available for translating. You could assign the messages as quoted strings to String constants, as follows:

```
const
   FindFile = "Find Customer"
   NotFound = "Couldn't find "
endConst
```

Then, you could use these constants in statements like the following:

```
msgStop(FindFile, NotFound + custTC.Name)
```

After you deliver this form, users can't change the source code, but they can use a resource editor to change the text of the constants FindFile and NotFound and translate them as appropriate.

You can use the same technique to refer to fields in a table. As shown in the following example, when you put a field name in quotes, the field name gets resourced. Suppose you write code that uses the value of the Street field of the *Customer* table. Suppose that this application is used in France, and all the field names are translated to French. The Street field would be named Rue. If your code used quoted strings as field names, you could use a resource editor to translate the references, and the application would run.

```
proc someProc
Var tc TCursor endVar
   tc.open("Customer.db")
   x = tc."Street"   ; "Street" is resourced, and can be edited
```

```
    y = tc.Street      ; Street is not resourced, and cannot be edited
endProc
```

Localizing forms

Labels and text on the form itself are not resourced; that is, they are not saved as resources so they can't be edited with applications like Borland's Resource Workshop. Because of this, it can be difficult to translate your forms (and the text that appears on them) to a different language.

To remedy this problem, you can use ObjectPAL to set the text of labels and text objects during initialization rather than hard-coding the text in the object. Place the code in each object's open method, using double-quoted strings so that the strings get resourced, as in the preceding section.

About character sets

Paradox recognizes two kinds of character sets: OEM and ANSI. An OEM set (also called a *code page*) consists of 256 characters, numbered from 0 to 255. Characters 0–127 are the same in each code page. Characters 128–255, called extended characters, are different for every code page. The extended characters include accented characters and special symbols for various languages. DOS 5.0 comes with several different code pages, but only one can be activated at a time. The pages are English, Multilingual (Latin I), Slavic (Latin II), Portugese, Canadian-French, and Nordic.

Paradox recognizes one set of ANSI characters. Characters 32–126 coincide with those in the OEM character sets. Other characters may be at different locations in OEM code pages or completely missing. For example, the code for the character ñ is 164 in the OEM English set, but 241 in the ANSI set.

Paradox and dBASE store tables with the OEM character set, while Windows (and therefore the Paradox itself) uses ANSI characters. Paradox uses the Language Driver file (described in online Help) to control sorting and other character-based table level operations. Note that the Language Driver does not control date, number, and currency formats.

Problems may arise because OEM code pages and the ANSI character set do not contain all the same characters. Paradox handles the OEM–ANSI conversion when reading and writing data in tables, but in other cases (for example, when writing to a text file, or exchanging information across a DDE link), it's up to you, the developer, to be aware of and manage the process. Table 33.3 lists the methods ObjectPAL provides for this purpose. All of the methods listed belong to the String type.

Table 33.3 Methods for OEM-ANSI conversion

Method	Description
ansiCode	Returns the integer ordinal value of a character according to the ANSI table
chr	Converts an ANSI code to an ANSI character
chrOEM	Converts an ANSI code to a one-character OEM string
oemCode	Returns the integer ordinal value of a character according to the OEM Table

Table 33.3 Methods for OEM-ANSI conversion (continued)

Method	Description
toANSI	Converts a string of OEM characters to ANSI characters
toOEM	Converts a string of ANSI characters to OEM characters

Documenting the application code

ObjectPAL provides several methods and procedures to keep track of the objects and methods in a form. These methods and procedures are defined for the UIObject type and the Form type. All begin with the prefix "enum", short for "enumerate", as in **enumSource**.

The method **enumSource** creates a Paradox table containing the names of all objects that have methods attached, along with the source code for each method. The method **enumSourceToFile** does the same thing but writes the information out to a text file instead of a table.

To create a table listing all the objects in a form, whether methods are attached or not, use **enumUIObjectNames**.

Interactive function

You can also use Paradox interactively to create a report listing the source code and the objects to which it is attached. The ObjectPAL Editor provides a Browse Sources function that creates a quick report of the source code in a form. To use this function, open an ObjectPAL Editor window, then choose Tools | Browse Source. See Chapter 8 for more information about the ObjectPAL Editor.

Part

Appendixes

This part of the manual contains reference material frequently needed by ObjectPAL programmers.

It includes the following appendixes:

- Appendix A, "PAL and ObjectPAL," discusses some of the differences between PAL and ObjectPAL.
- Appendix B, "Properties," lists the properties of each UIObject and the valid values for each property.

PAL and ObjectPAL

This appendix lists differences between ObjectPAL and PAL. Its purpose is to alert PAL programmers to significant differences in content and perspective between these two programming environments. Refer also to the *User's Guide*.

This appendix covers the following topics:

- Programming environment
- Language differences
- Table issues

Programming environment

The move from PAL to ObjectPAL programming requires a shift in your programming paradigm. This section lists some of the principal differences between the programming environments.

Procedural vs. object-based

PAL is a procedural language, so a PAL application can be described as a linear progression of commands and functions. In contrast, ObjectPAL is object-based, and ObjectPAL applications are event-driven. For programmers used to procedural languages, ObjectPAL requires a change in perspective for program design and coding. See Chapter 3 for an overview of objects and events and the object-based programming strategy.

UI-driven programming

In DOS Paradox, a PAL program acts as an automated user. That is, the PAL program emulates a user of DOS Paradox, and in fact, executes the application by driving the

interface as an automated, behind-the-scenes user. The effect of PAL commands on objects within the workspace is normally hidden from the user unless you issue the ECHO command.

In contrast, when you create an ObjectPAL application, you actually *create* the user interface and define its behavior. ObjectPAL offers new ways to manipulate tables without displaying them on the user's screen, so there is no concept corresponding to PAL ECHO.

Event model

The event and action model in ObjectPAL is quite different from the PAL event and (PAL 4.0) trigger mechanisms. You need to understand the event model to understand how ObjectPAL applications work. See Chapter 10 for details about how ObjectPAL handles events.

Scripts and forms

When you build a PAL application, the center of attention is the *script*. The application consists of one or more scripts which contain PAL statements that execute sequentially and define the behavior of the program.

When you build an ObjectPAL application, the center of attention is the *form*. The application consists of placing objects on the form. You can change the default behavior of these objects by attaching ObjectPAL code to them and to the form itself. The combined behavior of the form and its objects then defines the behavior of the program.

PAL and ObjectPAL use the terms *form* and *script* to describe different concepts. PAL places application code in the script and uses forms to display tables and to enforce referential integrity. ObjectPAL attaches application code to objects in the form and provides the script object as a place to put ObjectPAL code that is not attached to forms. PAL includes a script recorder; ObjectPAL scripts are always written, not recorded. Chapter 29 discusses scripts.

Dialog boxes

In Paradox for Windows, a dialog box is a special type of form. It has different characteristics and capabilities than the DOS Paradox dialog box. Chapter 26 discusses forms and dialog boxes.

Libraries

Libraries in ObjectPAL are similar in concept to PAL procedure libraries. Both save memory and ease maintenance by consolidating custom code. ObjectPAL libraries can contain methods, procedures (local to the library itself only), constants, and declared variables.

In ObjectPAL, memory management is handled automatically by the Windows environment.

Desktop, workspace, canvas

In PAL, the *canvas* is used to display output to the user. The terms *canvas* and *workspace* are not used in Paradox for Windows, and the term Desktop has a different meaning. Chapter 20 discusses ObjectPAL methods for working with the Desktop.

Containership hierarchy

The objects in a Paradox for Windows form coexist in what is called the *containership hierarchy*. The containership hierarchy is based on the visual spacial relationships between objects in the form and determines the scope of variables and custom code. See Chapter 6 for more information about containership.

Language

This section describes some principal differences between the PAL and ObjectPAL languages.

Language elements

PAL scripts consist of commands, functions, keywords, and custom procedures. In PAL, built-in capabilities are provided by commands and functions, and the term procedure is used strictly for user-defined functions. Commands include control structures, key and menu equivalents, and commands to manipulate data, memory, and input/output. Functions differ from commands syntactically, and a function always returns a value. Functions typically perform mathematical, statistical, or datatype conversion operations, or return status information. It is up to the programmer to ensure that a function is used in an appropriate context.

The ObjectPAL language consists of build-in methods, run-time library (RTL) methods and procedures, basic language elements, and custom methods and procedures. Methods and procedures usually return a value or an indication of success or failure. Custom methods and procedures are analogous to PAL user-defined functions.

ObjectPAL methods are organized by the type of object they operate on (Form, Table, UIObject, and so on), and the object operated on is part of the syntax of the method call. Procedures are similar but do not include an object name in their invocation.

Each object on a Paradox form has its own set of built-in methods. The ObjectPAL programmer may overwrite these with custom code; this is a common way to alter default behavior. Chapter 4 discusses the ObjectPAL language structure.

Variable scope

PAL variables are global in scope with the exception of variables that are formal parameters, variables that are explicitly declared to be *private*, and variables declared within closed procedures. In addition, PAL programmers use a concept called *dynamic scoping*.

In ObjectPAL, variable scope follows a specific set of constraints explained in Chapter 6. Variable scope is determined by the containership hierarchy.

Declaring variables

In PAL, variables are not declared. In ObjectPAL, declaring variables improves performance and is required in many cases.

Constants and properties

The ObjectPAL IDE provides access to hundreds of system-defined constants and properties that make programming easier and make programs easier to read and maintain. Many PAL functions are effectively replaced by ObjectPAL properties.

User input

The *accept* statement plays an important part in getting input from the user in PAL programming.

In ObjectPAL, the programmer gets user input from by using field objects, by building custom forms and dialog boxes, or by invoking one of several System type dialog procedures, such as **msgInfo**, **message**, and **msgYesNoCancel**. Additionally, some **view** methods accept user input.

Code management

PAL is interpreted, not compiled. PAL procedures can be saved in libraries for improved performance. PAL source code is saved in script files.

ObjectPAL code is compiled and saved in Windows DLLs. When you deliver an ObjectPAL application, the source code is stripped from the DLL.

Error processing

In PAL, the system variable *errorproc* is set to the name of the user-defined error handling procedure.

In contrast, ObjectPAL provides several different error handling facilities: critical errors, warning errors, the built-in **error** method, and the try...onFail block. It also includes an *error stack* and several related RTL error methods.

Other system variables

Two other important PAL system variables, *retval* and *autolib*, are not used in ObjectPAL.

ObjectPAL provides a Logical data type and Library objects.

Default display formats

PAL provides a number of default display formats that differ from those in ObjectPAL. Paradox for Windows uses the Windows defaults. The programmer can override all defaults in either product.

copyToArray and copyFromArray

In PAL, the commands COPYFROMARRAY and COPYTOARRAY copy the table name as the first item of the array, then copy field data to subsequent items.

The ObjectPAL methods **copyFromArray** and **copyToArray** copy field data only. The table name is supplied by the object variable (TCursor or UIObject).

Query variables

In PAL, an empty tilde variable is treated as a blank value. ObjectPAL treats an empty tilde variable as a null value. A null value is processed as if no condition was placed at that position. A null value is processed as if no condition was placed at that position.

Date arithmetic

In PAL, subtracting one date from another yields a number value as the result (Date – Date = Number). In ObjectPAL, Date – Date = Date.

Table issues

This section lists some principal differences that relate specifically to working with data in tables.

Data access and manipulation

In PAL, you must display tables in workspace objects (without ECHO turned on) to access and manipulate them.

In contrast, ObjectPAL provides direct program access to tables. It introduces the TCursor and Table variable types, which enable behind-the-scenes access and manipulation of tables. Chapter 18 discusses working with data in tables.

Referential integrity

DOS Paradox uses the *form* to create multi-table referential integrity.

Paradox for Windows maintains and enforces rules about data integrity across tables completely independent of the form construct. Referential integrity here is administered either through the interactive table utilities, or through explicit ObjectPAL program statements (search in the online ObjectPAL Help for "create").

Coedit

ObjectPAL Edit mode is equivalent to PAL *coedit* mode.

Table family

Paradox for Windows does not include the formal concept of table families from DOS Paradox. The important difference is that Paradox for Windows forms are *not necessarily linked* to one or more tables. Table binding may take place at run time, and forms may exist without any table binding.

When you prepare to ship a Paradox for Windows application, be sure to correctly include the necessary groups of files. (Refer to Chapter 33 for information.) The IDE includes table utilities such as copy and delete.

Field width truncation

If you try to insert data into a field object where the data is larger than the field definition, PAL will truncate the data and proceed. In ObjectPAL, this attempt will cause a run time error

Appendix

B

Properties

This appendix lists the properties and property values for each UIObject. A solid dot (●) marks a read-write property, a hollow dot (○) marks a read-only property. This information is also listed online. To display the list, open an ObjectPAL Editor window, and choose Language | Properties; then select an object name and a property.

Table B.1 UIObject properties

	Band	Bitmap	Box	Button	Cell	Crosstab	EditRegion	Ellipse	Field	Form	Graph	Group	Header	Line	List	Multirecord	OLE	Page	Record	TableFrame	TableView	Text	TVData	TVHeading
Alignment							●		●													●		●
Arrived	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○		○		
AttachedHeader																				○				
AutoAppend									●	●						●			●	●				
AvgCharSize							○		○													○		
BlankRecord									○	○						○			○	○	○		○	
BottomBorder	○	○	○	○	○		○	○	○		○	○	○	○	○	○	○	○	○	○	○	○	○	○
Border										●														
Breakable	●		●			●	●		●			●		●		●			●	●		●		
ButtonType				●																				
ByRows																●								
CalculatedField									●															
Caption										●														
CenterLabel				●																				
CheckedValue				●																				
Class	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○		○	○	○
Color			●		●	●	●	●	●		●		●	●		●		●	●	●	●	●	●	●
Columnar																●								

Table B.1 UIObject properties (continued)

	Band	Bitmap	Box	Button	Cell	Crosstab	EditRegion	Ellipse	Field	Form	Graph	Group	Header	Line	List	Multirecord	OLE	Page	Record	TableFrame	TableView	Text	TVData	TVHeading
ColumnPosition																				●				
ColumnWidth																								
CompleteDisplay							●		●														●	
ContainerName	☽	☽	☽	☽	☽	☽	☽	☽	☽	☽	☽	☽	☽	☽	☽	☽	☽	☽	☽	☽		☽		
ControlMenu										●														
CurrentColumn						●										●				●				
CurrentRecordMarker.Color																					●			
CurrentRecordMarker.LineStyle																					●			
CurrentRecordMarker.Show																					●			
CurrentRow						●										●				●				
CursorColumn									●													●		
CursorLine									●													●		
CursorPos									●													●		
DataSource															●									
DateFormat																							●	
Default									☽														☽	
DefineGroup	●																							
DeleteColumn																				●				
Deleted									☽	☽						☽			☽	☽	☽		☽	
DeleteWhenEmpty									●										●	●				
Design.ContainObjects	●	●	●	●	●	●		●		●	●		●		●	●	●	●		●	●	●		
Design.PinHorizontal	●	●	●	●	●	●	●	●	●		●	●	●	●		●	●	●		●		●		
Design.PinVertical	●	●	●	●	●	●	●	●	●		●	●	●	●		●	●	●		●		●		
Design.Selectable	●	●	●	●	●	●	●	●	●		●	●	●	●	●	●	●	●	●	●	●	●	●	●
Design.SizeToFit		●	●	●	●		●		●			●	●	●		●	●	●		●				
DesignModified										●														
DesignSizing																						●		
DesktopForm										●														
DialogForm										●														
DialogFrame										●														
DisplayType							☽		●															
Editing									☽	☽											☽			
End														●										
FieldName									●												☽		☽	
FieldNo									☽												☽		☽	
FieldRights									☽														☽	
FieldSize									☽														☽	
FieldType									☽														☽	

Table B.1 UIObject properties (continued)

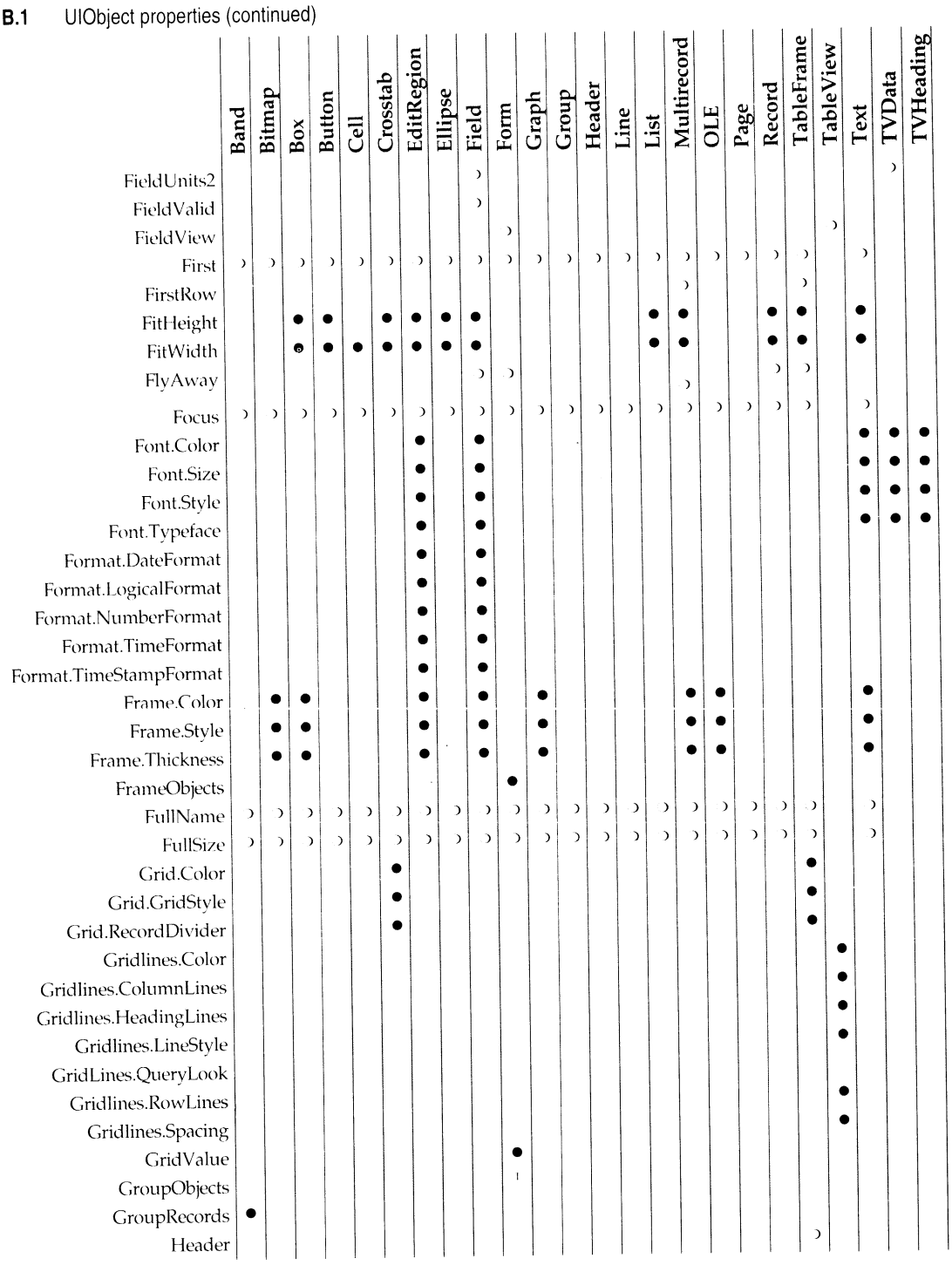


Table B.1 UIObject properties (continued)

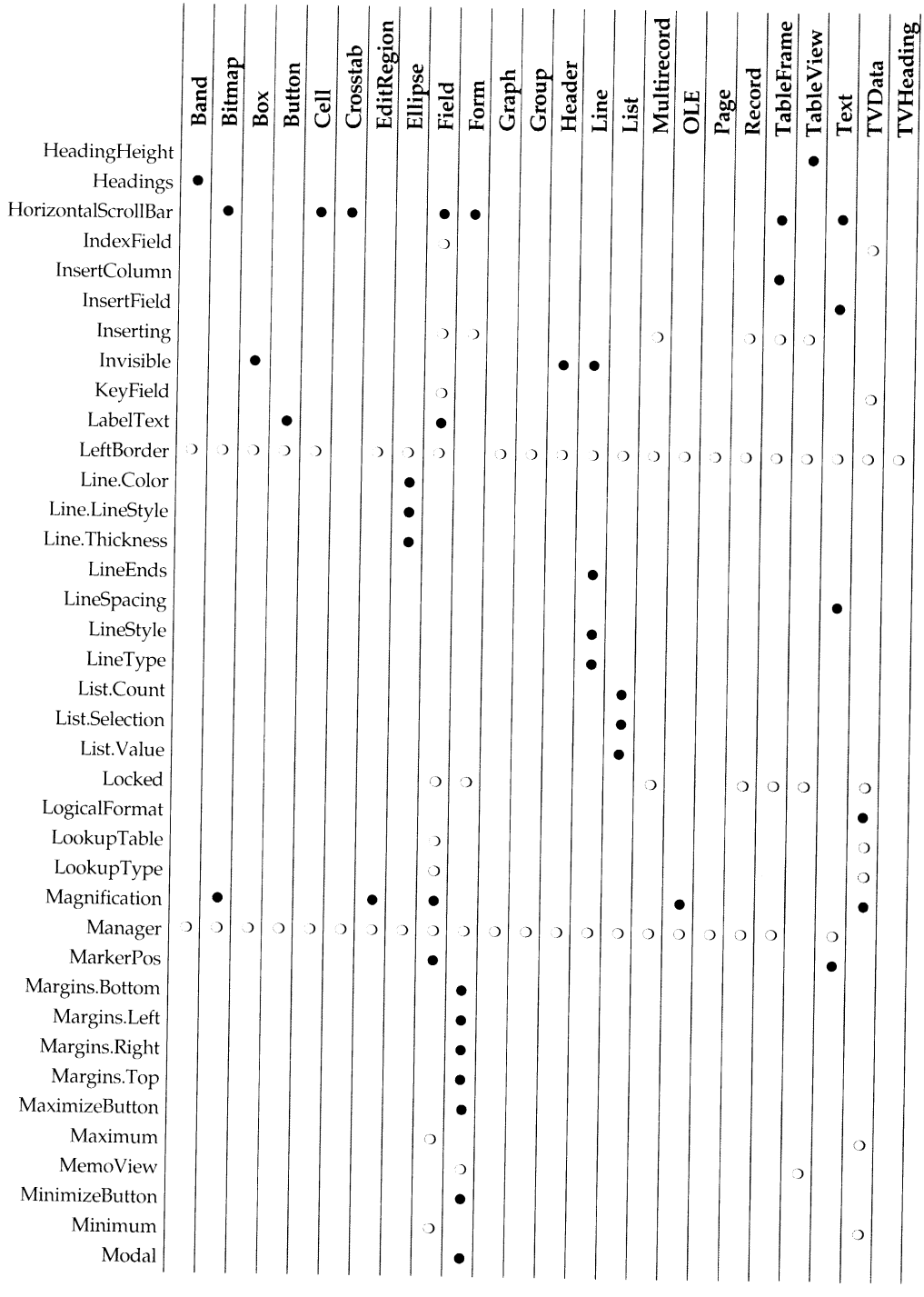


Table B.1 UIObject properties (continued)

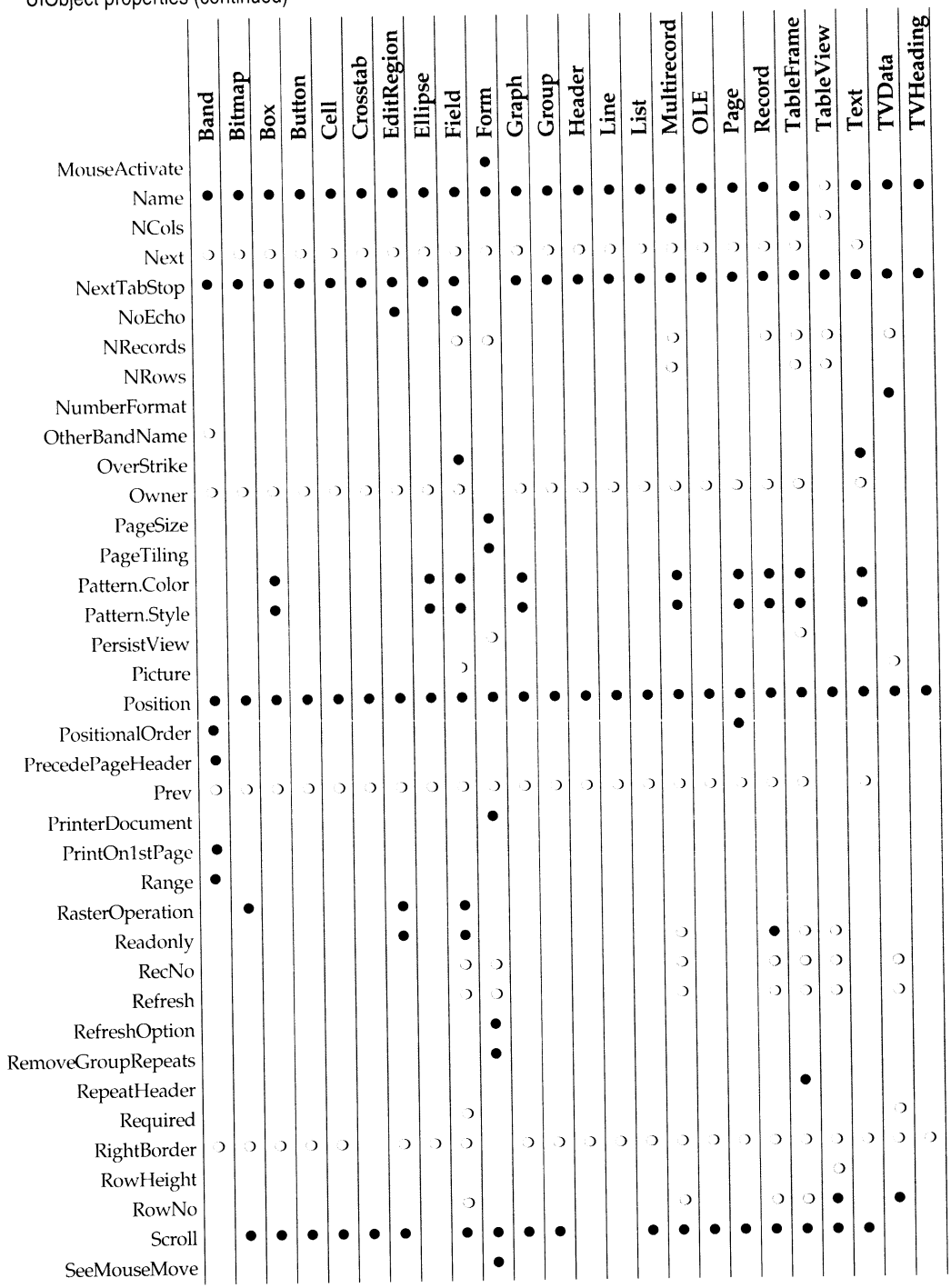


Table B.1 UIObject properties (continued)

	Band	Bitmap	Box	Button	Cell	Crosstab	EditRegion	Ellipse	Field	Form	Graph	Group	Header	Line	List	Multirecord	OLE	Page	Record	TableFrame	TableView	Text	TVData	TVHeading
Select	●	●	●	●	●	●	●	●	●		●			●	●	●	●	●	●	●	●	●	●	●
SelectedText									●													●		
SeqNo									○	○						○			○	○	○		○	
ShowGrid										○														
Shrinkable	●		●		●	●																		
Size	●	●	●	●	●	●	●	●	●	○	●	●	●	●	●	●	●	●	●	●	●	●	●	●
SizeToFit										●														
SnapToGrid										●														
SortOrder	●																							
SpecialField									●															
StandardMenu										●														
StandardToolbar										●														
Start														●										
StartPageNumbers	○																							
Style				●																				
SummaryModifier								●																
TabStop				●			●	●			●													
TableName						●		●		○	●					●			○	●	○		○	
Text								●														●		
ThickFrame										●														
Thickness														●										
TimeFormat																							●	
TimeStampFormat																							●	
Title										●														
TopBorder	○	○	○	○	○		○	○	○		○	○	○	○	○	○	○	○	○	○	○	○	○	○
TopLine									●													●		
Touched						○			●	●	○					●			●	●	○		○	
Translucent			●			●	●	●	●		●		●			●		●	●			●		
UncheckedValue				●																				
UpdateLink									●							●								
Value				●					●						●									
VerticalScrollBar		●			●	●			●	●					●	●			●	●		●	●	
Visible		●	●	●		●	●		●		●	●	●	●	●	●		●	●	●		●		
WideScrollBar									●					●	●	●				●		●		
Width																							●	
WordWrap						●			●													●		

Table B.1 UIObject properties (continued)

	Band	Bitmap	Box	Button	Cell	Crosstab	EditRegion	Ellipse	Field	Form	Graph	Group	Header	Line	List	Multirecord	OLE	Page	Record	TableFrame	TableView	Text	TVData	TVHeading
Xseparation																•								
Yseparation																•								

1. Write-only

Table B.2 Properties and property values

Property	Data type	Values	Description
Alignment	SmallInt	TextAlignCenter, TextAlignJustify, TextAlignLeft, TextAlignRight	Specifies the position of text relative to a field object or a text object.
Arrived	Logical	True, False	Specifies whether the object is active.
AttachedHeader	Logical	True, False	Determines whether a table frame's header is attached to the table frame.
AutoAppend	Logical	True, False	Determines whether Paradox automatically inserts a blank record when you're editing data in a form and you move the cursor past the end of a table.
AvgCharSize	Point	>0	Read-only property that gives the average width and height of a character in the current font.
BlankRecord	Logical	True, False	Reports whether a record is blank.
BottomBorder	LongInt	N/A	Returns the size of an object's bottom border (in twips).
Border	Logical	True, False	Reports whether a form's window has a border.
Breakable	Logical	True, False	Specifies whether an object can be split across page breaks.
ButtonType	SmallInt	CheckBoxType, PushButtonType, RadioButtonType	Specifies the display type of a button.
ByRows	Logical	True, False	Determines the record layout of a multirecord object in a form or report.
CalculatedField	Logical	True, False	Specifies whether a field is a calculated field.
Caption	Logical	True, False	Reports whether a form has a caption bar.
CenterLabel	Logical	True, False	Specifies whether a label is centered within a button.
CheckedValue	String	N/A	Specifies the string that the checkbox or radio button will write to its parent field object when a button is pressed.

Table B.2 Properties and property values (continued)

Property	Data type	Values	Description
Class	String	Band, Bitmap, Box, Button, Cell, Crosstab, EditRegion, Ellipse, Field, Form, Graph, Group, Header, Line, List, Multirecord, OLE, Page, Record, TableFrame, Text	Returns the class of a UIObject.
Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent	Specifies the display color of an object.
Columnar	Logical	True, False	Determines whether each individual record expands or contracts individually when you print or preview the report.
ColumnPosition	SmallInt	>0	Specifies the position (starting with 1) to which you want to move the current column.
ColumnWidth	LongInt	>0	Specifies the width in twips of the current column.
CompleteDisplay	Logical	True, False	Specifies whether to display the complete contents of a field.
ContainerName	String	N/A	Reports the name of an object's container.
ControlMenu	Logical	True, False	Specifies whether a form has a control menu.
CurrentColumn	SmallInt	>0	Determines the active column in crosstabs, multirecord objects, and table frames.
CurrentRecordMarker.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent	Specifies the display color of the current record of a table view.
CurrentRecordMarker.LineStyle	SmallInt	DashDotDotLine, DashDotLine, DashedLine, DottedLine, NoLine, SolidLine	Specifies the style of the line that marks the current record in a table view.
CurrentRecordMarker.Show	Logical	True, False	Specifies whether to highlight the current record in a table view.
CurrentRow	SmallInt	>0	Determines the active row in crosstabs, multirecord objects, and table frames.
CursorColumn	LongInt	N/A	The horizontal position of the insertion point in a field object, where position 0 is to the left of the first character.
CursorLine	LongInt	N/A	The vertical position of the insertion point in a field object, where the first line is line 1.
CursorPos	LongInt	N/A	The position of the insertion point in a field object, relative to the first character in the field. Counting begins with 0, the position to the left of the first character.
DataSource	String	N/A	The name of the table that provides items in a list.

Table B.2 Properties and property values (continued)

Property	Data type	Values	Description
DateFormat	String	Format specification	Specifies a field object's date format.
Default	String	N/A	The default value of a field.
DefineGroup	Logical	True, False	Specifies whether a report band defines a group.
DeleteColumn	SmallInt	>0	Specifies the number of the column to delete.
Deleted	Logical	True, False	Reports whether a record in a dBASE table has been flagged as deleted.
DeleteWhenEmpty	Logical	True, False	Causes the field to be deleted from the report if it has no data (including any labels, buttons, and so on).
Design.ContainObjects	Logical	True, False	Specifies whether an object can contain other objects.
Design.PinHorizontal	Logical	True, False	Specifies whether to prevent an object from moving horizontally.
Design.PinVertical	Logical	True, False	Specifies whether to prevent an object from moving vertically.
Design.Selectable	Logical	True, False	Specifies whether an object can be selected.
Design.SizeToFit	Logical	True, False	Specifies whether the object will change size to accomodate its contents.
DesignModified	Logical	True, False	Specifies whether a design is changed.
DesignSizing	SmallInt	TextFixedSize, TextGrowOnly, TextSizeToFit	Specifies design time sizing for a text box.
DesktopForm	Logical	True, False	Specifies whether a form's menus are used by other forms on the Desktop.
DialogForm	Logical	True, False	Specifies whether a form will open as a dialog box.
DialogFrame	Logical	True, False	When True and DialogForm and Border are also True, form has a conventional dialog box frame.
DisplayType	SmallInt	CheckBoxField, ComboField, EditField, LabeledField, ListField, RadioButtonField	Returns the display type of a field object.
Editing	Logical	True, False	Reports whether a form is in Edit mode, or a field object is active and being edited.
End	Point	N/A	Specifies the coordinates of the end of a line. See also: Start.
FieldName	String	N/A	Specifies the name of the field to which a field object or list is bound.
FieldNo	SmallInt	N/A	Reports the position of a field in a table, where the first field is field 1.
FieldRights	String	ReadOnly, ReadWrite, All	Reports the user's field rights.
FieldSize	SmallInt	N/A	The field's size (string and dBASE numbers).
FieldType	String	N/A	The field's data type.
FieldUnits2	SmallInt	N/A	Indicates the number of decimal places in a dBASE number field.

Table B.2 Properties and property values (continued)

Property	Data type	Values	Description
FieldValid	Logical	True, False	Reports whether a field passes its own value checks.
FieldView	Logical	True, False	Reports whether a field is in Field View.
First	String	N/A	Returns the name of the first child object in a container.
FirstRow	String	N/A	Returns the name of the first record object within a table frame or multirecord object.
FitHeight	Logical	True, False	Specifies whether an edit region expands vertically to accommodate text.
FitWidth	Logical	True, False	Specifies whether an edit region or crosstab cell expands horizontally to accommodate text.
FlyAway	Logical	True, False	Reports whether a record has moved to its sorted position in a table.
Focus	Logical	True, False	Reports whether an object's built-in **setFocus** method has been called. Set to False when the object's built-in **removeFocus** method is called.
Font.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent	Specifies the color of characters displayed in a field object or a text object.
Font.Size	SmallInt	>0	Specifies (in printer's points) the size of characters displayed in a field object or a text object.
Font.Style	SmallInt	FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline	Specifies the style of characters displayed in a field object or a text object.
Font.Typeface	String	Depends on system	Specifies the typeface of characters displayed in a field object or a text object.
Format.DateFormat	String	Format specification	Specifies the format for date values.
Format.LogicalFormat	String	Format specification	Specifies the format for logical values.
Format.NumberFormat	String	Format specification	Specifies the format for number values.
Format.TimeFormat	N/A	Format specification	Specifies the format for time values.
Format.TimeStampFormat	String	Format specification	Specifies the format for time stamps.
Frame.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent	Specifies the color of an object's frame.

Table B.2 Properties and property values (continued)

Property	Data type	Values	Description
Frame.Style	SmallInt	DashDotDotFrame, DashDotFrame, DashedFrame, DottedFrame, DoubleFrame, Inside3DFrame, NoFrame, Outside3DFrame, ShadowFrame, SolidFrame, WideInsideDoubleFrame, WideOutsideDoubleFrame	Specifies the style of an object's frame.
Frame.Thickness	SmallInt	N/A	Specifies the thickness of an object's frame in pixels.
FrameObjects	Logical	True, False	Specifies whether the dotted frame shows around objects in the designers.
FullName	String	N/A	Returns the full name (including containership path) of an object in a form.
FullSize	Point	N/A	In scrolling object, returns actual size if you could see the whole thing.
Grid.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent	Specifies the color of the grid in a table frame.
Grid.GridStyle	SmallInt	tfSingleLine, tfDoubleLine, tfTripleLine, tf3D, tfNoGrid	Specifies the style of grid lines in a table frame.
Grid.RecordDivider	Logical	True, False	Specifies whether dividing lines are displayed between records in a table frame.
GridLines.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent	Specifies the color of grid lines in a Table window.
GridLines.ColumnLines	Logical	True, False	Specifies whether to display column lines in a Table window.
GridLines.HeadingLines	Logical	True, False	Specifies whether to display heading lines in a Table window.
GridLines.LineStyle	SmallInt	DashDotDotLine, DashDotLine, DashedLine, DottedLine, NoLine, SolidLine	Specifies the style of grid lines in a Table window.
GridLines.QueryLook	Logical	True, False	Specifies whether a Table window displays grid lines in the same style as a Query Editor window.
GridLines.RowLines	Logical	True, False	Specifies whether to display grid lines in a Table window.
GridLines.Spacing	SmallInt	TextSingleSpacing, TextDoubleSpacing, TextTripleSpacing	Specifies the spacing of gridlines in a Table window.

Table B.2 Properties and property values (continued)

Property	Data type	Values	Description
GridValue	Point	>0	Determines the minimum grid interval in twips.
GroupObjects	Logical	True, False	Specifies whether to group selected objects in forms and reports.
GroupRecords	SmallInt	>0	Determines the number of records in a group in a report.
Header	String	N/A	Returns the name of a table frame's header object (if it has one).
Heading.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent	Specifies the color of the heading of a Table window.
Heading.Font.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent	Specifies the color of characters in the heading of a Table window.
Heading.Font.Size	SmallInt	Depends on system	Specifies the size of characters in the heading of a Table window.
Heading.Font.Style	SmallInt	FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline	Specifies the style of characters in a heading of a Table window.
Heading.Typeface	String	Depends on system	Specifies the typeface of characters in a heading of a Table window.
Heading.Justification	SmallInt	TextAlignTop, TextAlignBottom, TextAlignVCenter, TextAlignLeft, TextAlignRight, TextAlignCenter	Specifies the justification of characters in the heading of a Table window.
HeadingHeight	LongInt	>0	Determines (in twips) the height of the heading in a Table window.
Headings	String	"GroupOnly", "PageAndGroup"	Specifies which report headings to print.
HorizontalScrollBar	Logical	True, False	Specifies whether a table frame has a horizontal scroll bar.
IndexField	Logical	True, False	Reports whether a field object is bound to an indexed field in a table.
InsertColumn	SmallInt	>0	Specifies the position an inserted column will take.
InsertField	Point	N/A	Inserts a field object into a text box.
Inserting	Logical	True, False	Returns True when a record is being inserted anywhere in a form.

Table B.2 Properties and property values (continued)

Property	Data type	Values	Description
Invisible	Logical	True, False	Determines whether an object is visible at run time. Invisible applies only to boxes and lines in reports.
Justification	SmallInt	TextAlignTop, TextAlignBottom, TextAlignVCenter, TextAlignLeft, TextAlignRight, TextAlignCenter	Specifies the justification of data in a Table window.
KeyField	Logical	True, False	Reports whether a field object is bound to a key field in a table.
LabelText	String	N/A	Specifies the text displayed in a button's label.
LeftBorder	LongInt	N/A	Returns the size of an object's left border (in twips).
Line.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent	Specifies the color of a line for an ellipse.
Line.LineStyle	SmallInt	DashDotDotLine, DashDotLine, DashedLine, DottedLine, NoLine, SolidLine	Specifies the style of a line for an ellipse.
Line.Thickness	SmallInt	N/A	Specifies the thickness of a line for an ellipse.
LineEnds	SmallInt	NoArrow, OnBothEnds, OnOneEnd	Specifies whether (or where) to place arrows at the end of a line.
LineSpacing	SmallInt	TextDoubleSpacing, TextDoubleSpacing2, TextSingleSpacing, TextSingleSpacing2, TextTripleSpacing	Specifies the number of blank lines to print between each line of text in a field object or a text object.
LineStyle	SmallInt	DashDotDotLine, DashDotLine, DashedLine, DottedLine, NoLine, SolidLine	Specifies the style of a line.
LineType	SmallInt	CurvedLine, StraightLine	Specifies the type of a line.
List.Count	SmallInt	N/A	Specifies the number of items in a list.
List.Selection	SmallInt	N/A	Specifies the item selected from a list.
List.Value	AnyType	N/A	Determines the value of an item in a list.
Locked	Logical	True, False	Reports whether the table bound to a design object is locked.
LogicalFormat	String	Format specification	Specifies the format for logical values.
LookupTable	String	N/A	The name of the lookup table for a field object.
LookupType	String	JustCurrentField, AllCorresponding	Specifies the type of table lookup.

Table B.2 Properties and property values (continued)

Property	Data type	Values	Description
Magnification	SmallInt	Magnify100, Magnify200, Magnify25, Magnify400, Magnify50, MagnifyBestFit	Specifies the display magnification of a bitmap object. You can also enter a literal value.
Manager	String	N/A	Returns UIObject name of a form.
MarkerPos	LongInt	N/A	The "other end" of a selection. See also CursorPos.
Margins.Bottom	LongInt	>0	Specifies the height of the bottom margin in twips.
Margins.Left	LongInt	>0	Specifies the height of the left margin in twips.
Margins.Right	LongInt	>0	Specifies the height of the right margin in twips.
Margins.Top	LongInt	>0	Specifies the height of the top margin in twips.
MaximizeButton	Logical	True, False	Reports or specifes whether a form's window has a maximize box.
Maximum	String	N/A	Specifies the maximum value allowed in a field.
MemoView	Logical	True, False	Specifies whether a field object is in memo view mode.
MinimizeButton	Logical	True, False	Reports or specifies whether a form's window has a minimize box.
Minimum	String	N/A	Specifies the maximum value allowed in a field.
Modal	Logical	True, False	Specifies whether a dialog form is modal.
MouseActivate	Logical	True, False	Specifies whether a dialog form gets focus as the result of a MouseEvent.
NCols	SmallInt	N/A	Returns the number of columns in a table frame or multi-record object.
NRecords	LongInt	N/A	Reports the number of records in the table bound to a design object.
NRows	SmallInt	N/A	Returns the number of rows in a table frame or multi-record object.
Name	String	N/A	Specifies the name of a design object.
Next	String	N/A	Returns the name of the next object in the same container.
NextTabStop	String	N/A	Specifies the object name of the next tab stop on a form.
NoEcho	Logical	True, False	Specifies whether characters typed into a field object are displayed.
NRecords	LongInt	N/A	Reports the number of records in the table bound to a design object.
NRows	SmallInt	N/A	Returns the number of rows in a table frame or multi-record object.
NumberFormat	String	Format specification	Specifies the format for number values.
OtherBandName	String	N/A	Returns the name of a report band's counterpart (the "other" band in a pair of bands).

Table B.2 Properties and property values (continued)

Property	Data type	Values	Description
OverStrike	Logical	True, False	Specifies whether a field or text object is in overstrike (as opposed to insert) mode.
Owner	String	N/A	Name of the logical container of an object, irrespective of intermediate cosmetic objects.
PageSize	Point	>0	Determines the size of a page in a report in a design window.
PageTiling	SmallInt	StackPages, TileHorizontal, TileVertical	Determines how to arrange pages in a form. It uses PageTiling constants.
Pattern.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent	Specifies the color of a pattern.
Pattern.Style	SmallInt	Bricks, CrosshatchPattern, DiagonalCrosshatchPattern, DottedLinePattern, EmptyPattern, FuzzyStripesDownPattern, HeavyDotsPattern, HorizontalLinesPattern, LatticePattern, LeftDiagonalLinesPattern, LightDotsPattern, MaximumDotsPattern, MinimumDotsPattern, MediumDotsPattern, RightDiagonalLinesPattern, ScalesPattern, StaggeredLinesPattern, ThickHorizontalLinesPatter n, ThickStripesDownPattern, ThickStripesUpPattern, ThickVerticalLinesPattern, VerticalLinesPattern, WeavePattern, ZigZagPattern	Specifies the style of a pattern.
PersistView	Logical	True, False	Specifies whether to remain in Field View or Memo View.
Picture	String	N/A	A template that formats the value of a field.
PinHorizontal	Logical	True, False	Specifies whether to prevent an object from moving horizontally.
PinVertical	Logical	True, False	Specifies whether to prevent an object from moving vertically.
Position	Point	N/A	Specifies the coordinates of the upper right corner of a design object.
PositionalOrder	SmallInt		For Page objects, specifies the page number. For Band objects, specifies the position the band occupies, counting from the top.
PrecedePageHeader	Logical	True, False	Specifies whether a report band should appear before the page header.

Table B.2 Properties and property values (continued)

Property	Data type	Values	Description
Prev	String	N/A	Returns the name of the previous object in the same container.
PrintDocument	Logical	True. False	Specifies whether the document is designed for printer or screen.
PrintOn1stPage	Logical	True, False	Specifies whether to print a report band on the first page of the report.
Range	SmallInt	Depends on field type.	Used on report bands to convert a group on a field to a group on a range of that field (or to change the range on a range group).
RasterOperation	LongInt	MergePaint, NotSourceCopy, NotSourceErase, SourceAnd, SourceCopy, SourceErase, SourceInvert, SourcePaint	Specifies how to blend the colors in two overlapping design objects.
Readonly	Logical	True, False	Specifies whether a field object is read-only.
RecNo	LongInt	N/A	Reports the position of a record. (This can be a time-consuming operation for dBASE tables.)
Refresh	Logical	True, False	Reports when data displayed onscreen is being changed, either across a network, by an ObejctPAL statement, or user action.
RefreshOption	SmallInt	PrintFromCopy, PrintLock, PrintNoLock, PrintRestart, PrintReturn	Determines what to do when data changes while printing a report. It uses ReportPrintRestart constants.
RemoveGroupRepeats	Logical	True, False	Retains or suppresses repeated group values within a record band.
RepeatHeader	Logical	True, False	Determines whether the header is repeated on each page of a report.
Required	Logical	True, False	Reports whether a field object must be assigned a value for the record to be valid.
RightBorder	LongInt	N/A	Returns the size of an object's right border (in twips).
RowHeight	LongInt	>0	Determines the height (in twips) of a row in a Table window.
RowNo	SmallInt	N/A	Reports the row number of a record displayed in a table frame, mutli-record object, or table view, starting with 1.
Scroll	Point	N/A	How far you've scrolled.
SeeMouseMove	Logical	True, False	Specifies whether a form that does not have focus will respond to mouse movements.
Select	Logical	True, False	Specifies whether an object is selected.
SelectedText	String	N/A	Returns the selected text in a field object.
SeqNo	LongInt	N/A	The actual sequence number of a record as displayed, taking filters and indexes into account.
ShowGrid	Logical	True, False	Specifies whether the form or report grid is visible.
Shrinkable	Logical	True, False	Specifies whether a report band can be shrunk.

Table B.2 Properties and property values (continued)

Property	Data type	Values	Description
Size	Point	N/A	Specifies the coordinates of the lower left corner of a design object.
SizeToFit	Logical	True, False	When True, the form opens at the size of the underlying page. When False, the form opens at the size it was saved.
SnapToGrid	Logical	True, False	When SnapToGrid is True, design objects jump to the closest minor division of the grid when moved or resized. When SnapToGrid is False, object size and position are not affected by the grid.
SortOrder	Logical	True, False	Specifies the sort order of a report. True = Descending order, False = Ascending order.
SpecialField	SmallInt	DateField, NofFieldsField, NofPagesField, NofRecsField, PageNumField, RecordNoField, TableNameField, TimeField	Determines the type of a special field. It uses SpecialFieldTypes constants.
StandardMenu	Logical	True, False	Specifies whether a form uses the standard menus.
StandardToolbar	Logical	True, False	Specifies whether a form or report uses the standard ToolBar.
Start	Point	N/A	Specifies the coordinates of the start of a line. See also: End.
StartPageNumbers	SmallInt	>0	Determines the starting value for page numbers in a report.
Style	SmallInt	BorlandButton, WindowsButton	Reports or specifes the display style of a button.
SummaryModifier			
TabStop	Logical	True, False	Specifies whether a field object is a tab stop.
TableName	String	N/A	Specifies the name of a table to which a design object is bound.
Text	String	N/A	Specifies the characters displayed in a text object.
ThickFrame	Logical	True, False	When True, and DialogForm and Border also True, specifies a thick window frame instead of the usual pixel-wide frame.
Thickness	SmallInt	LWidth10Points, LWidth1Point, LWidth2Points, LWidth3Points, LWidth6Points, LWidthHairline, LWidthHalfPoint	Specifies the thickness of a line.
TimeFormat	N/A	Format specification	Specifies the format for time values.
TimeStampFormat	String	Format specification	Specifies the format for time stamps.
Title	String	N/A	Specifies the text of a form's caption.
TopBorder	LongInt	N/A	Returns the size of an object's top border (in twips).

Table B.2 Properties and property values (continued)

Property	Data type	Values	Description
TopLine	LongInt	N/A	The number of the line currently displayed at the top of a text object.
Touched	Logical	True, False	True when user has made changes not yet committed.
Translucent	Logical	True, False	Specifies whether the color of an object is translucent.
UncheckedValue	String	N/A	Specifies the value that a button will write to its parent field object when the button is unchecked.
UpdateLink	SmallInt	OLEUpdateAutomatic, OLEUpdateManual	Determines the update method for an OLE object.
Value	String	N/A	Specifies the value of a design object.
VerticalScrollBar	Logical	True, False	Specifies whether an object has a vertical scroll bar. Not valid for all UIObjects. Refer to chart.
Visible	Logical	True, False	Specifies whether an object is displayed.
WideScrollBar	Logical	True, False	Determines whether a scroll bar is wide or narrow (the default). If True, the scroll bar is wide; otherwise, it is narrow.
Width	LongInt	>0	Determines the width (in twips) of a column in a Table window.
WordWrap	Logical	True, False	Specifies whether to wrap lines that exceed the width of a field object.
Xseparation	LongInt	>0	Specifies the distance (in twips) between records in the indicated direction.
Yseparation	LongInt	>0	Specifies the distance (in twips) between records in the indicated direction.

Table B.3 Properties unique to graph objects

Property	Data type	Values
BackWall.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
BackWall.Pattern.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
BackWall.Pattern.Style	SmallInt	BricksPattern, CrosshatchPattern, DiagonalCrosshatchPattern, DottedLinePattern, EmptyPattern, FuzzyStripesDownPattern, HeavyDotsPattern, HorizontalLinesPattern, LatticePattern, LeftDiagonalLinesPattern, LightDotsPattern, MaximumDotsPattern, MinimumDotsPattern, MediumDotsPattern, RightDiagonalLinesPattern, ScalesPattern, StaggeredLinesPattern, ThickHorizontalLinesPattern, ThickStripesDownPattern, ThickStripesUpPattern, ThickVerticalLinesPattern, VerticalLinesPattern, WeavePattern, ZigZagPattern

Table B.3 Properties unique to graph objects (continued)

Property	Data type	Values
Background.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
Background.Pattern.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
Background.Pattern.Style	SmallInt	BricksPattern, CrosshatchPattern, DiagonalCrosshatchPattern, DottedLinePattern, EmptyPattern, FuzzyStripesDownPattern, HeavyDotsPattern, HorizontalLinesPattern, LatticePattern, LeftDiagonalLinesPattern, LightDotsPattern, MaximumDotsPattern, MinimumDotsPattern, MediumDotsPattern, RightDiagonalLinesPattern, ScalesPattern, StaggeredLinesPattern, ThickHorizontalLinesPattern, ThickStripesDownPattern, ThickStripesUpPattern, ThickVerticalLinesPattern, VerticalLinesPattern, WeavePattern, ZigZagPattern
BaseFloor.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
BaseFloor.Pattern.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
BaseFloor.Pattern.Style	SmallInt	BricksPattern, CrosshatchPattern, DiagonalCrosshatchPattern, DottedLinePattern, EmptyPattern, FuzzyStripesDownPattern, HeavyDotsPattern, HorizontalLinesPattern, LatticePattern, LeftDiagonalLinesPattern, LightDotsPattern, MaximumDotsPattern, MinimumDotsPattern, MediumDotsPattern, RightDiagonalLinesPattern, ScalesPattern, StaggeredLinesPattern, ThickHorizontalLinesPattern, ThickStripesDownPattern, ThickStripesUpPattern, ThickVerticalLinesPattern, VerticalLinesPattern, WeavePattern, ZigZagPattern
BindType	SmallInt	Graph1DSummary, Graph2DSummary, GraphTabular
CurrentSeries	SmallInt	N/A
CurrentSlice	SmallInt	N/A
GraphType	SmallInt	Graph2DArea, Graph2DBar, Graph2DColumns, Graph2DLine, Graph2DPie, Graph2DRotatedBar, Graph2DStackedBar, Graph3DArea, Graph3DBar, Graph3DColumns, Graph3DPie, Graph3DRibbon, Graph3DRotatedBar, Graph3DStackedBar, Graph3DStep, Graph3DSurface, Graph3DXY
Label.Font.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
Label.Font.Size	SmallInt	Depends on system
Label.Font.Style	SmallInt	FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline
Label.Font.Typeface	String	Depends on system
Label.LabelFormat	SmallInt	GraphHideY, GraphPercent, GraphShowY
Label.LabelLocation	SmallInt	LabelAbove, LabelBelow, LabelBottom, labelCenter, LabelLeft, LabelMiddle, LabelRight, LabelTop
Label.NumberFormat	N/A	Format specification

Table B.3 Properties unique to graph objects (continued)

Property	Data type	Values
LeftWall.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
LeftWall.Pattern.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
LeftWall.Pattern.Style	SmallInt	BricksPattern, CrosshatchPattern, DiagonalCrosshatchPattern, DottedLinePattern, EmptyPattern, FuzzyStripesDownPattern, HeavyDotsPattern, HorizontalLinesPattern, LatticePattern, LeftDiagonalLinesPattern, LightDotsPattern, MaximumDotsPattern, MinimumDotsPattern, MediumDotsPattern, RightDiagonalLinesPattern, ScalesPattern, StaggeredLinesPattern, ThickHorizontalLinesPattern, ThickStripesDownPattern, ThickStripesUpPattern, ThickVerticalLinesPattern, VerticalLinesPattern, WeavePattern, ZigZagPattern
LegendBox.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
LegendBox.Font.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
LegendBox.Font.Size	SmallInt	Depends on system
LegendBox.Font.Style	SmallInt	FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline
LegendBox.Font.Typeface	String	Depends on system
LegendBox.LegendPos	SmallInt	LegendLeft, LegendRight
LegendBox.Pattern.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
LegendBox.Pattern.Style	SmallInt	BricksPattern, CrosshatchPattern, DiagonalCrosshatchPattern, DottedLinePattern, EmptyPattern, FuzzyStripesDownPattern, HeavyDotsPattern, HorizontalLinesPattern, LatticePattern, LeftDiagonalLinesPattern, LightDotsPattern, MaximumDotsPattern, MinimumDotsPattern, MediumDotsPattern, RightDiagonalLinesPattern, ScalesPattern, StaggeredLinesPattern, ThickHorizontalLinesPattern, ThickStripesDownPattern, ThickStripesUpPattern, ThickVerticalLinesPattern, VerticalLinesPattern, WeavePattern, ZigZagPattern
MaxGroups	SmallInt	Depends on graph
MaxXValues	SmallInt	Depends on graph
MinXValues	SmallInt	Depends on graph
Options.Elevation	SmallInt	0 to 90 degrees
Options.Rotation	SmallInt	0 to 90 degrees
Options.ShowAxes	Logical	True, False
Options.ShowGrid	Logical	True, False
Options.ShowLabels	Logical	True, False
Options.ShowLegend	Logical	True, False
Options.ShowTitle	Logical	True, False

Table B.3 Properties unique to graph objects (continued)

Property	Data type	Values
Series.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
Series.Graph_Title.Font.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
Series.Graph_Title.Font.Size	SmallInt	Depends on system
Series.Graph_Title.Font.Style	SmallInt	FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline
Series.Graph_Title.Font.Typeface	String	Depends on system
Series.Graph_Title.Text	String	N/A
Series.Graph_Title.UseDefault	Logical	True, False
Series.Line.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
Series.Line.LineStyle	SmallInt	DashDotDotLine, DashDotLine, DashedLine, DottedLine, NoLine, SolidLine
Series.Line.Thickness	SmallInt	LWidth10Points, LWidth1Point, LWidth2Points, LWidth3Points, LWidth6Points, LWidthHairline, LWidthHalfPoint
Series.Marker.Size	SmallInt	MarkerSize0, MarkerSize2, MarkerSize4, MarkerSize8, MarkerSize12, MarkerSize18, MarkerSize24, MarkerSize36, MarkerSize54, MarkerSize72
Series.Marker.Style	SmallInt	MarkerBoxedCross, MarkerBoxed_Plus, MarkerCross, MarkerFilledBox, MarkerFilledCircle, MarkerFilledDownTriangle, MarkerFilledTriangle, MarkerFilledTriangles, MarkerHollowBox, MarkerHollowCircle, MarkerHollowDownTriangle, MarkerHollowTriangle, MarkerHollowTriangles, MarkerHorizontalLine, MarkerPlus, MarkerVerticalLine
Series.Marker	SmallInt	MarkerBoxedCross, MarkerBoxed_Plus, MarkerCross, MarkerFilledBox, MarkerFilledCircle, MarkerFilledDownTriangle, MarkerFilledTriangle, MarkerFilledTriangles, MarkerHollowBox, MarkerHollowCircle, MarkerHollowDownTriangle, MarkerHollowTriangle, MarkerHollowTriangles, MarkerHorizontalLine, MarkerPlus, MarkerVerticalLine
Series.Pattern.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
Series.Pattern.Style	SmallInt	BricksPattern, CrosshatchPattern, DiagonalCrosshatchPattern, DottedLinePattern, EmptyPattern, FuzzyStripesDownPattern, HeavyDotsPattern, HorizontalLinesPattern, LatticePattern, LeftDiagonalLinesPattern, LightDotsPattern, MaximumDotsPattern, MinimumDotsPattern, MediumDotsPattern, RightDiagonalLinesPattern, ScalesPattern, StaggeredLinesPattern, ThickHorizontalLinesPattern, ThickStripesDownPattern, ThickStripesUpPattern, ThickVerticalLinesPattern, VerticalLinesPattern, WeavePattern, ZigZagPattern
Series.TypeOverride	SmallInt	Graph2DArea, Graph2DBar, Graph2DLine, None
SeriesName	String	N/A
Slice.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

Table B.3 Properties unique to graph objects (continued)

Property	Data type	Values
Slice.Explode	Logical	True, False
Slice.Pattern.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
Slice.Pattern.Style	SmallInt	BricksPattern, CrosshatchPattern, DiagonalCrosshatchPattern, DottedLinePattern, EmptyPattern, FuzzyStripesDownPattern, HeavyDotsPattern, HorizontalLinesPattern, LatticePattern, LeftDiagonalLinesPattern, LightDotsPattern, MaximumDotsPattern, MinimumDotsPattern, MediumDotsPattern, RightDiagonalLinesPattern, ScalesPattern, StaggeredLinesPattern, ThickHorizontalLinesPattern, ThickStripesDownPattern, ThickStripesUpPattern, ThickVerticalLinesPattern, VerticalLinesPattern, WeavePattern, ZigZagPattern
TitleBox.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
TitleBox.Graph_Title.Font.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
TitleBox.Graph_Title.Font.Size	SmallInt	Depends on system
TitleBox.Graph_Title.Font.Style	SmallInt	FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline
TitleBox.Graph_Title.Font.Typeface	String	Depends on system
TitleBox.Graph_Title.Text	String	N/A
TitleBox.Graph_Title.UseDefault	Logical	True, False
TitleBox.Pattern.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
TitleBox.Pattern.Style	SmallInt	BricksPattern, CrosshatchPattern, DiagonalCrosshatchPattern, DottedLinePattern, EmptyPattern, FuzzyStripesDownPattern, HeavyDotsPattern, HorizontalLinesPattern, LatticePattern, LeftDiagonalLinesPattern, LightDotsPattern, MaximumDotsPattern, MinimumDotsPattern, MediumDotsPattern, RightDiagonalLinesPattern, ScalesPattern, StaggeredLinesPattern, ThickHorizontalLinesPattern, ThickStripesDownPattern, ThickStripesUpPattern, ThickVerticalLinesPattern, VerticalLinesPattern, WeavePattern, ZigZagPattern
TitleBox.Subtitle.Font.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
TitleBox.Subtitle.Font.Size	SmallInt	Depends on system
TitleBox.Subtitle.Font.Style	SmallInt	FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline
TitleBox.Subtitle.Font.Typeface	String	Depends on system
TitleBox.Subtitle.Text	String	N/A
TitleBox.Subtitle.UseDefault	Logical	True, False
TitleBoxName	String	N/A
XAxisName	String	N/A

Table B.3 Properties unique to graph objects (continued)

Property	Data type	Values
XAxis.Graph_Title.Font.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
XAxis.Graph_Title.Font.Size	SmallInt	Depends on system
XAxis.Graph_Title.Font.Style	SmallInt	FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline
XAxis.Graph_Title.Font.Typeface	String	Depends on system
XAxis.Graph_Title.Text	String	N/A
XAxis.Graph_Title.UseDefault	Logical	True, False
XAxis.Scale.AutoScale	Logical	True, False
XAxis.Scale.HighValue	Number	Depends on graph
XAxis.Scale.Increment	Number	Depends on graph
XAxis.Scale.Logarithmic	Logical	True, False
XAxis.Scale.LowValue	Number	Depends on graph
XAxis.Ticks.Alternate	Logical	True, False
XAxis.Ticks.DateFormat	N/A	Format specification
XAxis.Ticks.Font.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
XAxis.Ticks.Font.Size	SmallInt	Depends on system
XAxis.Ticks.Font.Style	SmallInt	FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline
XAxis.Ticks.Font.Typeface	String	Depends on system
XAxis.Ticks.NumberFormat	N/A	Format specification
XAxis.Ticks.TimeFormat	String	N/A
XAxis.Ticks.TimeStampFormat	String	N/A
YAxisName	String	N/A
YAxis.Graph_Title.Font.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
YAxis.Graph_Title.Font.Size	SmallInt	Depends on system
YAxis.Graph_Title.Font.Style	SmallInt	FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline
YAxis.Graph_Title.Font.Typeface	String	Depends on system
YAxis.Graph Title.Text	String	N/A
YAxis.Graph_Title.UseDefault	Logical	True, False
YAxis.Scale.AutoScale	Logical	True, False
YAxis.Scale.HighValue	Number	Depends on graph
YAxis.Scale.Increment	Number	Depends on graph
YAxis.Scale.Logarithmic	Logical	True, False
YAxis.Scale.LowValue	Number	Depends on graph
YAxis.Ticks.Alternate	Logical	True, False
YAxis.Ticks.DateFormat	N/A	Format specification

Table B.3 Properties unique to graph objects (continued)

Property	Data type	Values
YAxis.Ticks.Font.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
YAxis.Ticks.Font.Size	SmallInt	Depends on system
YAxis.Ticks.Font.Style	SmallInt	FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline
YAxis.Ticks.Font.Typeface	String	Depends on system
YAxis.Ticks.Number.Format	N/A	Format specification
YAxis.Ticks.TimeFormat	String	N/A
YAxis.Ticks.TimeStampFormat	String	N/A
ZAxisName	String	N/A
ZAxis.Graph_Title.Font.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
ZAxis.Graph_Title.Font.Size	SmallInt	Depends on system
ZAxis.Graph_Title.Font.Style	SmallInt	FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline
ZAxis.Graph_Title.Font.Typeface	String	Depends on system
ZAxis.Graph Title.Text	String	N/A
ZAxis.Graph_Title.UseDefault	Logical	True, False
ZAxis.Scale.AutoScale	Logical	True, False
ZAxis.Scale.HighValue	Number	Depends on graph
ZAxis.Scale.Increment	Number	Depends on graph
ZAxis.Scale.Logarithmic	Logical	True, False
ZAxis.Scale.LowValue	Number	Depends on graph
ZAxis.Ticks.Alternate	Logical	True, False
ZAxis.Ticks.DateFormat	String	N/A
ZAxis.Ticks.Font.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
ZAxis.Ticks.Font.Size	SmallInt	Depends on system
ZAxis.Ticks.Font.Style	SmallInt	FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline
ZAxis.Ticks.Font.Typeface	String	Depends on system
ZAxis.Ticks.NumberFormat	N/A	Format specification
ZAxis.Ticks.TimeFormat	String	N/A
ZAxis.Ticks.TimeStampFormat	String	N/A

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Borland

Corporate Headquarters: 100 Borland Way, Scotts Valley, CA 95066-3249, (408) 431-1000. Offices in: Australia, Belgium, Brazil, Canada, Chile, Denmark, France, Germany, Hong Kong, Italy, Japan, Korea, Latin America, Malaysia, Netherlands, Singapore, Spain, Sweden, Taiwan, and United Kingdom • Part # PDX1150WW21773 • BOR 7226