Paradox for Windows

Version 1.0

ObjectPAL™ Developer's Guide

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

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

10 9 8 7 6 5 4 3 2 1 R1

CONTENTS

TABLES

FIGURES

CHAPTER 1

Introduction

This manual is an intermediate-to-advanced guide for ObjectPAL™, Paradox for Windows' integrated language for developing Windows applications.

This chapter

- Gives a brief overview of ObjectPAL
- Describes how the manual is organized

You should read and walk through the examples in *Learning ObjectPAL* before using this manual.

The ObjectPAL Environment

ObjectPAL has two aspects:

- The Integrated Development Environment (IDE), including
  - The ObjectPAL Editor
  - The ObjectPAL Debugger
  - A mechanism for creating and playing scripts
  - Facilities for delivering completed applications to users
- The language itself, including
  - The ObjectPAL code, built in to every object, that defines how the object responds to events
  - The run-time library: predefined ObjectPAL methods and procedures that operate on objects and data, including methods for working with tables, for opening and closing forms, reports, and table views, and for doing calculations

- The basic language elements, including keywords for declaring procedures and variables and control structures like **for** loops and **if**...**then**...**else** blocks

Like C and Pascal, ObjectPAL is a *compiled* (as opposed to *interpreted*) language. That is, the ObjectPAL compiler translates the ObjectPAL code you write into machine code that a computer can execute. The compiler translates all the ObjectPAL code in a form at one time, rather than interpreting each instruction just before it executes. As a result, applications run faster.

Object types are tools

Paradox and ObjectPAL let you create compiled applications from the object types listed in Table 1-1. These object types, described in Chapters 6–11, are your tools. Each ObjectPAL tool has special characteristics and capabilities. The more you know about these tools—these object types—the better-equipped you'll be to solve your users' problems.

Table 1-1 Categories and object types

Category	Description	Object types
Events	Contain information about actions in Paradox	ActionEvent, ErrorEvent, Event, KeyEvent, MenuEvent, MouseEvent, MoveEvent, StatusEvent, TimerEvent, ValueEvent
Design objects	Create the user interface to your application	Menu, PopUpMenu, UIObject
Display managers	Control how data is presented to the user	Application, Form, Report, TableView
Data types	Define what type of data you're working with	AnyType, Array, Binary, Currency, Date, DynArray, Graphic, Logical, LongInt, Memo, Number, OLE, Point, SmallInt, String, Time
Data model objects	Let you work with data stored in tables	Database, Query, Table, TCursor
System data objects	Let you work with data not in tables	DDE, FileSystem, Library, Session, System, TextStream

Keep in mind, though, that you don't have to master every nuance of ObjectPAL to be productive. Start small. Using the tools provided in Paradox and ObjectPAL, you can build powerful applications incrementally. That is, you can build some of the tables, create some of the objects, and write some of the methods, test and refine these, then add more tables, objects, and methods, and so on until the application is complete.

The ObjectPAL documentation

The ObjectPAL documentation consists of three manuals, a quick reference guide, an online Help system, a sample application, and various smaller example files.

Manuals

The printed documentation for ObjectPAL consists of the following:

- *Learning ObjectPAL* presents background material and tutorial exercises to introduce fundamental ObjectPAL concepts.
- *ObjectPAL Developer's Guide* (this manual) provides the detailed information and examples you need to develop full-featured Windows database applications.
- *ObjectPAL Reference* is a complete reference to the built-in methods, basic language elements, and methods and procedures in the ObjectPAL run-time library. Each entry gives the syntax, a description, and a code example.
- *Quick Reference* lists ObjectPAL methods and procedures in alphabetical order, along with the syntax and a brief synopsis of each.

Online Help

The online Help system for Paradox provides comprehensive information about ObjectPAL, including conceptual material and a complete language reference. You can paste code from examples in the online Help into your own applications. To get help at any time, press *F1*.

Example files

The example files include a sample application, MAST. MAST shows how to accomplish many common programming tasks within the framework of an information-retrieval and data-entry application.

In addition to MAST, various smaller, less complex applications are provided to demonstrate specific concepts or programming approaches.

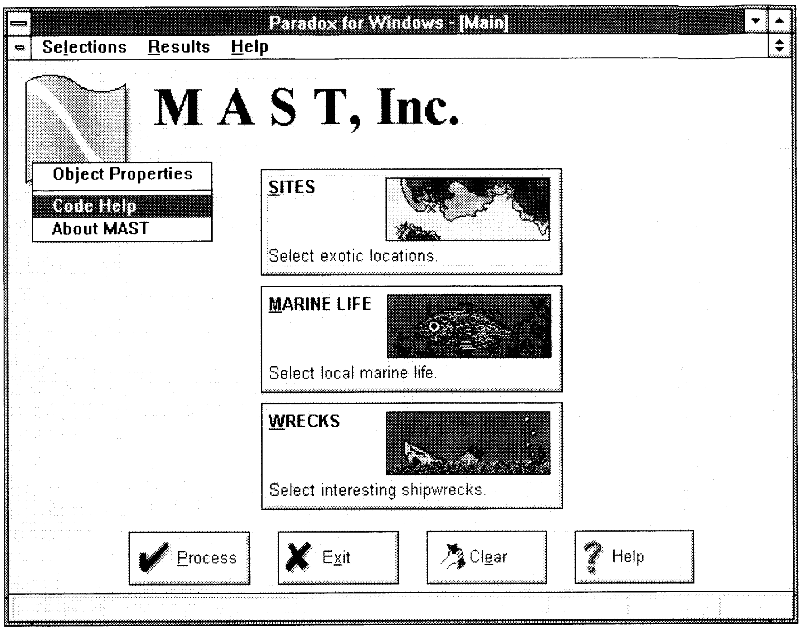
Inspect objects in example files for information on their ObjectPAL code.

All of the example applications include online Help. When you run any of these applications, you can press *F1* to get general information about how to use the application, and you can inspect (right-click) any object to get specific information about the ObjectPAL code attached to that object, as shown in Figure 1-1.

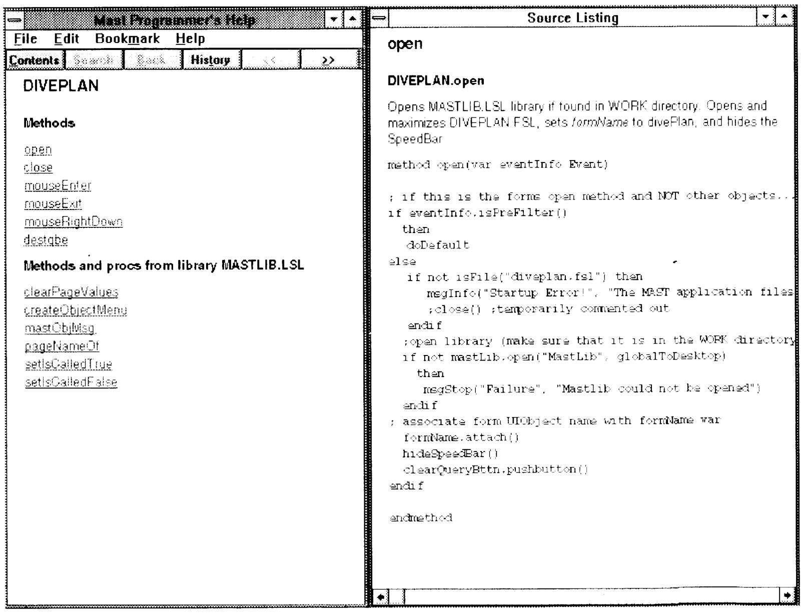
Figure 1-1 Using the Help system in the example applications

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.

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



When you right-click an object and choose Code Help, a Help window opens listing all the ObjectPAL code attached to that object. Choose a method from this list to open a second window containing the source code for the method and text that explains it.



DIVEPLAN

Methods

open
close
mouseEnter
mouseExit
mouseRightDown
destroy

Methods and procs from library MASTLIB.LSL

clearPageValues
createObjectMenu
mastObjMsg
pageNameOf
setIsCalledTrue
setIsCalledFalse

open

DIVEPLAN.open

Opens MASTLIB.LSL library if found in WORK directory. Opens and maximizes DIVEPLAN.FSL, sets *formName* to divePlan, and hides the SpeedBar

```
method open(var eventInfo Event)

; if this is the forms open method and NOT other objects...
if eventInfo.isPreFilter()
  then
   doDefault
else
   if not isFile("diveplan.fsl") then
      msgInfo("Startup Error!", "The MAST application files
      ;close() ;temporarily commented out
   endif
  ;open library (make sure that it is in the WORK directory
  if not mastLib.open("MastLib", globalToDesktop)
    then
     msgStop("Failure", "Mastlib could not be opened")
  endif
; associate form UIObject name with formName var
  formName.attach()
  hideSpeedBar()
  clearQueryBttn.pushbutton()
endif

endmethod
```

How to use this manual

The organization of this manual reflects the object-based nature of ObjectPAL.

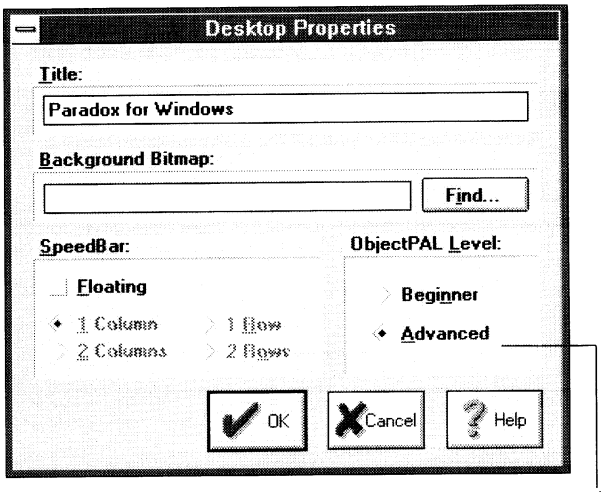
The *ObjectPAL Developer's Guide* is divided into these parts:

- Part I, "Fundamentals," describes the Integrated Development Environment, including the ObjectPAL Editor and Debugger, as well as the facilities for creating and delivering applications. It also presents details about the structure of ObjectPAL.
- Part II, "Object types," describes the ObjectPAL event model and discusses how to use ObjectPAL to work with objects in each of the following categories:
  - Events
  - Design objects
  - Display managers
  - Data types
  - Data model objects
  - System data objects
- Part III, "Special topics," discusses multiuser issues, error handling, how to use scripts, and how to deliver applications.
- Appendix A, "PAL and ObjectPAL," lists differences between the PAL and ObjectPAL languages.
- At the end of this manual are a glossary and an index.

The online Help covers main topics and provides reference information for every ObjectPAL method and procedure.

Important This manual presents the full range of ObjectPAL features and functions and assumes you have configured Paradox to do the same. This manual also assumes you have set the ObjectPAL Level to Advanced, as shown in Figure 1-2. To set the ObjectPAL Level, choose Properties | Desktop to open the Desktop Properties dialog box. Then, in the ObjectPAL Level panel, choose Advanced.

Figure 1-2 Setting the ObjectPAL Level



Title: Paradox for Windows

Background Bitmap:

ObjectPAL Level: Advanced

This manual assumes you have set the ObjectPAL level to Advanced, as shown in this figure

Printing conventions

Table 1-2 lists the printing conventions used in this book.

Note 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 **deleteDir** method defined for the FileSystem type. That's a reference to the method whose complete syntax is

deleteDir (const ***dirName*** String) Logical

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

Table 1-2 Printing conventions

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

ObjectPAL syntax notation is described fully at the beginning of the *ObjectPAL Reference*, and summarized here:

Table 1-3 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.

PART I

Fundamentals

This part of the *ObjectPAL Developer's Guide* presents the Integrated Development Environment (IDE) and gives an overview of how to work with the language.

Part I includes the following chapters:

- Chapter 2, "ObjectPAL overview," gives an overview of objects, events, and methods.
- Chapter 3, "The ObjectPAL Editor," explains how to use the built-in Paradox Editor to create and modify ObjectPAL code.
- Chapter 4, "The ObjectPAL Debugger," explains how to use the built-in debugging tool to test your code.
- Chapter 5, "Language structure," discusses the general structure and syntax of ObjectPAL code, including conventions for naming and using variables and constants.

CHAPTER 2

ObjectPAL overview

This chapter contains the following sections:

- "Objects" defines the word "object" in ObjectPAL terms.
- "An object-based strategy" presents a methodology for building Paradox applications.
- "Object categories" expains how ObjectPAL classifies objects according to type, and how these types are grouped into larger categories.
- "Where do I go from here?" points you toward those parts of the documentation that explain how to accomplish common programming tasks.

For a more fundamental discussion of these topics, see *Learning ObjectPAL.*

Objects

In Paradox, everything is an object—from the buttons and fields you create using tools in the SpeedBar, to tables and text files stored on disk, to menus and pop-up menus you create in code. Paradox recognizes two kinds of objects: *design objects,* the objects you place in a form; and *data objects,* which include files, data types, and programming structures. A formal definition says that an object consists of data and code. In ObjectPAL terms, objects have properties (like color, position, font, and line width) and methods (code that defines how the object behaves). Properties are data. Methods are code.

You're stepping into a complete object-based environment. Paradox objects come with built-in code, so most of the work you do developing ObjectPAL applications will feel more like customization and modification than coding from scratch. You don't have to take control of the entire program just to control a few things.

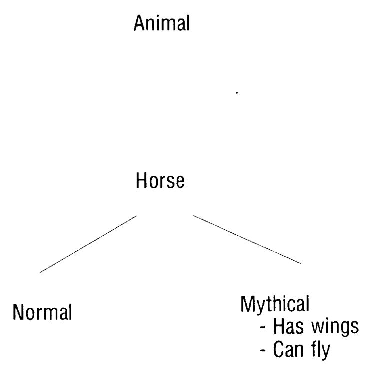
An object-based language improves applications development by providing tools more closely suited to the way people think about the world. With object-based programming, you can model the complexity of real-world systems and develop large-scale applications in the most efficient way possible.

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 2-1. The object-based paradigm lets you organize things in a program in much the same way they're organized in conversation.

Figure 2-1 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 and ObjectPAL

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")
```

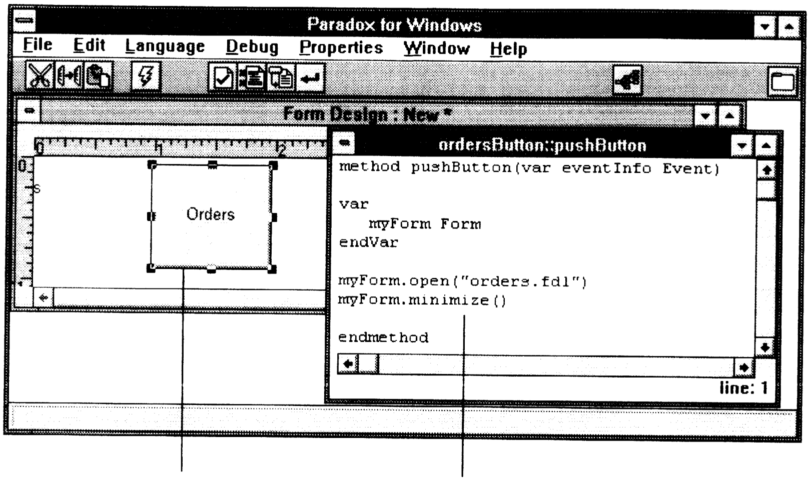
As you develop applications, you'll use one set of methods to operate on tables and another set to operate on text files, because tables and text files are objects of different types.

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.

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 2-2.

Figure 2-2 Specifying objects and actions (Hey you, do this)



```
method pushButton(var eventInfo Event)

var
   myForm Form
endVar

myForm.open("orders.fdl")
myForm.minimize()

endmethod
```

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 warrrant 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 a table.

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.

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 two other questions:

1. When should the code execute?
2. How many objects will use this code?

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. Here are some examples of 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.

Built-in methods execute in response to events.

When you interact with an object, you generate an event. 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 the *ObjectPAL Reference.)*

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 5. Table 2-1 lists some general guidelines.

Table 2-1 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
Many objects, many forms	In a library

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).

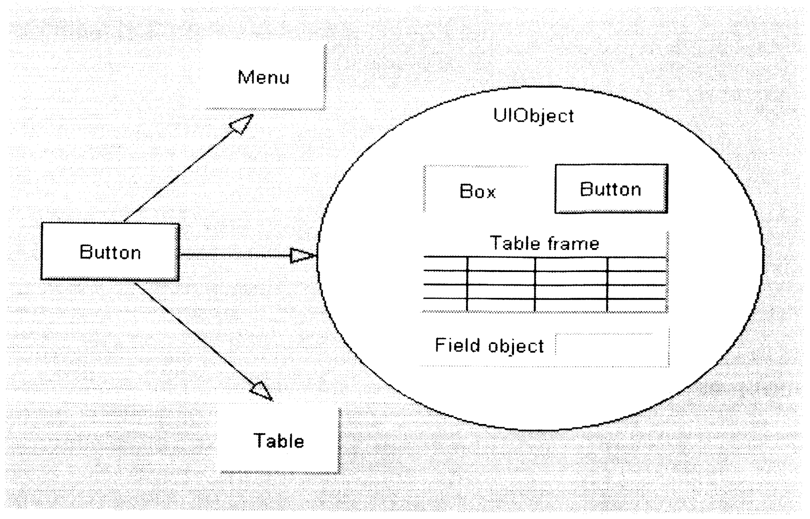
In most cases, an object and its type have the same name: Table type methods operate on tables, String type methods operate on strings, Menu type methods operate on menus, and so on. The exception is the UIObject type, which includes methods that operate on buttons, boxes, fields, table frames, multi-record objects, and the rest of the objects you can create using tools in the SpeedBar.

Note Pages in a form are UIObjects, and the form itself can behave as a UIObject. See Chapter 7 for more information and a complete list of UIObjects.

The type of object you're operating on dictates which methods to use.

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 2-3 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.

Object categories

This manual groups types into the categories listed in Table 2-2.

Table 2-2 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, Currency, Date, DateTime, DynArray, Graphic, Logical, LongInt, Memo, Number, OLE, Point, SmallInt, String, Time
Data model objects	Database, Query, Table, TCursor
System data objects	DDE, FileSystem, Library, Session, System, TextStream

Figure 2-4 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 the figure, 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 keypress), 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 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 2-4 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 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.

Data types

At the next level are 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, 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

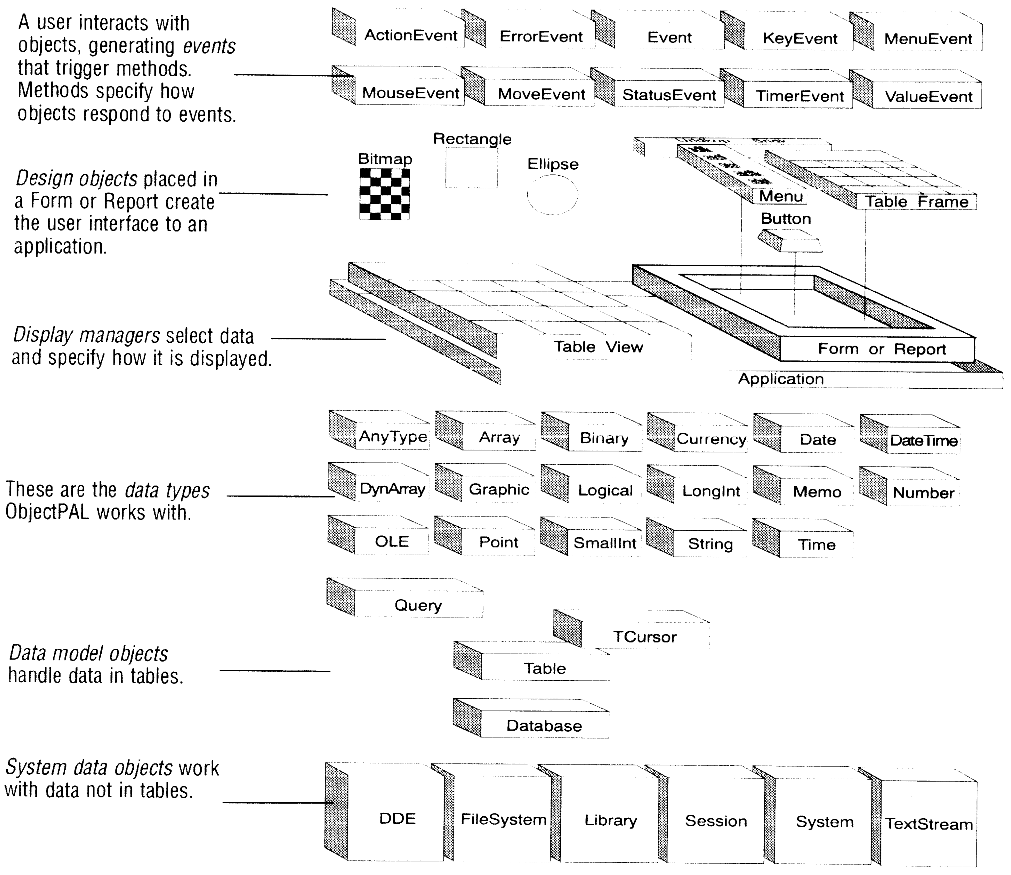
The *System data objects* are at the bottom of the figure. These objects store and access data about Windows, DOS, user counts, and the user's computer system, but *not* data stored in tables.

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.

Figure 2-4 Components of an application

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


A user interacts with objects, generating *events* that trigger methods. Methods specify how objects respond to events.
ActionEvent
ErrorEvent
Event
KeyEvent
MenuEvent
MouseEvent
MoveEvent
StatusEvent
TimerEvent
ValueEvent
Design objects placed in a Form or Report create the user interface to an application.
Bitmap
Rectangle
Ellipse
Menu
Table Frame
Button
Display managers select data and specify how it is displayed.
Table View
Form or Report
Application
These are the *data types* ObjectPAL works with.
AnyType
Array
Binary
Currency
Date
DateTime
DynArray
Graphic
Logical
LongInt
Memo
Number
OLE
Point
SmallInt
String
Time
Data model objects handle data in tables.
Query
TCursor
Table
Database
System data objects work with data not in tables.
DDE
FileSystem
Library
Session
System
TextStream

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 2-3 summarizes the functions of these objects.

Table 2-3 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 Chapter 10.
- Table frames, multirecord objects, and field objects are UIObjects and are described in Chapter 7.
- TableViews are described in Chapter 8.
- The following example applications:
  - Dive Planner (MAST)
  - 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 10.
- 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 sections:

- Entries for **msgAbortRetryIgnore**, **msgInfo**, **msgQuestion**, **msgRetryCancel**, **msgStop**, **msgYesNoCancel** in the "System" section of Chapter 11
- The section "Multi-form applications" in Chapter 8

Keyboard events

You can respond to any keypress in ObjectPAL, which means you can easily develop shortcut keys for your application. See the following sections and appendixes:

- The discussion of the event model in Chapter 6
- **keyPhysical** and **keyChar** in Chapter 2 in the *ObjectPAL Reference*
- The ANSI character set in Appendix B in the *ObjectPAL Reference*
- The list of Windows keycodes in Appendix C in the *ObjectPAL Reference*
- The Checkbook application in the Examples

Mouse events

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

- The "MouseEvent" section of Chapter 6
- The following example applications:
  - Dive Planner
  - Address Book
  - Paradox Paint

Menus

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

- The "MenuEvent" section of Chapter 6

- "Menu" and "PopUpMenu" in Chapter 7
- The following example applications:
    - Dive Planner
    - 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

- "Using tables and TCursors to program lists" in Chapter 10

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 sections:

- "Form" and "Multi-form applications" in Chapter 8

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 9
- "Memo" in Chapter 9
- "TextStream" in Chapter 11

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 sections:

- "FileSystem" in Chapter 11
- The description of the System type procedure **fileBrowser** in Chapter 11

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 sections:

- The "Library" section of Chapter 11
- Chapter 3 in the *ObjectPAL Reference*

DLLs

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

- ❐ Chapter 3 in the *ObjectPAL Reference*

CHAPTER 3

The ObjectPAL Editor

The ObjectPAL Editor provides the functions of a Windows text editor (like Notepad), along with special functions (like a syntax checker) for editing ObjectPAL methods. In the ObjectPAL Editor, you can display dialog boxes that list the methods for each object type, the syntax for each method, properties and property values for each object, and constants for things like window attributes and error codes. The Editor has close ties 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 errors so you can correct them before you try to run the application.

The Editor also works with the Debugger (described in the next chapter) to provide an integrated environment for creating, testing, and modifying methods. You can edit your methods, debug your code, and use the Object Inspector to display the methods, keywords, actions, and constants supported by ObjectPAL.

You can use another editor.

If you prefer to use another editor, you can. Refer to the discussion of the Properties menu, later in this chapter.

This chapter describes

❒ Starting the Editor

❒ Working with the Editor

❒ Leaving the Editor

❒ Using shortcuts

Starting the Editor

To start the Editor, inspect an object, then choose Methods from its property list. The Methods dialog box (Figure 3-1) lists the methods you can edit. The Methods dialog box also includes the items Var (for declaring variables), Const (for declaring constants), Type (for declaring data types), Procs (for declaring procedures), and Uses (for declaring external routines). Each of these items has its own Editor window—you open it by choosing the item, then choosing OK. You can open as many windows as your system allows, in any order.

Shortcut To open the Methods dialog box, select an object, then press *Ctrl+Spacebar.*

Choose a method (for example, **open**), then choose OK (or just double-click the method). An Editor window appears, similar to the window shown in Figure 3-2, containing some default text.

You can have several Editor windows open at once.

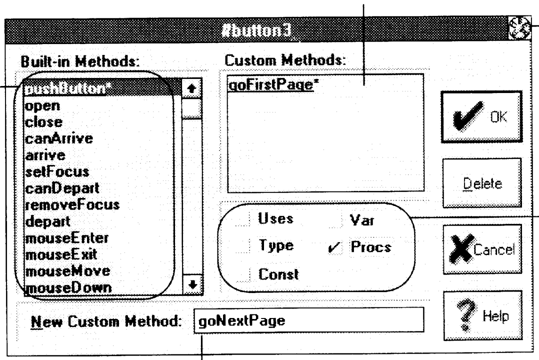
You must edit each method in its own window; however, you can edit more than one method at a time by opening multiple windows. To do so, select the methods you want to edit, then choose OK. To select more than one method, use *Shift+*click and *Ctrl+*click, or drag in the Methods dialog box. When you choose OK, the Editor opens a window for each method you select. For example, if you select the **open** and **close** methods, two windows appear when you choose OK, one for each selected method.

Figure 3-1 The Methods dialog box

To edit an existing custom method, choose one from the Custom Methods list. In this figure, **goFirstPage** is an existing custom method.

To edit built-in methods, choose one or more items from this list. In this figure, the **pushButton** method is chosen.

You can choose one or more items from the Methods dialog box, then choose OK to open several Editor windows at once



Click here to snap the dialog box in place and keep it open as you work with a number of objects. Click here again when you're ready to close it.

Check one or more of these items to open one or more separate Editor windows. In this figure, Procs is chosen.

To define a new custom method, type the name in the New Custom Method box. In this figure, **goNextPage** is the name of a new custom method.

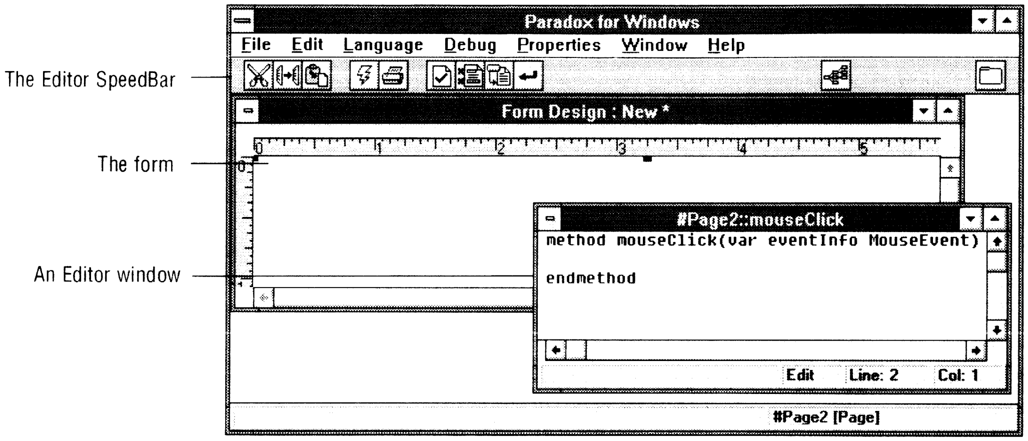
If you're editing the **open** method, the first line reads **method open (var eventInfo Event)**, the second line is blank, and the third line reads **endmethod**. If you accidentally change the default text, you can edit it as you would any other text.

The Editor is always in insert mode.

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 window are visible only to that method. To make a variable, constant, or procedure visible to all of an object's methods, open a separate window. This is called variable scoping. For more information about variable scoping, see Chapter 5.

Figure 3-2 The Editor



```
method mouseClick(var eventInfo MouseEvent)

endmethod
```

Working with the Editor

The following sections describe how to work with the Editor by using the Editor menus and the keyboard.

This section describes the items in each Editor menu, special Editor keys, and how to select text.

When you open an Editor window, the application menu bar changes to provide the code-editing functions described in the following sections.

Note See the "Shortcuts" section later in this chapter for a description of the SpeedBar buttons and keyboard shortcuts available in the Editor and the Debugger.

File menu

Items in the File menu 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.

Edit menu

This section describes the items in the Edit menu.

Undo	Undoes your last edit.
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 on the Clipboard.
Delete	Deletes selected text. Delete is dimmed if no text is selected.
PasteFromFile	Lets you specify a text file to paste into the current method at the insertion point.
CopyToFile	Copies selected text—*not* the contents of the Windows Clipboard—to a text file you specify.
Search	Searches the text from the insertion point to the end of the method, or to the beginning. You can turn case matching on and off by choosing a check box.
Search 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	Searches for text and replaces it with the given replacement value.
Replace Next	Replaces the next occurrence of text specified in Replace. Replace Next is dimmed until something is replaced.
Go To	Moves to a specific line. Choose Edit | Go To to display a dialog box, then enter a line number and choose OK. If the line number doesn't exist, the insertion point won't move.
Select All	Selects all text in the window.

Language menu

This section describes the items in the Language menu.

Check syntax	Checks the syntax of every method in the form (not just the method in the active Editor window). If syntax errors are found, a window opens for the corresponding method, with the insertion point positioned near the error. An error message appears in the Status line.
Note	If you change the code in more than one Editor window, save the changes before you check the syntax. Otherwise, the syntax checker

might report unexpected errors because it's not checking the latest code.

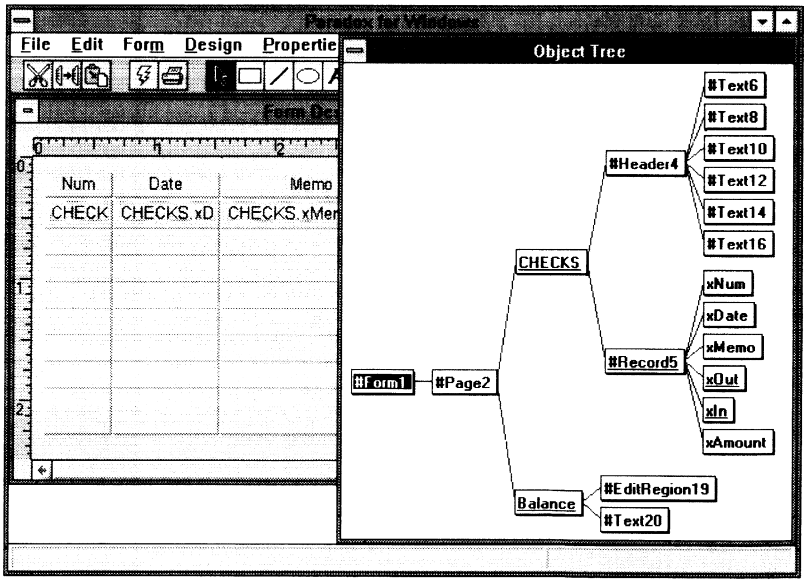
Next Warning Displays the next warning found by the compiler.

Methods Displays the Methods dialog box.

Object Tree Displays a tree diagram showing how objects in the current form are related. 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 3-3.

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 its Methods dialog box. You can use the mouse or arrow keys to move around in the Object Tree.

Figure 3-3 The Object Tree

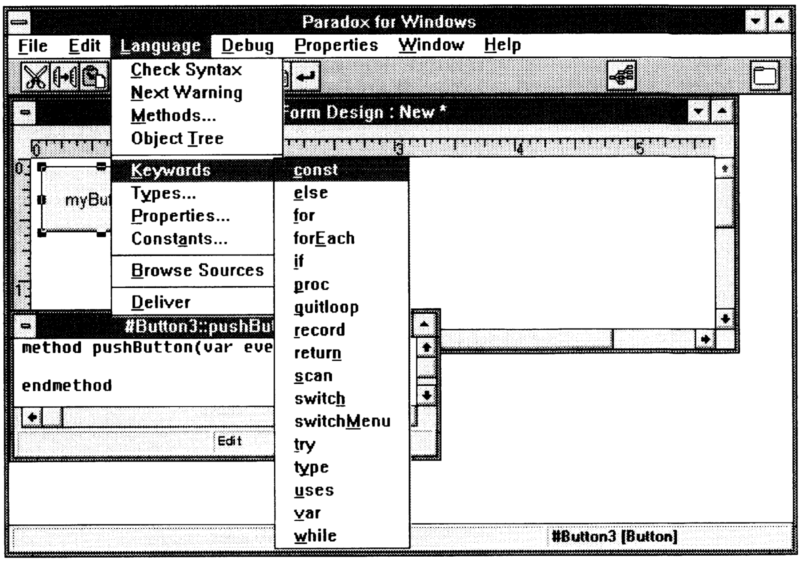


Keywords Displays a cascading menu of frequently used words that are reserved by Paradox. Use this menu to insert a keyword into a method without having to type it. The keywords are basic language elements, described in the *ObjectPAL Reference.*

Figure 3-4 shows the menu that appears when you choose Language | Keywords.

Figure 3-4 The Keywords menu

Choosing an item from the Keywords menu inserts it into your method at the insertion point



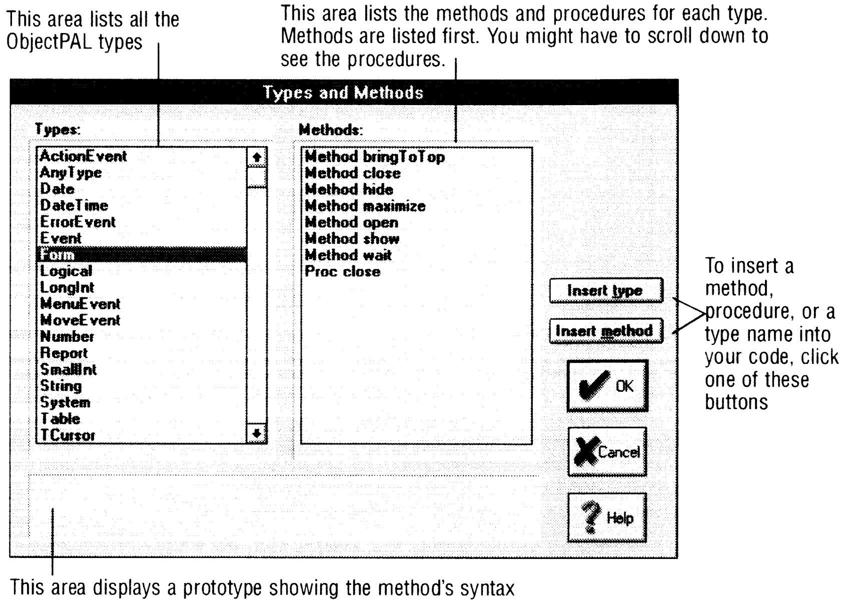
```
method pushButton(var eve

endmethod
```

Types Displays 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 3-5 shows the dialog box that appears when you choose Language | Types.

Figure 3-5 The Types dialog box


This area lists all the ObjectPAL types
This area lists the methods and procedures for each type. Methods are listed first. You might have to scroll down to see the procedures.
Types and Methods
Types:
ActionEvent
AnyType
Date
DateTime
ErrorEvent
Event
Form
Logical
LongInt
MenuEvent
MoveEvent
Number
Report
SmallInt
String
System
Table
TCursor
Methods:
Method bringToTop
Method close
Method hide
Method maximize
Method open
Method show
Method wait
Proc close
Insert type
Insert method
OK
Cancel
Help
To insert a method, procedure, or a type name into your code, click one of these buttons
This area displays a prototype showing the method's syntax

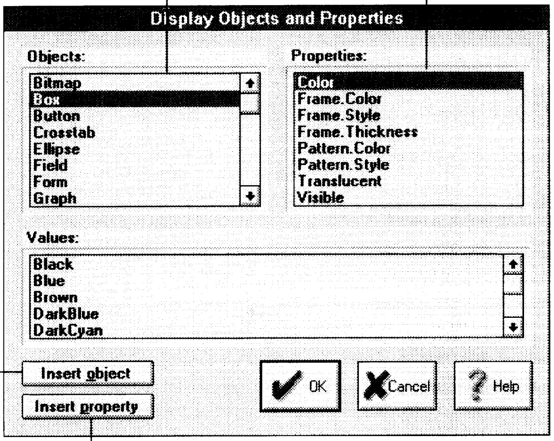
Properties Displays a dialog box listing objects and their properties. You can choose an object name to display the properties for that object. Then you can insert the property into your method. For example, to display a list of Button properties, choose Button.

Figure 3-6 shows the dialog box that appears when you choose Language | Properties.

Figure 3-6 The Display Objects and Properties dialog box

Choose an object from the Objects column to list its properties in the Properties column

Choose a property from the Properties column to list valid values in the Values column



Choose this button to insert the object name into your method

Choose this button to insert the property into your method

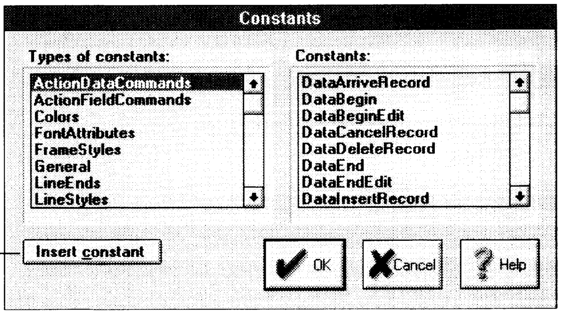
Constants Displays a dialog box that lists ObjectPAL constants. ObjectPAL provides many constants so you can easily specify things like colors, mouse shape, menu attributes, and window styles. You can choose a constant and choose Insert to insert it into your code.

Figure 3-7 shows the dialog box that appears when you choose Language | Constants.

Figure 3-7 The Constants dialog box

ObjectPAL constants are grouped into categories according to when and how they are used

The Types of constants column lists the categories. Choose a category name from this column to display a list of constants in the Constants column.



Choose this button to insert the selected constant into your method

Browse Sources Creates and displays a report listing the source code in the form or report you're working on. Paradox stores the data for this report in a table named PAL$SRC.DB in your private directory.

Deliver Creates a compiled form, library, or script, stripped of its ObjectPAL source code. Users cannot edit the design or the source code. For more information about delivering objects, see Chapter 15.

Debug menu

The Debug menu displays a list of Debugger functions. These functions are discussed in the next chapter.

Properties menu

This section describes the items in the Properties menu.

Desktop Displays a dialog box for setting the properties of the Desktop window. Refer to the *User's Guide* for information about this dialog box.

Window Sizing Sets the default size of Editor and Debugger windows. To use this function, first size the window. Then choose Properties | Window Sizing to display a dialog box containing two choices: **Use current size from now on** and **Return to default sizing**.

Alternate Editor Displays a dialog box where you can specify another editor to use to write and edit methods. (The editor you use must be able to save text-only files, without any formatting codes.) Enter the full path to your editor of choice, for example, C:\APPS\BRIEF\B.EXE.

When you use an alternate editor, you don't have access to the syntax checker or other online information provided in the ObjectPAL Editor, but you can switch between editors.

After you specify an alternate editor, a dialog box appears each time you edit a method. You can choose Yes to open the alternate editor, choose No to open the ObjectPAL Editor, or choose Cancel to cancel the operation. When you specify an alternate editor, your choice remains in effect until you exit Paradox.

Show Compiler Warnings When this item is checked, messages in the Status line 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.

Window menu

Items in the Window menu operate as described in the *User's Guide.*

Help menu

Items in the Help menu operate as described in the *User's Guide.*

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 3-1, 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+Ins* copies selected text to the Clipboard, and *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 the mouse, using the arrow keys with *Shift* held down, or clicking with *Shift* held down to extend the selection. You can 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.) You can select a word by double-clicking it.

When text is selected, typing a character (or pasting from the Clipboard) replaces the selected text with whatever you type or paste.

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 form).
  - Click the Save Source and Exit SpeedBar button.
- To view your form and see it in action, click the View Data SpeedBar button.

- ❒ To run your code immediately, click the Run SpeedBar button or choose Debug | Run.

If you've made changes to your code, but not saved them, Paradox displays a confirmation dialog box that lets you save or cancel your changes.

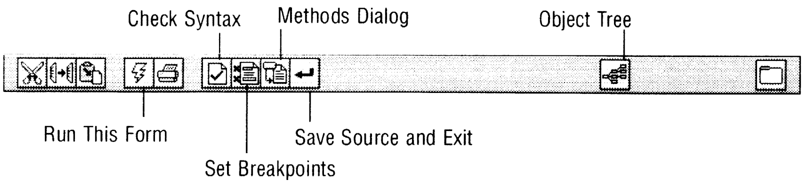
Shortcuts

The ObjectPAL Editor provides SpeedBar buttons and support for the right mouse button as shortcuts for common actions. Figure 3-8 shows the Editor SpeedBar buttons.

SpeedBar buttons

- ❒ Run This Form has the same effect as choosing Form | View Data.
- ❒ Check Syntax has the same effect as choosing Language | Check Syntax.
- ❒ Set Breakpoints has the same effect as choosing Debug | Set Breakpoint.
- ❒ Methods Dialog has the same effect as choosing Language | Methods.
- ❒ Save Source and Exit has the same effect as choosing Close from the window's Control menu.
- ❒ Object Tree has the same effect as choosing Language | Object Tree.

Figure 3-8 The Editor SpeedBar buttons



Using the right mouse button

When you edit a method, click the right mouse button to display a pop-up menu. This menu lists the same items as the Language menu.

When you debug a method, click the right mouse button to display a different pop-up menu. This menu lists the same items as the Debug menu (refer to the next chapter for more information about using the Debugger).

Editor and Debugger keyboard shortcuts

Table 3-1 lists keyboard shortcuts you can use in the Editor and the Debugger.

Table 3-1 Editor and Debugger keyboard shortcuts

Function-key shortcuts	Keyboard shortcuts	Function
F1		Help
F2		Accept changes
Shift+F2		Accept changes and close window
	Ctrl+Z	Search
	Ctrl+Shift+Z	Search and replace
	Ctrl+A	Search again
F3	*Ctrl+I*	Inspect variable (Debugger)
Ctrl+F3	*Ctrl+B*	Set breakpoint (Debugger)
Shift+F3	*Ctrl+K*	Stack backtrace (Debugger)
F5	*Ctrl+G*	Goto line
Ctrl+F5	*Ctrl+W*	Next Warning
F6	*Ctrl+M*	Language menu
F7		Step over (Debugger)
Shift+F7		Step into (Debugger)
F8		Run (Run to breakpoint if debugging)
Shift+F8		Save form to disk and run
F9	*Ctrl+Y*	Check syntax (Compile)
Ctrl+F9	*Ctrl+D*	Deliver form
	Ctrl+Ins	Copy
	Shift+Del	Cut
	Shift+Ins	Paste
	Ctrl+Spacebar	Open Methods dialog box
	*Ctrl+*Left Click	Select all

CHAPTER 4

The ObjectPAL Debugger

This chapter explains how to start the Debugger, work with it, and exit. 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 3.

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. This is where the Debugger is helpful. 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 window that lists each line of code as it executes.
- Inspect variables to make sure that values are being manipulated as you intended.

- Execute a method one line at a time (called *single-stepping*).
- Step over methods and procedures that you know are bug-free.

Using the Debugger

This section describes how to use the Debugger.

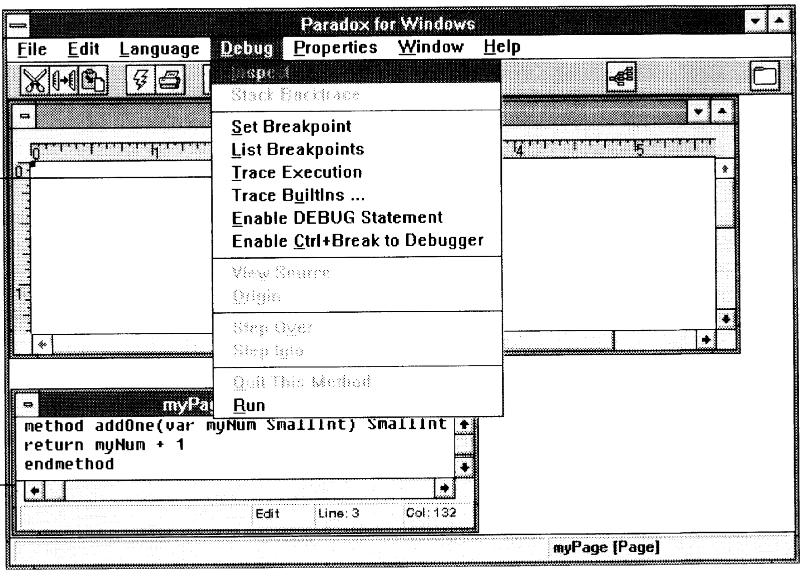
Starting the Debugger

First, display a method in an Editor window. Then, choose Debug to display the drop-down menu of Debugger functions shown in Figure 4-1.

Figure 4-1 The Debugger menu

The Debugger menu. Dimmed menu items become available when method execution suspends at a breakpoint.

An Editor window



```
method addOne(var myNum SmallInt) SmallInt
return myNum + 1
endmethod
```

Debug menu

This is what you can do using the Debug menu:

- *Inspect* displays and optionally changes the value of a variable. Available only when execution stops at a breakpoint.
- *Stack Backtrace* lists the methods and procedures called since your form or report started running. 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. Available only when execution stops at a breakpoint.

- *Set Breakpoint* sets breakpoints in a method to halt execution at specified lines. You can set as many breakpoints as your system memory allows.

Note All breakpoints are deleted when you change your code.

- *List Breakpoints* displays a dialog box listing the line numbers of breakpoints. It also lets you delete breakpoints.
- *Trace Execution* opens a window and lists each line of ObjectPAL code as it executes if this item is checked. Your setting for this item is saved with your form or report, so you don't have to check it every time you want to trace execution. When this item is not checked, execution procedes normally.

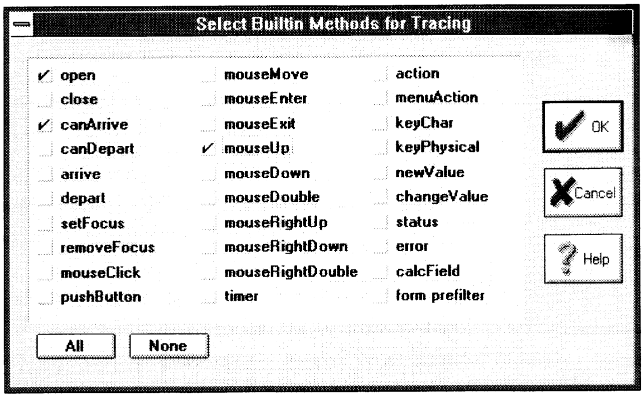
  The ObjectPAL Tracer performs this operation. For an example of how to use the ObjectPAL Tracer, see Chapter 7. ObjectPAL also provides procedures for controlling the ObjectPAL Tracer. Refer to the "System" section of the *ObjectPAL Reference* for details and examples.

- *Trace Built-ins* displays a dialog box listing all the built-in methods (shown in Figure 4-2). Check a built-in method to display information about the method in the Tracer window as it executes. You must have ObjectPAL code attached to at least one object for this option to take effect.

Note Checking boxes in this dialog box does not automatically turn on the Tracer. You must still choose Debug | Trace Execution (described previously), or else choose Debug | Enable Debug (described next) and use the **tracerOn** procedure in ObjectPAL code attached to an object.

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 4-2 The Trace Builtins dialog box



✓ open	mouseMove	action
close	mouseEnter	menuAction
✓ canArrive	mouseExit	keyChar
canDepart	✓ mouseUp	keyPhysical
arrive	mouseDown	newValue
depart	mouseDouble	changeValue
setFocus	mouseRightUp	status
removeFocus	mouseRightDown	error
mouseClick	mouseRightDouble	calcField
pushButton	timer	form prefilter

Checking a built-in method in this dialog box indicates that you want that method traced, whether or not it has any ObjectPAL code attached to it. 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 6 for information about the event model and the **isPreFilter** method.

- *Enable DEBUG Statement* stops 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, so you don't have to keep resetting it as you would a breakpoint. The setting for this item is saved with your form or report.

  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.

  When this item is not checked, DEBUG statements are ignored. Be sure to uncheck this item before delivering a form or report. Otherwise, the DEBUG statements will execute when your user runs the form or report.

- *Enable Ctrl+Break to Debugger* suspends execution and opens a Debugger window if you press *Ctrl+Break* when this item is selected. The Debugger window contains the active method or procedure, just as if a breakpoint had been encountered.

  When this item is not checked, pressing *Ctrl+Break* halts execution.

Note *Ctrl+Break* halts execution only of ObjectPAL methods and procedures. Other operations (for example, queries) are not affected.

- ❐ *View Source* displays another method's code in an ObjectPAL Editor window. This is a quick way to get to a specific method for a specific object. You can use the mouse or arrow keys to scroll through items in this dialog box, even though it has no scroll bars.
- ❐ *Origin* displays the method containing the current breakpoint, with the insertion point on the line containing the breakpoint. When execution suspends at a breakpoint, an Editor window displays the method containing the breakpoint.

  You can use *Origin* to return quickly to the breakpoint when you have several Editor windows open on your screen. This function is available only when execution is suspended at a breakpoint.
- ❐ *Step Over* single-steps through a method, treating procedures and custom methods as single steps. This function is available only when execution is suspended at a breakpoint.
- ❐ *Step Into* single-steps through every line in a method and every line in the procedures and custom methods the method calls. This function is available only when execution stops at a breakpoint.
- ❐ *Quit This Method* halts execution and closes any Debugger windows. This function is available only when execution is suspended at a breakpoint.
- ❐ *Run* exits the Debugger, saves changes, and switches from a design window to the View Data choice. If execution is suspended at a breakpoint, Debug | Run resumes from the breakpoint.

Note When breakpoints are set, or when Trace execution, Enable DEBUG, or Enable Ctrl+Break to Debugger are checked, execution speed is slower than normal.

Leaving the Debugger

There is no explicit command for leaving the Debugger. You can get back to your form or report by choosing Debug | Run, or by clicking the underlying form or report. To exit the Debugger, close the Debugger window.

A Debugger tutorial

This section provides a brief, hands-on tutorial for the Debugger. In it, you'll learn how to

- ❐ Check your methods for syntax errors

❐ Add and delete breakpoints

❐ Inspect variables

❐ Step through sections 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. Several dialog boxes appear. In the appropriate dialog box, choose OK to specify a blank form (not attached to any tables).
2. Inspect the page to display its menu, then name the page *myPage.*
3. Inspect *myPage* and choose Methods to open the Methods dialog box.
4. Choose the edit box labeled New Custom method. Type **addOne,** then choose OK.
5. 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. When 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 Methods dialog box.
10. Choose **pushButton**, then choose OK to open an Editor window (or double-click **pushButton**).
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. Choose Language | Check Syntax to check for syntax errors, and make corrections as needed.
13. Close the Editor window and save the new **pushButton** method.

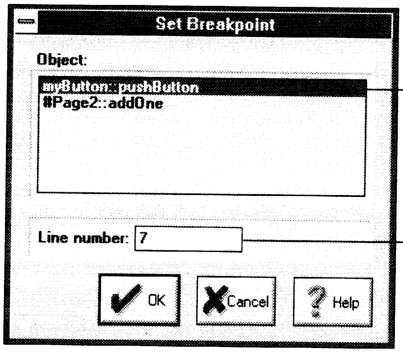
Setting breakpoints

Breakpoints are useful for stopping 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. When the syntax is correct, save your work and run the form.
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 switch back to a 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. Choose Debug | Set Breakpoint. A dialog box appears, listing the objects in the current form and a line number. By default, the line number is the line the insertion point is on, but you can type a line number directly.
7. Choose OK to set a breakpoint at line 7.

Figure 4-3 Setting a breakpoint



myButton::pushButton
#Page2::addOne

Line number: 7

Choose a method name from this list to set a breakpoint in that method

Use this edit box to specify the line where you want to set the breakpoint

8. Choose Debug | Run to leave the design window, then click the button *myButton*. The **pushButton** method executes until it reaches the line where you set the breakpoint, and then displays the method in an Editor window.
9. Choose Debug | Run (or choose Run This Form from the SpeedBar) to execute the remaining lines in the method. Execution resumes with the line containing the breakpoint.

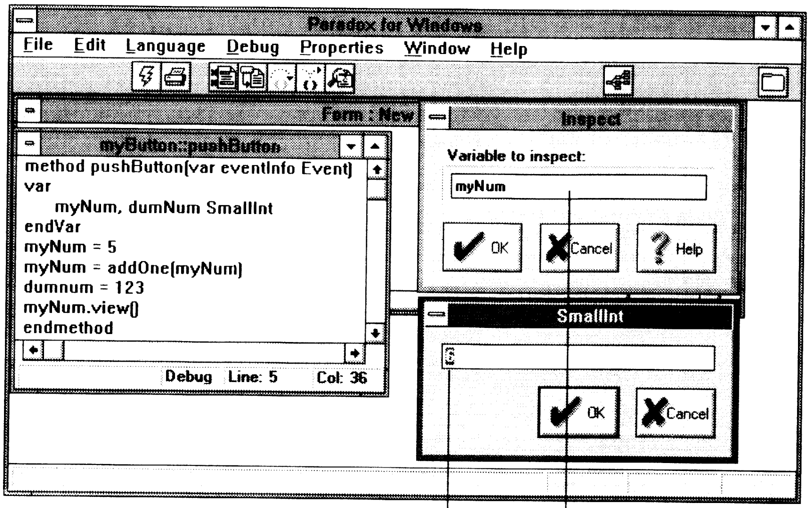
Inspecting a variable

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. Click the button *myButton* to execute the **pushButton** method from the beginning. Execution suspends when it reaches the breakpoint.
2. Move the insertion point into the word *myNum* (any *myNum* will do).
3. Choose Debug | Inspect. A dialog box displays the name of a variable to inspect. By default, it displays the word the insertion point is in (whether or not it's a variable), in this case, *myNum*. You can also drag the mouse to select an item to inspect.
4. Choose OK. 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.
5. Again, choose Debug | Inspect. Type **dumNum** in the text box, then choose OK. The dialog box displays **N/A**, 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.)

Figure 4-4 Inspecting a variable



```
method pushButton(var eventInfo Event)
var
    myNum, dumNum SmallInt
endVar
myNum = 5
myNum = addOne(myNum)
dumnum = 123
myNum.view()
endmethod
```

A dialog box appears containing the value of the specified variable

Specify here which variable to inspect, then choose OK

Deleting breakpoints

It's best to delete breakpoints as soon as you're done with them. To delete the breakpoint you just added,

1. Choose Debug | List Breakpoints to display a dialog box listing all breakpoints in all methods in the form. (This example lists one method and one breakpoint.)
2. Choose the breakpoint, then choose Delete to delete it.

Note Breakpoints are deleted when you change your code.

Stepping into and stepping over

You can also use breakpoints to step through each line of code to 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. To step into or over procedures, do one of the following:

- ❐ Choose Debug | Step Into when you want to suspend execution at a breakpoint in a custom method or procedure.

- ❐ Choose Debug | Step Over to treat the procedure or custom method as a single instruction.

The same code executes in either case.

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.

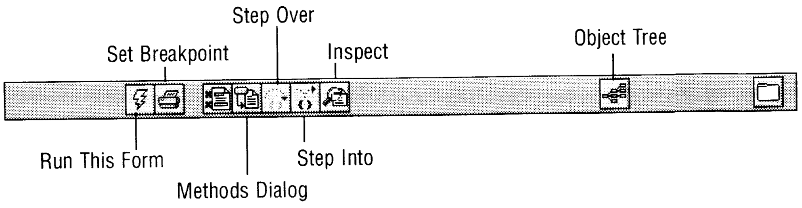
Shortcuts

The ObjectPAL Editor provides SpeedBar buttons and support for the right mouse button as shortcuts for common actions.

SpeedBar buttons

- ❐ *Run This Form* has the same effect as choosing Debug | Run from the menu.
- ❐ *Set Breakpoint* has the same effect as choosing Debug | Set Breakpoint.
- ❐ *Methods Dialog* has the same effect as choosing Language | Methods.
- ❐ *Step Over* has the same effect as choosing Debug | Step Over.
- ❐ *Step Into* has the same effect as choosing Debug | Step Into.
- ❐ *Inspect* has the same effect as choosing Debug | Inspect.
- ❐ *Object Tree* has the same effect as choosing Language | Object Tree.

Figure 4-5 The Debugger SpeedBar



Using the right mouse button

When you debug a method, click the right mouse button to display a pop-up menu. This menu lists the same items as the Debug menu.

When you edit a method, click the right mouse button to display a pop-up menu that lists the same items as the Language menu.

CHAPTER 5

Language structure

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

- ❒ The nature of ObjectPAL
- ❒ Dot notation
- ❒ Structure
- ❒ Control structures
- ❒ Data types
- ❒ Expressions
- ❒ Naming rules
- ❒ ObjectPAL terms
- ❒ Containers
- ❒ How to use variables
- ❒ How to define constants
- ❒ How to pass arguments to methods and procedures

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**, 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**.

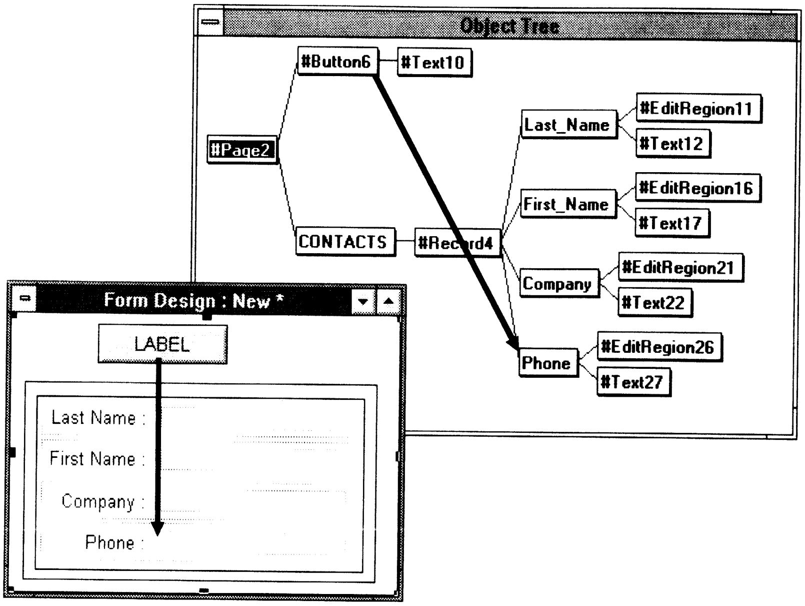
In text (as distinguished from code examples), this manual refers to methods and procedures by name only; it does not give the full syntax. For example, the previous paragraph refers to the **setTitle** method defined for the Form type. The **setTitle** method's complete syntax is

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

Refer to the *ObjectPAL Reference* for the complete syntax for every ObjectPAL method and procedure.

For a more complex example of dot notation, see Figure 5-1, which shows a form containing a button and a table frame bound to the *Contacts* table. The figure also shows the Object Tree, which diagrams the relationships between the objects.

Figure 5-1 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"

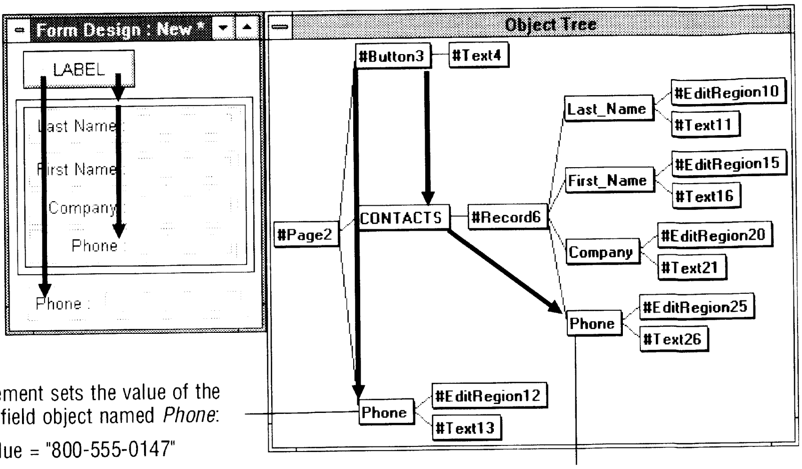
Code attached to the button's **pushButton** method can use the following statement to set the value of the *Phone* field object in the *Contacts* table:

```
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 5-2 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 5-2 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
```

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 methods have 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", "Nevil Patel") then
   tc.ProductName = "Paradox"
else
   tc.ProductName = "dBASE"
endIf
```

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

Case

ObjectPAL is case-insensitive.

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 (alphanumeric) data values. For example, the string "abc" is not equal to "ABC".

Note You can use **ignoreCaseInStringCompares**, defined for the String type, to specify if 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 Chapter 3 in the *ObjectPAL Reference.*

Data types

The ObjectPAL data types are listed in Table 5-1. These data types, as well as more complex types, are described in Chapter 9.

Table 5-1 ObjectPAL data types

Type	**Contains**
Alphanumeric	Up to 32,767 characters for a String variable, up to 255 characters for a quoted string
Currency	Money values ranging from ±3.4 * 10 $^{-4930}$ to ±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

Type	Contains
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 $\pm3.4 * 10^{-4930}$ to $\pm1.1 * 10^{4930}$
SmallInt	Integer values ranging from -32,768 to 32,767 (2 bytes)*
Time	Clock data in the form hour-minute-second-millisecond

* -32,768 cannot be stored in a Paradox table, because to Paradox, -32768 = Blank.

Table 5-2 shows how to combine and convert values of different types.

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 5-2 Combining data types

	M	A	E	T	D	C	N	L	S	&
M (Memo)	M	*	*	*	*	*	*	*	*	*
A (String)	*	A	*	*	*	*	*	*	*	*
E (DateTime)	*	*	E	E	E	*	E	*	E	*
T (Time)	*	*	E	T	T	*	T	*	*	*
D (Date)	*	*	E	T	D	*	D	D	D	*
C (Currency)	*	*	*	*	*	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)	*	*	*	*	*	*	*	*	*	&

* not allowed

Binary, Graphic, and OLE types cannot be combined, so they are omitted from this table.

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 3.65 to myNum
myNum = SmallInt(3.65)      ; Changes Number 3.65 to SmallInt, returns 3
myNum = SmallInt("12.67")   ; Changes String "12.67" to SmallInt, returns 12

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 5-3 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 9.

Note The (=) symbol also functions as an assignment operator. For more information and examples, refer to Chapter 3 in the *ObjectPAL Reference*.

Table 5-3 ObjectPAL operators for data types

Data Type/Category	Operator	Function
Alphanumeric (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 variables have been assigned:

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

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 alphanumeric fields and memo fields.

```
message("Income" + " Tax")    ; "Income Tax" (concatenation)
message("Income" + Deathly)   ; "Income Tax" (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)            ; 1/2/92 (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") ; 12:39:12 AM, 12/12/3812
```

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

```
message(HighNoon + Date("01/01/01") ; 12:39:12 AM, 12/12/3812
```

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

```
message(HighNoon + Date("01/01/01") ; 12:39:12 AM, 12/12/3812
```

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)        ; 28-Dec-1991 (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 5-4 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 5-4 Comparison operators

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

Alphanumeric 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). Blank values are considered to be less than all other values of the same data type.

You can compare alphanumeric (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
message(3 = "ABC")              ; False, since the types are mixed
message(3 <> "ABC")             ; True, since the types are mixed
```

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 5-3:

- 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)               ; Syntax  error; not compatible data types
message(3 <> 4)                  ; Syntax error; not compatible types
```

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 *ObjectPAL Reference* 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 5-5.

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 5-5 Operator precedence

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

Table 5-6 lists other symbols used in ObjectPAL expressions.

Table 5-6 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 5-7 lists pattern-matching symbols for working with character strings.

Table 5-7 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 methods, procedures, variables, and arrays

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.

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

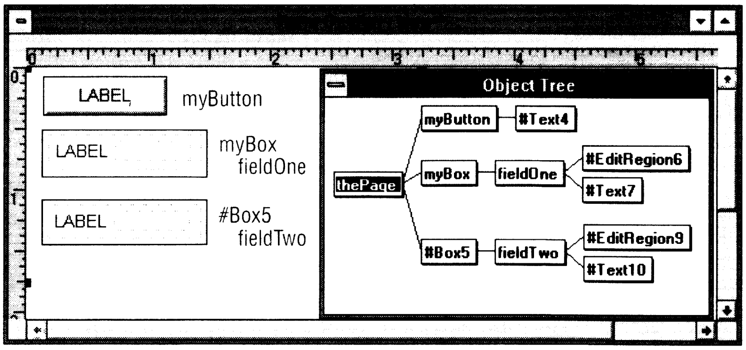
Default names

Figure 5-3 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 5-3 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



Methods 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 also bypass *#Box5* and address *fieldTwo* directly, like this:

```
fieldTwo.value = 755
```

Because *#Box5* is the object's default name, it's 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).

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"
```

Spaces in names

If an object gets its name from a field in a table, and if that name contains a space, Paradox replaces the space with an underscore. For example, suppose the *Parts* table contains a field named "Part num." In a form, that field object's name would be *Part_num*. You would set its value like this:

```
PARTS.Part_num.value = "AB123-55"
```

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 or names of object types, events, methods, or properties. See the list of reserved words in Appendix D in the *ObjectPAL Reference*.

Table 5-8 shows examples of valid and invalid variable names.

Table 5-8 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
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
```

Although you can give an object a name beginning with "#," it's strongly recommended that you don't. The default names Paradox gives design objects always begin with "#" (for example, *#button1320*), and 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 5-11 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 5-9.

Table 5-9 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

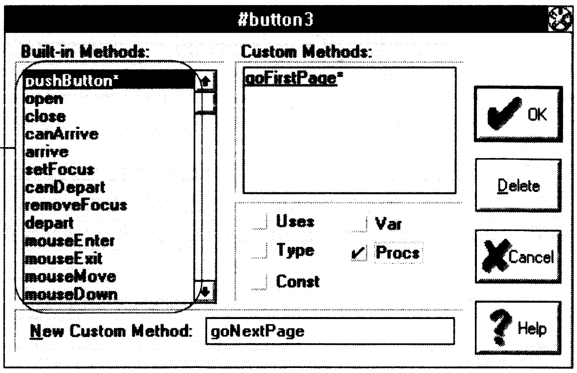
Every Paradox object comes with built-in methods (for example, **open**, **close**, and **mouseUp**) for each event it can respond to. These built-in methods specify an object's default behavior in response to a given event. You build Paradox applications by attaching ObjectPAL code to the built-in methods 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, inspect the object and select Methods from the Properties menu. Select one or more methods from the Methods dialog box (shown in Figure 3-1), then choose OK to open one or more ObjectPAL Editor windows. 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.

Note Each built-in method is described in Chapter 2 in the *ObjectPAL Reference.*

Figure 5-4 Selecting built-in methods to edit

To edit built-in methods, choose one or more items from this list. In this figure, the **pushButton** method is chosen.



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, arranged alphabetically by type, are presented in the *ObjectPAL Reference.*

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, 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, choose Methods, then choose the New Custom Method text box. In the text box, type a name for the custom method, then choose OK 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 Methods dialog box. To make changes, choose the name and open an ObjectPAL Editor window, just as you would to edit a built-in method.

Note
Custom methods attached to the form are available to all objects in the form.

To attach methods to a form, choose Properties | Form | Methods. You can also use the Object Tree. Refer to the discussion of the Object Tree in Chapter 3 for details. Another shortcut: when no objects are selected, press *Ctrl+Space.*

To share custom code among two or more forms, you must put the code in a library. See Chapter 11 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 5-10.

Table 5-10 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 next section).

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 Chapter 3 of the *ObjectPAL Reference.*

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.

To attach code to a form, choose Properties | Form | Methods. You can also use the Object Tree. See the discussion of the Object Tree in Chapter 3 for more information.

Basic language elements

Basic language elements include control structures like **if...then...else** 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 Chapter 3 in the *ObjectPAL Reference.*

Summary of differences

Table 5-11 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 5-11 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

Containers

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.

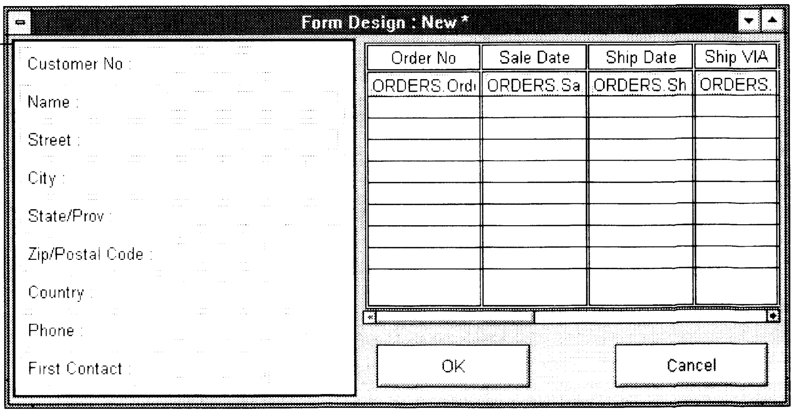
Objects in a form coexist in a hierarchy of containers. For example, when you place a table frame on a page of a form, the form contains the page, and that page contains the table. Position in this hierarchy is important because it defines what an object can get from other objects, including their methods, procedures, properties, and variables. The availability of resources follows what you see on the screen.

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 5-5, 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 5-5 Using a container to store shared code

The box contains the field objects; because of this, custom code attached to the box is available to all the field objects



Order No	Sale Date	Ship Date	Ship VIA
ORDERS.Ordi	ORDERS.Sa	ORDERS.Sh	ORDERS.

For another example, suppose a page in a form contains two buttons, as shown in Figure 5-6. 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

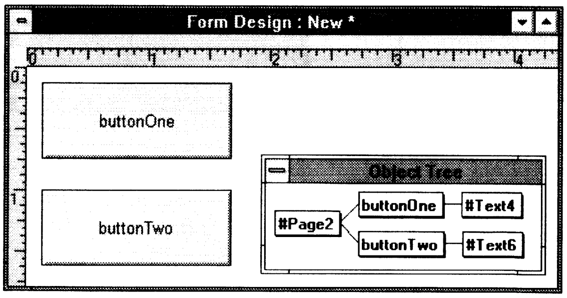
```
; buttonOne::pushButton
method pushButton(var eventInfo Event)
   theNumber = theNumber + 1
   message(theNumber)
endmethod
```

buttonTwo::pushButton method

```
; buttonTwo::pushButton
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 5-6 Containership and variables



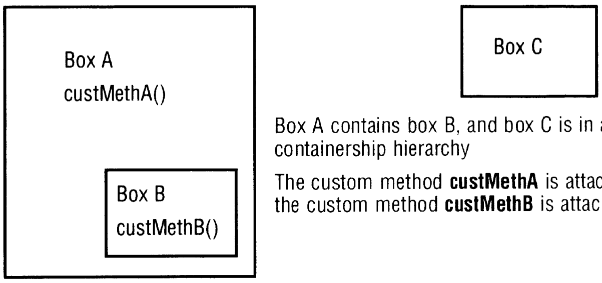
The underlying page contains both buttons, so a variable declared in the page's Var window is available to both buttons

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 methods, 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 the "Library" section of Chapter 11 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 5-7 shows three boxes, named *A*, *B*, and *C*. Box *A* contains box *B*.

Figure 5-7 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 5-7, 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.

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

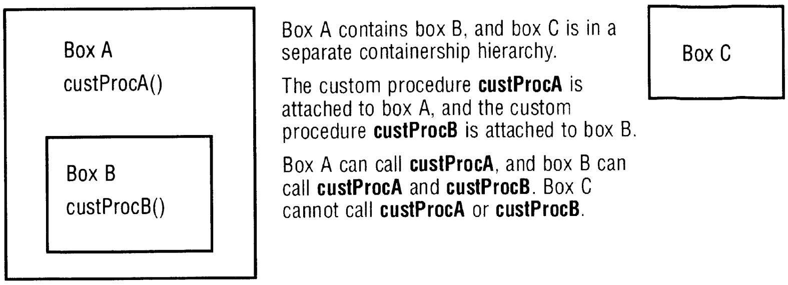
```
method custMethB()
   self.color = Red
endmethod
```

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 7.

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 5-8 shows three boxes, named *A*, *B*, and *C*. Box *A* contains box *B*.

Figure 5-8 Containership and custom procedures



Box A contains box B, and box C is in a separate containership hierarchy.

The custom procedure **custProcA** is attached to box A, and the custom procedure **custProcB** is attached to box B.

Box A can call **custProcA**, and box B can call **custProcA** and **custProcB**. Box C cannot call **custProcA** or **custProcB**.

Using the containership heirarchy shown in Figure 5-8, 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 (also called a *data 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 " + strVal(x)
```

You can use **isAssigned** to test whether a variable has been assigned a value, as in this example:

```
if myVar.isAssigned() = True 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 5-1. 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 this chapter); 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 | Show 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 myNum.isAssigned() = FALSE 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 myNum.isAssigned() = FALSE then
      myNum = 1
   else
      myNum = myNum + 1
   endIf

   message(myNum)
endMethod
```

Declared in the Var window

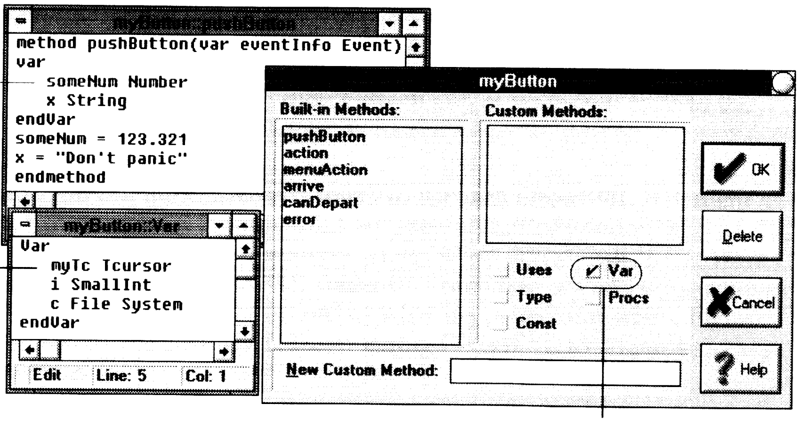
Variables declared in an object's Var window are visible to all methods attached to that object, and to any objects *that* object contains. A variable declared in an object's Var window is attached to the object and is accessible while the object exists in the form and the form is open.

To declare variables in the Var window, inspect an object, choose Methods, then choose Var and OK to display the Var window. The Var window is a text-editing window, just like the Method window. Figure 5-9 shows variables declared in a Method window and in a Var window.

Figure 5-9 Declaring variables in windows

Variables declared in a method are visible only to that method

Variables declared in the Var window are visible to all methods in an object, and to all objects an object contains



```
method pushButton(var eventInfo Event)
var
   someNum Number
   x String
endVar
someNum = 123.321
x = "Don't panic"
endmethod
```

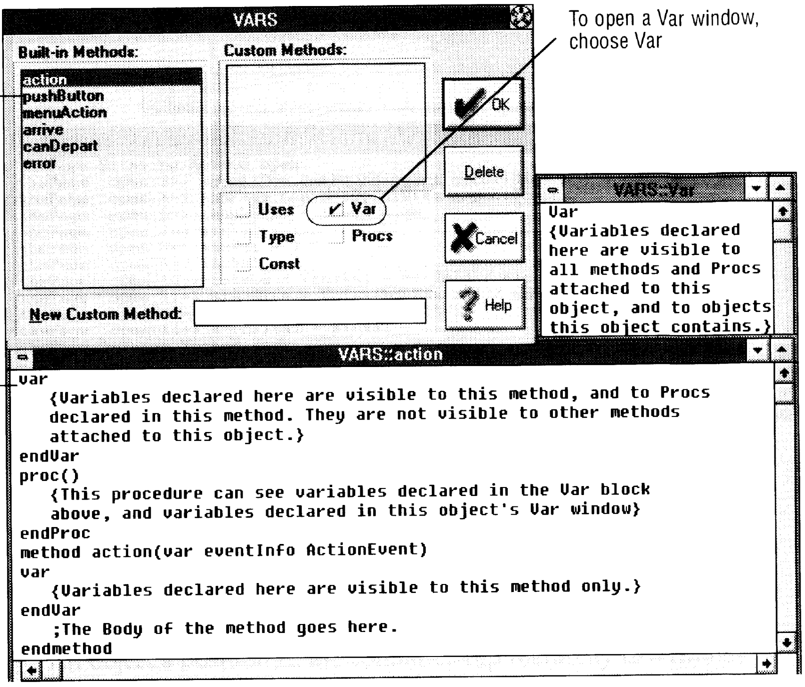
```
Var
   myTc Tcursor
   i SmallInt
   c File System
endVar
```

Click Var in the Methods dialog box to declare variables in an object's Var window

Scope: An example

The pseudocode in Figure 5-10 shows how the scope of a variable depends on where the variable is declared.

Figure 5-10 Scope of variables

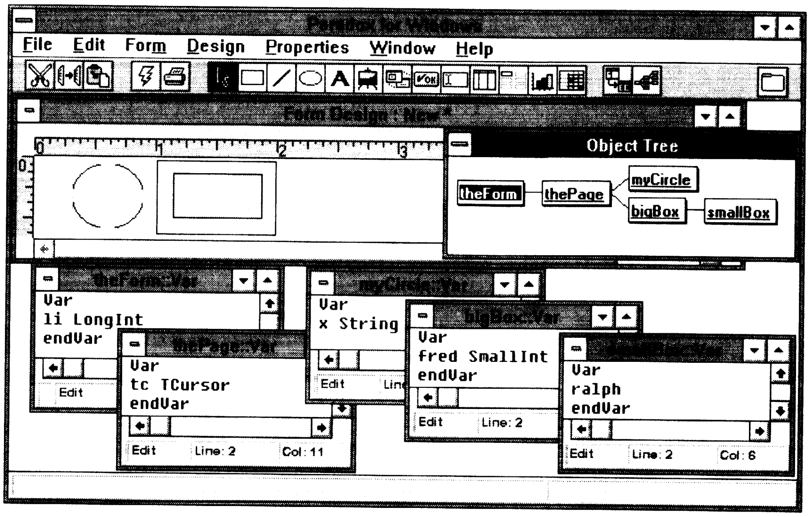

To open a Var window, choose Var
To open a method window like the one shown for **action**, choose a method from this list of built-in methods (or create a custom method)
This method window for **action** explains how the scope of a variable depends on where it is declared
VARS
Built-in Methods:
action
pushButton
menuAction
arrive
canDepart
error
Custom Methods:
OK
Delete
Cancel
Help
Uses
[✓ Var]
Type
Procs
Const
New Custom Method:
VARS::Var
VARS::action

```
Var
{Variables declared
here are visible to
all methods and Procs
attached to this
object, and to objects
this object contains.}
```

```
var
   {Variables declared here are visible to this method, and to Procs
   declared in this method. They are not visible to other methods
   attached to this object.}
endVar
proc()
   {This procedure can see variables declared in the Var block
   above, and variables declared in this object's Var window}
endProc
method action(var eventInfo ActionEvent)
var
   {Variables declared here are visible to this method only.}
endVar
   ;The Body of the method goes here.
endmethod
```

Figure 5-11 shows a form containing an ellipse, *myCircle*, and a rectangle, *BigBox*, which contains another rectangle, *SmallBox*.

Figure 5-11 The containership hierarchy



```
Var
li LongInt
endVar
```

```
Var
tc TCursor
endVar
```

```
Var
x String
```

```
Var
fred SmallInt
endVar
```

```
Var
ralph
endVar
```

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 5-11, *BigBox* can see the variable *fred* because it was defined in the *BigBox* Var window. Methods attached to *BigBox* can access *fred* directly:

```
; attached to BigBox
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.

SmallBox can see the variable *fred* too, because *SmallBox* is contained in *BigBox*. So *SmallBox* can access *fred* directly, as well:

```
; attached to SmallBox
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 heirarchy. *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 5-11 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.

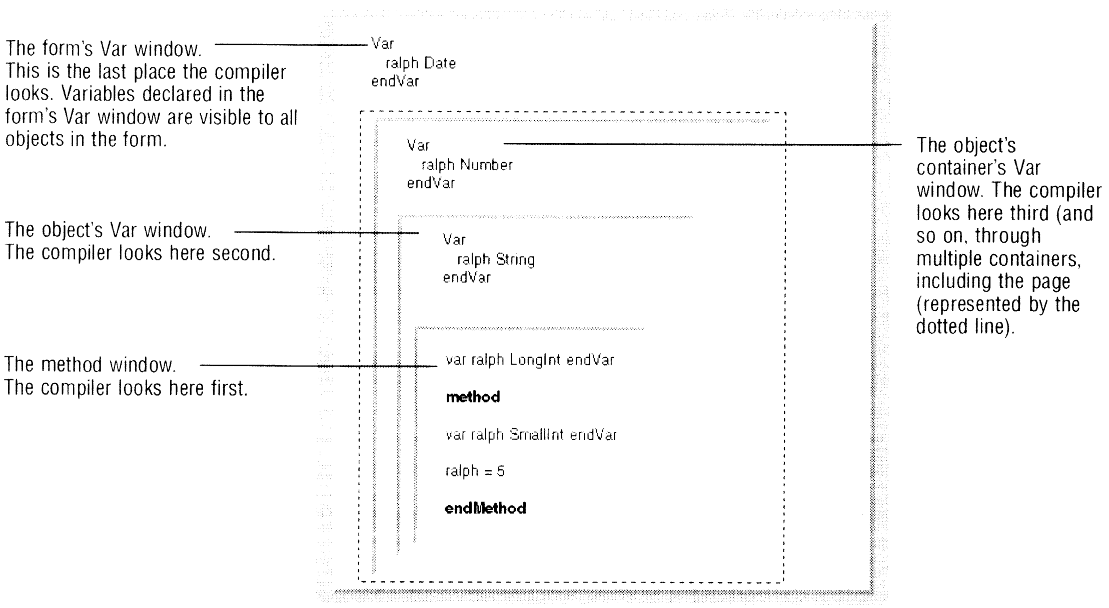
Note The relationship between containership and scope is a key concept in ObjectPAL. It applies not only to variables, but to object properties (discussed in Chapter 7), custom methods and procedures attached to objects (demonstrated in the examples in online), the way objects handle events, and the way ObjectPAL handles errors (discussed in Chapter 13).

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 5-12 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 5-12 Compile-time binding


The form's Var window. This is the last place the compiler looks. Variables declared in the form's Var window are visible to all objects in the form.
The object's Var window. The compiler looks here second.
The method window. The compiler looks here first.
The object's container's Var window. The compiler looks here third (and so on, through multiple containers, including the page (represented by the dotted line).
Var
ralph Date
endVar
Var
ralph Number
endVar
Var
ralph String
endVar
var ralph LongInt endVar
method
var ralph SmallInt endVar
ralph = 5
endMethod

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 your own 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 9.

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:
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

You can define constants for a single method, or open a Const window to define constants for all the object's methods.

Constants are like variables, except they're protected from change when the program runs, enabling the compiler to generate more efficient code. 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.

String constants are automatically put into resources, where they can be modified (using a tool like Borland's Resource Workshop) without affecting the source code.

The structure for declaring a constant is

```
const
   constName = value
   ; or
   constName = typeName (value)
   ; to cast the type
endConst
```

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

   c = Currency(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 Appendix H in the *ObjectPAL Referencc*
- ❐ In a Paradox ObjectPAL Editor window, by choosing Language | 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, Currency, 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 6.

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
```

PART II

Object types

This part of the manual contains the following chapters:

- ❐ Chapter 6, "Events," describes the ObjectPAL event model and presents objects that handle events—events generated by the user through the user interface, and events generated by ObjectPAL.
- ❐ Chapter 7, "Design objects," presents the objects that provide a user interface to an application: UIObject, Menu, and PopUpMenu.
- ❐ Chapter 8, "Display managers," presents objects that control how data is displayed to the user: Application, Form, MailMerge, Report, and TableView.
- ❐ Chapter 9, "Data types," presents the ObjectPAL data types: AnyType, Array, Binary, Currency, Date, DateTime, DynArray, Graphic, Logical, LongInt, Memo, Number, OLE, Point, SmallInt, String, and Time.
- ❐ Chapter 10, "Data model objects," discusses objects that provide access to and information about data stored in tables: Database, Query, Table, and TCursor.
- ❐ Chapter 11, "System data objects," discusses objects that store and manipulate data, but not data stored in tables: DDE, FileSystem, Library, Session, System, and TextStream.

CHAPTER 6

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 6-1.

Table 6-1 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

When you interact with an ObjectPAL application, you generate events. For each event, Paradox creates a packet of information and sends it to the form. By default, the form examines the event packet and dispatches it to the target object, which executes the appropriate built-in method. This behavior is governed by the ObjectPAL event model. 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

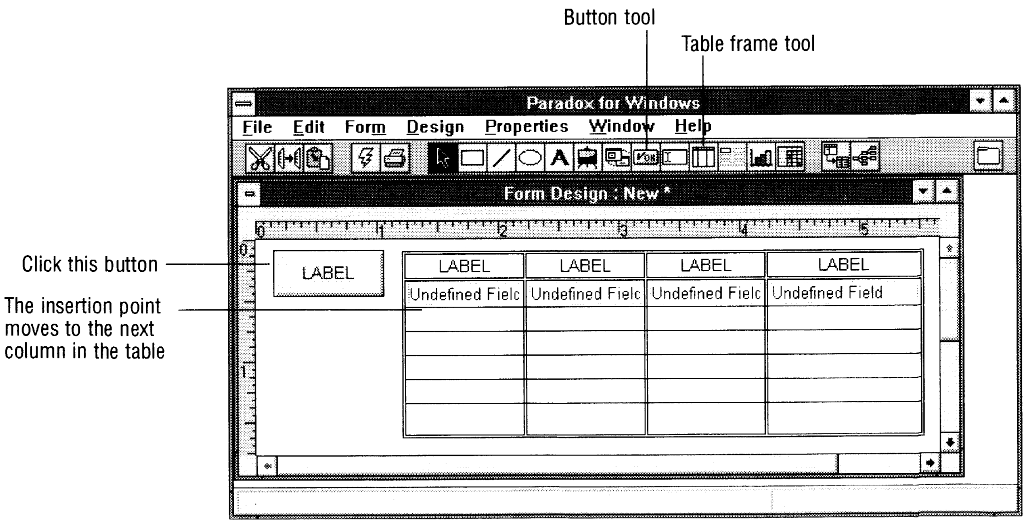
This example introduces the ObjectPAL *event model*, the rules Paradox uses to process events. The ObjectPAL event model is powerful and gives a programmer great flexibility. Understanding it is a key to getting the most out of ObjectPAL.

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 6-1 shows the completed form. The steps for creating it and attaching code follow.

Figure 6-1 The completed form



1. To begin, choose File | New | Form from the Desktop, and accept the defaults to create a blank form.

2. Use the Button tool to create a button in the upper left corner of the form.
3. Use the Table tool to create an empty table frame to the right of the button. Compare your form to Figure 6-1.
4. 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
```

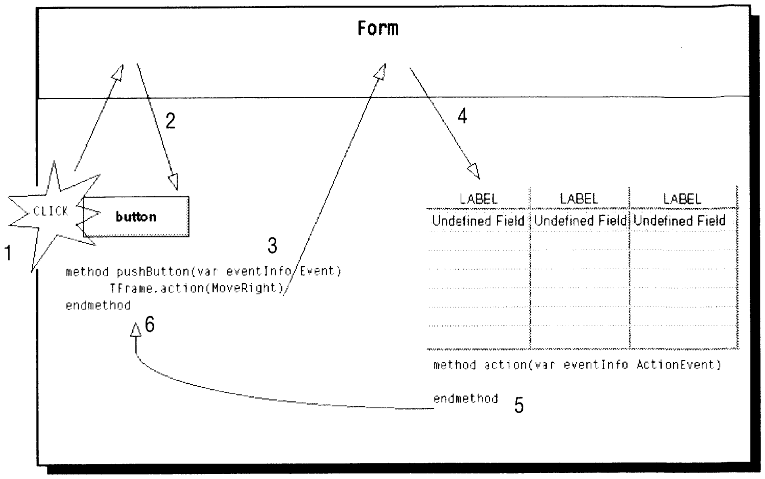
4. Close the Editor window.
5. Choose Form | View Data.
6. Click the button a few times. The insertion point (highlight) in the table frame moves to the right each time, until it reaches the rightmost field in the table frame.

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 6-2 shows what happens.

Figure 6-2 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.



```
method pushButton(var eventInfo Event)
    TFrame.action(MoveRight)
endmethod
```

```
method action(var eventInfo ActionEvent)

endmethod
```

Figure 6-2 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 executes, and the button appears to pop out. Processing for this event is complete.

Figure 6-2 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 6-2 and 6-3 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 Chapters 2 and 3 in the *ObjectPAL Reference.*

Table 6-2 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*	Executes when you click a button object
newValue**	Executes to report that a field object has a new value
changeValue**	Executes to post changes to a field value

* Buttons and drop-down edit lists only

** Field objects only

Table 6-3 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
mouseRightDouble	Executes when the right mouse button is double-clicked
keyPhysical	Executes when a key is pressed
keyChar	Executes when a keypress maps to an insertable ANSI character
action	Executes when a keypress maps to an action
menuAction	Executes when you choose an item from a menu or click a SpeedBar 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

Events go to the form first.

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.

External events "bubble."

In contrast, 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 6-3.

Figure 6-3 External events bubble

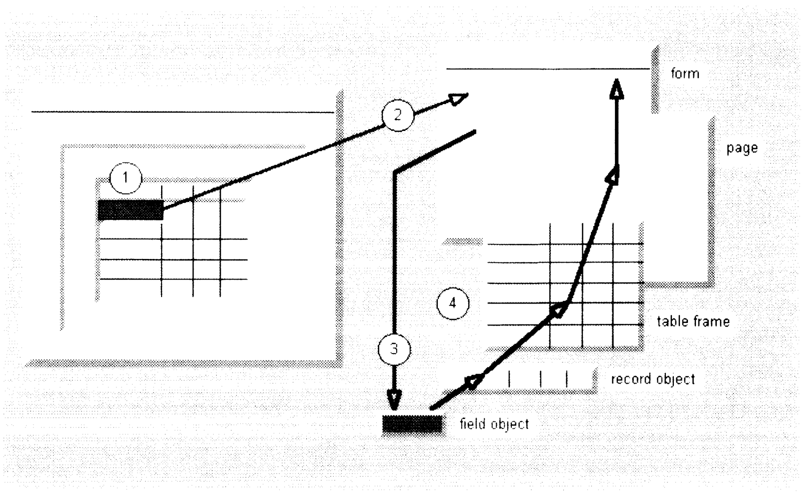


Figure 6-3 diagrams the flow of events that occurs when you type a character or click in a field object.

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 6-3, 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. (The methods for each type are listed in the *ObjectPAL Reference.)*

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 to get the x- and y-coordinates of the mouse at the time of the event.

```
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, 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 canDepart(var eventInfo MoveEvent)
   if self.value > 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 Language | Constants, then from the Types of Constants column, choose EventReturns. 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
       ; code here executes for each object on the form
   else
       ; code here executes just for the form itself

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()    ; code here executes for each object on the form
   else
             ; code here executes just for the form itself
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
             ; code here executes for each object on the form
   else
   beep()    ; code here executes just for the form itself

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
        ; code here executes for each object on the form
   else
        ; code here executes just for the form itself
        eventInfo.getTarget(theTarget)
        msgInfo("Target type:", theTarget.class)
        msgInfo("Target name:", theTarget.name)
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.*

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 a query to execute when you open a particular form. You *could* attach the following code to the form's built-in **open** method, and the query *would* execute when the form opens. For example,

Don't do this!

```
method open(var eventInfo Event)
   executeQBEFile("neword.qbe")
endMethod
```

But the query would also execute 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 **executeQBEFile** statement executes before the default code, so the query executes 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
      executeQBEFile("neword.qbe")
   endif
endmethod
```

In this example, the **executeQBEFile** 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 Appendix A of the *ObjectPAL Reference.*

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 2 of the *ObjectPAL Reference.*

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 13).

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 6-4 lists the constants.

Table 6-4 Reason constants

Method	Constant	Description
Event methods: **newValue**	FieldValue	Value was changed 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	Value was specified when the form opened.
ErrorEvent methods: **error**	ErrorCritical	Displays a message in a dialog box.
	ErrorWarning	Displays a message in the status area.
MenuEvent methods: **menuAction**	MenuNormal	SpeedBar 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.
	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.
	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 13.

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 2 in the *ObjectPAL Reference.*

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 field.")
      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 Language | 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 MaxUserAction 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 MaxUserAction. One way is to use a **message** statement:

```
message(MaxUserAction)
```

In this version of Paradox the difference between UserAction and MaxUserAction is 2047. That means the largest value you can use for an action constant is UserAction + 2047.

Methods for the ActionEvent type are closely related to the built-in **action** method. For more information, see Chapter 7 in this manual and Chapter 2 in the *ObjectPAL Reference.*

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 13.

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 MaxUserError to represent the minimum and maximum values allowed.

For example, suppose you want to define two action 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 MaxUserError. One way is to use a **message** statement:

```
message(MaxUserError)
```

In this version of Paradox the difference between UserError and MaxUserError 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.

Keyboard events and built-in methods

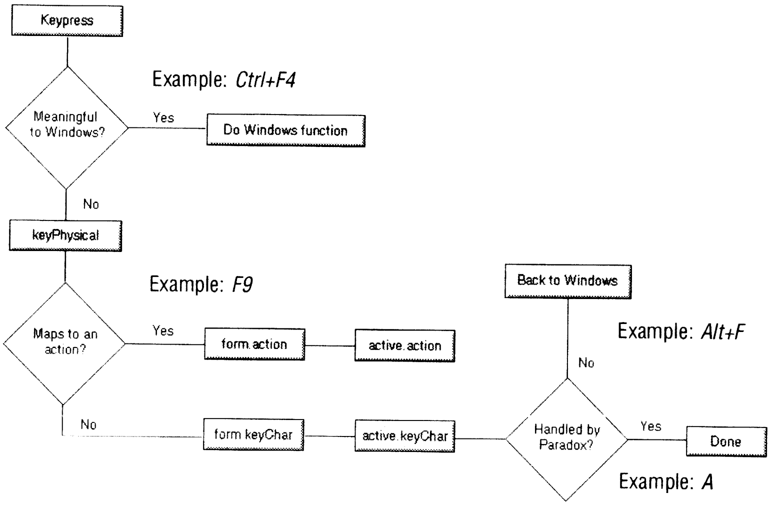
When you press a key in Paradox, one of the following happens:

- Windows intercepts the keystroke and performs an action.
- Windows passes the keystroke to Paradox, which performs an action.

- ❐ Windows passes the keystroke to Paradox, which passes it to the active object.

When Windows intercepts a keystroke, ObjectPAL performs no action. 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 6-4.

Figure 6-4 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 send or intercept characters in 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+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.vCharAnsiCode() = VK_LEFT  : x = x - 100
      case eventInfo.vCharAnsiCode() = VK_RIGHT : x = x + 100
      case eventInfo.vCharAnsiCode() = VK_UP    : y = y - 100
      case eventInfo.vCharAnsiCode() = 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 *ObjectPAL Reference*.

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
```

Key names and ANSI codes

The **vChar** method for the KeyEvent type returns the virtual key name as a string. Virtual key codes are constants used to represent a computer's standard keys, such as the letter *A* or the *Esc* key. Since different brands of computers might have a different set of keys, the virtual key codes are used when processing keyboard input. This enables one application to run on many different computers.

For example, when *Return* is pressed, **vChar** returns the string "VK_RETURN". In contrast, the **keyChar** method for the UIObject type takes a string as an argument. It does not take an integer ordinal value and it does not interpret the string; that is, it does not translate the sequence "and VK_RETURN" to the characters "and" and the ANSI code for VK_RETURN. **keyChar** doesn't recognize VK_RETURN as a virtual key, but rather as the string "VK_RETURN". To use these methods simultaneously, you must do some conversions. Table 6-5 lists the procedures ObjectPAL provides for this purpose (they're defined for the String type).

Table 6-5 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. This has limited utility. If you are using this in general programming, you are probably missing an easier way of doing something.

Here are some examples for converting between ANSI 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 SpeedBar 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.

For more examples showing how to work with menus, see Chapter 7.

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 action(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 MaxUserMenu 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 6-6.

Table 6-6 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 2 of the *ObjectPAL Reference.* This section presents an overview of how to use MoveEvent type methods.

Checking field values

One common task is to inspect the MoveEvent event packet to do some simple value checking. The best places to attach this kind of code are the built-in **canArrive** and **canDepart** methods, because these methods effectively ask for permission to perform the move, giving you a chance to inspect values and make decisions. In contrast, the built-in methods **arrive** and **depart** execute after the move has happened.

For example, suppose the following code is attached to the built-in **canDepart** method of a field object. When you try to move the pointer off this field object, this code executes. It checks the current value, 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 canDepart(var eventInfo MoveEvent)
   if self.value > 50 then
      msgStop("Stop", "Enter a value less than 50.")
      eventInfo.setErrorCode(CanNotDepart) ; CanNotDepart is a nonzero constant
   endIf
endmethod
```

This code prevents the pointer from leaving the field if it contains a value greater than 50; otherwise, it lets the default code determine whether the value is legal. You can use a similar technique to check a field's value before moving to it. For example, the following code displays a dialog box and prevents you from moving to the field if the value is less than 100:

```
method canArrive (var eventInfo MoveEvent)
   if self.value < 100 then
      msgStop("Value is too small.", "Press the INCREASE button.")
      eventInfo.setErrorCode(CanNotArrive)
   endIf
endMethod
```

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 **canDepart** 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:

```
; thisForm::Var
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 **canDepart** method for the form:

```
; thisForm::canDepart
method canDepart(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:

```
; thisForm::changeValue
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*:

```
; undoLast::pushButton
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
```

For more information about value checking, see the "ValueEvent: Handling field value changes" section later in this chapter.

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 13). 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
```

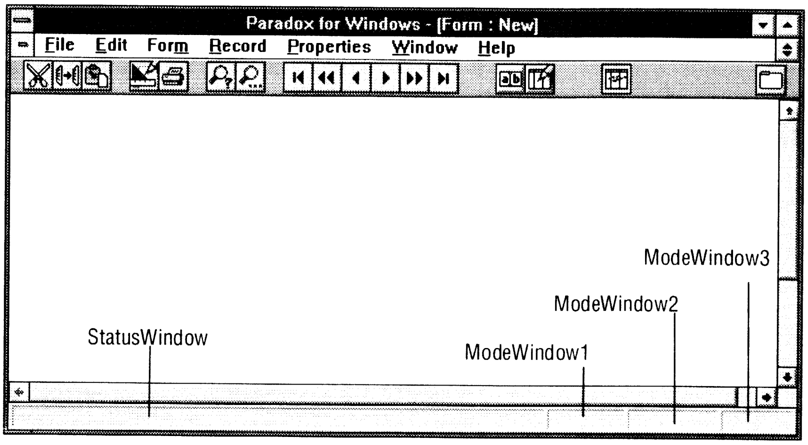
In this example, the custom method **doSomething** executes only when **canDepart** is called because of a user action; otherwise, the event is handled normally. (UserMove is an ObjectPAL constant.)

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 6-5. 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 6-5 The status bar


Paradox for Windows - [Form : New]
File Edit Form Record Properties Window Help
ModeWindow3
ModeWindow2
StatusWindow
ModeWindow1

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 SpeedBar. By default, moving the pointer over a SpeedBar button triggers the built-in **status** method (reason = StatusWindow) and displays a message in the status area; moving the pointer off a SpeedBar 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.

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 2 in the *ObjectPAL Reference.*

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.

```
; sendError::pushButton
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.

```
; Sale_Date::changeValue
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.

```
; thisForm::Var
Var
  UndoFlag Logical
endVar
```

This method is attached to the form's **open** method:

```
; thisForm::open
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:

```
; thisForm::changeValue
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:

```
; undoButton::pushButton
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
```

CHAPTER 7

Design objects

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 SpeedBar, 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 7-1 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 7-1 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 the following topics:

- UIObjects: Building blocks of the user interface
- Menu: A list at the top of the window
- PopUpMenu: Lists on demand
- Working with built-in menus and the SpeedBar

UIObjects: 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 SpeedBar (described in the *User's Guide*), the underlying page, the form, and more. A complete list of UIObjects is provided online. To display the list, open an ObjectPAL Editor window and choose Language | 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
- Using ObjectPAL in calculated fields

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:

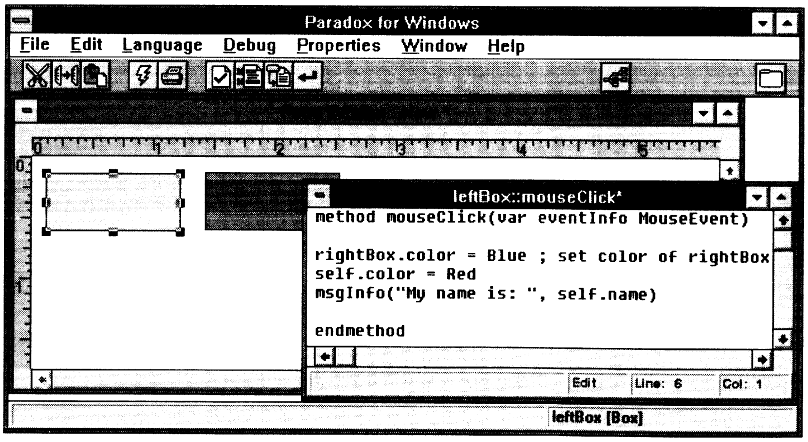
```
if income > outgo then
   result.font.color = Green ; use green characters if there is a profit
else
   result.font.color = Red ; if no profit, use red characters
endif
```

Objects can set their own properties and the properties of other objects. For example, see Figure 7-1, 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.

```
; this method is attached to leftBox
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 7-1 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.



```
method mouseClick(var eventInfo MouseEvent)

rightBox.color = Blue ; set color of rightBox
self.color = Red
msgInfo("My name is: ", self.name)

endmethod
```

For more information about object properties, see Appendix G in the *ObjectPAL Reference.* 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 G in the *ObjectPAL Reference.*

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 2 in the *ObjectPAL Reference* 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 the "Library" section of Chapter 11 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 5 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.

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 G in the *ObjectPAL Reference* 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 (commited). 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 Language | 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 Language | 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 F in the *ObjectPAL Reference*. Properties are also listed online.

Properties: Special cases

This sectioı 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.

  ```
  fred.fieldName = "[Parts.PartNum]"
  ```

  Quotes are required outside the square brackets. When referring to a table with its file extension (for example, CUSTOMER.DBF) or a field containing a space, (for example, Last Name), place quotes preceded by backslashes around the name of the table or field, as shown in the following statements:

  ```
  fred.fieldName = "[bookord.quant]"
  fred.fieldName = "[customer.\"last name\"]"
  fred.fieldName = "[customer.last name]"
  fred.fieldName = "[\"customer.dbf\".last name]"
  ```

  This is necessary because the period (in the file extension) and the space (in the field name) are special characters 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.

  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
```

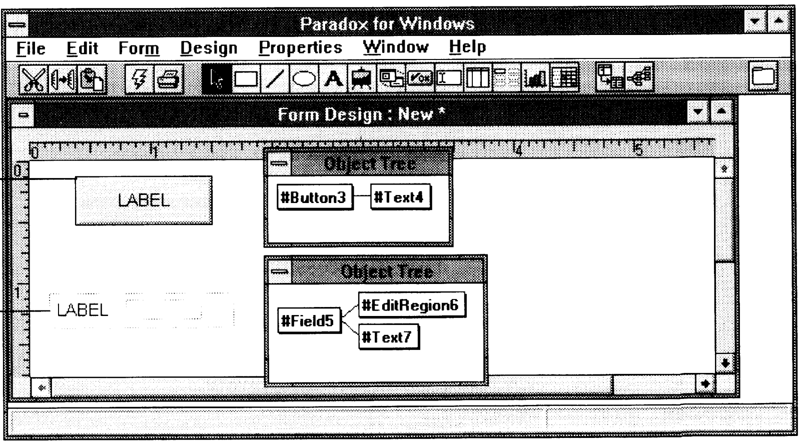
Remember, even when you don't use the VALUE keyword, you're working with the value of the object, not with the object itself.

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 Form | Object Tree. Figure 7-2 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 7-2 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).

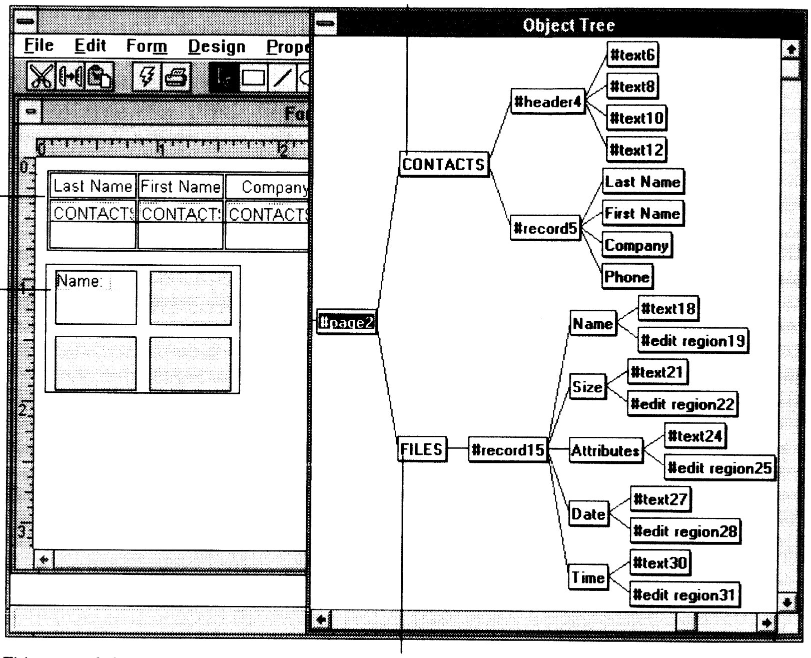
Table frames and multi-record objects are more complex compound objects. As shown in Figure 7-3, a table frame contains a record object that contains field objects that might contain text boxes. All these objects are created automatically when you create the table frame.

Figure 7-3 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 SpeedBar 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 2 in the *ObjectPAL Reference* for more information.

Keyboard events

You can use the built-in methods **keyChar**, **keyPhysical**, and **action** to respond to keyboard events. See Chapter 6 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 press 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 Chapters 2 and 6 in the *ObjectPAL Reference* 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 6 and the *ObjectPAL Reference* for more information and examples.

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.

Step 1: Getting started

1. Choose File | New | Form, and accept the default settings 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 Methods dialog box.
5. Choose **canArrive** and OK 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. Type the following text in the ObjectPAL Editor window:

   ```
   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. Close the window and save changes to the method.
8. Follow the previous steps (you can use the Clipboard to save some typing) to edit the **arrive**, **setFocus**, **canDepart**, **removeFocus**, and **depart** methods for *leftOutsideBox* as shown in the following method examples. (These are built-in methods, discussed in Chapter 2 in the *ObjectPAL Reference.)*

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
```

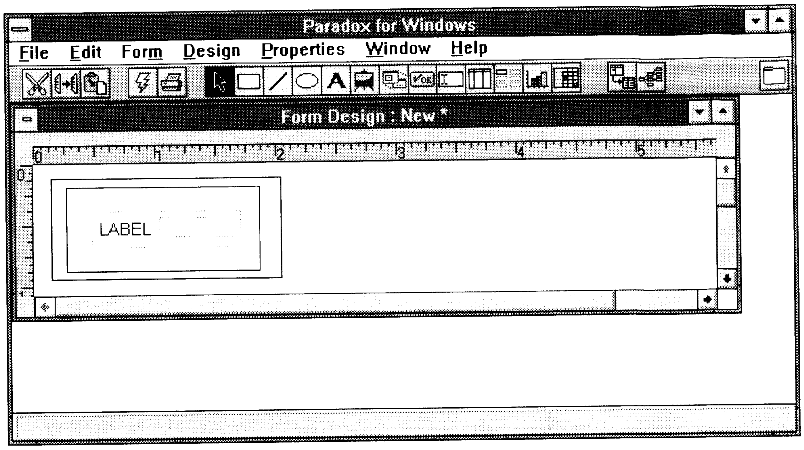
Step II: Copying an object and its methods

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 7-4. Leave the field object undefined.

Figure 7-4 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.

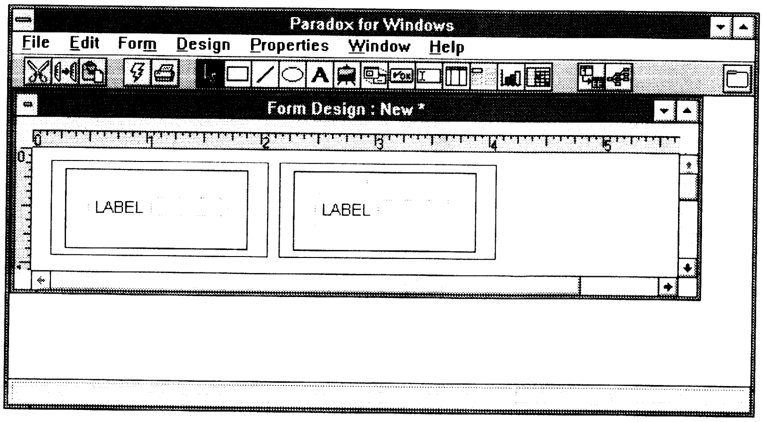
Stop for a moment and consider what you just did: by selecting one object, you selected all the objects that object contains. By duplicating one object, you duplicated all the objects that object contains, along with their methods. You've used the containership model to streamline the design process.

Step III: 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 7-5. Rename the outside box *rightOutsideBox* and rename the inside box *rightInsideBox*.

Figure 7-5 Move the duplicates



2. Save this form, then choose Form | View Data.

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**.

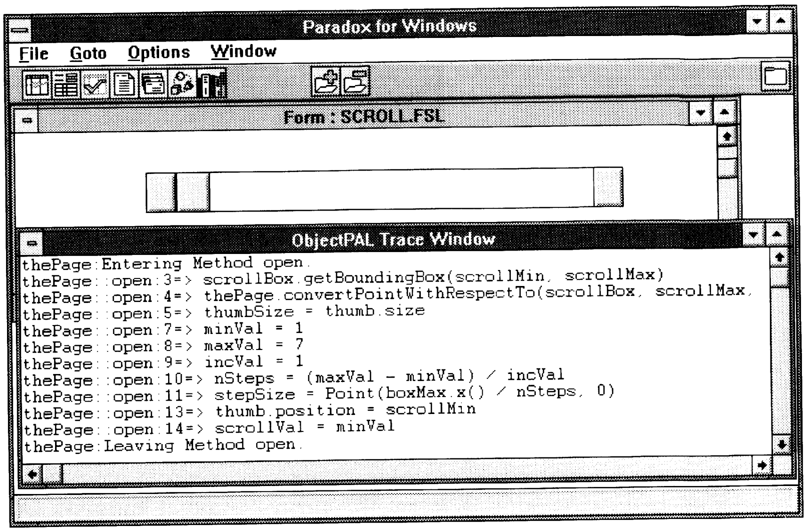
Note The two field objects are responding to the same sequence of events by executing their built-in methods. No messages appear, though, because you didn't attach any of your own code to the built-in methods.

Using the ObjectPAL Tracer

You can use the ObjectPAL Tracer to list each ObjectPAL statement as it executes. In the design window, open an Editor window and choose Debug | Trace Execution. Then, when you switch to the View Data choice, the ObjectPAL Tracer lists each line of ObjectPAL code as it executes, providing a record of what happened and when. Figure 7-6 shows the ObjectPAL Tracer listing ObjectPAL statements as they execute.

To display even more information, choose Debug | Trace Built-ins and check the built-in methods you want to trace as they execute, whether or not they have ObjectPAL code attached. Chapter 4 gives more information about using Trace Execution and Trace Built-ins.

Figure 7-6 The ObjectPAL Tracer lists ObjectPAL statements as they execute



```
thePage:Entering Method open.
thePage::open:3=> scrollBox.getBoundingBox(scrollMin, scrollMax)
thePage::open:4=> thePage.convertPointWithRespectTo(scrollBox, scrollMax,
thePage::open:5=> thumbSize = thumb.size
thePage::open:7=> minVal = 1
thePage::open:8=> maxVal = 7
thePage::open:9=> incVal = 1
thePage::open:10=> nSteps = (maxVal - minVal) / incVal
thePage::open:11=> stepSize = Point(boxMax.x() / nSteps, 0)
thePage::open:13=> thumb.position = scrollMin
thePage::open:14=> scrollVal = minVal
thePage:Leaving Method open.
```

Summary of demonstration

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.

For important information about the relationships between events, methods, and containers, see Chapter 6.

Actions and UIObjects

This section 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 7-2 describes the five basic action categories.

Table 7-2 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 SpeedBar 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.
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 Language | 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 Appendix H in the *ObjectPAL Reference*, 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 6): 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:

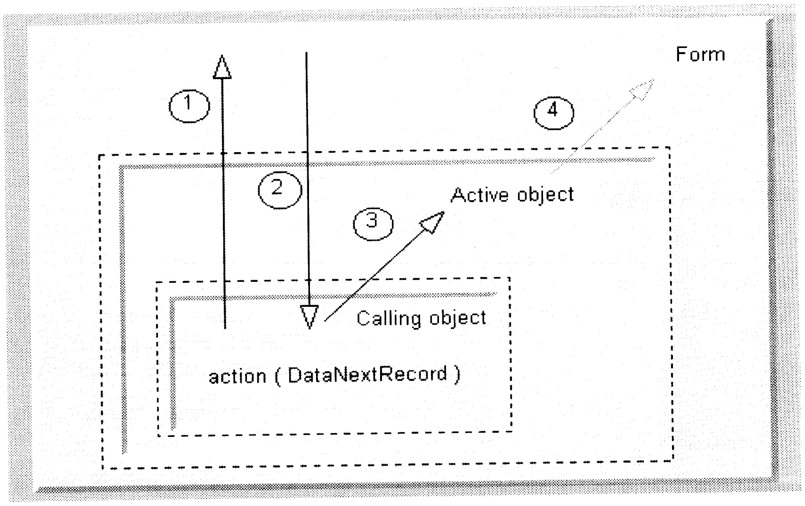
```
action(DataNextRecord)
self.action(DataNextRecord)
```

Figures 7-7 and 7-8 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 7-7, 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 7-7 Active object contains calling object

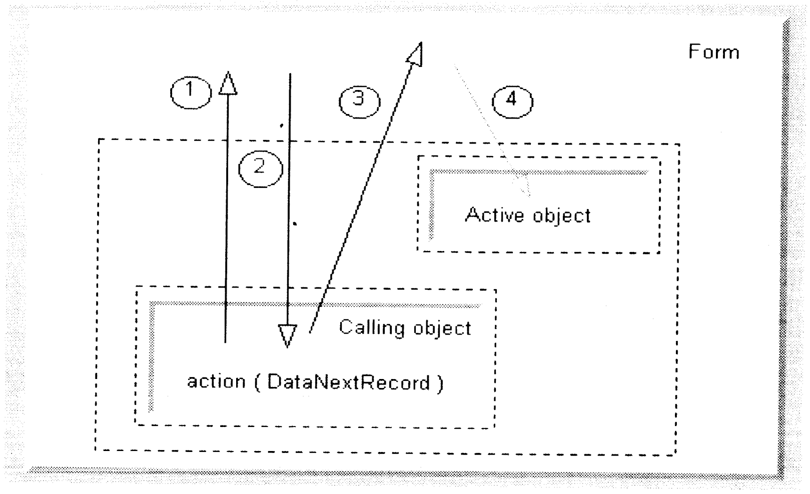
1. The calling object calls action(DataNextRecord), triggering the form's built-in **action** method.
2. The form dispatches the event to the built-in **action** method of the calling object.
3. The calling object bubbles the event to its container, and soon until it reaches the active object.
4. The active object performs the action if it can, or bubbles the event if it can't.



In Figure 7-8, 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 7-8 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

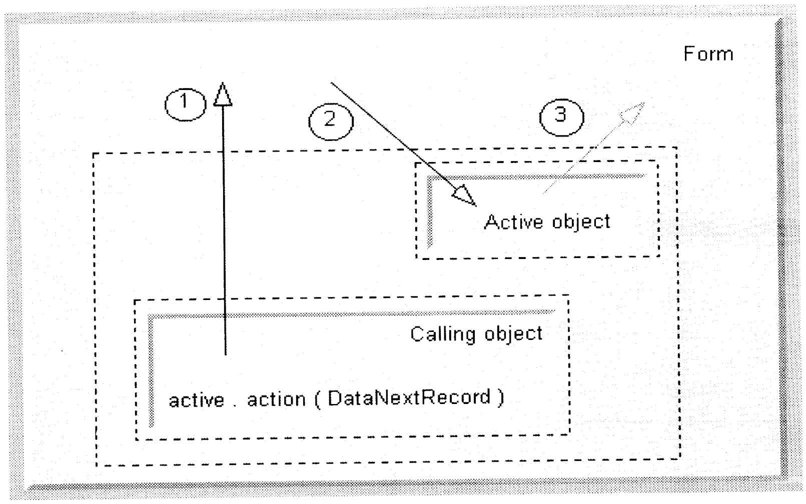
When you call **action** with an object variable, the results depend on which variable you use. (The object variables are *self, subject, container, active, lastMouseClicked,* and *lastMouseRightClicked.* They are described in Chapter 2 of the *ObjectPAL Reference.*) Calling **action** with *self* as 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 7-9 diagrams the effect of the 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 7-9 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.

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 initated in several ways. A DataArriveRecord action occurs whenever the form "arrives at" (displays) a new or different record. For example, moving to the next or previous record initiates a DataArriveRecord action; so does inserting or deleting a record; so does editing a record. DataArriveRecord is a useful general-purpose action.

Refer to *Learning ObjectPAL* 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 "least common container"; that is, 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 least common 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 Appendix H in the *ObjectPAL Reference* 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 10 for examples.

Example: Working with actions and records

The SpeedBar 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 modfied 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
```

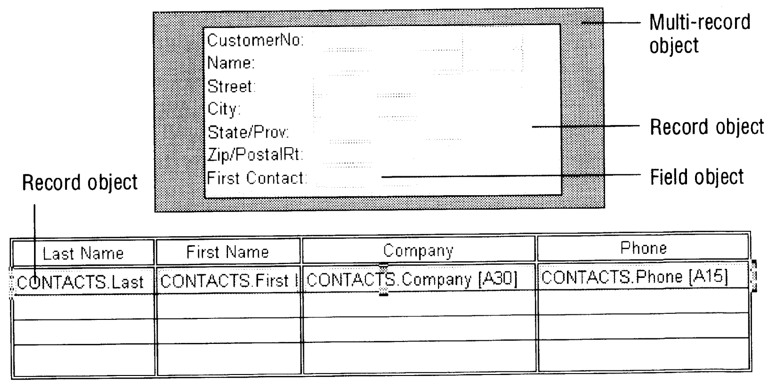
- Create a multi-record object, then define its record layout to display one record across and one record down (also called 1 x 1, pronounced "one by one").
- Use the record object contained in every table frame (the Object Tree provides easy access).

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 Language | Properties. Then, from the Objects column, choose Record. The properties appear in the Properties column.

Figure 7-10 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 7-10 Record objects in a multi-record object and a table frame



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:

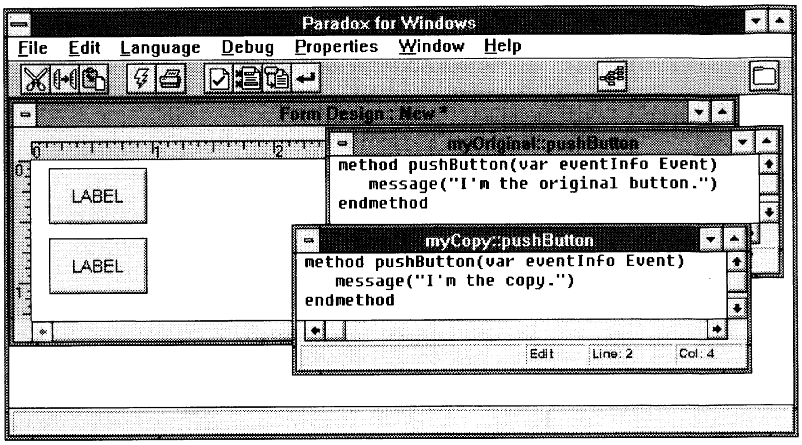
   ```
   method pushButton (var eventInfo Event)
      message("I'm the original button.")
   endMethod
   ```

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 7-11.

Figure 7-11 The original and the copy



```
method pushButton(var eventInfo Event)
   message("I'm the original button.")
endmethod
```

```
method pushButton(var eventInfo Event)
   message("I'm the copy.")
endmethod
```

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 SpeedBar.

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 SpeedBar, *every* subsequent field will have those properties and methods—this includes fields you create using the Field tool from the SpeedBar, 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 Language | Constants, then choose UIObjectTypes. 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
```

Using ObjectPAL in calculated fields

This section 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 section focuses on ObjectPAL issues.

As shown in Table 7-3, 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 7-3 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.

Element	Comments
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*:

```
; This code is attached to the field's open method
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:

```
; This is a custom method attached to theBox
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.

```
; This code is attached to the field's Var window
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*.

```
; This code is attached to the field's open method
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:

```
; This code is attached to the field's Var window
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.

```
; This code is attached to the field's open method
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:

```
; This 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:

```
; This 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:

```
; This code defines the calculated field
   itemsTC.recNo()
```

The following code is attached to the button's built-in **pushButton** method:

```
; This 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 push 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*.

Menu: A list at the top of the window

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.

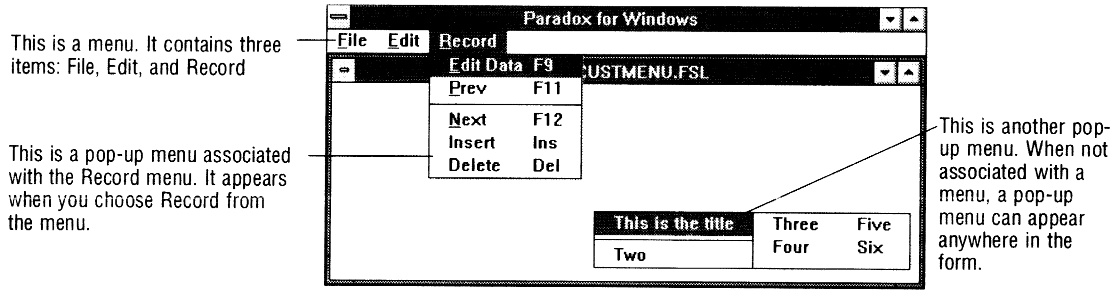
A typical application combines menus and pop-up menus (shown in Figure 7-12).

This section begins with a tutorial showing how to build, display, and respond to a menu. Following the tutorial are sections that present advanced techniques for working with menus, and explain how to

- ❒ Work with pop-up menus alone
- ❒ Work with Paradox's built-in menus and the SpeedBar
- ❒ Work with Windows system menus
- ❒ Respond to choices from Paradox's built-in menus

For more examples showing how to work with menus, run the MAST application and the Slot Machine game application. Both come with online Help and explanations of the code.

Figure 7-12 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.
Paradox for Windows
File Edit Record
Edit Data F9
Prev F11
Next F12
Insert Ins
Delete Del
:USTMENU.FSL
This is the title
Two
Three Five
Four Six
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

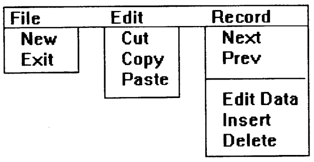
- ❒ Specify menu item ID numbers
- ❒ Specify the display attributes of menu items
- ❒ Provide keyboard access to menu items
- ❒ Inspect items in a menu

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 "Advanced menu techniques" later in this section.

The following tutorial shows how to build, display, and process choices from the menu shown in Figure 7-13. These basic techniques are presented in the context of a one-form, one-page application.

Figure 7-13 The finished menu for this part of the tutorial



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 tutorial shows how to write code to build and display the menu shown in Figure 7-13. The first step is to create a single-record form bound to the *Customer* table.

Creating the form

1. Choose File | New | Form to open the Data Model dialog box.
2. Add CUSTOMER.DB to the form's data model, then choose OK to close the Data Model 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.

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 Methods dialog box.
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.

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.

Attaching code 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:

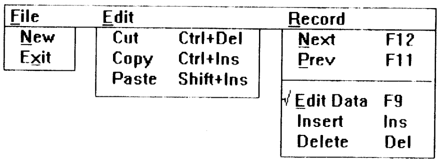
```
formVar.create() ; create a new form
```

Advanced menu techniques

The previous section presented 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 techniques presented in this section.

In this section, you'll build the full-featured menu shown in Figure 7-14. It's almost identical to the menu in the previous example, but you'll use advanced features to add more functionality.

Figure 7-14 A full-featured menu



At the end of this section is an example showing how to respond to choices from Paradox's 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 = 301
   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, RecordEdit)
```

The first part, **recordPop.addText**, you've seen before. It 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 7-4 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 7-4 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

Earlier in this chapter, an example showed 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(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, the sequence of events is:

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 Open, and pressing *F11* is the same as choosing New. 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 6 in this manual and Chapter 2 in the *ObjectPAL Reference*.

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(DatPriorRecord)
      otherwise              : doDefault
   endswitch
endif
endmethod
```

This example uses virtual keycodes (described in the "KeyEvent" section of Chapter 6). ObjectPAL provides constants for Windows virtual keys (like *F11* and *F12*), listed in Appendix C in the *ObjectPAL Reference* and online. To display the list, open an ObjectPAL Editor window and choose Language | Constants. Then, from the Types of Constants column, choose Keyboard. The constants appear in the Constants column.

Responding to choices from Paradox's 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's 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 respresents 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 *ObjectPAL Reference* for more information and examples.

PopUpMenu: Lists on demand

A PopUpMenu 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 PopUpMenu 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.

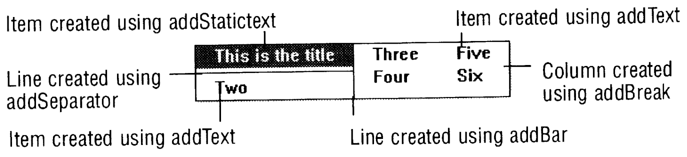
The following example shows how to build and display the pop-up menu shown in Figure 7-15. You probably won't ever build a menu like this one. Its purpose is to show at a glance the effects of the various menu-building methods.

```
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 7-15 Sample pop-up menu


Item created using addStatictext
Item created using addText
This is the title
Three
Four
Five
Six
Two
Line created using addSeparator
Column created using addBreak
Item created using addText
Line created using addBar

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 endVar
   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 the "Menu" section in this chapter 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.

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 SpeedBar

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 Language | 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 SpeedBar

The menu-handling techniques outlined in this chapter also work with SpeedBar 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 SpeedBar is equivalent to choosing Edit | Cut. So, you don't have to write separate handler methods for SpeedBar 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 SpeedBar 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() = MenuSystemMinimize then
                     ; MenuSystemMinimize 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.

```
; thisForm::open
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.

```
; thisForm::menuAction
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.

```
; replayButton::pushButton
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.

```
; fieldOne::menuAction
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.

CHAPTER 8

Display managers

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 8-1, and explains how to use their methods.

Table 8-1 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

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 the "UIObject" section of Chapter 7 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
- Working with a form's data model
- 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.fsl") ; 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.fsl")
formVar.open("orders.fsl")
formVar.minimize() ; minimize ORDERS.FSL
```

Table 8-2 lists commonly used Form type methods. Use these methods to work with a form as a display manager.

Table 8-2 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*.

```
;this code is attached to form0
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,

```
; Assume this method is attached to formOne
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")
myForm.getTitle(oldTitle)
; 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:

```
; this method is attached to another form, not to proteus
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 "Multi-form applications" later in this chapter.

Working with a form's data model

You can use ObjectPAL to manipulate a form's data model by getting and setting information about which table or tables the form is bound to. The methods listed in Table 8-3 are described in detail in the *ObjectPAL Reference.*

Table 8-3 Methods for working with a form's data model

Method	Description
dmAddTable	Adds a table to a form's data model
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

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 the name of a table in the current data model, including the alias. 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 val check file 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.

Using the form as a design object

The form can also 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, refer to Chapter 7.

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 Methods dialog box. (Shortcut: Press *Esc* repeatedly until no objects are selected, then press *Ctrl+Spacebar* to display the Methods dialog box.)
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,

```
; this code is attached to formOne
var
   formTwo Form
endVar
formTwo.open("orders.fsl")
formTwo.theText.text = "This text was sent by formOne."
formTwo.theEllipse.color = "Red"
```

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 section discusses the key concepts for building multi-form applications. The topics covered are:

- 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 SpeedBar, 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. For an example of a custom SpeedBar that uses a nonmodal dialog box, see the *Wreckbar* form of the MAST example application.

Position

The position coordinates for a dialog box are calculated with respect to the Desktop, while the position coordinates for a form are calculated with respect to the screen. 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 Desktop because dialog boxes are *not* MDI children. See "MDI child windows" later in this section.

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.

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 speedbar. 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 MAST 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("findWho.fsl", WinStyleDlgFrame, 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 *ObjectPAL Reference.*

```
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 Chapter 11.

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 *ObjectPAL Reference* 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 SpeedBar 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 runtime menu, which may cause menus to "flicker" in some applications.

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 8-4 lists the Windows style constants you can use with these methods. Constants are described in Appendix H in the *ObjectPAL Reference.*

Table 8-4 Constants for open and openAsDialog

openAsDialog	**open**
WinStyleBorder	
WinStyleCaption	
WinStyleControlMenu	
WinStyleDefault	WinStyleDefault
WinStyleDialog	
WinStyleDialogFrame	
WinStyleHidden	WinStyleHidden
WinStyleHScroll	WinStyleHScroll
WinStyleMaximize	WinStyleMaximize
WinStyleMaximizeButton	
WinStyleMinimize	WinStyleMinimize
WinStyleMinimizeButton	
WinStyleModal	
WinStylePopup	
WinStyleSizeBorder	
WinStyleSysMenu	
WinStyleThickFrame	
WinStyleTitleBar	
WinStyleVScroll	WinStyleVScroll

Note on properties 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.

Report: Format and print data

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 the 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.

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:

```
; Options in this record correspond to settings in the Print File dialog box
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.

For more information and examples, refer to the *ObjectPAL Reference*.

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 its namesake in the Form type 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 G in the *ObjectPAL Reference.*

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 10.

CHAPTER 9

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 true 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 9-1.

Table 9-1 Data types

Type	Description
AnyType	A catch-all for basic data types
Array	An indexed collection of data
Binary	Machine-readable data
Currency	Used to manipulate currency values
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
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 *ObjectPAL Reference*.) 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
- Currency
- Date
- DateTime
- Graphic
- Logical
- LongInt
- OLE
- Memo
- Number
- Point
- SmallInt
- String
- Time

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 forth.

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 7), 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 5.

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 5.
- 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, you must 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 customer 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[] 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 9-2 add data to a resizeable array. Each method will automatically expand the array to make room for the item or items being added.

Table 9-2 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 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 arl, ar2 Array[2] String endVar
arl[1] = "a"
arl[2] = "b"

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

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

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

message(ar2 >= arl) ; Displays True.
                     ; All items in arl >= 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
   arl Array[2] String
   ar2 Array[2] String
   ar3 Array[3] String
   ar4 Array[2] String
endVar

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

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

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

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

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

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

message(arl<>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 arl = 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 9-3 lists the methods for working with Binary variables and binary objects.

Table 9-3 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, refer to Chapter 3 of the *ObjectPAL Reference.* 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 *ObjectPAL Reference.* See also the description of the OLE type later in this chapter for techniques for editing and displaying data using the OLE protocol.

Currency: Money values

Currency values can range from ± 3.4 * 10^{-4930} to ± 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
      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.

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 9-4 lists commonly used methods and procedures defined for the Date type.

Table 9-4 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 Data value. For example, in the following example, *Result* is a Date value:

```
var D1, D2, Result Date endvar

d1 = Today()
d2 = Date("07/03/92")
Result = D2 - D1   ; Result = 04/15/24
```

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 = Now()                    ; get the current time
t2 = t1 + Time("12:00:00")    ; 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 SmallInt
endVar

d1 = Today()
d2 = Date("07/21/69")
Result = SmallInt(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 5.

DateTime: Calendar data and clock data combined

A DateTime variable stores data in the form hour-minute-second-millisecond year-month-day. DateTime values are used only in ObjectPAL calculations; you cannot store a DateTime value in a Paradox table. 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 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 9-5 lists commonly used methods defined for the DateTime type.

Table 9-5 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 Chapter 3 of the *ObjectPAL Reference.*

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, only contains 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
```

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 9-6 describes what each raster operation property does.

Table 9-6 Data 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 *Getting Started.*

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 *ObjectPAL Reference* for more information.

LongInt: Long integers

LongInt values are long integers; that is, they can be represented by a long series of digits. A LongInt variable occupies 4 bytes.

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, z LongInt
endVar

x = 2147483647 ; the upper limit value for a LongInt variable
y = 1
z = x + y ; causes an error
```

To work with boundary values, store the result in a variable of a type that can accommodate it. For example,

```
var
   x, y LongInt
   z    Number    ; declare z as a Number so it can hold the result
endVar

x = 2147483647    ; the upper limit value for a LongInt variable
y = 1
z = x + y         ; This statement is OK, because z is a Number
                  ; and can handle the large value
```

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() = True then
         return True
      endIf
   endFor
   return False
endProc

proc searchFromStart() Logical
   shipsTC.home()
   for i from 1 to recMarker
      if doSearch() = True 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 *ObjectPAL Reference.*

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 9-7 shows some examples of numeric constants:

Table 9-7 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, Currency, 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 in Chapter 11 because DDE data cannot be stored in a table.

Table 9-8 defines terms commonly used when discussing OLE operations:

Table 9-8 OLE terms and definitions

Terms	**Definition**
OLE Server	An application that can provide access to its documents via the OLE mechanism. Paradox is not an OLE server.
OLE Client	An application that can use the OLE mechanism to access documents created by an OLE server. Paradox is an OLE client.
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.

Table 9-9 lists the methods for working with OLE objects and OLE variables. These methods let you read, write, and work with an OLE Object.

Table 9-9 OLE type methods

Method	**Description**
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.
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.

Method	Description
getServerName	Returns a string that identifies the server. The string is a descriptive name, not necessarily the file name of the server.
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	Writes 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, because Paradox is not an OLE server.

An overview of OLE

ObjectPAL has two ways to retrieve an OLE object:

- From a table
- From the Clipboard

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 display it in the title bar to inform the user about what's happening. Paradox passes the integer to the server to specify an action to take.

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.

```
method pushButton(var eventInfo Event)
   var
      oleTC TCursor
      oleVar OLE
      oleVerbs DynArray[] SmallInt
      verbPop PopUpMenu
      verbChoice 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()
   chosenVerb = oleVerbs[verbChoice]
   if chosenVerb <> "" then
      oleVar.edit("PdoxWin", chosenVerb)
   endIf
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** to make sure the operation is possible. This step is optional.
4. Call **readFromClipboard** 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 an OLE object and a button. Name the OLE object *oleObject* and attach the following code to the button's built-in **pushButton** method:

```
method pushButton(var eventInfo Event)
   var
      oleGraphic OLE
      oleVerbs DynArray[] SmallInt
      verbPop PopUpMenu
      chosenVerb SmallInt
      verbChoice String
   endVar

   oleObject.edit()

   if oleGraphic.canReadFromClipboard() then
      oleGraphic.readFromClipboard()
      oleVar.enumVerbs(oleVerbs)
      forEach i in oleVerbs
         verbPop.addText(i)
      endForEach

      verbChoice = verbPop.show()
      chosenVerb = oleVerbs[verbChoice]
      if chosenVerb <> "" then
         oleVar.edit("PdoxWin", chosenVerb)
      endIf

   else
      msgStop("Stop", "Can't read OLE graphic from Clipboard.")
   endIf

   oleObject.value = oleGraphic
   oleObject.endEdit()
endMethod
```

4. Run the form and click the button. Paintbrush will open and let you edit the graphic, if all goes well, or display a dialog box if there's a problem. When you exit Paintbrush, you return to Paradox.

For more information, refer to the *ObjectPAL Reference.*

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 5 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
```

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, z SmallInt
endVar

x = 32767 ; the upper limit value for a SmallInt variable
y = 1
z = x + y ; causes an error
```

To work with boundary values, store the result in a variable of a type that can accommodate it. For example,

```
var
   x, y SmallInt
   z    LongInt   ; declare z as a LongInt so it can hold the result
endVar

x = 32767 ; the upper limit value for a SmallInt variable
y = 1
z = x + y

z.view() ; displays 32768 because z is a LongInt
         ; and can handle the large value
```

Note The value -32,768 cannot be stored in 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 = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
b = "bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb"
c = "cccccccccccccccccccccccccccccccccccccccccccccccccc"
d = "dddddddddddddddddddddddddddddddddddddddddddddddddd"
e = "eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee"
f = "ffffffffffffffffffffffffffffffffffffffffffffffffff"

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" section of the *ObjectPAL Reference* 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 9-10, 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 9-10 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 9-10 cannot be preceded by a backslash: it will cause a syntax error when you compile the code.

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** in the "String" section of the *ObjectPAL Reference.*

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 alphanumeric, 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$C". Although in this situation the input data type could be Currency (**$**), Number (**N**), LongInt (**L**), or SmallInt (**S**), you could place the output only in an alphanumeric 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 *ObjectPAL Reference.* 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
```

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. Whether a time is valid depends on the current Windows time format. For example, if the Windows time format is set to 12-hour format (such as hh:mm:ss), methods in the Time type consider hh:mm:ss a valid time format. To set Windows time formats from Paradox, use the procedure **FormatSetTimeDefault** defined for the System type.

Table 9-11 lists commonly used methods defined for the Time type.

Table 9-11 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

CHAPTER 10

Data model objects

ObjectPAL is a rich language that lets you develop a wide variety of applications. However, it's primarily a database language. This chapter presents the objects, types, and methods that work with tables. It describes the Data model objects shown in Table 10-1, which provide access to and information about data stored in tables.

Table 10-1 Data model objects

Type	Description
Database	A collection of tables
Query	Query by Example (QBE) queries
Table	A table
TCursor	A pointer to a table that is stored and manipulated in memory; it is not displayed. Use TCursors to work with data in tables without displaying the data; use table frames or multi-record objects (both are UIObjects) to display and manipulate data interactively.

This chapter also describes how to

- Use an alias with tables
- Open a database
- Create and execute queries
- Perform table-level operations and specify table attributes
- Use TCursors to manipulate data at the table level, record level, and field level without displaying the table
- Specify fields
- Search, read, and edit records in a table
- Use tables and TCursors to program lists

Database: A collection of tables

A Database variable provides a *handle*, a variable name 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, is a path to tables. 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 a table is 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 | Alias) 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.

Directory paths and the data model

When you add a table to the form's data model interactively, it stores the table's name as follows:

- ❒ Tables with absolute paths or aliases (such as "C:\TEMP\FOO" or ":MYALIAS:FOO") 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 that path name.
- ❒ Tables with no path or alias (that is, just the base name) use the path or alias where the form was found.

For example, 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 they aren't where the form expects them to be, Paradox displays an error message.

When the form finds a table, it uses the same algorithm to locate lookup tables. If a lookup table was defined with a full directory path, Paradox uses that path to locate the lookup table. If, however, the lookup table has no path, then Paradox expects to find it in the directory where the main table is located. Hence, in the previous example, if "MYLOOK.DB" is the lookup table, 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.

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 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, 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 ORDER.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.

Query: Retrieve data

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 "Manager" to "Supervisor" 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.

Before you create and run queries using ObjectPAL, you should be familiar with using queries interactively. See the *User's Guide* for more information.

To create a QBE (Query by Example) query using Paradox interactively, make a query form look like an example of the information you're interested in, then save it. To create a query using ObjectPAL, you can

- ❐ Execute a saved query
- ❐ Define and execute a query statement
- ❐ 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.

Using a query file

From a programming perspective, the simplest case is executing a query file. Assuming the query file has already been created and saved (for example, using queries interactively), use **executeQBEFile** 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.)

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 13.

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 10-2 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 instance, 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 = "301"

q = Query

c:\tables\orders.db | Order#         |
                    | Check ..~Ordnum |

endQuery

executeQBE(q, "ord301.db") ; store the results in :WORK:ORD301.DB
```

Table 10-2 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
	FIND	Find specified records in a table
	SET	Define specific records as a set for comparisons
Arithmetic operators	+	Addition or alphanumeric string concatenation
	-	Subtraction
	*	Multiplication

Category	Operator	Meaning
	/	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
	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 **executeQBEString**. 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. Notice the double backslashes before the table name—they're 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
```

Table: A description of tabular data

A Table variable represents and describes tabular data; 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 window, 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, set filters, do 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, table frame, or multi-record object instead. (See "Working with data in tables" later in this chapter.)

- You can specify table attributes, including filters, indexes, 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 tabular data.

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 "Table" section of the *ObjectPAL Reference.*

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, see the *ObjectPAL Reference.)*

Note These Table type methods don't open the table or do any verification; they only define a structure for the TCursor **open** method.

Use these methods after using **attach** and before using the Table variable to open a TCursor. (TCursors are discussed in the next section.) 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 **setFilter** 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.

TCursor: A pointer to the data in a table

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.

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

The relationship of a TCursor to a table is like that of 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 10-3 lists some methods for working with tables and TCursors.

Table 10-3 Methods for working with tables and 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
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.

```
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, 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 8). 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 7 and 8 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 10-1 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.

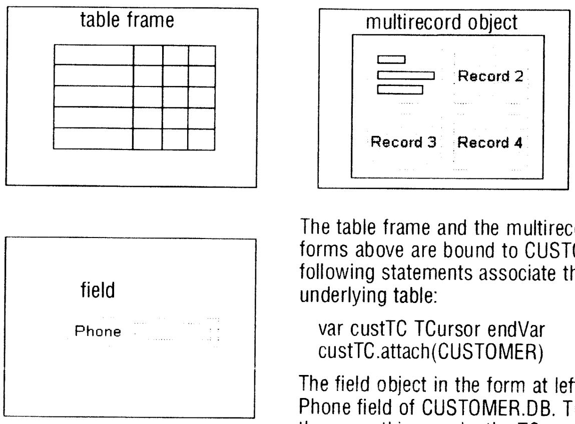
In the first two forms, which contain a table frame and a multi-record object respectively, the following statements make the TCursor *custTC* point to the *Customer* table. (By default, when a table frame or multi-record object is bound to a table, it takes the name of that table. The table frame and 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 10-1 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 10-2 shows how you can attach TCursors to UIObjects where the UIObject is bound to linked tables.

Note If you attach TCursor to a UIObject bound to linked tables with a single-valued detail table, the TCursor gets the structure and data of the master table only.

Figure 10-2 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:

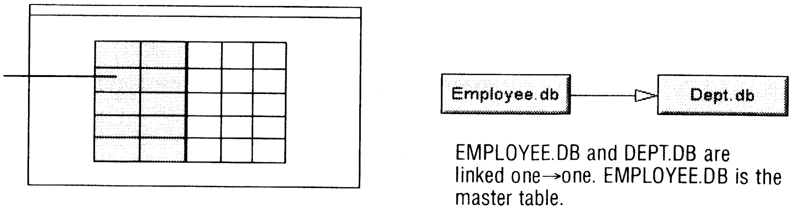
```
var thisTC TCursor endVar
thisTC.attach(ORDERS)
message(thisTC.nRecords()) ; displays 3
```

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 10-2 Attaching a TCursor to a UIObject bound to linked tables

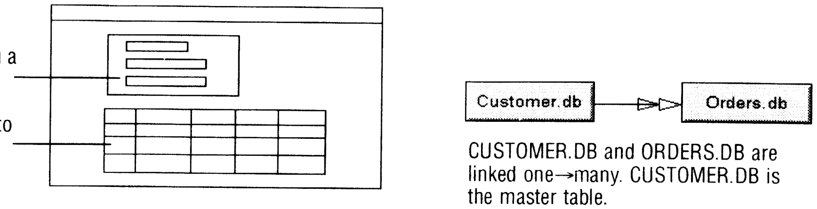
This table frame (EMPLOYEE) displays records from EMPLOYEE.DB and DEPT.DB. But the following statement makes the TCursor point to records in EMPLOYEE.DB only (shaded in the figure), because EMPLOYEE.DB is the master table:

custTC.attach(EMPLOYEE)



EMPLOYEE.DB and DEPT.DB are linked one→one. EMPLOYEE.DB is the master table.

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.

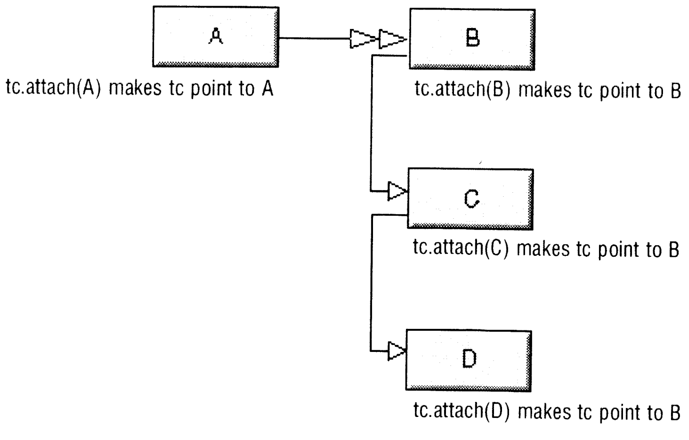


CUSTOMER.DB and ORDERS.DB are linked one→many. CUSTOMER.DB is the master table.

Attaching to multiple linked tables

Figure 10-3 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 10-3 Attaching a TCursor to linked tables


A
B
C
D
tc.attach(A) makes tc point to A
tc.attach(B) makes tc point to B
tc.attach(C) makes tc point to B
tc.attach(D) makes tc point to B

Using TCursors and tables

This section describes how to use TCursors to work with tables. It shows examples of the methods listed in Table 10-3 and describes many common tasks you'll perform with TCursors.

Editing with TCursors

To change data in a TCursor, use **edit**. When your changes are complete, post them to the underlying table by moving off the record or calling **postRecord**. 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
```

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
```

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 PARTS.DB and replaces every occurrence of "sprocket"with "TurboSprocket".

```
var
   tc TCursor
   newName, oldName, tableName String
endVar

oldName = "sprocket"
newName = "TurboSprocket"
tableName = "parts.db"

if tc.open(tableName) then
   if tc.locate("Part name", oldName) then
      tc.edit()
      tc."Part name" = newName ; this line changes the field value
      tc.endEdit()
   else
      msgStop("Stop", "Couldn't find " + oldName)
   endIf
else
   msgStop("Stop", "Couldn't open " + tableName)
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

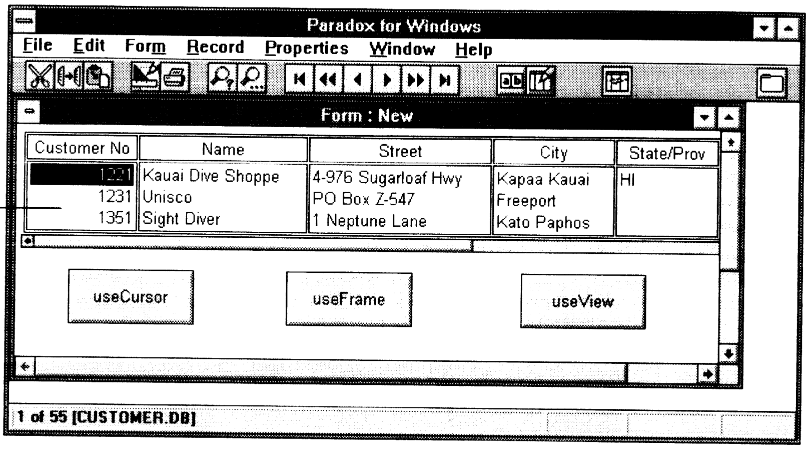
By using TCursors and table frames together, you can process data behind the scenes before displaying the results to the user. (A table frame is a UIObject.) This technique often improves performance. The key methods are the TCursor type method **attach**, which binds a TCursor to a table frame, and the UIObject type method **moveTo**, which synchronizes the table frame to the TCursor.

Figure 10-4 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 10-4 Table frame vs. TableView vs. TCursor

This table frame is bound to Customer.db. By default, its name is CUSTOMER.



Customer No	Name	Street	City	State/Prov
1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy	Kapaa Kauai	HI
1231	Unisco	PO Box Z-547	Freeport	
1351	Sight Diver	1 Neptune Lane	Kato Paphos	

1 of 55 [CUSTOMER.DB]

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.moveTo(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
```

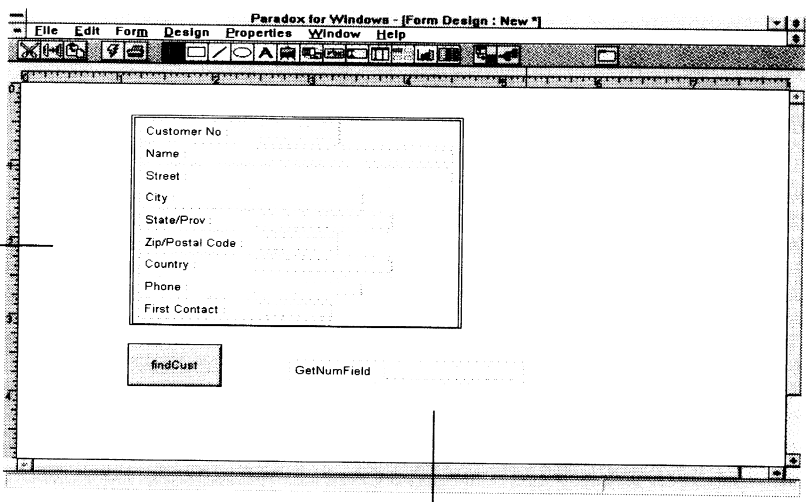
useView The following method is attached to *useView* and is also slower than *useCursor* because it updates the screen after each move.

```
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
```

Searching a table

Figure 10-5 shows a form containing a multi-record object (*CUSTOMER*) bound to the *Customer* table and defined to display one record across and one record down, an unbound field object (*getNumField*), and a button (*findCust*). Type a number in the field object, then click the button. If your number matches a value in the CustomerNo field of the *Customer* table, the corresponding record is displayed in the multi-record object.

Figure 10-5 Using a TCursor with a multi-record object


Paradox for Windows - [Form Design : New *]
File Edit Form Design Properties Window Help
Customer No
Name
Street
City
State/Prov
Zip/Postal Code
Country
Phone
First Contact
findCust
GetNumField
Multi-record object bound to CUSTOMER.DB
Unbound field object named *getNumField*

The **pushButton** method of *findCust* handles the processing.

```
method pushButton(var eventInfo Event)
var
   custTC  TCursor
   custNum String
endVar

custNum = getNumField.value
custTC.attach(customer)           ; associate custTC with the Customer table

if custTC.locate("Customer No", custNum) then  ; if the value is in the TCursor
   customer.moveToRecord(custTC)               ; display the record
   customer.Customer_No.moveTo()               ; move the highlight to the field
else
   msgInfo("Sorry", "Couldn't find " + custNum)
endIf

endmethod
```

This method uses the TCursor to perform the search without any display-processing overhead, and then displays the results in the multi-record object. The same technique works with a table frame.

Note You can accomplish these tasks without using TCursors. The UIObject type provides methods (for example, **locate**) you can use with table frames and multi-record objects. See Chapter 7.

Using moveToRecord and moveTo

Two statements in the previous example call the UIObject type methods **moveToRecord** and **moveTo**. Each statement uses a different syntax, and each achieves a different result.

Using moveToRecord The **moveToRecord** statement operates on the multi-record object named *customer*, instructing *customer* to display the record pointed to by the TCursor *custTC*.

```
customer.moveToRecord(custTC)
```

In effect, this statement synchronizes the multi-record object and the TCursor.

Note The **moveToRecord** method is valid only when the UIObject and the TCursor are associated with the same table. If an object is bound to the *Customer* table and a TCursor is bound to the *Orders* table, the following statement will fail:

```
customer.moveToRecord(ordersTC)
```

The previous statement will also fail if the UIObject is unbound. That is, if the object named *CUSTOMER* is not bound to the *Customer* table, it cannot be "moved" to a TCursor.

Using moveTo The **moveTo** statement, shown here, doesn't involve the TCursor.

```
customer.CustomerNo.moveTo()
```

This statement simply moves the highlight to the *CustomerNo* field object in the *CUSTOMER* table frame and makes the field object active. You can use the **moveTo** method with any UIObject, bound or unbound.

Using moveToRecord with TableView

The TableView type also provides a **moveToRecord** method, and it works like the first **moveToRecord** method discussed earlier. For example, the following statements attach a TCursor to a table window, then locate a record in the TCursor, and 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 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 more information about using **moveToRecord**, see the *ObjectPAL Reference.*

Using reSync

Although **moveToRecord** synchronizes a UIObject to a TCursor, it does not change the UIObject's table description. The UIObject "moves to" the current record of the TCursor, regardless of any filters, indexes, and so on, defined for the TCursor. Use the UIObject type method **reSync** to synchronize a UIObject to a TCursor and take the TCursor's table description into account.

For example, suppose a form contains a table frame bound to the *Customer* table. The following code associates a TCursor variable with the table frame, then calls **setFilter** to change the TCursor's table description. Then, the call to **reSync** makes the table frame display only the filtered data—this example displays records for customer numbers ranging from 2,000 to 4,000:

```
var
   custTC TCursor
endVar
custTC.attach(CUSTOMER)
custTC.setFilter(2000, 4000)
CUSTOMER.reSync(custTC)
```

Note Like **moveToRecord**, **reSync** works only when the UIObject and the TCursor are associated with the same table. See also the discussion of using **attach** with linked and unlinked tables, earlier in this chapter.

Using tables and TCursors to program lists

This section presents two techniques for programming drop-down lists like the one shown in Figure 10-6. 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.

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

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 Form | 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 Methods dialog box.

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)
   self.DataSource = "[customer.phone]" ; build list from Phone field of
                                         ;  Customer table
endMethod
```

6. Run the form.

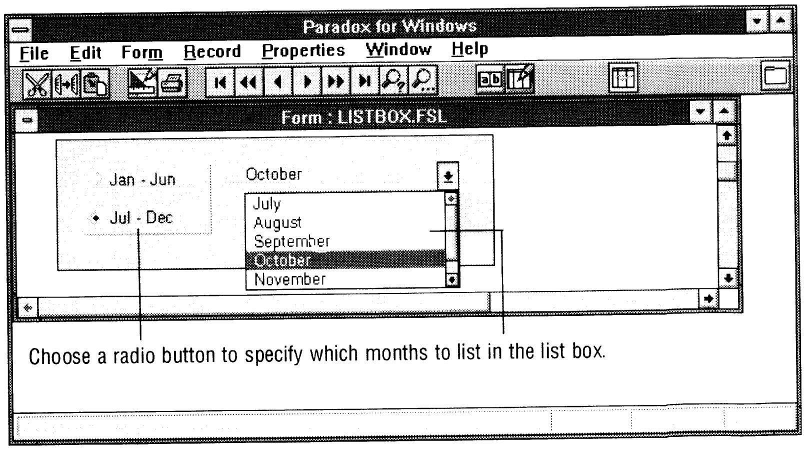
The following code builds a list using the values in the Phone field of CUSTOMER.DB. (The backslashes preceding the interior double quotes are required.)

```
self.DataSource = "[\"customer.db\".phone]"
; backslashes are required to include quotes in a string
```

The following example shows how to program drop-down edit lists like the one in Figure 10-6. 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 10-6 A drop-down edit list



Choose a radio button to specify which months to list in the list box.

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.

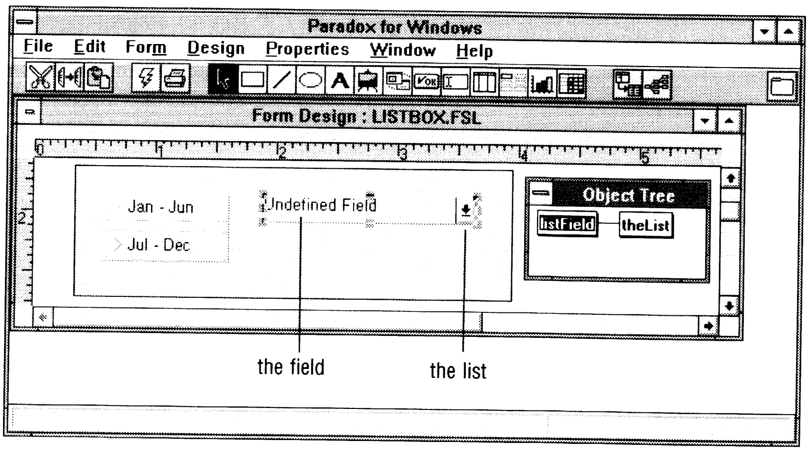
Creating the form

Place two field objects in the form.

1. Choose File | New | Form, and choose OK in the dialog boxes to create a blank, unbound form.
2. Choose the Field tool from the SpeedBar, 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 Form | Object Tree. As the Object Tree diagram in Figure 10-7 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 section.)

Figure 10-7 A drop-down edit field is a compound object

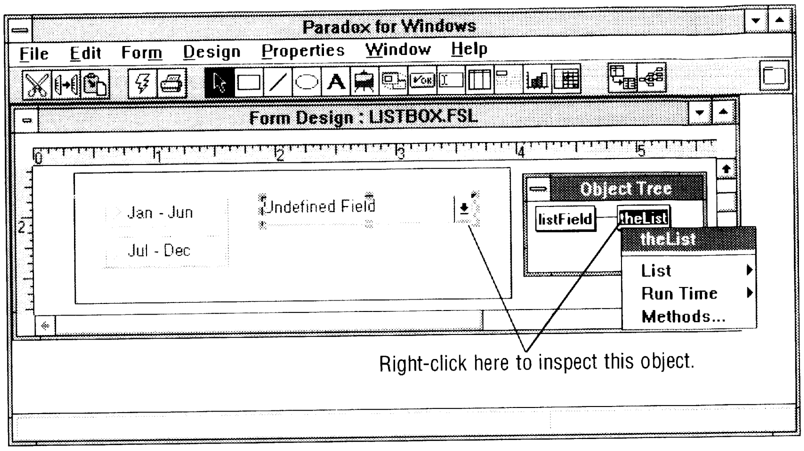
A Drop-down Edit field consists of two objects: the field itself and the list



5. Right-click the list object in the Object Tree to inspect it, as shown in Figure 10-8. Change its name to *theList*. Close the Object Tree window.

Figure 10-8 Inspecting an object using the Object Tree

You can use the Object Tree to inspect an object. Right-click the object's name.


Paradox for Windows
File Edit Form Design Properties Window Help
Form Design : LISTBOX.FSL
Jan - Jun
Jul - Dec
Undefined Field
Object Tree
listField
theList
theList
List
Run Time
Methods...
Right-click here to inspect this object.

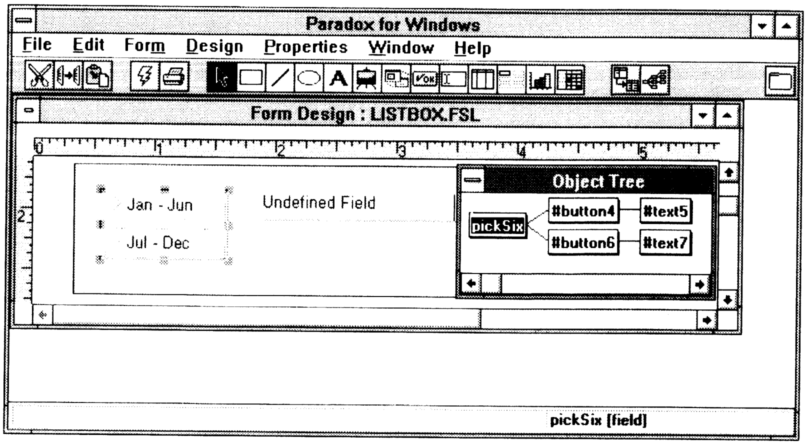
6. Place another field object to the left of *listField*. Inspect this field object, and change its name to *pickSix*.

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 Form | Object Tree again. As Figure 10-9 shows, radio buttons are compound objects, too.

Figure 10-9 Radio buttons shown in the Object Tree



Attaching methods

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

1. Close the Object Tree window.
2. Inspect *pickSix* and choose Methods.
3. From the Methods dialog box, choose **Var**, **open**, **close**, and **newValue**, and then choose OK to open four ObjectPAL Editor windows. 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 ValueEvent)
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. Choose Language | Check Syntax to check your methods for errors, and make corrections, if needed.
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 a drop-down list works

A list is like an array of strings: both index their elements, and counting begins at 1. You work with a drop-down list by manipulating these properties: List.Count, List.Selection, and List.Value.

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

❐ 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.

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

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.

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

Programming an auto-incrementing key field

A common database programming task is to automatically supply a unique key value for each new record. For example, if you're creating an order entry application, you will want to supply a unique order number for each new order. A lesson in *Learning ObjectPAL* shows a simple way to do this in a single-user application. The following example presents a technique you can use in a multi-user application.

Suppose you have already created a form for entering order data, and you want to generate a unique value to identify each new order. First, declare a TCursor in the page's Var window, making it available to all of the page's methods.

```
var
   ordNoTC TCursor
endVar
```

Next, attach the following code to the page's built-in **open** method. This code opens a TCursor onto the table that holds the next order number, and it sets the TabStop property of the *Order_No* field object to False. This prevents the user from moving the cursor into the field object and entering a value.

```
method open(var eventInfo Event)
   ordNoTC.open("OrdNum.db")
   ordNoTC.edit()
   Order_No.TabStop = False
endMethod
```

Tip It's a good idea to store key values in a separate table from the table where you're editing data. In this example, the key values represent order numbers, and they're stored in the *Ordnum* table. That way, you can lock the *Ordnum* table to get a unique order number without locking the entire *Orders* table (which prevents other users from working on it).

The last step is to attach code to the page's built-in **action** method. This code tries to lock the table that holds the next order number. If it succeeds, it assigns a value to the *Order_No* field object in the form and increments the value in the table. If it fails, it opens a dialog box to inform the user.

```
method action(var eventInfo ActionEvent)
   if eventInfo.id() = DataInsertRecord then
      doDefault
      if OrdNoTC.lockRecord() then
         Order_No.value = OrdNoTC.OrderNo + 1
         OrdNoTC.OrderNo = Order_No.value
         OrdNoTC.unlockRecord()
      else
         msgStop("Problem", "Couldn't lock the OrdNum table")
      endif
   endIf
endMethod
```

For more information about creating multiuser applications, refer to Chapter 12.

CHAPTER 11

System data objects

When developing ObjectPAL applications, you will occasionally need information about a user's Windows environment, workstation settings, file directories, and network information. The system data objects let you access and work with this information. These objects are distinct from data model objects because they don't store data in tables. This chapter explains these objects, listed in Table 11-1, and shows how to use them.

Table 11-1 System data objects

Type	Description
DDE	A dynamic data exchange link between Paradox and another application
FileSystem	Provides access to disk files
Library	A collection of custom code
Session	Information about applications and user counts
System	Information about the computer system running the application
TextStream	Text written to and read from disk files

DDE: Link to other applications

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 9.

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.

Items are optional.

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.

Items vary from one application to another.

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 Borland's 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
   dl DDE
   firstName AnyType
endVar

; open a link to the ObjectVision form named Address
dl.open("vision", "C:\\vision\\forms\\Address.ovd", "FirstName")
firstName = dl
msgInfo("First name", firstName)

dl.execute("[@APPEXIT]") ; @APPEXIT is an ObjectVision function
dl.close()
```

FileSystem: Access disk files and directories

The FileSystem type provides methods for working with disk files, drives, and directories. 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 11-2 lists some tasks and the methods to perform them.

Table 11-2 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 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 eample, 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 *Getting Started.*

In addition to the FileSystem methods listed in Table 11-2, 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.

```
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
```

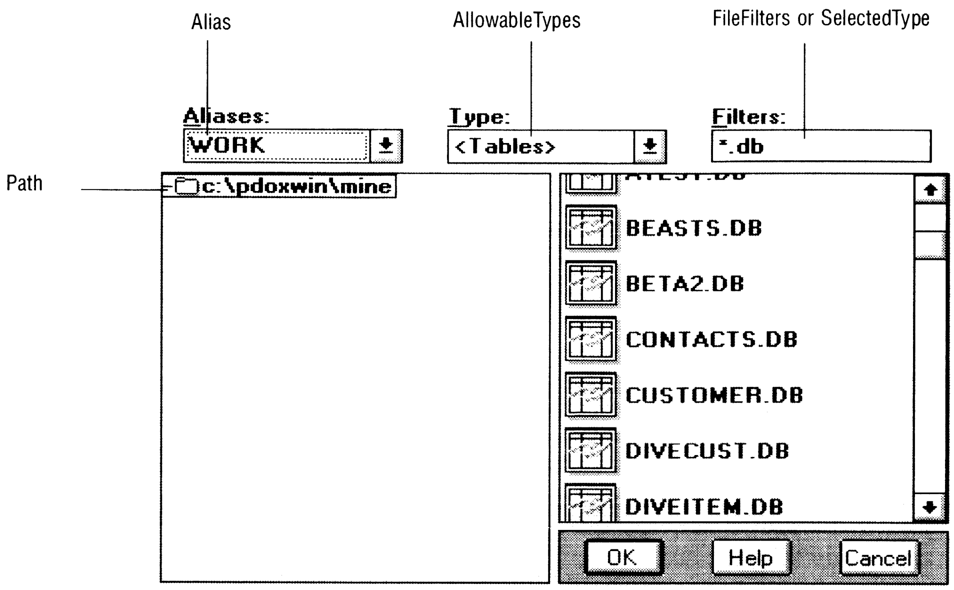
You can also pass a record as an argument to **fileBrowser** to specify what data the Browser displays. For example, you can make the Browser display Paradox tables only, or forms only, or forms and reports.

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 11-3
    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 11-1 shows the Browser and the areas affected by the various fields of the FileBrowserInfo record.

Figure 11-1 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 11-3 lists valid ObjectPAL constants to use in the SelectedType and AllowableTypes fields.

Table 11-3 Constants for the AllowableTypes and SelectedType fields

Constant	Extension	Description
fbFiles	*.*	All files
fbTable	*.db	Paradox tables
fbQuery	*.qbe	QBE files
fbForm	*.fsl, *.fdl	Paradox forms
fbReport	*.rsl, *.rdl	Paradox reports
fbScript	*.ssl, *.sdl	Paradox scripts
fbGraphic	*.bmp, *.pcx, *.tif, *.gif, *.eps	Assorted graphics file formats
fbBitmap	*.bmp	Windows Bitmap graphics

Constant	Extension	Description
fbFiles	*.*	All files
fbText	*.txt	Text files
fbAllTables	*.db, *.dbf	User and system tables
fbTableView	*.tv	Table view files
fbParadox	*.db	Paradox tables
fbDBase	*.dbf	dBASE tables
fbASCII	*.txt	Text files
fbQuattroProWindows	*.wt1	Quattro Pro for Windows worksheets
fbQuattroPro	*.wq1	Quattro Pro for DOS worksheets
fbQuattro	*.wkq	Quattro worksheets
fbLotus2	*.wk1	Lotus worksheets (version 2)
fbLotus1	*.wks	Lotus worksheets (version 1)
fbExcel	*.xls	Excel worksheets
fbIni	*.ini	.INI files
fbLibrary	*.lsl	Paradox libraries

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
```

Library: A collection of custom code

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 Methods dialog box 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 the *ObjectPAL Reference* for more information about built-in object variables.
- The scoping rules are different for libraries. (For more information, see "Controlling the scope of a library" later in this chapter.)

The next sections explain how to use libraries. They cover the following topics:

- Creating a library
- Adding code to a library
- Calling methods from a library
- Using Library type methods
- Controlling the scope of a library

Creating a library

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 Language | Deliver from an Editor window. See Chapter 15 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 which represents the library. *You can't place objects in this window.* Instead, use the Methods dialog box to attach code to the library. To display the Methods dialog box, right-click in the window or press *Ctrl+Spacebar.* Type in a name for the new custom method just as you would for any other object, and choose OK to open another Editor window (see *Learning ObjectPAL*).

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 Methods dialog box 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. For more information, see *Learning ObjectPAL.*

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 5 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 *ObjectPAL Reference* for details about declaring methods.

Adding custom procedures

From a library's Methods dialog box, 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 5 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 Chapter 3 in the *ObjectPAL Reference* for details about declaring procedures.

Declaring variables, constants, data types, and external routines

From a library's Methods dialog box, 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 2 and 5.

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 *ObjectPAL Reference* for more information about using a DLL.)

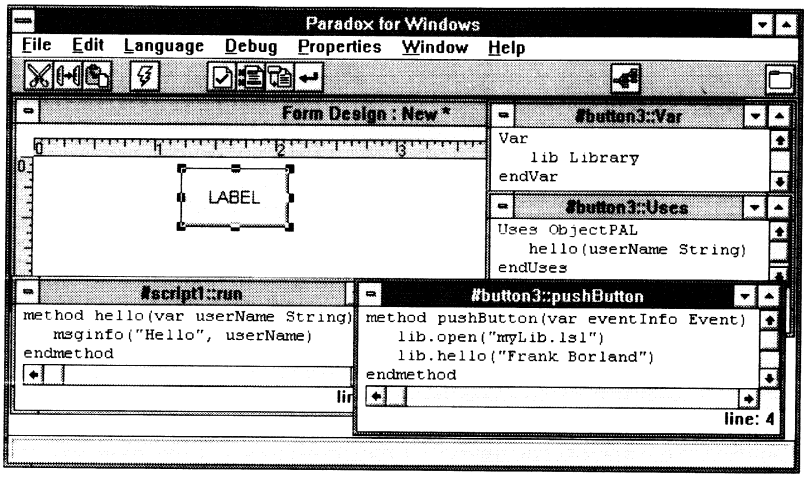
Next, *methodName* represents the name of the method to call, and *argList* represents a comma-separated list of argument/data type pairs, optionally preceded by the keywords **var** and **const**, as appropriate. (Chapter 5 explains how and when to use these keywords.)

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 11-2 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 11-2 Example of calling a custom method from a library



```
Var
   lib Library
endVar
```

```
Uses ObjectPAL
   hello(userName String)
endUses
```

```
method hello(var userName String)
   msgInfo("Hello", userName)
endmethod
```

```
method pushButton(var eventInfo Event)
   lib.open("myLib.lsl")
   lib.hello("Frank Borland")
endmethod
```

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.

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 the *ObjectPAL Reference* for more information about built-in object variables.

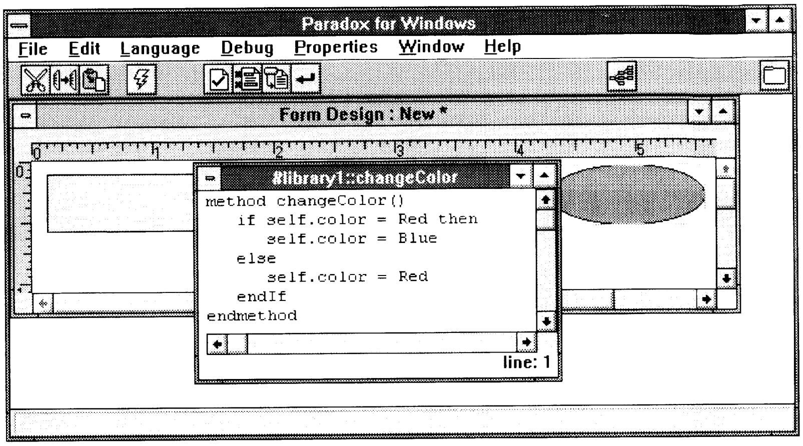
For example, Figure 11-3 shows a form that contains one box named *theBox* and one ellipse named *theEllipse.* The form opens a library that contains a custom method named **changeColor**.

```
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 11-3 Using *Self* in a library routine



```
method changeColor()
   if self.color = Red then
      self.color = Blue
   else
      self.color = Red
   endIf
endmethod
```

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.

Library methods

The Library type has the run-time methods listed in Table 11-4. 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 *ObjectPAL Reference.*

Demonstration forms

The form LIBDEMO.FSL demonstrates how to use a library to create a "thermometer" status indicator. 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 LIBDEMO.FSL. If you did not install the sample files, you can do so now. Follow the instructions in *Getting Started.*

Also, libraries are used in the sample application.

Table 11-4 Library methods

Method	Description
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).
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.
enumSource	Writes library code to a Paradox table.
enumSourceToFile	Writes library code to a text file.
execMethod	Executes a specified library method.

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

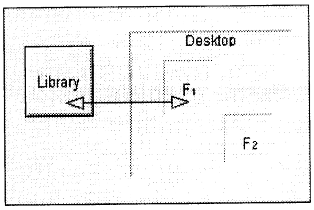
The Library type method **open** takes arguments that specify the scope. A library can be opened in one of the following ways:

- 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 11-4 shows the various **open** options.

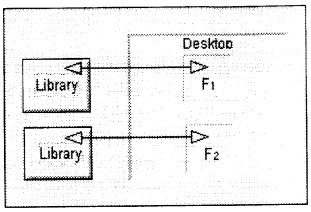
Figure 11-4 Using **open** to specify library scope

Diagram A



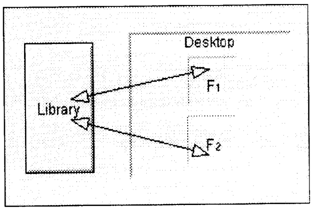
lib.open("mylib.lsl", PrivateToForm)

Diagram B



lib.open("mylib.lsl", PrivateToForm)

Diagram C



lib.open("mylib.lsl", GlobalToDesktop)

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 11-4, 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.

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 11-4, 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
```

Session: A channel to the database engine

A Session variable represents a channel to the database engine. Opening a Paradox application opens one session by default, 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 11-5 lists some Session tasks and the methods to perform them.

Table 11-5 Session tasks and methods

Task	Methods
Add to a Session	addAlias, addPassword
Session settings	blankAsZero, ignoreCaseInLocate, setRetryPeriod
Get information about a Session	getAliasPath, getNetUserName, retryPeriod
Enumerate information to tables	enumAliasNames, enumDatabaseTables, enumUsers

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:

- 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.

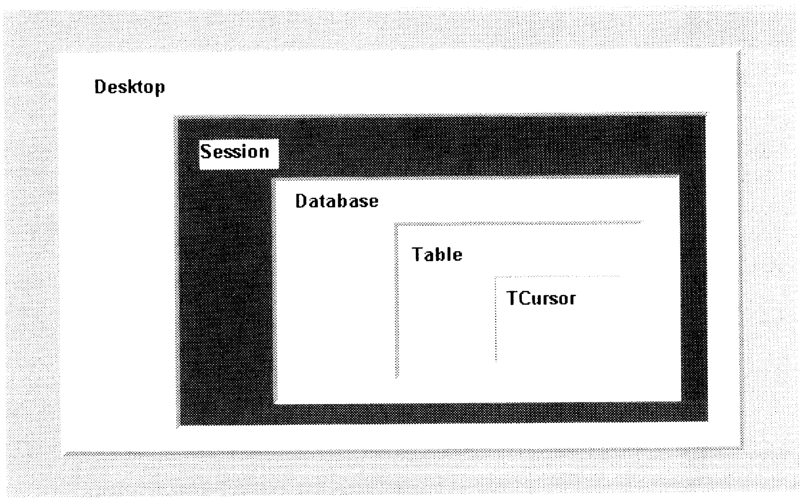
When the following code executes, it creates a handle to the default session (*defaultSes*) and opens a second session (*secondSes*).

```
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 11-5 diagrams this scoping hierarchy. (Refer to Chapter 10 for information about databases, tables, and TCursors.)

Figure 11-5 Session scoping hierarchy



You can use ObjectPAI 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
```

System: A catch-all type

The System type is a catch-all for methods (like **message, beep,** and **sleep**) that don't fit into other categories. The System type also includes methods (like **enumFontsToTable**, **readEnvironmentString**, **readProfileString**, and **sysInfo**) that report about the system the application is running on.

In addition, you'll find methods (like **msgInfo** and **msgYesNoCancel**) for displaying ObjectPAL's built-in dialog boxes, and **execute**, which executes DOS commands and Windows applications.

The following example opens the clock application that comes with Windows (the file CLOCK.EXE must be on your path).

```
execute("clock.exe")
```

This catch-all type also includes procedures for converting screen locations back and forth between pixels and twips. The names are **pixelsToTwips** and **twipsToPixels**.

Using Help

ObjectPAL provides a way for you to call the standard Windows Help application. The Help application is a separate Windows application. To use it, you must create a file of help information and context strings and compile it using the Microsoft Help compiler (included with Borland C++). The documentation that comes with the Help compiler explains in detail how to build and compile a Windows Help system.

For examples of how to do this, refer to the MAST sample application in your DIVEPLAN directory and the example applications online.

Once your help system is built and compiled, you can use the following methods to access it. These methods are documented in the *ObjectPAL Reference*:

- **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 the Help associated with a keyword stored in an alternate keyword table.

See also For more details about building ObjectPAL applications, see Chapter 15.

TextStream: Work with text files

A TextStream is 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 9.) 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. Counting begins with 1 (not with 0, as in some other languages).

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 11-6 lists tasks and the TextStream methods to perform them.

Table 11-6 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, readString, 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). **readString** 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(28)
   ; next read or write will begin at the 28th
   ; character in the file, the "m" in men's

   string3 = "women's souls."
   txt1.writeString(string3)
   txt1.close()

endMethod
```

PART III

Special topics

This part of the manual is divided into the following chapters:

- Chapter 12, "Multiuser applications," discusses how to build applications used on a network.
- Chapter 13, "Errors and error processing," explains how to handle errors.
- Chapter 14, "Creating and playing scripts," explains how to create and play ObjectPAL scripts.
- Chapter 15, "Packaging and delivering ObjectPAL applications," discusses how to deliver ObjectPAL applications to end users.

CHAPTER 12

Multiuser applications

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
- ❐ Use passwords to control access
- ❐ Use locks to control access
- ❐ Implement interprocess communication
- ❐ Work with (and program) aspects of multiuser applications

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
- Password protection, which provides precise access control
- Locking, which facilitates data integrity

- Interprocess communication, needed by some applications

The following sections discuss each of these tools.

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 Chapter 10. For information about using aliases interactively, see the *User's Guide.*

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 12-1.

Table 12-1 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** 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 a' tomatic 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 12-2 shows the locks you can place, in order of *decreasing* strength and *increasing* concurrency.

Table 12-2 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 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 12-3.

Table 12-3 Available locks for Paradox and dBASE tables and TCursors

	TCursor var	Table var	Quoted table name
Read	P	P*	P*
Write	P D	P* D*	P* D*
Full	P D	P D*	P D*

P = Paradox Table D = dBase Table *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 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 keywords 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, or until the session ends. 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 (also called an *exclusive* lock) prevents other users from accessing the table in any way—it gives you exclusive 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."

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.

```
Proc someInitProc()
var salesTbl Table endVar
   salesTbl.attach("sales.db")
   IF NOT lock(salesTbl, "FULL", "orders.db", "FULL") then
      ; strategy to deal with locked resources...
   ENDIF
endProc
```

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, described later in this chapter) 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 keyword 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 cursor off the record.

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 syncronization.

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.

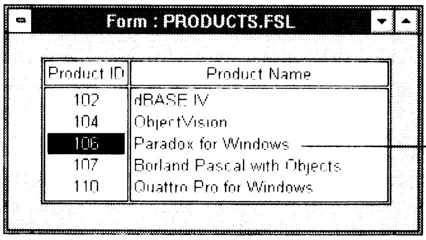
To help you, Paradox provides a read-only property called FlyAway for fields, records, table frames, multi-record objects, and forms. FlyAway is True if the most recent record unlock (or record post) caused the record to move from its current position; if the record stayed where it was, FlyAway is False. 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.

The following example forces the system to stay on the same visual record of the table frame, even when a commit causes a flyaway. To do this, the example cancels the move normally triggered by **canDepart**, which is a built-in method that's called when a record is committed.

```
method canDepart(var eventInfo moveEvent)
   if self.locked then         ; if record is locked
      doDefault                ; go ahead and do the commit
      if self.FlyAway then     ; if the record flew away to new location
         self.moveTo()         ; then, move to it
      endIf
   endIf
endMethod
```

For example, suppose a form contains a table frame (keyed on the Product ID field) as shown in Figure 12-1. 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 12-1 Using the FlyAway property



Form : PRODUCTS.FSL

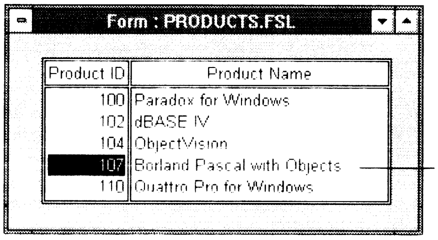
Product ID	Product Name
102	dBASE IV
104	ObjectVision
106	Paradox for Windows
107	Borland Pascal with Objects
110	Quattro Pro for Windows

When you change 105 to 106, this record doesn't fly away; however, if you change the Product ID to 100, this record does fly away.

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 12-2. 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 12-2 More about the FlyAway property



Form : PRODUCTS.FSL

Product ID	Product Name
100	Paradox for Windows
102	dBASE IV
104	ObjectVision
107	Borland Pascal with Objects
110	Quattro Pro for Windows

After record 3 flies away to its new position (record 1), the TCursor points to record 4.

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 does not move outside of its immediate neighbors before and after. 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 flyaway 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" earlier in this chapter.
- 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, then delete, a record the lock is removed by the delete operation. (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 preferrable 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.

Used together, a write lock and a read lock provide the optimal combination of locking and performance. Because other users cannot modify a table, this combination provides fast I/O; however, it still allows you to edit the table.

CHAPTER 13

Errors and error processing

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 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 bug.

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 utilize the powerful error-handling capabilities of Paradox. They are

- ❒ Run-time error levels
- ❒ The error stack
- ❒ The TRY...ONFAIL 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.

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
```

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

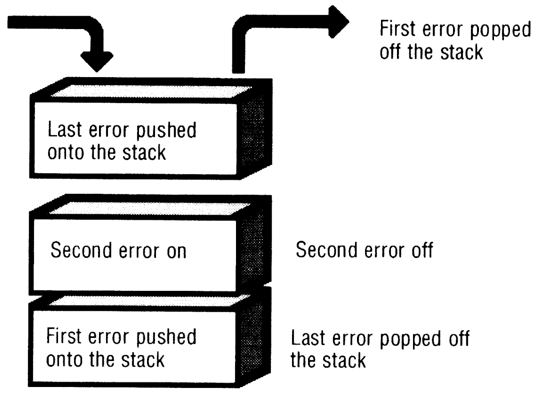
Paradox provides a simple mechanism called the *error stack*, which returns information about the most recently detected run-time error. Each time a run-time error occurs, ObjectPAL stores information about the error on the error stack. You can use a number of methods to examine and even modify this 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 13-1, 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 13-1 The error stack


First error popped off the stack
Last error pushed onto the stack
Second error on
Second error off
First error pushed onto the stack
Last error popped off the 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

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 Language | Constants. Then, from the Types of Constants column, choose Errors. The constants appear in the Constants column. Constants are also listed in Appendix H in the *ObjectPAL Reference*. You can use these constants in methods as shown in the examples in this chapter.

Using errorMessage

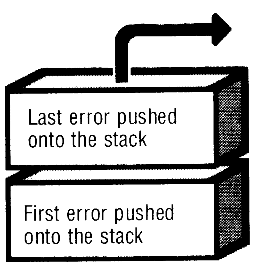
errorMessage returns the text of the error message on the top of the error stack. A typical use for **errorMessage** is to log error messages to a file. Another use is to combine the default error message with a custom message. The following code shows both uses by adding a custom error-logging routine 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 13-2). (To remove all layers from the stack, use **errorClear**

Figure 13-2 The error stack: pushing and popping information


Last error pushed onto the stack
First error pushed onto the stack

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

try statements

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:

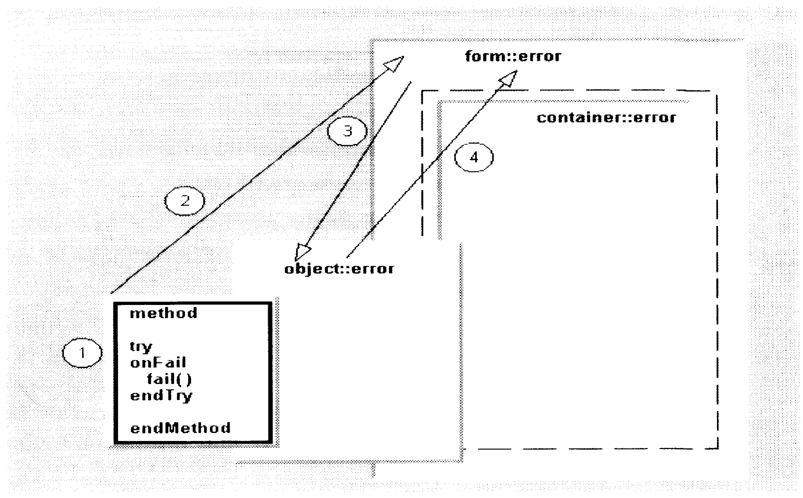
```
TRY
   [transaction block]
ONFAIL
   [
      recovery block
      [RETRY]                        ; optional keyword
      [FAIL ( [ errCode, errMsg] ) ]  ; optional procedure, optional arguments
   ]
ENDTRY
```

If the transaction succeeds, the program skips to the ENDTRY keyword and the error stack is cleared. If any statement within the transaction fails, 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.

You can nest one or more explicit TRY...ONFAIL blocks around critical sections of code. When a transaction block fails, the error bubbles 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 13-3.

Figure 13-3 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, 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 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 block, 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.

The event model is described in Chapter 6.

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.

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 block on 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
```

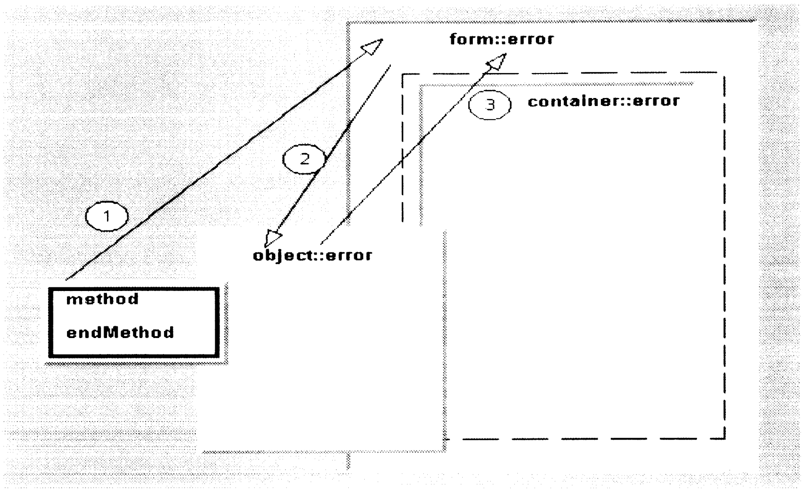
Note The **peBreak** error code is generated when a user interrupts program activity by typing *Ctrl+Break*. Therefore, depending on the needs of your application, it might be particularly useful to check for the **peBreak** error code in the ONFAIL block of critical operations.

error method

As Figure 13-4 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 13-4 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 onWarnings
- 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 6 for more information about StatusEvents.

Warning errors and TRY...ONFAIL

By default, a TRY...FAIL 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...FAIL 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...FAIL 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)
```

or

```
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.

CHAPTER 14

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, but it has no type, it doesn't appear in a window, and it 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.

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 upper case 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, check syntax, 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 Methods dialog box by choosing Language | 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 Language | Deliver. Saved scripts can be changed; delivered scripts cannot. See Chapter 15 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
```

CHAPTER 15

Packaging and delivering ObjectPAL applications

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, SpeedBar, 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
- SpeedBar position
- Ruler style

The Paradox development Desktop may have additional properties such as run/design mode and debugging and editing preferences.

The Desktop tracks passwords.

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 only one application at a time. However, multiple instances of Paradox can run concurrently on the same Windows platform and in this way support multiple concurrent Paradox applications.

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 11 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 15-1.

Table 15-1 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
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 15-2.

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 15-2.

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 15-2 File name extensions for saved and delivered objects

Object	Saved	Delivered
Form	.FSL	.FDL
Library	.LSL	.LDL
Report	.RSL	.RDL
Script	.SSL	.SDL

Performance tip For best performance, turn off the Enable Debug, Trace Execution, Trace Built-ins, and Enable Ctrl + Break To Debugger options. Refer to Chapter 4 for more infomation about these options.

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 Form | 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 Language | 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 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 MAST application and the examples included online. Also, refer to the entries for these methods in the *ObjectPAL Reference.*

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 preceeding 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 (listed in Appendix B of the *ObjectPAL Reference).* 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 the *User's Guide*) 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 15-3 lists the methods ObjectPAL provides for this purpose. All of the methods listed belong to the String type.

Table 15-3 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
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 Language | Browse Sources. See Chapter 3 for more information about the ObjectPAL Editor.

APPENDIX A

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* and *Getting Started.*

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 2 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 4 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. Chapters 7 and 8 discuss forms, and Chapter 14 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 8 discusses forms and dialog boxes.

Libraries

Libraries in ObjectPAL are similar in concept to PAL procedure libraries. Both save memory and ease maintainance 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 8 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 7 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 analagous 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 5 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 5. 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...FAIL 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.

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 10 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 (see **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 15 for more 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.

Glossary

active An ObjectPAL variable that represents the object that has focus (described below).

ANSI An acronym for American National Standards Institute; a sequence of eight-bit codes that defines 256 standard characters, letters, numbers, and symbols. The ASCII character set (see below) is the same as the first 128 ANSI characters.

application An ObjectPAL type that provides a handle to the Paradox Desktop.

Also a group of forms, methods, queries, and procedures forming a single unit, where users can enter, view, maintain, and report on their data.

argument Information passed to a method or procedure.

array A special kind of object that consists of several separate elements. Items in an array are designated by subscripts enclosed in square brackets, so that ar[1] and ar[2] are the first two items of an array named *ar*.

array element One component of an array. For example, the array created with the following declaration has seven string elements, ar[1] through ar[7].

```
ar Array [7] String
```

Elements are also called items.

ASCII An acronym for American Standard Code for Information Interchange; a sequence of seven-bit codes that define 128 standard characters, letters, numbers, and symbols. The IBM PC extends this code to 8 bits to include some special graphic characters. Windows products use the ANSI character set (described above).

blank A field or variable that has no value.

braces	The symbols { and }. Braces mark comments in code.
branching commands	Control structures that perform specific commands depending on whether certain conditions are met. Examples: **if, while**.
breakpoint	A flag set in source code that causes execution to suspend. Used in debugging.
bubbling	A process by which events pass from the target object up through the containership hierarchy.
built-in method	Pre-defined code that comes with every object you can place in a form. Built-in methods define an object's default response to events.
column	A vertical component of a Paradox table that contains one field. In a Paradox report, a vertical area containing one or more fields.
compound object	An object made up of two or more other objects. For example, a table frame is a compound object made of field objects and record objects.
concatenation	The joining of two or more strings to form a single string. The concatenation operator is a plus (+) sign.
constant	A constant represents a value that cannot be changed. For example, DataNextRecord is an ObjectPAL constant that specifies a move to the next record in a table.
containership	One object contains another object if the other object is completely within the borders of the first object. Containership affects the avaiability of variables, methods, and procedures.
control structure	A sequence of branching statements, such as **if...then... endIf** or **while...endWhile**, that affects the order in which statements execute.
Ctrl+Break	A key sequence that halts program execution. You can configure Paradox to respond to *Ctrl+Break* by opening the Debugger.
data	The information Paradox stores in a table.
Database	An object type that contains information about relationships between tables.
data type	The type of data that a field, variable, or array element contains.
DDE	Acronym for Dynamic Data Exchange. A way for Windows applications to share data.

deadlock	A situation created in a multiuser environment when two incompatible lock commands are issued concurrently.
Debugger	Part of the ObjectPAL Integrated Development Environment (IDE), the Debugger lets you interactively test and trace execution of commands in your methods.
Desktop	The main window in Paradox.
Display manager	A category of object types that includes Application, Form, Report, and Table View.
DLL	Acronym for Dynamic Link Library. A DLL is a program that allows Windows programs to share code that performs common tasks.
dynamic array	A special kind of array where each item has a string for an index. For example, ["Product"], ["Paradox for Windows"], ["Type"] ["Relational database"], ["Version"][1.0]
Editor	The component of the Paradox IDE used to create and edit ObjectPAL methods.
encrypt	To translate a table or script into code that cannot be read without the proper password.
event	The action that triggers a method (that is, causes code to execute). Also an object type (Event).
event-driven application	An application where code executes in response to events, as opposed to a procedural application, where code executes in a linear sequence.
event model	The rules that specifiy how events are processed by objects in a form.
example element	In a query statement, an arbitrary sequence of characters that represents any value in a field. In Paradox, you indicate an example element by pressing Example *F5* and typing the characters in the query image. In methods, you indicate example elements by preceding the characters with an underscore.
expression	A group of characters that can include data values, variables, arrays, operators, or functions that represent a quantity or value. An expression can evaluate to a specific data type or, in certain cases, first be converted to string values before it is evaluated.

field	One item of information in a table. A collection of related fields makes up one record.
field assignment	Use of dot notation to assign the value of an expression to a field.
field object	A UIObject that may or may not be associated with a field in a table.
field type	The kind of information that can be entered into a field of a table. *See also* Data type.
field value	The data contained in one field of a record. If no data is present, the field is considered blank. Field objects have a Value property.
Field View	Lets you move the insertion point through a field, character by character. It is used to view field values that are too large to be displayed in the current field width, or to edit a field value.
file	A collection of information stored under one name on a disk. For example, Paradox tables are stored in files.
FileSystem	An ObjectPAL type. FileSystem variables contain information about disk files.
focus	An attribute of an object. An object that has focus (also called the active object) is ready to handle keyboard input. Typically, the active object is highlighted.
format specification	The way in which a field value is displayed onscreen or output to a printer.
form	A window for displaying data and objects. Also an ObjectPAL type (Form). The form is the highest-level container object.
function keys	The 12 keys across the top of the keyboard. (Some keyboards have 10 keys at the far left of the keyboard labeled *F1* through *F10*.)
global variable	A variable available to all objects in a form. *See also* local variable.
handle	A variable you can use in code to manipulate objects.
Help	The Paradox online Help system. You can press *F1* at any point in Paradox to display information about the current operation.
hierarchy	The relationship of objects in a form, derived from their visual, spacial relationship. *See also* containership.

IBM extended codes	Keys or combinations of keys on the keyboard that do not correspond to any of the standard ASCII character codes and are given special "extended code" numbers between -1 and -132.
incremental development	A process of application development in which small parts or the general structure of the application are designed and tested interactively.
index	A file that determines the order in which Paradox can access the records in a table. The key field of a Paradox table establishes its primary index. *See also* key, secondary index.
insertion point	The place where text is inserted when you type. The insertion point is usually represented by a flashing vertical bar.
inspect	To view or change an object's properties. To inspect an object, either right-click it or select it with the keyboard and press *F6*. The object's menu appears. Choose from the menu the property you want to change.
key	A field or group of fields in a used to order records. *See also* key field.
keycode	A code that represents a keyboard character in ObjectPAL methods. May be an ANSI number or a string representing a key name known to Paradox.
key field	A field designated as all or part of an identifier of the records in a Paradox table. Establishing a key has three effects: the table is prevented from containing duplicate records, the records are maintained in sorted order based on the key fields, and a primary index is created for the table. *See also* index.
keyword	A word reserved by ObjectPAL. A keyword must not be used as the name of a variable, array, method, or procedure.
library	A collection of ObjectPAL code that can be used by objects in one or more forms.
lifetime	The length of time an item is active or available.
link key	In a linked multi-table form, the part of the subordinate table's key that is linked or matched to fields in the master table.
local variable	A variable that is available only to the method or procedure in which it is declared. *See also* global variable.

logical operator	One of three operators (AND, OR, or NOT) that can be used on logical data. For example, an AND between two logical values results in a logical value of True if both the original values are also True. Also known as Boolean operators.
logical value	A value (True or False) assigned to an expression when it is evaluated. Also known as a Boolean value.
loops	Control structures that repeat a series of commands until a certain condition is met. *See also* control structures.
menu	A display of the choices or options available. Using ObjectPAL, you can create and edit both application menus and pop-up menus.
menu choice	A command chosen from a menu.
message	A string expression displayed in the status line.
method	ObjectPAL code attached to an object that defines the object's response to an event.
normalized data structure	An arrangement of data in tables where each record includes the fewest number of fields necessary to establish unique categories. Rather than using a few records to provide all possible information, a normalized table spreads out information over many records using fewer fields. A normalized table provides more flexibility in terms of analysis.
object	An encapsulation of code and data. All entities that can be manipulated in Paradox are objects.
Object Tree	A diagram that shows how objects in a form are related in terms of containership.
parameter	The variable into which an argument is passed. Used in defining procedures.
picture	A pattern of characters that defines what a user can type into a field during editing or data entry, or in response to a prompt.
pixel	A single point on the screen. The name comes from *pic*ture *el*ement
point	An ordered pair of numbers that represents a location on the screen.
pointer	A visual marker that indicates the mouse location on screen.

post	Accept changes to a record and put the data into the table. Also called commit.
primary index	An index on the key fields of a Paradox table. A primary index determines the location of records, lets you use the table as the detail in a link, keeps records in sorted order, and speeds up operations. *See also* key, secondary index.
procedure	Code bracketed by the keywords PROC and ENDPROC. Unlike a method, it has no object to give it context.
prompt	Instructions displayed on the screen, usually in the status bar. Prompts ask for information or guide the user through an operation.
properties	The attributes of an object. You right-click an object to view or change its properties. *See also* inspect.
QBE	*See* Query by example.
query	A question you ask about information in a Paradox table, formulated in a query form. Also an ObjectPAL type (Query).
query by example (QBE)	The method of asking questions about data by providing examples of the answers you're looking for.
quoted string	Text enclosed in double quotation marks.
raster operation	An operation that specifies how colors are blended on the screen.
record	A horizontal row in a Paradox table that contains a group of related fields of data. Also an ObjectPAL type (Record).
record number	A unique number that identifies each record in a table.
relational database	A database design in accordance with a set of principles called the *relational model.* Data in a relational database must be organized into tables.
reserved words	The names of commands, keywords, functions, system variables, and operators. These words may not be used as ObjectPAL variables or array names. *See also* keyword.
restricted view	A detail table on a multi-table form, linked to the master table on a one-to-one or one-to-many basis, limited to showing only those records that match the current master record.

row	A horizontal component of a table, called a record, in Paradox.
run-time error	An error that occurs when a syntactically valid statement cannot be carried out in the current context.
run-time library	A collection of pre-defined methods and procedures that operate on objects in specific types.
scope	The accessibility or availability of a variable, method, or procedure to other objects.
script	A collection of ObjectPAL code that executes without opening a window.
secondary index	An index used for linking, querying, and changing the view order of tables.
Self	An ObjectPAL variable. Self refers to the object to which the currently executing code is attached.
session	A channel to the database engine. Also an ObjectPAL type (Session).
slash sequence	A backslash followed by one or more characters, to represent an ASCII character. Examples are \" or \018. Slash sequences are used for placing quotation marks within strings and including other characters that have special meaning to Paradox.
string	An alphanumeric value, or an expression consisting of alphanumeric characters. Also an ObjectPAL type (String).
structure	The arrangement of fields in a table.
subject	The object used to call a custom method. For example, in the following statement, *theBox* is the subject. `theBox.doSomething()`
substring	Any part of a string.
syntax error	An error that occurs due to an incorrectly expressed statement.
TableView	The representation of a table in Paradox table format in rows and columns. Also an ObjectPAL type.
target	The object for which an event is intended. For example, when you click a button, the button is the target.

TCursor	An ObjectPAL type. A pointer to the data in a table. Using TCursors, you can manipulate data without having to display the actual table.
tilde variable	A variable used in a query form, which must be preceded by a tilde (~).
transaction	A group of related changes to a database.
twip	A unit of measurement equal to 1/1440 of an inch (1/20 of a printer's point).
type	A way of classifying objects that have similar attributes. For example, all tables have attributes in common, and all forms have attributes in common, but the attributes of tables and forms are different. Therefore, tables and forms belong to different types.
validity check	A constraint on the values you can enter in a field. Sometimes called a *val check*.
variable	A place in memory to store data temporarily.

INDEX

PARADOX
FOR WINDOWS

BORLAND

Corporate Headquarters: 1800 Green Hills Road, P.O. Box 660001, Scotts Valley, CA 95067-0001, (408) 438-8400. Offices in: Australia, Belgium, Canada, Denmark, France, Germany, Hong Kong, Italy, Japan, Korea, Malaysia, Netherlands, New Zealand, Singapore, Spain, Sweden, Taiwan, and United Kingdom ▪ Part # PDX1110WW21773 ▪ BOR 2717