

# **Oracle Power Objects™ User's Guide**

*Version 1.0  
Part No. A25660-5  
July 1995*

Oracle Power Objects User's Guide, Version 1.0

Part No. A25660-5

Copyright © Oracle Corporation 1995

Contributing Authors: Thomas Grant, Jeff Levinger, Christopher Roberts

Contributors: Matthew Bennett, John Butcher, Paul Genteman, Adam Greenblatt, Tanya Jenkins, Jennifer Krauel, Ronnie Lashaw, David Levine, Diana Lorentz, Steven McAdams, Michael Roberts, Ned Young

This software/documentation contains proprietary information of Oracle Corporation; it is provided under a license agreement containing restrictions on use and disclosure and is also protected by copyright law. Reverse engineering of the software is prohibited.

If this software/documentation is delivered to a U.S. Government Agency of the Department of Defense, then it is delivered with Restricted Rights and the following legend is applicable:

**Restricted Rights Legend**

**Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of DFARS 252.227-7013, Rights in Technical Data and Computer Software (October 1988).**

**Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.**

If this software/documentation is delivered to a U.S. Government Agency not within the Department of Defense, then it is delivered with "Restricted Rights", as defined in FARS 52.227-14, Rights in Data - General, including Alternate III (June 1987).

The information in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. Oracle Corporation does not warrant that this document is error-free.

Oracle and SQL\*Net are registered trademarks and Oracle7, PL/SQL, and Oracle Power Objects are trademarks of Oracle Corporation.

All other products or company names are used for identification purposes only, and may be trademarks of their respective owners.

These programs were not developed for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It is the customer's responsibility to take all appropriate measures to ensure the safe use of such applications if the programs are used for such purposes.

---

# Preface

Welcome to Oracle Power Objects, a powerful object-oriented, client/server development tool. Oracle Power Objects is designed to help you develop both simple and advanced applications quickly and easily.

This preface has the following sections:

- Who Should Use This Manual?
- Conventions Used in This Manual
- The Oracle Power Objects Documentation Set
- Getting Help
- What Is Oracle Power Objects?

---

## Who Should Use This Manual?

This manual is designed for anyone using Oracle Power Objects, including the following:

- Intermediate and advanced developers in a GUI environment.
- Intermediate and advanced database application developers.

This manual assumes the following knowledge:

- Familiarity with relational database system (RDBMS) concepts.
- Familiarity with the operating system used on client systems.
- General experience with development concepts and techniques.

## Conventions Used in This Manual

### Text Conventions

The following text conventions indicate special terms in this manual.

<b>Convention</b>	<b>Type of Term</b>	<b>Examples</b>
<b>boldface</b>	button text labels	the <b>Cancel</b> button
	menu commands	the <b>File-Open...</b> menu command
	property and method names	the <b>RecordSource()</b> property the <b>Validate()</b> method
monospace	text to be entered by the reader	type the value <code>Smith</code>
	variable names	the variable <code>vEmpName</code>
ALL CAPS	SQL keywords	SELECT COMMIT
	table and column names	the EMP table the ENAME column
“in quotes”	file names	the “makedemo.sql” script
	object names	the “Total Salary” field the “Departments” form

## Keyboard

In this manual, the term “Return key” refers to the Return key on a Macintosh and to the Enter key in Microsoft Windows.

## Mice and Other Pointing Devices

Microsoft Windows and the Macintosh both support a wide array of mice and other pointing devices. Usually, a Macintosh mouse has only one button, while a PC mouse has two. In this manual, unless otherwise stated, the term “mouse button” refers to the standard mouse button on a Macintosh, to the *left* mouse button in Microsoft Windows.

## The Oracle Power Objects Documentation Set

The documentation accompanying Oracle Power objects includes the following:

- **Context-sensitive online help.** The online help file, PWROBJX.HLP, includes a complete language reference, descriptions of the objects used during development, and guidelines for completing many common development tasks. For more complete information on many important components of Oracle Power Objects, you should consult this User's Guide.
- *Getting Started with Oracle Power Objects.* This book provides a quick overview of Oracle Power Objects, as well as a tutorial. The Getting Started manual uses one of the sample applications, MLDEMO.POA, to illustrate how to perform many development tasks. The tutorial in the User's Guide provides a general introduction to Oracle Power Objects; for a more complete grasp of many aspects of Oracle Power Objects, you should consult this User's Guide.
- *Oracle Power Objects User's Guide.* The book you now hold explains all the important concepts behind Oracle Power Objects, describes in depth the components of an Oracle Power Objects database application, explains how to complete many important development tasks, and provides many important kinds of reference material. For a quick introduction to Oracle Power Objects, you should follow the tutorial in the Getting Started manual.
- **Sample Applications Guide.** This reference describes the development tasks illustrated in the sample applications provided with Oracle Power Objects.

---

## Getting Help

The online help file, PWROBJX.HLP, is context sensitive. In Windows, you can invoke context sensitive help by pressing F1 while the focus is on a particular component of the Oracle Power Objects desktop. The online help file opens and displays a relevant topic. In addition, you have several other ways to reach the reference material provided in the online help file:

- If you want to see the Table of Contents, choose the **Help-Contents** menu command. You can then navigate through several lists of functionally related topics.
- By entering a keyword in the top window, you then see a list of related topics appear in the bottom window. You can jump to a topic by double-clicking on its title in the bottom window.

## What Is Oracle Power Objects?

Oracle Power Objects is a software development tool that combines the database capabilities of high-end tools with the ease of use of low-end tools. In addition, Oracle Power Objects is designed for developing client/server applications, and it provides the benefits of an object-oriented development model.

### Object-Oriented Development

Object-oriented development is explained in Chapter 1, "Application Development with Oracle Power Objects" and Chapter 3, "Objects".

In Oracle Power Objects, every component of the application, from the database (or "back end") to the client interface (or "front end") is an object that can be identified and manipulated using the same techniques. All the objects with which you'll be working have properties and, in most cases, methods, regardless of the type of object.

This object-oriented approach to development simplifies the task of creating applications, and it reflects more closely than other development models how you think about the components of an application. Generally, when you work with the definition of a form, a report, or a table, you think about how to manipulate some aspect of the object. In the case of a form, for example, you try to determine its proper height or width, or more importantly, what tables or views should be represented on the form. By changing a property of a form, therefore, you set some significant characteristic of it.

Similarly, you want objects to perform actions or respond to them. For example, you want the form to do something when the user presses the **OK** button. By adding code to a method that is part of the pushbutton, you can determine what happens when the user pushes the button.

In other words, instead of using a different kind of tool for each kind of object, you use the same techniques and the same tool for all objects that are part of the database application.

## Local and Remote Database Capabilities

When you develop a database application, you do not want to be limited to a particular database platform. In addition, you may want to run a prototype of your application on a local database before tying up or modifying the remote server, which may be in use by many other clients. Once you have finished prototyping the application, you can then export the definitions of all the tables, views, sequences, and indexes to the remote server, minimizing the downtime needed to implement a new client/server application.

Oracle Power Objects provides connections to many leading relational database engines, including Oracle7, Microsoft SQL Server, and Sybase SQL Server. Additionally, on the client, Oracle Power Objects provides connectivity to Personal Oracle7 and to Blaze, a local RDBMS engine provided with Oracle Power Objects. Aside from providing an alternative to remote connectivity for many applications, both Oracle7 and Blaze can be powerful prototyping aids.

In addition to this flexibility in database platform choices, Oracle Power Objects further helps you develop applications by automating many aspects of database connectivity. By dragging the description of a table onto a form, you automatically connect the form to that table. This feature lets you focus on designing the application, instead of writing code to connect the front end to the back end.

## Familiar Programming Language

Oracle Power Objects uses Oracle Basic, a VBA-compliant programming language, for all the code you write. Oracle Basic follows the conventions of Visual Basic for Applications, so if you have programmed applications in Microsoft Visual Basic or Microsoft Access, you are already familiar with Oracle Basic.

## Rapid Application Development

All of the features described so far let you develop applications more quickly than has been possible before. In addition, Oracle Power Object's GUI-based interface makes it easy to manipulate the objects that form your application.

## Client/Server Architecture

Finally, while Oracle Power Objects gives you the flexibility to develop nearly any kind of application, it is specifically designed for building database applications in a client/server environment. You can select the database engine of your choice for its security, performance,

---

scalability, and other important features, while still using the same front end on the client system. Additionally, you can select the best way to enforce a particular business rule, from either the client or the server.



## Table of Contents

### Preface

## 1 Application Development with Oracle Power Objects

The Components of a Database Application . . . . .	1.2
Object-Oriented Development . . . . .	1.2
The Back End: Sessions and Databases . . . . .	1.4
The Front End: Recordsets and Bound Containers . . . . .	1.6
The Big Picture: The Front and Back Ends . . . . .	1.7
Transaction Processing . . . . .	1.7
The Oracle Power Objects Approach to Development . . . . .	1.8
Where Should I Start? . . . . .	1.8
Starting with the Front End . . . . .	1.9
Starting with the Back End . . . . .	1.10
How Should I Design My Classes and Libraries? . . . . .	1.12

## 2 The Oracle Power Objects Environment

Overview . . . . .	2.2
The Oracle Power Objects Desktop . . . . .	2.2
Launching Oracle Power Objects . . . . .	2.4
The Main Window . . . . .	2.5
Deleting a File Object . . . . .	2.8
Application Windows . . . . .	2.8
Designer Windows . . . . .	2.13
Running Forms and Reports . . . . .	2.16
The Database Session Window . . . . .	2.18
Database Object Windows . . . . .	2.20
The Table Editor window . . . . .	2.20
The Table Browser Window . . . . .	2.23
The View Editor window . . . . .	2.24
The View Browser window . . . . .	2.25
The Library Window . . . . .	2.26
References to Library Bitmaps . . . . .	2.26
The Property Sheet . . . . .	2.27
Sections of the Property Sheet . . . . .	2.28
Properties . . . . .	2.30
Methods . . . . .	2.33
The User Properties Window . . . . .	2.34
The Object Palette . . . . .	2.36
Moving and Resizing Objects . . . . .	2.37

### 3 Objects

Overview . . . . .	3.2
Types of Objects. . . . .	3.2
File Objects. . . . .	3.3
Database Objects. . . . .	3.3
Application Objects. . . . .	3.4
Designer Objects . . . . .	3.4
In-Memory Objects . . . . .	3.7
Containers . . . . .	3.8
Applications. . . . .	3.9
Libraries . . . . .	3.9
Sessions . . . . .	3.10
Forms, Reports, and Classes . . . . .	3.10
Designer Objects . . . . .	3.11
Object Containment Hierarchy . . . . .	3.12
Object Characteristics (Properties and Methods) . . . . .	3.13
Properties. . . . .	3.14
Methods . . . . .	3.17
Object Names. . . . .	3.20
Naming Rules . . . . .	3.20
Renaming Objects . . . . .	3.22
Hierarchical Names . . . . .	3.23
Object References . . . . .	3.25
Object Datatype . . . . .	3.25
Relative References. . . . .	3.25
Restrictions on Object References. . . . .	3.28
Object Classes and Inheritance . . . . .	3.29

### 4 Oracle Basic

Overview . . . . .	4.2
Oracle Basic Language Components. . . . .	4.3
Values and Datatypes . . . . .	4.3
Literals . . . . .	4.4
Variables. . . . .	4.6
Symbolic Constants . . . . .	4.10
Operators . . . . .	4.12
Built-in Functions. . . . .	4.17
Commands. . . . .	4.21
Expressions . . . . .	4.22
Object Properties. . . . .	4.23
Object Methods . . . . .	4.23

## 5 Methods and Method Code

Overview	5.2
Triggering Methods	5.2
Creating a Method	5.4
Writing Method Code	5.5
Methods, Default Processing, and Method Code.	5.6
Overloading Method Declarations	5.7
Suggestions and Cautions.	5.7
Debugging Method Code	5.9
The Run-Time Debugger	5.9
Setting a Breakpoint	5.12
Removing a Breakpoint	5.13
Setting a Watchpoint	5.13
Moving Execution to Any Point Within the Method	5.14

## 6 Databases

Overview	6.2
Database Sessions	6.3
Creating a Database Session	6.5
Connect Strings.	6.6
Activating and Deactivating a Database Session.	6.10
The Default Session	6.12
Using Sessions in an Application	6.13
Example: Creating a Logon Dialog Box	6.15
Properties and Methods of Database Session Objects	6.19
Blaze Databases	6.20
When Should I Use a Blaze Database?	6.20
When Should I Use an External Database?	6.21
Oracle7 Servers	6.21
Oracle7 Documentation	6.22
SQL Server Databases	6.22

## 7 Blaze Databases

Overview	7.2
Creating a Blaze Database	7.2
Schemas in a Blaze Database	7.3
Data Dictionary	7.4
SQL Language.	7.5
Blaze Database Files	7.5
Structure of a Database File.	7.6
Sessions.	7.7
Read-Write Sessions.	7.7
Read-Only Sessions	7.8
Specifications	7.8

## 8 Database Objects

Overview .....	8.2
Tables .....	8.6
Creating a Table .....	8.8
Editing Table Definitions .....	8.12
Editing the Contents of a Table .....	8.14
Using Tables .....	8.16
Views .....	8.16
Creating a View .....	8.18
Editing View Definitions .....	8.21
Editing the Contents of a View .....	8.23
Using Views .....	8.23
Indexes .....	8.24
Creating an Index .....	8.24
Using Indexes .....	8.25
Sequences .....	8.26
Creating a Sequence .....	8.26
Using Sequences in Applications .....	8.28
Using Sequences in SQL Statements .....	8.28
Synonyms .....	8.29
Creating a Synonym .....	8.29
Using Synonyms .....	8.30
Copying a Database Object .....	8.30
Deleting a Database Object .....	8.31

## 9 Structured Query Language (SQL)

Overview .....	9.2
SQL Language Components .....	9.2
Values and Datatypes .....	9.3
Objects .....	9.7
Literals .....	9.8
Operators .....	9.10
Functions .....	9.11
Expressions .....	9.11
Conditions .....	9.12
Commands .....	9.13
Procedural Extensions .....	9.14
Executing SQL Statements .....	9.15
The EXEC SQL Command .....	9.15
The SQLLOOKUP Function .....	9.21

## 10 Applications and Application Objects

Overview	10.2
Applications and File Objects	10.2
Creating a New Application	10.3
Application Object Types	10.4
Containers	10.6
Bindable Containers	10.6
Types of Containers	10.7
Forms	10.7
Reports	10.8
Embedded Forms	10.8
Repeater Displays	10.9
Other Containers	10.11
Controls and Static Objects	10.11
Bindable Controls	10.12
Control Values	10.12
Derived Values	10.13
Display Values and Internal Values	10.14
List Controls	10.15
Types of Controls	10.16
Chart Controls	10.16
Check Boxes	10.19
User-Defined Classes	10.20
Combo Boxes	10.22
Current Row Pointers	10.23
List Boxes	10.24
OLE Objects	10.24
Picture Controls	10.25
Popup Lists	10.25
Pushbuttons	10.26
Radio Buttons	10.26
Radio Button Frames	10.28
Scrollbars	10.28
Text Fields	10.29
Types of Static Objects	10.30
Lines	10.30
Ovals	10.30
Rectangles	10.31
Static Text	10.31
Interacting with Application Objects	10.31
Format Masks	10.31
Tab Order	10.38
Validation	10.39
Enabling and Disabling Controls	10.39

## 11 Forms

Overview	11.2
What Is a Form?	11.2
Developing Forms	11.3
Creating a New Form	11.3
Deleting a Form	11.5
Copying a Form	11.6
Cutting and Pasting a Form	11.6
Adding Objects to a Form	11.6
Testing a Form	11.8
Forms in Design Mode	11.8
Forms in Form Run-Time Mode	11.9
Forms in Application Run-Time Mode	11.10
Forms and Modality	11.11
Dialog Boxes	11.12
Forms and Window Styles	11.13
Controlling the Behavior of Forms	11.13
Printing a Form	11.14
Query by Form	11.14
Using Query By Form	11.15
Entering Criteria	11.16
Some Considerations When Entering Conditions	11.17
Clearing Query Conditions	11.17
Using QBF with Master-Detail Relationships	11.17
QBF Syntax	11.18
Queries, Conditions, and Forms: A Summary	11.19
The DefaultCondition property	11.19
The QueryWhere() method	11.19
Query by Form	11.20

## 12 Reports

Overview	12.2
The Areas of a Report	12.2
Creating a Report	12.3
Designing Areas of a Report	12.4
Resizing Areas of a Report	12.5
Adding Objects to a Report	12.6
Reports and Recordsets	12.7
Defining Filters for the Report	12.7
Populating Controls on a Report	12.7
Binding Controls to Columns	12.8
Using Derived Values	12.9
Using SQLLOOKUP	12.10
Working with Report Groups	12.10
Testing the Report	12.12

Printing a Report . . . . .	12.13
Standard Methods for Printing a Report . . . . .	12.13
Previewing a Report . . . . .	12.13
Starting a New Page . . . . .	12.14
Printing Headers and Footers . . . . .	12.14
Representing Master-Detail Relationships in a Report . . . . .	12.15
Using SQLLOOKUP . . . . .	12.15
Using a Bound Container . . . . .	12.16
Adding a Chart to a Report . . . . .	12.17
Charts for Report Groups . . . . .	12.18
Charts for the Entire Report . . . . .	12.18
Charts for Individual Records . . . . .	12.19
Other Report Considerations . . . . .	12.19
Page Width . . . . .	12.19
Fonts and Reports . . . . .	12.19
Graphics in Reports . . . . .	12.20

### 13 Classes

Overview . . . . .	13.2
Standard and User-Defined Classes . . . . .	13.2
The Object Inheritance Hierarchy . . . . .	13.3
Classes as Containers . . . . .	13.8
Object References to Master Class Definitions . . . . .	13.9
Developing Classes . . . . .	13.9
Creating User-Defined Classes . . . . .	13.9
Adding Objects to a Class . . . . .	13.10
Adding an Instance of a Class . . . . .	13.10
A Sample User-Defined Class . . . . .	13.10
Subclasses . . . . .	13.12

### 14 Menus, Toolbars, and Status Lines

Overview . . . . .	14.2
Menus . . . . .	14.3
Creating a Menu Bar . . . . .	14.5
Initializing a Menu Bar . . . . .	14.5
Creating Custom Menus . . . . .	14.7
Adding Menus to a Menu Bar . . . . .	14.12
Associating Menu Bars with Windows . . . . .	14.13
Handling Menu Selections . . . . .	14.14
Example: Creating a Menu Bar . . . . .	14.18
Toolbars . . . . .	14.20
Overview of Creating Toolbars . . . . .	14.21
Creating a Toolbar . . . . .	14.22
Initializing a Toolbar . . . . .	14.22

Adding Buttons to a Toolbar . . . . .	14.23
Associating Toolbars with Windows . . . . .	14.27
Handling Toolbar Clicks . . . . .	14.28
Example: Creating a Toolbar . . . . .	14.31
Status Lines . . . . .	14.34
Overview of Creating Status Lines. . . . .	14.34
Creating a Status Line. . . . .	14.35
Initializing a Status Line . . . . .	14.35
Adding Panels to a Status Line . . . . .	14.36
Associating Status Lines with Windows . . . . .	14.38
Updating Status Panels. . . . .	14.39
Example: Creating a Status Line . . . . .	14.42
Properties and Methods . . . . .	14.44
Menu-Related Properties and Methods. . . . .	14.44
Toolbar-Related Properties and Methods . . . . .	14.46
Status Line-Related Properties and Methods . . . . .	14.48

## 15 Oracle Power Objects Extensions

Overview . . . . .	15.2
OLE Data Objects and Controls . . . . .	15.2
OLE in Oracle Power Objects . . . . .	15.3
An Overview of OLE . . . . .	15.3
Linking and Embedding . . . . .	15.4
Classes of OLE Objects . . . . .	15.5
OLE Controls . . . . .	15.5
Binding OLE Controls . . . . .	15.8
OLE Data Objects and Files. . . . .	15.8
OLE Data Objects and the Clipboard . . . . .	15.9
Restrictions on OLE-Enabled Applications . . . . .	15.9
Dynamic Link Libraries . . . . .	15.10
Declaring DLL Procedures . . . . .	15.10
Sample Declarations. . . . .	15.11
Creating Flexibility in DLL Procedures. . . . .	15.12
Calling DLL Procedures. . . . .	15.12
Passing Arguments to a Procedure . . . . .	15.13
The Windows API . . . . .	15.13
Other Considerations . . . . .	15.14
OCX Controls . . . . .	15.14
Creating OCX Controls . . . . .	15.15
Importing an OCX into Oracle Power Objects . . . . .	15.16
OCX Properties and Methods. . . . .	15.17
Restrictions on OCX-Enabled Applications. . . . .	15.18



## 16 Compiling the Executable Application

Overview .....	16.2
Compiling for More Efficient Execution .....	16.2
Compiling for Standalone Execution .....	16.2
Generating Application Files and Executables .....	16.3
Creating Run-Time Executable Files .....	16.4
Creating Standalone Executable Files .....	16.4

## 17 Binding a Container to a Record Source

Overview .....	17.2
Relationship Between Containers and Record Sources .....	17.3
Relationship Between Controls and Record Sources .....	17.3
Other Ways to Connect Objects .....	17.4
Binding Objects Graphically .....	17.4
Binding Objects Manually by Setting Properties .....	17.8
Recordset Objects and Bound Containers .....	17.9
Structure of a Recordset .....	17.10
Shared Recordsets .....	17.19
Requerying a Recordset .....	17.21
Making Changes to Data in a Recordset .....	17.23
Standalone Recordsets .....	17.27
Properties and Methods of Recordsets and Bindable Objects .....	17.31

## 18 Defining Master-Detail Relationships

Overview .....	18.2
Master-Detail Relationships and Joins .....	18.2
Automated Joins in Oracle Power Objects .....	18.2
Referential Integrity .....	18.3
Defining Master and Detail Containers .....	18.4
Setting the Primary Key .....	18.6
Integrity Checks .....	18.6
LinkMasterUpd Property .....	18.6
LinkMasterDel Property .....	18.7
Options for Displaying Master and Detail Records .....	18.8
Placing the Detail Records on the Form .....	18.9
Placing Detail Records on a Separate Form .....	18.10
Creating a Drilldown Form .....	18.10
Other Options for Displaying Master-Detail Relationships .....	18.11

## **19 Using Constraints to Enforce Business Rules**

Overview of Constraints . . . . .	19.2
Constraints in the Database . . . . .	19.2
Types of Database Constraints . . . . .	19.3
Not Null Constraints . . . . .	19.3
Unique Constraints . . . . .	19.3
Primary Key Constraints . . . . .	19.4
Foreign Key Constraints . . . . .	19.7
Check Constraints . . . . .	19.7
List of Supported Constraints . . . . .	19.8
Other Database Integrity Checks . . . . .	19.8
Defining Database Constraints . . . . .	19.9
Defining Constraints in the Table Editor Window . . . . .	19.9
Defining Constraints Using SQL Commands or System Procedures . . . . .	19.10
Removing Database Constraints . . . . .	19.12
Dropping Constraints in the Table Editor Window . . . . .	19.12
Dropping or Disabling Constraints Using SQL Commands or System Procedures . . . . .	19.13
Constraints in the Application . . . . .	19.13
Control-Level Constraints . . . . .	19.14
Row-Level Constraints . . . . .	19.20
Master-Detail Constraints . . . . .	19.23
Session-Level Constraints . . . . .	19.26
Other Client-Enforced Constraints . . . . .	19.27
Using Database and Application Constraints Together . . . . .	19.28
Server-Enforced Constraints . . . . .	19.28
Client-Enforced Constraints . . . . .	19.29

## **A Suggested Coding Standards**

## **B List of Properties and Methods**

## **C Constants and Reserved Words**

## **Index**

# 1

---

## **Application Development with Oracle Power Objects**

This chapter covers the following topics:

The Components of a Database Application .....	1.2
The Oracle Power Objects Approach to Development .....	1.8

---

## The Components of a Database Application

Application development poses a number of challenges and choices for database application developers of all types. Oracle Power Objects has been designed to address these challenges, whether your application is designed to access data *locally*, from a database on the same PC or Mac where the executable file is situated, or *remotely*, from a database platform like Oracle7 or SQL Server. Oracle Power Objects can simplify client/server development in the following ways:

- **An object-oriented development model** lets you apply the same methodology to working with all objects, from tables and views on the “back end”, to forms, text fields, and bitmaps on the “front end”.
- **Drag-and-drop features** help you bind forms and reports to tables and views.
- **Automated transaction processing** removes the need to write large amounts of code, or any code at all in many cases, to manage transactions.

Before you use Oracle Power Objects, you should understand the nature of the object-oriented model behind it, as well as the features that simplify database connectivity and transaction management. This chapter provides an overview of how an Oracle Power Objects application is constructed, and how to begin taking advantage of its object-oriented and automated features.

### Object-Oriented Development

In the strictest object-oriented model, every component of an application is presented to the developer as an object. In these terms, an object has two different sets of components:

- **Properties** determine the appearance, behavior, and other features of the object. The background color of a form, the string used to open a database session, and the main table for a report are all properties of those objects.

During run time, the application can evaluate the value assigned to a property, and when required, assign a new value to it. When you press a pushbutton on a dialog, for example, the application checks to see if this button's **Enabled** property is set to the Boolean value of True. If so, then you can press the button; otherwise, nothing happens. At run time, the application can disable the pushbutton by assigning the Boolean value of False to the pushbutton's **Enabled** property.

- **Methods** perform some processing related to the object. For example, if you want to open a form when you press a pushbutton, you would add the necessary code to the **Click()** method of the pushbutton. Method code is written in the Oracle Basic programming language, and is added to the method itself through a window in the Property sheet.

Most standard methods (that is, those that are part of the object's default definition) have default processing associated with them. For example, the **OpenWindow()** method of a form has default processing that loads the form into memory and displays it. When a standard

For more information on writing method code, see Chapter 5, “Methods and Method Code”.

method has default processing, you can add code that either replaces the default processing or adds to it. As part of its default processing, one method often calls another. This means that if you interrupt the default processing for one method, the next in the chain will not be called.

You can also create your own user-defined methods, which you then add to an object.

During development, therefore, you work with the properties and methods of objects in two ways:

- Evaluating the current value assigned to a property, or assigning a new one to it.
- Calling a method, or defining what the method does through method code.

### Development in an Object-Oriented Environment

In addition to changing the characteristics of individual objects, you also define the relationships between objects. For example, when you bind a form to a table, you are defining the relationship between an application object and a database object. When you reorder the data retrieved from the table and displayed in the form, you are further defining the relationship between these two objects. The same holds true of a pushbutton that, when pressed, opens the form: the method code you add to the `Click()` method on the pushbutton defines a relationship between the pushbutton and the form.

Each object has a large range of standard properties and methods; thus, the characteristic you want to take, or the action you want to perform, is often already contained in a property or method. As needed, however, you can also add new, user-defined properties and methods. You add a new property to an object when you decide that it needs an added characteristic (for example, the security level needed to open a form). You add a new method when you want to define a new processing task not covered by the standard methods (for example, performing a tax calculation).

In essence, object-oriented development codifies how you already think about applications. Generally, before doing the actual development work, you envision what objects you need, what characteristics they have, and how an action taken on one object (for example, a pushbutton) affects another (for example, a form). You then have a common methodology for working with nearly every object in Oracle Power Objects. Regardless of whether the object is a database session or a radio button, you evaluate or change one of its properties, or call one of the object's methods to perform a task.

### Object-Oriented Development: Other Considerations

Object-oriented development has three other important features:

- **Modularity:** In an object-oriented development environment, objects should include a complete definition of their characteristics; nothing about their methods and properties should be defined outside the object itself. This feature makes it possible to copy an object into other forms, reports, user-defined classes, sessions, or applications without losing any aspect of its behavior, appearance, or other important characteristics.

- 
- **Hierarchical containment structure:** In an object-oriented application, some classes of objects can contain other objects. For example, a form can contain controls, static objects, and even other containers (such as a repeater display). Object-oriented development tools let you define how some objects contain other objects, and let you resolve the object containment hierarchy from top (for example, an application) to bottom (for example, a pushbutton inside a form).
  - **Reusability:** As much as possible, you should design objects to be reusable. In Oracle Power Objects, user-defined classes and bitmap objects are the most explicitly reusable kind of application objects, though you can easily copy entire forms and reports between applications. In the case of user-defined classes, you define a general type of control, and then *instantiate* it throughout applications. The instances of the class can be exact copies, or they can include modifications as needed. The important point, however, is that you can create the user-defined class, and then reuse it in any number of places.

## The Back End: Sessions and Databases

For more information on sessions and schemas, see Chapter 6, “Databases”.

For more information on database objects, see Chapter 8, “Database Objects”.

In Oracle Power Objects, the session object exists on the “back end” of the database application. Sessions provide access to database objects (tables, views, sequences, and indexes) stored in a database, as well as the data stored in tables and views. You can use multiple sessions in an application; in fact, a single form, report, or user-defined class also can access multiple sessions.

Through the properties and methods of the session object, you control whether the connection to the database is open or closed. Through the database schema defined for the session, you control the range of database objects available through the session. Many database engines use schemas to limit the database objects available to individual users. For example, a system administrator’s schema will have access to all database objects, but average users might be able to edit only the data from the tables and views needed to perform their jobs.

In Oracle Power Objects, the session is represented by a separate window, in which all the database objects available through that session appear. By double-clicking on the session icon, you can activate and deactivate the connection to the database. While the connection is active, the Database Designer window displays all database objects available in that session. In addition, during development, you can add new database objects and edit existing ones while the connection is open.

### Development Questions

When you develop the back end of a database application, you need to answer a number of questions:

- **What information should be stored in the database?**

In other words, what kinds of company data need to be stored and managed in the database, or are already maintained there?

- **How should I organize this information?**

You arrange the data into logical groupings by creating tables. For example, you would want to organize all global customer information in one table, while all orders made by these customers would appear in a different table. The task of breaking down these logical groups is called *normalization*, which is a vital procedure when working with relational databases. In an RDMBS, the tables must be organized sensibly before you can relate the information stored in separate tables. Normalization also prevents the duplication of the same data in separate tables. You must have intelligently designed the tables holding customer information and orders, for example, before you can view orders according to the customers who made them.

Normalization can be a complex process, well beyond the purview of this manual to describe. For more information, consult one of many excellent textbooks on normalization and database design.

As a database application developer, you also create views, which are virtual or logical tables that combine information from several tables. A view does not actually store data; instead, it provides a way to view related information from multiple tables.

- **Are unique ID numbers needed to sequence records?**

A sequence is a database object that generates a new unique ID number whenever you insert a new record into a table.

- **How can I fine tune the database to provide maximum performance?**

Part of the reason you normalize databases is to increase performance, so that queries do not have to sort through large amounts of irrelevant data. Additionally, by adding indexes to tables, you increase the speed with which the database engine can query data.

- **How can I provide maximum flexibility for querying?**

The most important reason to normalize information in a relational database is to provide a great deal of flexibility when querying data. By breaking data down into the smallest functional chunks possible, you make it easier to build queries connecting related information in countless ways, without a great deal of elaborate SQL programming. Users of the database, then, can easily examine relationships among different pieces of information as needed, without being tied to a rigid organization of the data.

- **How should I enforce constraints on the server?**

To ensure consistent application of a constraint, no matter who accesses the database, you should define many constraints on the server itself.

- **How can I maintain system security?**

As with constraints, consistent application of security rules is one of the important features of a client/server system. In systems supporting schemas, you need to intelligently define the schemas to include only the information needed in each schema, and no more. In many database engines, you can use a variety of server-enforced measures to control access to sensitive data.

---

## The Front End: Recordsets and Bound Containers

On client systems, two different kinds of objects can interact with a database: recordsets and bound containers.

### What Is a Recordset?

A recordset is a set of rows, queried from a table or view, that is held in the client system's memory. Normally, to prevent large network traffic and client memory problems, the application populates the recordset with only a subset of the entire recordset. When the application needs to display further records, or when it has to perform some aggregate operation involving all records in a table or view, it queries more records as needed.

When the user or the application makes changes to records, the changes are first recorded in the recordset, not in the database. At some point, the application sends the changes to the database. During this process, the application may have achieved a lock on the data, preventing other users from editing the same record, or it may recognize changes made to the record since it was originally queried.

Recordsets do not have to include all the columns from their associated table or view. In fact, in most cases, when a recordset is associated with a bound container, the recordset includes only those columns needed to display data within text fields and other controls in the container.

Most recordsets are associated with a bound container, though in Oracle Power Objects you can also create unassociated recordsets.

### What Is a Bound Container?

A bound container is an interface object that displays data queried from a table or view. After querying records, controls in the container can then display values from the columns in the queried recordset. In Oracle Power Objects, a bound container can be a form, a report, an embedded form, a user-defined class, or a repeater display. Not all bindable containers need to be bound to a record source: for example, many dialogs are forms that have no associated table or view. However, all bound containers have associated recordsets, which are populated by database queries while the container is loaded into memory.

Inside a bound container are bound controls, which are connected to columns within the associated recordset. For example, within a form designed to display customer records, several text fields display the customer's name, address, phone number, and other important information. The information in each text field comes from different columns in the recordset.

A container is one type of application object among many. The definitions of application objects are stored in .POA files, and eventually compiled into an executable application. Other application objects include controls (for example, pushbuttons), static objects (for example, lines), OLE objects, imported bitmaps, and user-defined classes.



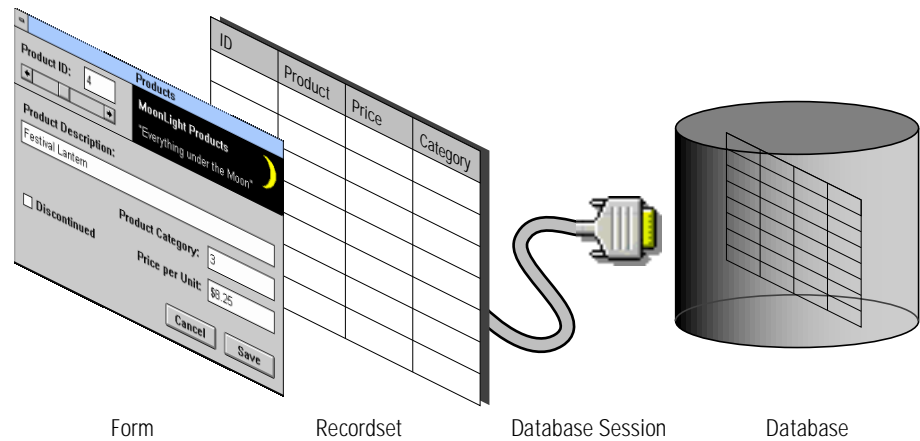
## The Big Picture: The Front and Back Ends

To summarize:

- The database contains objects used to store and organize information.
- A session provides access to the database, as well as to a collection of database objects determined by the user schema on the database.
- The front end of the application queries records as needed to populate recordsets.

Recordsets are normally associated with bound containers, which give the user an interface into the data stored in the database.

The following diagram illustrates the layers of an Oracle Power Objects database application:



## Transaction Processing

Based on this picture of an Oracle Power Objects application, any transaction must pass through several layers, each of which has a different role in controlling transaction processing.

- **Containers.** The container is where the user initiates both queries and changes to the recordset. After adding, deleting, or modifying a record, the user must commit or roll back the transaction (usually by pressing a button designed for this purpose) before closing the container. In addition, client-enforced business rules are defined here, limiting the transactions that the user can initiate. If the system needs to notify the user about the results of an attempted transaction, the information appears here, in the user interface.
- **Recordsets.** Before the user can enter changes to a recordset, the application performs referential and entity integrity checks on the data. In addition, the application regulates the amount of network traffic by selectively querying records to populate the recordset, as needed.

- 
- **Sessions.** After passing through the application and recordset layers, the transaction must then pass through the session. Several methods can be called on a session object to commit or roll back all pending transactions associated with that session.
  - **Databases** Ultimately, a user-initiated transaction reaches the database itself, where the database engine controls whether or not the transaction is completed. The database engine can enforce business rules on the server, check to see if the attempted transaction violates security rules, and perform other important transaction management functions. If the transaction is a query, the database engine can then send the desired information to the client.

In controlling transactions within a client/server application, therefore, Oracle Power Objects gives you the option of enforcing business rules, regulating network traffic, and maintaining the integrity of data. For example, a particular business rule can be enforced as a constraint on the database or as part of the client application.

## The Oracle Power Objects Approach to Development

Good development requires that you make important decisions at the very beginning, so that you do not have to redo a significant amount of work.

### Where Should I Start?

During development, you generally start with the back end or the front end, depending on the kind of application you are developing and the tool you are using. Desktop development tools let you develop the user interface first, building forms, reports, and other portions of the client application. After the user interface is completed, you then build the database objects if needed, and then connect the interface components to their associated tables and views. Generally, if you are working with a tool in which a significant amount of programming is needed to connect the front end components to the back end, you write these procedures last, after you are satisfied that the user interface is stable and your database objects have been defined. Otherwise, you will spend a great deal of time rewriting code, as interface components or database objects are redefined.

In contrast, high-end client/server tools often demand that you design the database objects first and then build the front end. Given the volume of database objects and the difficulty in managing the relationships among them, the back end must take priority in these kinds of applications. In addition, the tools themselves often make it difficult to redefine the front end, especially if you are using some kind of automated system to build forms and reports.

In reality, developers often switch between the back end and front end, instead of exclusively designing one or the other first. (However, normally one section of the database application gets emphasis first, even if it does not receive exclusive attention.) Oracle Power Objects is designed to simplify this kind of incremental development, as long as you have a clear picture of the kinds of objects you want to create and as the relationships among them.

## Starting with the Front End

Generally, you start developing the front end first when . . .

- You need to enforce standards, such as the FASB standards for financial applications, through the client interface.
- You need to focus your efforts on how the application fits or shapes the workflow process.
- You have a short time for prototyping the application, placing a premium on developing the portion users will see.
- You are using database objects that have already been created, and you simply have to build a front end for them. However, before you continue, you need to review whether the database may need some slight modifications. For example, you may decide that the enforcement of a particular business rule should occur on the client, rather than the server, obviating the need for a stored procedure designed for this task.
- The application does not depend a great deal on database access, so you can spend more time on the user interface.

In these cases, Oracle Power Objects lets you design the different forms and reports that comprise the user interface separately, and test them as you go. Therefore, application development that starts with the front end can be highly incremental: you can finish one piece before moving on to another, or you can work on several separate pieces simultaneously. Modifications are relatively easy to make, because database connectivity is not yet fully implemented. After creating the separate forms, reports, classes, and other application objects, you can then decide how the user will navigate among them. You can also add code to enforce business rules (also known as constraints), as well as perform important processing tasks, such as calculations.

As you develop the interface objects and their associated tables or views, you may find yourself making regular adjustments to both, as needed. Tables and views are often designed to mirror the forms and reports in which their records will appear.

When developing the front end first, ask yourself the following questions:

- **What are the major forms that will appear in the user interface?** These will probably be the first objects you design, along with their associated tables or views.
- **What workflow model will be imposed by the application?** In other words, you need to consider how easily the user can enter data, navigate between forms, and perform other operations within the application. Additionally, you need to determine whether working with your application organizes and paces tasks intelligently.
- **What objects need to be defined outside the Oracle Power Objects application and then imported?** For example, if you plan to add bitmap images or other OLE objects, you may need to develop some of these application resources first.
- **Where is the proper place to enforce constraints, and how should they be enforced?** For example, if you want to ensure that a transaction entered into a purchase order application is less than a certain amount, you may want to enforce this constraint on the client through

---

For more information on classes, see Chapter 13, "Classes".

Oracle Basic code, instead of on the server through a trigger. To do so, you would add the necessary Oracle Basic code to intercept the transaction before it is committed if it exceeds a predefined amount.

- **How should you break up the presentation of a single "document"?** In this sense, a document is a distinct entity, such as a purchase order, which is represented in the application. It may seem thorough to place every field that will contain data relevant to the purchase order on the same form, but for readability, you may want to break up the mega-form into several smaller forms.
- **What interface components will be repeated throughout the application?** When you have the same objects appearing repeatedly throughout an application, you should probably design them as user-defined classes, stored in either the application or a library. You can add instances of the user-defined classes to forms and reports, instead of re-creating the same objects over and over again. For example, if you design a custom set of database browsing controls, you should create them as a class, so that instances of the same class can easily inherit changes to the master class.

The major disadvantage of developing the front end first is that database design is usually one of the most important tasks in software development, but the user interface receives the greatest attention. What is well represented on a single form may be difficult to represent in a table or view, or even several related tables.

The major features of Oracle Power Objects that let you start development with the front end include:

- GUI tools for the rapid creation of new front-end objects.
- Reusable application objects, created as user-defined classes or library objects.
- The ability to test a single component of the user interface, such as the main form in the application, before working with other application objects.
- A run-time debugger that can interrogate properties, test the code entered for methods, and perform other important tests.

## Starting with the Back End

In this approach to development, you start with your *data model* (the design of all your tables and views) before you build the interface. In this case, the database objects take precedence for one of several possible reasons:

- **The application will be using a large number of tables and views, making it important to define these first before continuing.** As anyone who has worked with relational databases knows, it is better to have a clear definition of the database objects and their relationships first, to prevent problems caused by tinkering significantly with them later. If one table in a master-detail relationship has the datatype for its key value changed, for example, it can wreck the relationship between the master and detail tables.

- **You choose to enforce many constraints on the server.** In such cases, it makes sense to define the database objects, as well as the triggers and stored procedures that protect them, before you work on the user interface. You can then design the client to work with these server-enforced constraints in several ways, such as capturing error codes generated from the server and presenting them to the user in a meaningful way.
- **The user interface is effectively a window into the database.** In cases where fields on a form are most often simply representations of tables and views on the server, you can spend less time on designing the user interface.
- **Server-enforced security is a high priority.**
- **Performance requires that you begin working on many server components** (for example, indexes, normalized tables) before proceeding to other parts of the application.
- **Several different user interfaces will use the same tables and views.**
- **The application will use complicated master-detail relationships or calculated values.** In such cases, you need to design your tables and views carefully so that you can easily represent one-to-many relationships within the application. Additionally, an intelligent data model will save you time otherwise spent revising equations using data queried from the database.

When designing the back end first, you build your data model to provide an efficient and secure way to access information about “real world” topics. A real-world object can be any entity (for example, an employee, a general ledger transaction, an inventory item, etc.) you wish to describe in one or more tables.

When starting with the back end, ask yourself the following questions:

- **What database objects do I need?** In essence, what is your data model?
- **How should I organize my data?** Again, normalization is important here, as well as performance.
- **What are the primary tables or views?** In almost every data model, some tables are more central than others. You should therefore consider what will happen when many users try to access this table on a client/server network, a likely event in the case of important tables or views used by many people in your organization. Additionally, you should consider how to protect some of the data from destructive changes (for example, modifications to key values in a master-detail relationship), as well as unauthorized access.
- **What is the best way to enforce security on the server?** There are many ways to answer this need, from defining schemas that limit user access to database objects, to writing triggers that prevent changes to the database under certain conditions.
- **Which business rules should be enforced on the server?** While you may not want to overload the server with the job of enforcing every business rule, many constraints are important enough to enforce on the server, to ensure their consistent application.

---

## How Should I Design My Classes and Libraries?

Before you start designing your application, consider whether you will be using some objects or sets of objects repeatedly throughout development. Some examples include:

- A company logo that appears on many forms.
- A set of controls used to navigate among records.
- A set of controls used to filter and sort records.
- A set of radio buttons used to provide the same set of options in several forms.
- A series of text fields displaying customer, vendor, or company information in several forms.

Whether you are starting with the back end or front end first during development, these commonly used application objects should suggest themselves to you as you begin work on the front end.

Creating these objects as user-defined classes, stored in the application or in a library, has the following advantages:

- Once you have designed them, adding them to a form or report is as easy as dragging and dropping.
- In Oracle Power Objects, instances of a user-defined class or a library object inherit the properties and methods of the master class definition. So, too, do the instances inherit changes to these master class definitions. Therefore, modifying frequently used application objects is much faster when you create them as instances, since you need only modify the user-defined class or library object to make the same changes to all instances.

The rationale for creating a reusable application object as a user-defined class is simple: Do I want to use this object in this application only, or do I want to use it in multiple applications?

- If you want to use the object in several applications, create it as a library object. When you need to modify the object, it is separated into its own library, much like a dynamic link library in the PC environment.
- If you plan to use the object in one application only, create it as a user-defined class. By keeping the user-defined class in the application, you reduce the number of file objects that you have to maintain.

Currently, Oracle Power Objects lets you create user-defined classes and bitmaps as library objects. User-defined classes are the most common library object, because they provide all the functionality of both containers and controls.



---

# The Oracle Power Objects Environment

This chapter covers the following topics:

Overview .....	2.2
The Oracle Power Objects Desktop .....	2.2
The Main Window .....	2.5
Application Windows .....	2.8
Designer Windows .....	2.13
The Database Session Window .....	2.18
Database Object Windows .....	2.20
The Library Window .....	2.26
The Property Sheet .....	2.27
The User Properties Window .....	2.34
The Object Palette .....	2.36

---

## Overview

The Oracle Power Objects user interface (or *environment*) is designed to display the objects used in application development in a simple, logical, and useful way. Through a series of windows, you can perform the following development tasks:

- Add, modify, and delete objects in applications, libraries, and sessions.
- Determine the structure of database objects.
- Enter data into tables and views.
- Edit the methods and properties of application objects.
- Establish connections between application objects and database objects.
- Test application objects to determine their appearance and behavior at run time.

The various windows, toolbars, and other components of the environment are designed to represent the objects in a database application in a way that reinforces the object-oriented approach to development. In addition, the interface clearly separates the objects on the back end of the application (in the database) from those in the front end (in the application).

This chapter describes how the Oracle Power Objects environment is organized, and how to use its components to design database applications.

- **Note:** This chapter frequently refers to objects and procedures with which you may not be familiar, if this is the first time you have read this manual. In these cases, the documentation refers you to a later chapter for further explanation. You can return to this chapter later as a reference to the Oracle Power Objects development environment.

## The Oracle Power Objects Desktop

The *Oracle Power Objects Desktop* encompasses the entire Oracle Power Objects designer interface. Several menus and toolbars appear at the top of the window. These menus and toolbars may change, depending on the active window within the desktop.



*Menus* provide a set of menu commands for performing development tasks, such as opening an existing application file object or running a form. *Toolbars* display a series of buttons that provide shortcuts for many commonly used menu commands. The Status line at the bottom of the desktop displays messages about the currently selected item and the status of Oracle Power Objects.

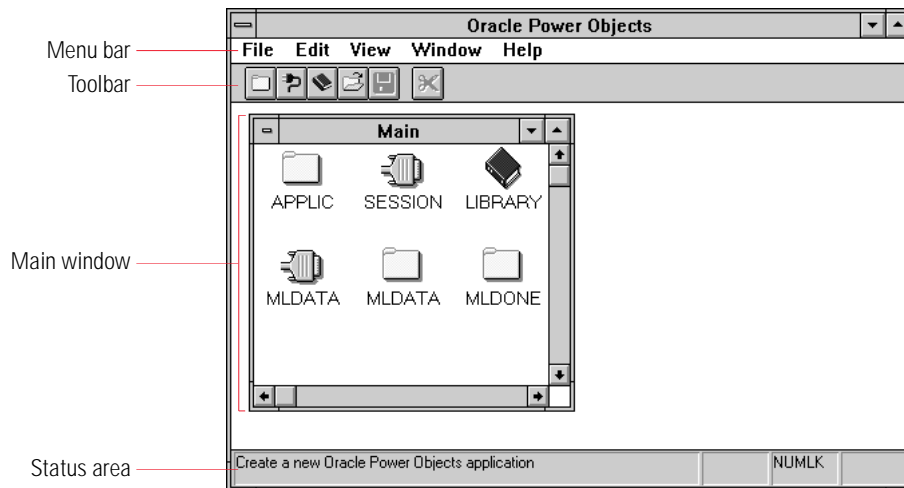
In addition, several types of windows and palettes appear in the desktop:

- *The Main window*, which contains icons for applications, database sessions, and libraries.



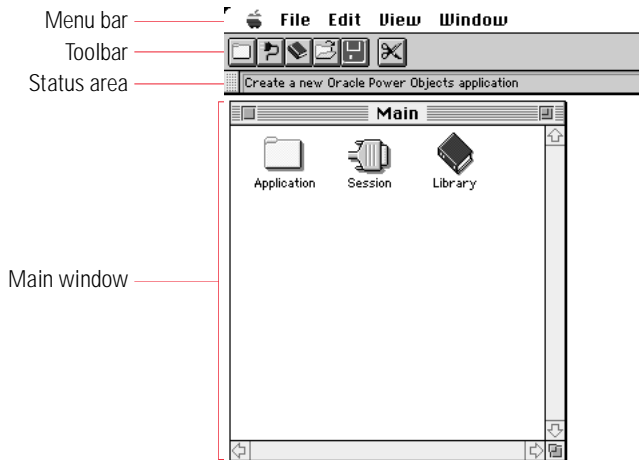
- *Table Editor and Browser windows*, used to define the structures of database objects as well as edit the data stored within them.
- *Application, Session, and Library windows*, which contains icons for several types of objects whose descriptions are stored in the application, session, or library file object.
- *Designer windows*, in which you design application and database objects.
- *The Object palette*, which provides a set of drawing tools used to create and manipulate many types of application objects. For example, by selecting one of the buttons on the Object palette, you can draw a check box on a form.
- *The Property sheet*, through which you can edit the properties and methods associated with an object. The contents of the Property sheet change to display the properties and methods of the currently selected object.
- *The Run-Time Debugger*, used to interrogate values of properties and variables, as well as perform other important tests.

In Windows, the Oracle Power Objects desktop appears as follows:



---

On the Macintosh, the Oracle Power Objects desktop appears as follows:

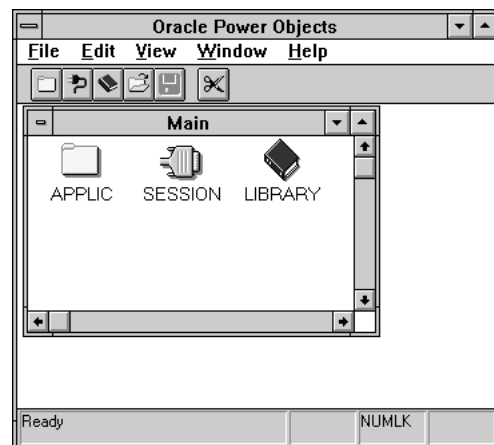


Several windows have their own toolbar and menu bar associated with them. For a complete listing of menus and toolbars, consult the *Getting Started* manual, or the topics covering Desktop menus and toolbars in the online help.

## Launching Oracle Power Objects

To launch the Oracle Power Objects Designer application, locate and double-click its icon in your Macintosh or Windows environment.

When you launch Oracle Power Objects, the Desktop window appears. In addition, the Main window appears in the area immediately below the menu and toolbar.






To quit Oracle Power Objects, choose the **File-Exit** menu command in Microsoft windows, or **File-Quit** on the Macintosh.

## The Main Window

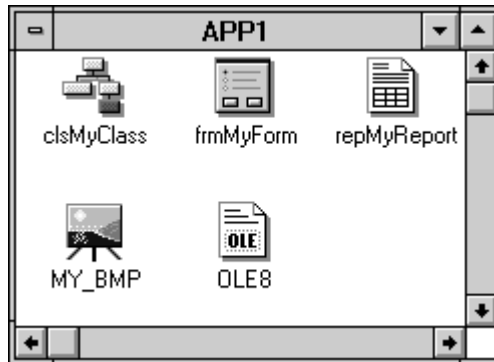
The Main window displays icons representing applications, sessions, or libraries that you have previously opened or created.

Applications, libraries, and sessions are *file objects*, because they have a corresponding file in the operating system. The icons appearing in the Main window for these objects include:

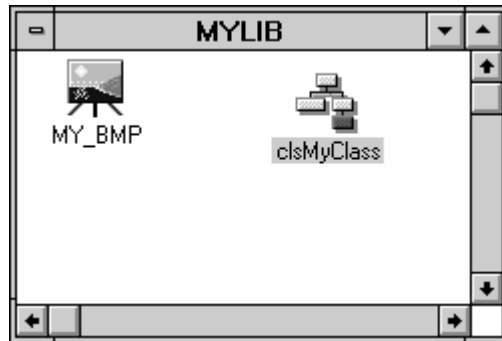
Icon	Object
	Application
	Session
	Library

By double-clicking on these icons, you open the Application, Session, or Library window corresponding to this file object. In these windows, you see other icons representing the contents of the application, session, or library.

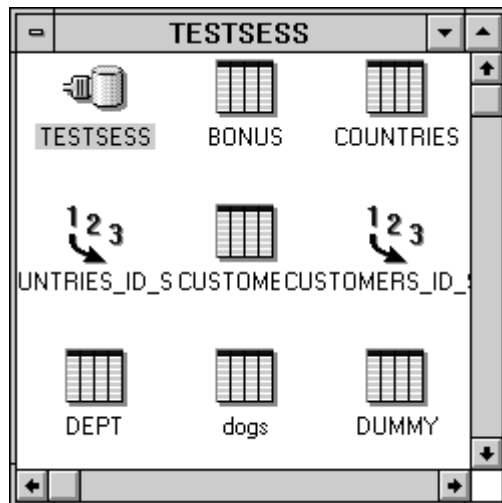
- **Applications** can contain forms, reports, user-defined classes, bitmaps, and OLE objects.



- **Libraries** can contain user-defined classes and bitmaps.



- **Sessions** can contain tables, views, sequences, and indexes.



You add objects to the Main window by creating new file objects or by opening existing file objects. The icons representing these file objects appear in the Main window every time you launch Oracle Power Objects. The file object names and their paths are stored in a file, PWROBJX.INI.

These file objects have the following extensions in Microsoft Windows:

Object	Extension	Example
Applications	.POA	MYAPPPOA
Sessions	.POS	SESS1.POS
Libraries	.POL	MYLIB.POL

For more information on compiling applications, see Chapter 16, "Compiling the Executable Application".

Eventually, the contents of these types of files are compiled into an executable file that defines the run-time application. In the case of sessions, the mapping information describing the database objects available in the session are compiled; the actual objects, as well as their data, exist within the database itself. Oracle Power Objects compiles the application into portable code (called "P-code") that can be moved between operating systems.

## Tasks in the Main Window

You can perform the following tasks in the Main window.

- Create new file objects.
- Add existing file objects to the Main window.
- Remove file objects from the Main window.
- Open the window for an Application, Library, or Session object.

### ☆ To create a new file object:

- 1 With the Main window as the active window, choose the appropriate menu command or button:



For applications, click the New Application button or choose **File-New Application**.

-or-



For sessions, click the New Session button or choose **File-New Session**.

-or-



For libraries, click the New Library button or choose **File-New Library**.

The Create As dialog for the operating system then appears, prompting you to enter a name for the new file object. In Microsoft Windows, Oracle Power Objects automatically appends the appropriate extension to the filename.

- 2 After assigning a filename to the file object, click the **OK** button in this dialog.  
The icon for the new object now appears in the Main window.

### ☆ To add an existing file object to the Main window:



- 1 With the Main window as the active window, click the Open button or choose the **File-Open** menu command.

- 2 Using the appropriate dialog from the operating system, select the file you wish to add to the Main window.

- 3 After you have selected the file, click the **OK** button in the dialog.  
The icon for the object now appears in the Main window.

---

☆ **To remove an object from the Main window:**

1 With the Main window as the active window, select the icon for the file object you wish to remove.



2 Click the Cut button from the toolbar.

The icon for the application, session, or library then disappears from the Main window. The file itself is *not* deleted from the operating system.

☆ **To open a Designer window for an application, library, or session:**

1 Make the Main window the active window.

2 Double-click on the icon representing the application, library, or session.

## Deleting a File Object

To delete a file object, you must remove it from the operating system, using the DEL command in DOS, the File Manager in Windows, or the Trash on the Macintosh.

If you delete a file object that is represented in the Main window of the Oracle Power Objects desktop, Oracle Power Objects will display an error message the next time you launch it. To avoid the error, cut the object from the Main window before deleting the file.

## Application Windows

Oracle Power Objects displays an application window when:



- You open an existing application in the Main window by double clicking on the icon representing the application, clicking the Open button, or choosing the **File-Open...** menu command.



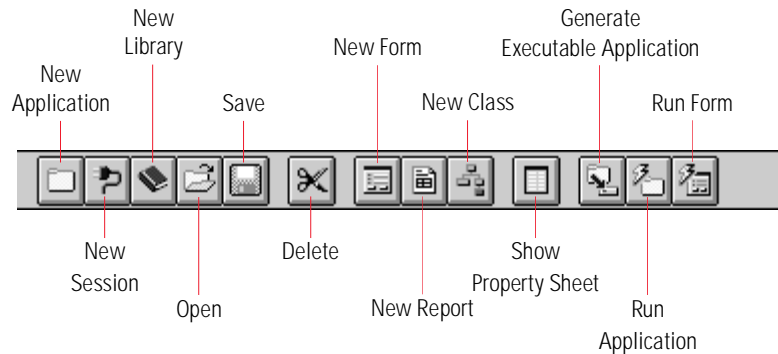
- You create a new application by clicking the New Application button or choosing the **File-New Application** menu command.

Applications contain several varieties of application objects. For more information on each type of object, see the chapters indicated below.

<b>Object Type</b>	<b>Chapter</b>
Objects (general)	Chapter 3, "Objects"
Forms	Chapter 11, "Forms"
Reports	Chapter 12, "Reports"

Object Type	Chapter
User-defined classes	Chapter 13, "Classes"
Bitmaps	Chapter 10, "Applications and Application Objects"
OLE objects	Chapter 15, "Oracle Power Objects Extensions"

When you open an Application window, the Application Window toolbar appears in the Oracle Power Objects desktop, providing shortcuts for several application-related tasks:



## Tasks in the Application Window

You can perform the following tasks when the Application window is the active window:

- Edit the properties and methods of the application.
- Create or delete forms, reports, user-defined classes, and OLE objects.
- Open these objects for further development. When you open a form, report, or user-defined class, it appears in its own Designer window, as described later in this section.
- Copy or move these objects between applications.
- Import and delete bitmaps.
- Export an application to a file.
- Import an application from a file.

### General Tasks

This section summarizes the general operations possible through an Application window.

☆ **To edit the properties and methods of the application:**

- 1 Make the Application window the active window.



- 2 If it is not open already, open the Property sheet for the application by clicking the Show Property Sheet button.
- 3 Enter the changes to the application's properties and methods through the Property sheet.

☆ **To create a new form, report, or user-defined class:**

- 1 Click the button on the Application Designer toolbar for the kind of object you wish to create. The Designer window for that application then appears.

☆ **To create a new OLE object as an application-level object:**

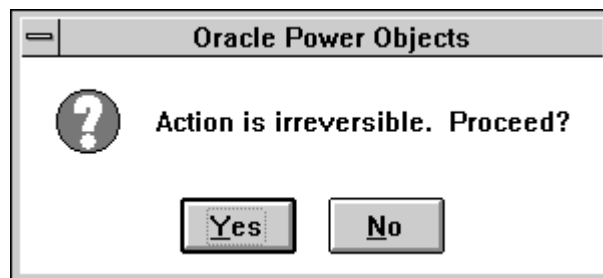
- 1 Choose the **Edit-Import Object** menu command. A special dialog for creating OLE objects appears.
- 2 In this dialog, select the kind of OLE object you wish to create. The server application interface for the OLE object appears.
- 3 Save the OLE object as an embedded object in Oracle Power Objects, or as a linked object in the server application.
- 4 Exit the server application to return to Oracle Power Objects.

☆ **To delete an object from the application:**

- 1 Select the icon in the Application window for the object you wish to delete.



- 2 Click the Cut button. Oracle Power Objects asks whether you wish to go ahead with the deletion.



- 3 Click **Yes**.

For more information on OLE server applications, see Chapter 15, "Oracle Power Objects Extensions".



☆ **To open an object for further development:**

- 1 Double-click on the icon for the object you wish to modify.

The Designer window for that object appears. In the case of bitmaps, you can preview the bitmap, but you cannot edit it.

☆ **To copy an object between applications:**

- 1 Select the object.
- 2 Holding down the Ctrl key in Microsoft Windows or the Option key on the Macintosh, drag the object from one application to another.

Oracle Power Objects copies the icon for that object from the first Application window to the next. Alternatively, you can follow this procedure to copy the object:

- 3 Select the icon for the object you wish to copy.



- 4 Click the Copy button, or choose the **Edit-Copy** menu command.

- 5 Open the Application window for the application into which you want to copy the object.



- 6 Click the Paste button or choose the **Edit-Paste** menu command.

☆ **To move an application object from one Application Window to another:**

- 1 Open both applications.
- 2 Select the object you want to move.
- 3 Drag the object into the open window for the other application.

Alternatively, you can follow this procedure to move an object between applications:

- 4 Select the icon for the object you want to move.



- 5 Click the Cut button or choose the **Edit-Cut** menu command.

- 6 Open the Application window for the application into which you wish to move the object.



- 7 Click the Paste button or choose the **Edit-Paste** menu command.

---

### File-Related Tasks

You must often export applications to flat files or import them from earlier versions of Oracle Power Objects. This flat file format cannot be edited through Oracle Power Objects; instead, it is used to port application, library, and session file objects between versions of Oracle Power Objects. This section summarizes the procedures for exporting and importing applications to and from files.

☆ **To export an application to a file:**

- 1 Select the object you want to write to a file.
- 2 Choose the **File-Write to File** menu command.
- 3 In the dialog that appears, enter the name of the file to which the application will be written, and select the directory in which it will appear.

Frequently, developers give exported files the .F extension (for example, MYAPP.F).

- 4 Click **OK**.

☆ **To read in an application from a file:**

- 1 Choose the **File-Read from File** menu command.
- 2 In the dialog that appears, select the file from which you wish to import the application.
- 3 Click the **OK** button.

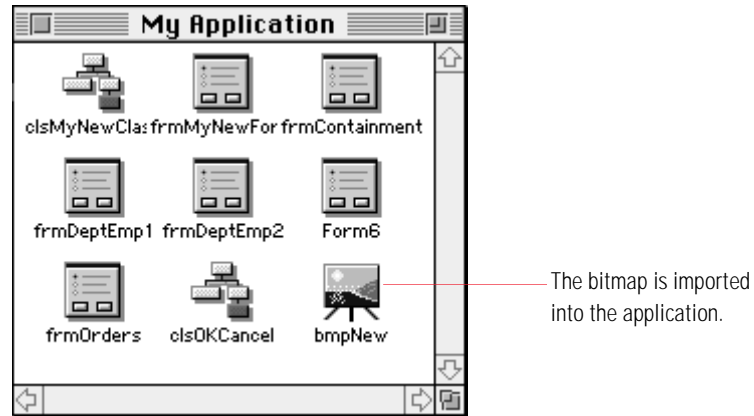
### Bitmap-Related Tasks

To use bitmaps (.BMP files) as application resources in Oracle Power Objects, you first import the bitmap into the application. You can then add it to picture boxes, pushbuttons, and various types of containers. When added, the bitmap appears across the face of the object. This section summarizes the procedures for working with bitmaps as application resources.

☆ **To import a bitmap:**

- 1 Choose the **File-Import BMP** menu command.
- 2 In the dialog that appears, select the directory and filename of the bitmap you want to import.
- 3 Click the **OK** button.

The bitmap then appears in the Application window, as shown below:



☆ **To export a bitmap to a .BMP file:**

- 1 Select the bitmap you want to export to a .BMP file in the operating system.
- 2 Choose the **File-Export BMP** menu command.
- 3 In the dialog that appears, select a filename and directory for the new .BMP file, then click **OK**.

☆ **To view a bitmap object:**

- 1 In the Application window, double-click on the icon for the bitmap.

A copy of the bitmap appears. You can only view the imported bitmap through this window; you cannot edit it.

## Designer Windows

For each container (form, report, or user-defined class) within an application, there is a corresponding Designer window. When this window is opened, you can perform the following tasks:

- Edit the properties and methods of the container.
- Add new objects to the container.
- Bind the container to a record source (a table or view) using the drag-and-drop binding features of Oracle Power Objects.

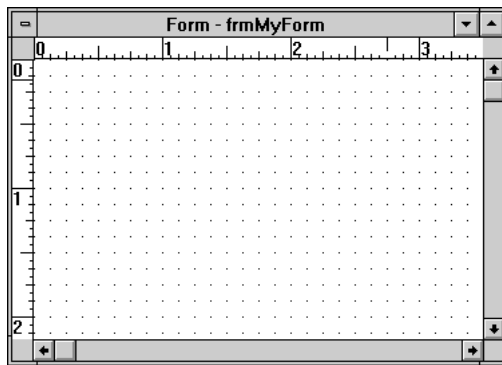
For more information about binding, see Chapter 17, "Binding a Container to a Record Source".

---

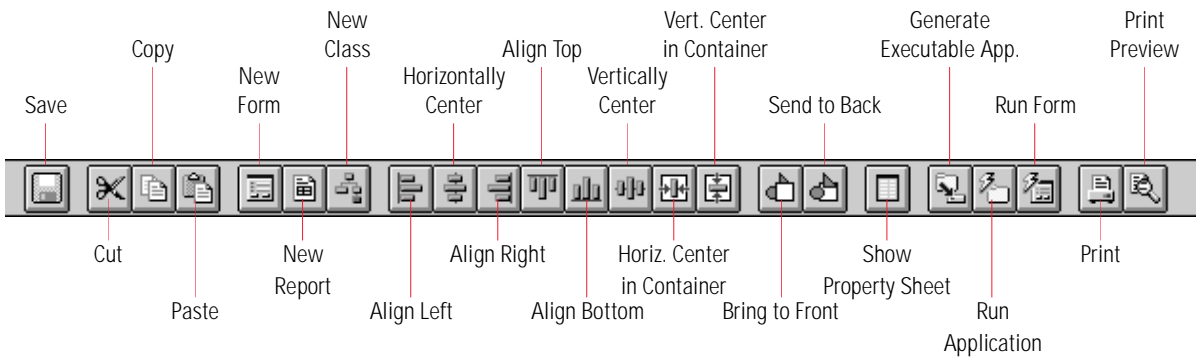
☆ **To open the Designer window for a container:**

- 1 Open the Application window for the application in which the container appears.
- 2 Double-click on the icon for the container.

The Designer window for the form, report, or user-defined class appears, as shown below:

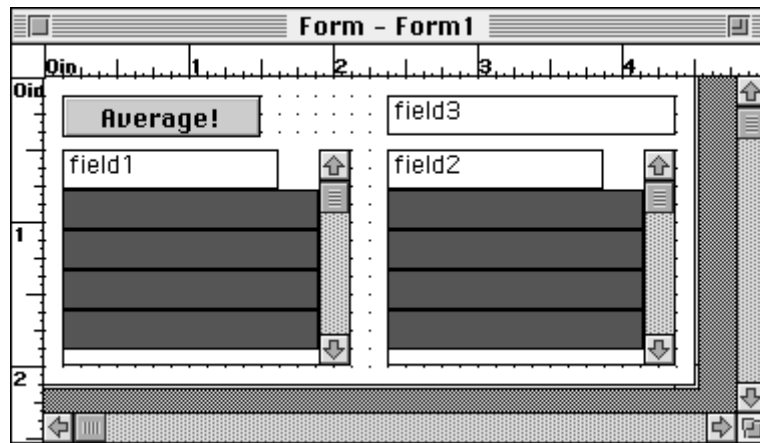


When you open a Designer window for a container, a new toolbar appears, replacing the Application Window toolbar. This toolbar displays buttons that provide shortcuts for many development tasks related to forms, reports, and user-defined classes.



### The Form Designer Window

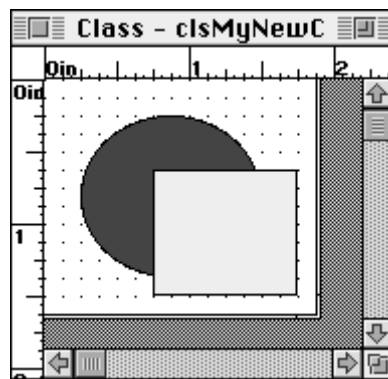
The Form Designer window displays a form, including all objects contained within it. Even if an object is designed to be invisible at run time (because its **Visible** property is set to False), it appears in the Form Designer window at design time. Controls bound to columns do not display data until you run the form.



### The Class Designer Window

For more information about classes and instances, see Chapter 13, "Classes".

The Class Designer window is identical to the Form Designer window, and objects on it appear and behave identically. However, you cannot run the user-defined class on its own. You must first create an instance of the class in a form or report to test it.

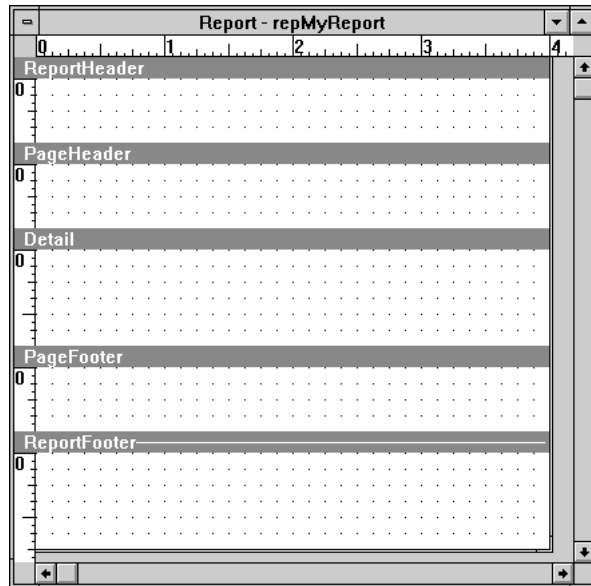


---

## The Report Designer Window

The Report Designer window displays all of the areas of a report, as described in Chapter 12, “Reports”. Each area has a header consisting of the title of the area, followed by a bar running across the width of the form. You can add objects to all of these areas. However, keep in mind that each area will appear in different location, and may be repeated many times when you run the report.

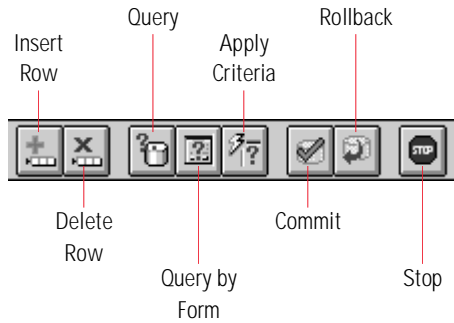
Objects appearing within the Report Designer window obey the same rules as objects in the Form and Class Designer windows.



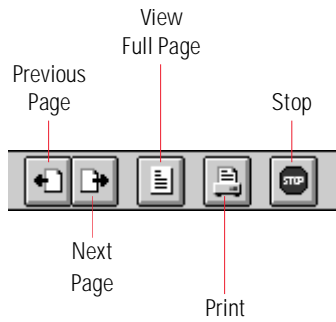
## Running Forms and Reports

When you run a form or report, one of two toolbars appears:

The **Form Run-Time toolbar** appears when you run a form. This toolbar displays several standard controls used to query and edit records.



The **Report Run-Time toolbar** appears when you run a report. This toolbar displays several standard controls used to preview and print a report.



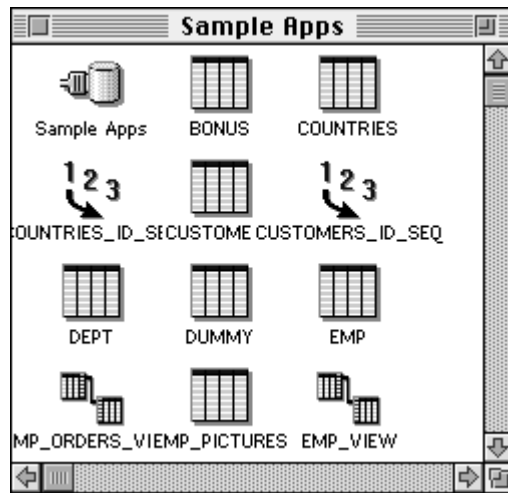
You can replace the Form Run-time and Report Run-time toolbars with your own custom toolbar. For information on creating custom toolbars, see Chapter 14, “Menus, Toolbars, and Status Lines”.

---

## The Database Session Window

Oracle Power Objects opens the Database Session window whenever you:

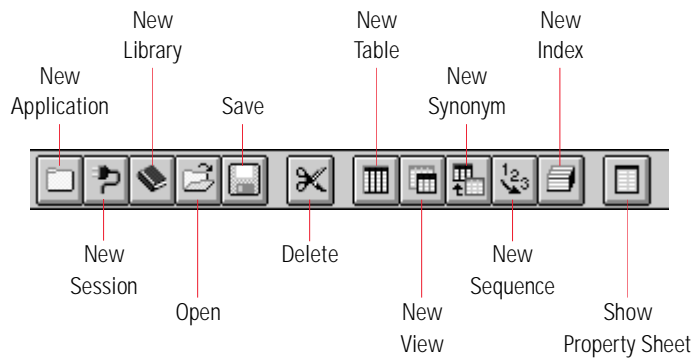
- Open an existing session.
- Create a new session.



For more information on databases, sessions, and schemas, see Chapter 6, "Databases".

The Database Session window displays the Connector control, with which you can open or close a connection to a database. More precisely, the session establishes a connection to a schema in a database, through which you can view the database objects (tables, views, sequences, and indexes) associated with that schema or user account.

While the Database Session window is the active window, the Session Window toolbar appears beneath the menu bar in the Oracle Power Objects desktop. This toolbar displays buttons used as shortcuts for several session-related tasks.





You use the Database Session window to perform the following development tasks:

- Open a connection to a database to view all database objects associated with the session.
- Close a connection to a database.
- Identify the schema through which the session establishes the connection.
- Add new tables, views, indexes, and sequences to a database accessed through the session.
- Delete objects from the database.
- Copy objects between sessions.

### General Tasks

This section summarizes the general tasks performed through the Database Session window. For more information on using Oracle Power Objects to modify database objects or edit data in a table or view, see later sections in this chapter.

#### ☆ To activate a database session at design time:

- 1 In the Main window, open the database session object by double-clicking on its icon.

The Database Session window opens, along with the session's Property sheet.



- 2 Double-click the Connector control to activate the session.

When the session becomes active, icons representing database objects appear in the Database Session window. The Connector control changes to the "active" state.

#### ☆ To close a connection to a database:

- 1 While the connection is open, double-click on the Connector control.

#### ☆ To associate a session with a schema:



- 1 If it is not open already, open the Property sheet for the session by clicking the Show Property Sheet button.
- 2 In the properties associated with establishing connections (for example, **DesignRunConnect**), enter the connect string.

For information on connect strings, see the section "Connect Strings" on page 6.6.



For information on designing tables and views, see Chapter 8, “Database Objects”.

☆ **To create a new table or view:**

- 1 Click the New Table or New View button.
- 2 In the Editor window that appears, define the structure of the table or view.

☆ **To copy a database object between sessions:**

- 1 Open the Database Session window for both sessions.
- 2 Click and drag the icon for the object from one window into the other.
- 3 Release the mouse button.

Oracle Power Objects creates a duplicate of the database object. When you copy a table, you also copy the data in the table to the other session.

## Database Object Windows

The database object windows are described in Chapter 8, “Database Objects”.

You can open other windows from the Database Session window, each representing a different type of database object. These windows are:

- The Table Editor window, used to build tables.
- The View Editor window, used to build views from base tables.
- The Table Browser window, used to edit data in a table.
- The View Browser window, used to edit data in a view.

### The Table Editor window

For information on tables, see the section “Tables” on page 8.6.

The Table Editor window displays the structure of a table, including the characteristics of each column in the table. Through this window, you enter a description of the table structure, which Oracle Power Objects uses to create or alter the object within the database.

To use the Table Editor window, you must open a connection to a database, as described in Chapter 6, “Databases”.

The Table Editor window appears as shown below:

Column Name	Datatype	Size	Prec	Not Null	Unique
EMPNO	NUMBER	4		✓	
ENAME	VARCHAR2	10			
JOB	VARCHAR2	9			
MGR	NUMBER	4			
HIREDATE	DATE				
SAL	NUMBER	7	2		
COMM	NUMBER	7	2		
DEPTNO	NUMBER	2			

The Table Editor window includes the following components:

Component	Description
Column Name field	The name of the column.
Datatype field	The datatype of the column.
Size field	The number of bytes allocated to the column. This information is required for some datatypes (for example, VARCHAR2).
Precision field	The precision of the data in the column, if the datatype is Float.
Not Null field	A flag indicating whether all values in this column must be non-null. If checked, the database enforces the Not Null constraint on this column.
Unique field	A flag indicating whether a value in this column must be unique. If checked, the database engine enforces the Unique constraint on this column.
Row Selector button	A control used to select a row in the table. When selected, the row and its contents are highlighted.
Primary Key tool	A control used to set a column as a primary key in the table.
Primary Key indicator	An icon indicating that a column is part of the primary key for a table. This icon always appears on the Selector button for a column.

---

<b>Component</b>	<b>Description</b>
Expand button	A control to toggle between the expanded and contracted versions of the Table Editor window. In the expanded view, all characteristics of columns are displayed; in the contracted view, only the name and datatype are visible.

When you open the Table Editor window, you can also open the Property sheet for a table. The Property sheet displays only one property, the name of the table. Once you save a table structure to a database, you cannot change the name of the table.

☆ **To open the Table Editor window for an existing table:**



- 1** In the Database Session window, double-click on the Connector control to establish a connection to the database.

The icons for all database objects accessed through the session appear.

- 2** Double-click on the icon for the table.

☆ **To create a new table through the Table Editor window:**

For more information on creating a table, see the section "Creating a Table" on page 8.8.



- 1** Click the New Table button.

The Table Editor window appears.

- 2** Assign a name to the table through its Property sheet.

This name will be assigned to the new table when Oracle Power Objects builds the table in the database.

- 3** Enter the characteristics of each column in the table.



- 4** Build the new table in the database by clicking the Save button.

## The Table Browser Window

You use the Table Browser window to view and edit data in an existing table, or populate a new table with data.

EMP							
EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	17-DEC-80	800		20
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30
7521	WARD	SALESMAN	7698	22-FEB-81	1250	500	30
7566	JONES	MANAGER	7839	02-APR-81	2975		20
7654	MARTIN	SALESMAN	7698	28-SEP-81	1250	1400	30
7698	BLAKE	MANAGER	7839	01-MAY-81	2850		30
7782	CLARK	MANAGER	7839	09-JUN-81	2450		10
7788	SCOTT	ANALYST	7566	09-DEC-82	3000		20
7839	KING	PRESIDENT		17-NOV-81	5000		10
7844	TURNER	SALESMAN	7698	08-SEP-81	1500	0	30
7876	ADAMS	CLERK	7788	12-JAN-83	1100		20
7900	JAMES	CLERK	7698	03-DEC-81	950		30
7902	FORD	ANALYST	7566	03-DEC-81	3000		20
7934	MILLER	CLERK	7782	23-JAN-82	1300		10

The Table Browser window displays the data in a spreadsheet-like format, with a row for each record and a column for each table column. When you edit a record in the Table Browser window, Oracle Power Objects locks that row, and displays the Lock indicator in the margin of the window. When you commit or roll back your changes to the database, the lock is released, and the indicator disappears.

☆ **To open the Table Browser window:**



- 1 With the Table Editor window for a table open, click the Run button.  
The Table Browser window appears.

☆ **To edit data in the table:**

- 1 Through the spreadsheet, add, delete, or modify rows in the table.
  - To add a row, begin typing in the empty row at the bottom of the window.
  - To delete a row, select the row by clicking on its Row Selector button and click the Delete key.
  - To modify a row, enter your changes in the columns you wish to change in that row.



Once you enter any changes, the Commit and Rollback buttons on the Table Browser toolbar become enabled, giving a graphical indication that the contents of the table have been changed.

- 2 Click the Commit or Rollback button, depending on whether you want to commit or undo your changes to the table.

For information on server-enforced constraints, see the section “Constraints in the Database” on page 19.2.

When you commit changes, the database engine enforces any constraints defined for the table.

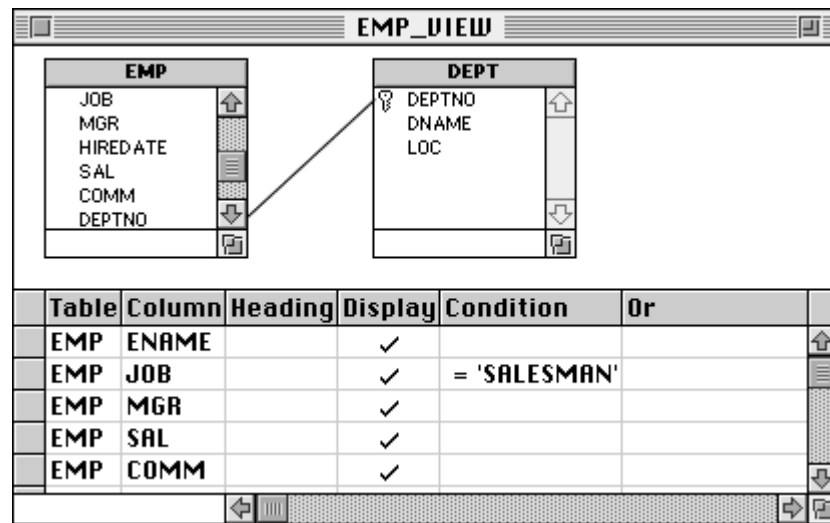
Entering data through the Table Browser window has two disadvantages:

- Oracle Power Objects does not create unique key values for data entered in a column that has a sequence associated with it. You must enter the unique value for that column yourself when you enter data through the window.
- If you have defined referential integrity rules through forms displaying data from the table, the Table Editor window does not enforce these constraints.

### The View Editor window

For information on views, see the section “Views” on page 8.16.

The View Editor window displays the base tables underlying a view, as well as the joins between columns in these base tables.



The components of the View Editor window are:

Component	Description
Table List area	A region containing a window for each base table in the view. Each of these smaller windows contains a scrolling list of the columns in the base table.
Join line	A line indicating a join between columns in separate base tables. Multiple join lines can exist in a view.
Column List area	A spreadsheet-like window containing all the columns in the view.
Table field	A section in the Column List area indicating the table in which a column appears.

<b>Component</b>	<b>Description</b>
Column field	A section in the Column List area indicating the name of the column.
Heading field	A section in the Column List area assigning an optional name for the column in the view.
Display field	A section in the Column List area indicating whether the column can be viewed and edited.
Condition field	A section in the Column List area used to enter conditions applied to data queried for the view.
Or field	A section in the Column List area used to enter a secondary condition applied to data queried from the view. The application applies the OR operator to this condition and the one defined in the Condition field.

☆ **To open the View Editor window for an existing view:**

- 1 In the Database Session window, double-click on the Connector control to connect to the database. Icons representing all objects accessed through the session appear in the window.
- 2 Double-click on the icon for an existing view.

☆ **To create a new view through the View Editor window:**

- 1 In the Database Session window, double-click on the Connector control to connect to the database. Icons representing all objects accessed through the session appear in the window.



- 2 Click the New View button.
- 3 In the View Editor window, enter the definition of the view.

## The View Browser window

The View Browser window looks and behaves exactly like the Table Browser window, with the following exceptions:

- The View Browser displays only those columns from the base tables that are part of the view.
- You cannot edit data in the view unless there is only one base table in the view.

☆ **To open the View Browser window:**



- 1 With the View Editor window as the active window, click the Run button.

---

## The Library Window

Libraries contain bitmaps and user-defined classes designed to be used in more than one application. When you create a new library or when you open an existing library, the Library window appears. This window looks and behaves much like the Application Designer window, and contains two of the same types of application objects.



You use the same techniques for working with bitmaps and user-defined classes appearing in the Library window that you use for working with the same objects in the Application Designer window. For a description of these procedures, see the discussion of the Application Designer window earlier in this chapter.

### References to Library Bitmaps

Through method code, you can refer to a bitmap, including a bitmap stored in a library. The syntax for referring to a library bitmap is *library\_name.bitmap\_name*.

For more information on referring to objects in libraries, see the section "Object Names" on page 3.20.



## The Property Sheet

A Property sheet displays the properties and methods associated with an object. When you select an object, the contents of the Property sheet change to display the properties and methods of the newly selected object.



During development, you use the Property sheet for many common tasks. Through the Property sheet you can perform the following development tasks:

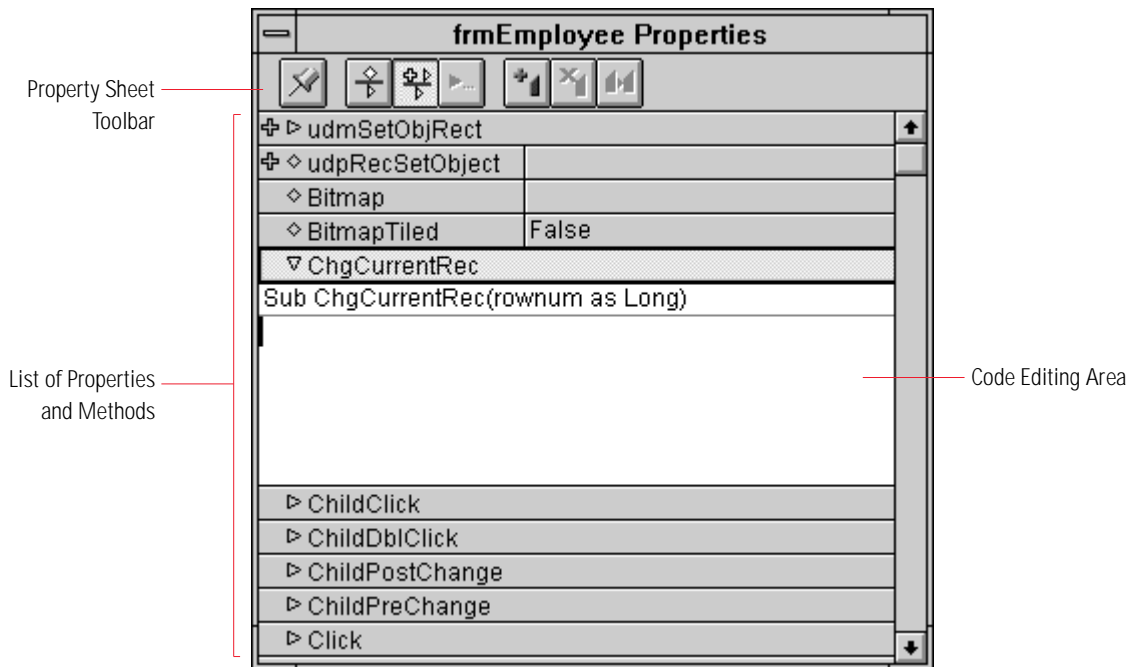
- View the names and settings of a property.
- View the name of a method and (through a secondary window in the Property sheet) the method code associated with it.
- Reorganize the properties and methods associated with an object to reflect one of several criteria.
- Edit a property.
- Edit a method.

This section summarizes the organization of the Property sheet, as well as the procedures for performing these development tasks through it.

## Sections of the Property Sheet

The Property sheet is organized into the following sections:

- A series of buttons used to organize, add, delete, and modify properties and methods.
- Several rows representing the properties and methods of the object.



The buttons appearing at the top of the Property sheet include:

Button	Description
Track Object	Displays the properties and methods of the object selected when the button was pressed, even if you select another object. When not pressed, the Property sheet displays the properties and methods of the currently selected object.
Group by Type	Displays all properties first, then all methods. When not pressed, all properties and methods appear as part of the same alphabetical list.
Group by Creator	Displays user-defined properties and methods first, then all standard properties and methods. When not pressed, displays all properties and methods as part of the same list.

Button	Description
Add Property/Method	Opens the User Properties window, where you create, modify, and delete user-defined properties and methods. For more information on this window, see later in this chapter.
Delete Property/Method	Deletes the currently selected user-defined property and method. Note that you cannot delete a standard property or method.
Reinherit	For objects in an instance of a user-defined class, instructs Oracle Power Objects to reinherit the original setting for the property or method from the master class definition. Classes and inheritance are described in Chapter 13, "Classes".

In the rest of the Property sheet, properties and methods are denoted by the following icons:

Icon	Description
◇	Property
▷	Method
◆	Overridden Property
▶	Overridden Method
⋮	Method Containing Method Code
⊕	User-Defined Property or Method

☆ **To view the Property sheet for an object:**

1 Select the object whose properties and methods you wish to review.



2 Click the Show Property Sheet button.

☆ **To open multiple property sheets:**



1 Click the Track Object button on the Property sheet.

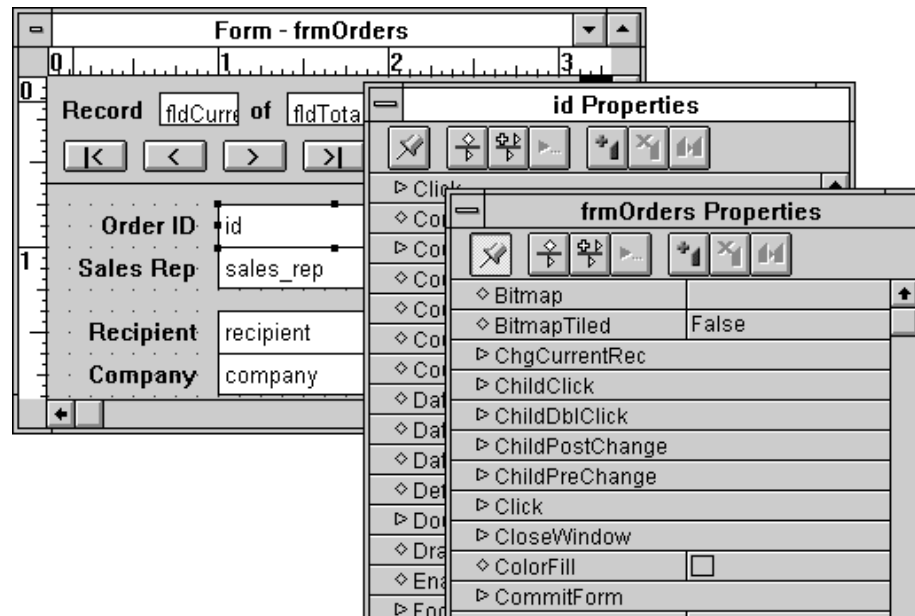
This instance of the Property sheet will continue to display the properties and methods of the selected object.

2 Select another object.



3 Click the Show Property Sheet button.

Another copy of the Property sheet appears, displaying the properties and methods of the second object.



## Properties

You can view both a property and its current setting in the same row of the Property sheet. Properties determine the appearance and behavior of objects, from their background color to the database objects to which they are bound.

Each type of object has a different set of standard properties and methods. When you create a new object, it has only the standard properties and methods for that type of object. You can add user-defined methods and properties to the new object, which then appear on the Property sheet.

You can set a property in several ways, depending on the property itself.

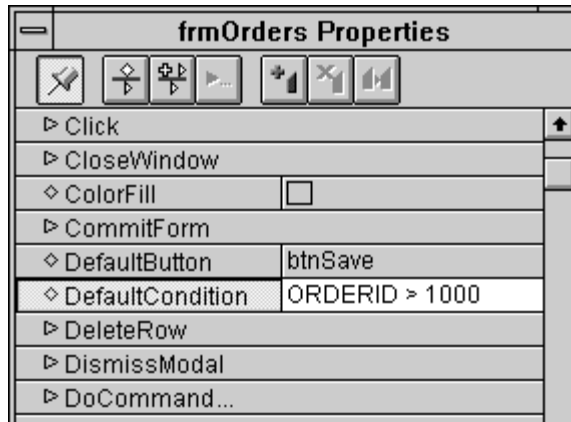
### ☆ To set a property:

- 1 Move the focus in the Property sheet to the property you wish to set.

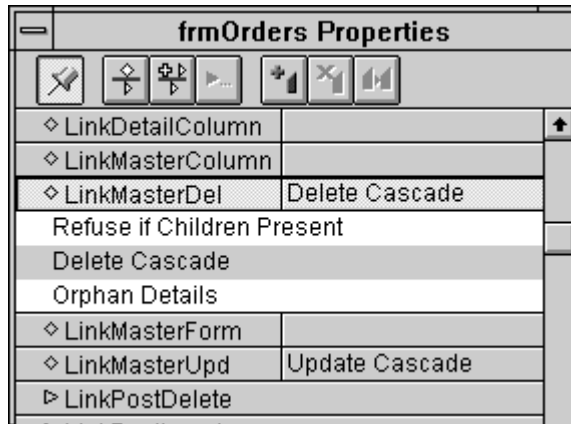
When you select a property, it appears as the only depressed row within the Property sheet.

- 2 Do one of the following, depending on the property:

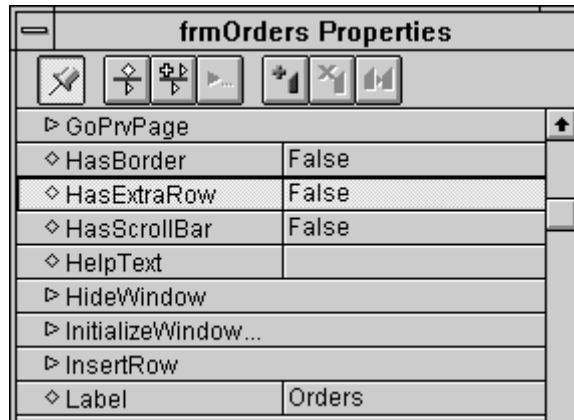
- Enter a new setting for the property in the window appearing to the right of the property's name.



- If a drop-down list of settings appears, select one of the settings from the list.



- If the property has two settings (for example, True and False), click on the window to the right of the property's name to toggle between these two values.

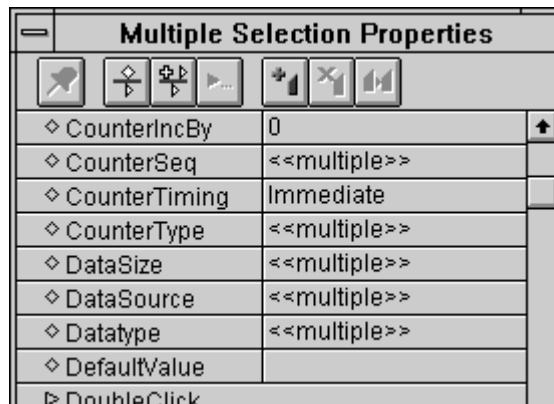


You can also set the properties of several objects simultaneously. The objects must all appear within the same container.

☆ **To set the same property in multiple objects:**

- 1 Select the objects, either by clicking and dragging across the area encompassing the objects, or holding down the Shift key while selecting each object in turn.

The Property sheet displays the properties and methods for all selected objects. If the selected objects have different settings for the same property, <<< MULTIPLE >>> appears as the setting for the property.

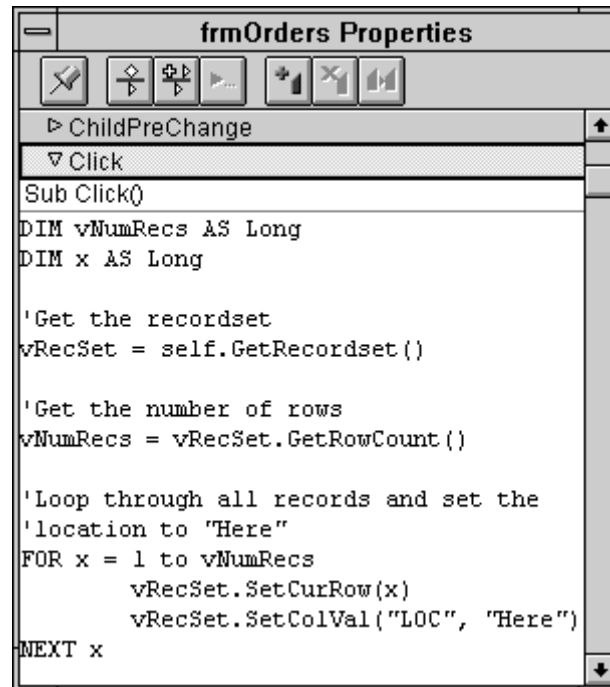


- 2 Enter the new setting for the property.

## Methods

For more information on methods and method code, see Chapter 5, "Methods and Method Code".

Each method has its own code window in the Property sheet, in which you enter method code. Until you open the code window, you can view only the name of the method, not the code associated with it.



When you first open the Property sheet, the code windows for all methods are closed.

### ☆ To open and close the code window:

- 1 Select the method whose code window you wish to open.
- 2 Click on the name of the method.
- 3 To close the code window, click on the name again.

At the top of the code window is the *method heading*, a single line of text that declares the characteristics of the method, including:

- The type of method (Sub for subroutines, Function for functions)
- The name of the method
- The list of arguments passed to the method, including their datatype, enclosed in parentheses
- The return value of the method, if the method is a function

Below the method heading is the *code area*, where you enter your own method code. Note that standard methods often have default processing associated with them, but this default processing does not appear in the code area. When you enter method code in the code area, you override this default processing. The description of each standard method in the online help describes the default processing, if any, performed by the method.

Note that you can cut, copy, and paste code using the keyboard accelerators **Ctrl-X** (cut), **Ctrl-C** (copy), and **Ctrl-V** (paste).

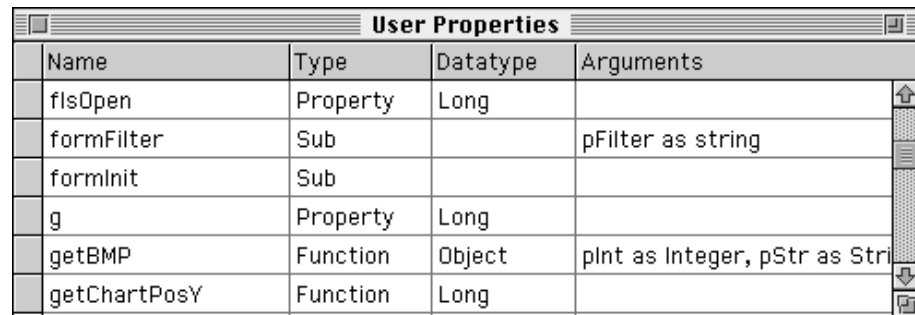
When you run a form or report, or you run the entire application, you can use the Run-Time Debugger to debug the method code you enter through the Property sheet.

For information on the Debugger, see the section "Debugging Method Code" on page 5.9.

## The User Properties Window


The User Properties window lets you define your own properties and methods and then add them to objects in an Oracle Power Objects application. These user-defined methods and properties add to the functionality of an application while staying within the object-oriented paradigm.

For more information on user-defined properties and methods, see the section "Creating a Method" on page 5.4.



Name	Type	Datatype	Arguments
flsOpen	Property	Long	
formFilter	Sub		pFilter as string
formInit	Sub		
g	Property	Long	
getBMP	Function	Object	plnt as Integer, pStr as Stri
getChartPosY	Function	Long	

### ☆ To open the User Properties window:

- 1 Open an object's Property sheet.
- 2  Click the Add Property/Method button from the Property sheet or choose the **View-User Properties** menu command.

The User Properties window consists of the following sections:

Section	Description
Name	Sets the name of the user-defined property or method.
Type	Identifies the item as either a property, a subroutine, or a function.



Section	Description
Datatype	Determines the datatype of a property, or the datatype of the return value of a method. Choices are <i>Boolean</i> , <i>Long</i> , <i>Double</i> , <i>String</i> , <i>Date</i> , or <i>Object</i> .
Arguments	Lists the arguments passed to a method. The name of each argument must be followed by its datatype, using the AS keyword (for example, Param1 As Long).

☆ **To create a user-defined property:**

- 1 Open the User Properties window and select the blank row at the bottom of the window.
- 2 In the Name column, enter a name for the property.
- 3 In the Type column, choose *Property*.
- 4 In the Datatype column, enter the datatype of the property.

☆ **To create a user-defined method:**

- 1 Open the User Properties window and select the blank row at the bottom of the window.
- 2 In the Name column, enter a name for the property.
- 3 In the Type column, choose *Sub* or *Function*.
- 4 In the Datatype column, enter the datatype of the return value, if any.
- 5 In the Arguments column, enter the arguments for the method, if any.

☆ **To add a user-defined method or property to an object:**

- 1 Select the object.
- 2 Open the User Properties window and select the user-defined method or property.  
To select the property or method, click on the Row Selector button at the left side of the row.
- 3 Holding down the mouse button, drag the mouse onto the object or its Property sheet.
- 4 Release the mouse button.  
The method or property now appears on the object's Property sheet, along with a graphic indicator marking it as a user-defined method or property.

☆ **To delete a user-defined method or property from an object:**

- 1 Select the object and open its Property sheet.
- 2 Select the user-defined method or property you wish to delete.



3 Click the Delete Property/Method button on the property sheet.

☆ **To delete a user-defined method or property from the User Properties window:**

1 Delete the property or method from all objects to which it has been added.

➤ **Note:** You cannot delete a user-defined method or property from the User Properties window until you delete all instances of it.

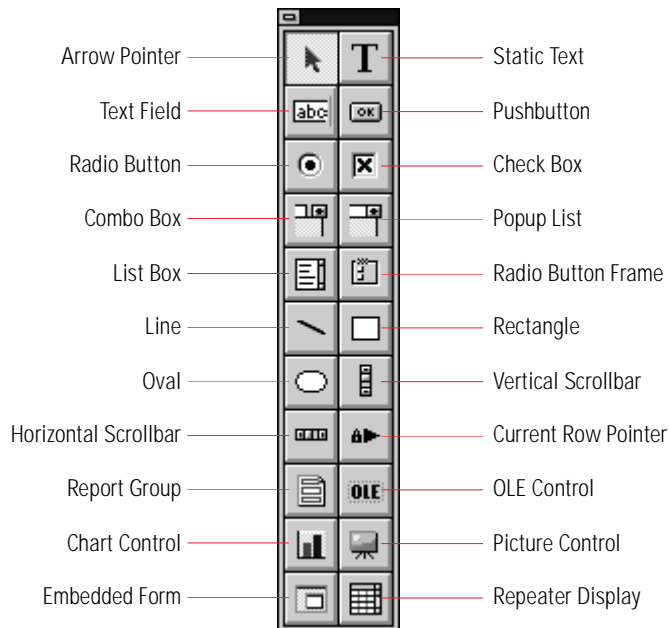
2 Open the User Properties window and select the method or property.

3 Click the Delete key.

## The Object Palette

The Object Palette, a floating toolbar, contains buttons that change the mouse cursor into different tools for designing application objects. Except for the Arrow Pointer, all of the tools on the Object palette change the cursor into a drawing tool used to create a particular kind of application object, such as a check box or a repeater display.

All of the objects created with the Object palette (controls, static objects, and containers) must appear within a form, report, or user-defined class.



The database object windows are described in the section "Database Object Windows" on page 2.20.

The Arrow Pointer tool changes the cursor back into a normal selection tool, used to select, move, and resize objects.

When you import an OCX control, the new custom control appears at the bottom of the Object palette. For more information on OCX controls, see Chapter 15, "Oracle Power Objects Extensions".

When designing database objects, you do not use the Object Palette. Instead, you use a series of windows to design tables, views, sequences, and indexes, and to set the properties of the database itself.

☆ **To create an application object with the Object palette:**

- 1 Open the Designer window for the container (form, report, or user-defined class) to which you wish to add the object.
- 2 Click the button on the Object palette corresponding to the type of object you wish to create.
- 3 In the Designer window for the container, either click on the container to create the new object, using a default size and position set by Oracle Power Objects,

-or-

Click within the container and drag across the area where you want the object to appear.

## Moving and Resizing Objects

Once you have created an object, you can use the selection tool to resize and reposition it. When an object is selected, "handles" appear around the object. By clicking and dragging these handles, you can resize the object. By clicking on an object without selecting the handles, you can drag the object across the container to a new position.

Normally, the edges of an object align with the grid in the background of the container. This behavior is known as *snap to grid*, and occurs when you resize and reposition the object. You can override the snap to grid behavior by holding down the Ctrl key in Windows, or the Option key in Macintosh, while moving or resizing the object. While the Ctrl key is depressed, you can resize or reposition the object in pixel increments, instead of increments of the grid.



# 3

---

## Objects

This chapter covers the following topics:

Overview .....	3.2
Types of Objects .....	3.2
Containers .....	3.8
Object Containment Hierarchy .....	3.12
Object Characteristics (Properties and Methods) .....	3.13
Object Names .....	3.20
Object References .....	3.25
Object Classes and Inheritance .....	3.29

---

## Overview

*Objects* are the building blocks of a database application. Objects represent discrete application components that can be created, reused, customized, and linked.

This chapter introduces the basic types of objects in Oracle Power Objects and concepts for working with them.

## Types of Objects

The objects available in Oracle Power Objects can be divided into several broad categories. These categories are mainly distinguished along functional lines: where the objects are located and how they are used.

<b>Category</b>	<b>Examples</b>	<b>Description</b>
<b>File Objects</b>	Applications Libraries Sessions	Container objects that group the components of a database application into a single operating system file.
<b>Database Objects</b>	Tables Views Indexes Sequences	Structures in a database that store, organize, and provide fast access to information.
<b>Application Objects</b>	Forms Reports Classes Bitmaps	Documents and resources that form the major building blocks of your application's user interface.
<b>Designer Objects</b>		
Static Objects	Rectangles Ovals Lines	Visual objects that add graphical appeal to a layout and delineate sections of a form or report.
Controls	Pushbuttons Text Fields Scrolling Lists	Objects that display values or allow the user to take some kind of programmatic action.
Containers	Embedded Forms Repeater Displays	Contain other objects or controls.
<b>In-Memory Objects</b>	Recordsets Toolbars Menus	Objects that are created at run time and have no direct visual representation during design time.

The following sections list the objects in each category and describe where the objects can be used. It also indicates whether each object is a *container* (can hold other objects) and whether it is *bindable* (can be associated with a database table, view, or column). Object containment is discussed in the section “Object Containment Hierarchy” on page 3.12. Binding objects is discussed in Chapter 17, “Binding a Container to a Record Source”.

The following sections do not explain in detail how to use the objects in each category. However, references are provided to complete information on each category of object.

## File Objects

File objects are represented as icons in the Main window of the Oracle Power Objects desktop. Each file object in the Main window corresponds to a file in your operating system.

Because file objects are actual files, they differ from other types of objects. The names of file objects are restricted to legal names within your operating system, as described in the section “Naming Rules” on page 3.20. You can copy, move, rename, and delete file objects using your operating system.

<b>Object</b>	<b>Locations</b>	<b>Cont</b>	<b>Bind</b>
Application	Main window	Yes	No
Session	Main window	Yes	No
Library	Main window	Yes	No

Applications are discussed in Chapter 10, “Applications and Application Objects”. Sessions are described in Chapter 6, “Databases”.

## Database Objects

Database objects are represented as icons in a Database Session window. Database objects appear to be “contained” within a session, but are actually stored in a database.

Database objects are created and maintained by a *database engine*, a component of the database in which they are stored. Because database engines differ, database objects of the same type can vary if they are stored in different databases. Oracle Power Objects provides a standard interface for working with database objects and the data they contain.

<b>Object</b>	<b>Locations</b>	<b>Cont</b>	<b>Bind</b>
Table	Session window	No	No
View	Session window	No	No

---

<b>Object</b>	<b>Locations</b>	<b>Cont</b>	<b>Bind</b>
Sequence	Session window	No	No
Index	Session window	No	No
Synonym	Session window	No	No

Database objects are discussed in Chapter 8, “Database Objects”.

## Application Objects

Application objects are represented as icons in an Application or Library window. Application objects are stored in the file object that contains them.

You modify the definitions of form, report, and class objects in the Form Designer, Report Designer, and Class Designer windows.

<b>Object</b>	<b>Locations</b>	<b>Cont</b>	<b>Bind</b>	<b>Notes</b>
Form	Application	Yes	Yes	
Report	Application	Yes	Yes	
Class	Application, library, form, report, class	Yes	Yes	
Bitmap	Application, library	No	No	Can be associated with pushbuttons, forms, reports, classes, and embedded forms
OLE Object	Application	No	No	Can be associated with OLE controls

Application objects are discussed in Chapter 10, “Applications and Application Objects”.

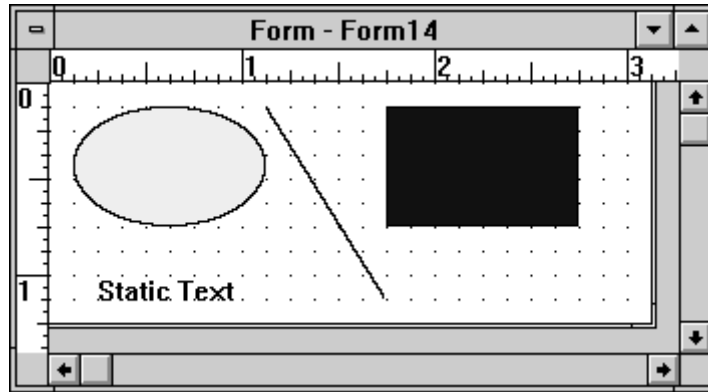
## Designer Objects

Several different types of objects can appear on a form, report, or class. These objects are collectively called *designer objects* because you work with them in the Form Designer, Report Designer, and Class Designer windows.



### Static Objects

Static objects are typically used to delineate areas of a form and to add visual appeal to the form's interface.



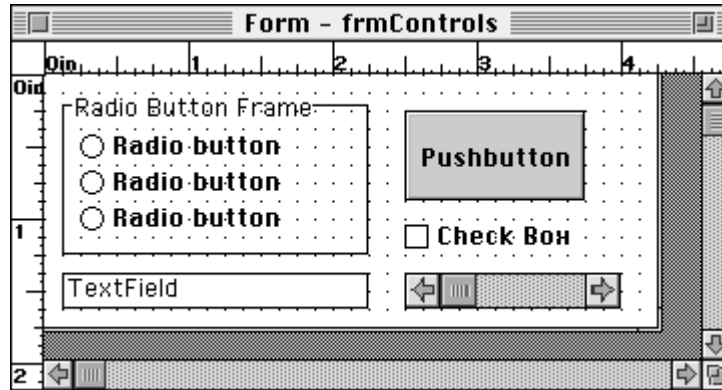
In this diagram, a static text object is used to provide a label on a form. Line objects are used to delineate areas of the form, and a rectangle object is used for graphical appeal.

<b>Object</b>	<b>Locations</b>	<b>Cont</b>	<b>Bind</b>
Static Text	Form, report, class	No	No
Rectangle	Form, report, class	Yes	No
Oval	Form, report, class	Yes	No
Line	Form, report, class	No	No

Static objects are discussed in the section "Types of Static Objects" on page 10.30.

## Controls

Controls are objects that display values or enable the user to take some kind of programmatic action.



In this diagram, a text field control displays a customer's name. A scrolling list control allows the user to select a product name. A pushbutton control allows the user to return to the main menu.

Object	Locations	Cont	Bind	Notes
Text Field	Form, report, class	No	Yes	
Popup List	Form, report, class	No	Yes	
Combo Box	Form, report, class	No	Yes	
List Box	Form, report, class	No	Yes	
Scrollbar	Form, report, class	No	No	
Radio Button	Form, report, class	No	Yes	Usually contained in a radio button frame
Pushbutton	Form, report, class	No	No	
Check Box	Form, report, class	No	Yes	
OCX Control	Form, report, class	No	Yes	
OLE Control	Form, report, class	No	Yes	Can be associated with an OLE Object
Chart Control	Form, report, class	No	Yes	
Picture Control	Form, report, class	No	Yes	Can be associated with a bitmap object

Object	Locations	Cont	Bind	Notes
Current Row Pointer	Form, report, class	No	No	

Controls are discussed in the section “Controls and Static Objects” on page 10.11.

### Containers

Forms, reports, and classes can contain objects that can, in turn, act as containers for other objects.

Object	Locations	Cont	Bind
Embedded Form	Form, report, class	Yes	Yes
Repeater Display	Form, report, class	Yes	Yes
Radio Button Frame	Form, report, class	Yes	Yes
Repeater Panel	Repeater display	Yes	No
Report Group	Report	Yes	Yes

Containers are discussed in the section “Types of Containers” on page 10.7.

### In-Memory Objects

In-memory objects are created only during run time. These objects do not appear directly on the screen; instead, you manipulate them indirectly by associating them with other objects or by executing properties and methods of the object.

### Recordset Objects

For more information about recordset objects, see Chapter 17, “Binding a Container to a Record Source”.

For information on creating recordsets, see the section “Standalone Recordsets” on page 17.27.

A *recordset object* contains a local copy of a set of records queried from a database table or view. Like a table, a recordset is organized into rows and columns.

Recordset objects manage the relationship between application objects and database objects. As the user browses through rows of data on a form or report, the necessary information is fetched from the database. As the user makes changes to the data, the recordset records the changes and writes them to the database when appropriate.

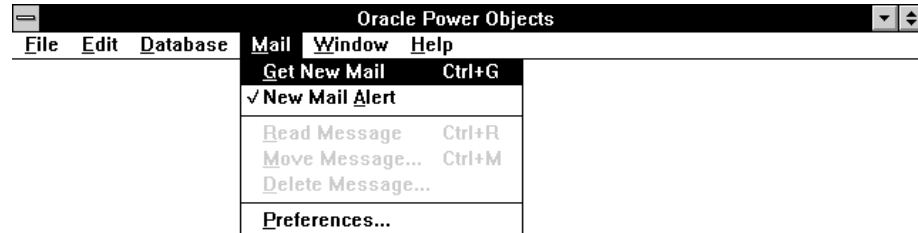
You define a recordset’s structure implicitly by associating objects in your application (such as forms, reports, and controls) with tables, views, and columns in a database. You can also create a recordset explicitly using the NEW operator.

The user accesses a recordset indirectly by manipulating containers and controls that are bound to a database object. You can access a recordset object directly in Oracle Basic code.

Menus, Menu Bars, Toolbars, and Status Lines are described in Chapter 14, "Menus, Toolbars, and Status Lines".

## Menus and Menu Bar Objects

A *menu bar* object contains one or more *menu* objects, each of which contains the definition of a menu. Menu bar objects can be associated with forms and reports.



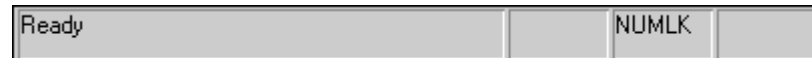
## Toolbar Objects

A toolbar object contains the definition of a toolbar. Toolbar objects, like menu bar objects, can be associated with forms and reports.



## Status Line Objects

A *status line* object provides summary information to the user. Status line objects, like menu bar and toolbar objects, can be associated with forms and reports.



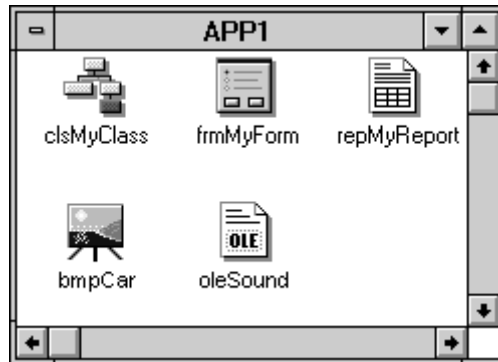
## Containers

Many types of objects can contain other objects. Containers provide logical groupings for the objects they contain and allow you to manipulate a collection of objects as a unit. For example, when you move or copy a form, you automatically move or copy the objects contained within the form as well.

Not all objects can act as containers, and different types of containers can hold different types of objects.

## Applications

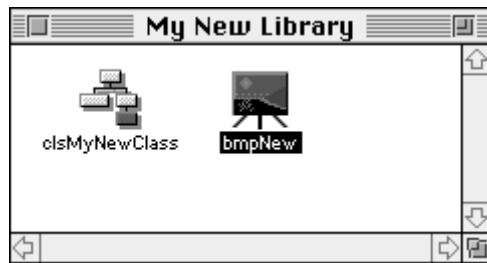
Applications can contain forms, reports, classes, bitmaps, and OLE objects. You can move and copy objects in an application from one application to another.



To see the objects contained within an application, open the application by double-clicking on its icon in the Main window.

## Libraries

Libraries can contain classes and bitmaps, which can in turn be used in applications or other libraries.

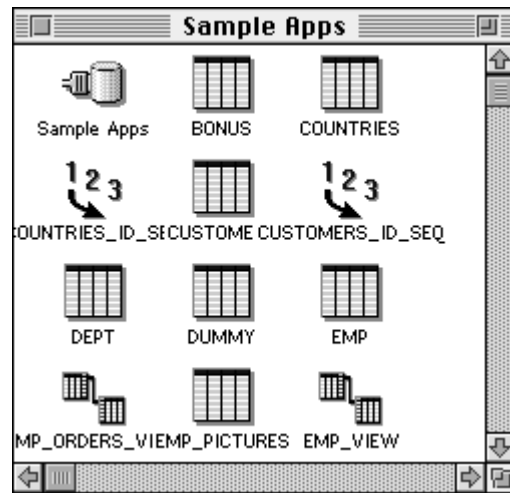


To see the objects contained within a library, open the library by double-clicking on its icon in the Main window.

---

## Sessions

Sessions are accessed as though they contain tables, views, indexes, and sequences. However, database objects are actually stored within and maintained by the database.

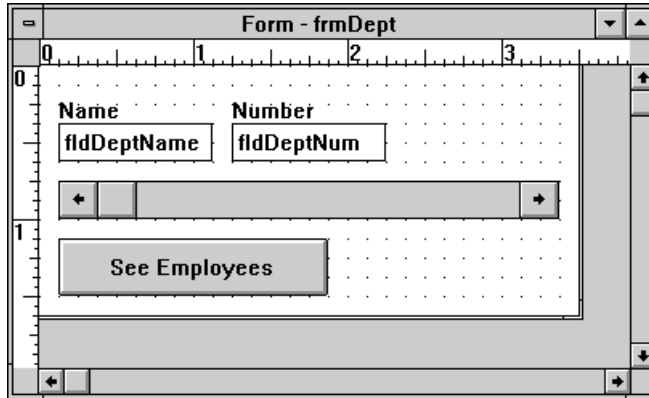


To see the objects “contained” within a session, open the session by double-clicking on its icon in the Main window. If the session is not currently active (connected to a database), you must also activate the session by double-clicking on the Connector control.

## Forms, Reports, and Classes

Forms, reports, and classes can contain a variety of objects. Objects on a form, report, or class usually have some kind of visual representation during run time.

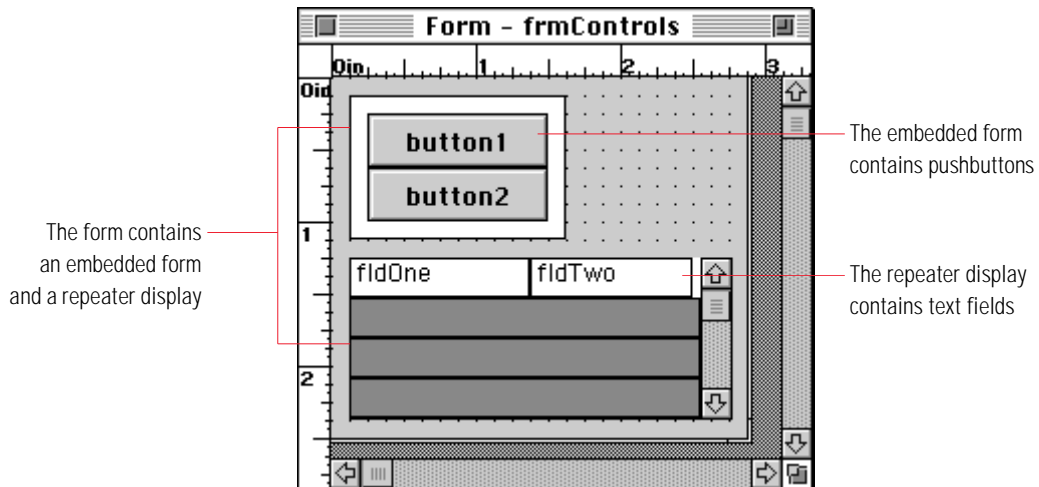
You edit the objects in a form, report, or class in the Form Designer, Report Designer, and Class Designer windows. When you open a form, report, or class, the contained objects are displayed automatically.



Reports can also contain *report group* objects, as described in Chapter 12, "Reports".

### Designer Objects

Some types of designer objects can contain other designer objects. For example, the following figure shows a set of pushbutton objects contained within a rectangle object, and a set of text fields contained within a repeater display object:



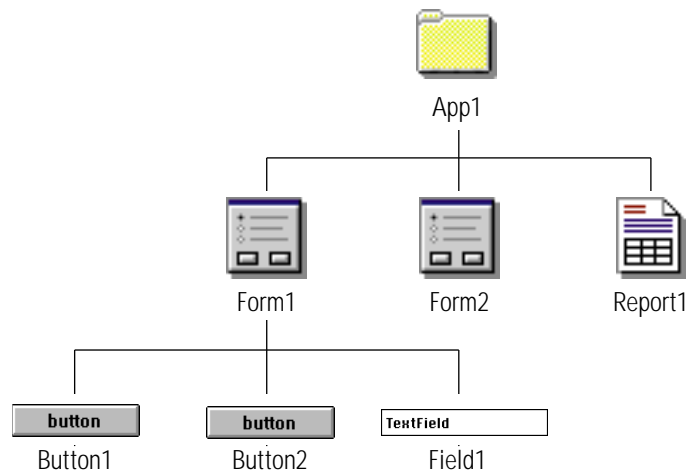
---

You select objects in the Designer windows by clicking on them. To select an object contained within another object, you click twice: once to select the container, and once to select the object. Each successive click selects the next level of objects.

## Object Containment Hierarchy

Objects contain one another in a hierarchy, called the *object containment hierarchy*. The object containment hierarchy is like the structure of a file system.

At the top level of the hierarchy is always a file object (an application, library, or session).



The following terms are used to refer to the positions of objects within the hierarchy:

- An object's *parent* or *container* is the object that immediately contains it. In the above figure, the form "Form1" is the parent of the pushbutton "Button1".
- All objects that share the same parent are called *siblings*. In the above figure, the pushbuttons "Button1" and "Button2" are siblings.
- An object's *children* are the objects immediately contained by it. In the above figure, the pushbutton "Button1" is a child of the form "Form1".
- A *top-level object* is an object that is not contained by any other object. A top-level object is always a file object (an application, library, or session). In the above figure, the application "App1" is the top-level object.

You can use the object containment hierarchy when referring to an object, as described in the section "Relative References" on page 3.25.



## Object Characteristics (Properties and Methods)

Objects have characteristics that can be customized by the developer. These characteristics are an intrinsic part of an object's definition—they are automatically moved, copied, or deleted along with the object.

Most objects have two types of characteristics: *properties* and *methods*.

**Properties** are object characteristics that contain values. Properties determine the appearance and behavior of objects. In developing an application, you set property values during both design time and run time.

**Methods** are procedural characteristics (subroutines and functions) that determine how the object responds to events or calls to the object. In developing an application, you call methods to accomplish various tasks, and add Oracle Basic code to methods to customize object behavior.

Each type of object in Oracle Power Objects has a predefined list of *standard* properties and methods that can be customized but cannot be removed. You can customize objects further by adding *user-defined* properties and methods, as described in Chapter 5, "Methods and Method Code".

You can set object characteristics during design time using the Property sheet; these settings are saved along with the application. Some properties can also be set during run time using Oracle Basic method code. Changes made during run time last only as long as the application continues to run.

To refer to an object characteristic, you typically append the property or method name to a reference to the object. The syntax of a characteristic reference is as follows:

### Syntax of Property and Method References

```
[ object_reference . ] characteristic_name [ ( argument
    [ , argument ... ] ) ]
```

*Object\_reference* is a reference to the object containing the property or method, as described in the section "Syntax of Object References" on page 3.28. If you omit *object\_reference*, the property or method is assumed to belong to the object containing the reference.

*Characteristic\_name* is the name of the property or method.

If the characteristic is a method that has *arguments*, the arguments must be specified in parentheses following the property or method name.

For example, to refer to the **Value** property of a field object, you might use a reference like the following one:

```
Form1.Field1.Value
```

---

To refer to the `QueryWhere()` method of a repeater display object, you might use a reference like the following one:

```
Repeater1.QueryWhere( "ENAME = 'KING' " )
```

## Properties

Properties control the appearance and behavior of an object. For example, some properties determine the name, color, size, or position of an object. Other properties determine how an object visible to the user (such as a text field) is linked to an object in a database (such as a column of a table).

The following table lists major categories of properties:

<b>Category of Properties</b>	<b>Examples</b>	<b>Description</b>
Control Behavior	<b>DefaultButton</b> <b>Enabled</b>	Properties that determine how controls behave in response to user actions.
Counter Generation	<b>CounterIncBy</b> <b>CounterTiming</b>	Properties that determine how unique values (such as primary key values) are derived from the database.
Internal Value	<b>DataType</b> <b>Value</b>	Properties related to the value stored in a control at run time, such as the source of the value and restrictions on the size or type of the value.
Miscellaneous	<b>Name</b> <b>WindowState</b>	
Object Appearance	<b>ColorFill</b> <b>FontName</b>	Properties related to the appearance of the object on screen, such as size, position, visibility, color, or font for text.
Hierarchical Reference	<b>Container</b> <b>Self</b>	Properties that return references to other objects in the object containment hierarchy, as described in the section “Relative References” on page 3.25.
Recordset	<b>DataSource</b> <b>DefaultCondition</b> <b>RecordSource</b>	Properties of a bound container or control that determine how the object’s values are derived from values in the database.
Report	<b>FirstPgFtr</b> <b>GroupCol</b>	Properties related to how a report is displayed or printed.
Session	<b>ConnectType</b> <b>DesignConnect</b>	Properties of a database session object.

For a full list of properties, see Appendix B, “List of Properties and Methods” or the topic “Standard Properties” in the online help.

### Property Datatypes

Each property has an associated datatype, which determines the type of value that can be stored in the property. Properties can have the following datatypes:

**Boolean** - a true/false value. A Boolean value is actually a value of datatype *Long Integer*. False corresponds to zero, True corresponds to -1.

**Long Integer** - a numeric value in the range -2147483648 to +2147483647.

**Double** - a floating-point value in the range +/- 2.2250738585072014e-308 to +/- 1.7976931348623158e+308 and zero.

**String** - a string of text characters.

**Date** - a date and time value in the range Jan 1, 100 AD to Dec 31, 9999 AD.

**Object** - a pointer to an object

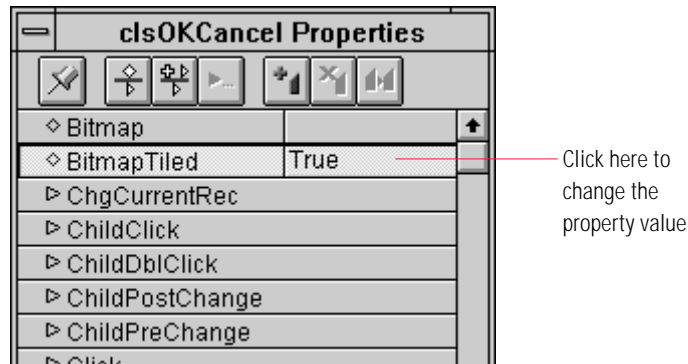
Some methods are limited to a predefined set of possible values. For example, Boolean methods are limited to two values: true or false. Other properties, such as **LinkMasterUpd**, can only be set to one of a list of values.

You change a property's value using the Property sheet or an Oracle Basic program. During development, you can also change a property's value using the Debugger.

### Setting a Property Using the Property Sheet

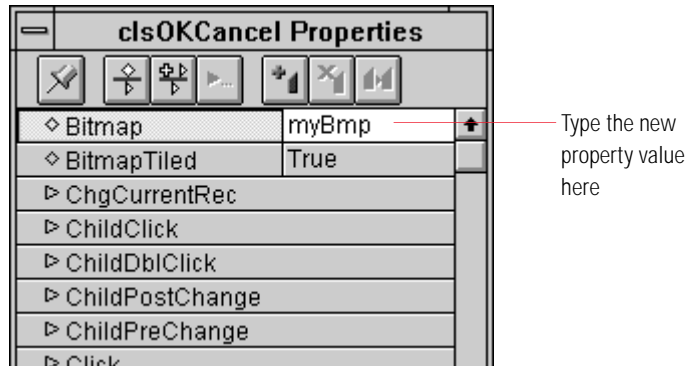
You can use the Property sheet to set or change a property's value during design time. When run-time mode is invoked, the property will initially contain the value you specify.

To change the value of a Boolean property, simply click in the area containing the word “True” or “False”. Each time you click, the property is set to the opposite value.

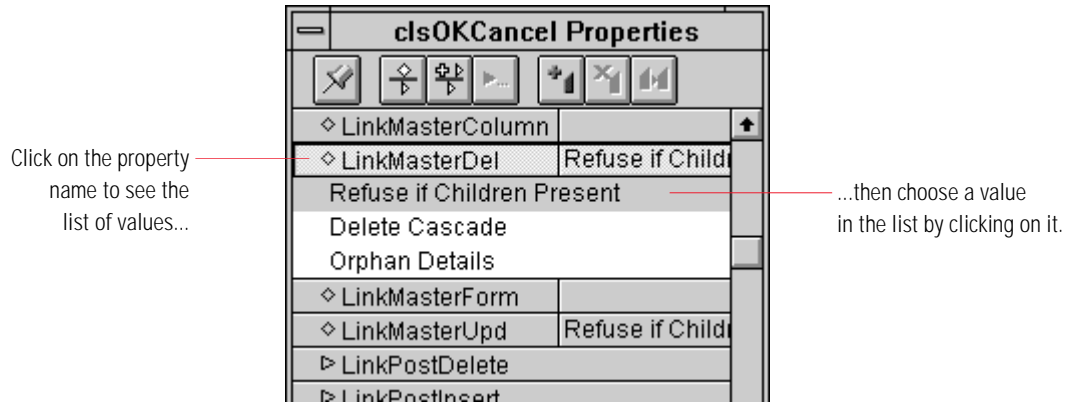


---

To set or change the value of a text-type property, you type directly in the area next to the property name.



To set or change the value of a list-type property, click on the name of the property. A list of possible values drops down beneath the property name. You set the property by clicking on the desired value.



You can also use the arrow keys to select a value in the list. To accept the selection, press Return.

### Setting a Property Using Oracle Basic

You can set the values of some properties during run time using the Oracle Basic assignment operator (=). Only properties that are designated in the online help as "writable at run time" in the property description can be modified through assignment.

To set the value of a Boolean property, you must assign it a numeric value: zero for False, any non-zero number for True (-1 is used by convention). You can also use the constants TRUE and FALSE.

```
Button1.Enabled = TRUE  
Button1.Enabled = -1
```

```
Field2.HasBorder = FALSE
Field2.HasBorder = 0
```

To set the value of a text-type property, you must assign it a string value, as in the following examples:

```
Frame1.Label = "Select Color:"
Field1.FontName = "Palatino"
Field2.FontName = Font_Name$
```

To set the value of a list-type property, you must assign it an integer corresponding to the desired value, as in the following example:

```
Field1.TextJustVert = 1
```

To improve the readability of this type of assignment, the Oracle Basic language includes a set of predefined constants for the possible values of list-type properties. For example, the following constants can be used to set the **TextJustVert** property:

```
TEXTJUSTVERT_TOP
TEXTJUSTVERT_CENTER
TEXTJUSTVERT_BOTTOM
```

To use the constant, you simply substitute it for the required value. For example, to set the **TextJustVert** property to "Top", you could use the following statement:

```
Field1.TextJustVert = TEXTJUSTVERT_TOP
```

The constants for setting list-type properties take the following form:

*property\_setting*

*property* is the name of the property.

*setting* is the text of the setting. Spaces are replaced by underscores.

A list of predefined constants is provided in Appendix C, "Constants and Reserved Words". The specific constants corresponding to each property value are described in the Power Objects online help topic for the property.

### Setting a Property Using the Debugger

You can also set the value of a property at run time using the Debugger. For information about using the Debugger, see the section "Debugging Method Code" on page 5.9.

## Methods

Methods determine how an object acts in response to two types of actions: *events* and *calls to the object*.

Calling a method is described in the section "Calls to the Object" on page 3.18.

---

## Events

An *event* is an occurrence or action that is recognized by the application. Events can be initiated when the user takes an action (such as clicking on a pushbutton), or by some other means (such as when the application loads a form into memory).

Examples of events:

- The user clicks on a pushbutton object (the pushbutton's **Click()** method is called).
- A form is loaded into memory (the form's **OnLoad()** method is called).
- The user has just inserted a new row of data into a form (the form's **PostInsert()** method is called).

## Calls to the Object

Not all methods are associated with events. Some methods are never executed unless the developer specifically makes a call to the method through Oracle Basic code. Other methods do respond to events, but might be called explicitly in certain situations.

You can make a call to an object to accomplish various development tasks. For example:

- You can call the **Connect()** method of a session to activate the session.
- You can call the **OpenWindow()** method of a form to open the form.
- You can call the **DeleteRow()** method of a repeater display to delete a row of data from the repeater display.

The following table lists major categories of methods:

<b>Category of Methods</b>	<b>Examples</b>	<b>Description</b>
Constraint/business rule	<b>RevertRow()</b> <b>Validate()</b>	Methods enabling you to validate data entered by the user before accepting it.
Container	<b>CloseWindow()</b> <b>RollbackForm()</b>	
Miscellaneous	<b>CloseApp()</b>	
Printing	<b>OpenPreview()</b> <b>OpenPrint()</b>	Methods enabling you to print or preview forms and reports.
Recordset	<b>ChgCurrentRec()</b> <b>GetColVal()</b> <b>InsertRow()</b>	Methods related to the recordset associated with a bound container, called to modify the recordset or to respond when the recordset is modified.
Session	<b>Connect()</b> <b>RollbackWork()</b>	Methods related to database session objects.
User action	<b>Click()</b> <b>FocusEntering()</b>	Methods called when the user takes an action in the interface, or called to initiate such an action.

For a full list of methods, see Appendix B, “List of Properties and Methods” or the topic “Standard Methods” in the online help.

## Types of Methods

Methods can be either *functions* or *subroutines*.

- A **subroutine** method is like a command. The subroutine can have one or more parameters.
- A **function** method returns a value. Functions, like subroutines, can have one or more parameters. The return value of a function can have any of the following datatypes:
  - *Long Integer* - a numeric value in the range -2147483648 to +2147483647.
  - *Double* - a floating-point value in the range +/- 2.2250738585072014e-308 to +/- 1.7976931348623158e+308 and zero.
  - *String* - a string of text characters.
  - *Date* - a date and time value in the range Jan 1, 100 AD to Dec 31, 9999 AD.
  - *Object* - a pointer to an object

A method's response to an event or a call can be determined in one or both of the following ways:

**Default processing.** If you do not customize a method, the event's *default processing* executes when the method is invoked. The default processing that is executed varies from method to method; some methods have no default processing, while others have extensive default processing. The default processing associated with each method is described in the online help.

**Method code.** Method code is an Oracle Basic function or subroutine that you write. You associate method code with a method by typing it directly into the Property sheet. Any method code you associate with a method overrides the default processing, which is not executed unless you specifically invoke it using the `Inherited.method_name()` syntax.

## Sequence of Methods

Methods are sometimes executed in sequence when responding to an event or call. This sequence can occur in two ways:

- The default processing of a method can invoke another method. For example, the default processing of the `Click()` method calls the `ChildClick()` method of the container of the clicked object. If you suppress the default processing of the `Click()` method, `ChildClick()` is not called.
- Methods can be called in sequence by Oracle Power Objects. For example, if you edit a master record in a master-detail relationship, Oracle Power Objects first calls the `PostChange()` method of the master container, and then calls the `LinkPostUpdate()` method of the detail container. `LinkPostUpdate()` is called even if the default processing of `PostChange()` is not executed.

Some events in Oracle Power Objects have a long method sequence. In order to respond to events correctly, you should understand the relevant method sequence.

---

Method sequences are described in the topic “Method Sequence” in the online help.

### Calling a Method

You call methods in the same way that you call other functions and subroutines. For example, you can invoke the **Click()** method of a pushbutton object with a statement like the following one:

```
Button1.Click()
```

You can call the **IsConnected()** method of a database session object with a statement like the following one:

```
a = Session1.IsConnected()
```

## Object Names

Most objects have a name, which is specified in the object’s **Name** property. You can use the name to refer to the object in properties and in Oracle Basic method code.

Oracle Power Objects provides other techniques for referring to objects, which are discussed in the section “Object Names” on page 3.20.

### Default Names

When you create an object, the object is usually assigned a default name, which you can change if you wish. The default name consists of the object type followed by an integer. For example, the first form created in an application is named “Form1”. The next form to be created is named “Form2”, and so on. If a default name is not available because an object already uses that name, Oracle Power Objects uses the next available number.

### Naming Rules

The following rules apply when assigning names to objects.

#### Database Object Names

Names for database objects must adhere to the object naming conventions for the database in which they are stored. These conventions vary from database to database.

For information about the database object naming conventions for Blaze databases, see the topic “Database Object Naming Conventions” in the online help. For the database object naming conventions for any other database, see the documentation accompanying that database.



### Application and Designer Object Names

Names for application objects and designer objects must adhere to the following rules:

- An object name must be from 1 to 39 characters long.
- An object name can contain letters, numbers, and the underscore character. Object names cannot contain any other characters, such as spaces, tabs, or returns. Additionally, the object name must begin with an alphabetic character or an underscore.
- An object name cannot be a reserved word. The following types of words are reserved:
  - The names of properties and methods, including user-defined properties and methods. Standard properties and methods are listed in Appendix B, "List of Properties and Methods".
  - All Oracle Basic reserved words (for example, commands, functions, and operators). Oracle Basic reserved words are listed in Appendix C, "Constants and Reserved Words".
  - The word "inherited"
  - The word "self"
- A name cannot be shared by two sibling objects (objects that share the same parent).
- Object names are not case sensitive, although case information is stored. The names "form1", "Form1", and "FORM1" are all considered to be the same.

### File Object Names

File objects have two names: the **Name** property of the object and the file name of the file in which the object is stored. These names do not have to be the same, and you can change either name at any time.

The **Name** property of a file object must adhere to the rules for naming application objects and designer objects. This is the name that appears beneath the object's icon in the Main window. You use this name to refer to the file object in properties and methods.

The file name of a file object is the name of the operating system file that holds the application, session, or library. This name does not have to adhere to the rules for naming application objects. However, the file name must adhere to the file naming conventions of the host operating system:

- In Windows, file names can have up to 8 characters, which can be letters, numbers, and the following special characters:

! @ # \$ % & ( ) - \_ { } ' ` ~

Windows file names also can have an *extension* of up to 3 characters. File objects in Windows use the following extensions:

<b>Object</b>	<b>Extension</b>
Application	.POA
Library	.POL

---

<b>Object</b>	<b>Extension</b>
Session	.POS

Windows file names are not case sensitive.

- On the Macintosh, file names can be up to 30 characters, which can be letters, numbers, and any symbols except a colon (:). Macintosh file names are not case sensitive, although case information is stored.

When you create a file object, the **Name** property is initially set to the file name that you specify. If the file name you specify does not follow the rules for naming application objects, Oracle Power Objects displays a dialog box prompting you to enter a valid name.

If you change the file name of a file object, the **Name** property of the object does not change. Similarly, changing the **Name** property of a file object does not change the file name of the object.



**Note:** If you change the file name of an object that is currently displayed in your Main window, Oracle Power Objects will be unable to find the file. You can open the file with the new name by clicking on the Open button.



## Renaming Objects

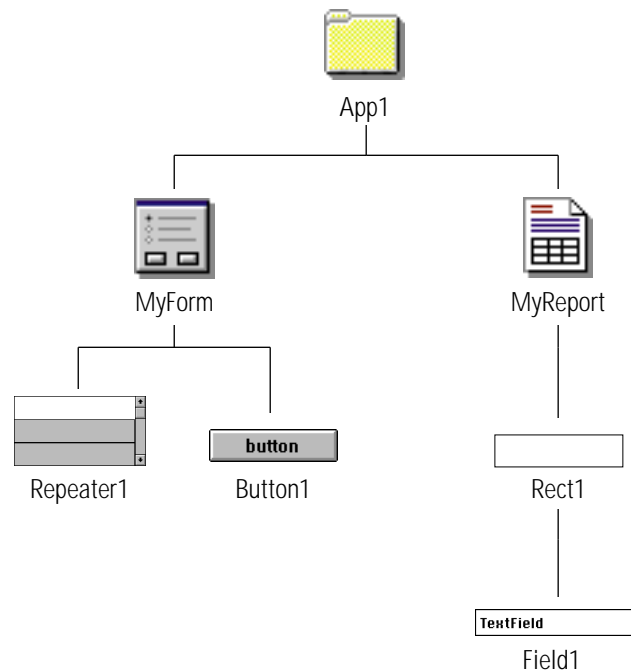
You can change the names of some types of objects at design time. Object names cannot be modified at run time.

If you rename an object by modifying its **Name** property, Oracle Power Objects cannot resolve references to the old object name. Changing the name of an object might cause an error when your application is compiled or run. You can correct this error by manually updating references to the object.

To avoid problems created by renaming objects, you should set the object's name as soon as you create it and avoid subsequently changing it.

## Hierarchical Names

You can refer to an object by specifying both its name and its location in the object containment hierarchy. This is called the object's *hierarchical name*. A hierarchical name consists of object names separated by periods. In a hierarchical name, the object to the left of the period contains the object to the right of the period.



For example, the button object “Button1” on the form “MyForm” could be described by the following hierarchical name:

```
MyForm.Button1
```

Similarly, a field object “Field1” in the rectangle “Rect1” on the report “MyReport” could be described by the following hierarchical name:

```
MyReport.Rect1.Field1
```

You can also use the keyword “Application” in a hierarchical name to refer to the application object in which the other objects are contained.

For example, to refer to the repeater display object “Repeater1”, you could use the following reference:

```
App1.MyForm.Repeater1
```

---

The formal syntax of a hierarchical name is as follows:

### Syntax of Hierarchical Names

```
{ object_name [.object_name] ...  
| APPLICATION [.object_name [.object_name] ...] }
```

Object\_name is the **Name** property of an object.

### Full Hierarchical Names

A reference that names all of the objects containing an object is called the object's *full hierarchical name*. The full hierarchical name always refers to the same object, even if there is another object with the same **Name** property elsewhere in the application.

A full hierarchical name always begins with the name of the application object or the identifier "Application", then includes the names of the objects in order of containment, separated by periods.

For example, the full hierarchical name of an object "Button2" located on a form "Form1" in the application "App1" is:

```
App1.Form1.Button2
```

Alternatively, the name can be written this way:

```
Application.Form1.Button2
```

### Partial Hierarchical Names

When you refer to an object, it is not always necessary to specify its full hierarchical name. For example, to refer to the report "Report1" from within method code associated with a property of the form "Form1", you could use the following reference:

```
Report1
```

Similarly, for any object contained within Report1, you do not have to specify its full hierarchical name—you only have to specify its name beginning with the report. For example, to refer to the field "Field1" on the report, you could use the following reference:

```
Report1.Field1
```

When you specify only part of the hierarchical name, Oracle Power Objects attempts to *resolve* the name (find the object the name refers to) by comparing the name you specify to the names of other objects in the application. References are resolved by examining objects beginning with the *current object* (the object from which the reference is made).

Oracle Power Objects resolves name references by examining objects in the following order:

- 1 Each child of the current object
- 2 The children of each child of the current object
- 3 If the object is not a form, report, or class, the parent of the current object
- 4 The children of the parent of the current object
- 5 Repeat steps 3 and 4 at the next level of the hierarchy

## Object References

You use an *object reference* when you need to identify an object in your application. For example, you use object references to read or set object properties, or to execute object methods. You also use object references to associate different types of objects with each other. For example, you use an object reference in the **ScrollObj** property of a scrollbar object to identify the container whose records can be scrolled through using the scrollbar.

You can specify an object reference in one of three ways:

- You can specify the object's full or partial hierarchical name, as described in the preceding section
- You can use an Oracle Basic variable of datatype *Object*
- You can specify a property or method that returns a value of datatype *Object*

### Object Datatype

Another way to refer to an object is to store in a variable or property a “handle” that refers to the object. An object handle is stored in a variable or property of datatype *Object*.

The following example stores a reference to the object “Form1.Button1” in the variable `vButton`. The variable is then used to execute the **Click()** method of the button.

```
Dim vButton as Object
vButton = Form1.Button1
vButton.Click()
```

The *Object* datatype is described in more detail in the section “Values and Datatypes” on page 4.3.

### Relative References

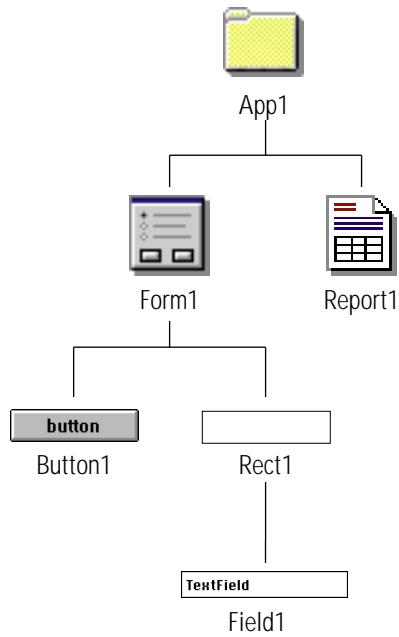
You can refer to an object by its relative position in the object containment hierarchy. Instead of specifying an object's name, you can substitute a property or keyword that describes its relationship to another object. This is called a *relative reference*.

---

You can use the following properties, methods, and keywords in a relative reference:

- The **GetContainer()** method of an object refers to the object's parent.
- The "Container" keyword refers to the parent of the object preceding it in the reference.
- The **GetTopContainer()** method refers to the top-level container of the object (the form, report, or class where the object is located).
- The "TopContainer" keyword refers to the top-level container of the object preceding it in the reference.
- The **FirstChild** property of a container object refers to the "first" child of the object. Oracle Power Objects arbitrarily chooses the "first" child from among the object's children.
- The **NextControl()** method of a container object returns a reference to the "next" child of the object. Each time **NextControl()** is evaluated, it returns a reference to another child object, until all of the child objects have been returned. After the last child object has been returned, **NextControl()** returns null.
- The **GetFirstForm()** method of an application object returns a reference to the "first" child of the application (a form or report). Oracle Power Objects arbitrarily chooses the "first" child from among the forms and reports in the application.
- The **GetNextForm()** method of an application object returns a reference to the "next" child of the application (a form or report). Each time **GetNextForm()** is evaluated, it returns a reference to another child object, until all of the child objects have been returned. After the last child object has been returned, **GetNextForm()** returns null.

Consider the following object containment hierarchy:



The following table describes how relative references would be resolved within this hierarchy:

<b>Reference</b>	<b>Refers to This Object</b>
Button1.Container	Form1
Button1.GetContainer()	Form1
Field1.GetContainer().GetContainer()	Form1
Field1.TopContainer	Form1
Field1.GetTopContainer()	Form1
Rect1.Container.FirstChild	Button1
Form1.FirstChild	Button1
Form1.NextControl() *	Rect1
Form1.NextControl() *	Null
Application.GetFirstForm()	Form1

---

<b>Reference</b>	<b>Refers to This Object</b>
Application.GetNextForm() *	Report1
Application.GetNextForm() *	Null

\* The results indicated apply only if references are evaluated in the order stated.

### Syntax of Object References

The full syntax of object references is as follows:

```
{hierarchical_name | object_variable | characteristic_name}
 [.characteristic_name [.characteristic_name] ...]
```

*hierarchical\_name* is the full or partial hierarchical name of an object, as described in the section “Syntax of Hierarchical Names” on page 3.24.

*object\_variable* is an Oracle Basic variable of datatype *Object*.

*characteristic\_name* is the name of a property of datatype *Object* (such as **ScrollObj**) or of a method returning a value of datatype *Object* (such as **NextControl()**).

### Restrictions on Object References

You use object references in two places: in Oracle Basic method code, and in property values of datatype *Object*.

When you specify an *Object* property value at design time (such as when you enter the name of a container in the **ScrollObj** property of a scrollbar), the reference cannot contain any properties or methods. Because of this restriction, you cannot use relative references that contain properties or methods.

You can, however, use the keywords “Container” and “TopContainer” in the reference. The “Container” keyword is like the **GetContainer()** method: it returns a reference to the parent of the object. The “TopContainer” keyword is like the **GetTopContainer()** method: it returns a reference to the top-level container of the object.

For example, you could set the **ScrollObj** property of a scrollbar object to the following value:

```
Scroll1.Container
```

However, the following value is *not* valid, because it contains a reference to a property:

```
Form1.FirstChild
```

In a user-defined class, the “Container” and “TopContainer” keywords are evaluated differently from **GetContainer()** and **GetTopContainer()**. “Container” and “TopContainer” cannot refer to any object above the level of the class. However, the **GetContainer()** and **GetTopContainer()** methods can be used to refer to objects above the level of the class—for example, you can use these methods to refer to the form containing an instance of the class.



## Object Classes and Inheritance

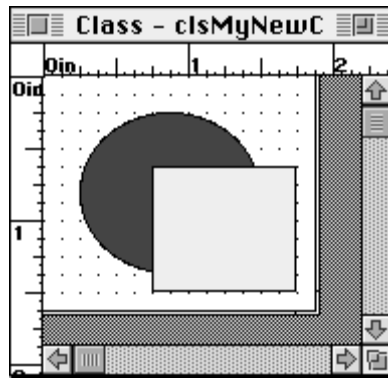
Class objects have a set of features that let you reuse and link components of your application. You can use the definition of a class object, including its property settings, the objects it contains, and its Oracle Basic method code, as a master for creating linked copies of the object called *instances* and *subclasses*.

An **instance** is a linked copy of a class object that appears on a form, report, or class.

A **subclass** is a class object that is based on the definition of another class object. As with a class, you can create an instance of a subclass.

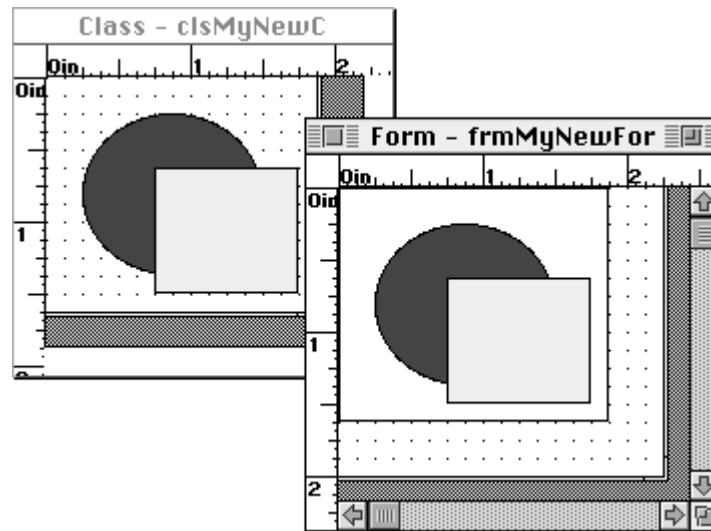
Instances and subclasses *inherit* the features of the object from which they are created—they share the same controls, property settings, and Oracle Basic method code as the master object. Any changes you make to the master object are automatically reflected in all of the object instances and subclasses, but you can also make local changes to an instance or subclass.

The following diagram shows the definition of a class called “clsMyNewClass”:

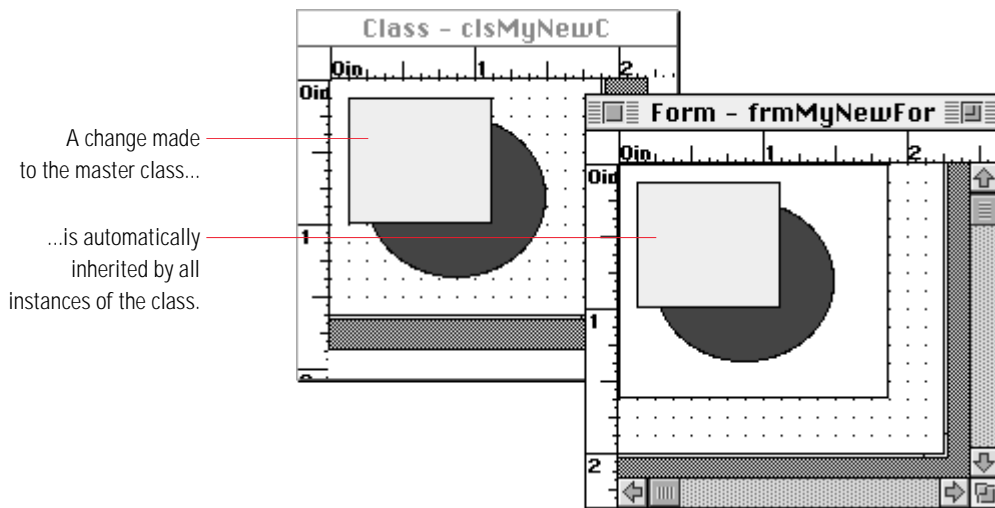


---

The following diagram shows an instance of the class “clsMyNewClass” on a form called “frmMyNewForm”:



The following diagram shows how the instance inherits changes made to the class definition:



Normally, an instance or subclass inherits all of the features of the master object. If you make a modification to an instance or subclass, the inheritance relationship for that feature is *overridden*. Once a feature is overridden, changes to that feature of the master object are not reflected in the instance or subclass.

Overridden properties or methods are indicated by filled-in diamond or arrow symbols in the property sheet.

It is possible to re-establish the inheritance relationship for properties and methods. To reinherit an object characteristic, select the name in the property sheet of the instance or subclass and click on the Reinherit button.



Some features that have been overridden in a subclass or instance cannot be reinherited. For example, if you have deleted an object within a subclass or instance, the deleted object cannot be restored.

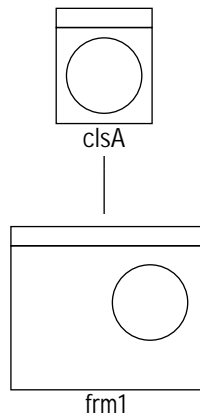


**Important:** Do not delete a class object when instances or subclasses of the object still exist. After the class object is deleted, you will be **unable** to open or run any object that contains instances of the deleted class.

### Object Inheritance Hierarchy

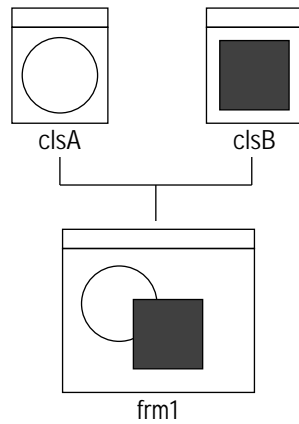
The relationships among classes, subclasses, and instances are described by the *object inheritance hierarchy*. The object inheritance hierarchy is like a family tree—a single object definition can inherit features from several different classes and subclasses.

The most simple type of object inheritance hierarchy is when an instance of a class inherits the features of a single master class. For example, in the following diagram, the form “frm1” contains an instance of the class “clsA”:

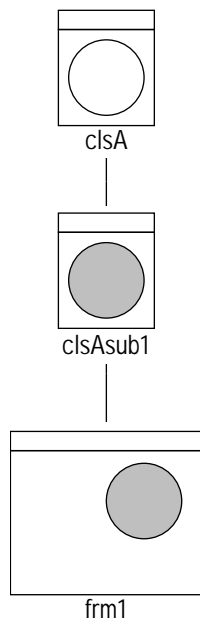


---

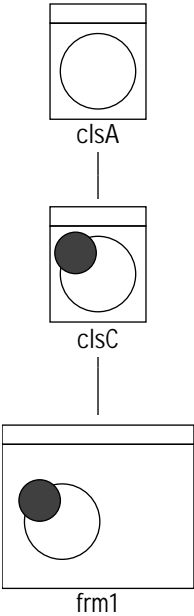
Instances of several different objects can appear in the same container. For example, in the following diagram, the form “frm1” contains instances of two classes, “clsA” and “clsB”:



The inheritance hierarchy becomes extended when subclasses or classes containing instances of other classes are introduced. For example, in the following diagram, the class “clsA” has a subclass called “clsAsub1”, and the form “frm1” contains an instance of “clsAsub1”:

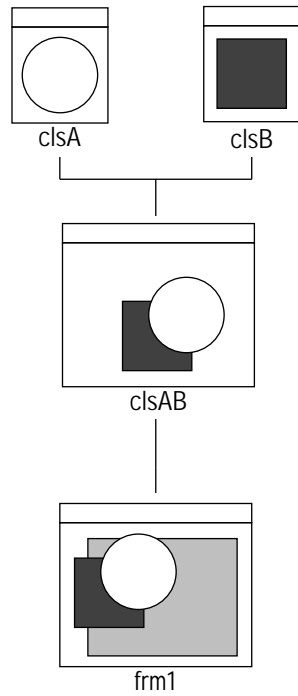


In the following diagram, the class "clsC" contains an instance of the class "clsA". The form "frm1" contains an instance of "clsC".



---

The inheritance hierarchy becomes more complex when a class contains instances of multiple master classes. For example, in the following diagram, the class “clsAB” contains instances of both “clsA” and “clsB”. The form “frm1” contains an instance of “clsAB”.



# 4

---

## Oracle Basic

This chapter covers the following topics:

Overview .....	4.2
Oracle Basic Language Components .....	4.3
Values and Datatypes .....	4.3
Literals .....	4.4
Variables .....	4.6
Symbolic Constants .....	4.10
Operators .....	4.12
Built-in Functions .....	4.17
Commands .....	4.21
Expressions .....	4.22
Object Properties .....	4.23
Object Methods .....	4.23

---

## Overview

Oracle Basic is a programming language that enables you to customize the existing capabilities of Oracle Power Objects and to create new ones. This full-featured language is an integral part of Oracle Power Objects, providing full support for its object-oriented features.

Oracle Basic commands enable you to define constants, variables, and procedures. They also make it possible to control input-output operations and branching for all the paths needed during execution to meet the needs of your applications and their users.

Oracle Power Objects incorporates most of the standard operations and procedures you need to prototype and implement successful database applications. These built-in components comprise the default processing for the standard Oracle Power Objects methods. You can then use Oracle Basic to modify, supplement, or replace this default processing.

This chapter describes the built-in components of Oracle Basic. (More detailed explanations can be found in the Oracle Power Objects online help). Chapter 5, “Methods and Method Code”, describes how you associate Oracle Basic code objects you design and how you invoke the code.

The Oracle Basic language has the following essential language components, all of which are described in this chapter:

- **Values.** A value is a piece of information with an associated datatype. Other components of Oracle Basic enable you to specify and manipulate values.
- **Literals.** A literal is a fixed data value, such as 32 or 'SMITH'.
- **Variables.** A variable is a name representing a value that can vary.
- **Symbolic constants.** A constant is a name representing a fixed data value.
- **Operators.** An operator manipulates individual values and returns a result.
- **Expressions.** An expression represents a value. Expressions can consist of literals, object values, and functions, singly or combined by operators.
- **Built-in functions.** A function performs calculations and returns a result. Functions usually operate on one or more *arguments* that you specify when you call the function, although some functions have no arguments.
- **Commands.** A command performs calculations or other operations. Commands and subroutines frequently return information, but they do not return a result value in the same way that functions do.

In addition, Oracle Basic provides support for the following object characteristics:

- **Object properties.** A property is an object characteristic that stores a value.
- **Object methods.** A method is a procedural characteristic of an object. Methods can be either *functions*, which return a value, or *subroutines*, which do not.

These elements and conventions are described briefly in the sections that follow; a more complete discussion appears in the online help.



## Oracle Basic Language Components

### Values and Datatypes

A *value* in Oracle Basic is a datum with a specific datatype. Commands, functions, and methods use values as parameters; variables and object properties store values; and operators and functions produce values as output.

Values are derived from *expressions*, which are discussed in the section “Expressions” on page 4.22.

For constants, the datatype is usually obvious: 3.14159265 is a noninteger numeric value; “Hello” is a string value.

The datatype of a variable is less immediately apparent. You can append to variables predefined suffixes (listed later in this chapter) to make their type instantly recognizable, but these are not required. If you do not use suffixes, you can declare any valid variable name to be of any valid type. If you make no declarations, Oracle Basic infers the appropriate type from the first context in which you use each variable.

Oracle Basic uses data of the following types:

<b>Datatype</b>	<b>Description</b>	<b>Range/Size</b>
<i>Null</i>	Indicates the absence of a value	
<i>Integer</i>	16-bit signed integer	-32,768 to +32,767
<i>Long Integer</i>	32-bit signed integer	-2,147,483,648 to +2,147,483,647
<i>Single</i>	4-byte single-precision floating-point value	+/- 1.401298E-45 to 3.402823E30; and zero
<i>Double</i>	8-byte double-precision floating-point value	+/- 4.94066E-324 to 1.79763134862315E308; and zero
<i>Date</i>	A value storing date and time information	Jan 1, 100 AD to Dec 31, 9999 AD
<i>String</i>	A value storing a string of text	1 to 32,767 characters (minus a small amount for system overhead)
<i>Object</i>	A reference to an object, such as a container or control	
<i>Variant</i>	Can store values of any datatype	

You cannot declare variables with the *Null* datatype, but a variable of any datatype can be assigned a null value.

---

A variable of datatype *Variant* can store any value given to it at any time, without the limitations imposed on variables specified as any other type. For example, a *Variant* variable can be assigned a *Date* value, then an *Object* value, then a *Long* value. Variables with a datatype other than *Variant* cannot store values of other datatypes (except Null, as described above). For example, once a variable is typed as *String*, attempting to store a number in it generates an error message.

➤ **Technical note:** All Oracle Basic values are stored internally as *Variant* values and interpreted by Oracle Basic within the ranges of values appropriate to the declared datatype.

You can use the VARTYPE function to check the datatype of any value—the value to be checked is included as the argument to the function. VARTYPE returns a *Long Integer* value that indicates the datatype of the value to be checked. The values returned by VARTYPE are listed in the following table:

<b>Value Type</b>	<b>VARTYPE Value</b>	<b>Notes</b>
Null	1	
<i>Integer, Long</i>	3	<i>Integer</i> and <i>Long</i> are the same internal type interpreted in different ranges.
<i>Single, Double</i>	5	<i>Single</i> and <i>Double</i> are the same internal type interpreted in different ranges.
<i>Date</i>	7	
<i>String</i>	8	
<i>Object</i>	9	

## Literals

A *literal* is a fixed data value. Each datatype has its own format for specifying literal values.

### Numeric Literals

Numeric literals are composed of digits. For numbers with fractional parts, a decimal point is used to separate the integer part from the fractional part. You can use a sign (+ or -) at the beginning of the number. You can also use scientific notation, indicated by the character “e” or “E”.

The datatype of a literal value is derived from the magnitude and format of the value. Oracle Basic generally assigns to a value the smallest datatype required to represent the value. The following table shows how Oracle Basic assigns datatypes to several example literal values:

<b>Value</b>	<b>Datatype</b>
3	<i>Integer</i>

<b>Value</b>	<b>Datatype</b>
32763	<i>Integer</i>
32788	<i>Long</i>
3.1	<i>Single</i>

### Text Literals

Text literals are strings of text that can include letters, numbers, spaces, tabs, and other special characters. You must always delimit (surround) literal strings with double quotation marks. The following values are text literals:

```
"Smith"
"The correct response is 42"
"Coming & Going"
```

To represent a double quote mark inside a literal, you can concatenate a double quote character generated with the CHR built-in function, as in the following example:

```
"He asked, " & CHR(34) & "Why did you go?" & CHR(34)
```

The CHR function is described in the topic "CHR Function" in the online help.

### Date and Time Literals

Date and time literals are text strings containing date and time information. You must delimit (surround) literal date and time values with pound symbols (#). Oracle Power Objects allows you to use most common date and time formats to specify date and time literals. The following values are date and time literals:

```
#1/31/94#
#12:00 AM#
#January 31, 1994#
#01-JAN-94 12:00:01#
```

If you specify a text literal containing date and time information where a date value is required, Oracle Basic will convert the text value to a date or time value, according to the following rules:

- If the string contains only one number (N), the date interpretation is "January 1, N".
- A string of the form "A-B-C" is interpreted as "year-month-day" if A > 100. Therefore, specifying a value above 12 for B or above 31 for C causes an error. If A < 100, Oracle Power Objects uses the date format specified in your operating system to determine which value (B or C) represents the month. The date format is specified in the "International" control panel in Windows, in the "Date & Time" control panel on the Macintosh.

- 
- If the string contains only two numbers (X Y), the date interpretation is for month and day, in the current year defined by the system date. If either X or Y is greater than 12, it is taken as the day, and the other as the month. If both are less than 12, the interpretation uses the control panel format. If either is greater than 31 or if both are greater than 12, an error occurs.
  - If the string contains three numbers (X Y Z):
    - Any values above 12 and 31 are taken as the month and year respectively, regardless of position in the string.
    - Two values above 31 or three between 12 and 31 cause an error.
    - A two-digit year is always taken as the nearest four-digit year ending with those two digits. For example, entering “8/19/21” during 1995 is interpreted as “August 19, 2021”. If the year of the system date were 2072, “8/19/21” would be interpreted as “August 19, 2121”.

For explicit control over the format in which you enter date and time values, you can use the CVDATE function. CVDATE is described in the online help.

## Nulls

You can specify a null value explicitly by using the word NULL.

## Variables

An Oracle Basic variable is a name representing a value that can vary. Variable names are constructed according to specific rules.

### Variable Names

A variable name must begin with a letter and can be up to 39 characters long. You can use any letter, number, or the underscore ( `_` ) character. An underscore is often used to represent a space, as in the following variable name:

```
client_invoice
```

All other characters are disallowed in variable names, including spaces, tabs, and line-break characters (carriage returns and line feeds).

## Declaring Variables

Variables can be declared either *implicitly* or *explicitly*.

You declare a variable *implicitly* by using a previously undeclared name as an output value (for example, as the output operand to the assignment operator). If the last character of the declaration is a variable type suffix, the variable's datatype is determined by the suffix; otherwise, the variable is of type *Variant*. Variable type suffixes are:

Datatype	Suffix
<i>Integer</i>	%
<i>Long Integer</i>	&
<i>Single</i>	!
<i>Double</i>	#
<i>String</i>	\$

Note that these suffix characters are *not* part of the variable name.

You declare a variable *explicitly* using a DIM, GLOBAL, REDIM, or STATIC statement. These statements have meanings as follows:

Statement	Meaning
GLOBAL	Declares array or nonarray variables that will be accessible from any method anywhere in your application. You must declare global variables in the (Declarations) section of an application.
DIM	Declares array or nonarray variables that are local to the method (subroutine or function) in which they are declared. Variables declared through DIM become undefined as soon as the method finishes execution.
STATIC	Declares array or nonarray variables that are local to the method (subroutine or function) in which they are declared, but retain their values throughout an application's execution.
REDIM	Declares local array variables and can redefine their dimension and extent.

You can assign a datatype in an explicit declaration, either by specifying a type suffix or through the AS clause. To specify a type suffix, simply add the appropriate suffix character to the name, as in the following examples:

```

DIM vIntegerVar%           ' Integer
DIM vLongArray&(100)      ' Long Integer
DIM vDoubleVar#           ' Double
GLOBAL gStringGlobal$     ' String
    
```

---

When you specify the AS clause, you specify the name of the variable datatype, as in the following examples:

```
DIM vIntegerVar AS Integer
DIM vLongArray(100) AS Long
DIM vObjectVar AS Object
GLOBAL gStringGlobal AS String
```

You cannot include both a type suffix and an AS clause in a variable declaration. For example, the following declaration is invalid:

```
DIM vInteger% AS Integer      'Invalid declaration
```

If you do not include a type suffix or AS clause, the variable is automatically declared as type *Variant*.

Certain variables are declared automatically by Oracle Power Objects in the body of a method (subroutine or function): a variable is automatically declared for each method parameter. These variables are always local in scope.

Whether a variable is declared implicitly or explicitly, the declaration establishes the name and datatype of the variable. You must always refer to the variable in the same way after declaration, including its type suffix (if specified at declaration).

### Variable Initial Values

The initial value for each variable datatype is as follows:

<b>Datatype</b>	<b>Default Initial Value</b>
All Numeric Types	Zero (0)
<i>Date</i>	Null
<i>String</i>	Zero-length string ( " ")
<i>Object</i>	Null
<i>Variant</i>	Null

### Scope of Variables

The *scope* of a variable declaration determines where the variable can be referenced. The scope of a variable is either *global* or *local*.

A variable of **global** scope is always available to any method in the application. You make a variable global in scope by declaring it with the GLOBAL keyword. Global variables can only be declared in the (Declarations) section of an application.

A variable of **local** scope is available only to the method (subroutine or function) in which the variable was declared. Variables you declare implicitly or with the DIM, STATIC, or REDIM statements are local in scope.

## Arrays

An *array* is a variable that can contain multiple values of the same datatype. Each value is stored in a separate array *element*. The *dimension* of an array indicates how many separate values are required to specify a unique element—in other words, the number of logical data axes of the array. For example, an array of dimension 1 has one axis, while an array of dimension 3 has three axes.

Each dimension has an *extent*, a range of values that indicates which elements are defined along the dimension. The extent represents the “size” of the dimension. For example, an extent of “0 to 10” represents an 11-element dimension. An extent of “-30 to -10” represents a 21-element dimension.

To refer to an element of an array, you specify one or more *index* values in parentheses following the array name. You specify one index value for each array dimension; each index value must fall within the extent of its dimension. For example, the following reference specifies element 4 of the array `vArray_1D`:

```
vArray_1D(4)
```

The following reference specifies the element at coordinates (3, 2) in the array `vArray_2D`—that is, the third element along the first dimension, the second element along the second dimension:

```
vArray_2D(3,2)
```

You can use an array element anywhere you can use a nonarray variable—for example, to receive a value from an assignment operation, or to provide an argument to a function.

To declare an array, you include parentheses following the variable name in a GLOBAL, DIM, STATIC, or REDIM statement. You can also include values to specify the dimension of the array and the extent of each dimension.

The full syntax of the DIM statement is as follows (GLOBAL, STATIC, and REDIM have similar syntaxes):

```
DIM varname [ ( [[lbound TO] ubound] ) ] [AS datatype]
      [varname [ ( [[lbound TO] ubound] ) ] [AS datatype]]...
```

`varname` is the name of the variable.

`lbound` is an integer representing the lower bound of a dimension's extent. `lbound` can be negative; however, `lbound` must be less than `ubound`. If `lbound` is not specified, the lower bound of the extent is zero.

`ubound` is an integer representing the upper bound of a dimension's extent. `ubound` can be negative only if a lower `lbound` value was specified. Each `ubound` value you specify represents a different array dimension.

---

`datatype` is a keyword representing the variable's datatype. If `datatype` is not specified, the variable is of datatype *Variant*.

For example, the following statement declares an array `vArray_1D` that has a dimension of 1:

```
DIM vArray_1D(0 TO 10)
```

The following statement declares an array `vArray_2D` that has a dimension of 2:

```
DIM vArray_2D(0 TO 10, 0 TO 10)
```

You can create a *dynamic* array by declaring an array without specifying the dimension and extents of the array. A dynamic array has a dimension of 1 when first declared, but you can subsequently modify the dimension and the extent of each dimension with a `REDIM` statement.

## Symbolic Constants

*Symbolic constants* (also called *constants*) are names, constructed like variable names, that you can use in place of specific constant values (such as numbers or strings). You define the value represented by the constant when you declare the constant.

### Constant Names

A constant name, like a variable name, must begin with a letter and can be up to 39 characters long. You can use any letter, number, or the underscore (`_`) character. All other characters are disallowed in constant names, including spaces, tabs, and line-break characters (carriage returns and line feeds).

### Declaring Constants

You declare a constant using the `CONST` statement, as in the following examples:

```
CONST BTN_OK = 1
CONST BTN_CANCEL = 2
CONST FIRST_STATE = "Delaware"
CONST MAX_WINDOWS% = 255
```

If the last character of the constant declaration is a variable type suffix, the constant's datatype is determined by the suffix; otherwise, the constant's datatype is determined by the datatype of the value assigned to it. Oracle Basic uses the simplest type that accurately represents the constant; for example, *Integer* is used for whole numbers below 32,767, and *Long* for higher integral values. Because a constant's value does not change, constants cannot have datatype *Variant*.

A constant's scope is determined by where the constant is declared. A constant declared in the (Declarations) section of an application is global in scope. You do not need to use the `GLOBAL` keyword in the declaration. A constant declared anywhere else is local in scope.



Using symbolic constants can make your code easier to read and understand, providing more immediate significance than literal values and more inherent stability than variables. Your code becomes less subject to inadvertent errors, easier for others to understand, and easier to modify. When you modify the value assigned to a constant, you modify the value used wherever the constant appears. This can also help you use the same code in different environments.

You can make your code more understandable by establishing a convention for all constants, making them easy to recognize as one reads your program. For example, you could always use underscores as the first and last characters of a constant.

### Predefined Constants

Oracle Basic includes a number of predefined constants to help clarify your code. You can use these constants anywhere in your application.

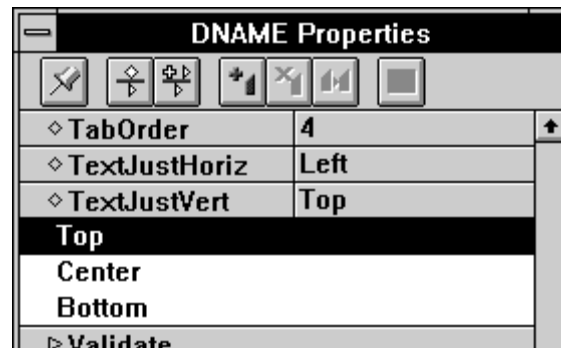
**Boolean constants.** Oracle Basic defines these constants to represent boolean (logical) values:

```
CONST TRUE = -1
CONST FALSE = 0
```

You can use these constants wherever a boolean value is required.

➤ **Note:** In most cases, you can use any nonzero number to represent the Boolean value True, not just -1. The value -1 is used for the True constant because of the operation of the bitwise NOT operator: NOT(-1) = 0, and therefore NOT(TRUE) = FALSE.

**Property value constants.** For list-type properties, Oracle Basic includes a set of predefined constants representing the possible values of the property. For example, the `TextJustVert` property has three possible values, as shown in the following diagram:



The following predefined constants can be used to set the `TextJustVert` property:

```
TEXTJUSTVERT_TOP
TEXTJUSTVERT_CENTER
TEXTJUSTVERT_BOTTOM
```

---

You can then use the constant by substituting it for the required value. For example, to set the **TextJustVert** property to "Top", you could use the following statement:

```
Field1.TextJustVert = TEXTJUSTVERT_TOP
```

The constants for setting list-type properties take the following form:

```
property_setting
```

*property* is the name of the property.

*setting* is the text of the setting. Spaces are replaced by underscores.

The specific constants corresponding to each property value are described in the Oracle Power Objects online help topic for the property.

## Examples

You might define a set of constants corresponding to the output values of the **VARTYPE** function, for use in checking the datatype of a value. The following code establishes a set of such constants, which would be easier to read and understand quickly than the integers they represent:

```
CONST VAR_NULL      = 1,      &
      VAR_INTEGER   = 3,      &
      VAR_LONG      = 3,      &
      VAR_SINGLE    = 5,      &
      VAR_DOUBLE    = 5,      &
      VAR_DATE      = 7,      &
      VAR_STRING    = 8,      &
      VAR_OBJECT    = 9
```

After declaring these constants, you can use them to check the datatype of a value, as in the following example:

```
SELECT CASE VARTYPE(vReturnValue)
      CASE VAR_NULL
          MSGBOX "The return value is null."
      CASE VAR_OBJECT
          MSGBOX "The return value is an object."
      CASE ELSE
          MSGBOX "The return value is: " & vReturnValue
END SELECT
```

## Operators

Oracle Basic provides several categories of operators for constructing or comparing arithmetic or string expressions, relational expressions, and logical expressions. This section describes the operators available in each category.

## Operator Precedence

Operator precedence establishes the order in which operations are performed when an expression is evaluated. Each category of operators is listed in order of relative precedence in an expression. Operators appearing at the top of the list have the highest precedence (they are evaluated first); operators appearing at the bottom of the list have the lowest precedence (they are evaluated last). Operators of equal precedence are evaluated from left to right. In general, arithmetic operators are evaluated before comparison operators, and comparison operators are evaluated before logical operators.

You can use parentheses to group operations in an expression. Operations inside parentheses or brackets are evaluated before operations outside parentheses.

For example, the numeric expression "2 \* 8 + 4 / 2" can return different values depending on how the operands are grouped, as shown in the following table:

<b>Grouping</b>	<b>Result</b>
2 * 8 + 4 / 2	18
(2 * 8) + (4 / 2)	18
((2 * 8) + 4) / 2	10
(2 * (8 + (4 / 2)))	20
2 * ((8 + 4) / 2)	12

## Arithmetic Operators

Arithmetic operators manipulate numeric operands and return a numeric value.

<b>Operator</b>	<b>Description</b>	<b>Example</b>	<b>Result</b>
^	Exponentiation	2^3	8
-	Negation	-(2+5)	-7
*, /	Multiplication, division	3*12/4	9
\	Integer division	17.46\3	5
Mod	Modulo arithmetic	17 mod 3	2
+,-	Addition, subtraction	9 + 11 - 5	15

As described in the section "Date Operators" on page 4.16, you can also use some arithmetic operators with *Date* values.

---

## String Operators

String operators manipulate string operands and return a *String* value.

Operator	Description	Example	Result
&	Concatenation—implicitly converts operands of any datatype to string values before concatenation.	"ant" & "hem" "anthem" & 1 "anthem" & NULL	"anthem" "anthem1" "anthem"
+	Concatenation—performs no implicit conversion of operands.	"writ" + "ten" "written" + 1 "written" + NULL	"written" Error NULL

## Comparison Operators

Comparison operators manipulate operands of many different datatypes and return a boolean value represented as a *Long Integer*. A return value of -1 means True; a return value of 0 means False.

Operator	Datatypes Supported	Description	Example	Result
=	All	Equality	(12/3 = 28/7)	True
<>	All	Inequality	(12/3 = 35/7)	False
<	All except <i>Object</i>	Less than	"b" < "bb"	True
>	All except <i>Object</i>	Greater than	"z" > "bb"	True
<=	All except <i>Object</i>	Less than or equal	5 <= 119	False
>=	All except <i>Object</i>	Greater than or equal	cos(1.6) >=2	False

## Logical Operators

Logical operators manipulate numeric operands and return a boolean value represented as a *Long Integer* value. A return value of -1 means True; a return value of 0 means False.

Operator	Description	Example	Result
NOT	Logical negation	NOT(1=2)	True
AND	Logical conjunction	(2=18/9) AND (1=2)	False
OR	Logical disjunction	(2=18/9) OR (1=2)	True
XOR	Exclusive disjunction	(2=18/9) XOR (7=7)	False
EQV	Logical equivalence	(6<2) EQV (2<3)	False

Operator	Description	Example	Result
IMP	Logical implication	(6<2) IMP (2<3)	True

### Bitwise Operations

The Oracle Basic logical operators perform *bitwise* operations on their operands. Bitwise operations compare individual digits of the binary representation of numbers.

When operating on logical values (-1 for True and 0 for False), these operators return the results described in the previous section. However, you can use these operators with other numeric values, as shown in the following table:

Operator	Example	Bitwise Equivalent
AND	198	1100 0110
	AND 173	AND 1010 1101
	= 132	= 1000 0100
	68	1000 0100
AND	AND 0	AND 0000 0000
	= 0	= 0000 0000
EQV	87	0101 0111
	EQV 205	EQV 1100 1101
	101	= 0110 0101
IMP	6	0000 0110
	IMP 42	IMP 0010 1010
	= 251	= 1111 1011
OR	157	1001 1101
	OR 195	OR 1100 0011
	= 223	= 1101 1111
	10	0000 1010
OR	OR 0	OR 0000 0000
	= 10	0000 1010
XOR	95	0101 1111
	XOR 150	XOR 1001 0110
	= 201	= 1100 1001
	0	0000 0000
XOR	XOR 0	XOR 0000 0000
	= -1	= 1111 1111
NOT	NOT 56	NOT 0011 1000
	= 199	= 1100 0111

---

<b>Operator</b>	<b>Example</b>	<b>Bitwise Equivalent</b>
NOT	0	NOT 0000 0000
=	-1	= 1111 1111

➤ **Note:** The exact bitwise representation of a numeric value is determined by the processor on which Oracle Power Objects is running.

One useful application of the bitwise operators is to check the setting of a particular bit of a numeric value using the AND operator. For example, the following method code checks the setting of third bit of the value in the variable `vNumber` (the bit representing  $2^2$ , that is, 4). The code displays a dialog with the word “set” if the bit is set, “not set” if the bit is not set.

```
IF (vNumber AND 4)
  MSGBOX "Set "
ELSE
  MSGBOX "Not set "
END IF
```

If `vNumber` were set to the value 6 (binary 0000 0110), the preceding code would display “set”, because the third bit of the value is set. If `vNumber` were set to the value 10 (binary 0000 1010), the preceding code would display “not set”, because the third bit of the value is not set.

Similarly, you can set the value of a particular bit using the OR operator. For example, the following method code sets the fourth bit of the value in the variable `vNumber` (the bit representing  $2^4$ , that is, 8):

```
vNumber = vNumber OR 8
```

If the fourth bit of the value in `vNumber` is set, it remains set. If the fourth bit is not set, it becomes set.

You check or set individual bits of numeric values when working with certain expressions, such as the `dlg_type` argument to the MSGBOX command, or the **Value** property of a current row pointer control. For more information, see the online help.

### Date Operators

Date operators manipulate date and numeric operands and return a *Date* or numeric value.

<b>Description</b>	<b>Example</b>	<b>Result</b>
Date - Date	4/13/95 - 4/8/95 = 5	Yields the number of days between them. If the time portion of the dates differ, the result includes a fractional component indicating the difference in time.
Date + Number	4/13/95 + 23 = 5/6/95	Yields a date later than original by the specified number of days

Description	Example	Result
Date - Number	4/13/95 - 312 = 6/5/94	Yields a date earlier than original by the specified number of days.

### Object Operators

Object operators return an *Object* value.

Operator	Description	Example
NEW	Object creation	mbrMenuBar1 = NEW MenuBar

Only *in-memory* objects (recordsets, menu bars, menus, toolbars, and status lines) can be created with the NEW operator.

### Built-in Functions

A *function* performs calculations and returns a result. *Built-in functions* are part of the Oracle Basic language; you cannot modify the behavior of built-in functions. Object methods can also be defined as functions, as described in Chapter 5, "Methods and Method Code"; you can customize object methods by adding method code to them.

You can use a function wherever a value is required by specifying the function name, followed by the function arguments in parentheses. Thus you can use it in an assignment statement, either alone or as part of a larger expression, or as an argument to some other function, either built-in or user-defined.

For example, to call the built-in function SQR, you could use the following method code:

```
DIM vResult AS Double
vResult = SQR(2)
```

### Built-in Functions

Built-in Oracle Basic functions have the following general categories:

Category	Description	Examples
File Input/Output	Enable you to read information from and write information to operating system files.	CURDIR EOF SEEK
General		ASC SPACE ENVIRON

---

<b>Category</b>	<b>Description</b>	<b>Examples</b>
Array/Subscript	Return information about the lower and upper bounds of array extents.	LBOUND UBOUND
Selection	Allow you to choose a value from multiple alternatives	CHOOSE IIF
Mathematical, Statistical, and Trigonometric	Perform operations on numeric values.	ABS COS SQR
SQL-related	Allow you to execute custom SQL statements and return error information after execution of a SQL statement.	SQLROWCOUNT SQLLOOKUP
String	Manipulate string values.	FORMAT LEN UCASE
Conversion	Explicitly convert values to a specified datatype or value format.	CINT VAL HEX
Aggregation	Calculate values based on values in a set of database rows. Aggregation functions are described below.	AVG COUNT
Date	Return system date and time information, and manipulate date and time values.	CVDATE DATE NOW
Financial	Perform financial calculations.	MIRR PMT RATE

For a complete reference to the built-in functions, see the topic “Oracle Basic Functions” in the online help.

### **Aggregation Functions**

An *aggregation function* is a built-in function that calculates a single value based on all the rows in a recordset, such as the count of such rows or the sum of column values. The argument provided to these functions is an expression that must include a reference to one or more bound controls from a single container, such as a text field in a repeater panel or on a form.

The following aggregation functions are available:



Function	Description
MAX	maximum value
MIN	minimum value
COUNT	number of non-null values
AVG	mean average of all values
STDEV	standard deviation of all values
SUM	sum of all values

The argument to any aggregation function is an expression including a reference to a bindable control associated with a recordset. For example, the following function calculates the sum of salaries displayed on the field “fldSal” on the form “frmEmp”:

```
vSalarySum% = SUM(frmEmp.fldSal.Value)
```

The following function calculates a count of all employees displayed on the field “fldEmp” on the repeater display “rptEmpDetail”:

```
vEmployeeCount% = COUNT(rptEmpDetail.fldEmp.Value)
```

In control references, you do not have to specify the **Value** property explicitly—a reference to the control object is assumed to refer to its **Value** property. For example, the following reference is identical to the preceding one:

```
vEmployeeCount% = COUNT(rptEmpDetail.fldEmp)
```

Aggregation functions are useful both in method code and in *derived values* (calculations located in the **DataSource** property of a control). For example, you could display a count of employees using a field object with the following **DataSource** property:

```
=COUNT(rptEmpDetail.fldEmp.Value)
```

In either case, the aggregation function must be invoked from an object *not contained by* the bindable container over which the aggregation is to be done—that is, not in the same container as those controls. For example, an aggregation function that refers to a field in a repeater display must be located outside the repeater display (on the form containing the repeater display, on another container within the form, or on a different form). An aggregation function that refers to a control on a form must be located on a different form.

You can specify compound expressions as the argument to an aggregation function. For example, you can use the SUM aggregation function to calculate the sum of a complex expression, as shown below:

```
vWithholdTax% = SUM((frmEmp.fldSal * .10) + (frmEmp.fldCom * .25))
```

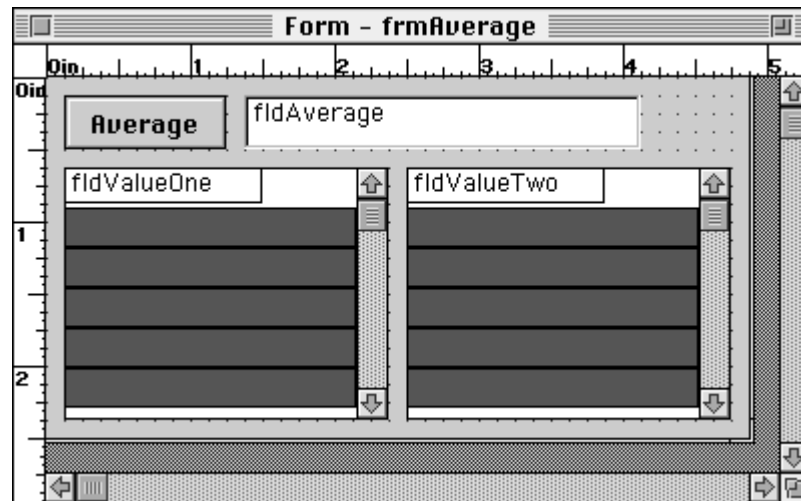
You can use an expression of any type in an aggregation function as long as all objects referenced in the expression are bound controls located within a single bound container.

---

The expression argument to an aggregation function is subject to the following rules:

- It must be of a datatype allowed by the function:  
AVG, SUM, and STDEV require a numeric argument.  
MIN and MAX require a numeric or string argument.  
COUNT can operate on any datatype.
- It must contain at least one reference to a bound control.
- It cannot contain references to object properties (other than **Value**) or methods.
- It cannot contain references to controls in multiple containers.

To illustrate the use of aggregation functions in an application, consider the following form, which contains a button and two separate repeater display objects:



The following commands could be added to the **Click()** method of the button object:

```
MSGBOX ( "The average is " & AVG(field1))  
MSGBOX ( "The average is " & AVG(field3))  
MSGBOX ( "The weighted average is " & AVG(field1+field2*2))  
MSGBOX ( "The average is " & AVG(field3))
```

However, the following command is invalid, because it attempts to aggregate values from fields in different containers:

```
MSGBOX ( "The average is " & AVG(field1+field3))
```

Similarly, the following commands are invalid for methods associated with any objects contained by **Repeater1**, because they are not in a container separate from the fields being aggregated:

```
MSGBOX ( "The average is " & AVG(field1))  
MSGBOX ( "The average is " & AVG(field2))
```

## Commands

A *command* is an Oracle Basic keyword that performs an operation but does not return a value. To use a command, you use its name on a line by itself. This usage is called *invoking* or *calling* the command. You pass *arguments* to a command when required, often in parentheses. Some commands require parentheses; others do not allow them. See the online help for the syntax of individual commands.

For example, to call the MSGBOX command, you could use the following method code:

```
MSGBOX "There has been a serious error."
```

## Commands

Oracle Basic commands have the general categories listed in the following table:

Category	Description	Examples
Database Management and Interaction	Enable you to send custom SQL statements to a database, as described in Chapter 9, "Structured Query Language (SQL)".	EXEC SQL
Execution Control	Control the flow of command execution. Allow you to define loops, jump to a specified code line, and handle errors.	CALL FOR WHILE IF
Variable and Constant Definition and Control	Declare variables and constants.	CONST DIM LET
Comments	Enable you to define code comments, as described in the section "Commenting Code" on page 5.8.	REM
Procedure Definition	Declare functions and subroutines. You do not have to enter procedure definition commands explicitly in Oracle Basic.	DECLARE END
File Input/Output	Allow you to read information from and write information to operating system files.	CLOSE PRINT#
Directory/File Management	Allow you to create, delete, and navigate among directories, and lock or rename files.	CHDIR LOCK
Miscellaneous		BEEP

---

For a complete reference to the built-in functions, see the topic “Oracle Basic Commands” in the online help.

## Expressions

An Oracle Basic *expression* can be formed by any of the following elements alone or combined with appropriate operators:

- Literals
- Constants
- Functions (both built-in functions and Object methods that are functions)
- Object properties
- Variables

The following values are expressions:

```
"Hello"  
10  
3.1415 * 2.71828  
TRUE AND 4<3  
SQRT(A^2 + B^2)  
"Press the " & LABEL & " button to do it."  
frmEmp.fldSal.Value * .25
```

## Expression Evaluation

An expression is *evaluated* to derive the value it represents. Expression evaluation involves evaluating functions and performing operations specified by operators. When functions are evaluated, the function is replaced by the return value of the function.

## Datatype Conversions

Because an expression represents a value, every expression has a datatype. The individual elements composing an expression also have datatypes. When elements of different types are combined in an expression, Oracle Basic performs certain implicit datatype conversions to simplify the expression and its result. The following guidelines further describe these operations.

## Operator Conversions

The ampersand (&) concatenation operator implicitly converts operands of all datatypes to *String* values before operating on them.

For the order in which Oracle Basic evaluates operators, see the section “Operator Precedence” on page 4.13.

## Numeric Conversions

*Implicit* numeric conversions are used when numeric values of different types are mixed together.

- Numeric expressions perform their indicated operations using the highest precision or capacity present in the values or variables that make up the expression. Thus, when adding an integer value to a double-precision value, Oracle Basic first “promotes” the integer to a double-precision value. Then the addition takes place, and a double-precision value is returned as the expression result.
- Integer division (`\`) and the `MOD` operator always return *Long* values.
- When floating-point values (*Single* or *Double*) are implicitly converted to integer values (*Integer* or *Long*), the fractional part is truncated. For example, `MM% = 3.1415` would cause `MM%` to hold the value 3.
- If a floating-point value (*Single* or *Double*) is too large to be assigned to an integer variable or property, an error occurs. For example, the statement `MM% = 32777` causes an error because the limit for *Integer* values is 32767. Similarly, if a *Double* value is too large to be assigned to the *Single* datatype, an error occurs.

To convert a numeric datatype *explicitly*, you can use a conversion function provided by Oracle Basic, such as `CDBL`, `CSNG`, `CINT`, `CLNG`, and `CSTR`.

## Object Properties

Object properties are described in the section “Properties” on page 3.14.

*Object properties* are characteristics of an object that store a value. Object properties behave like Oracle Basic variables. For example, you can assign values to such a property or use the property to represent a value in expressions.

Note that the scope of an object property or method is always global—it can be referenced from anywhere in the application.

## Object Methods

Object methods are described in the section “Methods” on page 3.17.

*Object methods* are procedural characteristics associated with objects. *Standard* methods are associated with objects by default; *user-defined* methods can be added after object creation. You can customize methods by adding Oracle Basic code to them, as described in the section “Writing Method Code” on page 5.5. Some methods also have *default processing* that executes if you do not customize their behavior.

Methods can be either functions or subroutines. Function methods return a value, just as built-in functions do, while subroutines do not.

A complete list of standard methods is included in Appendix B, “List of Properties and Methods”. For a complete reference to each method, see its topic in the online help.



# 5

---

## Methods and Method Code

This chapter covers the following topics:

Overview .....	5.2
Triggering Methods.....	5.2
Creating a Method .....	5.4
Writing Method Code .....	5.5
Methods, Default Processing, and Method Code.....	5.6
Overloading Method Declarations.....	5.7
Suggestions and Cautions .....	5.7
Debugging Method Code .....	5.9
The Run-Time Debugger .....	5.9
Setting a Breakpoint .....	5.12
Removing a Breakpoint .....	5.13
Setting a Watchpoint .....	5.13
Moving Execution to Any Point Within the Method.....	5.14

---

## Overview

A *method* is a procedural characteristic of an object. For example, the **OpenWindow()** method of a form loads the form into memory and displays it in the application interface. Methods are where you write Oracle Basic code to customize your application's behavior—all Oracle Basic code appears in methods. This chapter describes how you invoke methods and how to add Oracle Basic method code to them.

A method can be either a *function* or a *subroutine*. Both types of methods perform calculations; however, functions return result values while subroutines do not. Methods may (but need not) operate on one or more *arguments*.

Oracle Power Objects includes a set of predefined *standard* methods, most of which include *default processing* that executes when the method is triggered. You can also add *user-defined* methods to objects; user-defined methods have no default processing.

You can customize standard and user-defined methods by adding Oracle Basic method code. For standard methods, you can either completely override the default processing or interpose your own code before or after invoking that default processing.

### Triggering Methods

When a method is *triggered*, the method code or default processing associated with the method is executed. A method can be triggered in one of two ways:

**Through an event.** An *event* is an action that occurs in the application's interface. Events occur when a user takes an action (such as clicking on a control or closing the application), or in response to system activity.

**By calling the method.** You can call a method explicitly through Oracle Basic code, as described in the next section.

For more information about default processing, see the section "Methods, Default Processing, and Method Code" on page 5.6.



## Calling a Method

To call a method (either standard or user-defined) in Oracle Basic method code, use the following syntax:

```
[object_reference.] method_name [ ( [argument
    [, argument] ...] ) ]
```

`object_reference` is a reference to the object containing the property or method, as described in the section “Object Names” on page 3.20. If you omit `object_reference`, the property or method is assumed to belong to the object containing the reference.

`method_name` is the name of the method.

If the characteristic is a method that has `arguments`, the arguments must be specified in parentheses following the property or method name, as described in the next section.

For example, to refer to the `QueryWhere()` method of a repeater display object, you might use a reference like the following one:

```
Repeater1.QueryWhere("ENAME = 'KING'")
```

If the method is a *function*, it can return a value that you can use directly in an expression or in an assignment statement. A function method can be used like a built-in Oracle Basic function.

For example, the `GetRecordset()` method is a function. `GetRecordset()` returns a value of datatype *Object*, which you can assign to a variable. You can then use the variable wherever an reference to the recordset object is required. The following method code stores the return value of `GetRecordset()` in the variable `recVar`, and then uses `recVar` to execute methods of the recordset object:

```
DIM recVar AS Object
recVar = Form1.GetRecordset()
recVar.Connect()
```

If the method is a *subroutine*, its name is used like a command rather than like a function—that is, alone on a line rather than in an expression or assignment statement. Subroutines do not return values directly; however, a subroutine can modify an argument passed to it.

You can also call a subroutine by using a `CALL` statement. For example, the following method code acts identically to the previous example:

```
CALL sesOracle.Connect()
```

## Arguments

During execution, you pass *arguments* to a subroutine or function when required. These arguments are specified in parentheses following the method name. Function arguments can be passed either by *reference* (the default) or by *value*.

---

When you pass an argument by *reference*, you pass a “handle” to the memory space occupied by the variable used to specify the argument. Only variables can be passed by reference; any other type of expression is automatically passed by value. Any changes made to the argument while the function is executing affect the variable in the calling method.

In contrast, when you pass an argument by *value*, you pass only the current value assigned to the argument, not a reference to the argument itself. Any changes made to the argument have no effect outside of the subroutine or function. All nonvariable expressions are automatically passed by value; variables are passed by value only if the keyword BYVAL is used in the declaration of the function.

## Creating a Method

To create a user-defined function or subroutine, you must declare it and add it to an object in your application. Frequently, you add a function or subroutine to the form, class, or report where you will call it, but in fact you can add it to any application object.

Creating a user-defined method involves the following steps:

- Declaring the user-defined method.
- Adding the method to an application object.

### ☆ To declare a new user-defined function or subroutine:



- 1 Open the User Properties window by clicking the Add Property/Method button on the Property sheet or by choosing the **View-User Properties** menu command.
- 2 Scroll to the end of the list.
- 3 On the blank line at the end of the list, click in the Name field and enter the name of the new function.

The name you enter must not be a reserved word. However, you are permitted to *overload* a method by entering the name of an existing user-defined or standard method; method overloading is described in the section “Overloading Method Declarations” on page 5.7.

- 4 Click under the Type heading and select `Function` or `Sub` from the dropdown list.
- 5 For a function, click in the Datatype field and select the datatype for the return value.
- 6 If your function or subroutine will have arguments, click in the Arguments field and enter the arguments, specifying the datatype for each one.

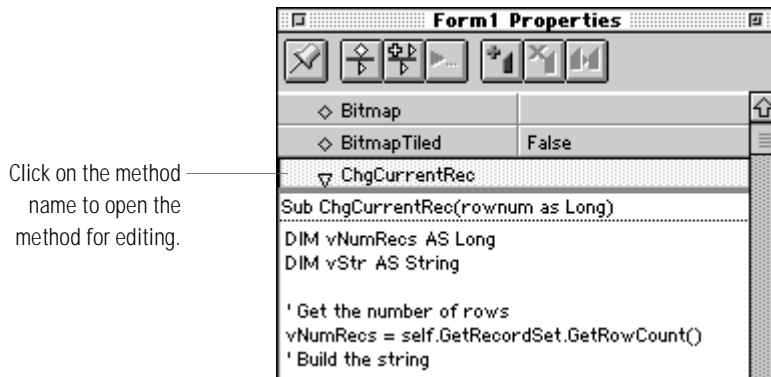
☆ **To add a function or subroutine to an application object:**

- 1 Open the User Properties window.
- 2 Click on the selection rectangle (the button to the left of the function or subroutine description) to select the function or subroutine.
- 3 While holding the mouse button down, drag from the selection rectangle to the chosen application object (or its property sheet) where you want to use the function or subroutine definition.  
The function or subroutine appears as part of the Property sheet, where it becomes a user-defined method.
- 4 The code window for this new method automatically opens when you complete the drag operation. Type in the desired Oracle Basic code defining what the function or subroutine does.

## Writing Method Code

To customize the behavior of a method, you add *method code* to it. Method code can be added to any standard or user-defined Oracle Power Objects method.

You add method code through the Property Sheet. All methods on the Property Sheet are indicated by arrow symbols. To open a method for editing, simply click on the method name, as shown in the following example:



The first line directly under the method name contains the method declaration—a SUB or FUNCTION statement. The declaration includes the name of the method, the method's arguments, and (for functions) the return value. You add method code in the area beneath the method declaration.

- 
- **Note:** Do not add an explicit END SUB or END FUNCTION statement to your method code. Oracle Power Objects automatically appends the appropriate statement to your code. If you include an END SUB or END FUNCTION statement, Oracle Power Objects returns an error.

Oracle Power Objects implicitly declares a set of variables in any method. A variable is declared for each argument in the method declaration; the variable is of the datatype indicated in the declaration. You can use these variables in your code like any other values. Note that assigning values to these variables is meaningful only if the argument was passed by reference; an argument passed by value can be set, but the argument variable disappears when the method ends.

For function methods, you assign the function's return value by assigning values to the name of the function. This name is not actually a variable; it is simply a convention for representing the return value. For example, to set the return value of the **Validate()** method, you assign a value to the name `Validate`, as in the following example:

```
Sub Validate() AS Long
  IF Self.Value < 0 THEN
    Validate = FALSE
  ELSE
    Validate = TRUE
  END IF
```

To terminate a method in the middle of execution, you can use the EXIT FUNCTION or EXIT SUB command. Either command terminates method execution immediately, and the application continues execution where the method was first called.

## Methods, Default Processing, and Method Code

Whenever you enter Oracle Basic code into a method, you automatically override any default processing normally performed by that method—the default processing is not executed. For a description of the default processing executed by each method, see the method topic in the online help.

You can, however, invoke that default processing by adding `Inherited.method_name` to your method code. For example, to invoke the default processing for the **Click()** method, you would include the following line in your method code:

```
Inherited.Click()
```

The default processing is executed at the point in the sequence of code where you call `Inherited.method_name`. You can, therefore, precede or follow that default processing with custom Oracle Basic code. If the standard method you are invoking takes arguments, then you must pass appropriate arguments in your call to invoke the `Inherited.method_name()`

procedure. For example, the `PreDelete()` method takes the number of the row to be deleted as its argument. To invoke the default processing for `PreDelete()`, you must include the row number as an argument, as in the following example:

```
Sub PreDelete(rownum as Long)
  MsgBox "About to delete row " & rownum
  Inherited.PreDelete(rownum)
```

The default processing for a method often includes the calling of a series of other methods in a predefined order. In such cases, the series of other methods in the default processing executes immediately after the `Inherited.method_name` statement, *before* the flow of control returns to any method code you may have placed after that statement.

## Overloading Method Declarations

Oracle Power Objects permits you to “overload” a method declaration by defining two methods that share the same name but have a different number of arguments. For example, you might want to create an “Add” function that behaves one way if it is passed two numeric values, another way if it is passed three values.

```
Sub udmAdd(Num1 AS Double, Num2 AS Double) AS Double
  udmAdd = Num1 + Num2

Sub udmAdd(Num1 AS Double, Num2 AS Double, Num3 AS Double) &
  AS Double
  udmAdd = Num1 + Num2 + Num3
```

You could add both of these methods to the same object, because they have a different number of arguments. Any subsequent call to the “udmAdd” method would be directed automatically to the appropriate method, depending on the number of arguments specified.

## Suggestions and Cautions

The following suggestions and cautions are provided to assist you in writing method code.

### Line Continuation

To break a single line of method code over multiple lines, you use the *line continuation character* at the end of a line to be continued on the next line. The line continuation character is an ampersand (&). For example, you could break a call to the `AppendMenuItem()` method over multiple lines as shown below:

```
mnuMail.AppendMenuItem("Move Message...", Cmd_MoveMessage, &
  0, "^M")
```

---

Note that this use of the ampersand is different from the ampersand concatenation operator—the line continuation character does not perform any concatenation. If you need to use the concatenation operator at the same location as the line continuation character, you specify two consecutive ampersands, as in the following example:

```
vReturnVal = MSGBOX("There was a problem connecting. " & &
    "Do you want to try again?", 49)
```

Note also that you cannot break a line in the middle of a literal string. To break such a line, you must divide it into two separate literal strings and concatenate them together, as in the previous example.

### Executing Multiple Statements on a Single Line

To execute more than one statement in a single line of code, you separate the statements with a colon (:). For example, you could execute both an assignment and a MSGBOX command with the following line of code:

```
varMessage = fldMessage.Value : MSGBOX fldMessage.Value
```

### Commenting Code

Your code is easiest to modify, repair, enhance, and maintain when it contains comments explaining what it is doing (and why and how). Oracle Power Objects provides two ways to comment your code: using an apostrophe or the letters `rem` to separate the comment from Oracle Basic code you want executed.

On any line, Oracle Basic ignores all characters to the right of an apostrophe or the keyword `REM` (or `rem`). There must be white space preceding `REM` or `rem`. If `REM` is on the same line as other text (for example, a command), it must be preceded by a colon (:).

The following examples demonstrate the use of code comments:

```
REM This text is ignored.
ON CHOICE GOTO 1,3,5 'Text after the apostrophe is ignored.
REM ON CHOICE GOTO 1,3,5 The ON CHOICE command is ignored.
ON CHOICE GOTO 1, 3, 5 rem Everything after "rem" is ignored.
```

### Coding Standards

To simplify both writing and reading method code, and to increase the portability and usefulness of your code, you should adopt a set of *coding standards* that determine naming and usage conventions. A set of suggested standards are provided in Appendix A, “Suggested Coding Standards”. These standards are used in the Oracle Power Objects documentation and sample applications. You can use these conventions as written, or adapt them to suit your needs.

## Creating a Repository for Methods

You can make a central repository for your functions and subroutines by creating a form that holds nothing except the Oracle Basic code defining all functions and subroutines. Calling any method of the form object automatically loads it into memory, but the form remains hidden. Alternatively, you can make the function part of a library object, usable by multiple applications.

### Cautions

When writing functions and subroutines, be careful to avoid the following errors:

**Self-invoking (recursive) methods.** A recursive method is one whose execution invokes itself again in the process of performing its operations. Such a method can cause stack overflow if the recursion does not lead to a final result quickly enough. For example, using a recursive method to compute “ $N * (N-1)$  until  $N=2$ ” might not cause a problem for  $N$  less than 25 or 250, but it might cause stack overflow if  $N=25,000$ .

**Self-modifying expressions.** A self-modifying expression changes the value of a variable or object property that is also used elsewhere in the same expression. The danger is the difficulty of predicting the exact order of execution for every syntactic variation of using the method and the changed or changing variable. Parentheses and placement alter the order of execution, and it is safer to avoid self-modifying expressions so as to avoid unexpected results.

## Debugging Method Code

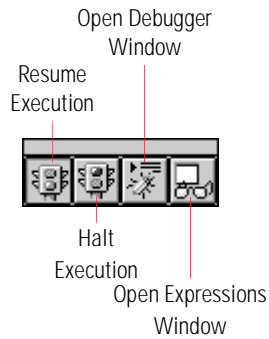
Any code you add to a method may need adjustment. Procedures do not always implement your initial design perfectly in the first few trials.

*Debugging* is that process of analyzing, fixing, and testing your code to make it produce your intended results. When execution runs into problems, the Oracle Power Objects Debugger shows you where the problem arises and enables you to examine values and step through code in order to fix it. The sections that follow explain how these facilities work.

### The Run-Time Debugger

The Debugger lets you debug your Oracle Basic code, interrogate the values assigned to variables and properties, and control the execution of your application during testing.

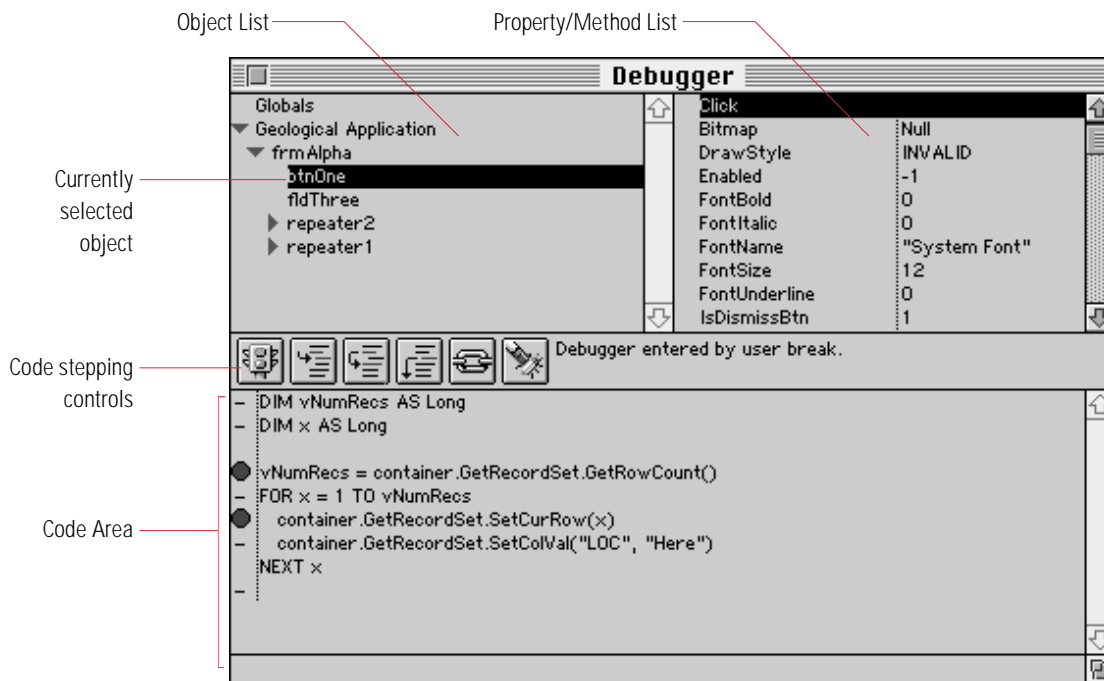
When you run an Oracle Power Objects application, form, or report, the Debugger palette appears:



The debugger palette lets you halt and resume program execution, as well as open the Debugger (Main) and Debugger (Expressions) windows.

### The Debugger (Main) Window

By opening the Debugger (Main) window, you can view all of the properties and methods of objects currently loaded into memory, including the method code added to a method. The Debugger (Main) window components are shown in the following figure.





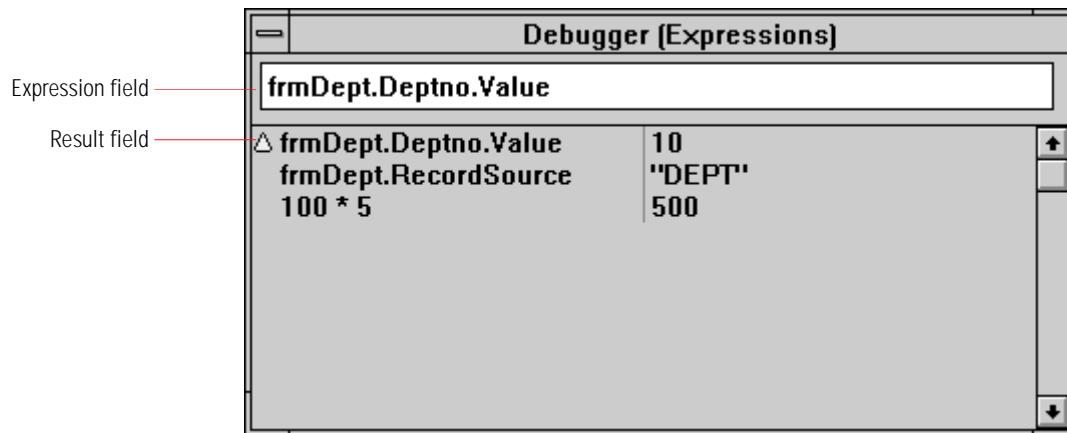
**To view the objects within a container**, click the name of the container from the Object pane of the Debugger (Main) window. The objects within the current application and container appear in a hierarchical list, following the object containment hierarchy.

**To view the properties and methods of an object**, click the name of the object. The complete list of its properties and methods, both standard and user-defined, then appear in the Properties pane of the Debugger (Main) window. A method appears in this list only if you have added method code to it.

**To view the code for a method**, click the method's name in the properties pane: its code then appears below, in the Code pane of the Debugger (Main) window. In this pane of the Debugger (Main) window, you can set breakpoints, step through method code, and move execution to any point within the code, as described in subsequent sections.

### The Debugger (Expressions) Window

The Debugger (Expressions) window lets you interrogate the values assigned to variables and properties, as well as evaluate the result of an expression. This window opens with the Debugger (Main) window if you click the Expressions button; otherwise it opens automatically when a breakpoint is reached during execution.



### Interrogating Values Through the Debugger

To interrogate values through the Debugger (Expressions) window, you enter an expression in the Expression field.

For example, to evaluate the **Enabled** property of a control named Field1, you would enter the following expression in the Expressions window:

```
Field1.Enabled
```

---

After you press Return, the result of the expression appears in the Result field, immediately below the Expression field. Note that the name of the control is needed as a qualifier: `Field1.Enabled`, not simply `Enabled`.

Expressions entered in the Debugger (Expressions) window must follow these rules:

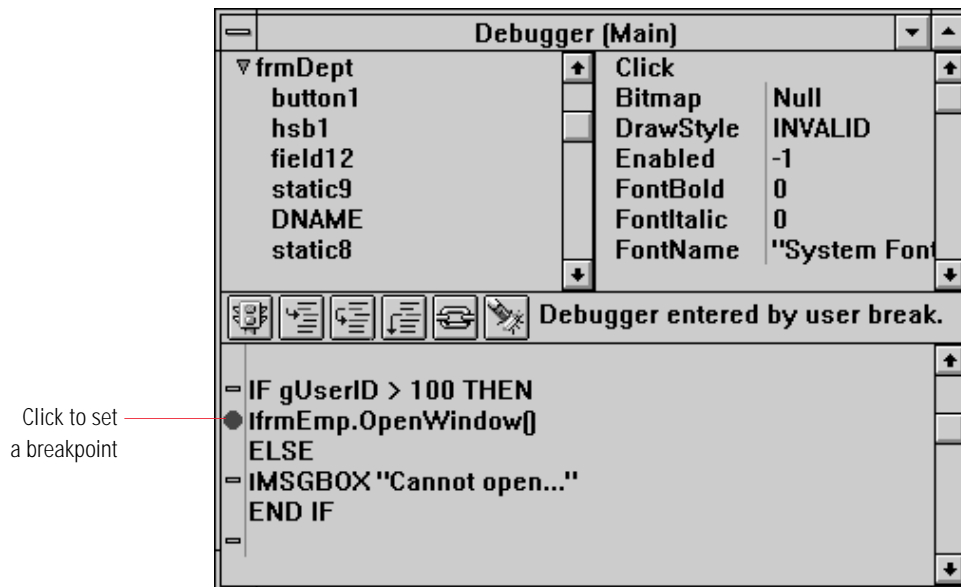
- Expressions must use Oracle Basic operators and functions, with the exception of the EXEC SQL command.
- An expression can be no longer than 1024 characters.

## Setting a Breakpoint

A *breakpoint* identifies where in the sequence of code you want execution to pause so that you can investigate the state your code has reached.

### ☆ To set a breakpoint:

- 1 Open the Debugger (Main) window.
- 2 Select an object, and then select a method you want to debug.  
Its code will appear in the Code List area of the window.
- 3 Click to the left of the line of code at which you want to set the breakpoint.  
A red stop symbol appears to the left of that line of code, indicating that a breakpoint has been set.



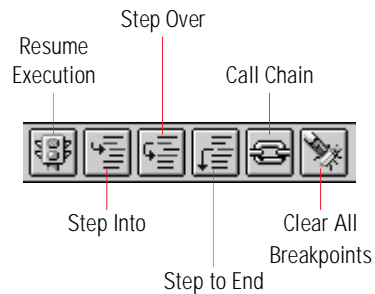


- Resume program execution by pressing the Resume Execution button on the debugger palette.  
The method code will halt execution when it reaches the breakpoint.

Once code has halted execution, you can display values in the Expressions window, as explained above in the section “Interrogating Values Through the Debugger” on page 5.11. If you set watchpoints (explained below), then the current value of each such variable or expression is displayed each time the code stops for a breakpoint.



You can resume execution by pressing the Resume Execution button. You can also step through the method code, using several buttons appearing in the Debugger (Main) window.



## Removing a Breakpoint

You can remove a breakpoint in two ways:

- Click on the red stop symbol to the left of the line of code where the breakpoint has been set.  
The circle then disappears, indicating that the breakpoint has been removed.

-or-



- Press the Clear All Breakpoints button on the Debugger (Main) window.  
As its name implies, this button, when pressed, clears all breakpoints set in the applications method code.

## Setting a Watchpoint

A watchpoint is a variable or expression, which you specify in the Expressions window, whose current value is updated whenever execution is interrupted.

---

☆ **To set a watchpoint:**

- 1 Open the Debugger (Main) window.
- 2 Select an object, and then select a method.
- 3 If desired, set breakpoints as described above.



4 If the Debugger (Expressions) window is not open, click the Expressions button on the Debugger palette.

5 Click in the white Expression field, and enter the expression whose value you want to see during execution of your code.

6 Press the Return key.

The item you entered appears on the list below, followed the current value of the expression. During execution, the expression value will be updated.

7 Position the cursor to the left of the expression until the cursor changes to a triangle, then click the mouse button.

The item is highlighted, and a yellow triangle appears to its left. This sets the watchpoint. The current value of each item you set in this way will be displayed at every breakpoint. Execution will continue when you explicitly instruct it to resume, by clicking the Resume Program Execution button on the Debugger palette.



## Moving Execution to Any Point Within the Method

The buttons labeled above let you move through code one line at a time. However, you can also move execution to any line by dragging the highlight (a blue-green line) from the line currently being executed up or down to the line you want executed next.

This technique lets you move the point of execution within a single method. You cannot move execution to a different method, however.

# 6

---

## Databases

This chapter covers the following topics:

Overview .....	6.2
Database Sessions .....	6.3
Blaze Databases .....	6.20
Oracle7 Servers .....	6.21
SQL Server Databases .....	6.22

---

## Overview

A *database* is a system that stores and organizes information. Oracle Power Objects enables you to build applications that access information stored in a *relational database*, a collection of objects (including tables, views, indexes, sequences, and synonyms) stored together in a file or a series of files and treated as a single logical unit. A *database engine* coordinates user access to the information in the database.

In Oracle Power Objects, you interact with a database through a *database session*, an object that contains the information necessary to establish a connection to the database. This chapter describes database session objects and general techniques for working with them.

This chapter then describes the types of databases supported by Oracle Power Objects: *internal databases* (also called *Blaze databases*) and *external databases*.

*Internal (Blaze) databases* are created and maintained by Oracle Power Objects. Blaze databases are compact and efficient, requiring few system resources to run. The objects in a Blaze database are stored in a single file on your hard disk.

*External databases* are created and maintained by a database engine outside of Oracle Power Objects, such as an Oracle7 Server. External databases can provide advanced security and reliability features for large-scale, mission-critical applications.

This release of Oracle Power Objects supports the following external databases:

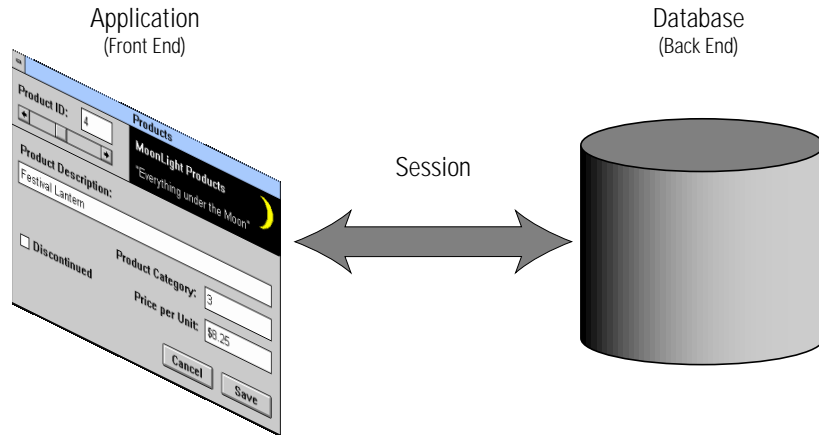
**Oracle7 Servers.** The Oracle7 Server is Oracle's Relational Database Management System (RDBMS) that runs on a wide variety of hardware and software platforms.

**SQL Server Databases.** SQL Server databases are available from Microsoft, Inc. and Sybase, Inc. Oracle Power Objects provides access to SQL Server databases through the DBLIB driver.

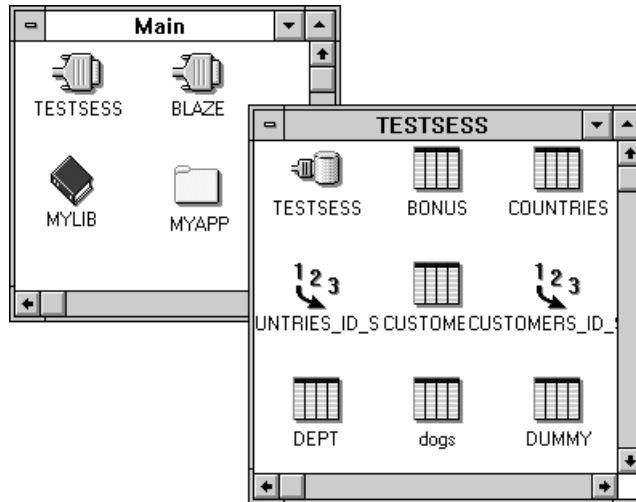
Each type of database is described in a separate section of this chapter.

## Database Sessions

A *database session* object (also called a *session*) is an object representing a connection between your application and a database. The database session provides the communication between the “front end” and “back end” portions of your application, as shown in the following diagram:



Each database session object is stored in its own disk file (this file has the extension .POS in Windows). Database session objects are displayed in the Main window of the Oracle Power Objects desktop, as shown in the following diagram:

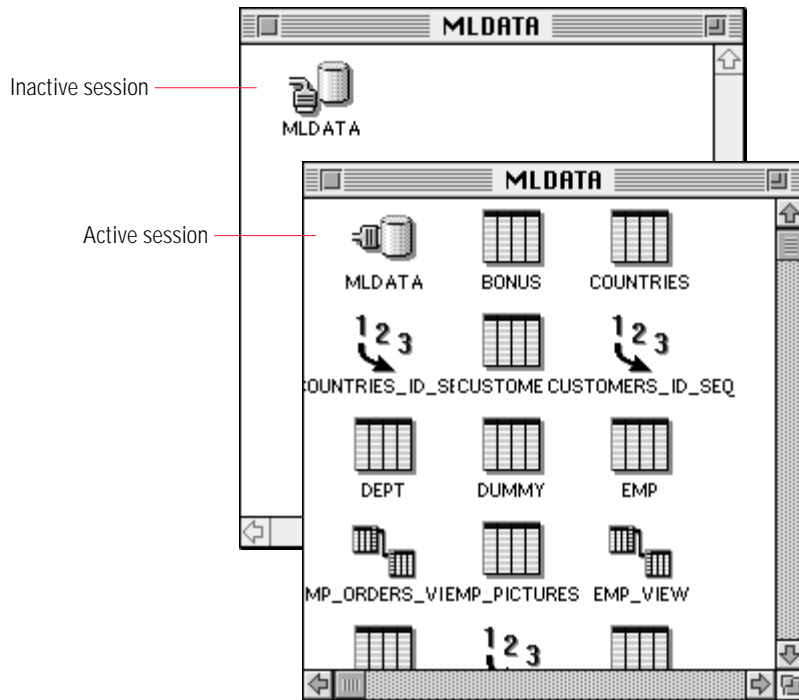


A database session object can be in one of two states: *inactive* or *active*.

An **inactive** session is not currently connected to the database. An inactive session does not display any database objects.

---

An **active session** is currently connected to a database, and “contains” all of the database objects that can be accessed through the connection.



You access a database session object as though it contained database objects. However, the session object does not actually contain your application's data. Instead, it contains information necessary to establish a database connection—for example, it can contain the username, password, and network address of an account on a server database. When your application requests information from the database session, the session automatically sends the appropriate request to the database and processes the result.

Each database session object provides access to a single schema or user account in a database. The session provides full access to the objects and features of the account, although some database features are available only through custom SQL code (for example, you must write custom SQL code to create or access a snapshot object on an Oracle7 Server). At design time, the database session shows icons representing database objects owned by the account. Only the most common types of database objects are represented (tables, views, indexes, and sequences). The session does not show database objects in other schemas or accounts to which the user has access.

Database session objects are independent from application objects. This independence provides flexibility in configuring how your applications connect to databases. For example, multiple applications can use the same database session, and a single application can use many different sessions at once. You can also use the same database session to connect to different databases at



different times—for example, you might want the session to connect to a Blaze database while you are designing your application, and to an Oracle7 Server when your finished application is deployed.

## Creating a Database Session

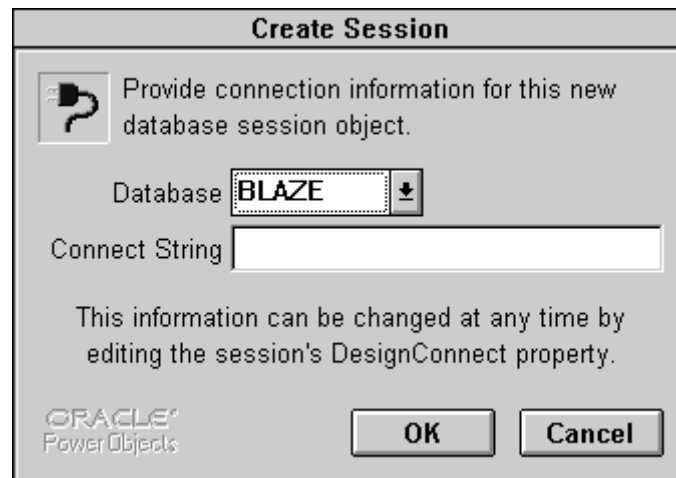
When you create a database session object, you create a separate file on your hard disk containing information on how to connect to a database.

☆ **To create a database session:**



- 1 In the Main window, click the New Session button or choose the **File-New Session...** menu command.

The Create Session dialog box appears, as shown in the following figure:



- 2 From the “Database” popup list, choose the type of database to which the session will be connected.

The database types listed correspond to the database driver files (.POD files) you have installed, described in the section “Database Drivers” on page 6.9.

- 3 In the “Connect String” field, type the connect string for the database to which the session will be connected. Do not enter a database type prefix.

Connect strings are described in the section “Connect Strings” on page 6.6. If you do not know the correct connect string, you can leave this field blank and edit the connection information later.

- 4 Click the **OK** button.

---

The standard file dialog box appears for your operating system.

- 5 Specify the name and location of the session file, then click the **Save** button.

The name must be a valid filename in your operating system. In Windows, the file is automatically given the extension “.POS”.

After you click **Save**, the dialog box disappears and a new session icon appears in the Main window. The Database Session window and Property sheet of the session are automatically opened. The **DesignConnect** property of the session is automatically set to match the values you entered in the “Create Session” dialog box.

- 6 In the Property sheet, modify connection-related properties if necessary.

These properties are described in the section “Connect Strings” on page 6.6.

## Connect Strings

A session establishes a database connection using a *connect string* that identifies the type and location of the database. A connect string has the following form:

`database_type: [username [/password] @] [address]`

`database_type` is the type of database to which you are connecting. Valid

`database_type` values are “blaze”, “oracle”, and “sqlserver”.

`username` is the username for the account. For a local Oracle7 Server, you can also use the keyword “INTERNAL” for username.

`password` is the account password.

`address` is the network address or file location of the database. The format of `address` depends on the type of database to which you are connecting and the network software you are using, as shown in the following table:

Database Type	Address Syntax	Examples
Blaze database	<code>file_path</code>	Blaze database in Windows: <code>C:\DATA\MLDB.BLZ</code> Blaze database on Macintosh: <code>Hard Disk:Data:MLDB.BLZ</code>
Local Oracle7 Server	<code>not required</code>	—
Remote Oracle7 Server using SQL*Net V1	<code>{net_protocol:server[:instance]   server_alias}</code>	Oracle7 Server on Netware: <code>x:Oracle_Server1</code> Oracle7 Server on TCP/IP: <code>t:Oracle_Server2:A</code>

Database Type	Address Syntax	Examples
Remote Oracle7 Server using SQL*Net V2	server_alias	Oracle_Server3
SQL Server database	server:database	Sql_Server1:pubs

For information about the specific syntax to use with your database, contact your system administrator.

### Connect String Examples

The following table shows fully formed connect strings in various configurations:

Example	Description
blaze:C:\DATA\MLDB.BLZ	Connection to default account on a Blaze database file in Windows.
blaze:Hard Disk:Data:MLDB.BLZ	Connection to default account on a Blaze database file on Macintosh.
blaze:dba/dba@C:\DATA\MLDB.BLZ	Connection to "DBA" account on a Blaze database file in Windows.
blaze:dba/dba@Hard Disk:Data:MLDB.BLZ	Connection to "DBA" account on a Blaze database file on Macintosh.
oracle:scott/tiger	Connection to "SCOTT" account on a local Oracle7 database (such as Personal Oracle7).
oracle:scott/tiger@x:Oracle_Server1	Connection to "SCOTT" account on a Netware account using SQL*Net V1.
oracle:scott/tiger@t:Oracle_Server2:A	Connection to "SCOTT" account on a TCP/IP account using SQL*Net V1.
oracle:scott@x:Oracle_Server1	Connection to "SCOTT" account with no password.

---

<b>Example</b>	<b>Description</b>
oracle:scott/tiger@Oracle_Server3	Connection to “SCOTT” account using SQL*Net V2.
sqlserver:Albert/rock@SqlServer1:pubs	Connection to “Albert” account in “pubs” database.
sqlserver:maryd@SqlServer2:pubs	Connection to “maryd” account with no password in “pubs” database.

The session object stores connect string information in three properties: **DesignConnect**, **DesignRunConnect**, and **RunConnect**. Each of these properties is used in different situations, as described below. By entering separate connect strings into each of these properties, you can configure the session to connect automatically to one session while you are developing your application, to another when you are testing your application, and to yet another when your finished application is deployed.

The **DesignConnect** value is used when you activate the session at design time (by double-clicking on the Connector control). When you create a database session, the connection information you enter into the “Create Session” dialog is stored in the **DesignConnect** property.

The **DesignRunConnect** value is used when you are testing your application. This value is used in the following situations:



- When you run a single form by clicking on the Run Form button or choosing the **Run-Run Form** menu command.



- When you run the full application by clicking on the Run Application button or choosing the **Run-Run Application** menu command.

The **RunConnect** property is used when the user runs a compiled application. This value is used in the following situations:

- When the user runs a compiled application with the Oracle Power Objects Run-time application.
- When the user runs a standalone compiled application.

If the **DesignRunConnect** or **RunConnect** property is left empty, the value in the **DesignConnect** property is used in its place.

All of these properties can be modified at design time. However, only the **RunConnect** property can be modified at run time, as described in the next section.

### Prompting the User for Connection Information

For some applications, it is not desirable to have the connection information “hard coded” into the database session object. For example, you might want to allow the user to specify at run time which database account they want to use, or you might not want to hard-code the account password into the session.

You can configure your application to prompt the user for connection information in two ways: automatically or manually.

- You prompt the user *automatically* by setting the **RunConnect** property of the session to a question mark (?). At run time, Oracle Power Objects will automatically display a dialog box prompting the user to enter connection information.



- You prompt the user manually by setting the value of the RunConnect property at run time through Oracle Basic method code. By changing the **RunConnect** value at run time, your application can use connection information specified by the user—for example, you can prompt the user to enter a database type, username and password. An example showing how to create a custom logon dialog box is provided in the section “Example: Creating a Logon Dialog Box” on page 6.15.

### Database Drivers

All information necessary to connect to a particular type of external database is stored in a *driver file* (in Windows, driver files have extension .POD). Driver files must be located in the same directory as the Oracle Power Objects executable (the Designer or Run-time application).

Driver files provide modular access to external databases. You can add support for other types of external databases simply by installing the appropriate driver file, along with any special networking software required by the database.

---

No driver file is required for Blaze databases. Blaze database support is built directly into the Oracle Power Objects Designer and Run-time applications.

➤ **Note:** When you generate standalone executable applications, the necessary driver files are automatically included in the executable file. Client users are not required to have the .POD file installed on their system, although they must have any required networking software.

## Activating and Deactivating a Database Session

In order to establish a database connection, you must *activate* a session; to disconnect from the database, you must *deactivate* the session. You can activate and deactivate sessions both at design time and at run time; however, you use different techniques to perform the task.

### Activating a Session at Design Time

You activate a session at design time in order to see and work with icons representing database objects. For example, you can double-click on the icon for a table or view to open the Table Editor or View Editor window. You can also use the icons in a session to bind a container or control to a database object, as described in Chapter 17, “Binding a Container to a Record Source”.

When you activate a session at design time, the value in the **DesignConnect** property of the session object is used to connect to the database.

#### ☆ To activate a database session at design time:

- 1 In the Main window, open the database session object by double-clicking on its icon.

The Database Session window opens, along with the session’s Property sheet.



- 2 Double-click the Connector control to activate the session.

When the session becomes active, icons representing database objects appear in the Database Session window. The Connector control changes to the “active” state.



### Activating a Session at Run Time

You must activate session objects at run time so that your application’s containers and controls can display values from the database. When you activate a session at run time, the value from the **DesignRunConnect** or **RunConnect** property of the session is used as the database connect string—the **DesignRunConnect** value is used at design run time, the **RunConnect** value at standalone run time.

Session objects are activated at run time either *automatically* or *manually*.

For information about the Table Editor and View Editor windows, see Chapter 8, “Database Objects”.

You configure a session to activate automatically at run time by setting the **ConnectType** property of the session object. The **ConnectType** property can have the following settings:

**Connect on Startup.** The session is activated automatically as soon as the application begins execution. This is the default setting. You should use this setting when your application needs to access the database immediately upon startup.

**Connect on Demand.** The session is activated automatically the first time it is referenced, either by a bound object or by a custom SQL statement executed through EXEC SQL or SQLLOOKUP. You should use this setting for sessions that do not need to be activated immediately upon startup—for example, when the database connection might not be required by the user.

**Connect Manually.** The session must be activated manually by executing the **Connect()** method of the session, as described below. You should use this setting when you need explicit control over when the connection is established—for example, when you need to prompt the user to enter connection information, or when you have already deactivated the session by executing the **Disconnect()** method.

To activate a session manually, you execute the **Connect()** method of the session object. For example, the following method code activates the session “sesScott”:

```
sesScott.Connect()
```

For information about referring to database sessions in method code, see the section “Using a Database Session Object in Oracle Basic” on page 6.13.

## Deactivating Sessions

In many cases, you do not have to deactivate session objects explicitly—all sessions are deactivated automatically when you quit the Oracle Power Objects Designer or Run-time application. Sessions are also deactivated automatically when you remove the database session icon from the Main window.

### Deactivating a Session at Design Time

You deactivate a session at design time when you are finished working with the database, or if you need to effect changes to the session object’s definition. For example, if you change the value in the **DesignConnect** property of the session, you must deactivate and then reactivate the session to use the updated connection information.

---

☆ **To deactivate a database session at design time:**



- 1 In the Main window, double-click the active Connector control.



When the session becomes inactive, the database object icons disappear from the Database Session window. The Connector control changes to the “inactive” state.

### Deactivating a Session at Run Time

You deactivate a session at run time when your application needs to disconnect from the database—for example, when an application has finished downloading information from a network server.

To deactivate a session manually at run time, you execute the **Disconnect()** method of the session object. For example, the following method code deactivates the session “sesScott”:

```
sesScott.Disconnect()
```

Before you call **Disconnect()**, you must close any forms or reports that are bound to the session, or that contain objects that are bound to the session (such as a repeater display). The **Disconnect()** method fails if you attempt to execute it when a form or report bound to the session is still open.

The **Disconnect()** method returns a value indicating whether it successfully deactivated the session object. **Disconnect()** returns 0 if the session was deactivated; it returns -1 if the session was not deactivated.

### The Default Session

The *default session* is the session object used for database access when a session is not explicitly specified. Each Oracle Power Objects application can have its own default session. You specify the default session by setting the **DefaultSession** property of the application to the name of the session object.

The default session is used in the following cases:

- For bound containers whose **RecSrcSession** property is empty.
- For EXEC SQL statements that do not include an AT clause.
- For SQLLOOKUP functions that do not include the optional first parameter, which specifies the session object used by the function.
- For translation lists that do not include an AT clause in the **Translation** or **ValueList** property.

Designating a default session is optional. If your application does not have a default session, you must explicitly specify a session name for all database access.

Once you have designated a default session, it is simple to switch the session your application uses for database access. To use a different session, simply change the **DefaultSession** property of the application to the name of the new session.

For more information about referring to database sessions in method code, see the section “Using a Database Session Object in Oracle Basic” on page 6.13.



## Using Sessions in an Application

Although session objects are separate from application objects, your database application will contain references to session objects everywhere database access is required. This section describes different ways you can use session objects in your application.

### Using a Database Session Object in a Bindable Container

You use database session objects when binding a container (such as a form or report) to a table or view in a database. As described in Chapter 17, “Binding a Container to a Record Source”, you can use the icons in the database session window to bind objects graphically.

When you bind a container to a table or view, you specify the name of the database session object that contains the table or view in the **RecSrcSession** property of the bindable container. If you do not specify a session in the **RecSrcSession** property, the table or view is assumed to be associated with the default session.

### Using a Database Session Object in Oracle Basic

You can control the behavior of session objects explicitly through Oracle Basic method code. For example, you can execute methods of the session object to activate or deactivate it, or to commit or roll back transactions associated with the session.

You can refer to a database session object in two ways:

- By supplying the **Name** property of the database session object.
- By executing the **GetSession()** method of a recordset object, which returns a reference to the associated database session object.

For example, you could issue the following statements to get a handle to the database session associated with the form “Form1”:

```
DIM recSetObj, sesObj AS Object
recSetObj = Form1.GetRecordSet()
sesObj = recSetObj.GetSession()
```

You can use either type of reference to examine and set properties and methods of a session object. For a list of session properties and methods, see the section “Properties and Methods of Database Session Objects” on page 6.19.

When you execute an EXEC SQL statement, you can indicate the session to which the statement should be sent in the optional AT clause of the statement. The AT clause can contain either the **Name** property of the session object or a variable containing a reference to the session object (the variable name is preceded with a colon). For example, the following method code directs an EXEC SQL statement to the session “Session1”:

```
EXEC SQL AT Session1 DROP TABLE temp
```

For more information about the EXEC SQL command, see the section “The EXEC SQL Command” on page 9.15.

---

The following method code uses a variable in the AT clause of an EXEC SQL statement to specify the session:

```
DIM recSetObj, sesObj AS Object
recSetObj = Form1.GetRecordSet()
sesObj = recSetObj.GetSession()
EXEC SQL AT :sesObj DROP TABLE temp
```

For more information about the SQL-LOOKUP function, see the section "The SQL-LOOKUP Function" on page 9.21.

Similarly, when you execute a SQLLOOKUP function, you can indicate the session to which the SELECT statement should be sent in the optional first argument to the function. As with the AT clause, the argument can be specified either as the **Name** property of the session object or as a variable. For example, the following method code directs a SQLLOOKUP function to the session "Session1":

```
DIM empName AS String
empName = SQLLOOKUP(Session1, "SELECT ename FROM emp" &
    "WHERE sal = (SELECT MAX(sal) FROM emp)")
```

For more information about translation lists, see the section "List Controls" on page 10.15.

You can also specify a session object explicitly when setting up a *translation list* (a list box, popup list, or combo box that "looks up" its values from a detail table). You specify the session object with an AT clause in the **TranslationList** property of a list box or popup list, or in the **ValueList** property of a combo box. The syntax of the AT clause is identical to that used with EXEC SQL. For example, the following **TranslationList** value specifies that its values should be derived from the session "Session1":

```
AT Session1 SELECT ename, empno FROM emp WHERE" &
    "job = 'SALESMAN'
```

## Tips and Techniques

By using session objects carefully, you can give your application a great deal of control over database connections. This section describes techniques you can use to optimize how your application uses sessions.

**Use separate sessions for separate transactions.** If your application allows independent transactions to be used simultaneously, use a separate session object for each transaction. Doing so allows each transaction to be committed separately, rather than requiring all transactions to be committed together. For example, if your application includes a form that displays employee information and a form that displays customer information, you might want to use a separate session for each form. Because these two forms display unrelated information, they do not need to be committed or rolled back together.

**Use a separate session for lookup tables.** Queries against locally stored tables are often faster than queries transferred over a network, so you can often enhance the performance of your application by storing tables that do not change much (such as lookup tables) in a local Blaze database. You can then create a separate session to use when looking up information from these tables. For example, you might store a detail table containing the names of countries or states in a Blaze database and provide a separate session for that database.

**Activate sessions only when needed.** If your application contains references to sessions that might not be required by the user, you can make your application's startup faster by setting the **ConnectType** property of those sessions to "Connect on Demand" or "Connect Manually".

### Example: Creating a Logon Dialog Box

This section describes how to create a custom logon dialog box that allows the user to specify connection information when your application starts up. You can customize the code in this example for use in your own applications.

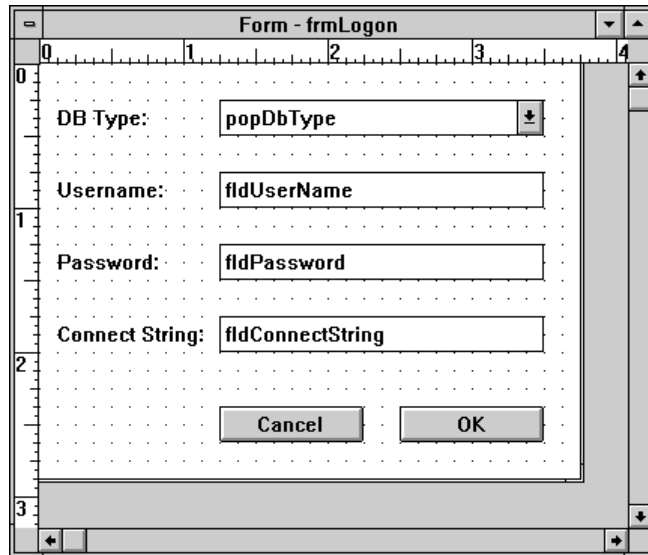
This example provides functionality similar to that provided by setting the **RunConnect** property of your database session object to a question mark (?), as described in the section "Prompting the User for Connection Information" on page 6.9. However, by creating your own logon dialog box, you can customize the appearance and behavior of your logon screen.

This example displays a logon dialog box when the application is first launched, as shown in the following figure:

The image shows a standard Windows-style dialog box for database logon. It has a title bar (not visible) and a main area with four labeled input fields. The 'DB Type' field is a dropdown menu currently set to 'Oracle'. The 'Username' field contains 'Scott', the 'Password' field contains 'Tiger', and the 'Connect String' field contains 'x:oraserv1'. At the bottom of the dialog are two buttons: 'Cancel' on the left and 'OK' on the right.

This example requires a session object called "Session1". You must set the **ConnectType** property of the session to "Connect on Demand".

The example also requires a form called "frmLogon", which contains the controls shown in the following diagram:



You must set the following object properties:

Object	Property	Value
frmLogon	WindowState	Standard Dialog
popDbType	Datatype	Long
popDbType	Translation	"Blaze" = 1 "Oracle" = 2 "SQL Server" = 3
fldUserName	Datatype	String
fldPassword	Datatype	String
fldAddress	Datatype	String

The following method code appears in the (Declarations) section of the application:

```
(Declarations)
CONST DBTYPE_BLAZE = 1
CONST DBTYPE_ORACLE = 2
CONST DBTYPE_SQLSERVER = 3
CONST BTN_OK = 1
CONST BTN_CANCEL = 2
```

The following method code appears in the **Initialize()** method of the application:

```

Sub Initialize()
DIM sesTheSession AS Object      'Reference to session object
DIM vReturnObj AS Object        'Return value of OpenModal
DIM vConnectString AS String    'User-defined connect string
DIM vReturnVal AS Long          'Return value of MSGBOX
DIM vQuit AS Long               'Did the user click Cancel?

'Get a reference to the session to which we will connect
'Here we use the session "Session1"
'You can substitute the name of a different session object
sesTheSession = Session1

vQuit = FALSE

'Repeat until the connection is established or the user
cancels
DO WHILE sesTheSession.IsConnected() = FALSE AND &
    vQuit = FALSE

'Display the logon dialog form and store the return value
'(the button clicked by the user) in vReturnObj
    vReturnObj = frmLogon.OpenModal(false)

'If the user clicked the OK button, build the connect
'string using the specified information
    IF vReturnObj = frmLogon.btnOK THEN

'Add the database type prefix
        SELECT CASE frmLogon.popDbType.Value
            CASE DBTYPE_BLAZE
                vConnectString = "Blaze:"
            CASE DBTYPE_ORACLE
                vConnectString = "Oracle:"
            CASE DBTYPE_SQLSERVER
                vConnectString = "SQLServer:"
        END SELECT

'Add the username, password, and database address
        vConnectString = vConnectString & &
            frmLogon.fldUserName.value & &
            "/" & frmLogon.fldPassword.value & "@" & &
            frmLogon.fldAddress.value
    
```

---

```

'Set the RunConnect property of the session
'to the connect string we have constructed
    sesTheSession.RunConnect = vConnectString

'Activate the session
    sesTheSession.Connect()

'If there was a problem connecting, let the user know
'and allow them to try again or cancel.
    IF NOT sesTheSession.IsConnected() THEN
        vReturnVal = MSGBOX("There was a problem connecting. " &
&
            "Do you want to try again?", 49)
        IF vReturnVal = BTN_CANCEL THEN vQuit = TRUE
    END IF

'If the user clicked the Cancel button in the logon dialog
'box instead of OK, quit the application
    ELSE
        vQuit = TRUE
    END IF

'If the session is not yet active, redisplay the logon
'dialog box
LOOP

'If the user cancelled, quit the application
IF vQuit = TRUE THEN
    Application.CloseApp()

'If the session was successfully activated,
'we can open other forms--here we open "frmEmployees".
'You can substitute the name of a different form.
ELSE
    frmEmployees.OpenWindow()
END IF

```

## Properties and Methods of Database Session Objects

This section lists the properties and methods associated with Database Session objects.

### Properties of a Database Session Object

A database session object has the following properties:

Property	Description
<b>ConnectType</b>	Determines how the session becomes active (automatically or manually).
<b>DesignConnect</b>	Connect string used when session is activated by double-clicking on the Connector control in the database session window.
<b>DesignRunConnect</b>	Connect string used when session is activated from the design run-time environment.
<b>Name</b>	Name of the database session object.
<b>RunConnect</b>	Connect string used when session is activated from the standalone run-time environment.

### Methods of a Database Session Object

A database session object has the following methods:

Method	Description
<b>CommitWork()</b>	Commits all pending transactions associated with the session, first flushing any recordset changes that have not yet been sent to the database.
<b>Connect()</b>	Connects to the database and activates the session.
<b>Disconnect()</b>	Disconnects from the database and deactivates the session.
<b>IsConnected()</b>	Indicates whether the session is currently active (not visible in Property sheet).
<b>IsWorkPending()</b>	Indicates whether the Record Manager has stored changes to a recordset that have not yet been flushed to the database (not visible in Property sheet).
<b>RollbackWork()</b>	Rolls back all pending transactions (that is, discards all uncommitted changes) associated with the session.

---

## Blaze Databases

A *Blaze database* is a compact and efficient relational database that is created and maintained by Oracle Power Objects. Oracle Power Objects includes all of the tools you need to create and work with Blaze databases.

Blaze databases support many of the features and capabilities of larger relational databases, but require significantly fewer system resources to run. They can contain all of the basic database objects: tables, views, indexes, sequences, and synonyms. As with most relational databases, they are accessed through the SQL relational querying and programming language. The Blaze-supported SQL language is a subset of the Oracle7 SQL language with a few added features. It is described in the topic “SQL Language Reference” in the online help.

Each Blaze database is stored as a single disk file in your operating system (which has the extension .BLZ in Windows). This file contains all user objects and data, as well as information necessary for client applications to connect to and disconnect from the database. Each Blaze database can have only one connection open to it at any time.

Unlike some other relational databases, a Blaze database has no server-based dedicated memory structures or processes. All information about the database, including data locks and transaction control information, is stored in the database file. All necessary in-memory operations (such as interpreting SQL statements and reading or writing data) are performed by the *Blaze database engine*, which is built into the Oracle Power Objects Designer and Oracle Power Objects Run-time applications. When two different client applications connect to the same Blaze database, each client uses its own copy of the database engine.

### When Should I Use a Blaze Database?

Blaze databases are ideal for light to moderate data access situations, including the following uses:

- **Local lookup tables.** If your application references external tables or views that remain relatively constant, such as a list of departments or product categories, you can improve performance by storing these tables locally in a Blaze database. Frequently updated tables can also be stored locally, but you must then be careful to synchronize the contents of separate databases.
- **Prototyping.** While developing an application that is designed to be run against an external database, you can copy the definitions of the relevant database objects to a Blaze database for testing purposes. This can ease the burden on shared resources, improve application performance during development, and allow you to work while disconnected from your network.
- **Small application deployment.** For relatively small (under 4 GB), non-mission-critical applications, a Blaze database can provide a compact, easily maintained data store that supports access by one user at a time.
- **Disconnected client access.** You can use a Blaze database to store information downloaded from a large database system, which the user can then examine and modify off-line.



- **Read-only databases.** A Blaze database can provide a structure for information on a read-only device, such as a CD-ROM.

## When Should I Use an External Database?

The following data access situations require the additional capabilities of an external database:

- **Large data storage requirements.** Blaze databases are limited to the maximum available file size on the host operating system. For Macintosh and Windows systems, the maximum file size is 4 GB.
- **Concurrent access.** External databases are designed to handle efficiently large numbers of concurrent sessions.
- **Mission-critical applications.** External databases provide advanced database backup and restoration features that are vital for mission-critical applications.
- **Strict security requirements.** Though Blaze databases support password protection and data encryption, they cannot offer the same level of protection as a secure database located on a secure operating system. External databases also offer additional security and administration features such as audits, roles, and protected schemas.
- **Access from other platforms and tools.** Currently, you can access a Blaze database only from Oracle Power Objects. If you need to use other database access tools, or you need access to the database from an operating system platform not supported by Oracle Power Objects, you should use an external database.

## Oracle7 Servers

The *Oracle7 Server* is a Relational Database Management System (RDBMS) available on a large number of operating system platforms. Oracle7 Servers provide efficient and effective solutions for the major features required of a database, including:

- Large databases and space management control
- Many concurrent database users
- High transaction processing performance
- High availability
- Industry accepted standards
- Manageable security
- Database enforced integrity
- Client/server (distributed processing) environments
- Distributed database systems
- Portability
- Compatibility
- Connectability

---

Oracle Power Objects provides full support for the features and capabilities of Oracle7 Servers. However, Oracle Power Objects does not provide tools to create or administrate Oracle7 Servers—you must purchase and install the Oracle7 Server software separately.

As with most relational databases, Oracle7 Servers are accessed through the SQL relational querying and programming language. You can access many features of the Oracle7 Server through the Oracle Power Objects Record Manager and the properties, methods, and windows associated with database access. You can access other Oracle7 Server features by executing custom SQL or PL/SQL statements using the EXEC SQL command.

## Oracle7 Documentation

For a full description of the features and capabilities of Oracle7 Servers, refer to the Oracle7 Server Documentation set. The following manuals in particular are recommended:

For general information about the Oracle7 Server and how it works, see the *Oracle7 Server Concepts Manual*.

For reference information about the SQL commands and functions supported by the Oracle7 Server, see the *Oracle7 Server SQL Language Reference Manual*.

For information about developing database applications within the Oracle7 Server, see the *Oracle7 Server Application Developer's Manual*.

For information about administering the Oracle7 Server, see the *Oracle7 Server Administrator's Guide*.

## SQL Server Databases

A *SQL Server* database is a multiuser Relational Database Management System (RDBMS) available from Microsoft, Inc. and Sybase, Inc. SQL Server is available on a wide range of platforms, although the list of platforms and versions supported varies depending on the database provider.

➤ **Note:** Certain configurations of Oracle Power Objects might not include support for SQL Server databases. To see whether your version of Oracle Power Objects includes SQL Server support, see the release notes accompanying the product.

SQL Server provides support for major features required of a database. For a full list of the features supported by your version of SQL Server, see the accompanying documentation.

A typical SQL Server installation includes set of *system databases* and *user databases*. *System databases* include the “master”, “model”, and “tempdb” databases. *User databases* are created and maintained as needed by the SQL Server system administrator.

This manual uses the term “database” or “server” to refer to a specific installation of SQL Server, including all databases within that installation.

## Supported SQL Server Databases

Oracle Power Objects currently supports any SQL Server database that can be accessed through the DBLIB library. This includes SQL Server databases available from Microsoft, Inc. and Sybase, Inc.

Oracle Power Objects allows you to use all database features supported through the DBLIB driver. Certain features of Sybase System 10 databases are not available through the DBLIB driver, including support for cursors.

For a complete list of supported SQL Server databases and features, see the Release Notes accompanying your version of Oracle Power Objects.

## Defining Primary Keys

You should always define Primary Key constraints for tables to be used with Oracle Power Objects. Oracle Power Objects uses primary key values to identify individual rows for database operations—for example, when updating or deleting a specific row. Your application may behave unexpectedly if it uses tables that do not include a Primary Key.

## Incremental Fetching

The DBLIB driver does not include support for multiple simultaneous cursors. Therefore, only one query result set can be processed at a time—all result rows from a query must be returned before a second query can be executed.

As described in the section “Fetching Rows from the Database” on page 17.14, you can control how result rows are fetched from the database by setting the **RowFetchMode** property of a bound container. When the **RowFetchMode** property is set to “Fetch All Immediately”, your application will behave the same way against all types of databases. However, when the **RowFetchMode** property is set to “Fetch as Needed” or “Fetch Count First”, your application might run more slowly against SQL Server databases. This is because all unfetched rows from a query must be fetched from the database before any additional queries can be executed.

## Bind Variables

The DBLIB driver does not include support for bind variables. To provide bind variable support, Oracle Power Objects automatically replaces bind variable references in EXEC SQL statements with literal data values.

For information about bind variables, see the section “Using Bind Variables” on page 9.16.

---

### **Generating Unique Table Values**

For information about generating unique values, see the section "Counter Fields" on page 19.17

SQL Server databases do not support sequence objects. Therefore, you must use an alternative technique to generate unique table values (for example, values for a Primary Key column in a table).

# 7

---

## Blaze Databases

This chapter covers the following topics:

Overview .....	7.2
Creating a Blaze Database .....	7.2
Schemas in a Blaze Database .....	7.3
Data Dictionary .....	7.4
SQL Language .....	7.5
Blaze Database Files .....	7.5
Sessions .....	7.7
Specifications .....	7.8

---

## Overview

A Blaze database is a compact and efficient relational database that is created and maintained by Oracle Power Objects. Oracle Power Objects includes all of the tools you need to work with Blaze databases.

This chapter describes how to create Blaze databases. It also includes technical details about the structure of a Blaze database.

## Types of Blaze Databases

Every Blaze database is either *read-write* or *read-only*.

**Read-write databases** can be both examined and modified by clients. Blaze databases are read-write by default, as this is the most common configuration.

**Read-only databases** can only be read by clients; they cannot be altered. You might choose to make a database read-only if it will never be altered (for example, because it will be distributed on a CD-ROM, or because it maintains company information that does not change). Access to a read-only database can be faster because no data locking is required.

In this version of Oracle Power Objects, you can create only read-write databases. However, in a future version of Oracle Power Objects, you will be able to convert a read-write database into a read-only database.

## Creating a Blaze Database

You create a Blaze database by creating a Blaze database file, which contains all of the object definitions and data in the database. In Windows, this file has the extension “.BLZ”. Blaze database files are described in the section “Blaze Database Files” on page 7.5.

☆ **To create a Blaze database:**

- 1 In the Main window, choose the **File-New Blaze Database...** menu command.

The standard Create File dialog box for your operating system appears.

- 2 Enter the name and location for your Blaze database file.

In Windows, this file is automatically given the extension “.BLZ”.

- 3 Click the **Save** button or press Return.

A Blaze database file is created with the name and location you specify.

Once you have created a Blaze database, you must create a Database Session object to connect to the database. Creating a Database Session object is described in the section “Creating a Database Session” on page 6.5.

## Schemas in a Blaze Database

A *schema* is a logical grouping of database objects within a database. Each database user (account) in the database has a schema, which must have the same name as the user. When you connect to an account on a database, you normally see and have access to the objects in the account's schema.

Blaze databases support multiple schemas: you can create, modify, and delete any number of separate user accounts in the database. However, you do not need to create or manage any schemas to use a Blaze database. If you do not specify a user name when you connect to a Blaze database, you are automatically connected to the default schema (called the “DBA” schema).

Schemas in Blaze databases are *unprotected*: any user in the database can see and modify the objects in any other schema. Therefore, Blaze databases do not require SQL commands that grant or revoke object privileges.

When a Blaze database is created, it has the following default schemas:

Schema	Description
DBA	Default location for user-created database objects. If a user name is not specified when connecting to a Blaze database, the user is connected to the DBA schema.
INDEXES	Stores database objects used by the database engine to enforce some types of constraints (such as Unique constraints and Primary Key constraints). Only indexes created automatically by the database engine are stored in the INDEXES schema; indexes created or named explicitly by the user are stored in the schema where the index object is created.
SYS	Stores data dictionary tables and data dictionary views that describe the structure of the database. Database users should not alter any object contained in the SYS schema, as doing so could permanently damage the database.
TEMP	Used by the database engine for creation of temporary database objects, such as temporary tables used while sorting data.

You can create and delete schemas in a Blaze database by issuing SQL commands, as described in the following table:

Command	Description
CREATE USER	Creates a database user and a schema for the user.
CREATE SCHEMA	Creates a set of database objects within a schema in a single transaction.

---

<b>Command</b>	<b>Description</b>
DROP USER	Deletes a database user and the user's schema, along with all objects contained within the schema.

For more information about these SQL commands, see the SQL Language Reference in the online help.

## Data Dictionary

Relational databases often contain a *data dictionary*, a set of read-only tables and views that provide information about the database.

The tables in a data dictionary usually can be modified only by the database engine itself. Users have access to read-only views on these tables. These data dictionary views also present the information from the data dictionary tables in a more readable and useful format.

The Blaze database engine provides the following data dictionary views:

<b>View</b>	<b>Description</b>
ALL_COLUMNS	Lists all columns of all tables in the database
ALL_CONSTRAINTS	Lists all table and column constraints in the database
ALL_IND_COLUMNS	Lists all indexed table columns in the database
ALL_INDEXES	Lists all indexes in the database
ALL_OBJECTS	Lists all objects (tables, views, indexes, sequences, and synonyms) in the database
ALL_SEQUENCES	Lists all sequences in the database
ALL_SYNONYMS	Lists all synonyms in the database
ALL_TABLES	Lists all tables in the database
ALL_USERS	Lists all users (schemas) in the database
ALL_VIEWS	Lists all views in the database
CAT	Synonym for ALL_OBJECTS
DUAL	Contains one column and one row; used to guarantee a known result
USER_SEQUENCES	Lists all sequences owned by the user
USER_SYNONYMS	Lists all synonyms owned by the user
USER_TABLES	Lists all tables owned by the user



View	Description
USER_VIEWS	Lists all views owned by the user
USER_COLUMNS	Lists all columns of all tables owned by the user
USER_INDEXES	Lists all indexes owned by the user
VERSION	Version of Blaze database engine that created the database file

➤ **Technical Note:** Blaze data dictionary tables and views are owned by the SYS schema.

You can use the Blaze data dictionary tables to describe a table by issuing a SELECT statement like the following one:

```
DIM vTableName AS String
DIM vColNames AS String
DIM vColTypes AS String
vTableName = "EMP"
EXEC SQL SELECT colname, ALL_COLUMNS.type &
      into :vColNames, :vColTypes &
      from ALL_TABLES, ALL_COLUMNS &
      WHERE UPPER(ALL_TABLES.name) = UPPER(:vTableName) &
      AND ALL_TABLES.objID = ALL_COLUMNS.objID
```

## SQL Language

As with most relational databases, Blaze databases are accessed through the SQL relational querying and programming language. The Blaze-supported SQL language is a subset of the Oracle7 SQL language with a few added features. It is described in the topic "SQL Language Reference" in the online help.

## Blaze Database Files

A Blaze database consists of a single binary disk file. This file contains all user objects and data, as well as information necessary for client applications to connect to and disconnect from the database. Each Blaze database can have only one connection open to it at any time.

Unlike some other relational databases (such as Oracle7 Servers), a Blaze database has no dedicated memory structures or processes. All information about the database, including data locks and transaction control information, is stored in the database file. Any necessary in-memory operations (such as interpreting SQL statements and reading or writing data) are performed by the *Blaze database engine*, which is built into the Oracle Power Objects Designer and Oracle Power Objects Run-Time applications. Each client application that connects to a Blaze database uses its own copy of the database engine.

---

The user data stored in the database has a logical structure that is distinct from the physical structure of the database file. The logical structure represents the database as a collection of objects: schemas, tables, views, indexes, sequences, and synonyms. When you use a Blaze database, you interact only with the logical structure of the database. The Blaze database engine provides the appropriate mapping between the logical and physical structures of the database.

## Structure of a Database File

A Blaze database file consists of two sections: a *header section* and a *data section*.

The *header section* is defined when the database file is created and remains fixed in size. The header section contains parameter information about the database, areas to store session-related information, and encryption information. Some portions of the header section are modified as the database changes; other portions do not change after creation.

The *data section* follows the header section and contains all of the user objects and data in the database. The data section grows automatically as more storage space is required, until the file reaches the capacity of its storage device or the maximum database size (2 PB). Unused space in the data section can be reclaimed, but the database file never shrinks.

The basic unit of storage in a Blaze database file is the *block*. Information in both the header section and the data section is stored in blocks. The block size of the database is specified when the database is created, and remains fixed for the lifetime of the database. The block size is always a multiple of the *sector size*, the smallest unit of information that can be read from or written to the disk on which the database is stored.

### The Header Section

The header section of the database file contains the following structures:

*Parameter blocks* - contain general information about the database file. The information in the parameter blocks is determined when the database file is initialized and the information does not change after that. This information includes:

- a unique database identifier
- the version number of the database
- size of logical sectors in the database
- size of blocks in the database
- maximum number of sessions allowed
- storage and space allocation parameters

*Transaction control area* - stores database information that is updated regularly. For example, the transaction control area contains information used to allocate object ID numbers.

*Session File Areas (SFAs)* - maintain information about database sessions. Sessions are described in the section “Sessions” on page 7.7.

*Lock areas* - synchronize multisession access to the database file. Instead of reading and writing information directly to the lock areas, clients record their locks by acquiring file system locks on specific bytes of the database file. This system ensures that abnormally disconnecting clients do not permanently tie up database resources.

### The Data Section

The data section of the database file stores the definitions and data of all *schema objects* (tables, views, indexes, sequences, and synonyms) within the database. A *schema* is a named collection of database objects.

Most user data in a Blaze database is stored in *tables*. A table is an unordered set of data rows within the database file. Each row in the database is identified by a unique ROWID, which is used internally for all access to the row's data. This ROWID is constant for the lifetime of the row, although it can be reused if the row is deleted.

Each row is logically subdivided into one or more *columns*, each of which contains a single piece of data. A Blaze database row can have a maximum of 8192 columns.

## Sessions

All interaction with a Blaze database takes place through a *session*, which represents a conversation between a client process and the database. A session corresponds to an area in the database file, which keeps track of the session's current state. Only one session can be open at any time.

➤ **Note:** Sessions within a Blaze database are not the same as database session objects. A session within a Blaze database is located on the server (inside the database), while a database session object is located on the client (your database application).

Blaze databases support two types of sessions: *read-write sessions* and *read-only sessions*.

### Read-Write Sessions

Normally, a Blaze database session is a *read-write session*, in which the client can both read and modify the database file. A read-write session is always in one of three states: live, dead, or available.

- A session becomes *live* when a client process connects to the database. A live session is indicated by a file lock on a byte corresponding to the Session ID.
- A session is *dead* when the process that was using it last disconnected abnormally. A dead session can be detected by the fact that its file system lock has cleared. Dead sessions are automatically “cleaned up” when necessary (for example, the session's current transaction is rolled back). This converts the dead session to an available session.
- A session is *available* when the process that was using it last disconnected normally. Available sessions can be used immediately by another client process.

---

Information about each session's state is maintained in a *Session File Area*, or *SFA*. The SFA performs a role similar to the System Global Area (SGA) in an Oracle7 Server. SFAs are stored in dedicated regions of the header section of the database file.

## Read-Only Sessions

In addition to read-write sessions, a Blaze database also supports *read-only* sessions, also called *viewer sessions*. A read-only session can be initiated by a client that does not have write access to the disk on which the database is stored. Read-only sessions have only two states: available or live.

A read-only session differs from a connection to a read-only database. A read-only database cannot be modified, so the database does not store object locking information. Although a read-only session is also incapable of modifying the database, locking information must be stored to ensure data consistency.

Read-only sessions maintain read consistency by placing file system locks on areas of the database file. Because file systems allow only a limited number of file system locks, read-only sessions are limited to 10 concurrent object locks, which limits the number of objects that can be referenced in a SQL statement.

## Specifications

This section describes numeric parameters and limits associated with Blaze databases.

<b>Parameter</b>	<b>Value</b>
Maximum concurrent sessions:	1
Maximum concurrent object locks for a read-only session:	10
Size of database sector:	512 bytes
Size of database block:	4 sectors (2 KB)
Maximum size of database file:	2 PB (or limit imposed by operating system)
Maximum columns in a table:	8192
Maximum size of a short column:	32 KB
Maximum size of a long column:	4 GB
Maximum total size of all short columns in a table:	1 MB



---

## Database Objects

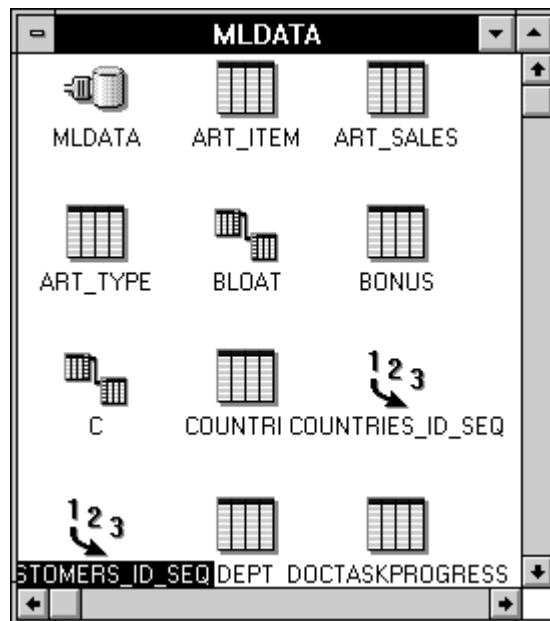
This chapter covers the following topics:

Overview .....	8.2
Tables .....	8.6
Views .....	8.16
Indexes .....	8.24
Sequences .....	8.26
Synonyms .....	8.29
Copying a Database Object .....	8.30
Deleting a Database Object .....	8.31

---

## Overview

*Database objects* store and organize information in relational databases. In Oracle Power Objects, database objects are represented by icons in a database session window.



Database objects, unlike application objects, are not created and maintained directly by Oracle Power Objects. Instead, these objects are created and maintained by a *database engine*, a component of the database in which the objects are stored. Because relational database engines have different capabilities, available object types and features vary from database to database.

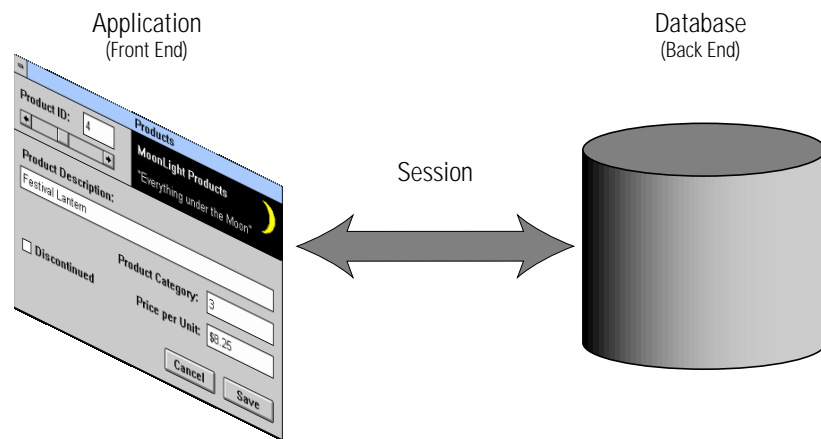
Database objects form the “back end” of an Oracle Power Objects application. In developing applications, you connect database objects to application objects (such as forms and reports). The application objects provide a “window” onto the database objects by presenting the stored information in a useful format. The process of connecting database objects to application objects is called *binding*.

This chapter discusses how to design and modify database objects; using them is discussed in the following chapters:

- Binding application objects to database objects is described in Chapter 17, “Binding a Container to a Record Source”.
- Creating master-detail relationships is described in Chapter 18, “Defining Master-Detail Relationships”.

### Database Objects and Sessions

Database objects appear to be contained within a session object. However, they are physically stored in the database to which the session provides access. Each database session window shows the database objects belonging to a single database user. The following diagram shows the relationship between sessions and databases:



In some databases (such as Oracle7 Servers and Blaze databases), each user's objects are stored in a separate *schema*. A schema is simply a named collection of database objects within a database. Each database user has a schema, which has the same name as the user. For example, the user SCOTT has a schema named SCOTT. For databases that support schemas, each database session object provides access to a single schema.

A database session window does not necessarily show all objects available to the user—it shows only the objects that the user owns (objects created by that user). For example, the database session window does not show public synonyms or objects belonging to other users for which the current user has permissions.

### Basic Database Object Types

The following types of database objects are common to most databases that Oracle Power Objects can access:

**Tables** are the database objects that actually store data. A single table generally stores information about a single topic (such as company employees or customer addresses). The information in a table is organized into *rows* and *columns*.

**Views** are customized presentations of the data in one or more tables. A view is like a "virtual table" that allows you to relate and combine data from multiple tables and views (called *base tables*). Views, like tables, are organized into rows and columns; however, views contain no data themselves. Views allow you to treat multiple tables or views as one database object.

---

**Indexes** provide fast access to individual rows in a table. Indexes store “pointers” to each row in the table in a format highly optimized for searching for and sorting data. Once you create an index, the index is automatically maintained and used whenever you access the indexed columns.

**Sequences** generate a series of integers, which can be used to provide unique identifiers for the rows of a table. You can use values from a sequence to ensure that a column contains no duplicate values (for example, a primary key column). Some databases, such as SQL Server, do not support sequences; for these databases, Oracle Power Objects provides alternative techniques for generating unique values.

**Synonyms** provide aliases to other database objects (tables, views, and sequences). Synonyms can provide public access to commonly used objects, and can hide the location and owner of an object.

Oracle Power Objects provides graphical interfaces for working with these basic database objects.

### Other Database Object Types

External databases (such as Oracle7 Servers) can contain many additional database objects (such as clusters, packages, snapshots, and roles) which are frequently used to provide extra security or to enhance performance. To access these objects, you must execute SQL commands using the Oracle Basic EXEC SQL command or SQLLOOKUP function.

### Properties and Methods of Database Objects

Database objects do not have properties and methods in the same way that application objects do because they are not created through the Oracle Power Objects object model. Instead, database objects are created and modified by the database engine of the database in which they are stored.

Database objects have an associated Property sheet, but you cannot add user-defined properties or methods to them. Most database objects have a **Name** property, which is provided for developer reference. The **Name** property can be changed using the Property sheet at design time and will rename the object in the database, but this property cannot be read or changed through Oracle Basic during run time.

### Operations on Database Objects

You can perform two general types of operations on database objects: *data definition operations* and *data manipulation operations*.

**Data definition operations** apply to the structure of a database object. They include creating, deleting, and modifying the structure of database objects. These operations are generally performed by the developer at design time.

For information about EXEC SQL and SQLLOOKUP, see Chapter 9, “Structured Query Language (SQL)”.



**Data manipulation operations** apply to the data stored in or accessible through the object. They include querying, inserting, updating, and deleting data rows. Data manipulation operations apply mainly to tables and views, although they are sometimes used with other database objects such as sequences. These operations can be performed both by the developer at design time and by the user at run time.

When you create, delete, or make changes to a database object from within Oracle Power Objects, the changes you specify are automatically converted into Structured Query Language (SQL) statements, which are then sent to the database engine for execution. Oracle Power Objects itself does not execute the modifications you specify—instead, it passes them along to the database engine.

Each type of operation has an associated set of SQL commands: data definition operations use *Data Definition Language* (DDL) commands, while data manipulation operations use *Data Manipulation Language* (DML) commands.

For information about SQL commands, see the section "Commands" on page 9.13.



**Important:** Executing any data definition operation (for example, creating a table) automatically commits *all* transactions associated with the session to which the operation applies. Therefore, you should avoid data definition operations when transactions are pending.

The types of operations you may perform on a database object are determined by the *privileges* you have for that object. By default, the owner of the object (the user who created the object) has all privileges for the object. For other users to access the object, the owner must grant privileges to other users.

The types of object privileges available vary from database to database, as does the SQL syntax for granting or revoking privileges. However, the following privilege types are common and apply to Oracle7 Servers. Blaze databases do not have object privileges—all users in a Blaze database have privileges for all objects in the database.

<b>Privilege Type</b>	<b>Actions Allowed</b>
ALTER	Changing the object structure.
DELETE	Removing data rows.
INSERT	Inserting data rows.
SELECT	Querying data rows.
UPDATE	Changing data rows.

To grant or revoke privileges, you must issue SQL statements. For more information about granting and revoking privileges, see the documentation accompanying your database.

---

## Database Object Names

Names for database objects must adhere to the object naming rules for the database in which they are stored. These rules vary from database to database.

For information about the database object naming rules for Blaze databases, see the topic “Database Object Naming Conventions” in the online help. For the database object naming rules for any other database, see the documentation accompanying that database.

## Tables

Tables are the fundamental data storage objects in a relational database. You store most of your application’s data in tables.

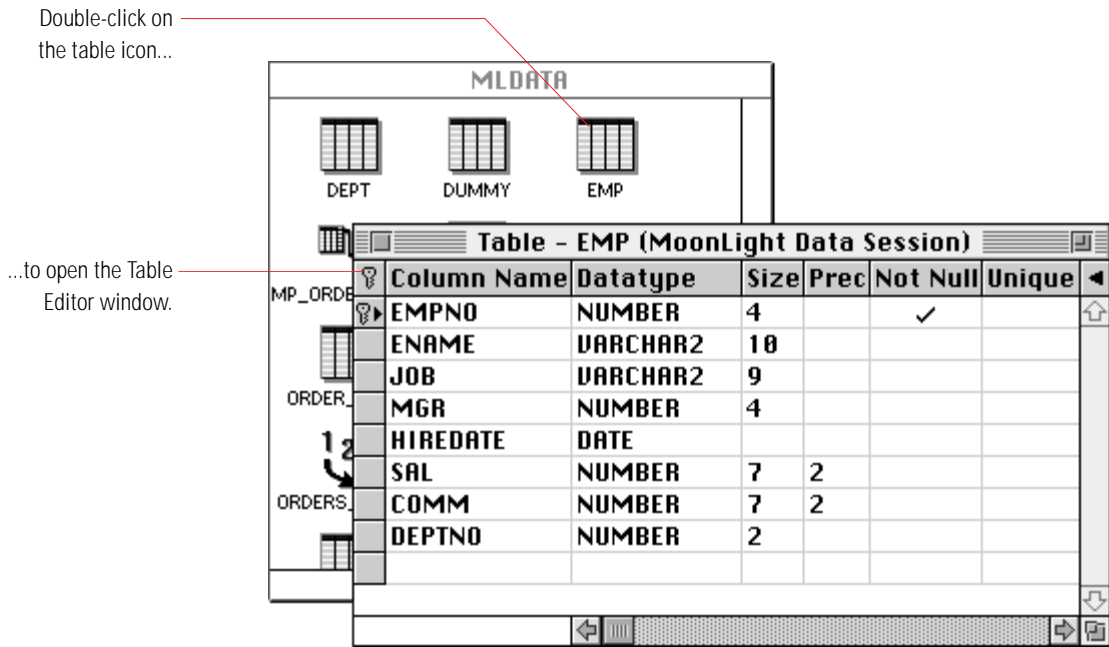
Column Name	Datatype	Size	Prec	Not Null	Unique
EMPNO	NUMBER	4		✓	
ENAME	VARCHAR2	10			
JOB	VARCHAR2	9			
MGR	NUMBER	4			
HIREDATE	DATE				
SAL	NUMBER	7	2		
COMM	NUMBER	7	2		
DEPTNO	NUMBER	2			

A table has two components: the *table structure* and the *table data*.

### Table Structure

You specify the *table structure* (also called the *table definition*) when you create the table. The table structure is the portion of the table that you design before adding any data to the table—it defines what kind of data the table will store and rules associated with entering, modifying, or deleting the data.

The table structure is visible in the Table Editor window. To open the Table Editor window for an existing table, you double-click on the table icon in the database session window.



The table structure includes the following information:

- **Table Name.** The name by which you refer to the table in properties, methods, and SQL statements.
- **Table Columns.** The categories of information stored in the table. Each column has a name and a datatype. Some types of columns can also be given an explicit size (for example, you could define a VARCHAR2 column to hold a maximum of 20 characters).
- **Table and Column Constraints.** Integrity constraints defined at the table level or at the column level. Integrity constraints are described in Chapter 19, “Using Constraints to Enforce Business Rules”.

You use the Table Editor window both to define the table structure when you first create the table, and to modify the table structure afterward. Both of these types of operations are discussed later in this section.

### Table Data

The *table data* is the information that is stored in the table. The table data is the portion of the table to which your application's users have access—for example, table data can be displayed in controls located on forms and reports.

All table data are stored in *rows*, each containing one piece of information for each column in the table structure. Users can insert, update, and delete rows of table data.

Developers can see and edit table data in the Table Browser window. To open the Table Browser window, you “run” the table when the Table Editor window is visible by clicking on the Run button in the toolbar.



EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	17-DEC-80	800		20
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30
7521	WARD	SALESMAN	7698	22-FEB-81	1250	500	30
7566	JONES	MANAGER	7839	02-APR-81	2975		20
7654	MARTIN	SALESMAN	7698	28-SEP-81	1250	1400	30
7698	BLAKE	MANAGER	7839	01-MAY-81	2850		30
7782	CLARK	MANAGER	7839	09-JUN-81	2450		10
7788	SCOTT	ANALYST	7566	09-DEC-82	3000		20
7839	KING	PRESIDENT		17-NOV-81	5000		10
7844	TURNER	SALESMAN	7698	08-SEP-81	1500	0	30
7876	ADAMS	CLERK	7788	12-JAN-83	1100		20
7900	JAMES	CLERK	7698	03-DEC-81	950		30
7902	FORD	ANALYST	7566	03-DEC-81	3000		20
7934	MILLER	CLERK	7782	23-JAN-82	1300		10

In the Table Browser window, each database column is represented by a vertical column of the spreadsheet. Each row stored in the table is represented by a horizontal row of the spreadsheet. The Table Browser window's scrollbars give you access to any rows or columns that do not fit in the window.

## Creating a Table

Because tables are the objects that will store most of your application's data, you should design your table objects carefully. Correct table design involves many considerations, and it is not within the scope of this chapter to present a complete discussion of this topic. However, the following basic table design principles are offered:

**Avoid duplicating information.** Use a separate table for each category of information you want to store. For example, you should not store department descriptions in a table that lists employee information. The process of designing tables to avoid data duplication is called *normalization*. You should normalize tables whenever possible to save space in your database and help prevent errors that can arise when information is duplicated.

Similarly, you should avoid storing values that can be calculated easily from existing values—for example, you should not store the sum of all individual items in an order, because the sum can be calculated using a simple formula.

**Include a primary key for most tables.** Almost every table you design should include a primary key column or set of columns. Primary keys are necessary to establish master-detail relationships between tables. Additionally, many databases enforce primary key constraints using an index, which can significantly improve the speed of search and sort operations that include the primary key column.

You should also include a primary key because Oracle Power Objects sometimes uses primary key values to identify individual rows. If a table lacks a primary key, Oracle Power Objects might be unable to perform certain table operations.

You might want to omit a primary key in special cases, such as tables for which an index would slow the performance of updates unacceptably, or tables that perform many-to-many relationships and so contain only foreign keys.

**Consider where you want to enforce constraints.** Many types of constraints are best enforced in the database. Chapter 19, “Using Constraints to Enforce Business Rules” includes a discussion of constraint types and the comparative advantages of enforcing constraints in the database or in your application.

Once you have decided on your table design, you can create the table. To create a table, you:

- Create a new table and specify the table name.
- Define the names and attributes of the table's columns.
- Define the table's primary key.
- Save the table.

You can also create a table with the SQL command CREATE TABLE.

### Creating a New Table

The first step in creating a table is to open a new, blank Table Editor window for the table. This step does not save any information in the database—you do not save the table definition until you have finished defining all of the table columns. However, it does create an Oracle Power Objects table object icon you can work with.

☆ **To create a new table:**



- 1** Activate the Database Session window of the database where you want to create the table. If the database session is not currently active, double-click the Connector control.



- 2** Click the New Table button in the Desktop toolbar. The Table Editor window appears on your screen, along with the table's Property sheet.

- 
- 3 If you wish, change the default table name in the **Name** property in the Property sheet.  
The table name must adhere to the database object naming rules for your database.

### Adding Columns to a Table

After creating a new table, you must add one or more columns to the table.

Although it is possible to add columns to a table after you have saved it for the first time, you generally should finish adding columns to the table before saving it. In most databases, you cannot drop a column after you have saved the table definition.

☆ **To add columns to the table:**

- 1 Enter a name for the column in the Column Name field.  
The column name must adhere to the database object naming rules for your database.
- 2 Select the column's datatype in the Datatype field.  
The list of available datatypes varies depending on the database in which the table will be stored.
- 3 If you wish to specify additional column information, enter or change the information in the appropriate fields.  
You can enter or change the information in the following fields:

<b>Field</b>	<b>Description</b>
Size	Specifies the maximum length of values in the column. Required for string columns. For Oracle7 Servers and Blaze Databases, the Size field corresponds to the "Precision" value of a numeric datatype declaration.
Precision	Specifies the numeric precision of floating-point or fixed-point numeric columns. For most datatypes, this is specified as decimal precision; for columns of datatype FLOAT, this is binary precision. For Oracle7 Servers and Blaze Databases, the Precision field corresponds to the "Scale" value of a numeric datatype declaration.
Not Null	If true, specifies a Not Null constraint for the column, ensuring that a value of null cannot be entered for the column.
Unique	If true, specifies a Unique constraint for the column, ensuring that the column has no duplicate values.

- 4 Follow steps 1 through 3 to specify additional columns.

For information about table datatypes, see the section "Values and Datatypes" on page 9.3.

Primary key constraints are described in the section "Primary Key Constraints" on page 19.4.

### Specifying a Primary Key

A table's *primary key* is a specially designated column or set of columns that uniquely identifies each row of data in the table. When you specify a primary key for a table, Oracle Power Objects creates a *Primary Key constraint* on the column, which ensures that each primary key value is unique and that there are no null values in the primary key.

You can define a primary key either for a single column or for a combination of columns. A composite primary key constraint ensures that each row has a unique combination of values in its key columns.

You can add or change a table's primary key at any time. However, for Oracle7 Servers and SQL Server databases, any change you make to a Primary Key constraint requires the database to create a new index object, which can be a time-consuming process.

☆ **To specify a primary key:**

- 1 In the Table Editor window, click on the name of the column you want to make a primary key.
- 2 Click the Primary Key tool icon.

A key icon appears in the area at the left of the column.

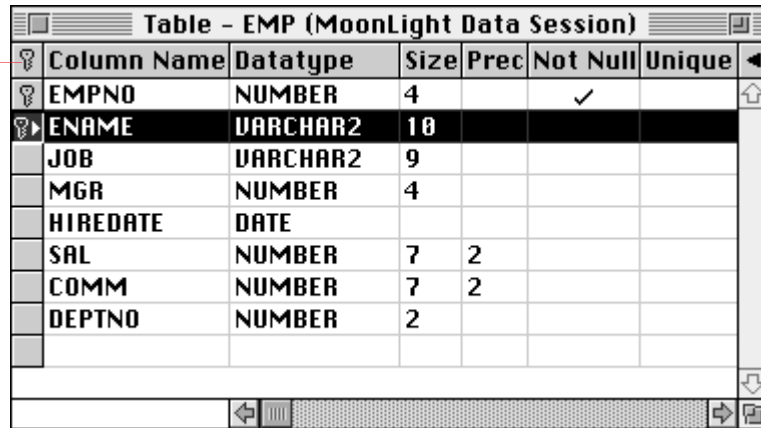
Click on the Primary Key tool to enable a Primary Key constraint.

Column Name	Datatype	Size	Prec	Not Null	Unique
EMPNO	NUMBER	4		✓	
ENAME	VARCHAR2	10			
JOB	VARCHAR2	9			
MGR	NUMBER	4			
HIREDATE	DATE				
SAL	NUMBER	7	2		
COMM	NUMBER	7	2		
DEPTNO	NUMBER	2			

Clicking on the Primary Key tool a second time removes the Primary Key constraint from the column.

- 3 For composite keys, repeat steps 1 and 2 for additional columns in the primary key.

Click on the Primary Key tool again to create a composite key.



Column Name	Datatype	Size	Prec	Not Null	Unique
EMPNO	NUMBER	4		✓	
ENAME	VARCHAR2	10			
JOB	VARCHAR2	9			
MGR	NUMBER	4			
HIREDATE	DATE				
SAL	NUMBER	7	2		
COMM	NUMBER	7	2		
DEPTNO	NUMBER	2			

### Saving the Table



Once you have finished defining the table's columns, you must save the table to apply the definition. You do so by clicking on the Save button or by choosing the **File-Save** menu command. Oracle Power Objects executes an implicit CREATE TABLE statement when you save a newly created table.

### Editing Table Definitions

You can edit the definition of a table in the Table Editor window. You edit a table definition when you want to add or delete a table column or when you want to change the names or attributes of existing columns. However, there are some kinds of changes that you cannot make to a table that has already been created. These restrictions depend on the type of database in which the table is stored and whether the table already contains data. For a complete list of restrictions on altering an existing table, see the documentation accompanying your database on the ALTER TABLE command or its equivalent.

You can also edit a table definition with the SQL command ALTER TABLE.

### Adding a Column to a Table

When you add a column to a table that contains data, the column value is set to null for all of the rows currently in the table. Therefore, you cannot set a new column's Not Null field to True if you add it to a table that contains data.



☆ **To add a column to a table:**

- 1 Open the Table Editor window of the table you want to change.
- 2 In the first blank cell of the Column Name field, enter the name of the new column.
- 3 In the Datatype field, choose the column's datatype.
- 4 If you wish, specify additional column information in the other fields of the Table Editor window.
- 5 Save the table by clicking the Save button or by choosing the **File-Save** menu command.



**Changing Column Attributes**

You can change the attributes of existing columns on a limited basis. To change a column attribute, simply change the value in the appropriate field of the Table Editor window.

**Column Name Field**

*Blaze Database:* You can change the name of a column at any time.

*Oracle7 Server, SQL Server:* You cannot change the name of a column after creation.

**Datatype Field**

*Blaze Database:* You can change the datatype of a column if the existing data in the column can be converted to the new datatype.

*Oracle7 Server:* You can change the datatype of a column only if the column contains no data or contains null in all rows.

*SQL Server:* You cannot change the datatype of a column after creation.

**Size Field**

*Oracle7 Server, Blaze Database:* You can always increase the size of a column, up to the size limit for the column datatype. However, you can decrease a column's size only if the column contains no data or contains null in all rows.

*SQL Server:* You cannot change the size of a column after creation.

**Precision Field**

*Blaze Database:* You can change the precision value of a column at any time, since Blaze stores the precision value you specify but does not enforce it.

*Oracle7 Server:* You can change the precision of a column only if the column contains no data or contains null in all rows.

---

### Not Null Field

*Oracle7 Server, Blaze Database:* You can set a column's Not Null field to true only if the column contains no null values. You can set a column's Not Null field to false at any time.

*SQL Server:* You cannot change a column's Not Null field after creation.

### Unique Field

*All databases:* You can change a column's Unique field at any time. For Oracle7 Servers and SQL Server databases, setting the Unique field to true may require creating a new index object for the column, which can be a time-consuming process.

## Editing the Contents of a Table

You can edit table data in the Table Browser window, a spreadsheet-like representation of the rows and columns in the table. You can insert, update, and delete data rows using the Table Browser window.

The Table Browser window is intended primarily for developers as a quick and simple way to view and edit table data. However, while constraints in the table definition are enforced, your modifications can violate constraints that you have defined using application objects that are bound to the table. Also, the Table Browser window does not provide any automatic way to generate primary key values—you must enter such values manually.

The Table Browser window lets you make the following types of modifications to data:

- Insert rows
- Edit values
- Delete rows

You can also edit data in a table with the SQL commands SELECT, INSERT, UPDATE, and DELETE.

### Inserting Rows into a Table

You insert a new row of data into the table by entering a set of values into the Table Browser.

☆ **To insert a row into a table:**



- 1 In the Table Browser window, click the Insert Row button in the toolbar.

-or-

Click in any field in the first empty row.

- 2 Enter values in the fields of the row.

Make sure that you enter values for all of the mandatory (Not Null) columns in the table.

### Editing Values in a Table

To edit values in the table, simply type directly over the old value. The new values you enter must obey any constraints you have included in the table definition. For example, you cannot enter a duplicate value in a column with a Not Null or Primary Key constraint.

After you edit a table value, a lock indicator appears next to the row, indicating that a lock has been acquired on the row's data. The lock is not released until you commit or roll back your changes.

### Deleting Rows from a Table

☆ **To delete a row from a table:**

- 1 In the Table Browser window, click the Row Selector button next to the row you want to delete. You can select multiple rows by Control-clicking (Option-clicking on Macintosh) additional Row Selector buttons.

Control-click on Row Selector buttons to select additional rows.

Table - EMP (MLDATA)								
EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO	
7369	SMITH	CLERK	7902	17-DEC-80	800		20	+
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30	
7521	WARD	SALESMAN	7698	22-FEB-81	1250	500	30	
7566	JONES	MANAGER	7839	02-APR-81	2975		20	
7654	MARTIN	SALESMAN	7698	28-SEP-81	1250	1400	30	
7698	BLAKE	MANAGER	7839	01-MAY-81	2850		30	
7782	CLARK	MANAGER	7839	09-JUN-81	2450		10	
7788	SCOTT	ANALYST	7566	09-DEC-82	3000		20	
7839	KING	PRESIDENT		17-NOV-81	5000		10	
7844	TURNER	SALESMAN	7698	08-SEP-81	1500	0	30	
7876	ADAMS	CLERK	7788	12-JAN-83	1100		20	
7900	JAMES	CLERK	7698	03-DEC-81	950		30	
7902	FORD	ANALYST	7566	03-DEC-81	3000		20	
7934	MILLER	CLERK	7782	23-JAN-82	1300		10	



- 2 Press the Delete key or click the Delete button in the toolbar or choose the Edit-Cut menu command.

The selected rows disappear from the Table Browser window.



### Saving Changes

To save your changes, click the Commit button in the toolbar. To undo changes instead, click the Rollback button. When you commit or roll back your changes, all data locks that have been acquired on table data are released and the lock indicators are removed from the Table Browser window.

If you have made any illegal changes to data (for example, inserted a row that is missing a value in a column with a Not Null constraint), an error occurs when you attempt to save your changes. You then cannot exit the Table Browser window until you either correct the problem or roll back your changes.

### Closing the Table Browser Window

When you have finished making changes to your data, you can close the Table Browser window by clicking on the Close button in the window's title bar. If you have made any changes that have not been committed or rolled back, you cannot close the Table Browser window until you commit or roll back your changes.

## Using Tables

You can use the data in a table in several ways in your application. The following uses are the most common:

**Bind the table to a container object.** You can associate a *bindable container* such as a form, report, or repeater display with a table using the container's **RecordSource** and **RecSrcSession** properties. You can then bind *controls* in the container, such as fields, radio buttons, and check boxes, to columns of the table.

**Create a translation list.** You can use the **Translation** property of popup lists and scrolling lists to generate a list of values linked to the table's rows. You can also set the **ValueList** property of a combo box to display suggested values.

**Issue a SQL statement.** The Oracle Basic command EXEC SQL lets you issue a custom SQL statement that refers to the table. SQL statements offer the most flexibility in accessing the table's values, but require knowledge of SQL and the specific feature set of the database containing the table.

## Views

A view can be thought of as a "stored query", because it is defined as a query that accesses information in one or more *base tables* or views. A views does not actually contain data, but acts as a "virtual table" that can be examined and, in some cases, modified in the same manner as tables.

Views are commonly used to provide the following features:

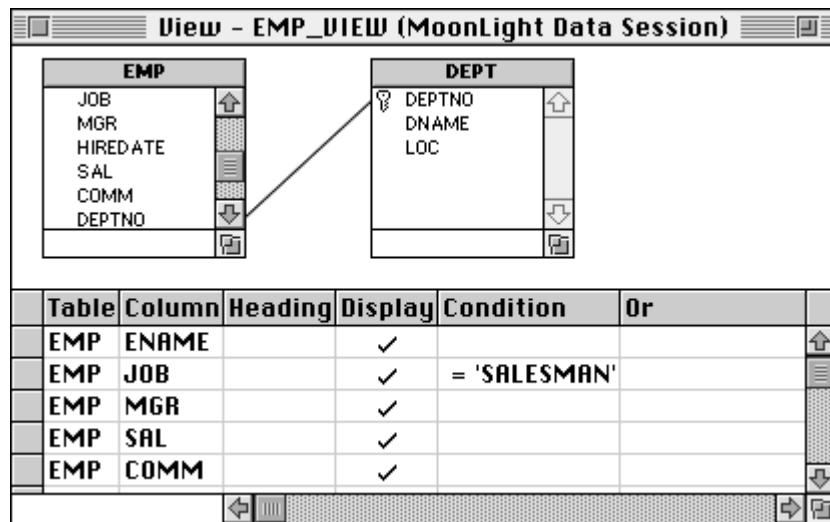
- **Hide data complexity** by joining information from a number of tables into a single logical unit.
- **Perform calculations** on data stored in tables to make information more meaningful.
- **Provide security** by restricting access to a subset of a table's rows and columns (not applicable to Blaze databases, which lack protected schemas).

Like a table, a view has two components: the *view structure* and the *view data*.

### View Structure

You specify the *view structure* (also called the *view definition*) when you create the view. The view structure is essentially a query that selects information from one or more base tables or views. The columns of the view correspond to the output list of the query.

The view structure is represented graphically in the View Editor window. To open the View Editor window for an existing view, you double-click on the view icon in the database session window.



The view structure displayed above translates into a SQL query of the following form:

```
SELECT EMP.ENAME, EMP.JOB, EMP.MGR, EMP.SAL,
       EMP.COMM, DEPT.DNAME
FROM EMP, DEPT
WHERE EMP.DEPTNO = DEPT.DEPTNO
AND EMP.JOB = 'SALESMAN'
```

➤ **Note:** Oracle Power Objects cannot edit the structures of certain views (for example, views that include an expression in the select list, or views containing GROUP BY or CONNECT BY clauses). When you open these types of views, Oracle Power Objects displays the View Browser window, described later in this chapter.

---

You use the View Editor window both to define the view structure when you first create the view, and to modify the view structure afterward. Both of these types of operations are discussed later in this section.

### View Data

The *view data* is the information that is stored in the base tables of the view. As with table data, view data is the portion of the view to which your application's users have access, although views are frequently used to display read-only information.

Developers can see and (if the view is read-write) edit view data in the View Browser window. To open the View Browser window, you “run” the view when the View Editor window is visible by clicking on the Run button in the toolbar.



As with table data, all view data are displayed as *rows* each containing one piece of information per view column. Users can select information from views just as with tables. However, the user can update or delete rows from a view only if the following criteria are true:

- The view has only one base table.
- The view's columns correspond directly to columns of the base table and include no expressions.

If either of these criteria is false, then the view is a *read-only view*.

### Creating a View

To create a view, you:

- Create a new view and specify the view name.
- Choose the base tables that provide the data underlying the view.
- Specify the relationships that join the base tables together.
- Select the table columns you want to appear in your view.
- (optional) Specify additional conditions to restrict the values that appear in the view.
- Save the view.

You can also create a view with the SQL command CREATE VIEW.

### Creating a New View

☆ **To create a new view:**



- 1 Activate the Database Session window of the session where you want to create the view. If the database session is not currently active, double-click the Connector control.



- 2 Click the New View button in the toolbar.

The View Editor window appears on your screen, along with the view's Property sheet.

- 3 If you wish, change the default view name in the **Name** property in the Property sheet.  
The view name must adhere to the database object naming rules.

### Choosing Base Tables for a View

The view's base tables appear in the Table List area of the View Editor window. You can use either a table or another view as a base table.

#### ☆ To add a base table to a view:

- 1 Activate the Database Session window of the session that contains the base table you want to use.
- 2 Select the table (or view) icon of the base table, then drag the icon into the Table List area of the View Editor window.

The definition of the table or view appears in the Table List area of the View Editor window.

#### ☆ To remove a base table from a view:

- 1 In the Table List area of the View Editor window, select the definition of the table or view.



- 2 Press the Delete key or click the Delete button in the toolbar or choose the **Edit-Cut** menu command.

The table's definition disappears from the Table List area.

### Joining Base Tables in a View

Oracle Power Objects represents joins between the view's base tables by lines that connect two columns together.

#### ☆ To join base tables:

- 1 In the Table List area of the View Editor window, select the column name of one of the columns you want to join.
- 2 Drag the selected column name onto the corresponding column in the other table's definition.

A join line appears between the columns. In the SELECT statement that underlies the view, the join line corresponds to a WHERE clause of the form "WHERE table1.col = table2.col".

---

☆ **To remove a join between tables:**

1 In the Table List area of the View Editor window, select the join line by clicking on it.



2 Press the Delete key or click the Delete button in the toolbar or choose the **Edit-Cut** menu command.

The join line disappears.



**Note:** All of the base tables in your view should be linked together by joins. The view will contain an unexpected set of data if it contains unjoined tables.

### **Adding Columns to a View**

The view's columns appear in the Column List area of the View Editor window. For each column of the view, there is a base table column that provides the data source.

☆ **To add columns to the view:**

1 In the View Editor window, double-click the desired column name or drag it from the Table List area to the Column List area.

-or-

Enter a column name in the Column Name field and the name of the containing table in the Table field.

2 Enter a name for the view column in the Heading field.

This is the name by which you refer to the view column in properties, methods, and SQL statements. The column name must adhere to the object naming rules of your database.

3 Choose whether you want the column to be displayed.

Click on the Display field to toggle between True (the column is displayed) and False (the column is not displayed). If the column is not displayed, you cannot use the column as a source of data (for example, you cannot SELECT a value from the column). However, the values in the Condition field and Or field values are still used in the view definition. Nondisplayed columns are typically used to add conditions that restrict the rows in the view.

4 If you wish to restrict the values that appear in the view, add conditions to the Condition field and Or field.

The condition you enter takes the form of a comparison operator followed by an expression (for example, >1000 or LIKE 'SM\* '). Only rows where the comparison evaluates to TRUE are included in the view.



If you enter values in both the Condition field and the Or field of the same column, the values are joined with a logical OR (union). The conditions for separate view columns are joined together with a logical AND (intersection).

Note that conditions in the Condition field and Or field are SQL conditions, not Oracle Basic conditions.

### Specifying Additional Conditions

You can specify additional conditions to restrict the view's values by adding nondisplaying columns to the view (columns whose Display field is set to False). You can add conditions in the Condition field and the Or field of nondisplaying columns.

### Saving the View



Once you have finished defining the view's columns, you must save the view to apply the definition. You do so by clicking the Save button in the toolbar or by choosing the **File-Save** menu command. Oracle Power Objects executes an implicit CREATE VIEW statement when you save a newly created view.

### Editing View Definitions

You can edit the definition of a view in the View Editor window. You edit a view definition when you want to add or delete a base table, add or delete a view column, or change the names or attributes of existing columns. The view definition is dropped and re-created when you save your changes.

You can also modify a view's definition with the SQL command CREATE OR REPLACE VIEW.

### Adding or Removing Base Tables

You can add or remove base tables from the view at any time by following the instructions in the section "Choosing Base Tables for a View" on page 8.19. Whenever you add a base table, make sure to join it to another base table in the view.

### Adding Columns to a View

☆ **To add a column to a view from a base table:**

- 1 Open the View Editor window of the view you want to modify.
- 2 In the Table List area, select the name of the column in the appropriate table definition.

---

**3** Drag the column name into the Column List area.

-or-

Double-click on the column name in the Table List area.

The column you selected appears in the Column List area.

**4** If you wish, add conditions for the column in the Condition field and Or field.

☆ **To add a nondisplaying condition column to a view:**

**1** Open the View Editor window of the view you want to modify.

**2** In the Table List area, select the name of the column in the appropriate table definition.

**3** Drag the column name into the Column List area.

-or-

Double-click on the column name in the Table List area.

The column you selected appears in the Column List area.

**4** Set the Display field to FALSE by clicking on it.

The check mark disappears from the Display field.

In the Condition field, enter a condition value. The condition you enter takes the form of a comparison operator followed by an expression (for example, `>1000` or `LIKE 'SM*'`). Only rows where the comparison evaluates to TRUE are included in the view.

Note that the condition in the Condition field is a SQL condition, not an Oracle Basic condition.

**5** If you wish, add an additional condition value to the Or field.

A condition value added to the Or field is added to the condition in the Condition field with a logical OR (union).

☆ **To delete a column or condition from a view:**

**1** Open the View Editor window of the view you want to modify.

**2** In the Column List area, click on the Row Selector button next to the column or condition that you want to delete.

You can select multiple columns or conditions by Control-clicking on additional Row Selector buttons (Option-clicking on Macintosh).



**3** Press the Delete key or click the Delete button in the toolbar or choose the **Edit-Cut** menu command.

The column or condition is deleted from the view.

### Modifying a View Column or Condition

In general, you can modify any of the fields that define a view column or condition. Simply select the old value, delete it, and type the new value.

### Editing the Contents of a View

Views are either read-only or read-write. You can edit the data in a read-write view in the View Browser window, a spreadsheet-like representation of the rows and columns in the view. For read-only views, you can use the View Browser window to examine but not change the view data.

You edit data in the View Browser window in the same way as with the Table Browser window, as described in the section “Editing the Contents of a Table” on page 8.14.

As with the Table Browser window, the View Browser window is intended primarily for developers as a quick and simple way to view and edit table data. However, while constraints in the view definition are enforced, your modifications can violate constraints that you have defined using application objects that are bound to the view. Also, the View Browser window does not provide any automatic way to generate primary key values—you must enter such values manually.

### Using Views

You can use the data in a view in several ways in your application. In general, views can be used anywhere you use a table, although many views are somewhat restricted because they are read-only. The following uses are the most common:

**Bind the view to a container object.** You can associate a *bindable container* such as a form, report, or repeater display with a view using the container's **RecordSource** and **RecSrcSession** properties. You can then bind controls in the container, such as fields, radio buttons, and check boxes, to columns of the view.

**Create a translation list.** You can use the **Translation** property of popup lists and scrolling lists to generate a list of values linked to the view's rows. You can also set the **ValueList** property of a combo box to display suggested values.

**Issue a SQL statement.** The Oracle Basic command EXEC SQL lets you issue a custom SQL statement that refers to the view. SQL statements offer the most flexibility in accessing the view's values, but require knowledge of SQL and the specific feature set of the database containing the view.

---

## Indexes

Indexes are database objects that provide fast access to individual rows in a table. You create an index to increase the performance of queries and sorting operations against the table's data. Indexes are also used to enforce some types of constraints on tables (for example, Unique and Primary Key constraints); these indexes are frequently created automatically when the constraint is defined. An index is an independent object that is logically separate from the table being indexed—creating or deleting an index does not affect the definition or data of the indexed table.

An index stores highly optimized versions of all of the values in one or more of the table columns. When a value is queried from an indexed column, the database engine uses the index to locate the desired value quickly.

Indexes must be maintained constantly to reflect any changes to the indexed rows of the table. The database engine automatically takes care of any necessary index maintenance when a value in an indexed column is inserted, updated, or deleted. Although this maintenance does not require any action on the user's part, it does reduce the performance of some data manipulation operations (except for queries). However, the reduced performance associated with maintaining the index is frequently offset by the increased data access speed that the index provides.

Indexes provide the greatest benefits for tables that are frequently queried but infrequently modified. Indexes are commonly used with the following types of columns:

**Primary Key columns.** Primary key columns are often used for searching and sorting, so an index can improve the performance of many types of operations. In addition, some types of databases automatically use an index to enforce a Primary Key constraint.

**Foreign Key columns.** Indexes can help improve the speed of joining two tables in a master-detail relationship.

**Frequently sorted columns.** Indexes store information in a presorted format, which increases the speed of sorting a table's data by an indexed column.

### Creating an Index

☆ **To create an index:**

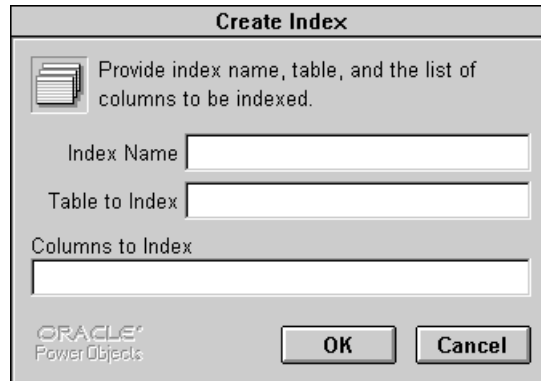


- 1 Activate the Database Session window of the database where you want to create the index. If the database session is not currently active, double-click the Connector control.



- 2 Click the New Index button in the toolbar, or choose the **File-New Index** menu command.

The Create Index dialog box appears on your screen.



3 Enter required index information.

Field	Information
Index Name	The index name. This must adhere to the database object naming conventions for your database.
Table to Index	The name of the table object for which the index is to be created.
Columns to Index	The name of the column or columns for which the index is to be created. For a multi-column index, separate column names by commas.

4 Click the OK button.

The index is created. An index icon appears in the Database Session window.

For some types of databases, index objects are created implicitly when you enable a Primary Key or Unique constraint on a table column or set of columns. You can enable these constraints in the Table Editor window.

For more information about constraints, see Chapter 19, "Using Constraints to Enforce Business Rules".

➤ **Note:** in a Blaze database, implicitly created indexes are stored in the INDEXES schema.

You can also create an index with the SQL command CREATE INDEX.

## Using Indexes

Once an index is created, it is always used whenever possible. To take advantage of an index, ensure that your queries use an indexed column or set of columns in the **DefaultCondition**, **QueryWhere()**, and **OrderBy** properties and methods (in the Property sheet) or in the WHERE and ORDER BY clauses (in an EXEC SQL statement).

---

## Sequences

Sequences are database objects that generate unique integers, which are typically used wherever unique data values are required in the database. Most commonly, sequences are used to generate values in a primary key column of a table. Whenever a value is selected from the sequence, the sequence is automatically incremented or decremented.

A sequence can have the following characteristics:

- **Name**—the name of the sequence.
- **Increment**—the spacing between values generated by the sequence (this number can be negative, but it cannot be zero).
- **Beginning number**—the first value generated by the sequence. This value cannot be altered after the sequence is created.
- **Minimum value**—the lowest value that can be generated by the sequence.
- **Maximum value**—the highest value that can be generated by the sequence.
- **Cycling**—if the sequence reaches its maximum or minimum value, it can “wrap around” to continue generating values.

### Creating a Sequence

☆ **To create a sequence:**

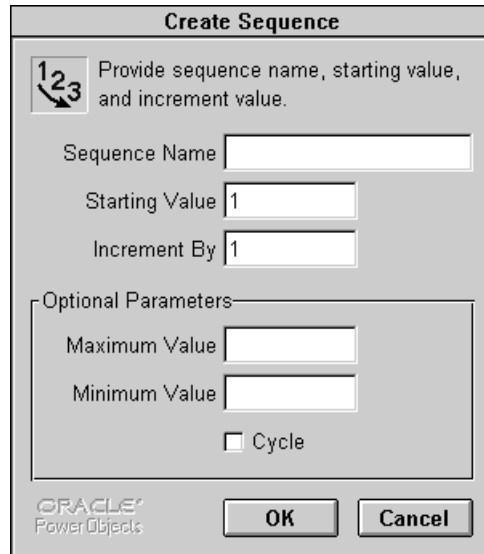


- 1 Activate the Database Session window of the database where you want to create the sequence.  
If the database session is not currently active, double-click the Connector control.



- 2 Click the New Sequence button in the toolbar, or choose the **File-New Sequence** menu command.

The Create Sequence dialog box appears on your screen.



3 Enter required sequence information.

Field	Information
Sequence Name	The sequence name. This must adhere to the database object naming conventions for your database.
Starting Value	The first value generated by the sequence. This value must be an integer. It cannot be altered after the sequence is created.
Increment By	The spacing between values generated by the sequence. This value must be an integer. It can be negative, but it cannot be zero.

4 Enter optional sequence parameters, if desired.

Field/Button	Information
Maximum Value	The highest value that can be generated by the sequence. This value must be an integer.
Minimum Value	The lowest value that can be generated by the sequence. This value must be an integer.
Cycle button	Indicates whether the sequence “wraps around” when it reaches its maximum or minimum value. If the Cycle button is checked, the sequence “wraps around”.

- 
- 5 Click the OK button.

The sequence is created. A sequence icon appears in the Database Session window.

You can also create a sequence with the SQL command CREATE SEQUENCE.

## Using Sequences in Applications

For information about sequences and other counters, see the section “Counter Fields” on page 19.17.

You can associate a sequence with a field or other control on a *bound container* (such as a form) using the **CounterType** and **CounterSeq** properties of the control. When a new row is inserted, the control's value is automatically selected from the named sequence.

A typical use of a sequence is to create a dedicated sequence object for each column requiring a unique value. The name of the sequence usually reflects the table and column name with which the sequence is associated—for example, a sequence that generates values for the EMPNO column of the EMP table might be named EMP\_EMPNO\_SEQ.

➤ **Note:** Some databases, such as SQL Server, do not support sequence objects. Oracle Power Objects provides alternative techniques for generating unique values for these databases.

## Using Sequences in SQL Statements

You can use a sequence in a SQL statement by referring to the .CURRVAL and .NEXTVAL pseudo-columns of the sequence.

Column	Meaning
.CURRVAL	reads the current value of the sequence without incrementing it
.NEXTVAL	increments the sequence and reads the new value

You cannot select a value directly from a sequence, as with a table or view. Instead, you must include the reference *sequence\_name*.CURRVAL or *sequence\_name*.NEXTVAL in a statement such as SELECT, INSERT, or UPDATE.

For example, to select a new value from a sequence called “EMP\_EMPNO\_SEQ”, you could issue a SELECT statement like the following one:

```
EXEC SQL SELECT emp_empno_seq.nextval FROM dual
```

You could use a sequence to insert an ID value into a table by issuing an INSERT statement like the following one:

```
EXEC SQL INSERT INTO emp(empno, ename, mgr) &  
VALUES (emp_empno_seq.nextval, 'PARKER', 7902)
```



## Synonyms

A synonym is a database object that provides an “alias” to another database object (a table, view, or sequence). Instead of using the name of a database object, you can provide the name of a synonym instead. Synonyms provide the following features:

**Provide public access to objects**—a synonym can provide all database users with access to commonly used objects such as data dictionary views.

**Hide information about objects**—a synonym masks the location and owner of a database object. Such a mask provides additional data security and simplifies commands that access the object.

### Creating a Synonym

☆ **To create a synonym:**



- 1 Activate the Database Session window of the database where you want to create the synonym. If the database session is not currently active, double-click the Connector control.



- 2 Click the New Synonym button in the toolbar, or choose the **File-New Synonym** menu command. The Create Synonym dialog box appears on your screen.

- 3 Enter required synonym information.

Field/Button	Information
Synonym Name	The synonym name. This must adhere to the database object naming conventions for your database.
For Object	The name of the table, view, or sequence object for which the synonym is to be created.

---

<b>Field/Button</b>	<b>Information</b>
Public button	Indicates whether the synonym is public (available to all users in the database). If the Public button is checked, the synonym is public.

- 4 Click the **OK** button.

The synonym is created. A synonym icon appears in the Database Session window.

You can also create a synonym with the SQL command `CREATE SYNONYM`.

## Using Synonyms

You use a synonym as though it were a table, view, or sequence. You can use the synonym to perform DML (Data Manipulation Language) and DDL (Data Definition Language) operations on the underlying object, as long as you have appropriate permissions.

In this release of Oracle Power Objects, you cannot see or work with synonyms in a Database Session window. However, you can use a synonym as a record source for a bindable container, or issue a SQL statement that refers to the synonym.

## Copying a Database Object

When you copy a database object, you create a duplicate of the object that has the same definitions and the same data. The duplicate object is not tied to the original object in any way.

You can copy a database object in two basic ways:

**To a different session of the same database type.** The copy duplicates most features of the original object, including the name. For tables and views, only features visible in the Table Editor window or View Editor window are copied—additional features (such as triggers or Check constraints) are not duplicated.

**To a session of a different database type.** Oracle Power Objects might have to perform conversions in order to copy the object. You cannot copy indexes or sequences to a different database.

Oracle Power Objects does not provide a way to duplicate a database object within the same database session.

Copying an object is equivalent to issuing a SQL `CREATE TABLE`, `CREATE VIEW`, `CREATE SEQUENCE`, or `CREATE INDEX` statement.

When copying a table to a session of a different database type, Oracle Power Objects first creates a table in the destination database that matches the definition of the source table. When an exactly matching column datatype is not available, Oracle Power Objects chooses the closest match. Only table definition elements visible in the Table Editor window are copied.

Oracle Power Objects then copies the table data to the new table. It first selects values from the source table and stores them in a temporary internal structure. It then inserts the values into the destination table. Two data type conversions are performed: one from the source table to the temporary internal structure, and one from the internal structure to the destination database.

☆ **To copy a database object:**



- 1 Open the Database Session windows of the source and destination databases.  
If the database sessions are not currently active, double-click the Connector control.
- 2 Select the icon of the object you want to copy.
- 3 Hold down the Alt key (Option on Macintosh), and drag the icon from the source Database Session window to the destination Database Session window.

The object icon appears in the Database window of your destination database. If the object name is already in use, Oracle Power Objects displays a dialog box prompting you to choose whether the existing object should be replaced. If you click **OK**, the original object in the destination database is dropped and a new object with the same name is created.

## Deleting a Database Object

When you delete a database object, you permanently remove the object and the object's contents from the containing database.

You cannot delete a database object from a session unless the user account to which the session is connected has the appropriate permissions. Otherwise, the operation fails and returns an error.

You can also delete a database object with the SQL commands **DROP TABLE**, **DROP VIEW**, **DROP SEQUENCE**, **DROP SYNONYM**, and **DROP INDEX**.



**Caution:** Once you delete a database object, the object cannot be recovered.

☆ **To delete a database object:**



- 1 Open the Database Session window of the session that contains the object you want to delete.  
If the database session is not currently active, double-click the Connector control.
- 2 Select the icon of the database object.



- 3 Press the Delete key or click the Delete button in the Desktop toolbar or choose the **Edit-Cut** menu command.

Oracle Power Objects displays a dialog box prompting you to confirm the action. To delete the object, click **Yes**. To cancel the operation, click **No**.



**Note:** Dropping a table automatically drops all of its indexes.



# 9

---

## Structured Query Language (SQL)

This chapter covers the following topics:

Overview .....	9.2
SQL Language Components .....	9.2
Values and Datatypes .....	9.3
Objects .....	9.7
Literals .....	9.8
Operators .....	9.10
Functions .....	9.11
Expressions .....	9.11
Conditions .....	9.12
Commands .....	9.13
Procedural Extensions .....	9.14
Executing SQL Statements .....	9.15
The EXEC SQL Command .....	9.15
The SQLLOOKUP Function .....	9.21

---

## Overview

*Structured Query Language*, usually abbreviated *SQL*, is a non-procedural database access language used by most relational databases. Statements in the SQL language describe operations to be performed on objects and sets of data in a database.

Oracle Power Objects uses SQL for all database access operations. In many cases, Oracle Power Objects issues SQL statements *implicitly*, meaning that Oracle Power Objects creates and executes a SQL statement “behind the scenes” so that you do not have to write SQL code to handle common operations. For example, when you define or alter a table using the Table Editor window, Oracle Power Objects automatically converts the information you specify into a SQL statement. The Record Manager also issues implicit SQL statements when application objects are bound to database objects, as described in Chapter 17, “Binding a Container to a Record Source”.

You use SQL *explicitly* in Oracle Power Objects in two ways:

- You use fragments of SQL (expressions, conditions, operators, and clauses) in some object properties and in some fields of the Table Editor and View Editor windows.
- You use fully formed SQL statements with the Oracle Basic EXEC SQL command and SQLLOOKUP function. These statements enable you to use database features that are not supported directly by the Oracle Power Objects interface, or give you explicit control over operations automated by Oracle Power Objects.

This chapter first describes the basic language components of SQL and then describes how to use EXEC SQL and SQLLOOKUP to execute custom SQL statements from Oracle Basic method code. However, this chapter does not provide a complete reference to the SQL language. Although most databases support the ANSI specification for the SQL language, the exact implementation and the language extensions provided vary from database to database. For a complete reference to the SQL language supported by your database, see the documentation accompanying it.

The SQL language supported by Blaze databases is described in the topic “SQL Language Reference” in the online help.

## SQL Language Components

The SQL Language has the following basic components:

- **Values.** A value is a piece of information with an associated datatype. Other components of SQL enable you to specify and manipulate values.
- **Objects.** An object is a named database structure that stores or organizes information. Database objects are described in Chapter 8, “Database Objects”.
- **Literals.** A literal (also called a *constant*) is a fixed data value, such as 32 or 'SMITH'.
- **Operators.** An operator performs an operation on one or more values and returns a result.

- **Functions.** A function performs calculations and returns a result. Functions usually operate on one or more *arguments* that you specify when you call the function, although some functions have no arguments.
- **Expressions.** An expression represents a value. Expressions can consist of literals, object values, and functions, singly or combined by operators.
- **Conditions.** A condition (also called a *search condition*) is a combination of one or more expressions and logical operators that evaluates to TRUE, FALSE, or unknown. In SQL, conditions and expressions are separate concepts and are not interchangeable.
- **Commands.** A command performs calculations or other operations. Commands frequently return information, but they do not return a result value in the same way that functions do.
- **Procedural extensions.** Extensions to basic SQL provide procedural capabilities, such as variables and flow-of-control statements.

Each of these components is discussed briefly in the sections of this chapter that follow.

## Values and Datatypes

A *value* is a piece of information with an associated datatype. Values are data such as numbers, text strings, dates, and images; these data can be stored in the rows and columns of tables. Other components of the SQL language enable you to specify and manipulate values.

Each database supports its own set of datatypes for values. These datatypes are used when defining a table column (all values in the column must have the specified datatype). The datatypes supported by Blaze databases, Oracle7 Servers, and SQL Server databases are listed in the following tables.

### Oracle7 Datatypes

The following datatypes are supported by Oracle7 Servers:

Datatype	Type	Range/Size	Notes
CHAR(n)	string	1 to 255 characters	Fixed-length string
VARCHAR(n)	string	1 to 32,000 characters	Equal to VARCHAR2
VARCHAR2(n)	string	1 to 2000 characters	Variable-length string
LONG	string	1 to 2 GB characters	Only one LONG/LONG RAW allowed per table
RAW(n)	binary	1 to 255 bytes	
LONG RAW	binary	1 to 2 GB	Only one LONG/LONG RAW allowed per table

---

<b>Datatype</b>	<b>Type</b>	<b>Range/Size</b>	<b>Notes</b>
NUMBER(p)	integer	1 to 9e125, 0, -1 to -9e125	
NUMBER(p,s)	fixed-point	1e-130 to 9.99...9e125, 0, -1e-130 to -9.99...9e125	
NUMBER	float	1e-130 to 9.99...9e125, 0, -1e-130 to -9.99...9e125	
DATE	date	Jan 1, 4712 BC to Dec 31, 4712 AD	
ROWID	string	—	Hexadecimal string representing the unique address of a row in its table
MLSLABEL	—	—	

### Blaze Datatypes

Blaze datatypes are very similar to Oracle7 Server datatypes, although there are a few minor differences. The following datatypes are supported by Blaze databases:

<b>Datatype</b>	<b>Type</b>	<b>Range/Size</b>	<b>Notes</b>
CHAR( <i>n</i> )	string	1 to 32K characters	Fixed-length string
VARCHAR( <i>n</i> )	string	1 to 32K characters	Equal to VARCHAR2
VARCHAR2( <i>n</i> )	string	1 to 32K characters	Variable-length string
LONG	string	1 to 4 GB characters	Any number of LONG/ LONG RAW columns allowed per table
RAW( <i>n</i> )	binary	1 to 32K bytes	
LONG RAW	binary	1 to 4 GB	
INTEGER	integer	2 <sup>31</sup> -1 to -2 <sup>31</sup> (4 bytes)	Signed long integer
NUMBER( <i>p</i> )	integer	2 <sup>31</sup> -1 to -2 <sup>31</sup> (4 bytes)	Precision ignored
NUMBER( <i>p,s</i> )	float	between +/- 1.79763134862315e308	Precision, scale ignored
NUMBER	float	between +/- 1.79763134862315e308	



Datatype	Type	Range/Size	Notes
DATE	date	Jan 1, 100 AD to Dec 31, 9999 AD	Different range from Oracle7
ROWID	string	—	Hexadecimal string representing the unique address of a row in its table
MLSLABEL	—	—	

### SQL Server Datatypes

The following datatypes are supported by SQL Server databases:

Datatype	Type	Range/Size	Notes
INT	integer	2 <sup>31</sup> -1 to -2 <sup>31</sup> (4 bytes)	
SMALLINT	integer	2 <sup>15</sup> -1 to -2 <sup>15</sup> (2 bytes)	
TINYINT	integer	0 to 255 (1 byte)	
FLOAT	float	1.7e-308 to 1.7e308, 0, -1.7e-308 to -1.7e308 (8 bytes)	
REAL	float	3.4e-38 to 3.4e38, 0, -3.4e-38 to -3.4e38 (4 bytes)	
MONEY	—	+922,337,203,685,477.5807 to -922,337,203,685,477.5808 (8 bytes)	
SMALLMONEY	—	214,748.3647 to -214,748.3648 (4 bytes)	
CHAR( <i>n</i> )	string	1 to 255 characters	Fixed-length string
VARCHAR( <i>n</i> )	string	1 to 255 characters	Variable-length string
TEXT	string	1 to 2 <sup>31</sup> -1 characters	
BINARY( <i>n</i> )	binary	1 to 255 bytes	
VARBINARY( <i>n</i> )	binary	1 to 255 bytes	Variable-length binary data
IMAGE	binary	1 to 2,147,483,647 bytes	
DATETIME	date	Jan 1, 1753 AD to Dec 31, 9999 AD	

---

<b>Datatype</b>	<b>Type</b>	<b>Range/Size</b>	<b>Notes</b>
SMALLDATETIME	date	Jan 1, 1900 AD to June 6, 2079 AD	
BIT	binary	1 bit	
TIMESTAMP	—	—	Value equal to VARBINARY(8) used to track concurrency
SYSNAME	—	—	
user-defined	any	range or size of underlying system datatype	

### Datatype Equivalencies

The following table compares corresponding datatypes in each database with the ANSI datatype specifications.

<b>ANSI Datatype</b>	<b>Oracle7 Datatype</b>	<b>Blaze Datatype</b>	<b>SQL Server Datatype</b>
CHARACTER( <i>n</i> ), CHAR( <i>n</i> )	CHAR( <i>n</i> )	CHAR( <i>n</i> )	CHAR( <i>n</i> )
CHARACTER VARYING( <i>n</i> ), CHAR VARYING( <i>n</i> )	VARCHAR( <i>n</i> ), VARCHAR2( <i>n</i> )	VARCHAR( <i>n</i> ), VARCHAR2( <i>n</i> )	VARCHAR( <i>n</i> )
NUMERIC( <i>p,s</i> ), DECIMAL( <i>p,s</i> ), DEC( <i>p,s</i> )	NUMBER( <i>p,s</i> )	NUMBER( <i>p,s</i> )	—
INTEGER, INT, SMALLINT	NUMBER(38)	NUMBER(38), INTEGER	INT, SMALLINT, TINYINT
FLOAT( <i>b</i> ), DOUBLE PRECISION, REAL	NUMBER	NUMBER	FLOAT, REAL
—	DATE	DATE	DATETIME, SMALLDATETIME
—	RAW( <i>n</i> )	RAW( <i>n</i> )	BINARY( <i>n</i> ), VARBINARY( <i>n</i> )
—	LONG RAW	LONG RAW	IMAGE
—	LONG	LONG	TEXT
—	ROWID	ROWID	—

ANSI Datatype	Oracle7 Datatype	Blaze Datatype	SQL Server Datatype
—	MLSLABEL	MLSLABEL	—
—	—	—	TIMESTAMP
—	—	—	BIT
—	—	—	SYSNAME
—	—	—	MONEY

### Nulls

If a database row lacks a value for a particular column, that column is said to be *null*, or to contain a null. Nulls are used in columns of any datatype when the actual value is unknown or when another value would not be meaningful.

A null is not equivalent to zero. In many databases, nulls are also not equivalent to zero-length strings, although Oracle7 Servers and Blaze databases currently treat zero-length strings as null.

Many other components of SQL, such as operators, functions, and conditions, have special rules associated with nulls. In general, an operator or non-group function given a null argument returns null (however, logical operators have special rules associated with nulls, as do special-purpose operators and functions such as IS NULL or NVL()). When the test in a condition returns null, the value of the condition is unknown, which is treated the same as FALSE.

For specific rules about how nulls are handled by your database, see the documentation accompanying it.

### Objects

Database objects are named structures within the database that store and organize information. Common types of database objects are *tables*, *views*, *indexes*, *sequences*, and *synonyms*. Database objects are described in Chapter 8, "Database Objects".

You frequently refer to database objects and parts of database objects in the text of SQL statements. For example, when updating a column value, you specify both the name of the table object and the name of the column. In general, you refer to database objects and parts by specifying the object or part name assigned at creation. Optionally, you can also specify the object's location—the schema (or, for SQL Server, the database and owner) containing the object.

For example, to specify the ENAME column in the EMP table, you could use the following reference:

```
EMP . ENAME
```

---

The following table summarizes the syntax for referring to objects and parts in each type of database supported by Oracle Power Objects:

<b>Database Type</b>	<b>Syntax</b>
Blaze	[ schema . ] object [ . part ]
Oracle7	[ schema . ] object [ . part ] [ @dblink ]
SQL Server	[ database . ] [ owner . ] object [ . part ]

In databases where object names can contain special characters (such as Oracle7 Servers and Blaze databases), you must use double quotes to delimit object names containing special characters. You must also use double quotes with these databases to preserve case information in object names.

For example, the following reference to the ENAME column in the "Employee\_Names(1)" table requires double quotes because it contains special characters:

```
"Employee_Names(1)".ENAME
```

Note that each individual object or part name must be enclosed in double quotes separately.

## Literals

A *literal* (also called a *constant*) is a fixed data value. You can use literals to specify information to be inserted into a table, or in arguments to operators, functions, or commands.

### Numeric Literals

Numeric literals are composed of digits. For numbers with fractional parts, a decimal point is used to separate the integer part from the fractional part. You can also use a sign (+ or -) at the beginning of the number. Scientific notation—indicated by the character "e" or "E"—is also allowed by all databases currently supported by Oracle Power Objects.

The following values are numeric literals:

```
42
3.14
-10
+25
3E10
2.8e-32
```

## Text Literals

Text literals are strings of text, which can include letters, numbers, spaces, tabs, and other special characters. You must always delimit (surround) text literals with single quotation marks. The following values are text literals:

```
'Smith'  
'The correct response is 42'  
'Coming & Going'
```

➤ **Note:** The standard string delimiter for SQL is different from the string delimiter in Oracle Basic. In Oracle Basic, literal strings are delimited by double quotation marks.

You must use a special convention to indicate single quote characters within text literals. For Oracle7, Blaze, and SQL Server databases, you use two consecutive single quotes to represent a single quotation mark inside a literal, as in the following example:

```
'She hasn''t yet arrived.'
```

## Date and Time Literals

The databases supported by Oracle Power Objects do not have a separate convention for date and time literals. However, these databases allow you to specify a date or time value using a text literal containing information stored in one or more *default formats*. These databases also provide additional ways to enter date values (for example, Oracle7 Servers and Blaze databases support the TO\_DATE function, which allows you to enter date values in a wide variety of formats).

For information about entering date and time literals in your database, see the documentation accompanying it.

## Nulls

Most databases let you explicitly specify a null value explicitly by using the word NULL.

## Other Literals

Some databases support additional conventions for literal values associated with special datatypes. For example, SQL Server databases allow you to specify monetary values by preceding a numeric value with a dollar sign (\$).

For information about other literals supported by your database, see the documentation accompanying it.

---

## Operators

*Operators* perform operations on individual values and return a result. The values manipulated by the operator are called *operands*. Operators fall into one of two general classes:

A **unary** operator operates on only one operand. A unary operator typically appears with its operand in the following format:

operator operand

A **binary** operator operates on two operands. A binary operator appears with its operands in this format:

operand1 operator operand2

Other operators with special formats accept more than two operands. Most operators return null when given a null operand, although certain operators have special rules regarding nulls.

### Precedence of Operators

When an expression contains multiple operators, the operators are evaluated in a fixed sequence according to their *precedence*. For example, the multiplication and division operators (\* and /) are evaluated before the addition and subtraction operators (+ and -).

You can use parentheses in an expression to override operator precedence. Expressions inside parentheses are evaluated before those outside parentheses.

### Categories of Operators

The following table lists common categories of operators:

Category	Description	Examples	Operation
Arithmetic	Manipulate numeric operands.	+	Addition
		-	Subtraction
		/	Division
		*	Multiplication
Character	Manipulate character string operands.		Concatenation
Comparison	Used in conditions that compare one expression with another.	=	Equality test
		!=	Inequality test
		>	“Greater than” test
		<	“Less than” test
		>=	“Greater than or equal to” test
		<=	“Less than or equal to” test
		IS NULL	Tests for nulls

Category	Description	Examples	Operation
Logical	Manipulate comparisons or other logical expressions.	NOT AND OR	Logical NOT (negation) Logical AND (intersection) Logical OR (union)
Set	Combine the results of two queries into a single result.	UNION  INTERSECT  MINUS	Returns all distinct rows selected by either query  Returns all distinct rows selected by both queries  Returns all distinct rows selected by the first query but not the second
Other	Some databases support other categories of operators. For example, SQL Server databases support bitwise operators.		

## Functions

A *function* performs calculations and returns a result. Functions usually operate on one or more *arguments* that you specify when you call the function. Arguments are included in parentheses following the function name.

For example, the ABS function returns the absolute value of its numeric argument. The function and its argument can be specified as follows:

```
ABS (-10)
```

The list of available functions varies from database to database. For a list of functions supported by your database, see the documentation accompanying it.

## Expressions

Expressions represent values. Expressions can consist of the following language components, singly or combined by operators:

- Literals
- Object values
- Functions

---

The following items are expressions:

```
42
ename
'SMITH'
12 * 12
NVL(sal, 0)
UPPER(ename) || ' works in department ' || TO_CHAR(deptno)
```

## Conditions

A *condition* (also called a *search condition*) is a combination of one or more expressions and logical operators that evaluates to TRUE, FALSE, or unknown. A condition could be said to be a value of a Boolean (logical) datatype, although SQL does not formally support such a datatype.

A condition frequently includes an object value compared against another value with a relational operator. For example, the following condition compares a value from the ENAME column against a literal string:

```
ename = 'SMITH'
```

Another common condition tests whether the value in a column is null. The IS NULL and IS NOT NULL operators perform this test. For example, the following condition tests whether the value in the COMM column is null. It returns True if the value is null, False otherwise.

```
comm IS NULL
```

You use only IS NULL and IS NOT NULL to test for the presence of nulls, because most other operators return Null when operating on a null operand. When the test in a condition returns null, the value of the condition is unknown, which is treated the same as False. For example, the following condition always evaluates to unknown:

```
comm = NULL
```

Conditions can be extended using Boolean operators such as AND, OR, and NOT. For example, the following condition requires that two individual conditions be True to return True:

```
deptno = 10 AND sal > 2000
```

Conditions are typically used in the WHERE and HAVING clauses of DML statements (described in the next section), although your database might support the use of conditions in additional locations. For example, the following SELECT statement uses a condition in a WHERE clause to specify that only employees having a job of “MANAGER” should be returned:

```
EXEC SQL SELECT ename, hiredate FROM emp WHERE job = 'MANAGER'
```



## Commands

The most commonly executed types of commands can be grouped into these categories:

- **Data Definition Language (DDL) commands.** Apply to the definition of a database object. They include creating, deleting, and modifying the structure of database objects.
- **Data Manipulation Language (DML) commands.** Apply to the data stored in or accessible through a database object. They include querying, inserting, updating, and deleting data rows. Data manipulation operations apply mainly to tables and views, although they are sometimes used with other database objects such as sequences.
- **Transaction Processing Language (TPL) commands.** Apply to the flow of a database transaction, allowing data manipulation operations within a transaction to be committed (made permanent) or rolled back (discarded).

Many databases support additional categories of commands. For more information about the categories of command supported by your database, see the documentation accompanying it.

### Data Definition Language Commands

The following DDL commands are used by most SQL databases:

Command	Description
CREATE <i>object</i>	Creates a database object such as a table, view, index, sequence, or synonym.
ALTER <i>object</i>	Modifies the definition of a database object.
DROP <i>object</i>	Removes a database object from the database.
GRANT	Grants database privileges to a user.
REVOKE	Revokes database privileges from a users.

### Data Manipulation Language Commands

The following DML commands are used by most SQL databases:

Command	Description
SELECT	Retrieves data from one or more tables or views
INSERT	Adds rows to a table or to a view's base tables.
UPDATE	Changes existing rows in a table or in a view's base tables.
DELETE	Removes rows from a table or from a view's base tables.

---

## Transaction Processing Language Commands

A *transaction* is a logical unit of work that is composed of one or more SQL statements. A transaction is an *atomic* unit; that is, the effects of all the SQL statements that constitute a transaction are treated as a unit and can either all be *committed* (applied to the database) or all *rolled back* (undone from the database).

The following TPL commands are used by most SQL databases:

<b>Oracle7 or Blaze Command</b>	<b>SQL Server Command</b>	<b>Description</b>
COMMIT	COMMIT TRANSACTION	Ends the current transaction, making permanent all its changes to the database.
ROLLBACK	ROLLBACK TRANSACTION	Discards work done in the current transaction.
SAVEPOINT	SAVE TRANSACTION	Identifies a point in a transaction to which you can later roll back.

## Procedural Extensions

*Procedural extensions* enhance SQL by providing procedural capabilities. These extensions typically allow you to mix SQL statements with procedural constructs. Although some common conventions exist, the types of procedural extensions available depend on the type of database you are using.

**PL/SQL** is Oracle's procedural language extension to SQL; it is available on Oracle7 Servers that include the procedural option. (Blaze databases do not currently support PL/SQL). Using PL/SQL, you can define and execute program units such as procedures, functions, and packages. PL/SQL includes support for variables and constants, cursors, exception handling, and flow-of-control statements such as loops.

**Transact-SQL** is an extended version of SQL available on SQL Server databases. Transact-SQL includes support for triggers and stored procedures, error-handling capabilities, and a control-of-flow language.

For a complete reference to the procedural extensions available for your database, see the documentation accompanying it.

## Executing SQL Statements

You execute explicit SQL statements in order to take advantage of database features not supported by the Oracle Power Objects interface. For example, you can execute SQL statements to create database objects such as snapshots or roles.

You can execute a fully formed SQL statement using the Oracle Basic EXEC SQL command or SQLLOOKUP function.

The **EXEC SQL** command sends a complete SQL statement to a database for execution. EXEC SQL allows you to specify and return values using *bind variables*.

The **SQLLOOKUP** function sends a complete SQL query (a SELECT statement) to a database for execution and returns a single value.

EXEC SQL and SQLLOOKUP are described below.

### The EXEC SQL Command

The Oracle Basic EXEC SQL command sends a complete SQL statement to a database for execution. EXEC SQL can also execute PL/SQL blocks (for Oracle7 Servers) and Transact-SQL statements such as system procedures (for SQL Server databases).

EXEC SQL takes the SQL statement to execute as its single argument, which you can specify either as an *unquoted* literal string or as a string variable that contains the SQL statement.

For example, you could issue a DELETE statement with the following command:

```
EXEC SQL DELETE FROM emp WHERE ename = 'SMITH'
```

The preceding command could be rewritten as follows:

```
DIM vSqlString AS String
vSqlString = "DELETE FROM emp WHERE ename = 'SMITH'"
EXEC SQL :vSqlString
```

To execute a SQL statement that continues over several lines, use an ampersand (&) at the end of a line to indicate that the statement continues on the next line. For example, you could rewrite the preceding statement as follows:

```
EXEC SQL DELETE FROM emp &
WHERE ename = 'SMITH'
```

You cannot use an ampersand to break a line in the middle of a quoted string value.

---

## Using Bind Variables

*Bind variables* enable you to provide input to or receive output from certain SQL statements. A bind variable is simply an Oracle Basic variable referenced in a SQL statement. You refer to the bind variable by preceding the variable name with a single colon (:). For example, the following DELETE statement uses the bind variable `vEmpName` to provide an input value:

```
EXEC SQL DELETE FROM emp WHERE ename = :vEmpName
```

In the following SELECT statement, the bind variable `vEmpNum` receives an output value:

```
EXEC SQL SELECT empno INTO :vEmpNum FROM emp &
WHERE ename = 'SMITH'
```

You cannot use bind variables when the SQL statement is specified as a string variable. To use a bind variable, you must pass an unquoted literal string as the argument to EXEC SQL.

➤ **Technical Note:** Bind variables are not allowed in string variable arguments to EXEC SQL because Oracle Power Objects resolves bind variable references at compile time. As a result, Oracle Power Objects must be able to examine the SQL statement when your application is compiled.

The following table shows examples of the most common locations where you can use bind variables:

Command	Locations	Example
DELETE	WHERE clause	EXEC SQL DELETE FROM emp & WHERE ename = :vEmpName
INSERT	VALUES clause	EXEC SQL INSERT INTO dept & (deptno, dname, loc) & VALUES (:vDeptNum, :vDeptName, & :vLocation)
SELECT	INTO clause	EXEC SQL SELECT ename INTO :vEmpName & FROM emp
	WHERE clause	EXEC SQL SELECT ename INTO :vEmpName & FROM emp WHERE empno = :vEmpNum
UPDATE	SET clause	EXEC SQL UPDATE emp & SET sal = sal + :vRaise
	WHERE clause	EXEC SQL UPDATE dept & SET loc = 'PHILADELPHIA' & WHERE deptno = :vDeptNum

For Oracle7 Servers, you can use bind variables in any location listed in the “Embedded SQL” command descriptions in the *Oracle7 Server SQL Language Reference Manual*. You can also use bind variables in PL/SQL blocks.

You cannot use bind variables to specify the names of database objects or parts. For example, the following method code is not valid:

```
DIM vColName AS String
vColName = "ENAME"
'The following statement is invalid
EXEC SQL SELECT :vColName INTO :vEmpName FROM EMP
```

A bind variable must be an Oracle Basic variable. You cannot use the **Value** property of an object as a bind variable. To use the **Value** property (or another object property), first assign it to an Oracle Basic variable, then use the Oracle Basic variable in the SQL statement. For example, the following method code inserts values derived from text fields into the DEPT table:

```
vDeptNum = fldDeptno.Value
vDeptName = fldDname.Value
vDeptLoc = fldLoc.Value
EXEC SQL INSERT INTO dept VALUES (:vDeptNum, :vDeptName, &
:vDeptLoc)
```

For output values, you can assign the value of an Oracle Basic variable to the **Value** property of a control after executing the SQL statement. For example, the following method code selects values from the DEPT table into text fields:

```
EXEC SQL SELECT dname, loc INTO :vDeptName, :vDeptLoc &
FROM dept WHERE deptno = 10
fldDname.Value = vDeptName
fldLoc.Value = vDeptLoc
```

Oracle Power Objects lets you use bind variables with any supported database. For databases that have native bind variables support (such as Oracle7 Servers and Blaze databases), Oracle Power Objects uses the native support. For databases that do not support native bind variables (such as SQL Server databases), Oracle Power Objects simulates the behavior of bind variables by replacing bind variable references with literal values just before the statement is sent to the database.

### Datatypes and Declaration of Bind Variables

A bind variable used for *input* must be declared (either implicitly or explicitly) in your method code prior to being referenced. The variable must be of a datatype that can be converted to the required datatype.

A bind variable used for *output* does not have to be declared prior to being referenced. If the variable is already declared, it must be of a datatype that can receive the output data. If an undeclared variable is used to receive output, Oracle Power Objects types the variable automatically according to the kind of data it receives. However, if the variable has an explicit type suffix, the variable's datatype is determined by the suffix.

---

## Using Bind Variable Arrays

You can use array variables to receive output from queries (SELECT statements). You must declare the variable as an array *before* using it as a bind variable. For example, the following statements use the variables `vEmpNum` and `vEmpName` to receive the result set of a query:

```
STATIC vEmpNum(14) AS Integer
STATIC vEmpName(14) AS String
EXEC SQL SELECT empno, ename INTO :vEmpNum, :vEmpName &
FROM emp
```

When a query returns multiple rows, each element of the array receives a value from a different database row: element 0 receives the value from the first row, element 1 receives the value from the second row, and so on. Values returned beyond the last element in the array are discarded. If the array's dimension exceeds the number of values returned, array elements beyond the last value returned are not modified. You can determine the number of values returned by a single query using the Oracle Basic `SQLROWCOUNT` function.

You can use both array and non-array bind variables in the same EXEC SQL statement. You can also use arrays of different sizes. Each bind variable will be filled with as many rows as it can hold.

You cannot use arrays to provide multiple input values to SQL statements. You can, however, use an individual element of an array to provide an input value, as in the following example:

```
STATIC vEmpNum(14) AS Integer
STATIC vEmpName(14) AS String
vEmpNum(1) = 7369
vEmpName(1) = "STEVENS"
EXEC SQL UPDATE emp SET ENAME = :vEmpName(1) &
WHERE EMPNO = :vEmpNum(1)
```

## Directing a Statement to a Session

By default, an EXEC SQL command is directed to the *default session*. The default session is designated by the **DefaultSession** property of the application object from which the EXEC SQL command is executed. If no default session is defined, EXEC SQL returns an error when you attempt to execute a statement directed to the default session.

You can indicate explicitly the session to which an EXEC SQL command is directed by including the AT clause in the SQL statement, as shown in the following example:

```
EXEC SQL AT session1 DELETE FROM emp WHERE ename = 'SMITH'
```

The AT clause specifies either the name of the session object or a bind variable containing a reference to the session object, as in the following example:

```
DIM sesForm1 AS Object
sesForm1 = Form1.GetRecordSet().GetSession()
EXEC SQL AT :sesForm1 DELETE FROM EMP WHERE ename = 'SMITH'
```

The variable must be of type *Object* and contain a reference to the session object - the variable cannot contain the name of the session object.

### Retrieving Result Information

Each EXEC SQL command you execute returns result information, which you can retrieve using Oracle Basic functions. The following information is available:

**Row Count.** The number of database rows that were successfully processed by the SQL statements. For example, after a SELECT statement, the row count indicates the number of rows that were returned by the query.

**Result Code.** An integer indicating whether the SQL statement executed successfully. A result code of zero indicates success; a nonzero result code indicates an error. Different error numbers are returned for different types of errors. The exact number returned depends on the database that processed the statement. Therefore, a database-independent application should not rely on receiving a particular result code.

**Result Message.** A text string that accompanies the result code. In the case of an error, the result message provides an explanation of the error encountered. The text of the message is derived from the database that processed the statement. Therefore, a database-independent application should not rely on receiving a particular result message.

**Error Class Code.** An integer representing a database-independent Oracle Power Objects error classification. While result codes vary from database to database, the same error class code is returned for the same type of error for all database types.

You can use the following Oracle Basic functions to retrieve this information:

Type of Information	Function
Row count	SQLROWCOUNT
Result code	SQLERRCODE
Result message	SQLERRTEXT
Error class code	SQLERRCLASS

By default, Oracle Power Objects does not provide any visual indication when an EXEC SQL command encounters an execution error. To have Oracle Power Objects automatically display an error dialog box when an EXEC SQL command encounters an error, you can execute the WHENEVER command.

The following method code enables the automatic display of error dialog boxes:

```
EXEC SQL WHENEVER SQLERROR RAISE
```

The following method code disables the automatic display of error dialog boxes:

```
EXEC SQL WHENEVER SQLERROR CONTINUE
```

---

The WHENEVER command applies to all EXEC SQL commands executed, regardless of database session or type. Therefore, you do not specify an AT clause with the WHENEVER command.

### Executing Procedural Extensions

You can use the EXEC SQL command to execute procedural extensions supported by your database, such as stored procedures and functions. Any SQL statement you execute with EXEC SQL is passed through to the database without modifications (except as noted in the section “Evaluation of SQL Statements” on page 9.21). Therefore, you can execute any command or statement that can be processed by your database.

For example, you could execute a PL/SQL block with the following method code:

```
EXEC SQL                                     &
BEGIN                                       &
    CREATE TABLE temp (id NUMBER);        &
    FOR i IN 1..100 LOOP                   &
        INSERT INTO temp(id) VALUES (i); &
    END LOOP;                               &
    COMMIT;                                 &
END;
```

You can also use PL/SQL to execute stored functions and procedures. For example, the following method code creates and executes a stored procedure named CREDIT:

```
'Create the procedure
EXEC SQL CREATE OR REPLACE PROCEDURE credit &
    (acc_no IN NUMBER, amount IN NUMBER) AS &
BEGIN                                       &
    UPDATE accounts SET balance = balance + amount&
    WHERE account_id = acc_no;           &
END;
'Execute the procedure
EXEC SQL                                     &
BEGIN                                       &
    credit(1, 100);                       &
    COMMIT;                               &
END;
```

For SQL Server databases, you can use EXEC SQL to execute system procedures. For example, the following method code executes the **sp\_primarykey** procedure to define a primary key for the EMP table:

```
EXEC SQL sp_primarykey EMP, EMPNO
```



You can also use EXEC SQL to execute Transact-SQL statements with procedural constructs, as in the following example:

```
EXEC SQL IF EXISTS (SELECT zip FROM authors &
WHERE zip = 94705) &
INSERT INTO local_authors &
SELECT * FROM AUTHORS WHERE zip = 94705
```

In many cases, you can execute procedural database operations without having to use procedural SQL code. Oracle Power Objects enables you to execute many types of procedural operations by mixing Oracle Basic code (such as FOR... NEXT loops) with EXEC SQL statements.

For example, the following method code inserts a set of sequential values into the ID column of a table named TEMP:

```
EXEC SQL CREATE TABLE temp (id NUMBER)
FOR i = 1 TO 100
EXEC SQL INSERT INTO temp(id) VALUES (:i)
NEXT i
EXEC SQL COMMIT
```

### Evaluation of SQL Statements

Oracle Power Objects does not evaluate SQL statements—it simply sends them to the database for execution. However, Oracle Power Objects does examine the text of SQL statements to perform the following tasks:

- Reassemble SQL statements that are broken over multiple lines of text.
- Evaluate the AT clause to identify the database session object to which the statement should be sent.
- Evaluate bind variables.

### The SQLLOOKUP Function

The Oracle Basic SQLLOOKUP function sends a complete SQL query (a SELECT statement) to a database for execution. The first value returned by the query is the return value of SQLLOOKUP. Any additional values are ignored.

SQLLOOKUP takes the SQL statement to be executed as its argument, which you can specify as a normal string value. An optional first argument identifies the database session object to which the SQL statement should be sent. If the first argument is omitted, the statement is sent to the default session.

For example, the following statement determines the number of rows in the EMP table and stores the value in the variable vNumRows:

```
vNumRows = SQLLOOKUP("SELECT COUNT(*) FROM emp")
```

---

The following statement executes the same query, but explicitly directs it to the session object "Session1":

```
vNumRows = SQLLOOKUP(Session1, "SELECT COUNT(*) FROM emp")
```

### Derived Values and Lookup Fields

The SQLLOOKUP function is particularly useful in *derived values* (Oracle Basic expressions used in the **DataSource** property of bindable control). SQLLOOKUP can "look up" a single value from a detail table without requiring you to create a master-detail relationship between two containers.

For example, the following derived value "looks up" a department name value based on the department number value displayed in the field "fldDeptNum". This derived value appears in the **DataSource** property of a text field object.

```
=SQLLOOKUP("SELECT dname FROM dept WHERE deptno = " +  
STR(fldDeptNum.Value))
```

➤ **Note:** The use of the + concatenation operator is explained below.

The return value of SQLLOOKUP is of a datatype determined by the information returned. The appropriate datatype is chosen by the database driver (the .POD file) that provides connectivity to the database executing the SQL statement.

You cannot use bind variables with the SQLLOOKUP function. To include input values, you must concatenate the value into the SQL string to be executed. For example, the following method code uses the variable vEmpNum to identify an employee number in the WHERE clause of a query:

```
DIM vEmpNum AS Integer  
DIM vEmpName AS String  
vEmpNum = 7788  
vEmpName = SQLLOOKUP("SELECT ename FROM emp WHERE empno = " +  
STR(vEmpNum))
```

When you construct a SQL statement by concatenating values, you must ensure that the resulting SQL statement will be valid. For example, consider the following SQLLOOKUP statement:

```
vEmpName = SQLLOOKUP("SELECT empno FROM emp WHERE ename = " &&  
vEmpName)
```

If vEmpName contains an unquoted value, an invalid SQL statement will result. For example, if vEmpName contains "Bob Smith", the following SQL statement will be sent to the database:

```
SELECT empno FROM emp WHERE ename = Bob Smith
```

The preceding statement is invalid because quotes are required around literal strings. Similarly, if empName contains Null, the following SQL statement will be sent to the database:

```
SELECT empno FROM emp WHERE ename =
```

The preceding statement is invalid because the condition in the WHERE clause is improperly terminated.

For string values, you can ensure that the statement is valid by concatenating SQL string delimiter characters (single quotation marks) around the values. For example, you could rewrite the preceding SQLLOOKUP statement as follows:

```
vEmpNum = SQLLOOKUP ( "SELECT empno FROM emp WHERE ename = '" &&
    vEmpName & "'" )
```

Note that if a value in `vEmpName` contained a single quote character, an invalid statement would result. Single quote characters in SQL text literals must be represented with two consecutive quotes.

For values of other datatypes, and for single-word string values, you can ensure that Nulls are handled correctly by using the `+` concatenation operator instead of the `&` operator. While the `&` operator treats a null argument as a zero-length string, the `+` operator returns Null if one of its arguments is Null. The following table demonstrates the difference between `&` and `+`:

<b>Expression</b>	<b>Value</b>
"Hello" & NULL	"HELLO"
"Hello" + NULL	NULL

If the SQL statement argument to SQLLOOKUP is Null, SQLLOOKUP returns Null and does not send the SQL statement to the database. For example, in the following method code, if the variable `empNum` is Null, the SQL statement is not executed:

```
vEmpName = SQLLOOKUP ( "SELECT ename FROM emp WHERE empno = " +&
    STR ( vEmpNum ) )
```

You should always use this technique when using SQLLOOKUP in a derived value. Doing so prevents errors from occurring due to null values.



# 10

---

## **Applications and Application Objects**

This chapter covers the following topics:

Overview .....	10.2
Application Object Types .....	10.4
Types of Containers .....	10.7
Controls and Static Objects .....	10.11
Types of Controls .....	10.16
Types of Static Objects .....	10.30
Interacting with Application Objects .....	10.31

---

## Overview

As described in Chapter 1, “Application Development with Oracle Power Objects”, an *application* provides the front end of your application, providing the means to query and enter data through the user interface. Applications contain several kinds of application objects, all of which are described in this chapter.

From an object-oriented standpoint, the application is itself a container object, with its own set of properties and methods. You can use the Oracle Basic commands for navigating through the object containment hierarchy to access the application at the top of this hierarchy.

## Applications and File Objects

An application is a file object, in the sense that stores its description (and that of all the application objects within it) in a file within the operating system. The file does not contain descriptions of recordset objects, since these special types of front-end objects are instantiated at run time only.

During design time, the file has the extension .POA; however, you can later compile it into an executable file. The compiled application file object then has a .PO or .EXE extension, depending on the type of executable you create. During the compile, Oracle Power Objects includes objects defined within libraries (.POL files) and sessions (.POS files) that the application uses. Note that these extensions are required only in Windows; the Macintosh version of Oracle Power Objects does not require filename extensions for applications, libraries, and sessions.

The application does contain binding information needed for database connectivity. These include references to record sources (tables and views) as well as the sessions through which they are accessed. Therefore, an application uses information defined in a session object to gain access to database objects.

In addition, the application can contain objects defined in libraries, including user-defined classes and bitmaps.



For more information on compiling an application, see Chapter 16, "Compiling the Executable Application".

For an application to work effectively with library objects and database objects, the libraries and sessions containing the descriptions of these objects must appear in the Main window during development. When you compile an application, these components are then added to the executable file.

### Application Naming Rules

The **Name** property of an application must follow several rules. For a complete list of these naming requirements, see the section "Naming Rules" on page 3.20.

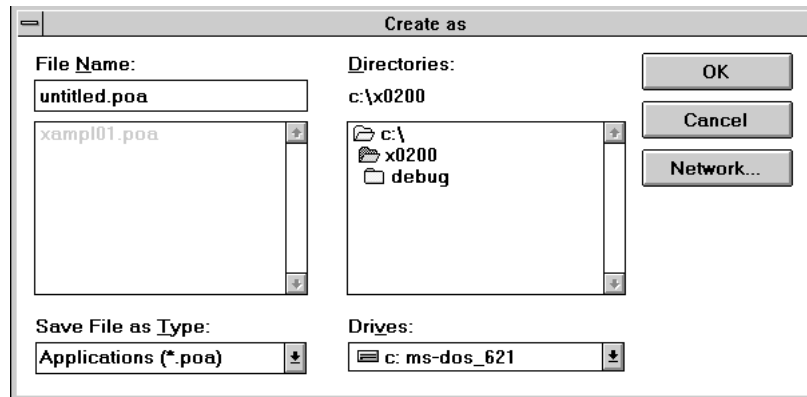
### Creating a New Application

☆ **To create a new application:**



- 1 In the Main window, choose the **File-New** menu command, or by clicking the New Application button.

The standard new file dialog box for your operating system appears.



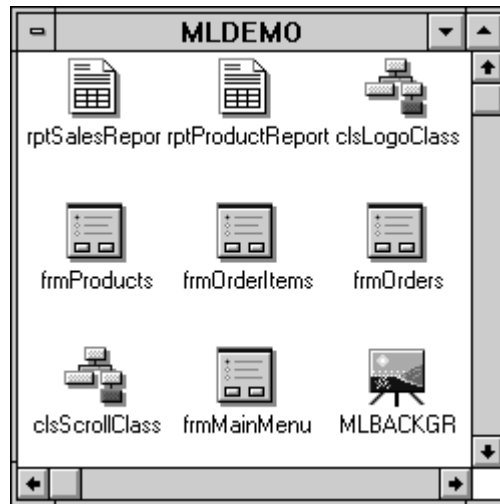
- 2 Assign a new filename to the application file, which holds the complete description of the application and click **OK**.

A window for the new application then appears. You can start defining the properties of the application, as well as add new application objects to it.

---

## Application Object Types

Applications contain many kinds of application objects. Some appear in the Application window, while others can only appear within a container.



The following table summarizes all general types of application objects, and notes where they can appear in the application:

Object	Description
<b>Containers</b>	Application objects that can contain other objects. Many containers are bindable, meaning that they can display records queried from a database. When a container is bound, it has an associated recordset object. Containers include forms, reports, embedded forms, repeater displays, rectangles, and ovals, as well as the application itself. Containers appear as icons in the Application window.
<b>Controls</b>	Objects designed to display data or give the user the ability to take actions through the user interface. Controls include text fields, radio buttons, radio button frames, pushbuttons, list boxes, combo boxes, popup lists, picture objects, scrollbars, check boxes, and current row pointers. Controls appear only within a container.
<b>Static Objects</b>	Display-only objects designed to organize and highlight other application objects appearing within a container. Static objects include lines, rectangles, ovals, and static text. Static objects appear only within a container.



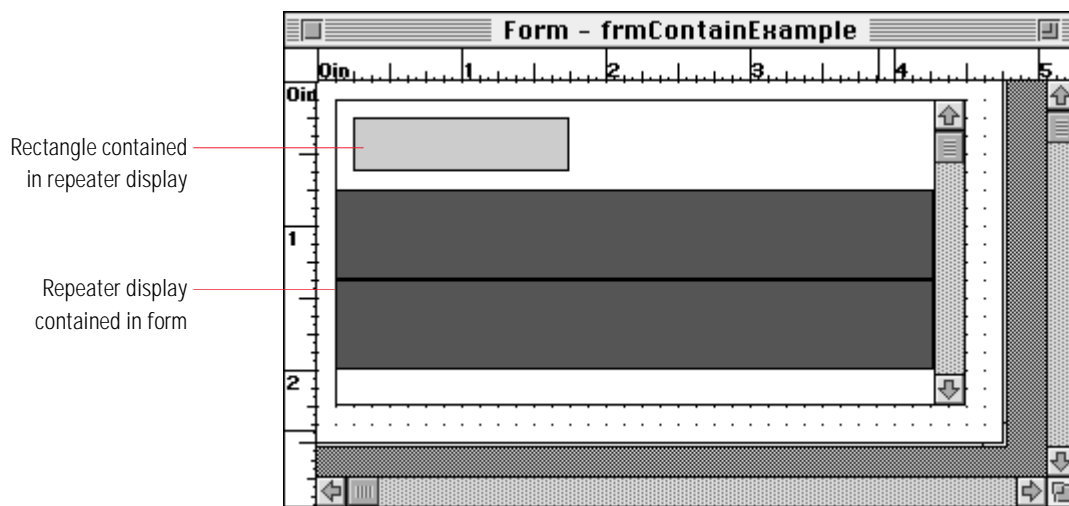
<b>Object</b>	<b>Description</b>
<b>OLE Objects</b>	Objects defined through the object linking and embedding (OLE) technology developed by Microsoft. Applications other than the Oracle Power Objects application in which they appear provide the interface for viewing and editing OLE objects. OLE objects can appear within the Application window, as well as within a container. However, OLE objects within the Application window can only be used within an application if they are dragged and dropped into a container.
<b>Bitmaps</b>	Graphic images imported into Oracle Power Objects as an application resource, and capable of being displayed on different application objects. As with OLE objects, bitmaps appear in the Application window, but they can also be added to application objects (for example, forms, pushbuttons).
<b>User-defined classes</b>	Sets of controls designed as reusable application objects. Once you create a user-defined class, it appears in the Application window, available for use within other containers. When you drag and drop a user-defined class into a container, you create an instance of the class that inherits the properties and methods of the master class definition. For more information on user-defined classes, see Chapter 13, "Classes".

The following sections describe each of these types of objects in detail. For more information on the properties and methods associated with each kind of application object, consult the online help.

---

## Containers

As described in Chapter 3, “Objects”, a *container* is an application object that can hold other application objects within it, including other containers. For example, a form can contain a repeater display, which in turn can contain text fields and other controls. Containers can be nested within other containers: For example, a form can contain a repeater display, which itself can contain a rectangle.



The relationship between containers and the objects within them is called the *object containment hierarchy*. The lower levels of the hierarchy include objects contained within other objects; the upper levels include forms, reports, and the application itself, all of which contain other objects.

Oracle Power Objects includes several commands and object identifiers for gaining a reference to objects within the object containment hierarchy (called *relative references*). These references can be resolved “upwards”, to the containers in which an object appears, or “downwards”, to the objects appearing within a container. These techniques are often needed to resolve properties of objects within the containment hierarchy, or call methods on these objects.

## Bindable Containers

Some containers are *bindable*, meaning that they can display records queried from the database. In some cases the user can enter changes to the recordset through the bound container. Bound containers are connected to a record source (a table or view) accessed through a session object.

For more information on relative references, see the section “Relative References” on page 3.25.

For more information on binding, see Chapter 17, “Binding a Container to a Record Source”.

After connecting (or *binding*) the container to a record source, you must add bound controls to the container to display values, as well as give the user the opportunity to edit the contents of a record. Controls are connected to a particular column in the container's record source, identified through the control's `DataSource` property.

## Types of Containers

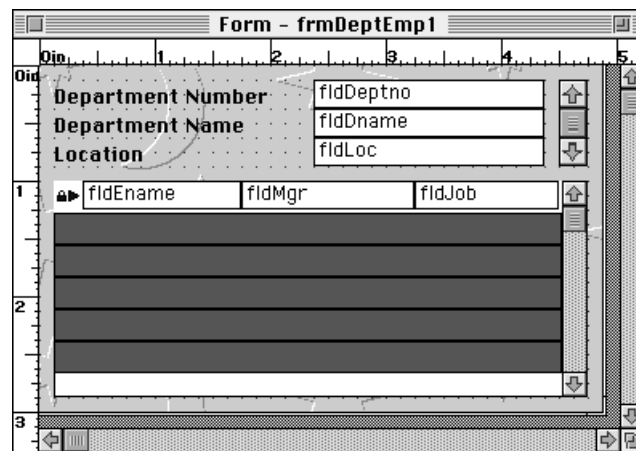
The types of containers used in Oracle Power Objects applications include:

### Forms

Forms commonly provide the core of the user interface in an application. Through standard forms as well as dialogs (a kind of form), the user performs the following tasks:

- **View data.** When a form is bound, the data can be queried from the database.
- **Enter data.** The user can insert, delete, or update records in the form's recordset.
- **Navigate to other parts of the application.** The form can contain controls used to open or close other forms or reports.
- **Set options applied to the entire application.** For example, you might create a dialog allowing the user to change the font used in the application.
- **Make links to other applications.** OLE objects appearing in a form can launch the application in which the OLE object's data is defined. In addition, you can use a form to call procedures defined in dynamic link libraries (DLLs), such as the Windows APIs.

A form can contain a wide variety of objects, including other containers, static objects, controls, bitmaps, and OLE objects. In addition, a bitmap can appear as the background for the form, as shown in the following figure:



During development, you often use several forms to represent the same set of data from different perspectives, or at different levels of detail. For example, a human resources form may present the same information about departments and employees in two different ways.

The image shows two overlapping windows titled "Departments and Employees".

The top window displays the following data:

Department Number	20				
Department Name	RESEARCH				
Location	DALLAS				
Employee Name	SMITH	Employee Number	7902	Job	CLERK
Employee Name	JONES	Employee Number	7839	Job	
Employee Name	SCOTT	Employee Number	756	Job	
Employee Name	ADAMS	Employee Number	7782	Job	
Employee Name	FORD	Employee Number	756	Job	

The bottom window displays the following data:

Department Name	ACCOUNTING	Location	NEW YORK
Department Name	RESEARCH	Location	DALLAS
Department Name	SALES	Location	CHICAGO
Department Name	OPERATIONS	Location	BOSTON
Employee Number	7782		
Employee Name	CLARK		
Manager	7839		
Job	MANAGER		
Hire Date	6/9/81		
Commission			
Salary	2450		

For more information on modality and window styles, see Chapter 11, "Forms".

Intrinsic to any form is its window, the border appearing around all or part of the form. Windows are often movable or resizable, depending on the *modality* or *window style* of the form.

## Reports

For more information on reports, see Chapter 12, "Reports".

Like a form, a report contains other application objects, many of which display data. Unlike a form, you cannot interact with the controls appearing on the form, nor can you edit the data appearing on the form. The report can display all or some of the data contained in the table or view designated as the report's record source, depending on the query conditions applied to report's recordset. When you preview or print the report, the application generates enough pages to display all records queried.

## Embedded Forms

An embedded form is a form-like application object placed within another container (normally, a form) that has many of the same properties as a form. Like forms, embedded forms are bindable.

As with forms, you cannot view multiple records in an embedded form at the same time, as you can in a repeater display.

Embedded forms are useful in the following situations:

- You want to review detail records one at a time, instead of viewing multiple detail records in a repeater display.
- You want to display a distinct set of information (for example, customer information) in one section of the form, and all of this data is part of the same record (to continue our example, a customer record from the CUSTOMER table).

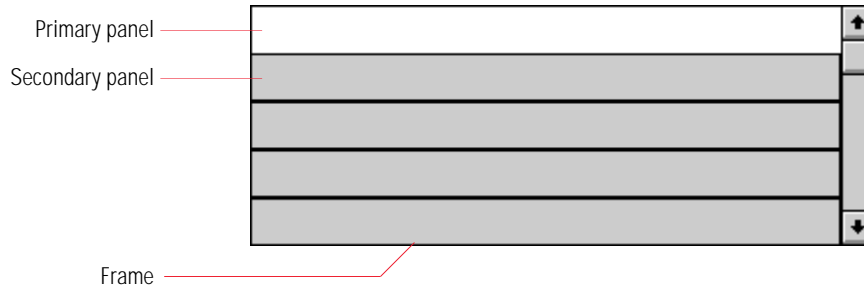
The screenshot shows a window titled "Form - frmCustomer". Inside, there is a main form with a grid-like structure. The top section has three rows: "Item Code" with field "fldItemCode", "Description" with field "fldItemDesc", and "Cost (US \$)" with field "fldCost". Below this is a section labeled "1" containing a sub-form with three rows: "Name" with field "fldName", "Contact" with field "fldContact", and "Telephone" with field "fldTelephone". Below that is a section labeled "2" containing two rows: "Amount Purchased" with field "fldAmount" and "Last Purchased" with field "fldLast". At the bottom of the sub-form is a section labeled "3" with a horizontal bar containing navigation icons. The main form also has a vertical bar on the right with navigation icons and a horizontal bar at the bottom with navigation icons.

## Repeater Displays

A repeater display is a bindable container capable of showing the same set of controls and static objects multiple times. When a repeater display is bound, it displays these sets of objects once per record.

---

The repeater display consists of three parts:



**Frame.** The frame surrounding the entire control. When you first click on the repeater panel, you select the frame.

**Panel.** The container displayed multiple times within the frame. To add objects to a repeater display, you add them to the primary panel. At runtime, the panel is displayed as many times as is needed to display all records in the repeater display's recordset. The other, gray panels are secondary panels, and are displayed only to show how the primary panel appears when repeated.

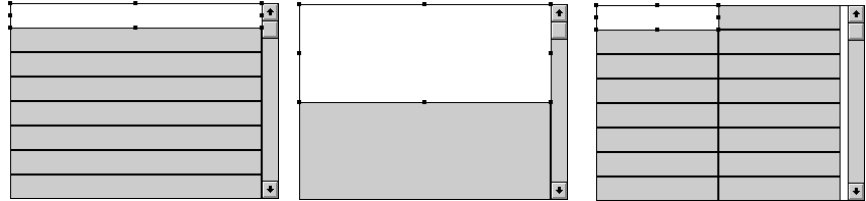
**Scrollbar.** A control that lets the user navigate through the repeated instances of the panel. The scrollbar is optional, and its presence or absence is determined by the **HasScrollBar** property. The scrollbar appears on the right side of the repeater display, regardless of the number of records appearing in the container.

## Resizing and Positioning Repeater Displays

You have several options when sizing and positioning the contents of a repeater display.

- **To display more than one panel horizontally within the repeater display's frame**, shorten the width of the primary panel until the desired number of panels appears in each row of the display.
- **To change the vertical distance between panels**, drag the gray panel directly beneath the primary panel to create the desired distance between panels.
- **To change the horizontal distance between panels**, drag the panel directly to the right of the primary panel (if more than one panel appears in a given row) to create the desired horizontal distance between panels.

- To create distance between the upper left-hand corner of the repeater display's frame and the upper left-hand corner of the topmost repeater panel, drag the primary repeater panel to the desired position.



### Repeater Displays and Aggregation Functions

You can use aggregation functions to calculate values based on controls within a repeater display. For example, you can use the `SUM()` function to add a set of numeric values displayed in a text field within the repeater display. You can use aggregation functions in two ways:

- **Calculated in Oracle Basic method code.** In this case, you refer to the contents of a control with the syntax `control_name.Value`. For example, to sum the contents of all instances of the "UnitCost" text field within a repeater display, you would use the syntax `sum(UnitCost.Value)`.
- **Set through the DataSource property.** Here, you use an aggregate function as a derived value for the control's `DataSource` property. In this case, you refer to the name of the control, not its `Value` property. In the case of the UnitCost field, you would set the `DataSource` property of a control outside the repeater display to the following:  
`=SUM(UnitCost)`

### Other Containers

Rectangles and ovals are also containers, in that they can contain lines, controls, and other containers. For more information on rectangles and ovals, see the section "Types of Static Objects" on page 10.30.

## Controls and Static Objects

As described earlier in this chapter, a control is an application object that can store data or provide the means for the user to take some action. In some cases, the control can perform both functions. For example, a text field is a control because it can store and display data, and the user can type data into it. Similarly, a pushbutton lets the user take an action (clicking the simulated button), but this type of control does not store data.

---

A static object is also an application object. Unlike a control, it does not store data, and it is not designed to give the user the means to take an action. Normally, a static object is designed for display only, to help organize and highlight other application objects appearing within a form or report. However, since static objects have several standard methods, you might design them to behave as controls. For example, you could add method code to the `Click()` method of a rectangle, so that it performs some action when the user clicks on it.

## Bindable Controls

Many controls are bindable, meaning that they can read data from a column in a table or view. In most cases, this column is part of the table or view specified as the record source for the container in which the control appears. (The record source is set through the **RecordSource** property of the container.)

In some types of objects, the value displayed is different from the value stored in the control. For example, in the case of list controls (list boxes, popup lists, and combo boxes), the values used to populate the list may come from a different table or view. When the list control stores a value, it is stored in the record source specified for the container.

Another exception is cases when the control uses the `SQLLOOKUP` function to query values. In these cases, the `SQLLOOKUP` bypasses the normal binding to the column, populating the control directly through a query. In these cases, the control is unbound, in the sense that its values are not part of the container's recordset. Even if the container and other controls appearing on it are bound, the control using `SQLLOOKUP` to populate it with values is not bound. Additionally, since its values are not part of a recordset, you cannot move among them; the application populates the control with the first value returned from the `SQLLOOKUP` query.

## Control Values

You can populate a control's value from a variety of sources, including:

- **Columns in a table** - Bound controls can read their values from the table or view to which the container holding the control is bound. You identify the column through the control's **DataSource** property.
- **User entry** - In the case of both bound and unbound controls, the user can enter values into the control.
- **Derived values** - The control's value is the result of some calculation, that can include values read from the properties of other controls (such as the **Value** property). The derived value can also use the `SQLLOOKUP` function to read a value from a table or view. You define this calculation through the control's **DataSource** property.
- **Default values** - You can assign a default value to many controls through their **DefaultValue** property.
- **Method code** - Through method code, you can assign new values to controls.

For more information on `SQLLOOKUP`, see the section "The `SQLLOOKUP` Function" on page 9.21.



- **Coded lists** - In the case of list controls, you can determine the internal and display values (see below for an explanation) through their **Translation** and **ValueList** properties.

For a description of how to bind a record to a record source, see Chapter 17, “Binding a Container to a Record Source”.

## Derived Values

You can have a control's internal value set as the result of a calculation, using derived values, instead of reading its values from a column in a table or view. After you set the **DataSource** property of the control to Derived Value, you can type in the expression used to determine the control's value in the DataSource section of the Property sheet.

The expression can use all Oracle Basic numeric functions. When referring to the internal value of a control, use the control's **Name** property. Optionally, you can use the syntax *control\_name.Value*.

For example, to set the internal value of fldThree to the sum of the internal values of fldOne and fldTwo, multiplied by two, you would enter the following for the control's derived value:

```
=(fldOne + fldTwo) * 2
```

-or-

```
=(fldOne.Value + fldTwo.Value) * 2
```

## Using SQLLOOKUP in a Derived Value

You can also use the SQLLOOKUP command for the derived value calculation. This is especially useful when the control needs to look up values in a foreign table (i.e., one other than the main table for the form, as specified through the RecordSource property).

For example, to display the department name instead of the department number for an employee record, you can enter the following expression for the DataSource property of the control:

```
=SQLLOOKUP(testsess, "SELECT dname FROM dept WHERE " "deptno  
= " + fldDeptno.Value)
```

In this syntax, testsess is the name of a session, and fldDeptno is a text field displaying the department number for an employee record.

## Containers and Derived Values

When setting the derived value for a control, you can refer to any other controls on the same container, or any appearing on another container within the container. For example, when setting a derived value for a control appearing on a form that also has a repeater display, you can refer to the

---

internal values of any controls on the form, plus any controls in the repeater display. However, controls on the repeater display cannot use controls on the main form as part of a derived value calculation.

### Aggregate Functions and Derived Values

Since controls in a repeater display appear multiple times (once for every record in the repeater), you may wish to use aggregate functions on the internal values of these controls. For example, you may want to sum all of the prices of line items within an invoice. In this case, the line items appear in the repeater display, and the unit price of each item is represented by a text field in the repeater display.

Aggregate values cannot be evaluated from within the container holding the controls whose internal values are being aggregated. For example, if you apply an aggregate function to values appearing in a repeater display, you must perform the calculation on the container holding the repeater display, not within the repeater display itself.

When setting derived values, you can use all Oracle Basic aggregate functions. In the case of summing prices for an invoice, you would create a text field on the main form that would use the SUM function as part of its derived value calculation. When setting the **DataSource** property for this text field, you would then enter something like the following:

```
=sum(repeater1.price)
```

### Display Values and Internal Values

All controls that can hold data have two values, the *internal* value and the *display* value. The display value is what appears in the control; the internal value is what is assigned to the control's **Value** property.

At times, the display value and the internal value will be different:

**List controls** - List boxes, combo boxes, and popup lists can translate the values assigned to their **Value** property. These translations are specified through the **Translation** or **ValueList** property of the control. For example, if you have a popup listing all customers, the display value may be the customer names, but the internal value is the numeric ID assigned to each customer. These values are queried from different columns in the same record source.

**Failed validation** - When the user enters a value in a control that fails to meet the business rule applied to this control, the **Validation()** method (in which the business rule is defined) returns FALSE, but the display value may not change. For example, if you enter a negative age for an employee, method code in the **Validate()** method for the control may return FALSE, preventing you from continuing further until you enter a valid age. In these cases, the control's display value is whatever the user entered, but the internal value is the original value assigned to

For more information on list controls and their values, see the section "List Controls" on page 10.15.

For more information on validation, see Chapter 19, "Using Constraints to Enforce Business Rules".

the control, before the user entered anything. This situation may also occur when the user tried to save a record that did not meet the constraints defined through the `ValidateRow()` method, which returned `FALSE` when the user tried to save the record to the database.

Note that in these cases, you can call the method `RevertValue()` to make the display value match the internal value when `Validate()` returns `FALSE`.

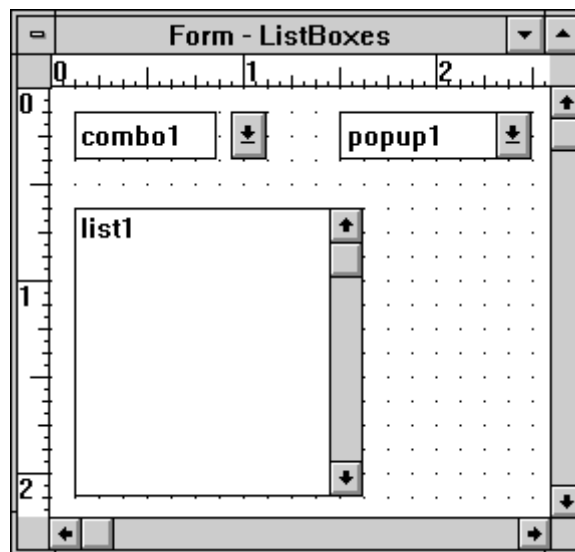
For more information on format masks, see the section "Format Masks" on page 10.31.

**Format masks** - When you apply a format mask to a text field or a combo box, you determine a format for information displayed in the control. A format mask controls the display of data to follow common conventions (for example, displaying a currency symbol before financial data). Again, the display value and the internal value are different. The display value includes formatting characters and often translates data (for example, showing the name of a month instead of a numeric value). The internal value, however, includes none of the extra characters, and the value is not translated.

For example, in the case of controls using the `Date` datatype, the underlying data is still a string of numerals defining a date and time. However, the format mask may set the display value to include colons to separate hours, minutes, and seconds, and it may display the names of months and days instead of their numeric equivalents.

## List Controls

Three controls in Oracle Power Objects can display a list of possible values as part of the control. In the case of list boxes and popup lists, the user can choose one of the items in the list. Combo boxes also allow the selection of a list item, but they also let the user enter a different value not on the list.



---

List controls are bindable, but the list of values appearing as part of the control are defined separately. The **Translation** property specifies the contents of the list for list boxes or popup lists, while the **ValueList** performs the same function for combo boxes. In both cases, you can “hard code” the contents of the list, by simply entering a list of values that can appear for the property. You can also query the values appearing in the list from a table or view.

As described earlier, the display value and internal value of list controls often vary, so the value selected in the list may not be the value assigned to the **Value** property of the control. In this sense, the display value is *translated* from the internal value of the control.

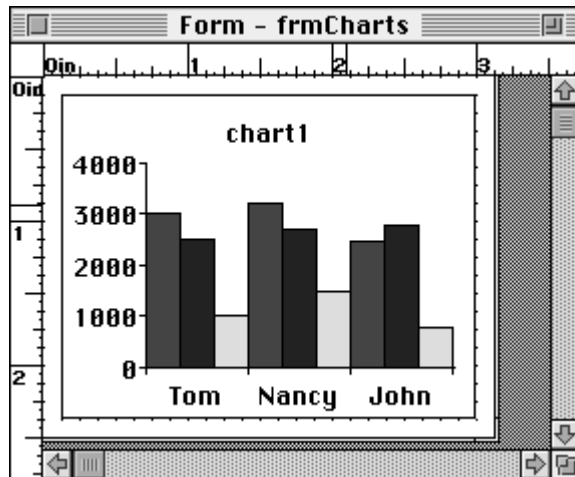
For more information on the **Translation** and **ValueList** properties, see their descriptions in the online help.

## Types of Controls

This section describes each kind of control in Oracle Power Objects, including suggestions on when to use some types of controls. For the properties and methods of each type of control, see the description of each control in the Oracle Power Objects online help.

### Chart Controls

Chart controls are bindable controls that display a chart or graph, using data queried from the database. Chart controls can bound to two columns or more from the same record source to display data within a chart or graph. If the chart object is not properly bound, it appears as a blank rectangle.

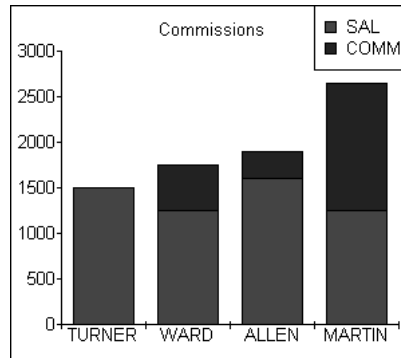


The chart has two important properties determining the range of information displayed, **ChartXCol** and **ChartYCols**. The **ChartXCol** property identifies the column used as the X-axis of the chart, while **ChartYCols** identifies the columns used as the Y-axis. If you want to build a bar chart depicting salaries and commissions for each employee, therefore, you would use EMP as the chart's **RecordSource**, ENAME for the **ChartXCol**, and SAL and COMM for its **ChartYCol** properties.

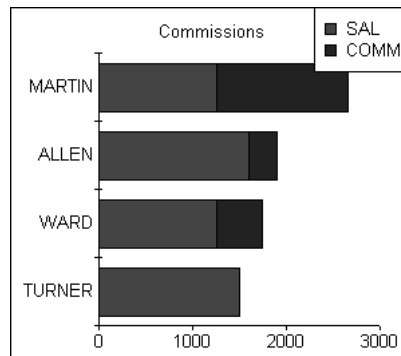
Chart objects can display a variety of different chart and graph types, including the following:

Chart Type	Example
------------	---------

Vertical bar



Horizontal bar

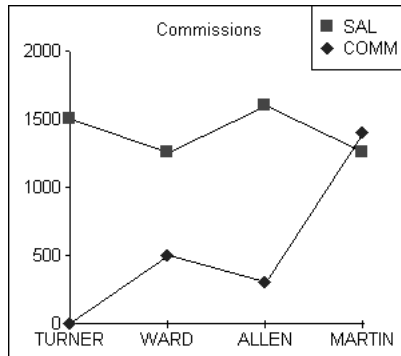


---

**Chart Type**      **Example**

---

Line



Pie

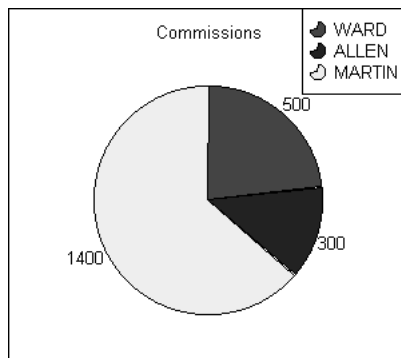


Chart controls can display legends, labels, and grid lines. You can have the application automatically size the chart, and you can determine the number of records displayed in it. To accomplish these tasks, chart controls have several unique properties

<b>Property</b>	<b>Description</b>
<b>ChartAutoFormat</b>	Determines whether the application automatically sizes the chart.
<b>ChartGap</b>	Determines the width between bars in a chart.
<b>ChartLabelStyle</b>	Determines whether the chart displays labels, and the contents of these labels.
<b>ChartLegendHAlign</b>	Determines the horizontal position of the legend.
<b>ChartLegendVAlign</b>	Determines the vertical position of the legend.
<b>ChartLineStyle</b>	Determines the appearance of grid lines in the chart.
<b>ChartMaxVal</b>	Sets the maximum value for the Y-axis of the chart.

Property	Description
<b>ChartMinVal</b>	Sets the minimum value for the X-axis of the chart.
<b>ChartOverlap</b>	Determines the degree of overlap between bars in a bar chart.
<b>ChartRowCount</b>	Sets the maximum number of records in the chart.
<b>ChartShowGrid</b>	Determines whether the chart shows a grid.
<b>ChartShowLegend</b>	Determines whether the chart has a legend.
<b>ChartStacked</b>	Determines how the chart stacks different bars belonging to the same record.
<b>ChartStyle</b>	Determines the style of the chart (vertical bar, horizontal bar, pie, or line).

### Chart Controls and the Record Manager

Since charts query records from a table or view, you must set its **RecordSource** and **RecSrcSession** properties to identify the record source and its session.

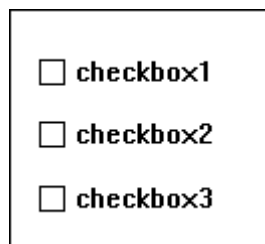
When the chart control performs a query, it fetches all records before drawing the chart or graph. Therefore, its **RowFetchMode** property must always be set to Fetch All Immediately.

Chart controls provide the means to display basic charts and graphs, using the Oracle Power Objects object-oriented approach to development. As an alternative, you can create charts as OLE objects, which can be added to an Oracle Power Objects applications. In these cases, the OLE server application provides the interface for defining and updating the graph, not the Oracle Power Objects application itself.

For more information on OLE objects, see Chapter 15, "Oracle Power Objects Extensions".

### Check Boxes

A check box is a bindable control used to indicate a mutually exclusive yes or no choice. To make the choice, you check or uncheck the box.



---

The appearance of the check box is related to its current value. When the check box is checked, its **Value** is the same as its **ValueOn** property. When unchecked, the control holds the value assigned through the **ValueOn** property; when unchecked, it holds the value assigned through **ValueOff** or null.

Although both check boxes and radio buttons provide mutually exclusive choices, radio buttons force a choice among grouped radio button controls. For check boxes, the choice applies only to one check box control, whose state (checked or unchecked) you determine.

### **Check Boxes and Null Values**

A check box containing a Null value appears unchecked, meaning that it is indistinguishable from a check box that the user has deliberately unchecked. When the user unchecks the check box, however, the application assigns the value specified in the control's **ValueOff** property to its **Value** property, which is not necessarily Null. To ensure that a check box does not contain a Null value, use the **DefaultValue** property to initialize the control to a non-Null value.

### **Check Boxes and Database Tables**

Since a check box is a bindable control, you can connect the check box to a column in a database table or view. The datatype of the check box (set through the **Datatype** property) must match the datatype of the column. In addition, the column should hold only two values, and they must correspond to the **ValueOn** and **ValueOff** properties of the check box.

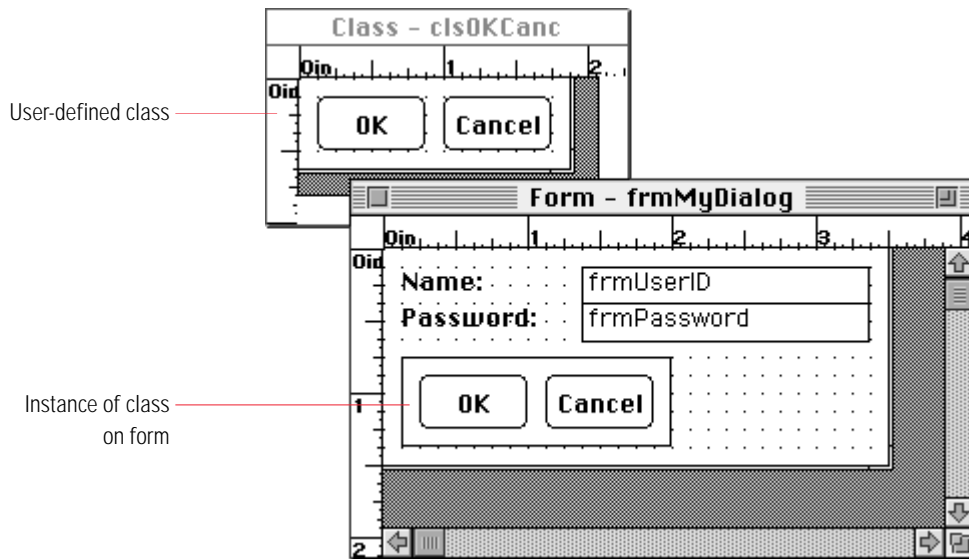
### **User-Defined Classes**

A user-defined class is a bindable container that can be reused throughout an application. When you add an instance of the user-defined class, the instance inherits the class' properties and methods. Additionally, any controls or static objects added to the user-defined class (also called the master class) also appear on its instance.

For more information on user-defined classes, see Chapter 13, "Classes".



User-defined classes normally consist of application components you plan to reuse in several places. For example, you can create a class of **OK** and **Cancel** pushbuttons which you can then add to all the dialogs in your application. After designing their default behavior (through the `Click()` method on both pushbuttons), you can later modify each instance of this class to fit the dialog on which it appears.



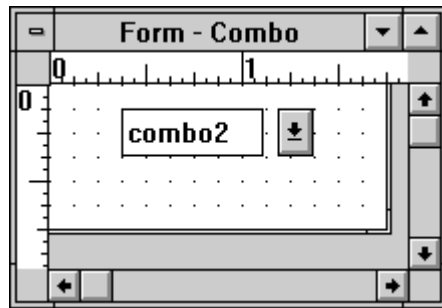
Some common uses of user-defined classes include the following:

- **Reusable application objects.** You can define a class containing some application object that is used frequently throughout the application. For example, if you often have a popup list showing customer names, then you can create that popup as a class, and then add instances of it to multiple forms.
- **Application customization.** When building a form, you can increase the ease of customization by adding an instance of a class instead of a normal control or static object. If you have a commonly used application object, such as the popup list described above, then you can customize all instances of the object by defining it as a class, then editing the class itself. The customization is then reflected in all instances.
- **Custom controls.** You can create a custom control such as a thermometer, a gauge, a calendar, or a series of **OK** and **Cancel** buttons as a user-defined class, then reuse them throughout an application.

---

## Combo Boxes

A combo box is a bindable list control that provides both a popup list of selectable items and a field for entering data. You use a combo box instead of a popup list or list box when you want to display a list of suggested values, but you also give the user the ability to enter a different value not appearing on the list.



The value for the combo control is the same as the text of the combo box. If the user selects an item from the list, the selected item replaces the contents of the text in the field.

### Populating a Combo Box's List

To populate the list section of the combo box, you must take the following steps:

- 1 Open the Property sheet for the combo box.
- 2 Enter an expression for the **ValueList** property to determine how the control displays values in its list.


### Combo Boxes and Foreign Tables





List boxes, combo boxes, and popup lists normally read their values from a different table than the main table for the form (as set through the **RecordSource** property). You can set a combo box to read its values from a foreign table through the **ValueList** property of the control.

For more information about the **ValueList** property, see the online help topic "ValueList property".

## Current Row Pointers

A current row pointer indicates the current row in a repeater display. Additionally, this control indicates the status of the currently selected record in any bound container.

Current row pointer 

	100	10" Beeswax Taper, Scarlet (set of 12)
	100	10" Beeswax Taper, Cobalt (set of 12)
	100	10" Beeswax Taper, Manila (set of 12)
	50	12" Rainbow Candle

When you display multiple records in a repeater display, a current row pointer sets the Record Manager's current row pointer to one of the records. You can click on the current row pointer appearing to make its corresponding row the current row.

Finally, the current row pointer indicates that a particular record is locked by displaying a lock icon next to the control.

The appearance of the current row pointer changes according to the status of the record to which it corresponds:

Appearance	Meaning
Gray arrow	Indicates the current row within the recordset, but the given recordset is not current.
White arrow	Indicates a row other than the current row.
Black arrow	Indicates the current row in the current recordset.
Lock next to arrow	The application has locked the current row in the database (normally because the row has been edited). The lock remains until the row's database transaction has been committed or rolled back.

Current row controls have a **Value** property that indicates the status of its associated row in the recordset (not the actual table or view). The lower-order bits provide the same row status information as the **GetRowStat()** method, while the remaining bits provide other information. To access this information, you must use a bitwise operator, such as IMP and OR.

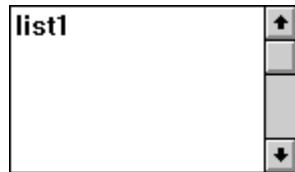
Value	Description
Value and 255	Value returned from <b>GetRowStat()</b> for the associated row
Value and 256	256 if the row is the active row, 0 if not
Value and 512	512 if the row is the current row, 0 if not
Value and 1024	1024 if the row is locked, 0 if not

---

<b>Value</b>	<b>Description</b>
Value and 2048	2048 if the row is the phantom “empty row” added to the end of a recordset, 0 if not

## List Boxes

A list box is a bindable control that presents a scrolling list of items for selection. Unlike a combo box, a list box restricts the user’s selection to the items appearing in the list. You use a list box in place of a popup list (which also restricts selection to listed items) when you want to keep the list’s choices visible while the user edits other controls.



Again, since a list box is a list control, its internal value may not match its display value. List boxes often read their values from a foreign table or view (that is, a different table or view than the record source of the container in which the list box appears).

## OLE Objects

For more information on OLE, see the section “OLE Data Objects and Controls” on page 15.2.

OLE objects are bindable controls built on the object embedding and linking (OLE) technology developed by Microsoft. OLE objects can appear only in Windows versions of Oracle Power Objects applications.

OLE objects are defined in a *server* application, and can appear in *client* applications for display and editing. However, when you edit the OLE object, the server application provides the interface for making any changes to its contents. For example, a Microsoft Word document or Microsoft Excel spreadsheet can appear within an Oracle Power Objects application, with all the same functionality as a document or spreadsheet originally defined in its native application. In these cases, Word and Excel are server applications, while the Oracle Power Objects application is the client.

OLE objects that are *embedded* store their data either in the Oracle Power Objects application, while *linked* objects store their data in the application that provides the interface for editing the data

## Picture Controls

A picture control is a bindable control that displays a graphic. The user can cut, copy, or paste picture values in this control. To add a new graphic, the user must define it in another application and paste it into the picture control. If the picture control is then bound, its contents (that is, the graphic) can be stored in and queried from a database.



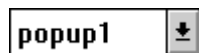
Currently, picture controls can display graphics in the .BMP and PICT formats. In Microsoft Windows, picture controls can only display 16 color graphics.

Picture controls must have the datatype *Long* to display a graphic. The graphic is stored using an appropriate datatype for the database storing the picture (for example, a PICTURE column for SQL Server, a LONG RAW column for a Blaze database or Oracle7 Server).

Graphics displayed in a picture object are not editable from within an Oracle Power Objects application. If you want to give the user the ability to modify a graphic within an Oracle Power Objects application, you should use an OLE control instead, linked to a drawing tool like Microsoft Paintbrush.

## Popup Lists

Popup lists are bindable control that presents a list of items for selection.



The list does not appear until you click on the popup list control. Once the list appears, you can use the scrollbar that appears on the right side of the control to scroll through the available items.

Like a list box, a popup list restricts the user's choices to only those items in the list. Often, a popup list is often preferable to a list box when space is at a premium.

Since a popup list is a list control, the value displayed in the list may match its corresponding internal value (the value actually assigned to the **Value** property of the control).

---

## Pushbuttons

A pushbutton emulates a physical pushbutton. Clicking the button usually triggers some action, as defined in the method code attached to the pushbutton's `Click()` method.



Unlike most other controls, pushbuttons do not have a **Value** property.

### Pushbuttons and Modal Dialog Boxes

If a pushbutton appears on a modal form, you can set it to dismiss the form without having to write any method code. When the user clicks the pushbutton, the application automatically hides the form if pushbutton's **IsDismissBtn** property set to `TRUE`. Note that this is true only if the form was opened using the `OpenModal()` method.

In this case, the form is hidden, not unloaded from memory (though it is effectively closed from the user's standpoint). To unload it from memory, you must use the `CloseWindow()` method.

The purpose of this behavior is to keep in memory a dialog box in which the user has entered some settings. While the dialog box remains hidden, the application can continue to read these settings from the controls appearing on the hidden form.

## Radio Buttons

Radio buttons are bindable controls that present a mutually exclusive set of choices. Radio buttons normally appear within a radio button frame, which groups the radio buttons and stores a value corresponding to the currently selected radio button.



For information on database connectivity for radio buttons, see the following description of radio button frames.

### Radio Button Groups

To create a group of radio buttons, you first create a radio button frame, and then add radio buttons to the frame. The radio button frame's **Value** property matches the **ValueOn** property of the currently selected radio button within the frame. Each radio button must therefore have a unique **ValueOn** property, and the radio buttons must have the same **Datatype** property as the radio button frame.

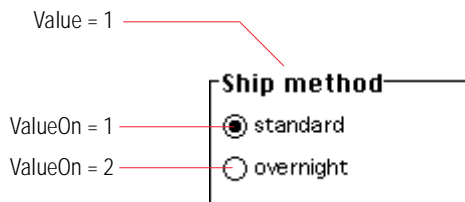
To determine programmatically the value of the selected radio button within a group, you read the **Value** property of the radio button frame, not from the radio buttons themselves. Similarly, to bind radio buttons to a table or view, you set the **DataSource** property of the frame, not of the individual radio buttons.

Radio buttons on a form that are not within a radio button frame are part of the same group; all radio buttons within a radio button frame comprise a different group. Once radio buttons have been grouped, the user can select only one button at a time from that group.

### Radio Button Values within a Group

The value corresponding to each radio button is assigned through the **ValueOn** property of each radio button. You can select any datatype, but you must use the same datatype for all radio buttons in the group. The **ValueOn** for each radio button must be unique within that group.

If the radio buttons are bound, then you must set the **ValueOn** property of the radio buttons to match the values stored in the corresponding column. For example, if you have two values in a table, 0 and 1, and you want to reflect these values in two radio buttons, then set one radio buttons **ValueOn** property to 0, and the other's to 1. By default, the **ValueOn** property of a radio button is its **Label** (a string value).



#### ☆ To create a radio button group:



**1** Using the Radio Button Frame tool, create a radio button frame object.



**2** Using the Radio Button tool, draw the radio buttons within the radio button frame.

**3** Set the **Datatype** property of the radio button frame and the radio buttons.

**4** Bind the radio button frame to a record source, if desired.

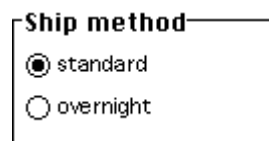
**5** Set the **ValueOn** properties of the radio buttons to match the values stored in the corresponding column, if the radio buttons are bound controls.

---

## Radio Button Frames

Radio button frames group radio buttons, making the choice among them mutually exclusive. In addition, the frame stores the value assigned to the currently selected radio button, and can store this value in a column specified as the radio button frame's DataSource.

For more information on radio button frames and radio buttons, including information on how to group and bind these controls, see the description of radio buttons above.



## Scrollbars

A scroll bar lets the user (1) scroll through records by moving a small rectangle (the thumb) across the length of the scrollbar, or (2) take some action by manipulating the scroll bar.



Oracle Power Objects provides two kinds of scrollbars, vertical and horizontal.

### Determining the Position of the Scrollbar's Thumb

Scrollbars have three important properties that determine the location of the thumb:

Property	Description
<b>ScrollMin</b>	A minimum value assigned to the scrollbar. The control has this value when you scroll all the way to the top for vertical scrollbars, or to the left for horizontal scrollbars.
<b>ScrollMax</b>	A maximum value for the scrollbar. The control has this value when you scroll all the way to the bottom for vertical scrollbars, or to the right for horizontal scrollbars.
<b>ScrollPos</b>	The current position of the scrollbar thumb. This integer value always falls between the values set for ScrollMax and ScrollMin.



## Scrollbars and Database Browsing

You can use a scrollbar to scroll through the records in a recordset. To make a scrollbar a recordset browser, set the **ScrollObj** property of the scrollbar to the **Name** of the bound container associated with the recordset. Note that when you add a scrollbar to a container, its **ScrollObj** property is set by default to the **Name** of the container.

## Scrollbars on Other Controls

Scrollbars (not scrollbar objects) can appear as part of other application objects. In these cases, the scrollbars are not the same as scrollbar controls; they are simply part of the container (for example, a repeater display) or control (for example, a text field). In some cases, you can use the **HasScrollBar** property to add or remove the intrinsic scrollbar.

## Text Fields

A text field is a bindable control used for displaying and entering data. Text fields can hold data read from a database table, entered by a user, or resulting from a derived value calculation.

**textfield21**

Text fields are also the default control type when you add database objects by dragging them from a table or view onto a form, report, or class.

When the user enters data into the text field, the field's **Value** property is not set until the focus leaves the field. When the focus moves out of the field, the **Validate()** method is triggered, applying any checks you want on the data entered.

After entering data into a text field, but before moving the focus out of it, the user can undo the new text by pressing the Escape key.

## Displaying Multiple Lines in a Text Field

To display multiple lines of text in a text field, you must have the control's **MultiLine** property set to true. In addition, if the control will be able to hold more lines of text than it can display at one time, then you should set the text field's **HasScrollBar** property to TRUE as well.

Currently, text fields do not have word wrap capabilities, so the user must manually enter line break characters to "wrap" text in the field.

---

## Types of Static Objects

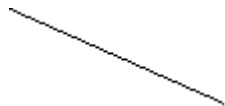
This section describes each kind of static object in Oracle Power Objects. For the properties and methods of each type of static object, see the description of each static object in the Oracle Power Objects online help.

### Lines

A line is a static object that displays a straight line.

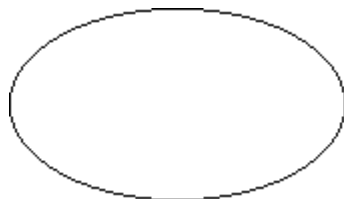


Commonly, a line separates sections of a form or report. Lines have a **Direction** property that indicate the slope of the line, as shown below.



### Ovals

A static object that displays an oval. Ovals are containers in which you can place other objects.



Commonly, ovals surround objects you wish to group or highlight on a form or report.

## Rectangles

A static object that displays a square or rectangle. Rectangles are containers in which you can place other objects.



Rectangles often surround objects grouped on a form or report.

To group radio buttons as bound controls, you use a radio button frame, not a rectangle. While a rectangle can group radio buttons, it cannot store the value of the currently selected radio button.

## Static Text

A static text object displays a line of text, normally used to label other objects.

### Static Text

You normally place static text to the left or immediately above the object you wish to label. Unlike a text field, the user cannot enter a value into a static text object. Additionally, static text objects cannot be bound to database objects and do not possess the **Value** property. However, you can change the text displayed in it at run time by setting the **Label** property through Oracle Basic code.

## Interacting with Application Objects

### Format Masks

Frequently, you want to provide formatting for text appearing in a control. For example, if a text field displays currency data, you want to add the appropriate currency symbol, commas, and decimal point to any monetary amount appearing in the field. In these cases, you use a *format mask* to determine the format and visual presentation of data in the control.

Format masks determine only how data is displayed, not how it is stored internally (that is, in the application or in the database). In the case of currency values, for example, the data is stored as the appropriate datatype (normally *Float*), without the formatting symbols.

In Oracle Power Objects, format masks are set through the **FormatMask** property of a control. Format masks apply only to text fields and combo boxes, which are the only controls in which you can type a value. To set this property, you enter a string of characters corresponding to the type of format mask you wish to add. Several standard masks exist in Oracle Power Objects, but you can enter your own mask as well.

Oracle Power Objects format masks are not identical to SQL format masks. However, they are identical to the formats that can be passed to the **FORMAT** command in Oracle Basic. For more information on SQL and Oracle Basic format masks, see the Oracle Power Objects online help.

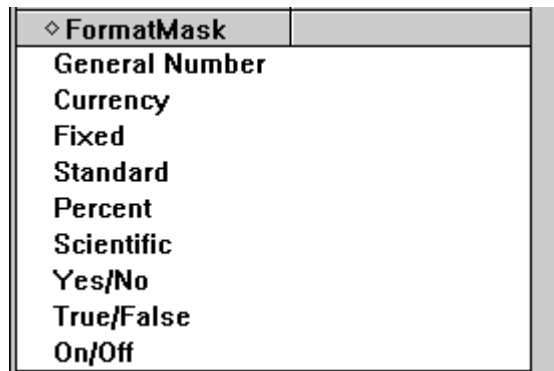
Format masks do not constrain the user from entering certain types of data; instead, they only determine how data appears in the control. To constrain the values entered by the user, you would add method code to the **Validate()** method of the control.

### Using Standard Format Masks

To assign a standard format mask to a text field or combo box, follow these steps:

- 1 Click on the **FormatMask** section of the Property sheet, on the actual name of the property (not the window to the immediate right of the name).

A dropdown list of standard format masks appears.



- 2 Select the desired format mask from this list.

The following standard format masks are available through the Property sheet:

<b>Format Mask</b>	<b>Datatype</b>	<b>Example</b>
General Date	<i>Date</i>	1/1/95 10:00:00 PM
Long Date	<i>Date</i>	Monday, JANUARY 1, 1995
Short Date	<i>Date</i>	1/1/95
Oracle Date	<i>Date</i>	1-Jan-95

<b>Format Mask</b>	<b>Datatype</b>	<b>Example</b>
Long Time	<i>Date</i>	10:00:00 PM
Medium Time	<i>Date</i>	10:00 AM
Short Time	<i>Date</i>	22:00
All Caps	<i>String</i>	HELLO THERE
Init Cap	<i>String</i>	Hello There
All Lowercase	<i>String</i>	hello there
General Number	<i>Double, Long</i>	1000.
Currency	<i>Double, Long</i>	\$1,000.00
Fixed	<i>Double, Long</i>	1000.00
Standard	<i>Double, Long</i>	1,000.00
Percent	<i>Double, Long</i>	100000.00%
Scientific	<i>Double, Long</i>	1+E3
Yes/No	<i>Double, Long</i>	Yes
True/False	<i>Double, Long</i>	True
On/Off	<i>Double, Long</i>	On

### Defining Your Own Format Masks

Optionally, you can define your own format mask through the Property sheet. For example, if you wanted to display currency information formatted with a currency symbol, commas, and a period, you would enter the following format mask:

\$S### , ### , ### . ##

In this mask, the double dollar sign (\$S) ensures that the appropriate international currency symbol used by the operating system appears at the beginning. The pound sign (#) instructs the application to display numeric digits only. The period (.) adds a decimal point to the format mask.

☆ **To enter a format mask:**

- 1 Check the **Datatype** property of the control to determine which format mask characters can be used.

See below for a table of all the format mask characters appropriate to each datatype.

- 
- 2 Click on the window appearing next to the word **FormatMask** on the Property sheet.  
The focus then moves to this small window, used for entering user-defined format masks.
  - 3 Enter the text string defining the format mask.

### Format Mask Characters

The following table summarizes format mask characters, organized by datatype:

#### Numbers

<b>Symbol</b>	<b>Meaning</b>
0	Displays a digit (0 if none specified).
,	Displays a thousand separator, and scales by 1,000 if at the end of a number.
#	Displays a digit, if possible; otherwise, displays nothing.
.	Displays a decimal point.
..	Displays a decimal point, regardless of the environment's control panel settings for decimal values.
\$	Displays local currency symbol.
\$\$	Displays international currency symbol for current locale.
%	Converts the value to a percentage (that is, multiplies by 100) and displays the percentage symbol following it.
o	Displays octal digits.
- + \$ ( ) <i>space</i>	Displays a literal character. To force display of a character not included in this list, precede the character with a backslash (\).
"x" 'x'	Displays a literal string.
X	Displays uppercase hexadecimal digits.
x	Displays lowercase hexadecimal digits.
R	Displays uppercase Roman numerals.
r	Displays lowercase Roman numerals.
E+, E-	Displays a value in exponential notation, with the exponential symbol (E) in uppercase. E+ displays a plus sign when there is a positive exponent; E- does not.

Symbol	Meaning
<i>e+</i> , <i>e-</i>	Displays a value in exponential notation, with the exponential symbol (e) in lowercase. <i>e+</i> displays a plus sign when there is a positive exponent; <i>e-</i> does not.
<i>E?</i>	Displays the exponential notation only when the number is too large or too small to fit in the space allocated.
<i>/x</i>	Truncates the number if it is too long to the left of the decimal for the format. Displays the literal character <i>x</i> in all digit positions if truncation occurs.
<i>*x</i>	Fills the number if it is too short for the format. Filling in from the left, the application adds the literal character <i>x</i> as needed to fill the space.

### Strings

Symbol	Meaning
@	Defines a character place holder. Displays a character or a space, if no character is provided for the specified position in the string.
&	Defines a character place holder. Displays a character or nothing, if no character is provided for the specified position in the string.
!	Fills string place holders from left to right rather than from right to left.
/	Truncates string if too long for mask. Truncates to the right, and fills from the left until it runs out of place holders.
<	Displays all subsequent characters in lowercase.
-	Does not change case of subsequent characters.
>	Displays all characters in uppercase.

### Dates

Symbol	Meaning
<i>c</i>	Displays the date as <i>dddd</i> and the time as <i>tttt</i> , in that order. Only date information is displayed if there is no fractional part in the date serial number; only time information is displayed if there is no integer portion.
<i>d</i>	Displays the day as a number without a leading zero (for example, 7).
<i>dd</i>	Displays the day as a number with a leading zero (for example, 01).
<i>ddd</i>	Displays the day as an abbreviation (for example, Mon.).

---

<b>Symbol</b>	<b>Meaning</b>
<i>dddd</i>	Displays the day as a full name (for example, Friday).
<i>ddd</i>	Displays a date serial number as a complete date ( <i>dmy</i> ), formatted according to the short date setting in the International section of the Windows Control Panel.
<i>dddddd</i>	Displays a date serial number as a complete date, formatted according to the long date setting in the International Section of the Windows Control Panel. The default setting is <i>mmmm dd, yyyy</i> .
<i>ww</i>	Displays the week of the year as a number (1 to 54).
<i>mm</i>	Displays the month as a number with a leading zero (for example, 02). If <i>m</i> immediately follows <i>h</i> or <i>hh</i> , the minute rather than the month is displayed.
<i>mmmm</i>	Displays the month as a full name (for example, August).
<i>y</i>	Displays the day of the year as a number (1 to 366).
<i>yyyy</i>	Displays the year as a four-digit number, from AD 100 to AD 9999.
<i>w</i>	Displays the day of the week as a number, from 1 for Sunday to 7 for Saturday.
<i>m</i>	Displays the month as a number without a leading zero (for example, 3). If <i>m</i> immediately follows <i>h</i> or <i>hh</i> , the minute rather than the month is displayed.
<i>mmm</i>	Displays the month as an abbreviation (for example, Dec.).
<i>q</i>	Displays the quarter of the year as a number, from 1 to 4.
<i>yy</i>	Displays the year as a two-digit number, from 00 to 99.
<i>h</i>	Displays the hour as a number without leading zeros, from 0 to 23.
<i>hh</i>	Displays the hour as a number with leading zeros, from 00 to 23.
<i>nn</i>	Displays the minute as a number with leading zeros, from 00 to 59.
<i>ss</i>	Displays the second as a number with leading zeros, from 00 to 59.
<i>AM/PM</i>	Uses the 12-hour clock and displays an uppercase AM or PM with every time.
<i>A/P</i>	Uses the 12-hour clock and displays an uppercase A or P with every time.
<i>AMPM</i>	Uses the 12-hour clock and displays the contents of the 1159 string (s1159) in the WIN.INI file with any hour before noon; displays the contents of the 2359 string (s2359) with any hour between noon and 11:59 PM. AMPM can be uppercase or lowercase, but the case of the string displayed matches the string as it exists in the WIN.INI file (default is uppercase).
<i>n</i>	Displays the minute as a number without leading zeroes, from 0 to 59.



<b>Symbol</b>	<b>Meaning</b>
<i>s</i>	Displays the second as a number without leading zeroes, from 0 to 59.
<i>tttt</i>	Displays the time serial number as a complete time ( <i>hms</i> ), formatted using the time separator specified in the International section of the Windows Control Panel. A leading zero is displayed if the Leading Zeros option is selected in the Control Panel, and the time is before 10:00 AM or 10:00 PM. The default format is h:mm:ss.
<i>am/pm</i>	Uses the 12-hour clock and displays a lowercase am or pm with each value.
<i>a/p</i>	Uses the 12-hour clock and displays a lowercase a or p with each value.
<i>1123</i>	Uses the 24-hour clock and displays the suffix for 24 hour time (for example, AM/PM) read from the environment's date and time settings.
<i>0</i>	Instructs the application to display midnight and noon as 0:00. This character must follow one of the other format mask characters used to format time information according to the 12-hour clock.

### Setting Format Masks through the Environment

The user can also set a format mask through the client operating system (Microsoft Windows, Macintosh, or OS/2). Any application developed through Oracle Power Objects can use the format mask for currency, time, and date information from the environment. In Microsoft Windows, these environment-level format masks are set through the Control Panel; on the Macintosh, they are set through the appropriate control panel.

Currently, the only elements read from the environment are separators and currency symbols. For example, an Oracle Power Objects application may have a dollar sign (\$) as part of a format mask. However, if the user then changes the international currency setting to the British pound character (£) through the operating system's control panel, then the Oracle Power Objects application uses the pound symbol instead of the dollar sign.

To read the environment settings, you must use specific format masks, as described below:

<b>Format Mask</b>	<b>Description</b>
\$\$	Reads the international currency symbol from the environment.
dddd	Displays a short date, using the date set in the environment.
ddddd	Displays a short date, using the date format set in the environment.
tttt	Displays the time, using the hour/minute/second format set in the environment.

---

When displaying date and time values, the application retains the separators “/” and “:”, used to divide sections (for example, hours, minutes, seconds) of the date or time. However, the application rearranges this information to match the environment’s settings for date and time formats.

## Tab Order

By specifying a tab order, you define the sequence of controls through which the focus moves when the user presses the Tab key.

In Oracle Power Objects, the tab order is defined through the **TabOrder** property of controls. A control that has a value assigned to this property is part of the tab order; otherwise, it is not.

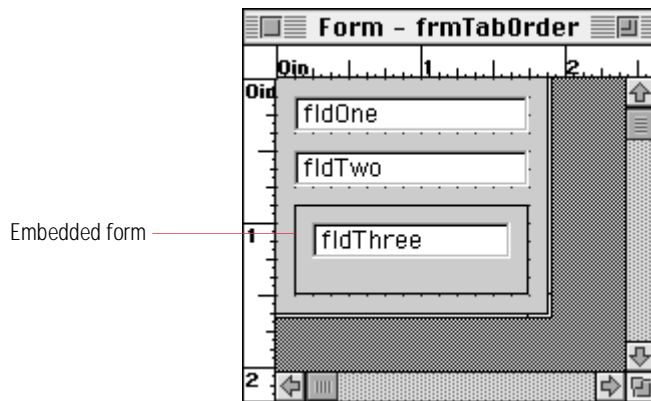
Each item in the tab order must have a unique number (greater than zero) assigned to its **TabOrder** property. When the user tabs, the focus moves from currently selected control to the control having the next highest value assigned to its **TabOrder** property. For example, if the focus is in a text field whose **TabOrder** property is set to 6, then the focus moves to the control having the **TabOrder** property set to 7 when the user tabs.

Note that the **TabOrder** values do not have to be consecutive; the focus moves to the control having next highest **TabOrder** value.

Once the focus moves to the control having the highest **TabOrder** value, the focus then moves to the control with the lowest **TabOrder** value when the user tabs again.

## Containers and the Tab Order

Unless a container appearing within a form is part of the tab order, the focus cannot move to any controls within the container while the user tabs. This rule applies to any container that can appear within another container, including embedded forms, repeater displays, rectangles, ovals, and user-defined classes. As long as the container itself has any value assigned to its **TabOrder** property, the focus can then extend into the container.



## Validation

For more information on enforcing business rules, see Chapter 19, "Using Constraints to Enforce Business Rules".

When the user enters data, you often want to validate it, to enforce *business rules* (also termed *constraints*). For example, in an invoice entry form, you want to make sure that the user cannot enter an apply date earlier than the entry date.

These checks can occur when the user enters data into a control, or when the user attempts to write changes to the database. Oracle Power Objects provides the means to enforce business rules at both times, through two key methods, **Validate()** and **ValidateRow()**.

## Enabling and Disabling Controls

Often, you want to make it impossible for the user to change the contents of a control. For example, when the user enters an invoice, you want the user to enter line items and other information, but not necessarily edit the customer information displayed on the form. In these cases, you need to disable some controls you do not want the user to edit, while enabling others.

In Oracle Power Objects, enabling and disabling controls is possible by assigning values to either the **Enabled** or **ReadOnly** property. When **Enabled** is set to False, or **ReadOnly** is set to True, the user cannot enter data into the control, or interact with it at all. However, in the case of the **Enabled** property, a disabled control changes the color of text within it to gray, to give a visual cue that the control is disabled. When the **ReadOnly** property is set to True, however, the text is not "grayed".





# 11

---

## Forms

This chapter covers the following topics:

Overview .....	11.2
Developing Forms .....	11.3
Testing a Form .....	11.8
Forms and Modality .....	11.11
Forms and Window Styles .....	11.13
Controlling the Behavior of Forms .....	11.13
Query by Form .....	11.14
Queries, Conditions, and Forms: A Summary .....	11.19

---

## Overview

Forms give the user the ability to view and enter data, navigate through the application interface, and perform a variety of other tasks. Considerations when designing forms include:

- Determining the behavior of a form, including setting the characteristics necessary to give a form the behavior of a dialog box.
- Creating, copying, and deleting forms.
- Adding objects to a form.
- Testing forms, both separately and as part of an application.
- Giving the user the ability to filter the records displayed in a form.

Another important aspect of forms is binding them to record sources, as discussed in Chapter 17, “Binding a Container to a Record Source”. This chapter discusses the other important development concerns related to forms.

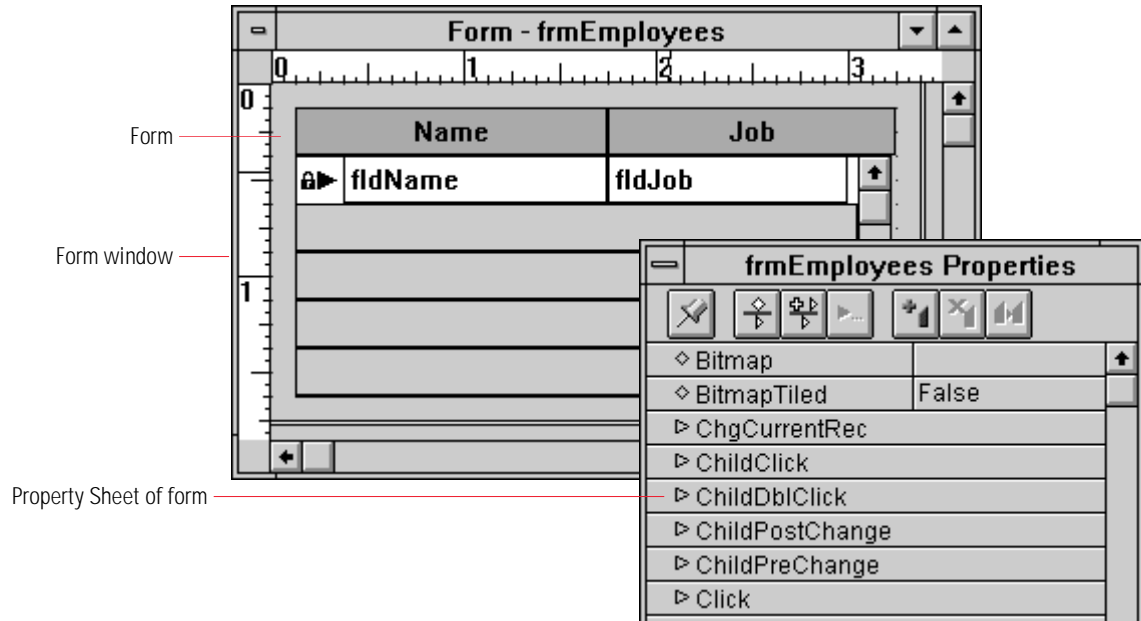
### What Is a Form?

A form is a kind of *application object* whose description is stored as part of an application. A form is also a *container*, in that it can contain other application objects. A form is also *bindable*, meaning that it can be associated with a table or view. When the form is bound, controls on the form can display information queried from a database. The user can then add, delete, or modify records with these controls.

The form consists of two components:

- **The form itself**, which can contain controls, static objects, and other containers.
- **The window**, which surrounds all or some of the form. The window can be resizable, and it can also allow the user to minimize the form to an icon. The window can also display a button that lets the user resize the window or close the form. If the form can be resized, the form commonly displays a scrollbar when the window displays only a part of the entire form.

The components of a form are shown below:



## Developing Forms

Three basic procedures for working with forms include creating a form, deleting a form, and copying a form between applications.

### Creating a New Form

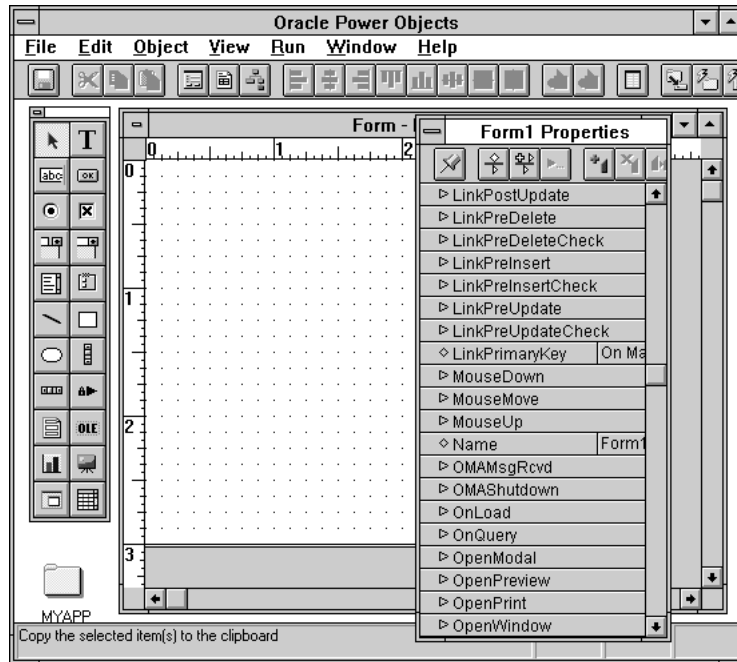
☆ **To create a new form:**



- 1 With the Application window active, click the New Form button on the Application Designer toolbar.

---

A new form now appears in the Form Designer window, as does the Property sheet for the form. Additionally, an icon for the form appears in the Application window.



**2** Assign a name to the form through its **Name** property.

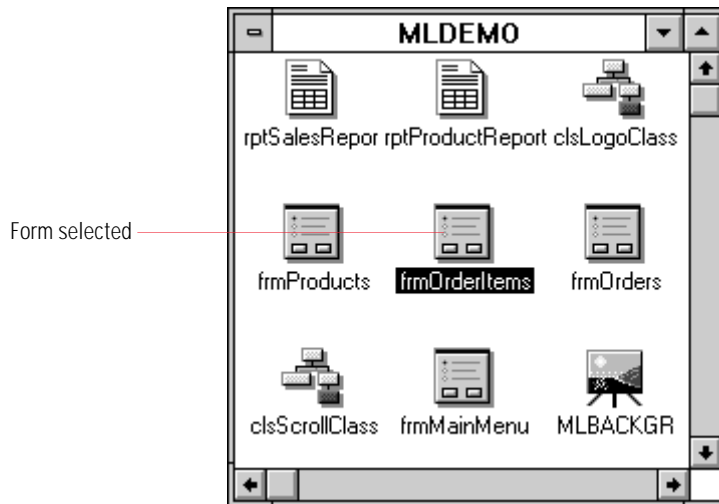
Oracle Power Objects automatically assigns a default name to a form when you first create it (for example, `Form16`). However, by giving your form a descriptive name (for example, `frmVoucherEntry`), you can remember more easily its name and purpose.



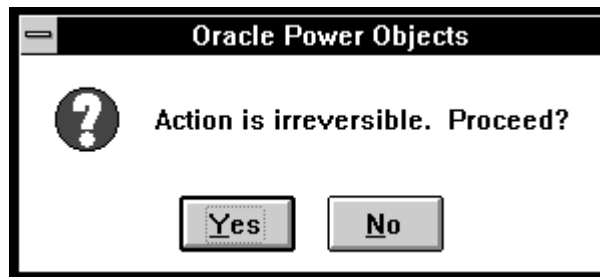
## Deleting a Form

☆ **To remove a form from an application:**

**1** Select the form icon in the Application window.



**2** Click the Cut button in the Application Design toolbar, or choose the **Edit-Cut** menu command. Oracle Power Objects then warns you that deleting the form is an irreversible action.





**3** Click **OK** or press Return.

The form is now permanently deleted from the application.

---

## Copying a Form

☆ **To copy a form between applications:**

- 1 In the Application window of one application, select the form.
-  2 Click the Copy button on the Form Designer toolbar, or choose the **Edit-Copy** menu command.
- 3 Open the window for the application into which you are copying the form.
-  4 Click the Paste button on the Form Designer toolbar, or choose the **Edit-Paste** menu command.



A copy of the form now appears in the second application.

Oracle Power Objects does not maintain any connection between these two copies of the form. If you want to create copies of a bindable container that inherit the properties and methods of the original version, you should instead create the form as a user-defined class.

For information on classes, see Chapter 13, “Classes”.

## Cutting and Pasting a Form

☆ **To cut a form from one application and paste it into another:**

- 1 In the Application window, select the form you wish to cut.
-  2 Click the Cut button or choose the **Edit-Cut** menu command.
- 3 Open the Application window for the application into which you want to paste the form.
-  4 Click the Paste button or choose the **Edit-Paste** menu command.

The form now appears as part of the other application.

## Adding Objects to a Form

Once you have created a new form, you can begin adding controls, static objects, and other containers to it. These objects comprise the form's interface—they display information, visually delineate areas of the form, and provide controls for the user.

☆ **To add a new object to a form:**

- 1 Open the Form Designer window for the form by double-clicking on the form's icon.
- 2 Choose the appropriate drawing tool from the Object palette.

Each type of object that you can add to the form is represented by a different button. When you select a button, the cursor then changes to indicate the type of object you are going to add to the form.

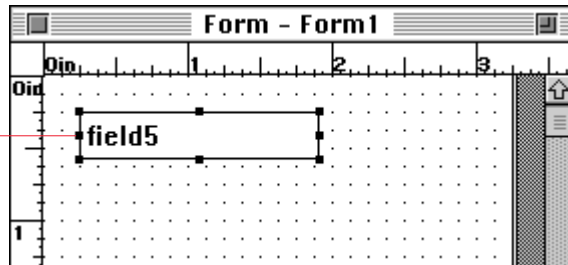


3 Click on a region of the form to create an object with the default size

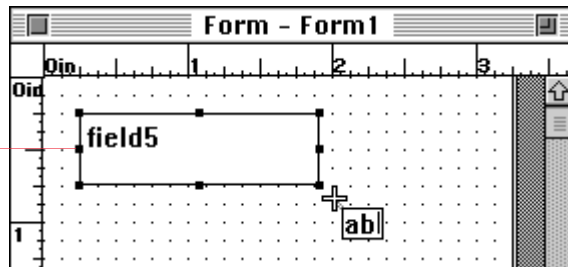
-or-

Click on a region of the form and drag the mouse across it to create an object with a custom size.

Click once to create an object with the default size.



Click and drag to create an object with a custom size.



The new object now appears within the region of the form where you clicked and dragged. Oracle Power objects assigns a default name to the object (for example, `Field1`).

- 4 If desired, assign a descriptive name to the object (for example, `fldEmpName`).
- 5 Add a descriptive label to appear on the title bar of the form by entering text for the form's **Label** property.

---

## Testing a Form

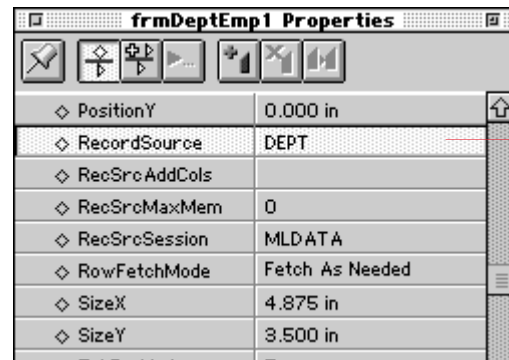
Testing a form is an important part of the development process. Before you compile the application, you want to ensure that forms you have designed behave as expected.

In Oracle Power Objects, you can test forms individually (Form Run-Time mode), or you can test them as part of the entire application (Application Run-Time mode). Forms appear and behave differently in Design mode (when you are editing the form's properties and methods), Form Run-Time mode, and Application Run-Time mode.

### Forms in Design Mode

At design time, the form displays all objects contained within it, including objects that are invisible at run time (that is, whose **Visible** property is set to **FALSE**). You can add, move, resize, and delete objects in the form. You can use the Property sheet to view and edit the properties and methods of each object within the form, as well as the form itself.

If the form is bound to a table or view, any bound controls within the form do not display data queried from the associated table or view. In fact, the form itself has no associated recordset at this time, because the application only instantiates recordset objects at run time. However, through the Property sheet, you can view and change the record source to which the table is bound, as well as the session in which it appears. Additionally, you can use the Property sheet to view and change the columns in the record source to which controls are bound.



You can use the Property Sheet to examine and set properties of the form.

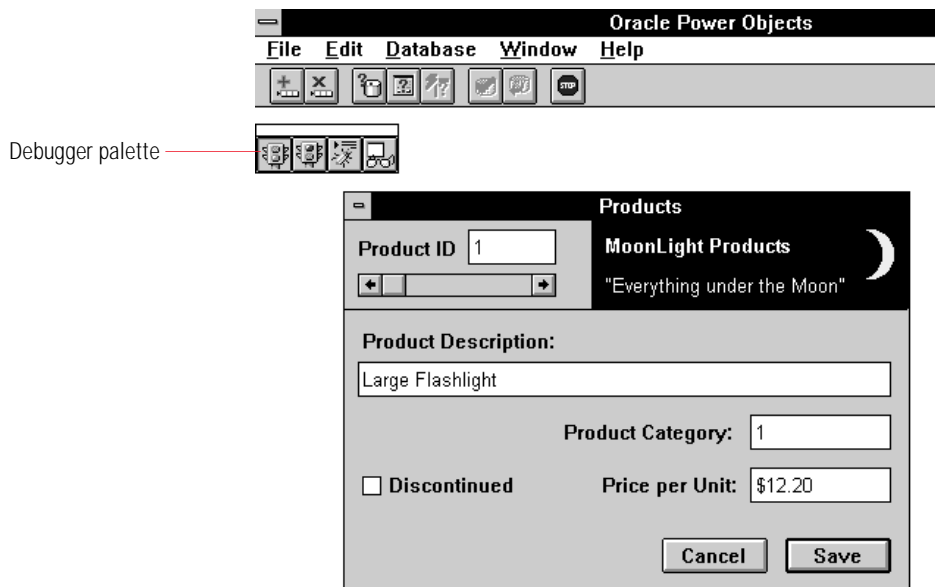
When you make changes to objects during design mode, your changes are permanent. When you customize objects by changing property values or adding method code, these changes are saved and used whenever you examine or run the application.

## Forms in Form Run-Time Mode



Once you have reached a point where you want to test a form, you can enter Form Run-Time mode by clicking the Run Form button or choosing the **Run-Run Form** menu command. The form then appears as it will at run time, including data queried from the database.

When the form appears in Form Run-Time mode, the Debugger palette appears. You can use this palette to open the Debugger.



For information about the debugger, see the section “Debugging Method Code” on page 5.9.

When you run a form, you no longer have object design tools available to you. You can execute Oracle Basic method code, but you cannot add new method code to objects. You can change property values, but these changes are temporary—they are not stored after execution stops.

For forms that are bound to a table or view, a recordset object is created for the form at run time. You can manipulate this recordset through certain properties and methods (for example, `SetCurRow()` and `GetColVal()`). Recordsets are described in Chapter 17, “Binding a Container to a Record Source”

When you view a form in Form Run-Time mode, Oracle Power Objects runs only that form, not the entire application. Oracle Power Objects cannot resolve references to objects and variables defined outside the form itself, including:

- 
- **Object references to other forms and reports, as well as other objects appearing on them.** When method code referencing these objects is executed, Oracle Power Objects displays an error message and stops running the form.
  - **References necessary for a shared recordset.** If you have established a shared recordset that refers to a container object outside the current form, Oracle Power Objects stops running the form and displays an error message like the following one:



Shared recordsets are described in the section “Shared Recordsets” on page 17.19.

- **Declarations made in the application’s Property sheet.** The form cannot resolve references to any global variables or Dynamic Link Library (DLL) procedures declared in the (Declarations) section of the application’s Property sheet.

Additionally, if you have added method code to the **AppInit()** or **OnLoad()** method of the application, this code does not execute when you test the form alone.

However, if you run the form in Application Run-Time mode, Oracle Power Objects compiles the entire application. You can then fully test the form, including components of the form that reference other application objects.

## Forms in Application Run-Time Mode



You can enter Application Run-Time Mode by clicking on the Run Application button or by choosing the **Run-Run Application** menu command. When you enter Application Run-Time mode, you run the entire application, not just the currently selected form. As with Form Run-Time mode, the form appears as it will to the user, including any data queried from the database. Additionally, the form can resolve all references to application objects outside the form itself, such as other forms and reports.

Although Application Run-Time mode provides the most complete test of a form, it has some disadvantages:

- **You must navigate to the form through the application.** In some cases, this may require opening other forms, or taking other steps, before accessing the form you want to test.
- **Problems with the form may arise from sources outside the form itself.** If you want to isolate a problem associated with a form, it is often simpler to run the form alone, not the entire application. You can then eliminate the possibility that the problem arises from outside the form.
- **It may take longer to perform the test.** Since Oracle Power Objects compiles any form or application you test, the compile will take longer when you test the entire application.
- **Unnecessary database connections may be established.** Application objects other than the form you wish to test may make database connections when you test the application, leading to other possible complications.

For these reasons, you should test a form in Application Run-Time mode only when you need to test the entire application, or you need to test the dependency between different forms.

## Forms and Modality

The modality of a form determines whether you can access other parts of the application or the environment while the form is open. Most forms are opened *non-modally*, which means that you can freely access any other part of the application or environment. However, two other forms of modality limit what the user can access:

Modality	Description
Application-Modal	The user cannot access any other part of the application before responding to the form.
System-Modal	The user cannot access anything else in the client operating system before responding to the form.

The modality of the form can be determined by the method used to open it. In Oracle Power Objects, the following two methods load the form into memory and set its modality:

Method	Description
<code>OpenWindow()</code>	Opens a form non-modally.
<code>OpenModal()</code>	Opens a form modally. <code>OpenModal()</code> takes one argument, which determines the type of modality (0 or FALSE for application-modal; any nonzero, non-null value or TRUE for system-modal).

---

## Dialog Boxes

You can use a form as a dialog box by opening it with the **OpenModal()** method instead of **OpenWindow()**. The form then has the expected behavior of a dialog box, blocking access to the application or the environment until the user responds to the dialog.

A dialog box blocks access to the rest of the application until the user responds.

The screenshot shows a modal dialog box titled "Orders" overlaid on a background application window. The dialog box contains the following fields and controls:

- Order ID: 19
- Product: Large Flashlight (dropdown menu)
- Quantity: 1
- Unit Price: \$12.20
- Discount: (empty field)
- Item Total: \$12.20
- Buttons: Cancel, OK

The background application window shows a header for "MoonLight Products" and a table with columns for Order Date, Ship Date, Ship method, and Country. The table contains one row with values: 5/26/95, 5/26/95, standard, and USA.

When you open a modal form, you must provide the user with the means to dismiss it. Normally, you close the dialog through one or more pushbuttons appearing on it. In Oracle Power Objects, you can designate a pushbutton on the modal form for this purpose through the pushbutton's **IsDismissBtn** property. When **IsDismissBtn** is set to TRUE, the form is hidden when the user clicks the pushbutton (that is, its **Visible** property is set to FALSE).

In Windows, users can also dismiss a dialog box by double-clicking the System Command box in the upper-left corner of the dialog box.

When dismissed, the form is not unloaded from memory, but hidden. The reason for this behavior is that dialog boxes are commonly used to enter setup information used in other parts of an application.

For example, in one dialog box the user might enter a user ID and password before accessing the rest of the application. By hiding this dialog, the application can continue to reference the user ID entered in a text field on this modal form.

If you want to close a modal form, you must call the **CloseWindow()** method on it, either before or after it is hidden.

➤ **Note:** You can also display simple dialog boxes using the Oracle Basic **MSGBOX** and **INPUTBOX** statements. For information about these statements, see the topics "MSGBOX Command", "MSGBOX Function", "INPUTBOX Command", and "INPUTBOX Function" in the On-Line Help.



## Forms and Window Styles

You can determine other aspects of the appearance and behavior of a form through the form's **WindowStyle** property. The setting for this property determines:

- The nature of the form's window.
- The user's ability to move and resize the form.

In many cases, you apply a particular window style to modal forms only. For example, the *Plain Dialog* window style is normally applied to a dialog, not a non-modal form.

The available window styles include:

<b>Format</b>	<b>Description</b>
<i>Standard Document</i>	A form that the user can resize and reposition.
<i>Fixed-Size Document</i>	A form that the user can reposition but not resize.
<i>Document Without Maximize</i>	Identical to the Standard Document style, but the form does not have a maximize button.
<i>Standard Dialog</i>	A form that the user cannot resize or reposition. In Windows, the dialog has a smaller border and no title bar; on the Macintosh, it has a thick border.
<i>Plain Dialog</i>	Like a Standard Dialog, except that it has a thick border surrounding it (Macintosh only).
<i>Alternate Dialog</i>	Like a Standard Dialog, except that it has a shadow beneath it (Macintosh only).
<i>Movable Dialog</i>	Like a Standard Dialog, except that the user can reposition it.
<i>Palette</i>	A movable form that the user cannot resize (Macintosh only).
<i>Palette Box</i>	Like Palette, except that the form has a Close pushbutton (Macintosh only).

## Controlling the Behavior of Forms

Oracle Power Objects provides several ways to control the behavior of a form, through several methods that open, close, or hide the form. The following table summarizes these methods and the tasks they perform:

<b>Method</b>	<b>Description</b>
<b>OpenWindow()</b>	Loads a non-modal form into memory and displays it.
<b>OpenModal()</b>	Loads a modal form into memory and displays it.

---

<b>Method</b>	<b>Description</b>
<b>HideWindow()</b>	Hides a form but does not unload it from memory.
<b>CloseWindow()</b>	Removes the form from the user interface and unloads it from memory.
<b>OpenPreview()</b>	Opens a form in Print Preview mode (see below).
<b>OpenPrint()</b>	Prints a form (see below).

For a complete description of each method, see its description in the Oracle Power Objects online help file.

## Printing a Form

To print a form, you call either of two methods, **OpenPrint()** or **OpenPreview()**, both of which are standard methods of forms. **OpenPreview()** opens the form in Print Preview mode, showing the user how it will appear when printed. The application displays the Print Preview toolbar, giving the user the ability to page through the printed version of the form, and ultimately print the form.

Optionally, you can print the form without previewing it, by calling the **OpenPrint()** method. When you call this method, the application first displays the system dialog for printing options.

When the user clicks **OK**, the application prints the form.

When you print a form, the application prints one copy of the form for every record associated with it, or just one copy if no records are currently displayed in it. The application starts a new page for each copy of the form.

## Query by Form

All applications you create with Oracle Power Objects give the user the ability to filter records displayed on a bound form. By entering conditions into a copy of the form, the user can remove all records that do not meet the specified conditions. For example, in a voucher entry form, the user could limit the vouchers displayed in the form to only those entered after a certain date, or only those whose dollar value exceeds a particular amount.

This capability is called *Query by Form*. Since Query by Form (QBF) is available only when the form displays records queried from a database, you can only use it when testing the form (Form Run-Time mode), testing the application (Application Run-Time mode), or running the compiled application (Standalone Run-Time mode).



To use QBF, the user enters criteria in a copy of the form, opened by clicking the Query by Form button on the Form Run-Time toolbar or choosing the **Database-Query by Form** menu command. The user can enter a different condition in each bound control, using SQL syntax. Effectively, the user is entering a collection of criteria that are linked together into a single WHERE

clause, which is used in the query that selects the form's information from the database. These criteria are linked together by logical AND operators—only rows that meet all specified criteria are displayed.

By default, QBF is part of all applications created with Oracle Power Objects; you do not have to do any development work to implement it. The user can use QBF on any bound container, including containers that appear within forms (for example, a repeater display).

However, the developer can disable QBF by disabling or overriding the menu command and toolbar buttons that provide access to QBF. For information on customizing menus and toolbars, see Chapter 14, “Menus, Toolbars, and Status Lines”.

## Using Query By Form

### ☆ To use Query By Form:



- 1 Click the QBF button on the Form Run-Time toolbar, or choose the **Database-Query By Form** menu command.

A duplicate form, the “Find What?” form, appears, displaying an empty control for every control appearing in the original form.

- 2 In the “Find What?” form that appears, enter the desired conditions. To enter conditions, use the QBF syntax described in the section “QBF Syntax” on page 11.18. You can add conditions in one or several controls.



- 3 Click the Apply Criteria button, or choose the **Database-Apply Query** menu command.

Oracle Power Objects immediately queries the recordsets displayed on the form, applying the specified conditions. The results of your query are displayed in the original form.

The results of Query By Form apply to all recordsets displayed on the form, including recordsets in a master-detail relationship.

## Entering Criteria

You enter criteria by setting the values of bound controls in the “Find What?” form (unbound controls are automatically disabled). You can enter criteria by setting the values in the following types of controls:

- **Radio Button Groups, Check Boxes, Popup Lists, and List Boxes.** If you select a value in one of these control types, only rows that have the selected value are returned.
- **Text Fields and Combo Boxes.** You can enter a criterion that is translated into a SQL condition, as described below.

When you enter criteria, you use SQL-like syntax for entering conditions, with a few exceptions. As part of this syntax, you can enter wildcards as well as literal strings. Wildcards are characters that represent one or more unspecified characters—for example, the underscore wildcard (\_) represents any one character, while the percent wildcard (%) represents any number of characters.

The following table summarizes some common criteria.

<b>Type of Criterion</b>	<b>Examples</b>
Simple value	SMITH 32 01-JAN-90
String with wildcard	SM_TH SM%
Value preceded by operator	>15.45 != JONES < 05-MAY-89 LIKE SM%
Null test	IS NULL IS NOT NULL
“in” test	IN (1, 2, 3) NOT IN (SMITH, JONES, KING)
“between” test	BETWEEN 1000 AND 5000

Type of Criterion	Examples
Compound criteria	>10.0 AND <22.5 LIKE SM% OR LIKE SP% SMITH or JONES or KING =21 or (>=11 and <=15)

## Some Considerations When Entering Conditions

- Do not specify the column name. It is automatically appended to the criterion you enter.
- If you enter a simple value (without an operator), Oracle Power Objects performs an equivalence (=) comparison, except for string values, for which Oracle Power Objects performs a LIKE (wildcard) comparison.
- Quotes are optional around literal values. However, if you use quotes, you must use single quotes.
- You can specify criteria in multiple bound controls. These criteria are joined into a single condition by the AND operator (only rows that match all criteria are displayed).
- You cannot include SQL functions or subqueries in your criteria.

## Clearing Query Conditions



You can clear the query conditions from the original form by clicking on the Query button on the Form Run-Time toolbar, or by choosing the **Database-Query App** menu command. The “Find What?” form is *not* closed when you clear the query conditions; it remains visible until you close it. The application also clears the query conditions entered by the user whenever the `Query()` or `QueryWhere()` methods of the form are called programmatically.

## Using QBF with Master-Detail Relationships

When a “Find What?” form is visible that contains both a master container and a detail container joined by a master-detail relationship, you can use QBF to find all of the master records that match criteria specified in the detail recordset. To use this feature, the primary key column in the master-detail relationship must be located in the detail recordset (the `LinkPrimaryKey` property of the detail container must be set to *Here (on Detail)*).

To find the master records associated with detail criteria, simply enter criteria in the detail container as described above. For example, if you had an “Employees” form that contained a “Department” detail embedded form, you could enter a department name of “OPER%” into the department name field of the “Departments” embedded form. When you applied the criteria, all employees in the “OPERATIONS” department would be displayed in the master form.

---

When a “Find What?” form is visible that contains only a detail container, and the primary key column in the master-detail relationship is located in the detail recordset, the link condition between the master and detail containers is temporarily relaxed when you apply query criteria. The detail recordset displays all rows that match the criteria you specified, not just the detail rows of the currently displayed master row.

## QBF Syntax

This section summarizes the syntax for specifying conditions in the “Find What?” form. This syntax can be entered only in text fields and combo boxes; as described earlier in this chapter, you use other techniques to specify conditions in radio buttons, check boxes, popup lists, and list boxes.

### Syntax of *expr*

```
simple_expr [ {AND | OR} expr ] | (expr) [ {AND | OR} expr ]
```

### Syntax of *simple\_expr*

```
value [ {comp_op value | [IS] [NOT] NULL | special_op } ]
```

### Syntax of *comp\_op*

```
= | != | <> | ^= | > | < | >= | <=
```

### Syntax of *special\_op*

```
[NOT] IN (value [, value ...] )  
| [NOT] BETWEEN value AND value  
| [NOT] LIKE value
```

### Syntax of *value*

*value* can be any string of characters surrounded by single quotes, or any string of characters delimited by white space, close parenthesis, or (in the case of IN), a comma.

## Queries, Conditions, and Forms: A Summary

For more about querying records, see Chapter 17, "Binding a Container to a Record Source".

You can control the ways in which a form queries records through:

- The **DefaultCondition** property
- The **QueryWhere()** method
- Query by Form

### The **DefaultCondition** property

When the application calls the **Query()** method on a form or other bound container, it applies whatever condition is specified in the **DefaultCondition** property of the container. You assign a string to this property that determines the **WHERE** clause of the query, even though you do not add the statement **WHERE** to the string. If no condition is specified for the **DefaultCondition** property, the application queries all records from the record source to populate the form's recordset.

For example, if you wanted to limit a query to employee records with an employee ID greater than 100, you would enter the following string for the **DefaultCondition** of the form displaying these records:

```
ENAME > 100
```

Here, **ENAME** is the name of a column in the recordset, not the name of the corresponding bound control on the form.

Note that you do not put quotes around the string when you enter a **DefaultCondition** through the Property sheet. However, if you were to change the **DefaultCondition** property programmatically, you would need to surround the string with quotes in the method code.

### The **QueryWhere()** method

Instead of calling the **Query()** method on a container, you can call the **QueryWhere()** method. This standard method of bindable containers takes one argument, a string specifying the contents of the **WHERE** clause for the query. Again, you do not need to add the statement **WHERE** to this string; the application adds it while building the SQL statement for the query.



The condition passed as an argument to **QueryWhere()** overrides any condition set through the **DefaultCondition** property.

For example, to query only those invoice records whose total value is greater than \$100, you might enter the following method code:

```
frmInvoiceEntry.QueryWhere("TOTAL > 100")
```

The following table summarizes when the **Query()** and **QueryWhere()** methods are called.

---

<b>Method</b>	<b>When Called</b>
 <b>Query()</b>	The user opens the form. The user clicks the Query button on the Form Run-Time toolbar. The user clicks the Apply Criteria button after entering QBF conditions. Method code calls the <b>Query()</b> method.
 <b>QueryWhere()</b>	Method code calls the <b>QueryWhere()</b> method.

For more information on these properties and methods, see the online help topics discussing each one.

## Query by Form

As described in the section “Query by Form” on page 11.14, the Query by Form feature of Oracle Power Objects applications lets the user specify conditions. These conditions are added to the condition set through the **DefaultCondition** property of the form. After entering conditions in the “Find What?” form and clicking the Apply Criteria button, the application calls the **Query()** method on the form. If conditions have been specified through **DefaultCondition** and the “Find What?” form, the application applies both sets of conditions.

Similarly, if the **QueryWhere()** method has been called on the form, the application adds the **QueryWhere()** condition to the conditions entered through the “Find What?” form.



# 12

---

## Reports

This chapter covers the following topics:

Overview .....	12.2
The Areas of a Report .....	12.2
Creating a Report .....	12.3
Designing Areas of a Report .....	12.4
Reports and Recordsets .....	12.7
Populating Controls on a Report .....	12.7
Working with Report Groups .....	12.10
Testing the Report .....	12.12
Printing a Report .....	12.13
Representing Master-Detail Relationships in a Report .....	12.15
Adding a Chart to a Report .....	12.17
Other Report Considerations .....	12.19

---

## Overview

For information on binding, see Chapter 17, “Binding a Container to a Record Source”.

In Oracle Power Objects, *reports* are bindable containers that let you preview and print data queried from a database. Therefore, you design reports using many of the same techniques used to design forms, including the techniques for binding the container and its controls to a record source.

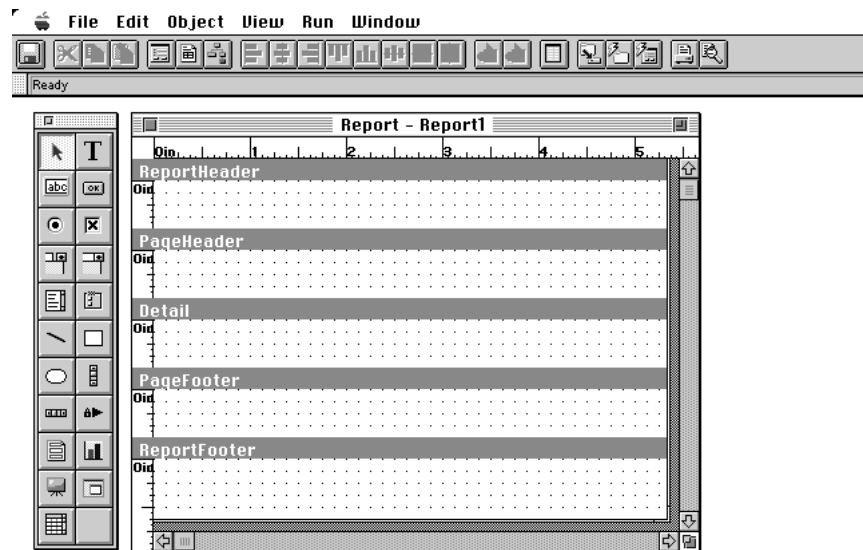
Report design has some special considerations for the following reasons:

- Unlike forms, reports are divided into separate areas (report header, page footer, detail, etc.), each of which has a different purpose.
- Most of the areas of a report are *repeated containers*, much like the panels of a repeater display. Therefore, you cannot write method code that attempts to deal with a single object within that container. However, you can use aggregate functions on the controls within report areas.
- Displaying master-detail relationships requires slightly different techniques than in forms.
- Some controls do not work as expected on a report, though you can add them to the report for display only.

This chapter describes how to create reports and summarizes some important options for report design.

## The Areas of a Report

Reports are divided into several functionally distinct areas, each of which performs a different task within the report. Each area is itself a named container object into which you place other application objects.



Reports consist of the following areas:

Area	Description
Report Header	Printed at the beginning of a report. The report header appears once, at the top of the report's first page.
Page Header	Contains all objects printed at the top of each page of the report. (The page header may or may not appear on the first page of the report).
Group Header	Printed at the beginning of each report group, and determines the column(s) used to define the report group. The group header appears between the Detail section and the page header. For more information on report groups, see the section "Working with Report Groups" on page 12.10.
Detail	Defines the body of the report, where the majority of information queried from the database is displayed. Any objects appearing within the Detail area repeat once for every record queried from the database.
Group Footer	Contains all objects that are printed at the end of a report group. The group footer appears between the Detail section and the page footer.
Page Footer	Contains all objects printed at the bottom of each page of the report. (The page footer may or may not appear on the last page of the report).
Report Footer	Contains all objects displayed at the end of a report. The report footer appears at the bottom of the report's last page.

The bulk of information in a report appears in the Detail area, while the rest of the report consists of different sorts of headers and footers. Some headers and footers display information relevant to rows (specifically, the group headers and footers). Other areas mark different pages of the report, or the beginning and end of the report.

The Group Header and Group Footer areas are extensions of a *report group*, a special feature of a report. You create a report group to determine how the report will sort records, so that all records having a matching value (such as department ID, salary grade, job classification) appear together. The defining feature of a report group is the column used for sorting records, specified through **GroupCol** property of the Group Header area. When you create a report group, Oracle Power Objects adds the Group Header and Group Footer areas to the report. Additionally, you can perform calculations on groups of records through controls appearing in the group header and footer.

## Creating a Report

Before setting up the individual areas of a report, you must create the report and bind it to a record source.

---

☆ **To create a new report:**



- 1 With an Application window as the active window, click the New Report button.  
A new report appears in the Oracle Power Objects desktop.
- 2 In the report's Property sheet, enter a new name for the report through its **Name** property.
- 3 If desired, enter a title for the report through the report's **Label** property.  
The title appears on the title bar of the report.

☆ **To bind a report to a record source:**

- 1 From the Session Designer window, select the table or view to be used as the report's record source.
- 2 Bind the report to this record source in one of three ways:
  - Drag the icon for the table or view onto the Detail area of the report. Controls bound to the columns in the record source appear in the Detail area.
  - Open the Table Designer or View Designer window for the record source and drag columns from this window onto the Detail area of the report. Controls bound to the selected columns appear in the Detail area.
  - In the report's Property sheet, enter the name of the session for the **RecSrcSession** property, and the name of the record source for the **RecordSource** property. Later, you can add the controls bound to columns in this record source.

Commonly, you create a view for a report, to simplify the task of joining related information from several tables in the report. The view lets you apply several SELECT criteria to rows queried for the view, such as ORDER BY, UNION, CONNECT BY, and START WITH. For more information, consult the SQL language reference manual for your database platform.

## Designing Areas of a Report

This section summarizes the general techniques for working with the different areas of a report, including:

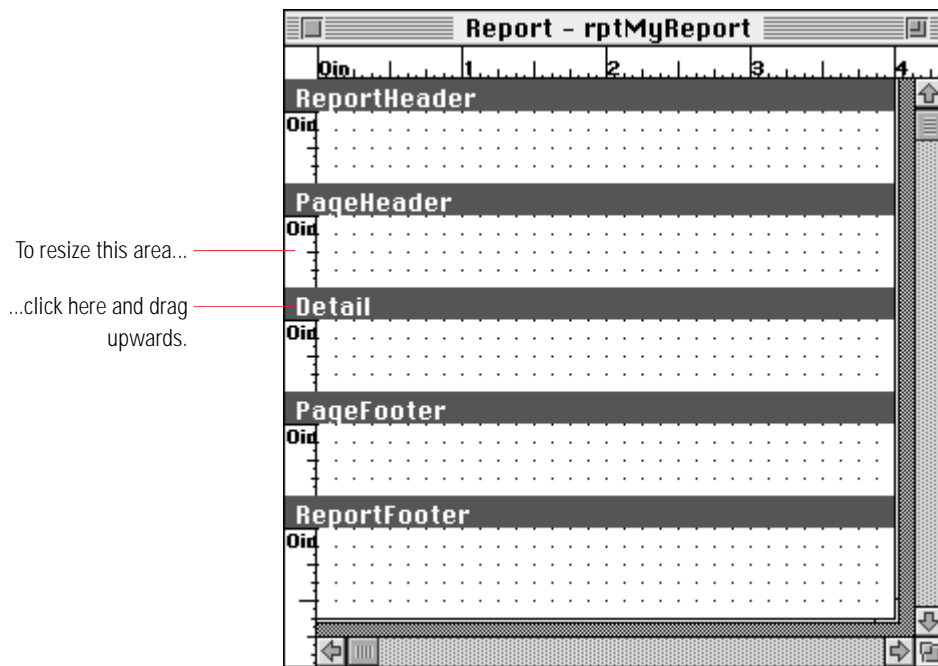
- Resizing an area of a report.
- Adding objects to a report area.

## Resizing Areas of a Report

You will often need to resize the areas of a report to fit the controls, static objects, and containers displayed in that area. You can resize each area vertically, but not horizontally. Therefore, you can change only the height of a report area, not its width (except by changing the width of the entire report).

☆ **To resize an area of a report:**

- 1 Open the Report Designer window.
- 2 Click on the title of the report area immediately beneath the area you wish to resize, or the horizontal line appearing to the right of this title.



- 3 Holding down the mouse button, drag this title bar up or down until the report area has the desired height.
- 4 Release the mouse button.

Note that a report area does not need to have any height at all. When resizing the report, you can move the area's title against the title of the area immediately beneath it, giving the area an effective height of 0 pixels or inches. In this case, the area does not appear at all when the report is printed.

---

All of the sections of the report are contained by the report; no section is contained within another section, including nested report groups.

## Adding Objects to a Report

You can add any kind of control, static object, or container to a report. However, the user cannot interact with these controls, so they are often useless within a report. The following application objects are of limited usefulness on a report:

<b>Object</b>	<b>Description</b>
Pushbuttons	The user cannot click a pushbutton.
Scrollbars	The user cannot move the thumb of a scrollbar.
List controls	The user cannot select items in a list box, combo box, or popup list. Generally, a text field is preferable to one of these controls in a report.
OLE objects	If the OLE object requires a media player (for example, a .WAV file player), then the user cannot review the data in the OLE object.
Repeater displays	Because it cannot display a usable scrollbar, a repeater display may contain more records than it can display in the space allotted on the report.
Embedded forms	An embedded form cannot display more than one record at a time. However, embedded forms can be useful when displaying a master-detail relationship (more information is provided in later in this chapter).

You add objects to a report in the same way that you add objects to a form.

### ☆ To add objects to a report:

- 1 Select the drawing tool corresponding to the desired type of object (for example, text field, line) from the Object palette.
- 2 Click on the report area where you wish the new object to appear,  
-or-  
Click and drag across the region of the report area where you wish the new object to appear.

### ☆ To move an object between report areas:

- 1 Select the object by clicking on it.
- 2 While keeping the mouse button depressed, drag the object to a different report area.
- 3 Release the mouse button.

For information on populating controls with data, see the section "Creating a Report" on page 12.3.

## Reports and Recordsets

Reports are bindable containers with many of the same recordset-related properties and methods as forms. Therefore, you can use many of the same techniques for working with a form's recordset when designing reports. These include:

- Defining conditions (filters) for the report to limit the records queried from its record source.
- Intercepting methods related to recordset operations.
- Displaying master-detail relationships on the report.

### Defining Filters for the Report

You can define filters for the report through the **DefaultCondition** property and **QueryWhere()** method of the report.

You use the **DefaultCondition** property to determine a default condition applied to the report's recordset. The application applies this condition when it opens the report and calls the **Query()** method to populate its recordset. You assign a value to the **DefaultCondition** property through the report's Property sheet.

Alternatively, you can call the **QueryWhere()** method on the report to filter records. This method takes a single argument—a string defining the SQL condition applied to the report's recordset. **QueryWhere()** requeries the report, overriding any condition set through **DefaultCondition**.

You can call **QueryWhere()** on reports in Print Preview mode only if they were opened with the **OpenPreview()** method.

For more information on **DefaultCondition** and **QueryWhere()**, see the “DefaultValue property” and “QueryWhere() method” topics in the online help.

For more information on **OpenPreview()** and Print Preview mode, see the section “Testing the Report” on page 12.12.

## Populating Controls on a Report

When you display controls on a report, you have three ways to populate them with values:

- Bind the control to a column.
- Use a derived value to populate the control with the result of an expression.
- Use the **SQLLOOKUP** function to populate the control.

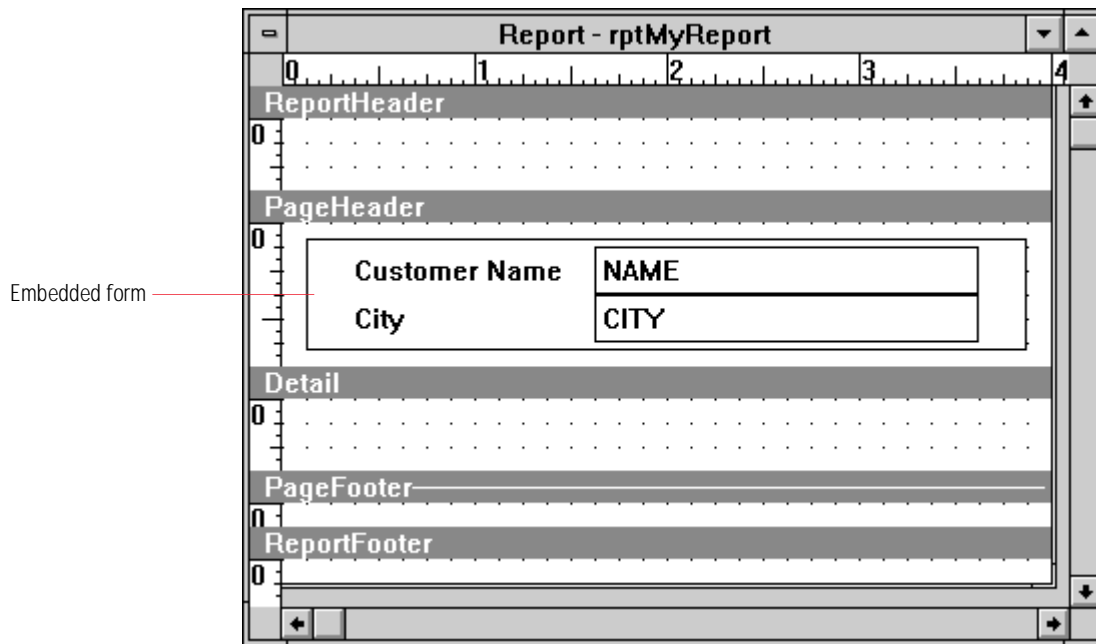
For information about the **SQLLOOKUP** function, see the section “The SQLLOOKUP Function” on page 9.21

---

## Binding Controls to Columns

You can bind controls in a report to columns in the report's record source. You bind the control by identifying the column to which it is bound through the **DataSource** property of the control, using the techniques described earlier in this chapter.

You can also add bound controls to a bound container (embedded form, user-defined class, or repeater display) appearing within the report. In these cases, the controls are bound to the container's record source, not the report's. This technique is often necessary for displaying master-detail relationships within the report, as described in the section "Testing the Report" on page 12.12.



For more information on displaying data in bound controls, see Chapter 17, "Binding a Container to a Record Source".

When displaying data in a control within a report, you must follow the same rules for displaying data in various control types on forms. For example, to properly display values queried from the database, a check box must have its **ValueOn** and **ValueOff** properties set to matching values found in the column to which the control is bound. For a list box to display meaningful values, you must set the **Translation** property of this control.



## Using Derived Values

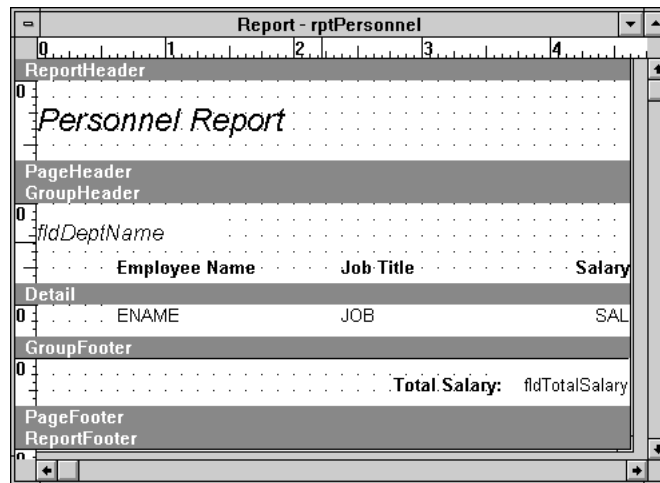
For more information on derived values, see the section "Control Values" on page 10.12.

You can also use derived values to populate controls, by entering an expression for the control's **DataSource** property. Derived values follow the same rules on reports as they do on forms.

Derived values that use aggregate functions impose a special limitation. A control using a derived value aggregate function cannot contain a reference to another control at the same level of the containment hierarchy. Instead, the control using the derived value must appear outside the container where the aggregate value appears.

For example, if you want to have a text field display a grand total of all employee salaries for a department, that field cannot appear within the Detail area of the report, where the individual salary values appear. Instead, you can place this text field within the Group Header or Group Footer, to ensure that the total appears for each department.

In the report shown below, the text field "fldTotalSalary" appears in the Group Footer area of the report, where it totals the salaries for all employees in a given department.

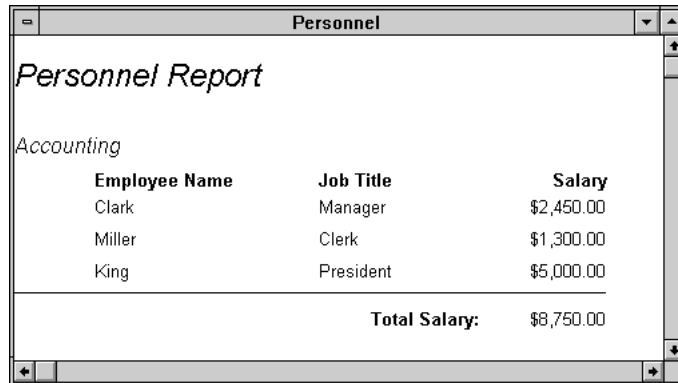


In this case, the **DataSource** property of "fldTotalSalary" is set to the following expression:

```
=SUM(rptPersonnel.Detail.SAL.Value)
```

---

When this report is run, it displays the following results:



Employee Name	Job Title	Salary
Clark	Manager	\$2,450.00
Miller	Clerk	\$1,300.00
King	President	\$5,000.00
<b>Total Salary:</b>		<b>\$8,750.00</b>

## Using SQLLOOKUP

For more information on SQLLOOKUP, see the section "The SQL-LOOKUP Function" on page 9.21.

The third way to populate report controls is by using the SQLLOOKUP function for the **DataSource** property of a control. In this case, the result of a SQL query populates the control. Therefore, if the query is designed to return a value from a column, you want to write the query so that it returns a value from no more than one row.

For example, in the case of a listing of employees by department, you may want to display the name of the division in which they work, instead of the numeric department ID. In this case, you would use SQLLOOKUP to display the text from the LOC column of the DEPT table, corresponding to the DEPTNO value for each employee. When the report is printed, the name of the location (for example, "DALLAS") appears, instead of the numeric department ID (for example, 10).

## Working with Report Groups

You organize information on a report through *report groups*. A report group separates information according to the values in one column of the report's record source, so that you can see records grouped by that column. For example, you can view employee records from the EMP table grouped by department, by defining a report group that uses the DEPTNO column for sorting records. The report group adds two new areas, the group header and group footer, to the report.

The column used to organize records is called the *group-by column*. You specify this column through the **GroupCol** property of a report group.

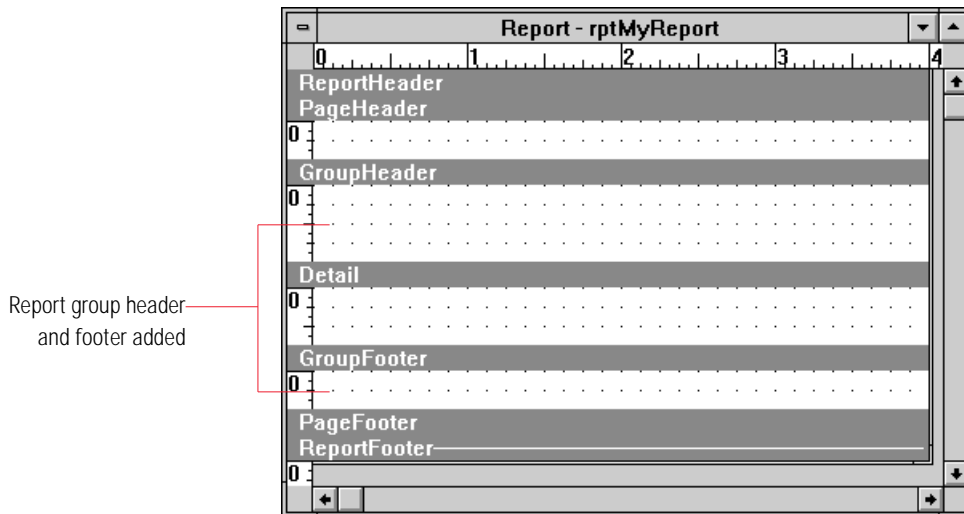
When you create a report group, the report displays all records belonging to one value in the specified column before printing another group of records sharing the same value in the group-by column. In our example, if you specify DEPTNO as the group-by column, the report prints all employee records that have a DEPTNO value of 10 (that is, belonging to one department) before printing employee records with a DEPTNO of 20.

☆ **To create a report group:**



- 1 Open the Report Designer window for the report.
- 2 Choose the Report Group tool from the Object palette.
- 3 Click on the report.

A new report group header and footer appears in the report.



- 4 Set the **GroupCol** property of the report group to the name of the column you wish to use as the group-by column.

Note that you can create nested report groups, by adding a new report group to an existing group header or footer. Each report group sorts the records for the report group immediately beneath it; the innermost report group then sorts records appearing within the Detail area of the report.

**Report Groups and Recordsets**

Each report group has its own recordset. Therefore, the group header and group footer, which jointly represent the report group, share the same recordset. Additionally, you can share the recordsets between these objects and some other object in the report, such as a chart control.

---

## Testing the Report

If you have bound the report to a record source, and bound columns appear within the Detail area of the report, you can test the report.

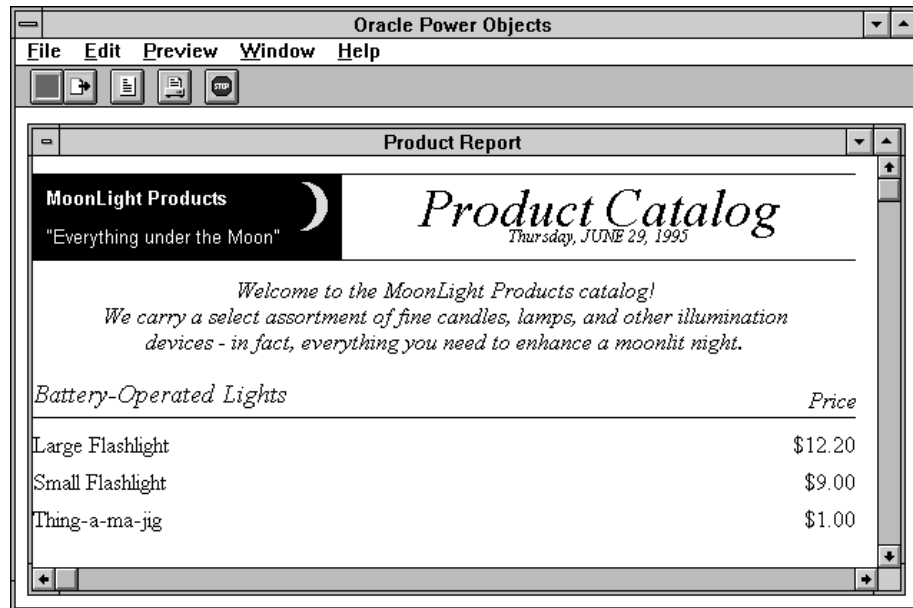
☆ **To test a report:**

1 Open the Report Designer window for the report.



2 Click the Run Form button, or choose the **Run-Run Form** menu command.

The report now appears in *Print Preview mode*, described in the next section. You can now see how the different areas of the report will appear, and then print a copy of the report.



If the only bound controls appear in the Detail area of the report, then the report prints a simple listing of all queried records, listing in ascending order according to the field designated as the primary key in the record source for the report. However, you can use report groups to better organize this information, as described later in this chapter.

## Printing a Report

When planning how to print a report, you have several options, including:

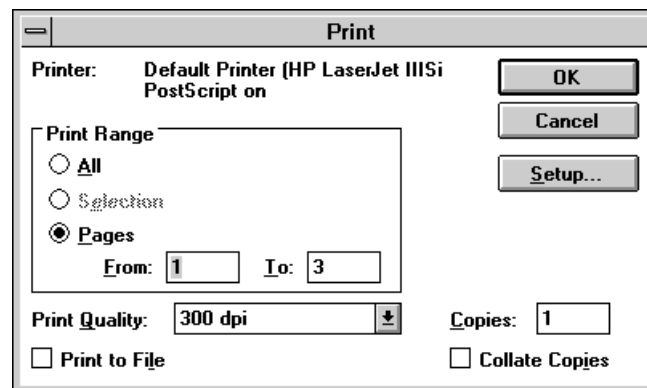
- Whether to preview the report before printing it.
- Whether to print a page break when the report reaches a new value for the group-by column in a report group.
- Whether to print headers and footers.

This section describes these options in detail, including the properties and methods related to printing a report.

### Standard Methods for Printing a Report

Reports have two standard methods for printing, `OpenPrint()` and `OpenPreview()`. `OpenPrint()` prints the report immediately, while `OpenPreview()` gives the user the ability to view a copy of the report before printing it. `OpenPreview()` displays the report in Print Preview mode, as described in the next section.

In both cases, the application displays the printer options dialog for the environment before printing the report. For example, in Windows, the dialog shown below appears when you print a report:



At this point, you can print the report by clicking the **OK** button, or cancel the task by clicking the **Cancel** button.

### Previewing a Report

Oracle Power Objects gives you the option of viewing a report in Print Preview mode before printing it. In this mode, you see a copy of the form as it will appear when printed.



You preview a report by clicking on the Run Form button or the Preview Report button, or by choosing the **Run-Run Form** menu command. In Print Preview mode, you cannot interact with the contents of the form. However, the Report Run-Time toolbar has several buttons for browsing and printing the report, as shown in the following table:

<b>Button</b>	<b>Name</b>	<b>Description</b>
	Previous Page	Moves to the previous page of the report. Available only if the report has more than one page.
	Next Page	Moves to the next page of the report. Available only if the report has more than one page.
	View Full Page	Sizes the report to display a full page on the screen.
	Print	Prints the report.

## Starting a New Page

When a report reaches the bottom of a printed page, the application starts a new page for the report. You can also instruct the application to start a new page when it reaches a new value for the group-by column in a report group. For example, in a report of employee records from the DEPT table in which DEPTNO is the group-by column, you can begin a new page when the report finishes printing records from one department and starts printing from another.

A property of the group header, **PageOnBreak**, determines whether the report begins a new page when the value in the group-by column changes. If **PageOnBreak** is set to True, the report begins a new page when it reaches a new value; if the property is set to False, the report continues printing on the same page.

## Printing Headers and Footers

If a header or footer has any height at all, it appears in the report when the application prints the report. However, in some parts of the printed report, the header need not appear. For example, the page header frequently does not appear on the first page of a report. You set one of the following properties to determine whether the header or footer appears.

<b>Property</b>	<b>Applies To</b>	<b>Description</b>
<b>FirstPgHdr</b>	Page Headers	Determines whether the page header appears on the first page of the report. If set to True, the page header does appear on the first page; if set to False, it does not.

Property	Applies To	Description
FirstPgFtr	Page Footers	Determines whether the page footer appears on the first page of the report. If set to True, the page footer does appear on the first page; if set to False, it does not.
LastPgFtr	Page Footers	Determines whether the page footer appears on the last page of the report. If set to True, the page footer does appear on the last page; if set to False, it does not.

## Representing Master-Detail Relationships in a Report

A report's structure is different from that of a form, so you use slightly different techniques for representing master-detail relationships within the report.

Commonly, you want to display the master records at the top of a page, with the details appearing beneath the master record. To organize master and detail records in this fashion, the Detail area of the report usually displays the detail records. You then have two options for displaying information from master records in a group header, using `SQLLOOKUP` or a bound container.

### Using `SQLLOOKUP`

You can use the `SQLLOOKUP` function to define the `DataSource` property of one or more controls in the group header. Each of these control will then display information queried from a column in the table or view that contains master records. The report fetches a new set of values from the master record source every time it reaches a new primary key value.

---

For example, in the report shown below, information from DEPT appears in the group header, while information from EMP appears in the Detail area. When the report is printed, each report header prints the department number and location from the DEPT table, followed by the employees working at that department.

The screenshot shows a report window titled "Report1". The report is organized into a group header and a detail area. The group header contains a table with two columns: "Department" and "Location". The "Department" column has the value "20" and the "Location" column has the value "Dallas". Below the group header is the detail area, which contains a table with three columns: "Employee ID", "Name", and "Job". The detail area lists six employees: SMITH (CLERK), ADAMS (CLERK), FORD (ANALYST), SCOTT (ANALYST), and JONES (MANAGER). The report window has standard scroll bars and a title bar.

Department	Location
20	Dallas

Employee ID	Name	Job
7369	SMITH	CLERK
7876	ADAMS	CLERK
7902	FORD	ANALYST
7788	SCOTT	ANALYST
7566	JONES	MANAGER

In this example, the group-by column for the Group Header area of the report is DEPTNO, which appears in both the DEPT and EMP tables. To print the location with the department number, the developer has added a text field to the Group Header that uses the following expression for its **DataSource** property:

```
=SQLLOOKUP(sesMySession, "select LOC from DEPT where LOC =" &  
    repDepartments.GroupHeader.fldDeptNo)
```

In this code, *fldDeptNo* is the name of the text field displaying the department number in the Group Header. Every time the report generates a new group header, the report displays the location name (queried from the LOC column in DEPT) corresponding to the new value for the group-by column, DEPTNO.

## Using a Bound Container

Another way to display information from master records in a group header is to add a bound container to the Group Header area. Effectively, the bound container (normally an embedded form) is a detail of the Group Header area, though it displays master records. However, since the Group Header appears above each group of records displayed in the Detail area, master and detail records appear in their normal representation (master record first, followed by associated detail records).



The advantage of this technique for displaying master-detail relationships is that you can easily present bound controls representing several columns from the master recordset, using Oracle Power Objects' drag-and-drop binding capabilities. Otherwise, you would need to write several SQLLOOKUP statements for the **DataSource** properties of these controls.

For more information on master-detail relationship properties, see Chapter 18, "Defining Master-Detail Relationships".

To make this master-detail relationship work, you set three properties of the container added to the Group Header area:

Property	Setting
<b>LinkDetailColumn</b>	The column specified as the group-by column for the report group. This column is named in the <b>GroupCol</b> property of the Group Header, and is the foreign key in the master-detail relationship.
<b>LinkMasterColumn</b>	The column in the master recordset used as the primary key. This column must exist in the table or view identified as this container's record source, but it does not have to be represented by a bound control in the report.
<b>LinkMasterForm</b>	The name of the Group Header area of the report. The default name is <code>GroupHeader</code> .

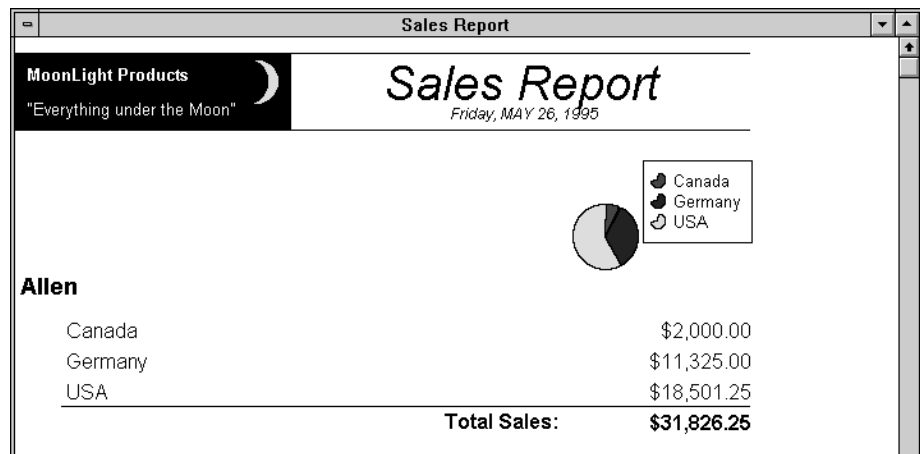
When you print or preview the report, the bound container in the Group Header area displays a new master record every time the group-by column's value changes to display a new group of detail records.

## Adding a Chart to a Report

For more information on chart controls, see the section "Chart Controls" on page 10.16.

Charts are a common feature of reports. Oracle Power Objects enables you to add a chart control to a report to graphically display information relevant to all or part of a report.

The range of data you wish to portray in the chart determines where you place the chart control.



---

## Charts for Report Groups

If you want to display a chart for each report group, you must add the chart to the Group Header or Group Footer area of the report. The chart then becomes a detail of this report area.

☆ **To create a chart for a report group:**



- 1 Choose the Chart tool from the Object palette.
- 2 Draw the chart within the Group Header or Group Footer area of the report.
- 3 Set the following properties of the chart control:

<b>Property</b>	<b>Setting</b>
<b>LinkDetailColumn</b>	The foreign key in the chart's record source.
<b>LinkMasterColumn</b>	A matching column in the record source for the report.
<b>LinkMasterForm</b>	The name of the Group Header or Group Footer area of the report.

The chart then appears once for every report group, displaying information relevant only to that group.

## Charts for the Entire Report

If you want to create a chart that summarizes information from the entire report, you commonly place the chart at the beginning or end of the report. In Oracle Power Objects, you place the chart in the Report Header or Report Footer area of the report.

☆ **To create a chart for an entire report:**



- 1 Choose the Chart tool from the Object palette.
- 2 Draw the chart within the Report Header or Report Footer area of the report.
- 3 Set the following properties of the chart control:

<b>Property</b>	<b>Setting</b>
<b>LinkDetailColumn</b>	The foreign key in the chart's record source.
<b>LinkMasterColumn</b>	A matching column in the record source for the report.
<b>LinkMasterForm</b>	The name of the Report Header or Report Footer section of the report.

## Charts for Individual Records

If you want to display a chart relevant to a single record, add the chart to the Detail area of the report. You must define the chart as a detail container of the Detail area of the report, so that the information displayed in the chart is relevant only to the record loaded into the Detail area of the report.

☆ **To create a chart for a single record:**



- 1 Choose the Chart tool from the Object palette.
- 2 Draw the chart within the Detail area of the report.
- 3 Set the following properties of the chart control:

Property	Setting
<b>LinkDetailColumn</b>	The foreign key in the chart's record source.
<b>LinkMasterColumn</b>	A matching column in the record source for the report.
<b>LinkMasterForm</b>	The name of the Detail section of the report (normally, <i>Detail</i> ).

The chart then repeats once for every record displayed in the Detail area of the report, displaying data relevant only to that record.

## Other Report Considerations

When designing reports, you should take the following additional issues into consideration.

### Page Width

On high-resolution monitors, a wide report that fits within the application window may not fit on an 8.5" by 11" sheet of paper. Therefore, you should size your reports to fit within the printed page, not within the interface space available to the application. By specifying the **SizeX** and **SizeY** properties of the report in inches, you can easily size the report to fit within a printed page.

### Fonts and Reports

As with forms, you should be sure that the client systems on which your application is installed have the fonts used in your report. If the application looks for a font that is not installed, it will try to use the substitute font defined for the environment. In this case, the report may appear quite different from your original design. Whenever possible, you should use fonts that are common to

---

an operating system, such as Arial on Microsoft Windows, or Times on the Macintosh. Additionally, you should choose a scalable font, such as a PostScript or TrueType font, for all objects within a report.

## **Graphics in Reports**

You can display graphics within reports, using picture objects and OLE objects. For example, you may want to display an employee's photograph as part of a report. When displaying graphics, you should remember that:

- Performance will suffer, because querying graphics slows down any database application.
- The printer may not be able to print the graphics at the same resolution at which they are displayed in the application.

# 13

---

## Classes

This chapter covers the following topics:

Overview .....	13.2
The Object Inheritance Hierarchy .....	13.3
Classes as Containers .....	13.8
Developing Classes .....	13.9
A Sample User-Defined Class .....	13.10
Subclasses .....	13.12

---

## Overview

In Oracle Power Objects, a class is any definition of an object used to create copies (or *instances*) of that object. Making a copy of the class is often called *instantiating* the class. In object-oriented terminology, the *master class definition* determines the characteristics and behavior of the object, which are then inherited by all instances of the class.

For example, in Oracle Power Objects, a master class definition of pushbuttons determines the appearance and behavior of all pushbuttons you create. When you add a pushbutton to a form, you are creating an instance of the master class definition of pushbuttons.

## Standard and User-Defined Classes

Pushbuttons are an example of a *standard* class, predefined within Oracle Power Objects. Each pushbutton you add to a form inherits the standard properties and methods of pushbuttons, as well as their standard appearance and behavior, from the master class definition. Other types of objects that have properties and methods (forms, sessions, check boxes, recordsets, etc.) are also examples of standard classes provided with Oracle Power Objects. You cannot change the list of properties and methods assigned to standard classes by default, although you can add properties and methods to individual instances of a standard class.

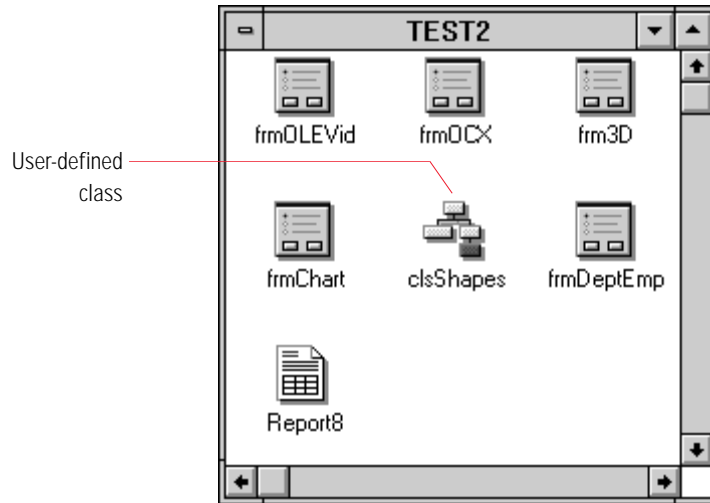
You can also create *user-defined classes*, a special kind of bindable container. You can add instances of the user-defined class to other containers. The user-defined class can contain controls and static objects, and it can be bound to a record source in the same fashion as other bindable containers.

Unlike other bindable containers, however, you must first create the user-defined class (essentially, a master class definition), and then add instances of it to other containers. These instances are part of the *object inheritance hierarchy*. When you make a change to the master class definition, the instances of the class inherit these changes.

User-defined classes simplify development by allowing you to create reusable application objects. For example, you can create a set of **OK** and **Cancel** pushbuttons as a user-defined class. You can add instances of this class to dialogs. Doing so eliminates the need to create the same set of pushbuttons repeatedly, every time you create a dialog. When you make a change to the master class, such as an improvement in the method code executed when the user clicks the **OK** pushbutton, all instances of this user-defined class inherit the changes.

For information about the object inheritance hierarchy, see the section “Object Inheritance Hierarchy” on page 3.31.

Once created, user-defined classes appear in the Application window, along with forms, reports, and OLE objects. You use a class in much the same way as a bitmap or OLE object, adding these reusable application objects to forms, reports, and other user-defined classes.



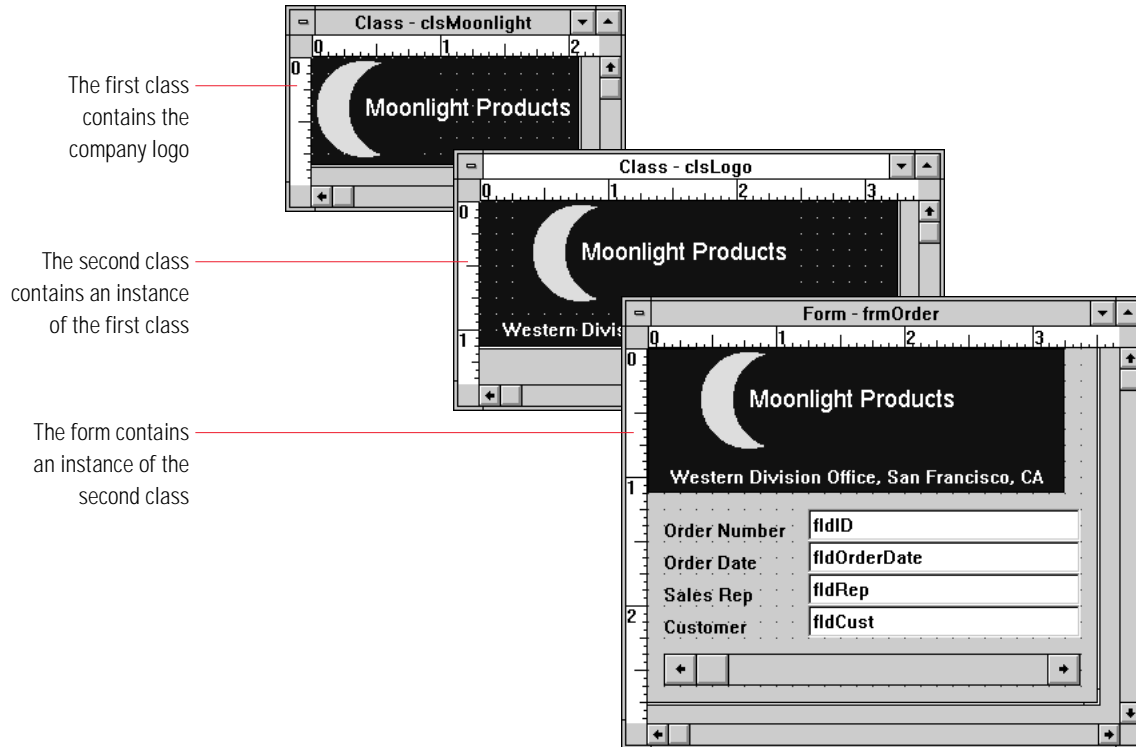
This chapter describes how you can work with user-defined classes to define reusable application objects.

## The Object Inheritance Hierarchy

For more about the object inheritance hierarchy, see the section "Object Inheritance Hierarchy" on page 3.31.

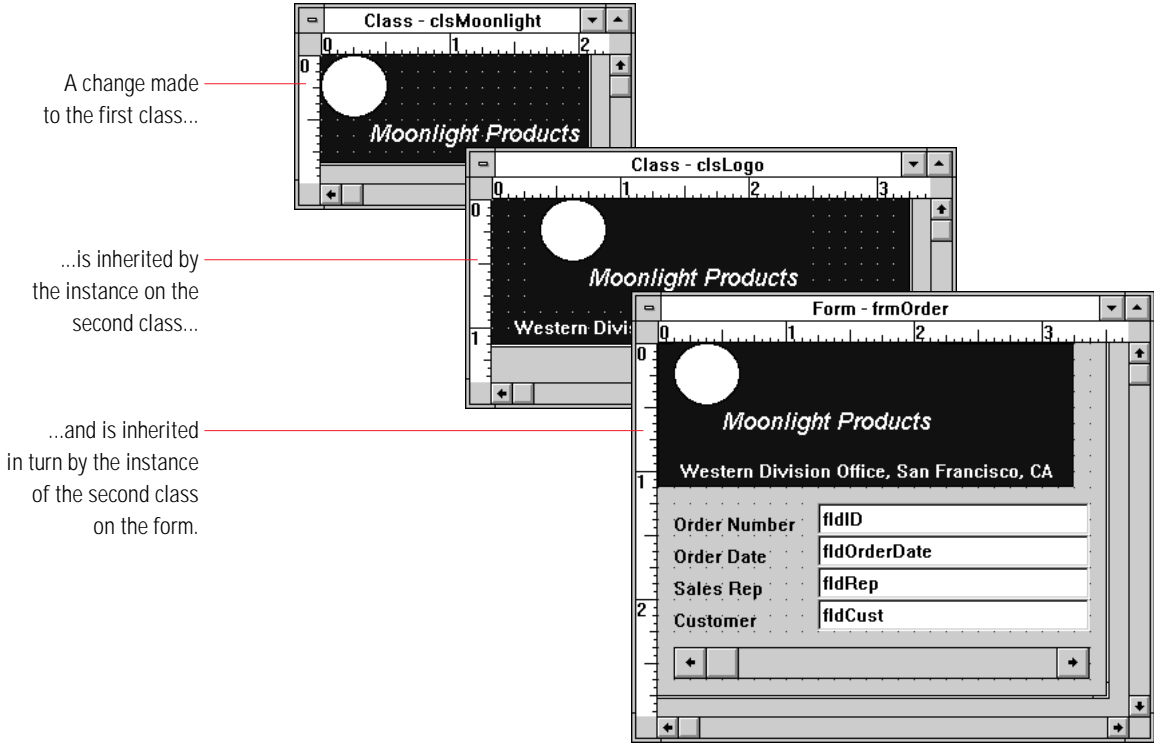
To work with classes, you need to understand how inheritance works. The *object inheritance hierarchy* determines how changes propagate from the master class definition to instances of the class. The changes cascade down the object inheritance hierarchy, in a relationship called the *chain of propagation*.

To illustrate the object inheritance hierarchy, shown below are two user-defined classes and a form. One class, displaying the company logo, appears within the other, the label for a particular company location. The label class has been added to an order entry form used at that location.

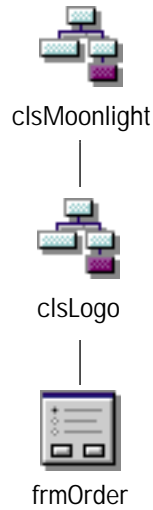




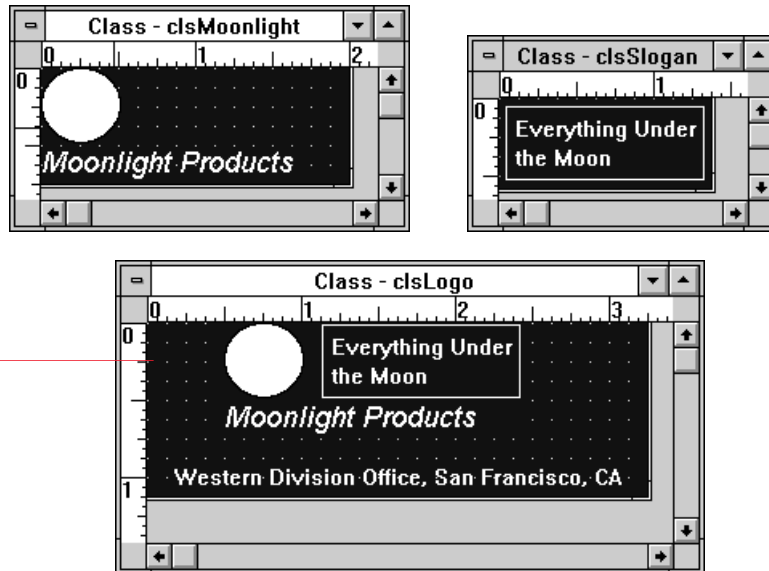
The instance of the logo class normally inherits any changes made to the company logo. So, too, does the instance of the logo class in the instance of the label class, appearing on the form.



To help you visualize this relationship, the diagram below illustrates the object inheritance hierarchy in this example.

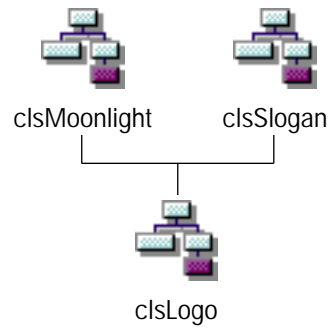


The hierarchy can have as many levels as you wish, with instances of several different classes appearing within the same master class definition. For example, we could add an instance of another user-defined class, displaying the slogan of the company, in the company office label class. The company office label would then include instances of two separate classes.



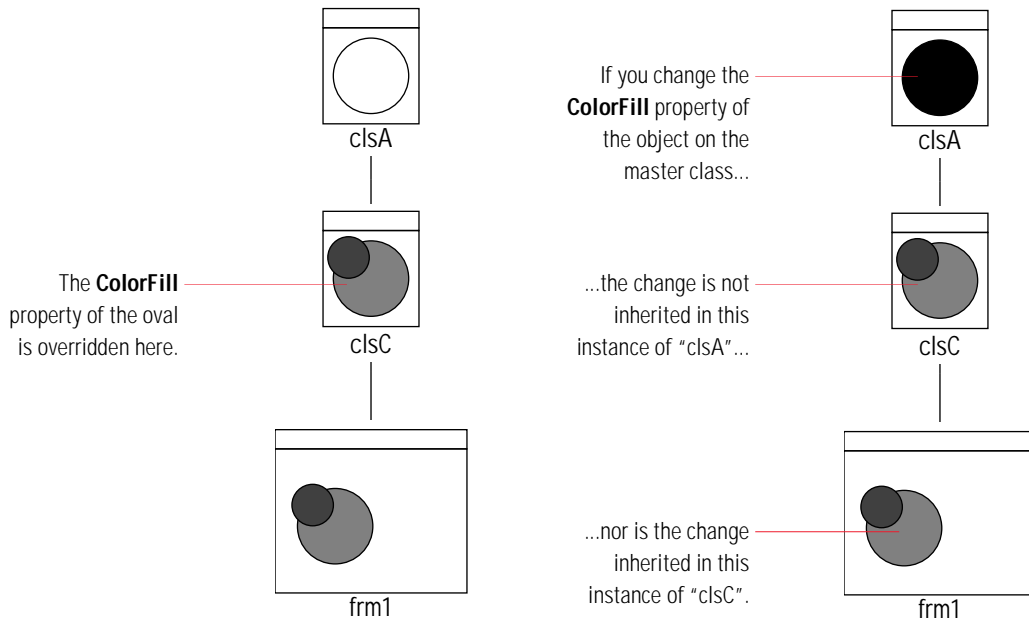
This class contains instances of two other classes.

The diagram below illustrates the object inheritance hierarchy in this example.



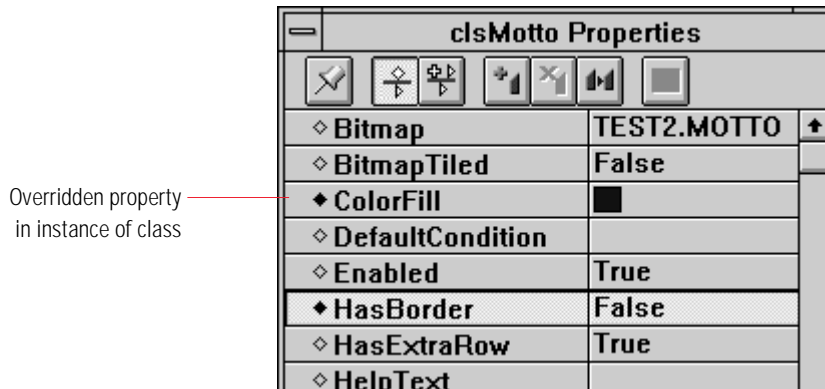
### Editing an Instance of a Class

Once you change a property or method in an instance, that instance no longer inherits subsequent changes to the same property or method in the master class. Additionally, if the instance appears within another class, the instance of the first class that is nested within the instance of the second class does not inherit the changes.



- You override a property when you enter a new setting for the property through the Property sheet at design time, or when you change the setting programmatically at run time. When you override a property, the diamond-shaped symbol next to its name is filled in.


- You override a method when you enter method code in the code window in the Property sheet. Note that the code window in the instance appears empty—code from the master class definition does not appear in the code window. Even if you paste into this window the exact code from the master class definition, the appearance of *any* method code in the instance’s code window breaks the chain of propagation. When you override a method, the arrow-shaped symbol next to its name is filled in.



### Reinheriting Properties and Methods

After you override a property or method in an instance, you can instruct an instance to reinherit the current properties and methods of the master class definition, erasing the modifications. The chain of propagation is no longer broken, and the instance then inherits subsequent changes to that property or method in the master class.

☆ **To reinherit the property or method of the master class:**

- 1 Select the overridden property or method on the instance’s Property sheet.
- 2  At the top of the Property sheet, click the Reinherit button.  
The instance then reinherits the setting for that method or property from the master class.

## Classes as Containers

User-defined classes are bindable containers, like forms and reports. However, unlike these containers, they cannot appear on their own, within a window—instead, you add them to other containers, just as you add embedded forms, repeater displays, rectangles, and ovals to containers. Therefore, user-defined classes have none of the properties or methods associated with windows (for example, `OpenModal()` and `WindowState`).

For information about the object containment hierarchy, see the section "Object Containment Hierarchy" on page 3.12.

Because user-defined classes are containers, they and the objects appearing within them are part of the *object containment hierarchy*.

You can get an object reference to the container in which an instance of the class appears using the **GetContainer()** method in Oracle Basic. For example, if you want to trigger the **ValidateRow()** method on the form in which a user-defined class appears when the user clicks a pushbutton, you would add the following code to a method of the class instance:

```
GetContainer().GetContainer().ValidateRow()
```

In this case, you use the **GetContainer()** method first to identify the class instance in which the pushbutton appears, and the second time to identify the form in which the instance appears.

### Object References to Master Class Definitions

You cannot resolve object references to master class definitions through Oracle Basic. Therefore, you cannot make run-time changes to a master class definition that its instances then inherit.

## Developing Classes

This section summarizes the basic development tasks associated with user-defined classes.

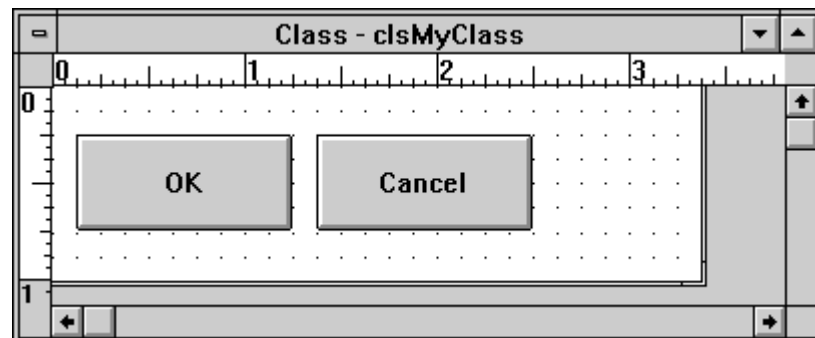
### Creating User-Defined Classes

☆ **To create a user-defined class:**

- 1 Select the Application window.
- 2 Click the New Class button, or choose the **File-New Class** menu command.



A new user-defined class appears within its Class Designer window.



---

You can then begin adding objects to the user-defined class in the same fashion as you would add application objects to any container.

## Adding Objects to a Class

In addition to controls and static objects, you can also add other user-defined classes to a class. All of these objects then appear on all instances of the class, inheriting the properties and methods assigned to them through the master class definition.

## Adding an Instance of a Class

Once you have created a user-defined class, you can add instances of it to forms, reports, and other user-defined classes.

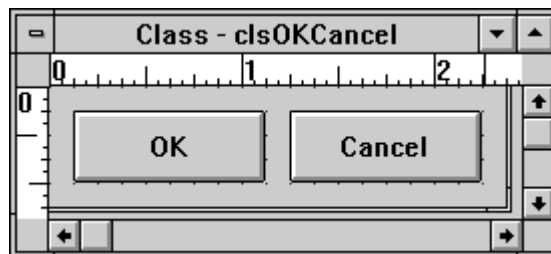
☆ **To add an instance of a class:**

- 1 Open the Designer window for the object to which you wish to add the instance.
- 2 From the Application window, drag the icon for the user-defined class and drop it on the container.
- 3 Size and position the class, as desired.
- 4 Make any additional modifications to the instance.

When you make changes to a property or method of the instance or of any object contained by the instance, the icon for the property or method becomes filled in.

## A Sample User-Defined Class

To illustrate how to employ user-defined classes in an application, the following figure shows the **OK** and **Cancel** button class described earlier in this chapter. The user-defined class is just large enough to encompass the two pushbuttons.



Normally, **OK** and **Cancel** buttons have the following behavior when they appear on a dialog:

- When the user clicks the **Cancel** button, the dialog disappears, and the application takes no action.
- When the user clicks the **OK** button, the dialog disappears. In this case, the application may take an action, or it may save some setting entered through the dialog.

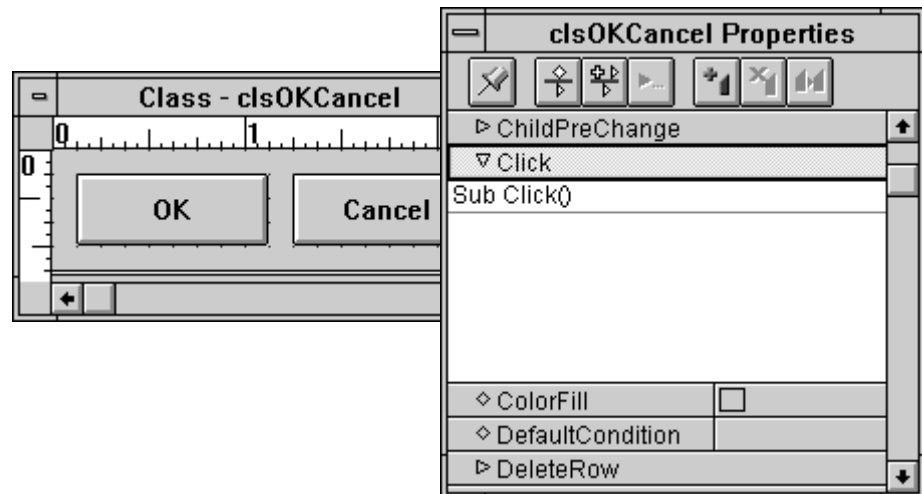
To simulate this behavior, the **Cancel** button has the following method code entered in its **Click()** method, to respond when the user clicks the pushbutton:

```
Sub Click()
    GetContainer.GetContainer.HideWindow()
```

For information about working with dialogs and forms, see Chapter 11, "Forms".

In this case, the dialog is hidden, but no changes occur in the application. The **HideWindow()** method hides the form but does not close it, so that the application can refer to settings entered in them.

The **OK** pushbutton has no method code entered for it, since the behavior of this pushbutton varies depending on the dialog on which it appears. In some cases, the application takes an action when it is clicked; in others, it simply remembers a setting entered in the dialog. Although in both cases the window is hidden, you should not enter the call to **HideWindow()**. Any code defining the behavior of the pushbutton in an instance of the class would override the call to **HideWindow()** in the **Click()** method of the master class. Therefore, you should simply add the call to **HideWindow()** in the method code you add to the **Click()** method of the *instance*.



Once you have finished defining it, you can add instances of the **OK** and **Cancel** class to dialogs in your application. After adding the instance, you then enter method code to the **Click()** method of that instance's **OK** pushbutton defining what happens when the user clicks this button.

Later, you may want to make modifications to the user-defined class, such as adding further code to the **Click()** method of the **Cancel** pushbutton, or altering the appearance of the pushbuttons. Every instance of the class then inherits these changes (unless you have already overridden properties or methods of the **Cancel** pushbutton).

---

## Subclasses

You can create a subclass of a user-defined class. The subclass acts as a user-defined class on its own, but it is effectively a copy of the master class definition. The subclass contains all the objects appearing within the master class definition, and it inherits the property and method settings of the master class definition. As with an instance, you can break the chain of propagation by changing the settings for properties and methods in the subclass.

Subclasses give you the ability to create modified versions of the same user-defined class that continue to inherit changes to the master class definition. Instances of the subclass inherit changes to the subclass and the master class definition, subject to the rules of the object inheritance hierarchy.

☆ **To create a subclass:**

- 1 Select the icon for a user-defined class in the Application window.
- 2 Choose the **Edit>Create Subclass** menu command.

The Class Designer window for the subclass then appears, displaying the new linked copy of the master class definition.



# 14

---

## Menus, Toolbars, and Status Lines

This chapter covers the following topics:

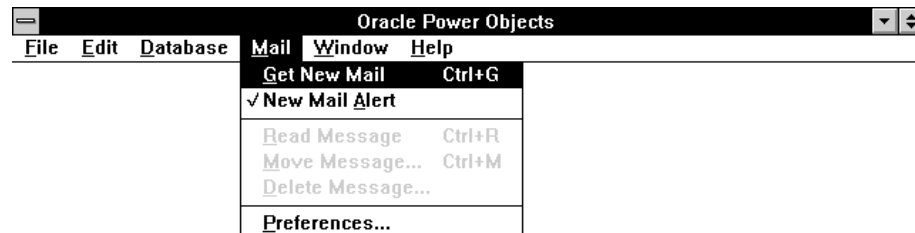
Overview .....	14.2
Menus .....	14.3
Toolbars .....	14.20
Status Lines .....	14.34
Properties and Methods .....	14.44

---

## Overview

*Menus*, *toolbars*, and *status lines* are graphical components you can add to your application to provide the user with commands, shortcuts, and status information.

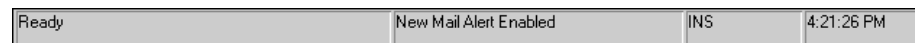
*Menus* provide lists of commands that the user can select to execute an action.



*Toolbars* provide buttons that the user can click to execute an action. Toolbars typically provide graphical shortcuts for commonly used menu commands.



*Status Lines* provide summary information to the user. For example, they can include nonmodal messages to the user (such as indicating when an action has completed).



The objects described in this chapter are *in-memory objects*—they exist only at run time. You can display them only when you invoke design run time by clicking the Run Form or Run Application button, or when you run a compiled application in standalone run time.

Because menus, toolbars, and status line objects do not exist at design time, you do not create and modify them as you do most other types of objects. Instead, you must execute Oracle Basic method code the first time they are displayed, usually in the **OnLoad()** method of your application or the **InitializeWindow()** method of a form or report. You also write method code to customize the appearance and behavior of these objects, usually in the **TestCommand()** and **DoCommand()** methods of your application or of individual forms or reports in your application.

You follow the same general steps to create menus, toolbars, and status lines in your application:

- 1 Create the object using the Oracle Basic NEW operator.

The NEW operator returns a reference to the newly created object, which you can store in a variable or property of datatype *Object*.

- 2 Add items to the object.

For example, you must add menus to a menu bar object, menu items to a menu object, buttons to a toolbar object, and panels to a status line object.

**3** Associate the object with a form or report.

The object you create is not actually displayed unless you associate it with a form or report.

**4** Add method code to handle user interaction with the object.

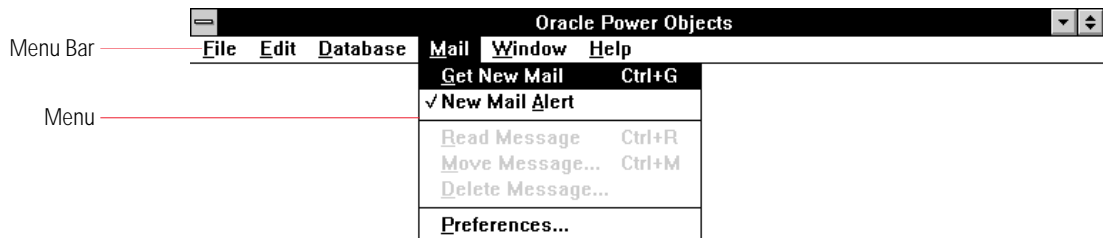
You update the status of the object's items by adding method code to the **TestCommand()** method of a form or application (for example, to determine whether menu items and toolbar buttons appear enabled or disabled).

You respond to user actions by adding method code to the **DoCommand()** method of a form or application (for example, to execute commands when the user chooses a menu item or clicks on a toolbar button).

The specific steps you should follow to create each type of object are described in the following sections.

## Menus

Menus are contained in a *menu bar*, which is displayed at the top of the application window (in Windows) or at the top of the screen (on Macintosh). Each window (form or report) in your application can display its own menu bar; when the window is active, the associated menu bar appears in the appropriate location.



By default, forms and reports in run-time mode display a menu bar that provides access to many common commands. This section describes how to customize the default menu bars, either by adding custom items to the menus that appear by default, by adding or by replacing them altogether.

An Oracle Power Objects menu bar can contain three types of menus:

**System default menus.** These are standard menus for an application in the operating system where your application is running. System default menus on the Macintosh are Apple, File, and Edit. System default menus in Windows are File, Edit, Window, and Help.

**Application default menus.** These are standard menus for the type of Oracle Power Objects window being displayed. Examples of application default menus are the Database and Preview menus.

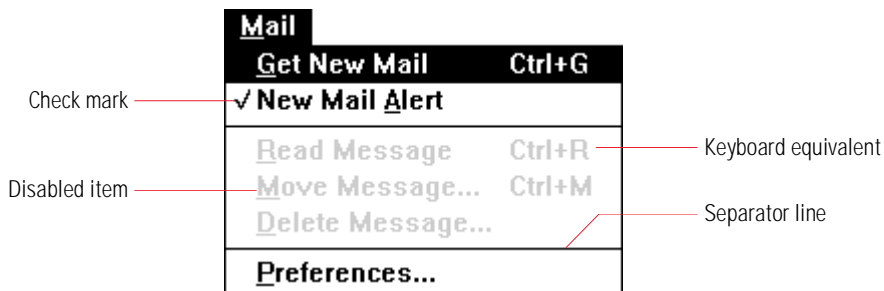
**Custom menus.** These are menus that you design yourself. You can add, modify, or delete components of the system or application default menus, or you can create entirely new menus.

---

You can control which of these types of menus appear in a menu bar that you create.

In the Oracle Power Objects object model, a menu consists of a *menu bar* object that displays one or more *menu* objects. These menu objects are not actually contained within the menu bar object—you can associate the same menu object with any number of menu bar objects. Each menu has a set of associated *menu items*. These items are not separate objects; they are simply parts of the menu object.

Besides the names of commands, menus also can display additional information:



**Separator lines** group related commands within the menu.

**Keyboard equivalents** (also called *accelerators*) are associated with commands in the menu. The user can execute menu commands simply by typing the keyboard equivalent sequence.

**Disabled** commands are displayed in grayed-out text and indicate commands that the user cannot select.

**Check marks** indicate commands that can be turned on or off.

You can control all of these features for menus that you create, both when you create the menu and after creation.

To create a menu, you follow the general steps described below. Each step is described in more detail in the sections of this chapter that follow.

**1** Create a menu bar object using the `NEW MenuBar` statement.

**2** Initialize the menu bar, if desired.

To initialize a menu bar with system default menus, call the `SysDefaultMenuBar()` method.

To initialize a menu bar with system default and application default menus, call the `DefaultMenuBar()` method.

**3** Create custom menus using the `NEW Menu` statement.

After creating the menu, set the menu's `Label` property to the title you want the menu to display.

**4** Add items to the custom menus.

To append an item to the end of the menu, call the `AppendMenuItem()` method.

To insert an item at a specified position, call the `InsertMenuItem()` method.

**5** Add the custom menus to the menu bar.

To append a menu to the end of menu bar, call the **AppendMenu()** method.

To insert a menu at a specified position, call the **InsertMenu()** method.

**6** Associate the menu bar with a form or report by calling the **SetMenuBar()** method.

You normally do this in the **InitializeWindow()** method of the form or report.

**7** Add method code to control the appearance and behavior of menu items.

Add code to the **TestCommand()** method to handle enabling and disabling of menu items.

Add code to the **DoCommand()** method to perform a command when an enabled menu item is selected.

## Creating a Menu Bar

You create a menu bar object using the Oracle Basic **NEW** operator. Typically, you store a reference to the newly created object in a variable of datatype *Object*.

For example, the following method code creates a menu bar object and stores a reference to it in the variable `mbrMenuBar1`:

```
DIM mbrMenuBar1 AS Object
mbrMenuBar1 = NEW MenuBar
```

You create menu bars in one of two locations, depending on when you want the object to be created:

To create a menu bar when your application starts up, you add method code to the **OnLoad()** method of your application. When you create a menu bar in **OnLoad()**, you should store a reference to the menu bar in a global variable so it can be accessed from other methods in your application.

To create a menu bar when a form or report is first displayed, you add method code to the **InitializeWindow()** method of the form or report. If you do not add method code to **InitializeWindow()**, this method installs the default menu bar (containing the system default and application default menus).

## Initializing a Menu Bar

You *initialize* a menu bar to add either the system default menus or the system default and application default menus. You cannot add the application default menus without also adding the system default menus.

---

## System Default Menus

To initialize a menu bar with the system default menus, you call the `SysDefaultMenuBar()` method of the menu bar object. This method deletes any existing menus before initializing the menu bar.

For example, the following method code initializes the menu bar “MenuBar1” with the system default menus:

```
MenuBar1.SysDefaultMenuBar()
```

The system default menus in Windows are:

- File
- Edit
- Window
- Help

The system default menus on the Macintosh are:

- The Apple menu
- File
- Edit

The commands in the system default menus for Windows and Macintosh are described in the online help.

## Application Default Menus

To initialize a menu bar with the system default and application default menus, you call the `DefaultMenuBar()` method of the form or report with which you want to associate the menu bar. You pass a reference to the menu bar object as the argument to `DefaultMenuBar()`. This method deletes any existing menus before initializing the menu bar.

For example, the following method code initializes the menu bar “MenuBar2” with the system default and application default menus for the form “Form1”:

```
Form1.DefaultMenuBar(MenuBar2)
```

The `DefaultMenuBar()` method returns a different set of menus depending on whether you call it for a form object or a report object.

For a **form**, `DefaultMenuBar()` returns the system default menu plus the Database menu.

For a **report**, `DefaultMenuBar()` returns the system default menu plus the Preview menu.

The commands in these menus are described in the online help.

## Creating Custom Menus

You create a menu object using the Oracle Basic `NEW` operator. Typically, you store a reference to the newly created object in a variable of datatype *Object*. For example, the following method code creates a menu object and stores a reference to it in the variable `CustomMenu1`:

```
DIM CustomMenu1 AS Object
CustomMenu1 = NEW Menu
```

After creating a menu object, you must set the **Label** property of the object. The **Label** property determines the menu name that appears in the menu bar. If you do not set the **Label** property of the menu, the user will be unable to see or choose items from the menu.

The following method code creates a menu object and sets its label to "Mail":

```
DIM mnuMail AS Object
mnuMail = NEW Menu
mnuMail.Label = "Mail"
```

You can choose a letter of the menu label to act as a menu shortcut in Windows. The letter you identify is marked with an underscore. The user can select the menu using the keyboard by pressing the Alt key, then the letter you specify.

To mark a letter of the label as a menu shortcut, precede the letter with an ampersand (&). For example, the following method code creates a menu object labeled "Mail" and specifies the letter "M" as the menu shortcut:

```
DIM mnuMail AS Object
mnuMail = NEW Menu
mnuMail.Label = "&Mail"
```

The preceding method code creates a menu that appears as shown in the following diagram (after the other steps of menu creation have been followed):



Menu shortcuts are not applicable for the Macintosh. If you specify a menu shortcut, the Macintosh does not display an underscore (nor does it display the ampersand)—your menu will appear correctly on both Macintosh and Windows.

### Adding Items to a Menu

After creating a menu object, you add *items* to the menu by calling the `AppendMenuItem()` or `InsertMenuItem()` method of the menu object.

`AppendMenuItem()` appends an item to the end of a menu.

`InsertMenuItem()` inserts a menu item at a specified location in a menu. You specify the position where the item is to be inserted as the first parameter to `InsertMenuItem()`.

---

When you call either `AppendMenuItem()` or `InsertMenuItem()`, you specify parameters that contain the following information:

- The **position** of the menu item (`InsertMenuItem()` only).
- The **label** of the menu item.
- The **command code** of the item, which is used elsewhere in your application to specify the application's response when the user selects the item.
- A **help context** for the menu item, which can be used to create context-sensitive online help for a Windows application.
- A **keyboard equivalent** for the menu item.

The full syntax of these methods is:

### Syntax of `AppendMenuItem()`

```
Sub AppendMenuItem(itemLabel as String, cmdCode as Integer,
    helpCtx as Integer, accel as String)
```

### Syntax of `InsertMenuItem()`

```
Sub InsertMenuItem(pos as Integer, itemLabel as String,
    cmdCode as Integer, helpCtx as Integer, accel as String)
```

For example, the following method code appends the command item "Get New Mail" to the "mnuMail" menu:

```
mnuMail.AppendMenuItem("Get New Mail", Cmd_GetNewMail, 0, &
    "^G")
```

The following method code inserts the command item "New Mail Alert" into the second position in the "mnuMail" menu:

```
mnuMail.InsertMenuItem(2, "New Mail Alert", &
    Cmd_NewMailAlert, 0, NULL)
```

### Label

The label is a string containing the text to appear in the menu. The label can contain only letters and numbers.

As with the **Label** property of a menu object, you can choose a letter of the menu item label to act as a menu shortcut for the item in Windows. To mark a letter of the label as a menu shortcut, precede the letter with an ampersand (&).



To add a separator line instead of a command item, you specify a hyphen character (-) as the label of the item and null values for the other parameters. For example, the following method code appends a separator line item to the end of the "Mail" menu:

```
mnuMail.AppendMenuItem("-", NULL, NULL, NULL)
```

### Command Code

The command code is an integer value identifying the menu command. You use this integer to refer to the menu command in the `TestCommand()` and `DoCommand()` methods, as described in the section "Handling Menu Selections" on page 14.14.

Command codes have two requirements:

- You must define a unique command code for each custom menu command in your application.
- Command codes you define cannot conflict with command codes that Oracle Power Objects uses for items in default menus.

➤ **Note:** Although two menu commands cannot share the same command code, a menu command might use the same command code as a toolbar button that performs the same action as the menu command.

To ensure that these requirements are met, you should define command codes in the following way:

- Define a symbolic constant for each menu command. You should declare these constants in the (Declarations) section of your application so they are globally available to all methods in your application.
- Define the value of each constant by adding a different value to the predefined integer constant `Cmd_FirstUserCommand`. Doing so ensures that your command codes will not conflict with the current any future version of Oracle Power Objects.

For example, the following method code defines a set of constants for items in the "Mail" menu:

```
CONST Cmd_GetNewMail = Cmd_FirstUserCommand + 1
CONST Cmd_NewMailAlert = Cmd_FirstUserCommand + 2
CONST Cmd_ReadMessage = Cmd_FirstUserCommand + 3
CONST Cmd_MoveMessage = Cmd_FirstUserCommand + 4
CONST Cmd_DeleteMessage = Cmd_FirstUserCommand + 5
CONST Cmd_Preferences = Cmd_FirstUserCommand + 6
```

The following method code uses the constant `Cmd_GetNewMail` to define the command code for the "Get New Mail" menu item:

```
mnuMail.AppendMenuItem("Get New Mail", Cmd_GetNewMail, 0, &
    "^G")
```

---

## Help Context

The help context is an integer that Oracle Power Objects passes to your help system when the user invokes online help (for example, by pressing the F1 key in Windows while the menu item is selected). This integer identifies the specific help topic that is to be displayed. For example, in Windows, the help context is passed to the WinHelp function.

If you specify a null help context (or a help context of zero), the user cannot invoke online help for the item.

## Keyboard Equivalent

The keyboard equivalent is a string identifying a keyboard equivalent for the command (the equivalent appears to the right of the menu item when the menu is displayed). The string you specify uses the following syntax:

```
[^] [+] {char | fkey}
```

Element	Description
<b>^</b>	Indicates that the System Command key is part of the keyboard equivalent. The System Command key is the Control key in Windows, the Command key on Macintosh.
<b>+</b>	Indicates that the Shift key is part of the keyboard equivalent (available only in Windows).
<i>char</i>	Specifies a standard alphabetic or numeric key equivalent in the range "A" to "Z" (either upper- or lowercase; the case of the character does not matter). An alphabetic key equivalent always includes the System Command key even if you do not specify it—for example, the keyboard equivalents "A" and "^A" are identical.
<i>fkey</i>	Specifies a function key in the range "F1" to "F12" (available only in Windows).

For example, the following method code specifies a keyboard equivalent of Control-G (Command-G on Macintosh) for a menu item "Get New Mail":

```
mnuMail.AppendMenuItem("Get New Mail", Cmd_GetNewMail, 0, &
    "^G")
```

When the preceding method code is executed, it creates a menu item with keyboard equivalents appropriate to the platform on which Oracle Power Objects is running.

### Examining, Modifying, and Deleting Menu Items

The following methods enable you to examine, modify, and delete items in an existing menu object:

Method	Description
<b>GetItemCount()</b>	Returns the number of items in the menu, including separator lines.
<b>GetMenuItem()</b>	Returns information about a specified menu item.
<b>SetMenuItem()</b>	Updates information about a specified menu item.
<b>DeleteMenuItem()</b>	Deletes a specified menu item.

The **GetItemCount()** method returns the number of items in a menu object, including separator lines. For example, the following method code displays a dialog box containing the number of items in the "mnuMail" menu:

```
MSGBOX "There are " & mnuMail.GetItemCount() & " menu items."
```

The **GetMenuItem()** and **SetMenuItem()** methods enable you to examine and change the menu item information specified in the **AppendMenuItem()** and **InsertMenuItem()** methods. These methods return or modify a specified piece of information about a specified menu item.

The syntax of these methods is as follows:

#### Syntax of GetMenuItem()

```
Function GetMenuItem(pos as Integer, what as Integer) as Variant
```

#### Syntax of SetMenuItem()

```
Sub SetMenuItem(pos as Integer, what as Integer, val as Variant)
```

The `pos` parameter specifies the position of the item to set or get.

The `what` parameter specifies the type of information to get or set. The following table lists the types of information you can get or set and the symbolic constant you use to specify each type of information:

Information	Datatype	Constant
Label	String	MenuPart_Label
Command Code	Integer	MenuPart_Command
Help Context	Integer	MenuPart_Help

---

<b>Information</b>	<b>Datatype</b>	<b>Constant</b>
Keyboard Equivalent	String	MenuPart_Accel

For **SetMenuItem()**, the `val` parameter specifies the new value for the item.

You can delete an existing item from a menu by calling the **DeleteMenuItem()** method of the menu object. You specify the position of the menu item to delete as the argument to **DeleteMenuItem()**.

For example, the following method code deletes the third item from the “Mail” menu:

```
mnuMail.DeleteMenuItem(3)
```

## Adding Menus to a Menu Bar

After creating a custom menu, you add the menus to a menu bar object by calling the **AppendMenu()** or **InsertMenu()** method of the menu bar object.

**AppendMenu()** appends a menu to the end of a menu bar. The menu is inserted before any system default menus that traditionally appear at the end of the menu bar (such as the Help menu in Windows).

**InsertMenu()** inserts a menu item at a specified location in a menu bar.

When you add a menu to a menu bar, the menu is associated with the menu bar but not contained by it. If you delete a menu bar object, the associated custom menus are not deleted along with it.

The following method code appends the menu “mnuMail” to the end of the menu bar “mbrMailForm”:

```
mbrMailForm.AppendMenu(mnuMail)
```

The following method code inserts the menu “mnuMail” at the third position in the menu bar “mbrMailForm”:

```
mbrMailForm.InsertMenu(3, mnuMail)
```

## Examining, Modifying, and Deleting Menus

The following methods enable you to examine, modify, and delete menus in an existing menu bar object:

<b>Method</b>	<b>Description</b>
<b>GetMenuBar()</b>	Returns a reference to the menu bar associated with the form or report.
<b>GetMenu()</b>	Returns a reference to a specified menu object in the menu bar.
<b>GetMenuCount()</b>	Returns a count of all menus in the menu bar.

Method	Description
<b>RemoveMenu()</b>	Removes a menu from a specified position in the menu bar. However, the menu object is not deleted. You can delete menu objects with the Oracle Basic DELETE statement or with the <b>DeleteAllMenus()</b> method.
<b>DeleteAllMenus()</b>	Removes all menus from the menu bar and deletes the menu objects.

The **GetMenuBar()** method returns a reference to the menu bar object associated with a form or report. For example, the following method code stores a reference to the menu bar associated with the form "Form1" in the variable `mbrForm1`:

```
DIM mbrForm1 AS Object
mbrForm1 = Form1.GetMenuBar()
```

The **GetMenu()** method returns a reference to a specified menu object. For example, the following method code stores a reference to the third menu of "mbrForm1" in the variable `mnul`:

```
DIM mnul AS Object
mnul = mbrForm1.GetMenu(3)
```

The **GetMenuCount()** method returns the number of menu objects associated with a specified menu bar. For example, the following method code displays a dialog box containing the number of menus in the "mbrMailForm" menu bar:

```
MSGBOX "There are " & mbrMailForm.GetMenuCount() & " menus."
```

You can remove existing menus from a menu bar by calling the **RemoveMenu()** or **DeleteAllMenus()** method of the menu bar object.

**RemoveMenu()** removes a menu from a specified location in the menu bar but does not delete the menu object. You specify the location of the menu as the argument to **RemoveMenu()**.

**DeleteAllMenus()** deletes all of the menu objects associated with a menu bar.

For example, the following method code removes the third menu from the menu bar "MenuBar1":

```
MenuBar1.RemoveMenu(3)
```

The following method code deletes all menus associated with the menu bar "MenuBar2":

```
MenuBar2.DeleteAllMenus()
```



**Caution:** Do not call **DeleteAllMenus()** if any of the menus in the menu bar are associated with other menu bar objects.

## Associating Menu Bars with Windows

After adding menus to a menu bar, you associate the menu bar with a window by calling the **SetMenuBar()** method of the form or report that is displayed in the window. You specify the menu bar object as the argument to **SetMenuBar()**.

---

To associate the menu bar with a window when the window is first displayed, you call **SetMenuBar()** from within the **InitializeWindow()** method of the form or report. If you do not create a custom menu bar within **InitializeWindow()**, the default menu bar object (containing the system default and application default menus) is associated with the window instead.

➤ **Note:** Entering method code usually overrides all of the default processing associated with that method. However, unless you call **SetMenuBar()** from within **InitializeWindow()**, Oracle Power Objects automatically installs the default menu bar object even if you do not include the **Inherited.InitializeWindow()** statement in your method code.

For example, the following method code creates a menu bar called “mbrSysDefault” from within the **InitializeWindow()** method:

```
Sub InitializeWindow()  
    DIM mbrSysDefault AS Object  
    mbrSysDefault = NEW MenuBar  
    mbrSysDefault.SysDefaultMenuBar()  
    Form1.SetMenuBar(mbrSysDefault)
```

You can also call the **SetMenuBar()** method after the window has already been displayed—for example, to switch the menu bar associated with the form or report.

The **InitializeWindow()** method is also where you specify custom toolbars and status lines for a window, as described later in this chapter.

## Handling Menu Selections

When creating a custom menu, you must add method code to specify how the application responds when the user selects a command. Your method code must handle two aspects of menu selections: setting the status of menu items, and executing code when an item is selected.

You set the status of menu items to determine how they are displayed when the user “pulls down” the menu to examine the items. Menu items can appear enabled or disabled, and can also appear checked or unchecked.

You execute code when the user actually selects the item (only enabled items can be selected).

### Setting the Status of Menu Items

You set the status of each menu item in a custom menu by adding method code to the **TestCommand()** method. You can add method code either to the form or report object with which the menu is associated or to the application object. Adding method code to the application object allows you to respond to commands in a centralized location—for example, your method code can respond the same way to commands in two different menu bars.

For an extended example of method code in **InitializeWindow()**, see the section “Example: Creating a Menu Bar” on page 14.18.

Oracle Power Objects calls **TestCommand()** whenever the user “pulls down” a menu to examine the items. **TestCommand()** is actually called once for each item in the menu, each time with the command code of a different menu item as the argument to the method. The return value of **TestCommand()** determines the status of the item.

You specify the return value of **TestCommand()** using a predefined symbolic constant. The following table summarizes the possible return values for **TestCommand()**:

Constant	Meaning	Example
TestCommand_Enabled	The command appears enabled.	<b>Preferences...</b>
TestCommand_Checked	The command appears enabled with a check mark.	✓ <b>New Mail <u>A</u>lert</b>
TestCommand_Disabled	The command appears disabled.	<b>Delete Message...</b>
TestCommand_Disabled_Checked	The command appears disabled with a check mark.	✓ <b>New Mail <u>A</u>lert</b>

If you do not explicitly set the return value of **TestCommand()**, the default return value is equal to zero. A return value of zero prompts Oracle Power Objects to check internally whether it handles the command. If it does not handle the command, it disables the command (equivalent to returning **TestCommand\_Disabled**).

Method code added to **TestCommand()** generally takes the form of a SELECT CASE statement. Each command code to be handled is a separate case in the statement. For example, the following method code determines the status of two menu commands identified by the constants **Cmd\_GetNewMail** and **Cmd\_NewMailAlert**. A global variable **gNewMailAlert** keeps track of the status of a menu item that can appear checked.

**(Declarations)**

```
CONST Cmd_GetNewMail = Cmd_FirstUserCommand + 1
CONST Cmd_NewMailAlert = Cmd_FirstUserCommand + 2
GLOBAL gNewMailAlert AS Integer
```

**Sub Initialize()**

```
gNewMailAlert = TRUE 'Initialize New Mail Alert status
```

---

```
Function TestCommand(cmdCode as Integer) as Long
```

```
SELECT CASE cmdCode
  CASE Cmd_GetNewMail
    TestCommand = TestCommand_Enabled
  CASE Cmd_NewMailAlert
    IF gNewMailAlert THEN
      TestCommand = TestCommand_Checked
    ELSE
      TestCommand = TestCommand_Enabled
    END IF
END SELECT
```

The **TestCommand()** method is also where you specify the status of toolbar buttons and status line panels, as described elsewhere in this chapter.

### Executing Code When a Menu Item is Selected

You specify the application's response to a menu selection by adding method code to the **DoCommand()** method. You can add method code either to the form or report object with which the menu is associated or to the application object. As with **TestCommand()**, adding method code to the application object allows you to respond to commands in a centralized location—for example, your method code can respond the same way to commands in two different menu bars.

Oracle Objects calls **DoCommand()** whenever the user selects an item from a menu (or when the user clicks on a toolbar button). The command code of the menu item is passed as the argument to **DoCommand()**. The return value of **DoCommand()** indicates whether the command was handled. By default, **DoCommand()** returns False (the command was not handled).

Oracle Power Objects calls **DoCommand()** at two levels:

- 1 The form or report with which the menu is associated.
- 2 The application containing the form or report.

If **DoCommand()** is not handled at either of these levels, it is then sent to Oracle Power Objects itself, which checks internally to see if it handles the event. **DoCommand()** is passed along this entire sequence unless **DoCommand()** returns True, at which point it is not passed along any further.

Method code added to **DoCommand()** generally takes the form of a SELECT CASE statement. Each command code to be handled is a separate case in the statement. For example, the following method code responds to two menu commands identified by the constants `Cmd_GetNewMail` and `Cmd_NewMailAlert`. A global variable `gNewMailAlert` keeps track of the status of a menu item that can appear checked.



**(Declarations)**

```
CONST Cmd_GetNewMail = Cmd_FirstUserCommand + 1
CONST Cmd_NewMailAlert = Cmd_FirstUserCommand + 2
GLOBAL gNewMailAlert AS Integer
```

**Sub Initialize()**

```
gNewMailAlert = TRUE 'Initialize New Mail Alert status
```

**Function DoCommand(cmdCode as Integer) as Long**

```
SELECT CASE cmdCode
  CASE Cmd_GetNewMail
    Self.GetNewMail()
    DoCommand = TRUE
  CASE Cmd_NewMailAlert
    gNewMailAlert = NOT(gNewMailAlert)
    DoCommand = TRUE
END SELECT
```

For a complete list of predefined command code constants, see Appendix C, "Constants and Reserved Words".

You can also add method code to **DoCommand()** to override or supplement the way Oracle Power Objects responds to items in the application or system default menus. Oracle Basic includes a set of predefined constants corresponding to the command codes for items in the default menus.

For example, the following method code customizes the way that Oracle Power Objects responds to the **File-Quit** menu command (**File-Exit** in Windows) by displaying a dialog box requesting confirmation. If the user clicks the **OK** button, **DoCommand()** returns False and the normal processing is executed. If the user clicks the **Cancel** button, **DoCommand()** returns True and the application does not quit.

**Function DoCommand(cmdCode as Integer) as Long**

```
CONST BTN_OK = 1

SELECT CASE cmdCode
  CASE Cmd_Quit
    IF MSGBOX("Are you sure you want to quit?", 33, &
      "Confirm Action") = BTN_OK THEN
      DoCommand = FALSE
    ELSE
      DoCommand = TRUE
    END IF
  END SELECT
```

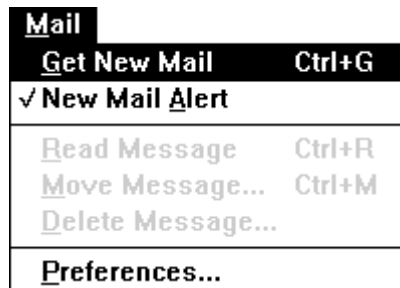
To execute an existing menu item from method code, you can call the **DoCommand()** method with the command code of the item to be called. For example, the following method code executes the **Database-Commit** menu command:

```
DoCommand(Cmd_Commit)
```

---

## Example: Creating a Menu Bar

This section provides a complete example of creating a custom menu in an application. The method code listed below adds a menu called “Mail” to the default menu bar for a form. The “Mail” menu appears as shown in the following diagram:



You can use this example in combination with the toolbar example described in the section “Example: Creating a Toolbar” on page 14.31, which creates a toolbar containing many of the same commands as the “Mail” menu.

The following method code appears in the (Declarations) section of the application:

```
(Declarations)
'Declare constants for menu commands
CONST Cmd_GetNewMail = Cmd_FirstUserCommand + 1
CONST Cmd_NewMailAlert = Cmd_FirstUserCommand + 2
CONST Cmd_ReadMessage = Cmd_FirstUserCommand + 3
CONST Cmd_MoveMessage = Cmd_FirstUserCommand + 4
CONST Cmd_DeleteMessage = Cmd_FirstUserCommand + 5
CONST Cmd_Preferences = Cmd_FirstUserCommand + 6
GLOBAL gNewMailAlert AS Integer
```

The following method code appears in the **Initialize()** method of the application:

```
Sub Initialize()
gNewMailAlert = TRUE 'Initialize New Mail Alert status
```

The following method code appears in the **InitializeWindow()** method of the form “frmMail”:

```
Sub InitializeWindow()
'Declare variables to hold menu bar and menu objects
DIM mbrMailForm AS Object
DIM mnuMail AS Object
```

```

'Create the menu bar object and initialize it with the system
'default and application default menus
mbrMailForm = NEW MenuBar
frmMail.DefaultMenuBar(mbrMailForm)

mnuMail = NEW Menu           'Create custom menu
mnuMail.Label = "&Mail"      'Add label to menu

'Add items to menu, including separator lines and keyboard
'equivalents
mnuMail.AppendMenuItem("&Get New Mail", Cmd_GetNewMail, 0, &
    "^G")
mnuMail.AppendMenuItem("New Mail &Alert", Cmd_NewMailAlert, &
    0, NULL)
mnuMail.AppendMenuItem("-", NULL, NULL, NULL)
mnuMail.AppendMenuItem("&Read Message", Cmd_ReadMessage, &
    0, "^R")
mnuMail.AppendMenuItem("&Move Message...", Cmd_MoveMessage, &
    0, "^M")
mnuMail.AppendMenuItem("&Delete Message...", &
    Cmd_DeleteMessage, 0, NULL)
mnuMail.AppendMenuItem("-", NULL, NULL, NULL)
mnuMail.AppendMenuItem("&Preferences...", Cmd_Preferences, &
    0, NULL)

mbrMailForm.AppendMenu(mnuMail)  'Append menu to menu bar

'Associate the menu bar object with the form
frmMail.SetMenuBar(mbrMailForm)

```

The following method code appears in the **TestCommand()** method of the form "frmMail":

```

Function TestCommand(cmdCode as Integer) as Long
SELECT CASE cmdCode
    CASE Cmd_GetNewMail
        TestCommand = TestCommand_Enabled
    CASE Cmd_NewMailAlert
        IF gNewMailAlert THEN
            TestCommand = TestCommand_Checked
        ELSE
            TestCommand = TestCommand_Enabled
        END IF
    CASE Cmd_ReadMessage, Cmd_MoveMessage, Cmd_DeleteMessage
        IF ISNULL(lstMessageList.value) THEN
            TestCommand = TestCommand_Disabled
        ELSE

```

```

        TestCommand = TestCommand_Enabled
    END IF
    CASE Cmd_Preferences
        TestCommand = TestCommand_Enabled
    END SELECT

```

The following method code appears in the **DoCommand()** method of the form “frmMail”:

```
Function DoCommand(cmdCode as Integer) as Long
```

```

SELECT CASE cmdCode
    CASE Cmd_GetNewMail
        Self.GetNewMail()
        DoCommand = TRUE
    CASE Cmd_NewMailAlert
        gNewMailAlert = NOT(gNewMailAlert)
        DoCommand = TRUE
    CASE Cmd_ReadMessage
        Self.ReadMessage(lstMessageList.value)
        DoCommand = TRUE
    CASE Cmd_MoveMessage
        Self.MoveMessage(lstMessageList.value)
        DoCommand = TRUE
    CASE Cmd_DeleteMessage
        Self.DeleteMessage(lstMessageList.value)
        DoCommand = TRUE
    CASE Cmd_Preferences
        frmPrefs.OpenModal(false)
        DoCommand = TRUE
END SELECT

```

## Toolbars

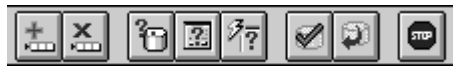
Toolbars are displayed in an area directly beneath the menu bar (in this release of Oracle Power Objects, you cannot customize the location of a toolbar). Each window (form or report) in your application can display its own toolbar; when the window is active, the associated toolbar appears beneath the menu bar for that window. Only one toolbar can be associated with a given window.



By default, forms and reports in run-time mode display a toolbar that provides shortcuts to common commands. This section describes how to customize the default toolbars, either by adding custom buttons to those that appear by default or by replacing them altogether.

An Oracle Power Objects toolbar can contain two types of toolbars:

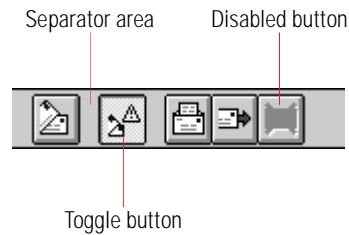
**Application default toolbars.** These are standard toolbars for the type of Oracle Power Objects window being displayed. The following diagram shows the application default buttons for forms:



**Custom toolbars.** These are toolbars that you design yourself. If you wish, you can also create custom versions of the application default toolbars instead of using those provided by Oracle Power Objects.

Each toolbar object has a set of associated *toolbar buttons*. These buttons are not separate objects; they are simply parts of the toolbar object.

Toolbars can have the following features:



**Separator areas** group related buttons within the toolbar.

**Disabled buttons** are displayed in a grayed-out bitmap and indicate buttons that the user cannot click.

**Toggle buttons** indicate commands that can be turned on or off.

You can control all of these features for toolbars that you create, both when you create the toolbar and after creation.

## Overview of Creating Toolbars

To create a toolbar, you follow the general steps described below. Each step is described in more detail in the sections of this chapter that follow.

- 1 Create a toolbar object using the `NEW Toolbar` statement.
- 2 Initialize the toolbar, if desired.  
To initialize a toolbar with application default buttons, call the `DefaultToolbar()` method.
- 3 Add buttons to the toolbar.  
To append a button to the end of toolbar, call the `TBAppendButton()` method.  
To insert a button at a specified position, call the `TBInsertButton()` method.

- 
- 4 Associate the toolbar with a form or report by calling the `SetToolbar()` method.

You normally do this in the `InitializeWindow()` method of the form or report.

- 5 Add method code to control the appearance and behavior of buttons.

Add code to the `TestCommand()` method to handle enabling and disabling of buttons.

Add code to the `DoCommand()` method to perform a command when an enabled button is clicked.

## Creating a Toolbar

You create a toolbar object using the Oracle Basic `NEW` operator. Typically, you store a reference to the newly created object in a variable of datatype *Object*.

For example, the following method code creates a toolbar object and stores a reference to it in the variable `Toolbar1`:

```
DIM Toolbar1 AS Object
Toolbar1 = NEW Toolbar
```

You create toolbars in one of two locations, depending on when you want the toolbar to be created:

To create a toolbar when your application starts up, you add method code to the `OnLoad()` method of your application. When you create a toolbar in `OnLoad()`, you should store a reference to the toolbar in a global variable so it can be accessed from other methods in your application.

To create a toolbar when a form or report is first displayed, you add method code to the `InitializeWindow()` method of the form or report. If you do not add method code to `InitializeWindow()`, this method installs the default toolbar (containing the application default buttons).

## Initializing a Toolbar

You initialize a toolbar to add the application default buttons. To initialize the toolbar, you call the `DefaultToolbar()` method of the form or report with which you will associate the toolbar. You pass a reference to the toolbar object as the argument to `DefaultToolbar()`. This method deletes any existing buttons before initializing the toolbar.

For example, the following method code initializes the toolbar “Toolbar2” with the application default buttons for the form “Form1”:

```
Form1.DefaultToolbar(Toolbar2)
```

The `DefaultToolbar()` method returns a different set of buttons depending on whether you call it for a form object or a report object.

For a form, `DefaultToolBar()` returns the following buttons:



For a report, `DefaultMenuBar()` returns the following buttons:



These toolbars are described in the online help.

## Adding Buttons to a Toolbar

After creating a toolbar object, you add buttons to it by calling the `TBAppendButton()` or `TBInsertButton()` method of the toolbar object.

`TBAppendButton()` appends a button to the end of a toolbar.

`TBInsertButton()` inserts a button at a specified location in a toolbar. You specify the position where the item is to be inserted as the first parameter to `TBInsertButton()`.

For example, the following method code appends a button to the “tbrMailForm” toolbar:

```
tbrMailForm.TBAppendButton(Cmd_GetNewMail, GetNewMl, &
    ToolbarStyle_PushBtn, 0)
```

The following method code inserts a button into the third position in the “tbrMailForm” toolbar:

```
tbrMailForm.TBInsertButton(3, Cmd_NewMailAlert, NwMlAlrt, &
    ToolbarStyle_Toggle, 0)
```

When you call either `TBAppendButton()` or `TBInsertButton()`, you specify parameters that contain the following information:

- The **position** of the button on the toolbar (`TBInsertButton()` only).
- The **command code** of the button, which is used elsewhere in your application to specify the application's response when the user clicks the button.
- A **bitmap** for the button, which contains the image displayed on the button's face.
- The **style** of the button, which indicates whether the button is a *pushbutton*, a *toggle button*, or a *separator area*.
- A **help context** for the button, which can be used to create context-sensitive online help for a Windows application.

The full syntax of these methods is:

### Syntax of `TBAppendButton()`

```
Sub TBAppendButton(cmdCode as Integer, bitmap as Object,
    style as Integer, helpContext as Integer)
```

---

## Syntax of TBIInsertButton()

```
Sub TBIInsertButton(pos as Integer, cmdCode as Integer,  
    bitmap as Object, style as Integer, helpContext as Integer)
```

### Command Code

The command code is an integer identifying a toolbar button (separator areas always have the command code zero). You use this integer to refer to the button in the **TestCommand()** and **DoCommand()** methods, as described in the section “Handling Toolbar Clicks” on page 14.28.

Command codes have two requirements:

- You must define a unique command code for each toolbar button in your application.
- Command codes you define cannot conflict with command codes that Oracle Power Objects uses for items in default toolbars.

➤ **Note:** Although two toolbar buttons cannot share the same command code, a toolbar button might use the same command code as a menu command that performs the same action as the toolbar button.

To ensure that these requirements are met, you should define command codes in the following way:

- Define a symbolic constant for each menu command. You should declare these constants in the (Declarations) section of your application so they are globally available to all methods in your application.
- Define the value of each constant by adding a different value to the predefined integer constant `Cmd_FirstUserCommand`. Doing so ensures that your command codes will not conflict with the current any future version of Oracle Power Objects.

For example, the following method code defines a set of constants for items used in a toolbar:

```
CONST Cmd_GetNewMail = Cmd_FirstUserCommand + 1  
CONST Cmd_NewMailAlert = Cmd_FirstUserCommand + 2  
CONST Cmd_ReadMessage = Cmd_FirstUserCommand + 3  
CONST Cmd_MoveMessage = Cmd_FirstUserCommand + 4  
CONST Cmd_DeleteMessage = Cmd_FirstUserCommand + 5  
CONST Cmd_Preferences = Cmd_FirstUserCommand + 6
```

The following method code uses the constant `Cmd_GetNewMail` to define the command code for a toolbar button that gets new mail:

```
tbrMailForm.TBAppendButton(Cmd_GetNewMail, GetNewMl, &  
    ToolbarStyle_PushBtn, 0)
```



### Bitmap

The bitmap argument is a reference to a bitmap object in your application. The display area in a toolbar button is 16 pixels by 16 pixels. A larger bitmap object will be cropped to fit the display area.

### Button Style

The button style determines the appearance and behavior of the button. The button style is usually specified using a symbolic constant.

Constant	Meaning
ToolbarStyle_PushBtn	The button is a standard pushbutton.
ToolbarStyle_Toggle	The button is an on/off toggle button.
ToolbarStyle_Separator	The button is a separator providing a 10-pixel-wide space between the adjoining buttons.

### Help Context

The help context is an integer that Oracle Power Objects passes to your help system when the user invokes online help. This integer identifies the specific help topic that is to be displayed. For example, in Windows, the help context is passed to the WinHelp function.

If you specify a null help context (or a help context of zero), the user cannot invoke online help for the button.

### Examining, Modifying, and Deleting Buttons

The following methods enable you to examine, modify, and delete buttons in an existing toolbar object:

Method	Description
GetToolBar()	Returns a reference to the toolbar associated with a form or report.
TBGetCount()	Returns a count of all buttons in the toolbar, including separator areas.
TBGetButton()	Returns a specified piece of information about a specified button in the toolbar. You can get the button's command code, the bitmap, the button style, or the help context.
TBSetButton()	Modifies a specified piece of information about a specified button in the toolbar. You can set the button's command code, the bitmap, the button style, or the help context.

---

<b>Method</b>	<b>Description</b>
<b>TBDeleteButton()</b>	Deletes a button from a specified position in the toolbar.
<b>ClearToolbar()</b>	Deletes all buttons from the toolbar.

The **GetToolbar()** method returns a reference to the toolbar object associated with a form or report. For example, the following method code stores a reference to the toolbar associated with the form "Form1" in the variable `tbrForm1`:

```
DIM tbrForm1 AS Object
tbrForm1 = Form1.GetToolbar()
```

The **TBGetCount()** method returns the number of buttons in a toolbar object, including separator areas. For example, the following method code displays a dialog box containing the number of buttons in the "tbrMailForm" toolbar:

```
MSGBOX "There are " & tbrMailForm.TBGetCount() & " buttons."
```

The **TBGetButton()** and **TBSetButton()** methods enable you to examine and change the button information specified in the **TBAppendButton()** and **TBInsertButton()** methods. These methods return or modify a specified piece of information about a specified toolbar button.

The syntax of these methods is as follows:

#### **Syntax of TBGetButton()**

```
Function TBGetButton(index as Integer, part as Integer) &
as Variant
```

#### **Syntax of TBSetButton()**

```
Sub TBSetButton(index as Integer, part as Integer, &
val as Variant)
```

The `index` parameter specifies the position of the button to set or get.

The `part` parameter specifies the type of information to get or set. The following table lists the types of information you can get or set and the symbolic constant you use to specify each type of information:

<b>Information</b>	<b>Datatype</b>	<b>Constant</b>	<b>Notes</b>
Command Code	Integer	ToolbarPart_Command	
Bitmap	Object	ToolbarPart_Bitmap	You cannot set the bitmap of a separator area.
Style	Integer	ToolbarPart_Style	You cannot change the style of an existing button.

Information	Datatype	Constant	Notes
Help Context	Integer	ToolbarPart_Help	

For **TBSetButton()**, the `val` parameter specifies the new value for the item.

You can delete an existing button from a toolbar by calling the **TBDeleteButton()** method of the toolbar object. You specify the location of the button to delete as the argument to **TBDeleteButton()**. For example, the following method code deletes the third button from the "tbrMailForm" toolbar:

```
tbrMailForm.TBDeleteButton(3)
```

You can delete all buttons from a toolbar by calling the **ClearToolbar()** method of the toolbar object. For example, the following method code deletes all buttons from the "tbrMailForm" toolbar:

```
tbrMailForm.ClearToolbar()
```

### Associating Toolbars with Windows

After creating a toolbar, you associate it with a window by calling the **SetToolbar()** method of the form or report that is displayed in the window. You specify the toolbar object as the argument to **SetToolbar()**.

To associate the toolbar with a window when the window is first displayed, you call **SetToolbar()** from within the **InitializeWindow()** method of the form or report. If you do not create a custom toolbar within **InitializeWindow()**, the default toolbar object is associated with the window instead.

➤ **Note:** Entering method code usually overrides all of the default processing associated with that method. However, unless you call **SetToolbar()** from within **InitializeWindow()**, Oracle Power Objects automatically installs the default toolbar object even if you do not include the **Inherited.InitializeWindow()** statement in your method code.

For example, the following method code creates a toolbar called "tbrAppDefault" from within the **InitializeWindow()** method:

```
Sub InitializeWindow()
    DIM tbrAppDefault AS Object
    tbrAppDefault = NEW Toolbar
    Form1.DefaultToolbar(tbrAppDefault)
    frmMail.SetToolbar(tbrAppDefault)
End Sub
```

You can also call the **SetToolbar()** method after the window has already been displayed; for example, to switch the toolbar associated with a form or report.

For an extended example of method code in **InitializeWindow()**, see the section "Example: Creating a Toolbar" on page 14.31.

---

If you do not want any toolbar to appear in the window, you must pass the value `NULL` as the argument to `SetToolBar()`, as in the following example:

```
Form1.SetToolBar(NULL)
```

The `InitializeWindow()` method is also where you specify custom menu bars and status lines for a window, as described elsewhere in this chapter.

## Handling Toolbar Clicks

When creating a custom toolbar, you must add method code to specify how the application responds when the user clicks on a button. As with menus, your method code must handle two aspects of toolbar clicks: setting the status of toolbar buttons, and executing code when a button is clicked.

You set the status of toolbar buttons to determine how they are displayed on screen. Toolbar buttons can appear enabled or disabled, and can also appear toggled or untoggled.



You execute code when the user actually clicks the button (only enabled buttons can be clicked).



### Setting the Status of Toolbar Buttons

You set the status of each button in a custom toolbar by adding method code to the `TestCommand()` method. You can add method code either to the form or report object with which the toolbar is associated or to the application object. Adding method code to the application object allows you to respond to commands in a centralized location—for example, your method code can respond the same way to commands that appear both on a toolbar and in a menu bar.

Oracle Power Objects calls `TestCommand()` at regular intervals when it is not performing any other actions (it is usually called several times each second). `TestCommand()` is also called when the user clicks a toolbar button. When `TestCommand()` is called during system idle time, it is actually called once for each button on the toolbar, each time with the command code of a different toolbar button as the argument to the method. When `TestCommand()` is called by a user click, it is called only for the toolbar button on which the user clicks. The return value of `TestCommand()` determines the status of the button at that time.

You specify the return value of `TestCommand()` using a predefined symbolic constant. The following table summarizes the possible return values for `TestCommand()`:

Constant	Meaning	Example
<code>TestCommand_Enabled</code>	The button appears enabled.	
<code>TestCommand_Checked</code>	The button appears enabled and toggled.	

Constant	Meaning	Example
TestCommand_Disabled	The button appears disabled.	
TestCommand_Disabled_Checked	The button appears disabled and toggled.	

If you do not explicitly set the return value of **TestCommand()**, the default return value is equal to zero. A return value of zero prompts Oracle Power Objects to check internally whether it handles the command—if it does not handle the command, it disables the command (equivalent to returning `TestCommand_Disabled`).

➤ **Note:** You can return a value of `TestCommand_Checked` for both pushbuttons and toggle buttons. In both cases, the button appears appropriately toggled.

Method code added to **TestCommand()** generally takes the form of a SELECT CASE statement. Each command code to be handled is a separate case in the statement. For example, the following method code determines the status of two toolbar buttons identified by the constants `Cmd_GetNewMail` and `Cmd_NewMailAlert`. A global variable `gNewMailAlert` keeps track of the status of a toggle button.

**(Declarations)**

```
CONST Cmd_GetNewMail = Cmd_FirstUserCommand + 1
CONST Cmd_NewMailAlert = Cmd_FirstUserCommand + 2
GLOBAL gNewMailAlert AS Integer
```

**Sub Initialize()**

```
gNewMailAlert = TRUE 'Initialize New Mail Alert status
```

**Function TestCommand(cmdCode as Integer) as Long**

```
SELECT CASE cmdCode
CASE Cmd_GetNewMail
TestCommand = TestCommand_Enabled
CASE Cmd_NewMailAlert
IF gNewMailAlert THEN
TestCommand = TestCommand_Checked
ELSE
TestCommand = TestCommand_Enabled
END IF
END SELECT
```

The **TestCommand()** method is also where you specify the status of menu items and status line panels, as described elsewhere in this chapter.

---

## Executing Code When a Button is Clicked

You specify the application's response to a button click by adding method code to the **DoCommand()** method. You can add method code either to the form or report object with which the toolbar is associated or to the application object. As with **TestCommand()**, adding method code to the application object allows you to respond to commands in a centralized location—for example, your method code can respond the same way to commands that appear both on a toolbar and in a menu bar.

Oracle Power Objects calls **DoCommand()** whenever the user clicks on a toolbar button (or when the user selects an item from a menu). The command code of the button is passed as the argument to **DoCommand()**. The return value of **DoCommand()** indicates whether the command was handled. By default, **DoCommand()** returns **False** (the command was not handled).

Oracle Power Objects calls **DoCommand()** at two levels:

- 1 The form or report with which the toolbar is associated
- 2 The application containing the form or report.

If **DoCommand()** is not handled at either of these levels, it is then sent to Oracle Power Objects itself, which checks internally to see if it handles the event. **DoCommand()** is passed along this entire sequence unless **DoCommand()** returns **True**, at which time it is not passed any further.

Method code added to **DoCommand()** generally takes the form of a **SELECT CASE** statement. Each command code to be handled is a separate case in the statement. For example, the following method code executes code in response to two toolbar buttons identified by the constants **Cmd\_GetNewMail** and **Cmd\_NewMailAlert**. A global variable **gNewMailAlert** keeps track of the status of a toggle button.

```
(Declarations)
```

```
CONST Cmd_GetNewMail = Cmd_FirstUserCommand + 1
CONST Cmd_NewMailAlert = Cmd_FirstUserCommand + 2
GLOBAL gNewMailAlert AS Integer
```

```
Sub Initialize()
```

```
gNewMailAlert = TRUE 'Initialize New Mail Alert status
```

```
Function DoCommand(cmdCode as Integer) as Long
```

```
SELECT CASE cmdCode
CASE Cmd_GetNewMail
    Self.GetNewMail()
    DoCommand = TRUE
CASE Cmd_NewMailAlert
    gNewMailAlert = NOT(gNewMailAlert)
    DoCommand = TRUE
END SELECT
```

For a complete list of predefined command code constants, see Appendix C, "Constants and Reserved Words".

You can also add method code to **DoCommand()** to override or supplement the way Oracle Power Objects responds to items in default toolbars. Oracle Basic includes a set of predefined constants corresponding to the command codes for items in the default toolbars

For example, the following method code customizes the way that Oracle Power Objects responds to the Rollback button by displaying a dialog box requesting confirmation. If the user clicks the **OK** button, **DoCommand()** returns False and the normal processing is executed. If the user clicks the **Cancel** button, **DoCommand()** returns True and the application does not roll back the current transaction.

```
Function DoCommand(cmdCode as Integer) as Long
CONST BTN_OK = 1

SELECT CASE cmdCode
CASE Cmd_Rollback
IF MSGBOX("Are you sure you want to roll back?", 33, &
"Confirm Action") = BTN_OK THEN
DoCommand = FALSE
ELSE
DoCommand = TRUE
END IF
END SELECT
```

To execute the code associated with an existing button, you can call the **DoCommand()** method with the command code of the button. For example, the following method code executes the code associated with the Commit button:

```
DoCommand(Cmd_Commit)
```

### Example: Creating a Toolbar

This section provides a complete example of creating a custom toolbar in an application. The method code listed below adds a custom toolbar to a form. The toolbar appears as shown in the following diagram:



This example can be used in combination with the menu example described in the section "Example: Creating a Menu Bar" on page 14.18, which creates a menu containing many of the same commands.

---

The following method code appears in the (Declarations) section of the application:

```
(Declarations)
'Declare constants for menu commands and toolbar buttons
CONST Cmd_GetNewMail = Cmd_FirstUserCommand + 1
CONST Cmd_NewMailAlert = Cmd_FirstUserCommand + 2
CONST Cmd_ReadMessage = Cmd_FirstUserCommand + 3
CONST Cmd_MoveMessage = Cmd_FirstUserCommand + 4
CONST Cmd_DeleteMessage = Cmd_FirstUserCommand + 5
CONST Cmd_Preferences = Cmd_FirstUserCommand + 6
GLOBAL gNewMailAlert AS Integer
```

The following method code appears in the **Initialize()** method of the application:

```
Sub Initialize()
gNewMailAlert = TRUE 'Initialize New Mail Alert status
```

The following method code appears in the **InitializeWindow()** method of the form "frmMail":

```
Sub InitializeWindow()
DIM tbrMailForm AS Object

tbrMailForm = NEW Toolbar

tbrMailForm.TBAppendButton(Cmd_GetNewMail, GetNewMl, &
    ToolbarStyle_PushBtn, 0)
tbrMailForm.TBAppendButton(0, NULL, ToolbarStyle_Separator, &
    0)
tbrMailForm.TBAppendButton(Cmd_NewMailAlert, NwMlAlrt, &
    ToolbarStyle_Toggle, 0)
tbrMailForm.TBAppendButton(0, NULL, ToolbarStyle_Separator, &
    0)
tbrMailForm.TBAppendButton(Cmd_ReadMessage, ReadMsg, &
    ToolbarStyle_PushBtn, 0)
tbrMailForm.TBAppendButton(Cmd_MoveMessage, MoveMsg, &
    ToolbarStyle_PushBtn, 0)
tbrMailForm.TBAppendButton(Cmd_DeleteMessage, DelMsg, &
    ToolbarStyle_PushBtn, 0)

'Associate the toolbar object with the form
frmMail.SetToolbar(tbrMailForm)
```



The following method code appears in the **TestCommand()** method of the form "frmMail":

```
Function TestCommand(cmdCode as Integer) as Long
SELECT CASE cmdCode
CASE Cmd_GetNewMail
    TestCommand = TestCommand_Enabled
CASE Cmd_NewMailAlert
    IF gNewMailAlert THEN
        TestCommand = TestCommand_Checked
    ELSE
        TestCommand = TestCommand_Enabled
    END IF
CASE Cmd_ReadMessage, Cmd_MoveMessage, Cmd_DeleteMessage
    IF ISNULL(lstMessageList.value) THEN
        TestCommand = TestCommand_Disabled
    ELSE
        TestCommand = TestCommand_Enabled
    END IF
CASE Cmd_Preferences
    TestCommand = TestCommand_Enabled
END SELECT
```

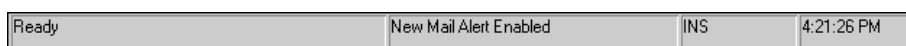
The following method code appears in the **DoCommand()** method of the form "frmMail":

```
Function DoCommand(cmdCode as Integer) as Long
SELECT CASE cmdCode
CASE Cmd_GetNewMail
    Self.GetNewMail()
    DoCommand = TRUE
CASE Cmd_NewMailAlert
    gNewMailAlert = NOT(gNewMailAlert)
    DoCommand = TRUE
CASE Cmd_ReadMessage
    Self.ReadMessage(lstMessageList.value)
    DoCommand = TRUE
CASE Cmd_MoveMessage
    Self.MoveMessage(lstMessageList.value)
    DoCommand = TRUE
CASE Cmd_DeleteMessage
    Self.DeleteMessage(lstMessageList.value)
    DoCommand = TRUE
CASE Cmd_Preferences
    frmPrefs.OpenModal(false)
    DoCommand = TRUE
END SELECT
```

---

## Status Lines

Status lines are displayed at the bottom of the application window (in Windows) or directly under the toolbar area (on Macintosh). In Windows, the location of the status line is fixed; on the Macintosh, the user can drag the status line to a different location. Each window (form or report) in your application can display its own status line—when the window is active, the associated status line appears in the appropriate location.



Status lines display information in *panels*—rectangular areas within the status line. An Oracle Power Objects status line can display three types of panels:

The **Summary panel** displays Summary help information when the user moves the cursor over an object for which summary help has been defined. This panel is installed automatically when the status line is created and cannot be deleted. You can, however, disable the display of Summary help information.

**System default panels** are standard panels for an application in the operating system where your application is running. System default panels on Windows are Caps Lock, Num Lock, and Scroll Lock panels. There are no system default panels on Macintosh.

**Custom panels** are panels that you design yourself. Custom panels can display many different types of information—for example, they can display page numbers, show the current system time, or indicate editing modes in your application.

By default, forms and reports do not display a status line in run-time mode. To display a status line, you must create the status line object yourself.

### Overview of Creating Status Lines

To create a status line, you follow the general steps described below. Each step is described in more detail in the sections of this chapter that follow.

- 1 Create a status line object using the `NEW StatusLine` statement.
- 2 Initialize the status line, if desired.  
To initialize a status line with system default panels, call the `SysDefaultStatusLine()` method.
- 3 Add custom panels to the status line, if desired.  
To insert a panel at a specified position, call the `InsertStatusPanel()` method.
- 4 Associate the status line with a form or report by calling the `SetStatusLine()` method.  
You normally do this in the `InitializeWindow()` method of the form or report.

**5** Add method code to control the appearance of status panels.

Call the `SetStatusPanelMsg()` method to set the text of a given status panel.

Call the `SetStatDispList()` method to enable automatic updating of a panel.

Add code to the `TestCommand()` method to handle automatic display of text in the panel.

## Creating a Status Line

You create a status line object using the Oracle Basic `NEW` operator. Typically, you store a reference to the newly created object in a variable of datatype *Object*.

For example, the following method code creates a status line object and stores a reference to it in the variable `StatusLine1`:

```
DIM StatusLine1 AS Object
StatusLine1 = NEW StatusLine
```

You create status lines in one of two locations, depending on when you want the status line to be created:

To create a status line when your application starts up, you add method code to the `OnLoad()` method of your application. When you create a status line in `OnLoad()`, you should store a reference to the status line in a global variable so it can be accessed from other methods in your application.

To create a status line when a form or report is first displayed, you add method code to the `InitializeWindow()` method of the form or report. If you do not add method code to `InitializeWindow()`, no status line is associated with the form or report.

When created, a status line automatically contains one panel (the Summary panel). This panel cannot be deleted from the status line.

## Initializing a Status Line

You initialize a status line to add the system default panels. To initialize the status line, you call the `SysDefaultStatusLine()` method of the status line object. This method deletes any existing panels before initializing the status line, except the Summary panel which cannot be deleted.

For example, the following method code initializes the status line "StatusLine1" with the system default panels:

```
StatusLine1.SysDefaultStatusLine()
```

The system default panels on Windows are:

- Caps Lock
- Num Lock
- Scroll Lock

---

There are no system default panels on the Macintosh.

## Adding Panels to a Status Line

After creating a status line object, you add panels to it by calling the **InsertStatusPanel()** method of the status line object. **InsertStatusPanel()** inserts a panel in a specified position on a status line.

You specify the position where the panel is to be inserted as the first parameter to **InsertStatusPanel()**. Since the first panel of a status line object is always the Summary panel, you must insert panels beginning at position 2.

For example, the following method code inserts a panel into the second position in the “slnMailForm” status line:

```
slnMailForm.InsertStatusPanel(2, 400, 100)
```

When you call **InsertStatusPanel()**, you specify parameters that contain the following information:

- The **position** to insert the panel, which must be 2 or higher.
- The **width** of the panel specified in pixels.
- The **maximum message length** for the panel. Text displayed in the panel exceeding the maximum length is truncated to the maximum.

The full syntax of **InsertStatusPanel()** is:

### Syntax of InsertStatusPanel()

```
Sub InsertStatusPanel(pos as Integer, wid as Integer,  
maxMsgLen as Integer)
```

Panels are inserted into the status line at the right edge of the status line. The Summary panel occupies all of the area not covered by added panels.

If you want to designate the panel to be updated automatically, you must set additional information using the **SetStatDispList()** method, as described in the section “Updating Status Panels” on page 14.39.

## Modifying and Deleting Panels

The following methods enable you to get information about and modify the panels in an existing status line object:

<b>Method</b>	<b>Description</b>
<b>GetStatusLine()</b>	Returns a reference to the status line associated with the form or report.
<b>GetStatCount()</b>	Returns a count of all panels in the status line.

Method	Description
<b>GetStatPanel()</b>	Returns a specified piece of information about a specified panel in the status line. You can get the text currently displayed in the panel, the panel's width, the command code, and the message strings associated with the panel's status.
<b>DeleteStatusPanel()</b>	Deletes a panel from a specified position in the status line. The Summary panel cannot be deleted.
<b>ClearStatusLine()</b>	Deletes all panels from the status line except the Summary panel, which cannot be deleted.

The **GetStatusLine()** method returns a reference to the status line object associated with a form or report. For example, the following method code stores a reference to the status line associated with the form "Form1" in the variable `slnForm1`:

```
DIM slnForm1 AS Object
slnForm1 = Form1.GetStatusLine()
```

The **GetStatCount()** method returns the number of panels in a status line object. For example, the following method code displays a dialog box containing the number of panels in the "slnMailForm" status line:

```
MSGBOX "There are " & slnMailForm.GetStatCount() & " panels."
```

The **GetStatPanel()** method allows you to examine and change the panel information specified in the **InsertStatusPanel()** and **SetStatDispList()** methods. This method returns a specified piece of information about a specified button. The syntax of **GetStatPanel()** is:

**Syntax of GetStatPanel()**

```
Function GetStatPanel(pos as Integer, what as Integer) &
as Variant
```

The `pos` parameter specifies the position of the panel.

The `what` parameter specifies the type of information to get. The following table lists the types of information you can get and the symbolic constant you use to specify each type of information:

Information	Datatype	Constant
Panel Text	String	StatusLinePart_Text
Width	Integer	StatusLinePart_Width
Command Code	Integer	StatusLinePart_Command
"Enabled" Text	String	StatusLinePart_Msg_Enabled
"Checked" Text	String	StatusLinePart_Msg_Checked

Information	Datatype	Constant
“Disabled” Text	String	StatusLinePart_Msg_Disabled
“Disabled and Checked” Text	String	StatusLinePart_Msg_Disabled_Checked

The information specified in the last four parameters is described in the section “Updating Status Panels Automatically” on page 14.39.

You can delete an existing panel from a status line by calling the **DeleteStatusPanel()** method of the status line object. You specify the position of the panel to delete as the argument to **DeleteStatusPanel()**. You can delete any panel except the Summary panel (at position 1). For example, the following method code deletes the third panel from the “slnMailForm” toolbar:

```
slnMailForm.DeleteStatusPanel(3)
```

You can delete all panels (except the Summary panel) from a status line by calling the **ClearStatusLine()** method of the toolbar object. For example, the following method code deletes all panels from the “slnMailForm” toolbar:

```
slnMailForm.ClearStatusLine()
```

## Associating Status Lines with Windows

After creating a status line, you associate it with a window by calling the **SetStatusLine()** method of the form or report that is displayed in the window. You specify the status line object as the argument to **SetStatusLine()**.

To associate the status line with a window when the window is first displayed, you call **SetStatusLine()** from within the **InitializeWindow()** method of the form or report. If you do not create a custom status line within **InitializeWindow()**, no status line is associated with the window.

For example, the following method code creates a status line called “slnSysDefault” from within the **InitializeWindow()** method:

```
Sub InitializeWindow()
    DIM slnSysDefault AS Object
    slnSysDefault = NEW StatusLine
    slnSysDefault.SysDefaultStatusLine()
    Form1.SetStatusLine(slnSysDefault)
```

You can also call the **SetStatusLine()** method after the window has already been displayed—for example, to switch the status line associated with a form or report.

The **InitializeWindow()** method is also where you specify custom menu bars and toolbars, as described earlier in this chapter.

For an extended example of method code in **InitializeWindow()**, see the section “Example: Creating a Status Line” on page 14.42.

## Updating Status Panels

When creating a custom status line, you must add method code to update status panels. You can update the text in status panels in the following ways:

You can update the text in a status panel manually by calling the `SetStatusPanelMsg()` method of the status panel object. You pass the new text to be displayed as the argument to `SetStatusPanelMsg()`.

You can designate a panel to be updated automatically by calling the `SetStatDispList()` method of the status panel object. You then add method code to the `TestCommand()` method of the container or application to handle updating the status panel.

You can disable the automatic display of Summary help in the Summary panel by setting the `HelpTextVisible` property of the status line to false.

### Updating Status Panels Manually

You update the text in a status panel manually for messages that need to be updated only at specific times—for example, a panel that displays system messages such as “File saved” or “Commit complete”. You update the text in a status panel manually by calling the `SetStatusPanelMsg()` method. For example, the following method code sets the text of panel 2 in the status line “slnMailForm”:

```
slnMailForm.SetStatusPanelMsg(2, "Message Deleted")
```

### Updating Status Panels Automatically

You update the text in a status panel automatically when you need to perform regular checks to determine the panel's text. For example, a panel that displays the current system time needs to be updated constantly. To designate a panel for automatic update, you call the `SetStatDispList()` method.

For example, the following method code designates panel 3 of the status line “slnMailForm” for automatic update:

```
slnMailForm.SetStatDispList(3, Cmd_InsertModePanel, "INS", &  
"OVR", NULL, NULL)
```

➤ **Note:** You cannot use the techniques described in this section with the Summary panel unless you disable the automatic display of Summary help, as described in the section “Disabling the Display of Summary Help” on page 14.42.

---

When you call `SetStatDispList()`, you specify parameters that contain the following information:

- The **position** of the panel, which must be 2 or higher.
- The **command code** of the panel, which is used in the `TestCommand()` method to check on the panel's status.
- The string of text to display when the panel is **enabled**.
- The string of text to display when the panel is **disabled**.
- The string of text to display when the panel is **enabled and checked**.
- The string of text to display when the panel is **disabled and checked**.

The full syntax of the `SetStatDispList()` method is:

### Syntax of `SetStatDispList()`

```
Sub SetStatDispList(pos as Integer, cmdCode as Integer,
    enabled as String, disabled as String, checked as String,
    disabledChecked as String)
```

The command codes you specify must not conflict with command codes that Oracle Power Objects uses for other items (for example, menu commands and toolbar buttons). To avoid conflicts, you should define each command code by adding a different value to the constant `Cmd_FirstUserCommand`. Doing so also ensures that your command codes will not conflict with the current or any future version of Oracle Power Objects.

For example, the following method code defines a set of constants for items in a status panel:

```
CONST Cmd_StatusMessagePanel = Cmd_FirstUserCommand + 100
CONST Cmd_InsertModePanel = Cmd_FirstUserCommand + 101
CONST Cmd_SystemTimePanel = Cmd_FirstUserCommand + 102
```

The following method code uses the constant `Cmd_InsertModePanel` to define the command code for an "Insert Mode" status panel:

```
slnMailForm.SetStatDispList(3, Cmd_InsertModePanel, "INS", &
    "OVR", NULL, NULL)
```

Once you have designated a panel for automatic update, you set the status of the panel by adding method code to the `TestCommand()` method. You can add method code either to the form or report object with which the status line is associated or to the application object. Adding method code to the application object allows you to update panels in a centralized location—for example, your method code can respond the same way to panels on two different status lines.

Oracle Power Objects calls `TestCommand()` at regular intervals when it is not performing any other actions (it is usually called several times each second). `TestCommand()` is actually called once for each panel in the status line, each time with the command code of a different panel as the argument to the method. The return value of `TestCommand()` determines the status of the panel (and thus the text that the panel displays).



You specify the return value of `TestCommand()` using a predefined symbolic constant. The following table summarizes the possible return values for `TestCommand()`:

<b>Constant</b>	<b>Meaning</b>
<code>TestCommand_Enabled</code>	The panel displays the “enabled” text string specified in <code>SetStatDispList()</code> .
<code>TestCommand_Checked</code>	The panel displays the “checked” text string specified in <code>SetStatDispList()</code> .
<code>TestCommand_Disabled</code>	The panel displays the “disabled” text string specified in <code>SetStatDispList()</code> .
<code>TestCommand_Disabled_Checked</code>	The panel displays the “disabled and checked” text string specified in <code>SetStatDispList()</code> .

If you do not explicitly set the return value of `TestCommand()`, the default return value is equal to zero. A return value of zero prompts Oracle Power Objects to check internally whether it handles the command—if it does not handle the command, it disables the command (equivalent to returning `TestCommand_Disabled`).

Method code added to `TestCommand()` generally takes the form of a SELECT CASE statement. Each command code to be handled is a separate case in the statement.

For example, the following method code determines the status of two status panels identified by the constants `Cmd_InsertModePanel` and `Cmd_SystemTimePanel`. A global variable `gInsertMode` keeps track of a text editing mode (“Insert” or “Overwrite”).

#### (Declarations)

```
CONST Cmd_InsertModePanel = Cmd_FirstUserCommand + 101
CONST Cmd_SystemTimePanel = Cmd_FirstUserCommand + 102
GLOBAL gInsertMode AS Integer
GLOBAL gSystemTime AS String
GLOBAL gSystemTimeOld AS String
```

#### Sub Initialize()

```
gInsertMode = TRUE 'Initialize Insert/Overwrite mode
```

#### Function TestCommand(cmdCode as Integer) as Long

```
SELECT CASE cmdCode
CASE Cmd_InsertModePanel
IF gInsertMode THEN
TestCommand = TestCommand_Enabled
ELSE
TestCommand = TestCommand_Disabled
END IF
```

---

```

CASE Cmd_SystemTimePanel
    gSystemTime = FORMAT(NOW(), "H:NN:SS AMPM")
    IF gSystemTime <> gSystemTimeOld THEN
        frmMail.GetStatusLine().SetStatDispList(4, &
            Cmd_SystemTimePanel, gSystemTime, NULL, &
            NULL, NULL)
        gSystemTimeOld = gSystemTime
    END IF
    TestCommand = TestCommand_Enabled
END SELECT

```

The `TestCommand()` method is also where you specify the status of menu items and toolbar buttons, as described earlier in this chapter.

### Disabling the Display of Summary Help

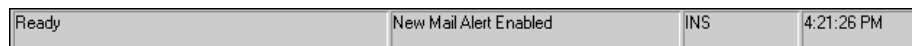
You can disable the automatic display of Summary help in the Summary panel by setting the `HelpTextVisible` property of the status line to False, as shown in the following method code:

```
slnMailForm.HelpTextVisible = False
```

When you disable the automatic display of Summary help, you can use the Summary panel like any other status line panel (however, you cannot delete it). You can re-enable the display of Summary help by setting `HelpTextVisible` to True again.

### Example: Creating a Status Line

This section provides a complete example of creating a status line in an application. The method code listed below creates a status line containing panels that show summary help, a status message, an Insert/Overwrite mode indicator, and the current system time. This status line appears as shown in the following diagram:



The following method code appears in the (Declarations) section of the application:

```

(Declarations)
'Declare constants for status line panels
CONST Cmd_StatusMessagePanel = Cmd_FirstUserCommand + 100
CONST Cmd_InsertModePanel = Cmd_FirstUserCommand + 101
CONST Cmd_SystemTimePanel = Cmd_FirstUserCommand + 102
GLOBAL gNewMailAlert AS Integer
GLOBAL gInsertMode AS Integer
GLOBAL gSystemTime AS String
GLOBAL gSystemTimeOld AS String

```

The following method code appears in the **Initialize()** method of the application:

```
Sub Initialize()
    gNewMailAlert = TRUE    'Initialize New Mail Alert status
    gInsertMode = TRUE     'Initialize Insert/Overwrite mode
```

The following method code appears in the **InitializeWindow()** method of the form "frmMail":

```
Sub InitializeWindow()
    'Declare variable to hold status line object
    DIM slnMailForm AS Object

    'Create the status line object
    slnMailForm = NEW StatusLine

    'Add panels to status line
    slnMailForm.InsertStatusPanel(2, 300, 100)
    slnMailForm.InsertStatusPanel(3, 80, 3)
    slnMailForm.SetStatDispList(3, Cmd_InsertModePanel, "INS", &
        "OVR", NULL, NULL)
    slnMailForm.InsertStatusPanel(4, 80, 12)
    gSystemTime = FORMAT(NOW(), "H:NN:SS AMPM")
    gSystemTimeOld = gSystemTime
    slnMailForm.SetStatDispList(4, Cmd_SystemTimePanel, &
        gSystemTime, NULL, NULL, NULL)

    'Associate the status line object with the form
    frmMail.SetStatusLine(slnMailForm)
```

The following method code appears in the **TestCommand()** method of the form "frmMail":

```
Function TestCommand(cmdCode as Integer) as Long
    SELECT CASE cmdCode
        CASE Cmd_InsertModePanel
            IF gInsertMode THEN
                TestCommand = TestCommand_Enabled
            ELSE
                TestCommand = TestCommand_Disabled
            END IF
        CASE Cmd_SystemTimePanel
            gSystemTime = FORMAT(NOW(), "H:NN:SS AMPM")
            IF gSystemTime <> gSystemTimeOld THEN
                frmMail.GetStatusLine().SetStatDispList(4, &
                    Cmd_SystemTimePanel, gSystemTime, NULL, &
                    NULL, NULL)
                gSystemTimeOld = gSystemTime
```

```

        END IF
        TestCommand = TestCommand_Enabled
    END SELECT

```

The following method code appears in the **DoCommand()** method of the form "frmMail":

```

Function DoCommand(cmdCode as Integer) as Long
SELECT CASE cmdCode
CASE Cmd_NewMailAlert
    gNewMailAlert = NOT(gNewMailAlert)
    frmMail.GetStatusLine.SetStatusPanelMsg(2, &
        IIF(gNewMailAlert, "New Mail Alert Enabled", &
            "New Mail Alert Disabled"))
    DoCommand = TRUE
CASE Cmd_DeleteMessage
    Self.DeleteMessage(lstMessageList.value)
    frmMail.GetStatusLine().SetStatusPanelMsg(2, &
        "Message Deleted")
    DoCommand = TRUE
END SELECT

```

## Properties and Methods

This section lists all of the properties and methods you use to develop menu bars, toolbars, and status lines.

### Menu-Related Properties and Methods

#### Methods of Menu Bar Objects

Method	Description
<b>AppendMenu()</b>	Appends a menu to the end of the menu bar. The menu is inserted before any system default menus that traditionally appear at the end of the menu bar (such as the Help menu in Windows).
<b>ClearMenuBar()</b>	Removes all menus from the menu bar. However, the menu objects are not deleted from the system. You can delete menu objects with the Oracle Basic DELETE statement or with the <b>DeleteAllMenus()</b> method.
<b>DeleteAllMenus()</b>	Removes all menus from the menu bar and deletes the menu objects.
<b>GetMenu()</b>	Returns a reference to a specified menu object in the menu bar.

Method	Description
<code>GetMenuCount()</code>	Returns a count of all menus in the menu bar.
<code>InsertMenu()</code>	Inserts a menu at a specified position in the menu bar.
<code>RemoveMenu()</code>	Removes a menu from a specified position in the menu bar. However, the menu object is not deleted. You can delete menu objects with the Oracle Basic DELETE statement or with the <code>DeleteAllMenus()</code> method.
<code>SysDefaultMenuBar()</code>	Initializes a menu bar with the system default menus. This method deletes any existing menus before initializing the menu bar.

### Properties of Menu Objects

Property	Description
<code>Label</code>	The menu label displayed in the menu bar. To mark a letter of the label as a menu shortcut in Windows, prefix the letter with an ampersand (&).

### Methods of Menu Objects

Method	Description
<code>AppendMenuItem()</code>	Appends an item to the end of the menu. You must specify the item's label, a command code, a help context, and a keyboard equivalent.
<code>DeleteMenuItem()</code>	Deletes an item from a specified position in the menu.
<code>GetItemCount()</code>	Returns a count of all items in the menu, including separator lines.
<code>GetMenuItem()</code>	Returns a specified piece of information about a specified item in the menu. You can get the item's label, the command code, the help context, or the keyboard equivalent.
<code>InsertMenuItem()</code>	Inserts an item at a specified position in the menu. You must specify the item's position, label, a command code, a help context, and a keyboard equivalent.
<code>SetMenuItem()</code>	Modifies a specified piece of information about a specified item in the menu. You can set the item's label, the command code, the help context, or the keyboard equivalent.

---

## Methods of Form and Report Objects

<b>Method</b>	<b>Description</b>
<b>InitializeWindow()</b>	Called when a form or report is first displayed. You generally customize a window's menu bar, toolbar, and status line using method code in <b>InitializeWindow()</b> .
<b>DefaultMenuBar()</b>	Initializes a menu bar with the system default menus and application default menus appropriate to the form or report. This method deletes any existing menus before initializing the menu bar.
<b>DoCommand()</b>	Called when the user selects an item from a menu or clicks on a toolbar button. The command code of the item or button is passed as the argument to <b>DoCommand()</b> .
<b>GetMenuBar()</b>	Returns a reference to the menu bar associated with the form or report.
<b>SetMenuBar()</b>	Associates a menu bar with the form or report.
<b>TestCommand()</b>	Called to determine the status of menu items, toolbar buttons, and status panels. For menus, called for each item in a menu when the user "pulls down" the menu to examine the items. Menu items can appear enabled, disabled, checked, and unchecked.

## Methods of Application Objects

<b>Method</b>	<b>Description</b>
<b>DoCommand()</b>	Called when the user selects an item from a menu or clicks on a toolbar button. The command code of the item or button is passed as the argument to <b>DoCommand()</b> .
<b>TestCommand()</b>	Called to determine the status of menu items, toolbar buttons, and status panels. For menus, called for each item in a menu when the user "pulls down" the menu to examine the items. Menu items can appear enabled, disabled, checked, and unchecked.

## Toolbar-Related Properties and Methods

### Methods of Toolbar Objects

<b>Method</b>	<b>Description</b>
<b>ClearToolbar()</b>	Deletes all buttons from the toolbar.

Method	Description
<b>TBAppendButton()</b>	Appends a button to the end of the toolbar. You must specify the button's command code, a bitmap, a button style, and a help context.
<b>TBDeleteButton()</b>	Deletes a button from a specified position in the toolbar.
<b>TBGetButton()</b>	Returns a specified piece of information about a specified button in the toolbar. You can get the button's command code, the bitmap, the button style, or the help context.
<b>TBGetCount()</b>	Returns a count of all buttons in the toolbar, including separator buttons.
<b>TBInsertButton()</b>	Inserts a button at a specified position in the toolbar. You must specify the button's position, command code, a bitmap, a button style, and a help context.
<b>TBSetButton()</b>	Modifies a specified piece of information about a specified button in the toolbar. You can set the button's command code, the bitmap, the button style, or the help context.

### Methods of Form and Report Objects

Method	Description
<b>InitializeWindow()</b>	Called when a form or report is first displayed. You generally customize a window's menu bar, toolbar, and status line using method code in <b>InitializeWindow()</b> .
<b>DefaultToolbar()</b>	Initializes a toolbar with the application default buttons appropriate to the form or report. This method deletes any existing buttons before initializing the toolbar.
<b>DoCommand()</b>	Called when the user selects an item from a menu or clicks on a toolbar button. The command code of the item or button is passed as the argument to <b>DoCommand()</b> .
<b>GetToolbar()</b>	Returns a reference to the toolbar associated with a form or report.
<b>SetToolbar()</b>	Associates a toolbar with the form or report.
<b>TestCommand()</b>	Called to determine the status of menu items, toolbar buttons, and status panels. For toolbars, called for each button at regular intervals when Oracle Power Objects is not performing any other actions. Also called when the user clicks a toolbar button. Toolbar buttons can appear enabled, disabled, toggled, and untoggled.

---

## Methods of Application Objects

Method	Description
<b>DoCommand()</b>	Called when the user selects an item from a menu or clicks on a toolbar button. The command code of the item or button is passed as the argument to <b>DoCommand()</b> .
<b>TestCommand()</b>	Called to determine the status of menu items, toolbar buttons, and status panels. For toolbars, called for each button at regular intervals when Oracle Power Objects is not performing any other actions. Also called when the user clicks a toolbar button. Toolbar buttons can appear enabled, disabled, toggled, and untoggled.

## Status Line-Related Properties and Methods

### Properties of Status Line Objects

Property	Description
<b>HelpTextVisible</b>	Controls whether the system automatically displays Summary help information in the Summary panel. <b>HelpTextVisible</b> is True by default; to disable the display of Summary help, set <b>HelpTextVisible</b> to False.

### Methods of Status Line Objects

Method	Description
<b>ClearStatusLine()</b>	Deletes all panels from the status line except the Summary panel, which cannot be deleted.
<b>DeleteStatusPanel()</b>	Deletes a panel from a specified position in the status line. The Summary panel cannot be deleted.
<b>GetStatCount()</b>	Returns a count of all panels in the status line.
<b>GetStatPanel()</b>	Returns a specified piece of information about a specified panel in the status line. You can get the text currently displayed in the panel, the panel's width, the command code, or the message strings associated with the panel's status.
<b>InsertStatusPanel()</b>	Inserts a panel at a specified position in the status line. You must specify the panel's width and the maximum message length that can be displayed in the panel.



Method	Description
<code>SetStatDispList()</code>	Designates a panel to be updated automatically. You must specify the panel's command code and the message strings associated with the panel's status.
<code>SetStatusPanelMsg()</code>	Sets the text displayed in a specified panel of the status line.
<code>SysDefaultStatusLine()</code>	Initializes a status line with the system default panels. This method deletes any existing panels before initializing the status line.

### Methods of Form and Report Objects

Method	Description
<code>InitializeWindow()</code>	Called when a form or report is first displayed. You generally customize a window's menu bar, toolbar, and status line using method code in <code>InitializeWindow()</code> .
<code>GetStatusLine()</code>	Returns a reference to the status line associated with the form or report.
<code>SetStatusLine()</code>	Associates a status line with the form or report.
<code>TestCommand()</code>	Called to determine the status of menu items, toolbar buttons, and status panels. For status lines, called at regular intervals for each panel that has been designated to be updated automatically. Setting the status of the panel changes the text displayed in the panel.

### Methods of Application Objects

Method	Description
<code>TestCommand()</code>	Called to determine the status of menu items, toolbar buttons, and status panels. For status lines, called at regular intervals for each panel that has been designated to be updated automatically. Setting the status of the panel changes the text displayed in the panel.



# 15

---

## Oracle Power Objects Extensions

This chapter covers the following topics:

Overview .....	15.2
OLE Data Objects and Controls .....	15.2
Dynamic Link Libraries .....	15.10
OCX Controls.....	15.14

---

## Overview

Oracle Power Objects is designed to be *extensible*, meaning that you can add custom controls, shared data objects, and procedures defined outside Oracle Power Objects. In Microsoft Windows, you can add three types of extension:

- Object linking and embedding (OLE) shared data objects
- Procedures defined in dynamic link libraries (DLLs), including the Windows DLLs.
- OCX custom controls

This chapter summarizes techniques for working with these extensions.

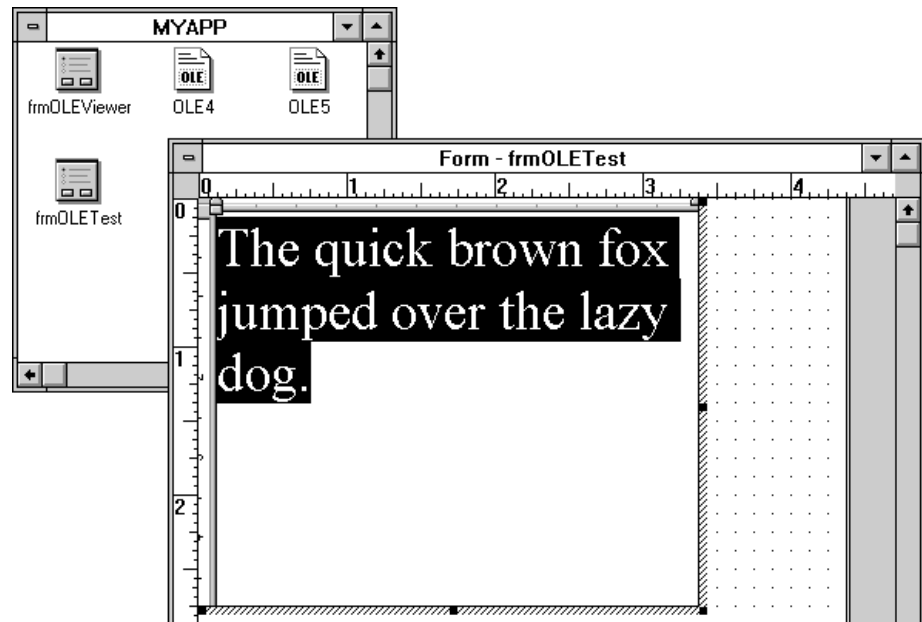
## OLE Data Objects and Controls

*Object Linking and Embedding* (OLE) is a technology for sharing information across applications. In this technology, one application provides the interface for editing this information, while the data is actually stored in a second application. For example, you can display a Microsoft Word document or an Excel spreadsheet in an Oracle Power Objects application. If the user chooses to edit this object, the application displays the user interface from Word or Excel to edit the document or spreadsheet.

OLE features in Oracle Power Objects are available on the Microsoft Windows platform only. To use the OLE capabilities of Oracle Power Objects, you must have the appropriate OLE dynamic link libraries (DLLs) installed on your system. These DLLs are provided with Oracle Power Objects, and can be added to your WINDOWS\SYSTEM directory when you install Oracle Power Objects. The versions of these OLE DLLs provided with Oracle Power Objects are compatible with OCX custom controls, described in the section “Dynamic Link Libraries” on page 15.10. The Oracle Power Objects Installer explains how to install these DLLs, and which versions of the equivalent DLLs they might overwrite.

## OLE in Oracle Power Objects

In Oracle Power Objects, an OLE control is a bindable control used to display an OLE data object. You can store in a table OLE data from the control, including graphics, animation, spreadsheets, word processing documents, charts, and a wide variety of other data.



Before you begin using OLE controls in your application, you should understand the special features of OLE technology in general. This chapter first summarizes some of the basic features of OLE technology, and then describes how to use these capabilities in an Oracle Power Objects application.

### An Overview of OLE

In OLE terminology, the *client* application is where data appear, while the *server* application provides the interface for editing these data. For example, an Oracle Power Objects application in which an Excel spreadsheet appears is the client, while Microsoft Excel itself is the server. This use of the terms *client* and *server* is not analogous to the client and server components of a database application.

---

The server application must be installed on the user's system to be used as an interface for editing. Without the server application, the object may display normally, but the user cannot edit it. For example, if you add a Paintbrush picture to an application, the user can see the graphic whether or not Paintbrush is installed on that client system. However, if Paintbrush is not installed, the user cannot edit the picture.

In some cases, the user can neither view nor edit the OLE data object if the server application is not installed. For example, in the case of QuickTime movies, the user cannot view a movie unless a QuickTime viewer is installed on the client. Whether or not the user can view the data object differs from one OLE object to another.

## Linking and Embedding

When an OLE object appears in an application, it can either be *linked* or *embedded*.

A *linked* object is stored in an operating system file, using the standard file format for that kind of object. The Oracle Power Objects application stores only a reference to the linked object, including the filename and path.

For example, a linked Microsoft Word document appearing in an Oracle Power Objects application is stored in a Word .DOC file somewhere in the operating system. However, you can make changes to the linked OLE object through either the client or server application.

➤ **Note:** If you change the name or location of a linked file, Oracle Power Objects will no longer be able to find it or display its data.

In contrast, the data for *embedded* objects are stored in the client application, not the operating system. For example, the data for an embedded Microsoft Word document would be stored in the Oracle Power Objects application where the document appears. You can make temporary changes to an embedded object, but these changes are undone when you close the client application. For example, if the user edits a Word document embedded in an Oracle Power Objects application, the changes will last only until the user closes the application. This behavior is consistent with other features of Oracle Power Objects: unless the user saves changes to a database or file, then the changes are lost when the application closes down.

The important exception to this rule is embedded OLE objects that appear in a bound OLE control. To save an OLE data object in a table, the OLE object must be embedded, not linked. In this case, the user can save the edited OLE object to a table, and the changes will not be discarded when the user closes the application.

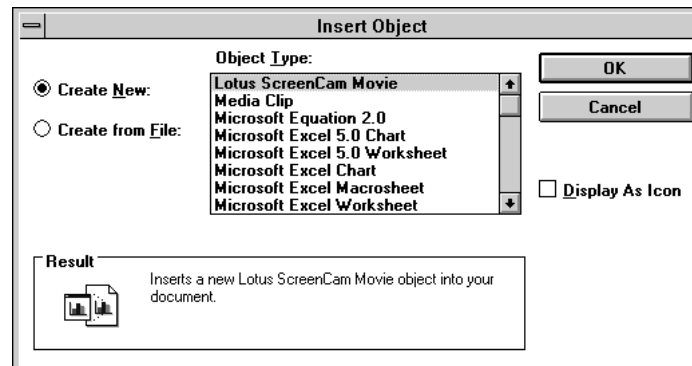
## Classes of OLE Objects

OLE data objects are formatted for a particular server application. Each type of data format defines a separate *class* of OLE data objects. For example, Excel spreadsheets are an example of a class of OLE data objects, designed to be viewed and edited through Excel as a server application. Each class of OLE data objects stores information about the sever application that created it.

Several classes of OLE objects are commonly found in client systems running Microsoft Windows, such as:

- Microsoft Word for Windows documents
- Microsoft Excel spreadsheets
- Microsoft Paintbrush pictures
- Sound (.WAV) files
- QuickTime movies

When you install a new application that can act as an OLE server, Microsoft Windows adds this application to its list of OLE object classes. When you create a new OLE data object, the name of the application then appears in the special dialog for specifying the class of OLE data objects.



## OLE Controls

To allow the user to view and edit an OLE data object, you must display it in an OLE control. You create an OLE control in the same fashion as you create other controls, except that you can launch the server application from the OLE control. You can then view the OLE data object stored in the OLE control through the server application's interface.

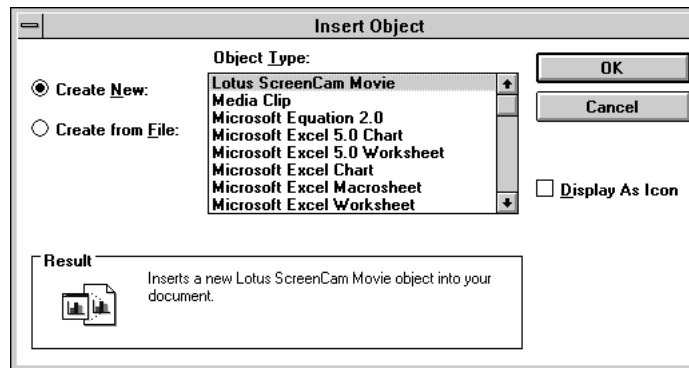
OLE controls in Oracle Power Objects are *bindable*, so you can save the contents of the OLE control in a table. You can specify the class of OLE data object displayed in the OLE control when you create the control. However, the control can display other classes of OLE data objects queried from the same column in a record source. All columns containing OLE data objects must use the LONG RAW datatype.

☆ **To create an unbound embedded OLE object through the Object palette:**



- 1 Choose the OLE Control tool from the Object palette.
- 2 Draw the new control on a container.
- 3 In the dialog that appears, select the OLE object class.

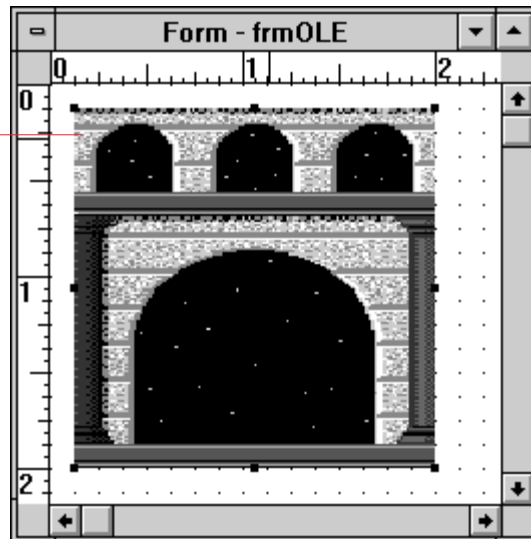
This dialog displays an entry for every OLE server application that has been installed on the client system. You must have the *Create New* option selected at this point.



- 4 Once you have selected the OLE object class, click OK.

The new OLE control now appears within the container, and the interface to the server application appears.

New OLE control





- 5 Edit the new OLE data object, and save it through the server application.

Server applications differ on how to perform this step. In some cases, you need to import or paste the contents of a file into the server application before you can edit it.

- 6 Choose **File-Update** or an equivalent menu command to save the new OLE data object and exit the server application.

Frequently, a dialog asking if you want to return to Oracle Power Objects and update the OLE data object appears.

- 7 Click the **OK** button to return to Oracle Power Objects.

Alternatively, you can paste an OLE object into an Oracle Power Objects application. A new OLE control appears by default for the OLE data object.

☆ **To create an unbound embedded OLE object by pasting it into Oracle Power Objects:**

- 1 From the server application, copy information for a new OLE object to the Clipboard and choose the **Edit-Paste Special** menu command

-or-

Choose the **Edit-Insert Object** menu command.

The server application for the OLE data object then appears.

If you were pasting the OLE object, the data appears in the server application. Otherwise, you can enter the contents of the new OLE data object through the server application.

- 2 Choose **File-Update** or an equivalent menu command to save the new OLE data object and exit the server application.

A dialog asking if you want to return to Oracle Power Objects and update the OLE data object appears.

- 3 Click the **OK** button to return to Oracle Power Objects.

☆ **To create a linked OLE object:**



- 1 Choose the OLE Control tool from the Object palette.

- 2 Draw the new control on a container.

- 3 In the dialog that then appears, select the *Create from File* option.

- 4 Click the **Browse...** button to view a directory of files, and select the file to which the OLE control will be linked.

Alternatively, you can enter the filename and path in the window appearing next to the **Browse...** button.

---

5 Check the Linked check box on the dialog.

6 Click **OK**.

The new OLE control displays data read from the linked file.

☆ **To run or edit the contents of an OLE control:**

1 Double-click on the OLE control to launch the server application.

If you edit the object, an additional step is required when you are finished.

2 Choose the **File-Update** menu command or its equivalent from the server application to update the OLE data object and return to the Oracle Power Objects application.

## Binding OLE Controls

OLE controls can be bound to columns with the LONG RAW datatype on Oracle7 and Blaze, and IMAGE on SQL Server. Bound OLE controls must be embedded, not linked.

You use the same techniques for binding OLE controls that you use when you bind any other kind of bindable control.

☆ **To bind an OLE control:**

1 After creating the new control, enter the name of a column from the container's record source in the **DataSource** property of the control

-or-

With the Table or View Designer window open, select the desired column and drag it onto the OLE control.

## OLE Data Objects and Files

Linked OLE objects are always stored in files, using the file format appropriate to the class of data. For example, when you save a linked Microsoft Word document in an Oracle Power Objects application, the document is stored in a .DOC file in the operating system. The user can edit the file containing the data through the server application, without having to run the Oracle Power Objects application in which there is a link to the file.

Embedded objects behave differently. You can write the contents of an embedded OLE data object to a file, but they are stored in a slightly different format than "native" files created in the server application. However, you can read from and write to these files through Oracle Power Objects, using the server application's interface.

For more information on binding, see Chapter 17, "Binding a Container to a Record Source".

Files containing OLE data objects can store only one object. If you write to an existing file, you overwrite its contents with the new OLE data object.

OLE controls have two methods that control file input and output for embedded OLE data objects:

<b>Method</b>	<b>Description</b>
<b>WriteColToFile()</b>	Writes data from the OLE control to the file. The contents of the file cannot be read from the server application. However, you can read the contents of the file into an OLE control using the <b>ReadColFromFile()</b> method.
<b>ReadColFromFile()</b>	Reads data from the file to the OLE control. The OLE data object in the file must have been written to the file with the <b>WriteColToFile()</b> method.

Both methods take one *String* argument, which specifies the path and filename of the file containing the data object.

## OLE Data Objects and the Clipboard

Three other methods let you copy OLE data objects between an OLE control and the Clipboard.

<b>Method</b>	<b>Description</b>
<b>CanPasteFromClipboard()</b>	Returns True if an OLE data object is in the Clipboard, or False if there is not.
<b>PasteFromClipboard()</b>	Pastes an OLE data object from the clipboard into the OLE control.
<b>CopyToClipboard()</b>	Copies the OLE data object stored in the OLE control to the clipboard.

For more information on these methods, consult the online help.

## Restrictions on OLE-Enabled Applications

You should note that if you create an application that uses OLE controls, and you then try to modify or run the application on a Macintosh or a Windows system that does not have the proper OLE DLLs installed, then the application will not open or run.

---

## Dynamic Link Libraries

A dynamic link library (DLL) is a file object that stores procedures called from other applications. DLLs are available only on the Microsoft Windows platform.

You can use two kinds of DLLs: your own, and the Windows DLLs included with the operating system. By writing C code and compiling it, you define your own DLL. The basic system library files behind Microsoft Windows include a wide array of predefined procedures that create and manipulate objects and data in the Windows environment.

To use any DLL, you must declare it first in the (Declarations) section of your application's Property sheet, specifying the following DLL components:

- The name of the procedure (used when calling it).
- The type of the procedure (subroutine or function). If the procedure is a function, it has a return value, which must be declared.
- The DLL file in which the procedure is defined. This file must be in the same directory as the Oracle Power Objects application, or in a directory included in the operating system's path. Alternatively, you can specify the path with the filename.
- The arguments passed to the procedure, including their datatype, as well as whether the argument is passed by value or by reference.
- For functions, the datatype of the return value.

Once you make the declaration, you can then call the procedure from any part of an Oracle Power Objects application. This section summarizes the techniques for declaring and calling DLL procedures.

This section does not instruct you on the methods for writing and compiling DLLs. For this information, consult a C programmer's manual.

### Declaring DLL Procedures

All DLL procedures used within an application must be declared at the application level, even if they are used only within a single form or report. All such declarations must occur in the (Declarations) section of the application's Property sheet.

➤ **Note:** The (Declarations) section appears only in the Property sheet of an application object.

For more information about arguments, see the section "Passing Arguments to a Procedure" on page 15.13.

To declare a DLL procedure, use the following syntax:

```
DECLARE type name LIB "library" (arguments) AS return_type
```

Item	Description
<i>type</i>	The type of procedure, either SUB for subroutines or FUNCTION for functions.
<i>name</i>	The name of the procedure as it appears in the library.
<i>library</i>	The filename of the DLL. Frequently, the file has the extension .DLL, though in the case of the Windows API, you do not enter a .DLL extension. (For more information, see the section "The Windows API" on page 15.13) If the DLL is not in the application directory or in the path, you must enter the DLL's path as part of the filename.
<i>arguments</i>	The arguments passed to the DLL. Each argument must have a declared datatype. In addition, if you pass the parameter variable for the argument by value, you must use the ByVal keyword.
<i>return_type</i>	The datatype of the return value.

## Sample Declarations

Here is a sample declaration for a procedure:

```
DECLARE FUNCTION MyProc LIB "MYLIB.DLL" (ByVal Arg1 As &
Integer) AS Long
```

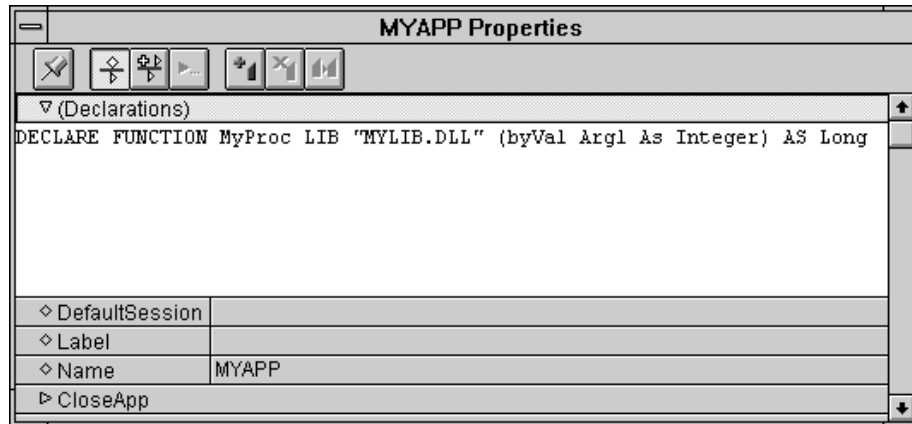
This declaration identifies a procedure *MyProc* as a function defined in the file MYLIB.DLL. This function takes a single argument, passed to the procedure by value, and returns a value of the Long datatype. The ampersand (&) is a line concatenation character used only for displaying code within a limited space; the symbol is necessary only if the procedure declaration extends over multiple lines in the Property sheet's code window.

The following declaration identifies a Windows API procedure used to play sound (.WAV) files.

```
Declare Function sndPlaySound Lib "MMSYSTEM.DLL" &
(ByVal x$, ByVal Y%) As Integer
```

---

Once you have declared a DLL procedure, it is global to the application: you can call it from any object within the application.



## Creating Flexibility in DLL Procedures

DLL procedures can be flexible in two ways:

- 1 The same procedure can take a variable number of arguments.
- 2 The same argument can accept values of multiple datatypes.

For information about overloading method declarations, see the section "Overloading Method Declarations" on page 5.7.

Procedures are commonly defined to accept a fixed number of arguments. If the call to a function does not include a value for every argument in the argument list, then Oracle Power Objects fails to make the call, and the application displays an error. However, some procedures can be designed to treat some arguments as optional. In such cases, you can pass successfully only some of the arguments in the argument list, without causing an error. For information on writing DLL procedures with optional arguments, consult a programmer's manual for the C language. Similarly, you can "overload" a method, so that it can take a variable number of arguments.

You can also build flexibility into the datatype required for an argument. If you use the *As Any* keyword for the datatype, the argument accepts any valid datatype in Oracle Basic. For example, the following sample declaration indicates that the second argument can accept any datatype:

```
DECLARE SUB MySub LIB "MYLIB.DLL" (Arg1 As String, Arg2 As Any)
```

## Calling DLL Procedures

To call a DLL procedure, use the following syntax:

```
procedure_name (arguments)
```

If the procedure is a function, you must also capture the return value in a variable through Oracle Basic. For example, the following call to a DLL procedure captures a return value in a *Long* variable:

```
DIM vRetVal AS Long
vRetVal = MyProc(100)
```

Alternatively, you can capture the return value in a property, as shown in this example:

```
fldDateApplied.Value = MyProc(100)
```

You can use the CALL statement in Oracle Basic when calling a procedure, but use of this statement is purely optional.

## Passing Arguments to a Procedure

Arguments passed to a procedure can be values (such as 100, "Name"), variables (such as vArg1, vApplyDate), or properties (such as fldUserName.Value).

Unless you use the *ByVal*/keyword when naming the argument in the function declaration, you pass an argument by reference to the procedure. The value of the variable or property passed by reference can be changed within the procedure. Only variables and properties can be passed by reference; all other values must be passed by value.

For example, the following procedure declaration includes a parameter called OldDate that is passed by value:

```
(Declarations)
DECLARE FUNCTION FixDate LIB "MYLIB.DLL" (OldDate AS &
Date)
```

When this procedure is called, the argument is supplied through a variable called vDateApplied. The value of vDateApplied can be changed within the procedure, since this variable is passed by reference:

```
fldDateApplied.Value = FixDate(vDateApplied)
```

When you pass an argument by reference, you pass the memory address of the variable or property, instead of the value stored in that variable. In contrast, when you pass an argument by value, you pass the value in that variable. Therefore, you can use this value in the DLL procedure, but you cannot change the value assigned to the variable. You specify that you pass an argument by value in the function declaration only, not in the actual call to the function.

## The Windows API

Procedures in the Windows Application Programming Interface (API) provide powerful capabilities for working with objects and data within the Windows environment. Windows includes API procedures for a wide array of tasks, from allocating memory to displaying images. Descriptions of

---

these procedures are widely available, including both third-party books and Microsoft publications. For a complete listing of these procedures, including declarations and explanations of each of them, consult those references.

The Windows API procedures are defined in several libraries, not all of which have the extension .DLL as part of their filename. When you declare a Windows API procedure, you therefore use the library file's name without the extension. The major libraries for Microsoft Windows, as identified in API procedure declarations, include:

- USER
- GDI
- MMSYSTEM
- KERNEL

Most declarations of a Windows API procedure will identify one of these libraries for the LIB section of the declaration.

## Other Considerations

As a developer, you do not need to know how the procedure is implemented in a DLL to call it. In development terminology, the declaration of the procedure provides an interface to the procedure, by providing the means to call the procedure, pass arguments to it, and capture a return value from it. You can make effective use of a DLL procedure without knowing the means through which the procedure interprets the arguments, performs processing, and returns a value.

However, the fact that a DLL is opaque to the developer calling procedures defined in it can be a disadvantage. DLLs cannot use the object-oriented Oracle Basic language to manipulate objects within Oracle Power Objects. Other developers working on the same application may find it more difficult to understand and debug a procedure in a compiled DLL than its equivalent defined through method code within Oracle Power Objects. Therefore, if there is not a pressing reason (such as performance) for defining a procedure in a DLL, you should define it through method code in Oracle Power Objects.

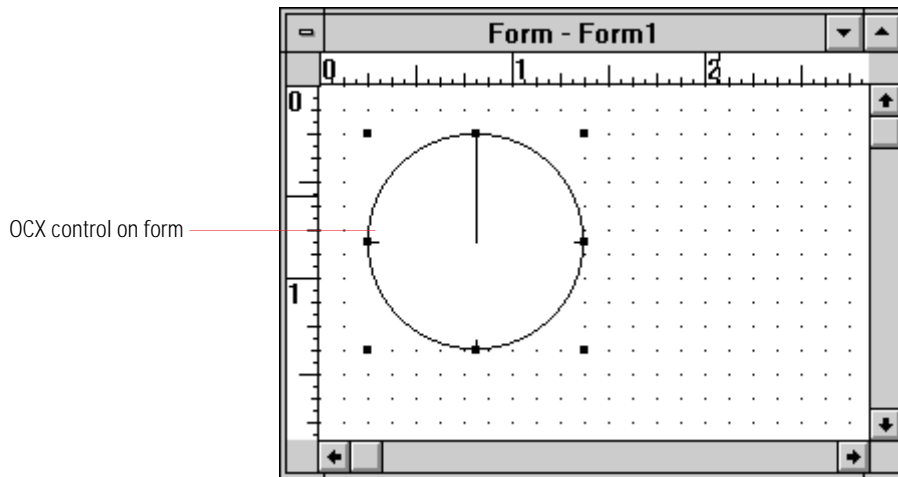
## OCX Controls

OLE Control Extension (OCX) controls are custom controls that you can add to Oracle Power Objects applications, as well as applications created for Visual Basic version 4.0. When you add a new OCX custom control to Oracle Power Objects, a new button for its drawing tool appears on the Object palette. You then add a control of this type to a container in the same fashion as you would add a pushbutton, text field, repeater display, or any other application object appearing on the Object palette.



Examples of possible OCX controls include:

- A spin button control
- A clock control
- An animation control



## Creating OCX Controls

OCX controls are defined through C code, then compiled into a file that normally has the .OCX or .DLL extension. Microsoft Corporation has published the API for OCX controls, so that you can designate components of the OCX as properties and methods. Oracle Power Objects recognizes these components and interprets them as properties and methods, according to the following rules:

- **If the name of the OCX feature matches the name of an Oracle Power Objects standard property or method**, then the OCX control's Property sheet displays these features as standard properties and methods. In the OCX, you should not define any processing or return value for a standard method, because Oracle Power Objects already performs default processing when the method is called (which includes the datatype of the return value). If Oracle Power Objects interprets the feature as a standard property, then the OCX property should have the same datatype as the standard property in Oracle Power Objects.
- **If the name of the OCX feature does *not* match the name of an Oracle Power Objects standard property or method**, then Oracle Power Objects identifies the feature as unique to that OCX control. If the feature is a property, then you must give it a datatype that Oracle Power Objects can interpret. If the feature is a method, then you must give it a datatype for the return value that Oracle Power Objects can interpret. Additionally, you must define the default processing for the method within the OCX.

---

For a full description of the API for OCX controls, consult an OCX reference manual.

## Importing an OCX into Oracle Power Objects

Before you can use an OCX control, you must import it into Oracle Power Objects.

☆ **To import an OCX custom control into Oracle Power Objects:**

- 1 In the active Designer window for a form, report, or user-defined class selected, choose the **File-Load OLE Control** menu command.
- 2 In the dialog that appears, select the path and filename of the custom control.  
OCX controls generally have the .OCX or .DLL extension in their filenames.
- 3 Click **OK** to import the selected OCX.

The new OCX now appears at the bottom of the Object palette.

The OCX control is added to the Object palette



☆ **To add an OCX control to a container:**

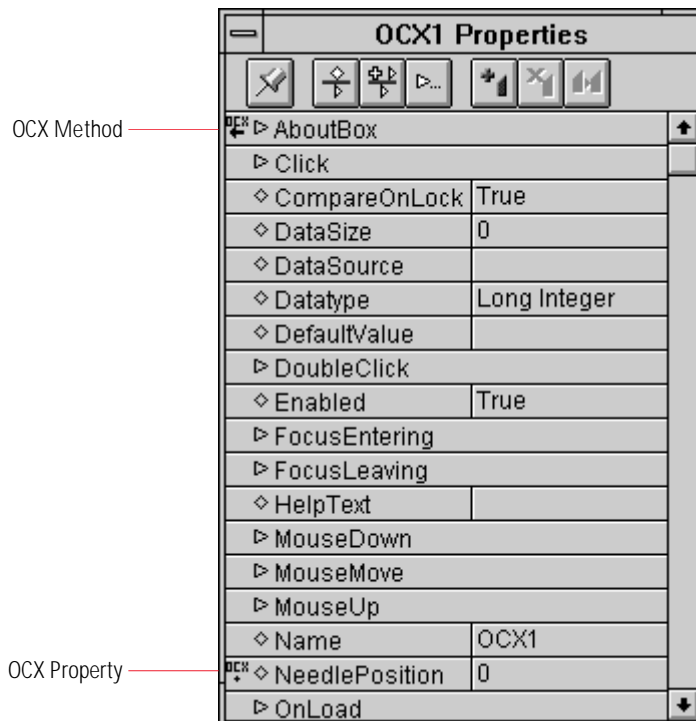
- 1 Select a Designer window for a form, report, or user-defined class.
- 2 Click on the button in the Object palette for the OCX's drawing tool.
- 3 Draw the new OCX control on the container by clicking on the approximate location where you want the control to appear, applying a default size to the OCX.

-or-

Click and drag across the region of the container where you want the control to appear, then release the mouse button when it has approximately the dimensions you want to give it.

## OCX Properties and Methods

You can add standard properties and methods to an OCX (for example, **Click()** and **Value**), as well as properties and methods unique to the OCX. In the latter case, the property or method appears on the Property sheet with a special symbol appearing near its name, designating it as an OCX-specific property or method.



Some important considerations about OCX-specific properties and methods:

- You can override the default processing for an OCX-specific method by adding method code to it. Additionally, you can call the overridden processing with the `Inherited.method_name()` statement.
- OCX-specific methods cannot return object references.
- OCX controls are not bindable. Even if you add **RecordSource** and **RecSrcSession** properties to an OCX control, Oracle Power Objects will not be able to query records for the control.

OCX controls often have a built-in Property sheet that you can summon by doubleclicking on the control at design time. These Property sheets display the **Name** property, as well as any properties specific to that OCX. If you make changes through this Property sheet, they are reflected in the standard Oracle Power Objects Property sheet.

---

## **Restrictions on OCX-Enabled Applications**

Any application created with an OCX control must include the .DLL or .OCX file with it when you distribute it.

If you create an application that uses OCX controls, and you then try to modify or run the application on a Macintosh or a Windows system that does not have the proper OLE DLLs installed, then the application will not open or run.

# 16

---

## Compiling the Executable Application

This chapter covers the following topics:

Overview .....	16.2
Generating Application Files and Executables .....	16.3
Creating Run-Time Executable Files .....	16.4
Creating Standalone Executable Files .....	16.4

---

## Overview

Compiling your application makes it execute more efficiently. Oracle Power Objects provides two compilation techniques. One technique produces a file that requires the Oracle Power Objects Run-Time application (called PWRRUN.EXE on Windows) to run it. The other technique produces a standalone file that integrates your application with that run-time executable, enabling standalone execution on comparable machines whether Oracle Power Objects is installed or not.

### Compiling for More Efficient Execution

Once your application is fully tested, you can compile it to provide more efficient execution. The result is a file that can be run directly by the Oracle Power Objects run-time executable. The compiled file uses the same filename as the source, with an extension of .PO rather than .POA on Windows. For example, the compiled version of MYAPPPOA is MYAPPPO.

The Oracle Power Objects Run-Time application coordinates the execution, on any Macintosh or Windows machine, of all the Oracle Power Objects features built into your application. All the features it relies on are available, including session/database access, table/row/column control and manipulation, and the user interface you created, without requiring additional resources from the environment or the operating system. However, the Oracle Power Objects Run-Time application does not include the database driver files (.POD files) required for database access. You must include any required .POD files along with the Oracle Power Objects Run-Time application.

### Compiling for Standalone Execution

You can also create a self-contained, standalone executable that combines the Oracle Power Objects run-time executable with your application (this file has the extension .EXE on Windows). Being self-contained, this file can run directly when launched on any Macintosh or Windows machine (corresponding to the machine you compiled on), without the need to launch Oracle Power Objects first.

Creating a standalone executable also enables you to distribute your application widely. Your users need not have Oracle Power Objects or the Oracle Power Objects Run-Time application. However, the standalone executable does not include the database driver files (.POD files) required for database access. You must distribute any required .POD files along with your executable file.

## Generating Application Files and Executables

You can use the toolbar or the Run menu to compile your application.



If you click the Generate Application button on the toolbar, the following dialog appears:



Clicking **OK** while the “Separate application file” radio button is selected causes Oracle Power Objects to present a dialog enabling you to name the run-time executable file you want created, such as MYAPPPPO.

If you click the “Standalone executable” radio button instead, Oracle Power Objects presents a dialog enabling you to name the executable file you want created, such as MYAPPEXE. On the Macintosh, a standalone executable file is given a file type of APPL.

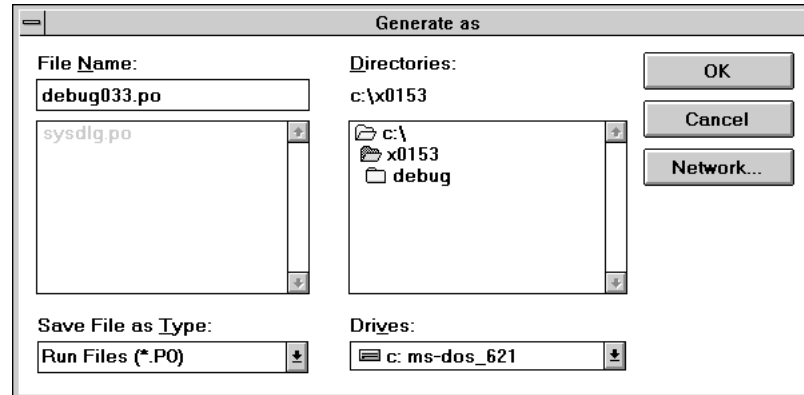
If you use the Run menu instead of the toolbar, choosing the **Run-Compile Application** menu command displays the dialog for creating run-time executable (.PO) files, and choosing **Run-Generate Executable** displays the dialog for creating standalone executable (.EXE) files.

Generated applications automatically include any bitmaps, user-defined classes, and Database Session objects referenced by the application. Such references are in Oracle Basic scripts, referenced by name such as *lib\_name.bitmap\_name*, where *lib\_name* is the name of the library.

---

## Creating Run-Time Executable Files

The dialog for creating run-time executable (.PO) files follows:

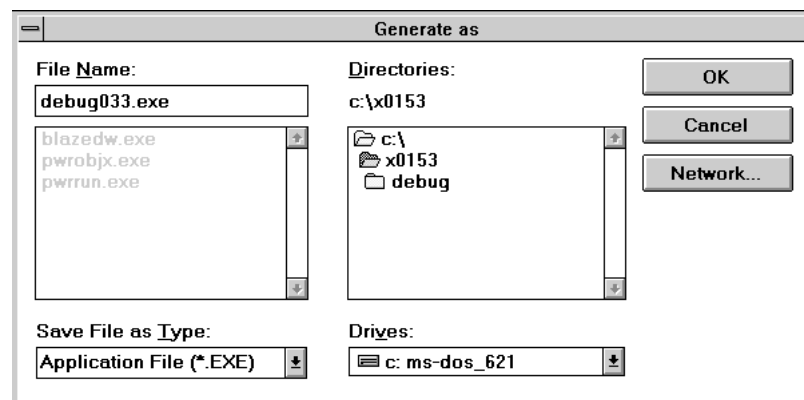


You can change the name, drive, and directory. When you click OK, Oracle Power Objects creates your run-time executable file.

When you later run the Oracle Power Objects Run-Time application, it searches the current directory for run-time executable files. When you choose one, Oracle Power Objects runs the specified application.

## Creating Standalone Executable Files

The dialog for creating standalone executable (.EXE) files follows:





Standalone executable files can be run anywhere other executable files can be run, such as from the DOS prompt or through the Windows File-Run command. Similarly, executable applications on the Macintosh can be launched directly without first launching Oracle Power Objects.



# 17

---

## **Binding a Container to a Record Source**

This chapter covers the following topics:

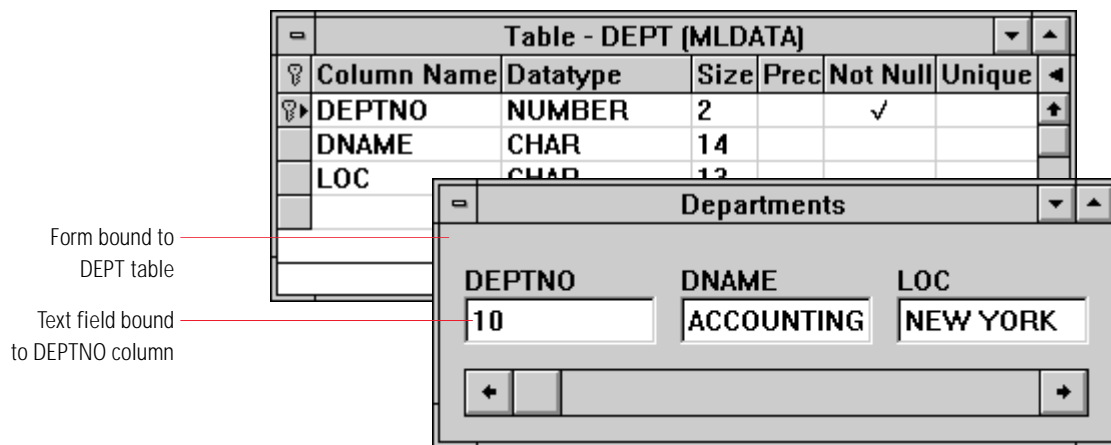
Overview .....	17.2
Binding Objects Graphically.....	17.4
Binding Objects Manually by Setting Properties .....	17.8
Recordset Objects and Bound Containers .....	17.9

---

## Overview

*Binding* is the primary development technique for connecting application objects (containers and controls) to database objects (tables and views). Once an application object has been bound to a database object, the application object displays at run time data derived from the table or view.

For example, if you bind a form to a table, the form displays values from the rows of the table, as shown in the following diagram:



Binding involves the following steps:

- 1 Binding a container (such as a form or report) to a *record source* (a table or view)
- 2 Binding controls within the container to columns of the record source
- 3 Determining how rows are derived from the record source by modifying properties of the bound container

This chapter first describes two ways of binding application objects to database objects:

- **Graphically**, using drag-and-drop functionality to automate the binding process.
- **Manually**, by setting the values of several key properties of bound containers and controls.

The chapter then describes *recordsets*, the in-memory objects that mediate between bound application objects and record source database objects.

## Relationship Between Containers and Record Sources

A container that can be bound to a record source is said to be *bindable*. For a container to be bindable, it must have a set of properties that define its relationship to the record source. The most important of these properties are described in the section “Binding Objects Manually by Setting Properties” on page 17.8. A complete list of container properties related to binding is provided in the section “Properties and Methods of Recordsets and Bindable Objects” on page 17.31.

The following types of containers are bindable:

- Embedded forms
- Forms
- Repeater displays
- Reports
- User-defined classes

For information on master-detail relationships, see Chapter 18, “Defining Master-Detail Relationships”.

The location of a bound container is not important—it operates the same way whether it is contained within bindable or nonbindable containers. If a bound container is located inside another bound container, each container operates independently unless the containers are associated. To associate bound containers, you can establish a *master-detail relationship*. Alternatively, you can use a *shared recordset*, as described in the section “Shared Recordsets” on page 17.19.

A container can be bound to only one record source at a time. For example, you cannot bind the same form to two tables at once, although other containers *on* the form (such as repeater displays or embedded forms) can be bound to other tables or views. To display information from multiple tables or views in the same container, you can use the following techniques:

- Create a view that relates the objects and use the view as the record source. This technique enables you to present complex data from several tables (however, the data in multi-table views is read-only).
- Use a translation list to display information from detail tables. This technique enables you to display read-write values in scrolling lists, popup lists, and combo boxes.
- Use the SQLLOOKUP function in the **DataSource** property of a field to display information from a detail table. This technique enables you to display read-only values in derived fields.

For information about views, see the section “Views” on page 8.16.

For information about translation lists, see the section “List Controls” on page 10.15.

## Relationship Between Controls and Record Sources

A control that can be bound to a column of a record source is said to be *bindable*. For a control to be bindable, it must have a **DataSource** property to define the column from which it derives its value. A control cannot be bound unless it is contained in a bound container.

For information about SQLLOOKUP, see the section “The SQLLOOKUP Function” on page 9.21.

---

The following types of controls are bindable:

- Chart controls
- Check boxes
- Combo Boxes
- List Boxes
- OLE controls
- Picture controls
- Popup Lists
- Radio button frames
- Radio buttons \*
- Text fields

\* Radio button objects, while technically bindable, are not normally bound directly to a column. Instead, the radio button frame object containing a group of radio buttons is bound to the column.

When a control is bound to a column, the control's **Value** property is linked to the values in the column. As the user scrolls through records (for example, using a scrollbar control), the **Value** property of the control is automatically set to the column value for the displayed row.

## Other Ways to Connect Objects

Aside from binding, two additional techniques allow you to connect application objects with database objects:

- Use the `SQLLOOKUP` function to derive a single value from a database object.
- Use the `EXEC SQL` command to execute any valid SQL command and transfer information between the database and the interface.

For information about `SQLLOOKUP` and `EXEC SQL`, see Chapter 9, "Structured Query Language (SQL)".

## Binding Objects Graphically

The simplest way to bind objects is to use drag-and-drop procedures to establish the relationship. This involves dragging elements from a database window (a Database Session window, Table Editor window, or View editor window) into a Designer window (a Form Designer window, Report Designer window, or Class Designer window).

There are two ways to use drag-and-drop to bind objects:

- **Drag a table, view, or column onto a container.** Doing so binds the container to the record source and automatically creates controls that represent columns from the record source.

- **Drag a column onto a control.** Doing so binds the control to the column and automatically binds the control's container to the column's record source. Some properties of the control (such as the **Name** and **Datatype** properties) are automatically updated.

These techniques are described below.

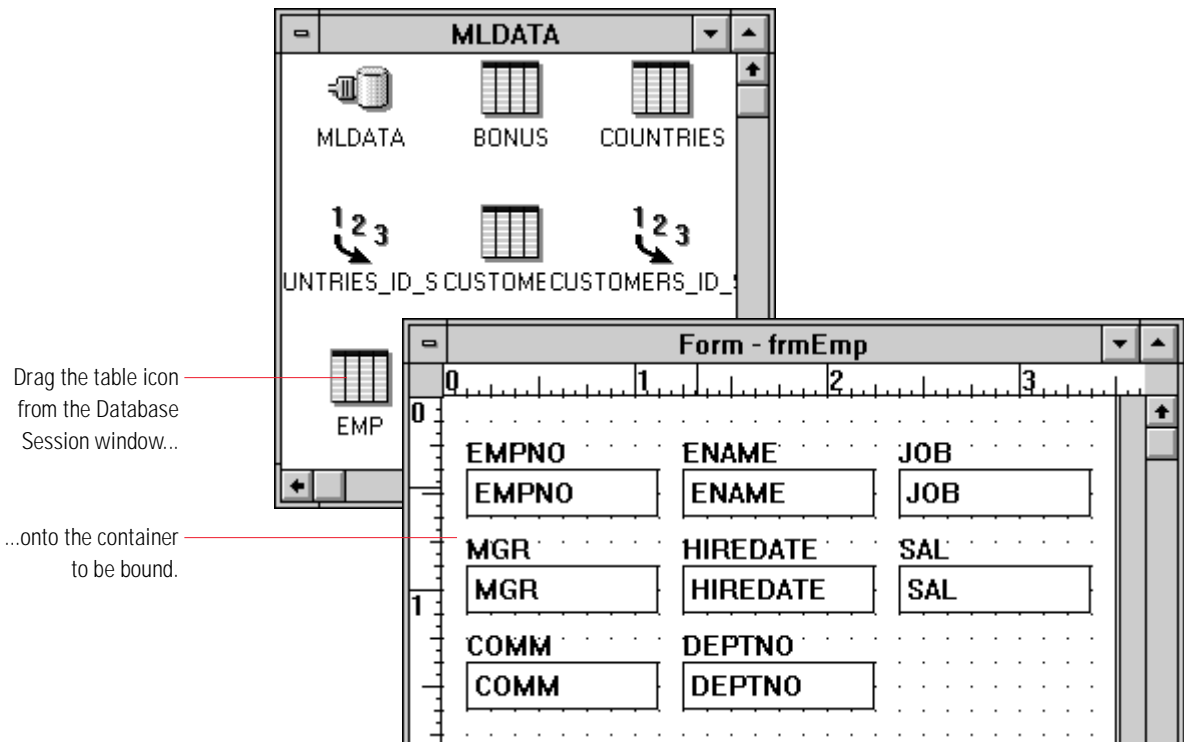
☆ **To bind a container to a table or view:**

- 1 Open the Designer window of the form, report, or class of the container to be bound.
- 2 Without closing the designer window, open the Database Session window containing the record source you want to bind to.

To open the database session window, activate the Main window by choosing the **Window-Main** menu command, then double-click on the session icon.

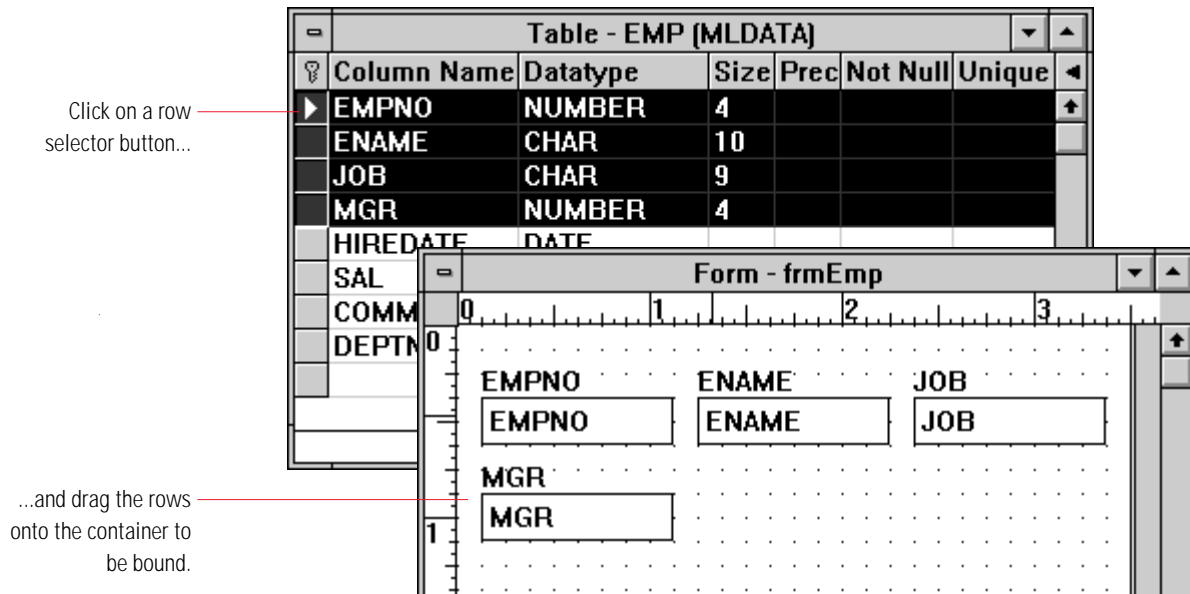


- 3 If the session is not currently active, double-click on the Connector control to activate the session.
- 4 Click on the icon representing the table or view and drag the icon from the Database Session window directly onto the container.



☆ **To bind a container to individual columns:**

- 1 Open the Designer window of the form, report, or class of the container to be bound.
- 2 Without closing the designer window, open the Database Session window containing the record source you want to bind to.  
To open the database session window, activate the Main window by choosing the **Window-Main** menu command, then double-click on the session icon.
- 3 Select the database columns you want by holding down the Control key (Command on Macintosh) and clicking on the Row Selector button next to the spreadsheet row representing each database column.
- 4 Click on one of the selected row selector buttons and drag the spreadsheet rows from the Database Session window directly onto the container.



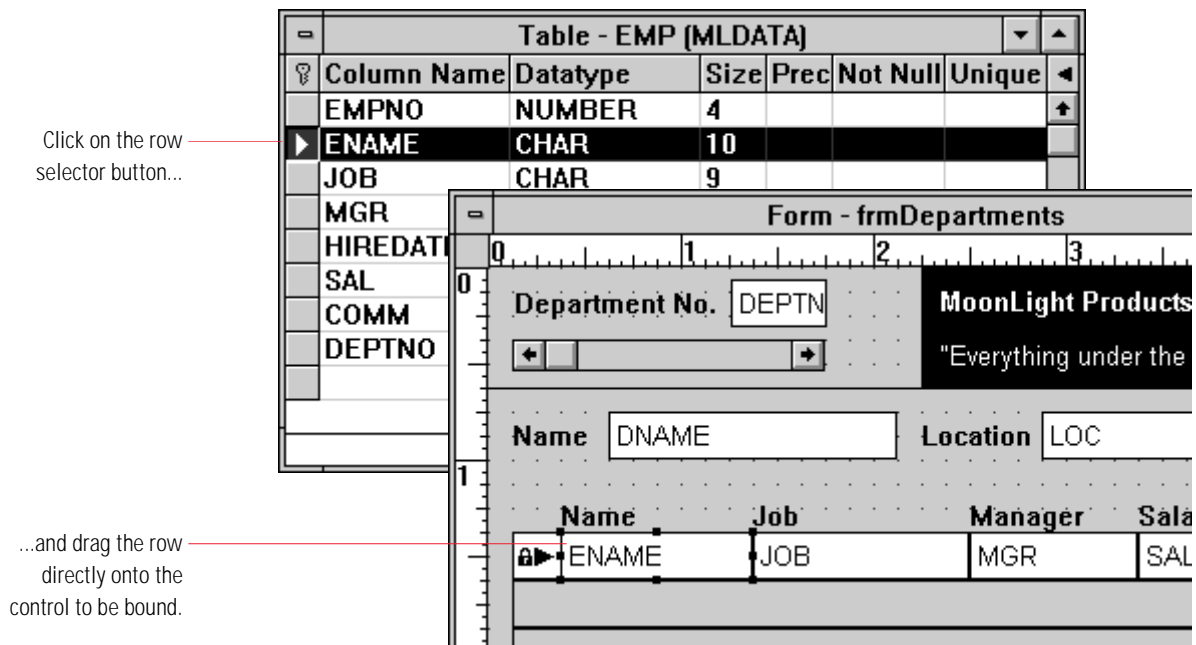
☆ **To bind a column to an existing control:**

- 1 Open the Designer window of the form, report, or class of the control to be bound.
- 2 Without closing the designer window, open the Database Session window containing the record source you want to bind to.

To open the database session window, activate the Main window by choosing the **Window-Main** menu command, then double-click on the session icon.



- 3 Select the database column you want by clicking on the Row Selector button next to the spreadsheet row representing the column.
- 4 Click on the row selector button and drag the spreadsheet row from the Database Session window directly onto the control.



When you bind database objects graphically, the following container properties are automatically set:

- The **RecordSource** property is set to the name of the record source table or view.
- The **RecSrcSession** property is set to the name of the session from which you dragged the record source.

In addition, the following control properties are automatically set:

- The **DataSource** property is set to the name of the column associated with the control.
- The **DataType** property is set to a value appropriate for the datatype of the associated column.
- If the column is a string-type column (for example, CHAR or VARCHAR2), the **DataSize** property is set to the column size. For example, when you associate a VARCHAR2(100) column with a field, the field's **DataSize** property is set to 100.
- The **Name** property of the control is set to the name of the column (upper-case letters are used). If another object already exists with the column name, an integer is automatically appended to the name.
- The **ScrollWithRow** property of the control is set to True.

---

## Binding Objects Manually by Setting Properties

You can also bind a container manually by setting two specific properties of the container (these properties are set automatically when you use drag-and-drop). To bind a container manually, you set the following properties:

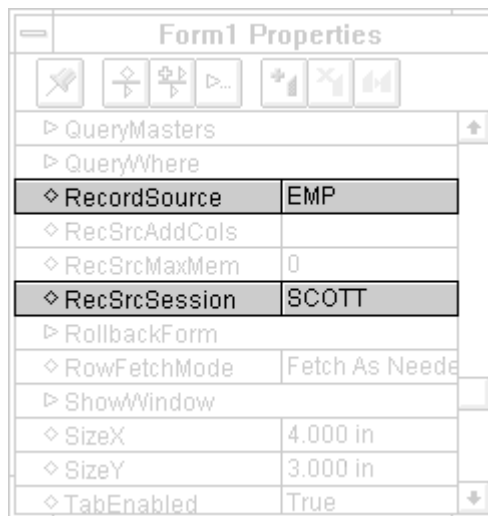
- Set the **RecordSource** property of the container to the name of the record source table or view.
- Set the **RecSrcSession** property of the container to the name of the database session object containing the record source. If you do not specify a **RecSrcSession**, the default session is used (the session specified by the **DefaultSession** property of the application, if defined).

Once you have bound the container, you can create controls within the container and bind those controls to columns of the record source. To bind controls, you set the following properties:

- Set the **DataSource** property of each control to the name of a column of the record source.
- Set the **DataType** and **DataSize** property of each control to values appropriate to the data in the column.

For example, to associate the form “Form1” with the EMP table in the “Scott” session, you would set the following properties:

- Set the **RecordSource** property to `EMP`.
- Set the **RecSrcSession** property to `SCOTT`.



To associate a field on “Form1” with the ENAME column of the EMP table, you would set the following properties:

- Set the **DataSource** property to `ENAME`.
- Set the **DataType** property to `String`.
- Set the **DataSize** property to `10`.

Once you have bound the objects to the database, you can set additional properties of the bound container to determine how rows are derived from the record source. Two commonly set properties are **OrderBy** and **DefaultCondition**. These properties can be set both at design time and at run time.

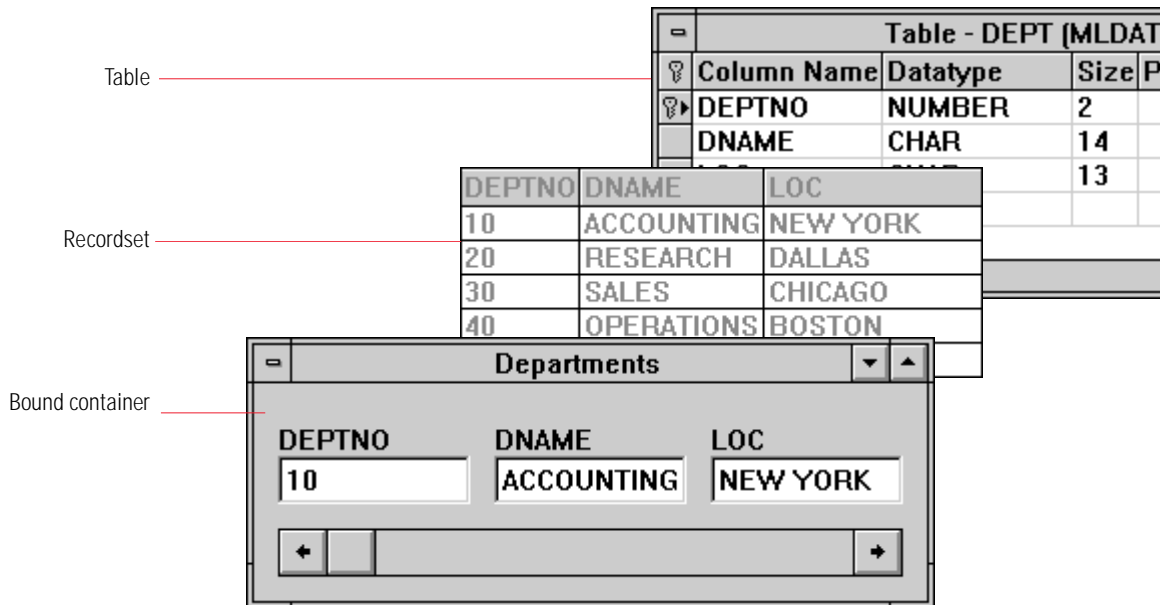
- The **OrderBy** property specifies the name of the column or columns by which the rows in the recordset should be ordered. Rows are retrieved in ascending order by default; you can specify descending order by including the word “DESC” after the column names. If **OrderBy** is not set, the rows in the recordset are unordered.
- The **DefaultCondition** property specifies a SQL condition (a WHERE clause) used to restrict the rows in the recordset. For example, to have the recordset contain only rows that have a SAL column value exceeding 5000, you could set the **DefaultCondition** property to `SAL > 5000`.

Other container properties related to binding are described in the following sections of this chapter. A complete list of container properties related to binding is provided in the section “Properties and Methods of Recordsets and Bindable Objects” on page 17.31.

## Recordset Objects and Bound Containers

Each bound container in your application has an associated *recordset object*. A recordset object contains a local copy of a set of rows queried from a database table or view. As the user browses through rows of data in a bound container, the necessary rows are fetched from the database. As the user makes changes to the data, the recordset records the changes and writes them to the database when appropriate.

A recordset is thus an intermediary between objects that the user can see and objects in the database.



For information about translation lists, see the section "List Controls" on page 10.15.

Recordset objects are also associated with translation lists (Popup List and List Box controls whose **TranslationList** property has been set).

Recordset objects that are associated with application objects are called *container recordsets*. You can also create *standalone recordsets* by executing Oracle Basic code; these recordsets are not associated with any application object. Standalone recordsets are described in the section "Standalone Recordsets" on page 17.27.

## Structure of a Recordset

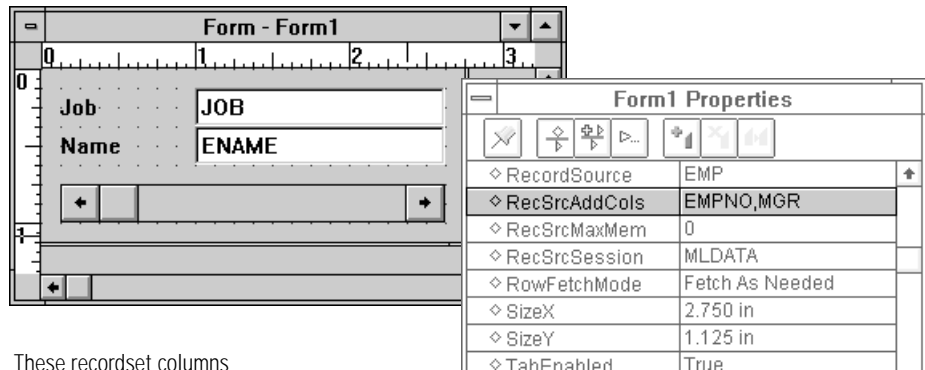
A recordset, like a table or view, is organized into rows and columns. The rows and columns in the recordset are derived from the structures of the database and application objects that are bound together.

A recordset generally has one column for each bound control within the corresponding container. However, if multiple controls are bound to the same column, the column appears only once in the recordset.

Each recordset column has a datatype (an Oracle Basic datatype, not a SQL datatype). Oracle Power Objects automatically determines the appropriate datatype to use to represent a particular database column.

You can add nondisplayed columns to a recordset by setting the **RecSrcAddCols** property of the associated container. **RecSrcAddCols** can contain the names of one or more columns of the record source, separated by commas. These columns can be accessed only by executing methods of the recordset object itself, as described in the section “Accessing a Recordset Programmatically” on page 17.16.

The following diagram shows a recordset with columns derived from the **RecSrcAddCols** property:



These recordset columns are derived from controls on the form.

JOB	ENAME	EMPNO	MGR
PRESIDENT	KING	7839	
MANAGER	JONES	7566	7839
MANAGER	BLAKE	7698	7839
MANAGER	CLARK	7782	7839
CLERK	MILLER	7934	7782
ANALYST	SCOTT	7788	7566
ANALYST	FORD	7902	7566
SALESMAN	ALLEN	7499	7698

These columns are derived from the **RecSrcAddCols** property.

For information on master-detail relationships, see Chapter 18, “Defining Master-Detail Relationships”.

For recordsets in a master-detail relationship, the columns that are used to establish the relationship are always included in the recordsets. Specifically, the column identified by the **LinkMasterColumn** is always included in the master recordset, and the column identified by the **LinkDetailColumn** is always included in the detail recordset. If these columns are not bound to controls or specified in **RecSrcAddCols**, they are included as nondisplayed columns.

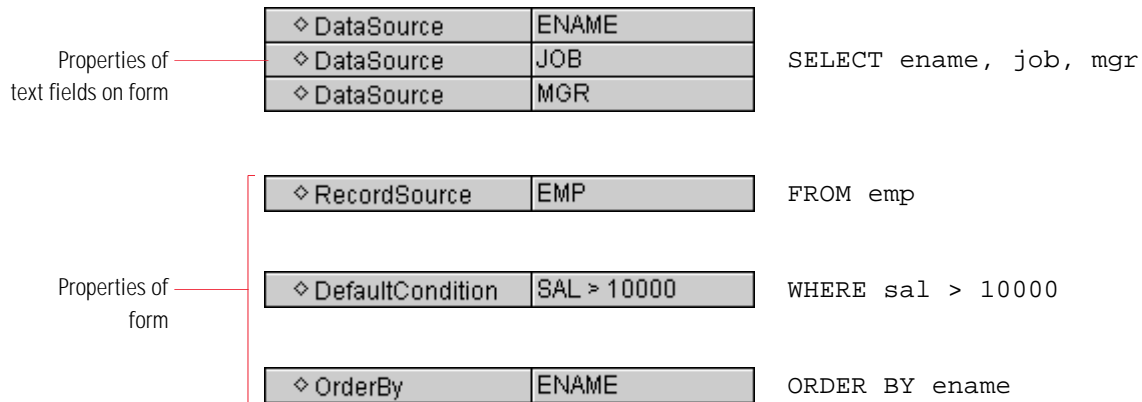
For some database types, Oracle Power Objects automatically adds one or more particular nondisplayed columns used internally for identification purposes. For example, a recordset bound to a table in an Oracle7 Server always has a column corresponding to the ROWID column of the table.

A recordset also has *unbound columns* corresponding to unbound controls within the container. Any unbound control whose **ScrollWithRow** property is set to True has a column in the recordset. However, the data in these columns are not sent to the database.

The rows in a recordset are a subset of the rows in the record source. The total set of rows that belong to the recordset are defined as the result set of a database query (a SQL SELECT statement). This query is derived from the following elements:

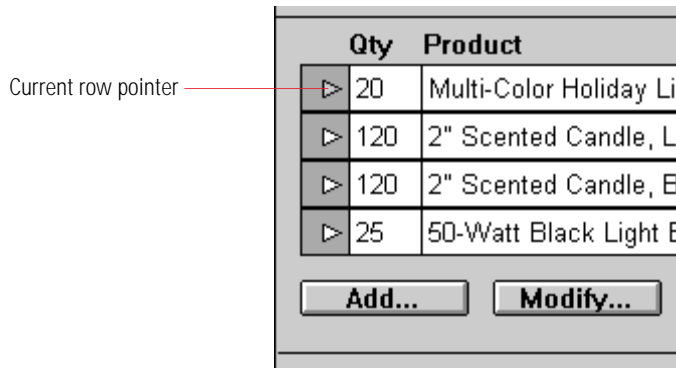
- The select list of the query is composed of all columns referenced in the **DataSource** properties of bound controls within the container, as well as any columns referenced in the **RecSrcAddCols** property of the container.
- The WHERE clause of the query can come from the **DefaultCondition** property of the container, or it can be explicitly specified as the argument to the **QueryWhere()** method. If the container is the detail container in a master-detail relationship, the WHERE clause also includes the conditions necessary to specify the join between master and detail containers.
- The ORDER BY clause of the query comes from the **OrderBy** property of the container.

The following diagram shows how a set of properties are translated into a query:



### The Current Row

A recordset also maintains the concept of a *current row*. The current row is always visible on screen. The values in the current row can be read and (if appropriate) modified, either by the user or by method code.



When the recordset is queried (for example, when it is first displayed), the current row is automatically set to the first row in the recordset. The current row can be changed in the following ways:

Scrollbar objects are described in the section "Scrollbars" on page 10.28.



- The user manipulates a scrollbar object associated with the recordset.
- In a repeater display, the user clicks on or begins editing a control in a different panel of the repeater display.
- A new row is inserted into the recordset, which can happen in the following ways:
  - The user clicks on the Insert Row button in the Form Run-Time toolbar
  - The user scrolls to the "blank row" at the end of the recordset and begins to enter values
  - The **InsertRow()** method of the container associated with the recordset is executed

As soon as the new row is inserted, the current row is set to the new row.



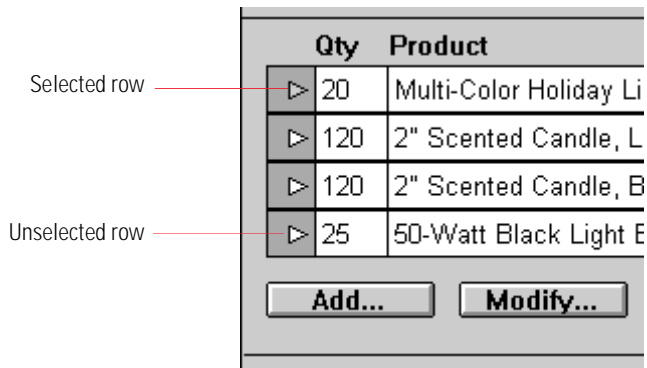
- A row is deleted from the recordset, which can happen in the following ways:
  - The user clicks on the Delete Row button in the Form Run-Time toolbar
  - The **DeleteRow()** method of the container associated with the recordset is executed

As soon as the row is deleted, the current row is set to the row immediately following the deleted row. When the last row in the recordset is deleted, the current row is set to the new last row.

- The **GoNextLine()**, **GoPrvLine()**, **GoNextPage()**, **GoPrvPage()**, or **GoPos()** method of the container associated with the recordset is executed.
- The **SetCurRow()** method of the recordset is executed.

---

The current row of a recordset can be indicated graphically with a *current row pointer* control located on the bound container. Current row pointer controls are particularly useful when multiple rows of a recordset are displayed on screen at the same time, as with the repeater display shown in the following diagram:



Current row pointers also indicate whether a data lock has been acquired on the row's data, as described in the section "How Locked Rows Are Indicated" on page 17.27.

### The Blank Row

When a recordset is displayed in a bound container on screen, the bound container by default displays a "blank row" after the last row in the recordset. This blank row appears only when the **HasExtraRow** property of the bound container is set to True.

Although the blank row behaves as though it is part of the recordset's data, it is not. The blank row provides a convenient way for a user to enter a new row of data into the recordset.

When the blank row is visible on the screen, the user can insert a new row simply by entering values into any of the container's controls (bound or unbound). To prevent the user from inserting new rows using the blank row, set the **HasExtraRow** property of the container to False.

### Fetching Rows from the Database

A recordset does not necessarily hold the entire result set at all times. The rows in the recordset need to be *fetched* from the database after the query has been executed. In other words, the query defines in the database the set of rows that will eventually be copied into the recordset, and a fetch actually retrieves rows from the database into the recordset. Oracle Power Objects always fetches rows automatically—you do not have to issue commands for rows to be fetched.



The rows in a recordset are fetched according to the setting of the **RowFetchMode** property of the container corresponding to the recordset. **RowFetchMode** can have the following settings:

**Fetch All Immediately**—All rows in the result set are fetched from the database as soon as a row is accessed. For large recordsets, this setting can impose a high memory overhead and affect performance significantly.

**Fetch As Needed**—A subset of rows is fetched when the recordset's data are first displayed or accessed. The subset displayed is 20 rows, or the least multiple of 20 required to show all rows on screen in the case of a repeater display. When an unfetched row is accessed (for example, the user scrolls to the row), additional rows are fetched in blocks of 20.

**Fetch Count First**—Behaves the same as “Fetch as Needed”, except a count of the rows in the final query result set is determined when the first block of rows is fetched. This is done by executing a separate SQL statement where “COUNT(\*)” is substituted for the query's select list. The “Fetch Count First” setting can be used to ensure the correct appearance of user interface components (such as scrollbars) that are tied to the number of rows in the query's result set.

Two recordset methods, **GetRowCount()** and **GetRowCountAdvice()**, enable you to determine the number of rows in a recordset.

The **GetRowCount()** method returns the number of rows currently in the recordset. When the **RowFetchMode** property of the corresponding container is set to “Fetch All Immediately”, **GetRowCount()** always returns an accurate count of the rows in the final query result set. When **RowFetchMode** is set to “Fetch as Needed” or “Fetch Count First”, **GetRowCount()** returns only the number of rows that have already been fetched.

The **GetRowCountAdvice()** method returns a count of the rows in the final query result set, using the same technique as the “Fetch Count First” setting of the **RowFetchMode** property. **GetRowCountAdvice()** can be called only for recordsets whose **RowFetchMode** property is set to “Fetch Count First.”

When **RowFetchMode** is set to “Fetch As Needed” or “Fetch Count First”, you can fetch recordset rows explicitly using the **FetchAllRows()** or **FetchToRow()** method.

The **FetchAllRows()** method fetches all database rows that have not yet been fetched. If all rows have already been fetched, **FetchAllRows()** has no effect.

The **FetchToRow()** method fetches all database rows up to and including a given row number, which is specified as the argument to **FetchToRow()**. If all rows up to the specified row have already been fetched, **FetchToRow()** has no effect.

---

## Storage of Recordset Rows

The rows in a recordset are stored primarily in memory. However, if the amount of data in the recordset exceeds a specified limit, additional rows are stored in a temporary file on the local hard disk. This behavior ensures that memory remains available for other recordsets and other operations that require memory.

The approximate amount of data stored in memory is determined by the **RecSrcMaxMem** property of the bound container associated with the recordset. By default, **RecSrcMaxMem** is set to the minimum value of 4 KB (data are written to disk as soon as the recordset occupies over 4 KB). Increasing the **RecSrcMaxMem** can improve the speed of scrolling through rows, because more rows are then stored in memory at one time. However, increasing **RecSrcMaxMem** reduces the amount of memory available to other components of the application.



**Tip:** Do not increase the **RecSrcMaxMem** property unless you are certain that all systems running your application will have enough memory to complete other operations.

Note that **RecSrcMaxMem** does not apply to data in columns of type LONG or LONG RAW. Data in these types of columns are always stored in temporary disk files.

## Accessing a Recordset Programmatically

A recordset object, like other types of objects, has methods that you can execute using Oracle Basic code. Because recordset objects do not exist at design time, you cannot examine or customize these characteristics using the Property sheet.

Recordset objects do not have **Name** properties. To execute a method of a recordset, you must obtain a handle to the recordset object by executing the **GetRecordset()** method of the bound container associated with the recordset.

For example, to execute the **InsertRow()** method of the recordset object associated with the form "Form1", you could execute the following method code:

```
Form1.GetRecordset().InsertRow()
```

Often, it is convenient to store the reference to the recordset in a variable of datatype *Object*. You can then use the variable to supply a reference to the recordset in subsequent method code. For example, the following method code stores a reference to the recordset associated with "Form1" in the variable `RecSet`:

```
DIM RecSet AS Object
RecSet = Form1.GetRecordset()
RecSet.DeleteRow()
Ename$ = RecSet.GetColVal("ENAME")
```

The rows and columns in a recordset are identified numerically. Rows and columns are numbered with integers beginning with 1.

	1	2	3
1	10	ACCOUNTING	NEW YORK
2	20	RESEARCH	DALLAS
3	30	SALES	CHICAGO
4	40	OPERATIONS	BOSTON

You can use these numeric identifiers to refer to specific rows and columns of the recordset. For example, the `SetCurRow()` method takes a row number as its argument, and the `SetColVal()` method takes a column number as its first argument.

The following method code includes a row number as the argument to the `SetCurRow()` method, which specifies the current row of the recordset:

```
RecSet.SetCurRow(10)
```

The following method code includes a column number as the first argument to the `SetColVal()` method, which sets the value of a specified column in the current row:

```
RecSet.SetColVal(2, "SMITH")
```

The `SetColVal()` method can take the column name as its first argument instead of the column number. However, using the column name takes longer for the system to execute. If you are going to use `SetColVal()` more than once for the same column in a method, you should execute the `GetColNum()` method first to speed program execution.

To determine the column number associated with a column name, you can execute the `GetColNum()` method. For example, the following method code returns the column number of the ENAME column into the variable `ColNum`:

```
Dim ColNum as Long
ColNum = RecSet.GetColNum("ENAME")
```

To determine the column name associated with a column number, you can execute the `GetColName()` method. For example, the following method code returns the column name of column 6 into the variable `ColName`:

```
Dim ColName as String
ColName = RecSet.GetColName(6)
```

To determine the row number of the current row, you can execute the `GetCurRow()` method. For example, the following method code returns the row number of the current row into the variable `RowNum`:

```
Dim RowNum as Long
RowNum = RecSet.GetCurRow()
```

---

To determine the column number associated with a given control object, you can execute the **GetBindColumn()** method of the control. For example, the following method code returns the column number of the column associated with the field "fldEmp" into the variable `ColNum`:

```
Dim ColNum as Long
ColNum = fldEmp.GetBindColumn()
```

Note that every control on a bound container has a recordset column unless the control's **ScrollWithRow** property is set to `False`. If the control is not associated with a recordset column, **GetBindColumn()** returns `Null`.

### Copying Values Between Recordsets

To copy a value directly from one recordset to another, you can execute the **CopyColFrom()** method. **CopyColFrom()** is particularly useful when copying long values (such as values in a `LONG` or `LONG RAW` column). The syntax of **CopyColFrom()** is:

```
Function CopyColFrom(dstCol as Variant, srcRec as Object,
    srcCol as Variant) as Integer
```

`dstCol` specifies the number or name of the destination recordset column (the column to receive the value). The value is copied into the current row of the recordset.

`srcRec` specifies the source recordset object (the object from which the value is to be copied).

`srcCol` specifies the number or name of the source recordset column. The value is read from the current row of the source recordset.

**CopyColFrom()** returns `True` if the value was copied successfully, `False` otherwise.

The following method code copies a value from the `ENAME` column of the "recEmp" recordset to the `EMPNAME` column of the "recEmpCopy" recordset:

```
vReturnVal = recEmpCopy.CopyColFrom("EMPNAME", recEmp,
    "ENAME")
```

### Transferring Data Between Recordsets and Files

The **WriteColToFile()** and **ReadColFromFile()** methods enable you to transfer data directly between recordset objects and operating system files. These methods are useful for long data (such as values in a `LONG` or `LONG RAW` column).

To write a value from a recordset into a file, you execute the **WriteColToFile()** method. The syntax of **WriteColToFile()** is:

```
Function WriteColToFile(col as Variant, fileNumber as
    Integer) as Integer
```

`col` specifies the number or name of the source column. The value is read from the current row of the recordset.

For information about opening files, see the topic "OPEN command" in the online help.

`FileNumber` specifies the number of the file to receive the value, which must be open in BINARY mode with write access.

**WriteColToFile()** returns True if the value was written successfully, False otherwise.

The following method code writes a value into the file "PictureFile" from the PICTURE column of a recordset bound to the EMP\_PICTURES table:

```
OPEN "PictureFile" FOR BINARY ACCESS WRITE AS 1
vReturnVal = recEmpPictures.WriteColToFile("PICTURE", 1)
CLOSE 1
```

To read a value into a recordset from a file, you execute the **ReadColFromFile()** method.

**ReadColFromFile()** can read a value of any datatype; however, the value must have been previously written into the file using **WriteColToFile()**. The syntax of **ReadColFromFile()** is:

```
Function ReadColFromFile(col as Variant, fileNumber
    as Integer) as Integer
```

`Col` specifies the number or name of the destination column. The value is read into the current row of the recordset.

For information about opening files, see the topic "OPEN command" in the online help.

`FileNumber` specifies the number of the source file, which must be open in BINARY mode with read access and positioned at the beginning of the data to be read.

**ReadColFromFile()** returns True if the value was read successfully, False otherwise.

➤ **Note:** You can read values of one datatype into a column of a different datatype; for example, you can read a numeric value into a string column. However, you cannot read a long value (a value from a LONG, LONG RAW, TEXT, or IMAGE column) into a non-long column; nor can you read a non-long value into a long column.

The following method code reads a value from the file "PictureFile" into the PICTURE column of a recordset bound to the EMP\_PICTURES table:

```
OPEN "PictureFile" FOR BINARY ACCESS READ AS 2
SEEK 2, 1
vReturnVal = recEmpPictures.ReadColFromFile("PICTURE", 2)
CLOSE 2
```

## Shared Recordsets

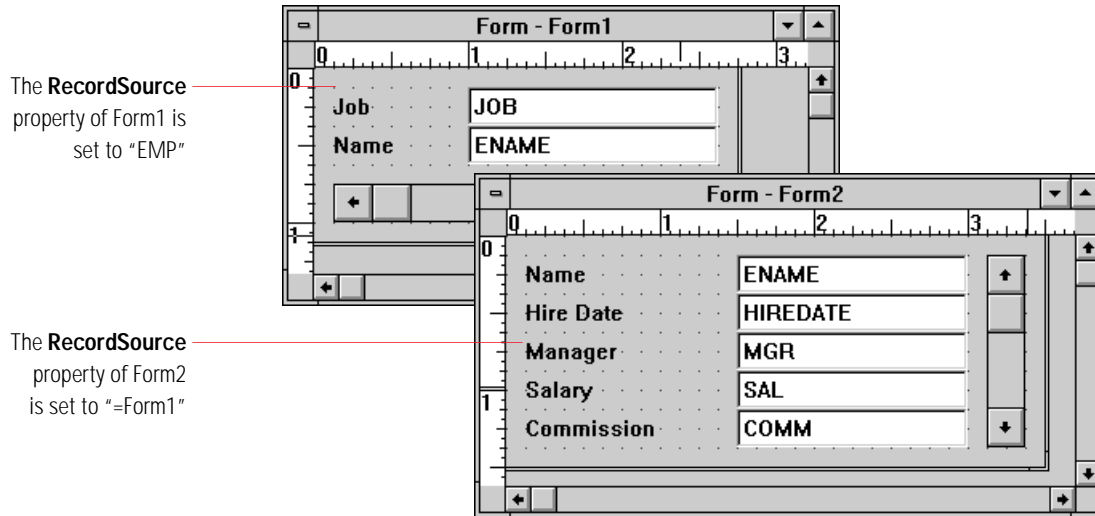
It is possible for two or more containers to be bound to the same recordset. This configuration is called a *shared recordset*. In a shared recordset, one container is bound to the record source in the normal way. Additional containers share the same recordset by using the following syntax in the **RecordSource** property:

```
=container_reference
```

Where *container\_reference* is a reference to the container object whose recordset is to be shared. For example, to cause a recordset to be shared by a form “Form1” and a repeater display “Repeater1”, you could set the **RecordSource** property of “Repeater1” to the following value:

=Form1

In a shared recordset, the columns of the recordset are determined by the controls and **RecSrcAddCols** properties of both containers.



The columns of the shared recordset are determined by the controls on both forms.

ENAME	JOB	MGR	HIREDATE	SAL	COMM
SMITH	CLERK	7902	12/17/80	800.0	
ALLEN	SALESMAN	7698	2/20/81	1600.0	300.0
WARD	SALESMAN	7698	2/22/81	1250.0	500.0
JONES	MANAGER	7839	4/2/81	2975.0	
MARTIN	SALESMAN	7698	9/28/81	1250.0	1400.0
BLAKE	MANAGER	7839	5/1/81	2850.0	
CLARK	MANAGER	7839	6/9/81	2450.0	
SCOTT	ANALYST	7566	12/9/82	3000.0	
KING	PRESIDENT		11/17/81	5000.0	
TURNER	SALESMAN	7698	9/8/81	1500.0	0.0
ADAMS	CLERK	7788	1/12/83	1100.0	

For more information on validation methods, see Chapter 19, “Using Constraints to Enforce Business Rules”.

Changes can be made to the recordset from any of the containers sharing the recordset—for example, the recordset’s current row can be changed, or the values in the recordset can be modified. Any such changes are reflected immediately in both bound containers, not just the one being edited. Note that changes made to a recordset will trigger the validation methods only of the container where the changes are made.

### When to Use a Shared Recordset

You should use a shared recordset whenever two containers display information from the same record source. For example, your application might include a repeater display with a list of employees and a window that shows employee details. Since both containers are bound to the EMP table, using a shared recordset ensures that their contents always match. Performance is also improved because the same database operations apply to both containers.

A shared recordset can be useful when the record source has too many columns to fit on a single container. In this case, you can use two forms with shared recordsets, each of which displays a portion of the columns from the record source.

➤ **Note:** You cannot use a shared recordset in the **RecordSource** property of a form.

### Requerying a Recordset

Because a recordset is a local copy of information stored in a database, it is possible for the information in the recordset to become out of date. For example, other users could make changes to the record source after the recordset was originally queried.

To keep the recordset's information current, the database is periodically requeried. This requerying happens in two ways: automatically and manually.

#### Automatic Requerying

The contents of a recordset are automatically requeried in the following circumstances:

- The bound container is closed and reopened.
- The current row of a master recordset changes (all detail recordsets are automatically requeried)

#### Manual Requerying

You can explicitly requery a recordset's contents in the following ways:

- Execute the **Query()** method of the bound container associated with the recordset.
- Execute the **QueryWhere()** method of the bound container to apply a specified condition to the query.
- Modify the **OrderBy** property of the bound container.
- Click on the **Query** button in the Form Run-Time toolbar.
- Use **Query By Form** to apply a query condition.



For information on Query By Form, see the section "Query by Form" on page 11.14.

When a recordset is requeried, the SELECT statement that determines the recordset's contents is re-executed, and rows are fetched from the database according to the setting of the **RowFetchMode** property.

---

Note that when a recordset is requested, the current record is automatically reset to the first row in the recordset. If your application needs to maintain the current record when requesting, you must store an identifier for the current row before the query is executed, then afterwards scroll the recordset to the row.

The following example shows how you might maintain the current record for a form bound to the EMP table. This method code appears in the `Query()` method of the form. Whenever the form is requested, the method code in the `Query()` method is executed and scrolls the new recordset to the row previously displayed.

In this case, the EMP table's primary key column (EMPNO) is used to supply the unique identifier. To use this code with a different record source, you would have to use a key value appropriate to that table or view.

```
Sub Query()  
DIM Empno AS long      'The employee number of the current row  
DIM OldRow AS long     'Row number of previous row in loop  
DIM RecSet AS object   'The form's recordset  
  
'Get the form's recordset  
RecSet = Self.GetRecordset()  
  
'Get the value of EMPNO for the current row  
Empno = VAL(RecSet.GetColVal("EMPNO"))  
  
'Execute the default processing of the Query() method  
Inherited.Query()  
  
'Scroll the new recordset to the old current row  
DO WHILE VAL(RecSet.GetColVal("EMPNO")) <> Empno  
    OldRow = RecSet.GetCurRow()  
    Self.GoNextLine()  
  
'If we have reached the last record in the recordset,  
'exit the DO loop  
IF RecSet.GetCurRow() = OldRow THEN EXIT DO  
  
LOOP
```



## Making Changes to Data in a Recordset

The data in a recordset can be changed in the following ways:

- By editing existing data in bound controls
- By inserting or deleting rows

These techniques are described below.

### Editing Data in Bound Controls

A user can make changes to most types of bound controls if the control is visible and enabled. For example, the user can:

- Type in a field or combo box
- Choose a value from a list box, popup list, or combo box
- Cut a picture from or paste a picture into a picture control
- Select a radio button in a radio button group
- Click on a check box

When the user manipulates data in a bound control, your application can validate the changes before they are written to the recordset.

You can also make changes to the values in controls programmatically by setting the **Value** property of the control. For example, you could set the value of a field object "Field1" with the following statement:

```
Field1.Value = "Green"
```

Programmatic changes to the **Value** property are subject to the same validation techniques as changes made by a user to the control.

You can also modify the data in a recordset by executing the **SetColVal()** method of the recordset to set the value of one column of the current row. Note that when you make changes directly to the recordset object, the validation methods are *not* triggered.

### Inserting or Deleting Rows

The user can insert or delete recordset rows using toolbar buttons or scrollbar controls. For example, the user can:



- Click the Insert Row button in the Form Run-Time toolbar to insert a row
- Click the Delete Row button in the Form Run-Time toolbar to delete a row
- Scroll to the "blank row" at the end of the recordset and begin to enter values to insert a new row

When the user applies these techniques to insert or delete a row, your application can validate the row before it is flushed to the database.

For information about validating changes to data, see Chapter 19, "Using Constraints to Enforce Business Rules".

---

You can insert or delete rows programmatically by executing methods of the recordset object or of the associated container. For example, you can:

- Execute the **InsertRow()** method of the container or recordset to insert a new row
- Execute the **DeleteRow()** method of the container or recordset to delete the current row

When you execute methods of the bound container, your application can validate the row before it is sent to the database. However, when you make changes directly to the recordset object, the validation methods are *not* triggered.

### How Recordset Changes Are Sent to the Database

The changes you make to a recordset are not always sent to a database immediately. Instead, they are stored in memory and periodically *flushed* (sent) to the database. This provides the following advantages:

- **Reduces network traffic** by sending related changes together, rather than individually. Network traffic can be significantly reduced by using deferred flushing, as described below.
- **Avoids database errors** by not sending incomplete operations to the database. For example, a newly inserted row should not be sent to the database until the user has entered values for all of the mandatory (NOT NULL) columns.

When changes are flushed to the database, the changes you have made to the recordset are translated into SQL operations (INSERT, UPDATE, DELETE) that are sent to the database engine for execution. The fact that changes have been sent to the database does not mean that the changes have been made permanent—you must explicitly commit your changes to save them, or roll them back to remove them.

When an update to a row is flushed to the database, *all* values in that row are flushed, including values that have not been modified (except values from LONG or LONG RAW columns, which are flushed only if they have changed). If the information in the recordset is out of date, this can cause information entered by another user to be overwritten with older values. This problem can be avoided by setting the **CompareOnLock** property of the bound controls in your container to True, as described in the section “How Oracle Power Objects Acquires Locks” on page 17.26.

➤ **Technical Note:** Oracle Power Objects flushes all column values at the same time for performance reasons. In databases that support shared cursors (such as Oracle7 Servers), the database can use the same UPDATE statement for each row in the recordset and each concurrent user without reparsing.

Oracle Power Objects provides two transaction models for determining when changes you make to a recordset are sent to the database: immediate flushing and deferred flushing.

**Immediate Flushing** — changes are sent to the database as soon as the row is no longer current, which can happen in any of the following ways:

- The user scrolls to another row in the recordset
- In a repeater display, the user clicks on another row in the recordset

- The recordset is requiered (either automatically or manually)
- The user commits changes
- The focus moves to a container associated with a different recordset

Changes are not sent in the following circumstances:

- The user cancels the editing of a row
- The **RevertRow()** method is called
- The user rolls back changes

Immediate flushing is used by default.

**Deferred Flushing** — changes are sent to the database immediately prior to a Commit operation, which can happen in any of the following ways:



- The user presses the Commit button
- The **CommitForm()** method is called
- The **CommitWork()** method of the session is called

When deferred flushing is enabled, it is possible that some changes cannot be flushed to the database successfully. If changes cannot be flushed, Oracle Power Objects returns an error.

In this release of Oracle Power Objects, you cannot set the transaction model for container recordsets. However, you can set the transaction model for standalone recordsets, as described in the section “Standalone Recordsets” on page 17.27.

### Data Consistency and Locking

If the database objects associated with the recordset are accessible to other users, it is important that the consistency of the data in the recordset be maintained. Maintaining data consistency has several components:

- Ensure that the user does not try to edit a row that has been deleted by another user
- Ensure that other users cannot modify a row that the user has edited but not committed changes
- Ensure that the user does not try to edit a row that has been modified after the recordset was queried from the database

The primary means of ensuring data consistency in Oracle Power Objects is to acquire a *data lock* on information that the user edits.

Relational databases generally lock a row automatically when the row is updated. However, Oracle Power Objects acquires the data lock separately from the update operation because changes are not sent to the database immediately after the user has made them. In the case of immediate flushing, changes are not sent until the user has scrolled to a different row. In the case of deferred flushing, changes are not sent until the user attempts to commit the transaction.

Oracle Power Objects provides two locking models that determine when a data lock is acquired: pessimistic locking and optimistic locking.

---

**Pessimistic Locking** — a data lock is acquired as soon as the user begins to change data in a row but before any changes are allowed. If the lock cannot be acquired, user changes are not allowed. Pessimistic locking is used by default.

**Optimistic Locking** — data locks are not acquired until the user is about to commit changes to the database. Optimistic locking is normally used in combination with deferred flushing to reduce network traffic and to minimize access to heavily used shared resources.

No matter which locking model is currently used, you can explicitly request a lock on a row of data by executing the **LockRow()** method of the recordset.

In this release of Oracle Power Objects, you cannot set the locking model for container recordsets. However, you can set the locking model for standalone recordsets, as described in the section “Standalone Recordsets” on page 17.27.

### **How Oracle Power Objects Acquires Locks**

Oracle Power Objects attempts to acquire the least restrictive lock possible when locking a row. For databases that support row-level locking (such as Blaze databases and Oracle7 Servers), Oracle Power Objects acquires an exclusive row lock. For databases that do not support row-level locking (such as Microsoft SQL Server or Sybase SQL Server), Oracle Power Objects acquires an exclusive page or table lock. In either case, the lock is released only when the transaction is committed or rolled back.

Before locking a row, Oracle Power Objects attempts to verify that the row in the recordset is identical to the row in the database. To do so, Oracle Power Objects checks the contents of each bound control whose **CompareOnLock** property is set to True. Nondisplayed columns of the recordset are checked automatically.

If the contents of a row differ between the recordset and the database, Oracle Power Objects does not attempt to acquire the lock and displays an error dialog box to the user.

Normally, the **CompareOnLock** property should be set to True for most types of controls. However, if a read-only control displays a large amount of data (such as a picture control) and will not change very often, you can enhance performance by setting **CompareOnLock** to False.

### How Locked Rows Are Indicated

The recordset maintains information about which rows in the recordset have been locked. This information can be indicated graphically with a current row pointer control placed on the bound container. The current row pointer control displays a lock icon for each locked row.



For more information about current row pointer controls, see the section “Current Row Pointers” on page 10.23

You can also determine programmatically the status of the current row of a recordset by calling the `GetRowStat()` method of the container associated with the recordset. `GetRowStat()` returns an integer with one of the following values:

Value	Meaning
0	The row is unlocked.
1	The row is newly inserted into the recordset, and the insertion has not been completed.
2	The row is locked but unchanged.
3	The row is locked and has been changed. Changes have not yet been flushed to the database.
4	The row is locked and has been changed. Changes have been flushed to the database.



### Standalone Recordsets

A *standalone recordset*, unlike the *container recordsets* described earlier in this chapter, is not associated with any application object. You create a standalone recordset through Oracle Basic method code. You can use standalone recordsets to store data that the user does not need to view or edit directly.

---

There are two types of standalone recordsets: *bound* and *unbound*.

A **bound** recordset is linked to a record source table or view.

An **unbound** recordset is not linked to a record source. Oracle Power Objects does not automatically transfer data between an unbound recordset and a database; you must perform any such data transfers manually.

Note that a container recordset is automatically a *bound* recordset, as is the recordset associated with a list box or popup list control whose **TranslationList** property has been set to a database query, or a combo box control whose **ValueList** property has been set to a database query. You can create an *unbound* recordset by defining a standalone recordset. An unbound recordset is also associated with a list box, popup list, or combo box that contains a list of constants in the **TranslationList** or **ValueList** property (rather than a query).

You create a standalone recordset using the Oracle Basic NEW operator. The NEW operator returns a reference to the newly created recordset, which you can store in a variable or property of datatype *Object*.

### Creating an Unbound Recordset

To create an unbound recordset, you use the NEW Recordset syntax, which is:

```
NEW RECORDSET( [inMemory [, maxMem]])
```

*inMemory* is a Boolean value (a Long Integer) specifying whether the recordset must always be stored in memory. If *inMemory* is True, the recordset's data are always stored entirely in memory; if *inMemory* is False, the data may be swapped to disk when the recordset exceeds the maximum size. *inMemory* is False by default.

*maxMem* is a Long Integer value specifying the maximum amount of data (in KB) that is stored in memory when *inMemory* is False. If *inMemory* is True, *maxMem* is ignored. *maxMem* is 4 KB by default; it cannot be set to less than 4 KB.

The following method code creates an unbound recordset that must always be stored in memory:

```
DIM recRecordset1 AS Object  
recRecordset1 = NEW Recordset(True)
```

The following method code creates an unbound recordset that is swapped to disk when its size exceeds 10 KB:

```
DIM recRecordset2 AS Object  
recRecordset2 = NEW Recordset(False, 10)
```

Once you have created the recordset, you add columns to it by executing the **AddColumn()** method of the recordset (**AddColumn()** applies only to unbound recordsets).

The syntax of `AddColumn()` is:

```
SUB AddColumn(colname AS String, datatype AS Long)
```

`colname` is the name of the column, which you can use to refer to the column in some methods.

`datatype` is an integer identifying the column datatype. You generally specify `datatype` using a predefined symbolic constant. The following datatype constants are defined:

<b>Constant</b>	<b>Meaning</b>
<code>RecDty_String</code>	<i>String</i>
<code>RecDty_Integer</code>	<i>Integer</i>
<code>RecDty_Long</code>	<i>Long Integer</i>
<code>RecDty_Double</code>	<i>Double</i>
<code>RecDty_Date</code>	<i>Date</i>

The following method code adds a "Name" column and a "Salary" column to the recordset object "recEmployee":

```
DIM recEmployee AS Object
recEmployee = NEW Recordset(False)
recEmployee.AddColumn("Name", RecDty_String)
recEmployee.AddColumn("Salary", RecDty_Double)
```

You cannot add columns to a recordset containing data. Therefore, you should call `AddColumn()` immediately after creating a new recordset.

### Creating a Bound Recordset

To create a bound recordset, you use the `NEW DBRecordset` syntax, which is:

```
NEW DBRECORDSET(session [, deferred [, optimistic
    [, inMemory [, maxMem]]])
```

`session` is a reference to the database session object containing the record source.

`deferred` is a Boolean value (a Long Integer) specifying whether deferred flushing is used. If `deferred` is True, deferred flushing is used; if `deferred` is False, immediate flushing is used. `deferred` is False by default. Deferred flushing is described in the section "Data Consistency and Locking" on page 17.25.

`optimistic` is a Boolean value (a Long Integer) specifying whether optimistic locking is used. If `optimistic` is True, optimistic locking is used; if `optimistic` is False, pessimistic locking is used. `optimistic` is False by default. Optimistic locking is described in the section "Data Consistency and Locking" on page 17.25.

---

`inMemory` is a Boolean value (a Long Integer) specifying whether the recordset must always be stored in memory. If `inMemory` is True, the recordset's data are always stored entirely in memory; if `inMemory` is False, the data may be swapped to disk when the recordset exceeds the maximum size. `inMemory` is False by default.

`maxMem` is a Long Integer value specifying the maximum amount of data (in KB) that is stored in memory when `inMemory` is False. If `inMemory` is True, `maxMem` is ignored. `maxMem` is 4 KB by default; it cannot be set to less than 4 KB.

The following method code creates a recordset object associated with the database session object "sesOracle":

```
DIM recRecordset3 AS Object
recRecordset3 = NEW DBRecordset(sesOracle)
```

The following method code creates a recordset object associated with the database session object "sesOracle", specifying pessimistic locking and deferred flushing:

```
DIM recRecordset4 AS Object
recRecordset4 = NEW DBRecordset(sesOracle, FALSE, TRUE)
```

After creating the recordset, you bind it to an object in the associated session by executing the `SetQuery()` method of the recordset object. The syntax of `SetQuery()` is:

```
SUB SetQuery(query AS String, updatable AS Long)
```

`query` is a SQL SELECT statement that determines how the recordset's rows are derived from the record source.

`updatable` indicates whether you can make changes to the recordset. If `updatable` is False, the recordset is read-only. To set `updatable` to True, the query you specify must meet the following criteria:

- It must select values only from a single record source
- If the record source is a view, the view must be a read-write view
- All columns with Primary Key or Not Null constraints must be included in the select list of the query
- The select list of the query cannot be an asterisk (\*), which represents all columns in the record source in some databases

The following method code creates a recordset bound to the EMP table, containing columns for the EMPNO and ENAME columns:

```
DIM recEmp AS Object
recEmp = NEW DBRecordset(sesOracle)
recEmp.SetQuery("SELECT empno, ename FROM emp", TRUE)
```

The SELECT statement can use any syntax supported by the database containing the record source. For example, it can contain clauses such as WHERE and ORDER BY, as in the following example:

```
recEmp.SetQuery("SELECT empno, ename FROM emp WHERE " &&
"job = 'CLERK' ORDER BY sal", TRUE)
```



For a read-only recordset, you can include multiple tables in the select list, as in the following example:

```
recEmpDept.SetQuery("SELECT emp.ename, dept.dname FROM " &&
    "emp, dept WHERE emp.deptno = dept.deptno", FALSE)
```

### Deleting a Recordset

You delete a recordset using the Oracle Basic DELETE command. You specify the object to delete as the argument to DELETE. For example, the following method code deletes the recordset "recEmp":

```
DELETE recEmp
```

You can delete more than one item in the same statement by specifying multiple objects separated by commas, as in the following example:

```
DELETE recEmp, recDept
```

## Properties and Methods of Recordsets and Bindable Objects

This section provides a quick reference to object properties and methods related to recordsets and binding. For syntax declarations and complete descriptions of these properties and methods, see the online help.

### Methods of Recordset Objects

Method	Description
CopyColFrom()	Copies a value into a specified column of the recordset from a specified column of another recordset. The value in the current row of the source recordset is copied; the current row of the destination recordset receives the value. <b>CopyColFrom()</b> returns True if successful, False otherwise.
DeleteRow()	Deletes the current row from the recordset.
FetchAllRows()	Fetches all rows that have not yet been fetched.
FetchToRow()	Fetches all rows up to and including a specified row number.
GetColCount()	Returns the number of columns in the recordset, including unbound and system columns.

---

<b>Method</b>	<b>Description</b>
<b>GetColName()</b>	Returns the name of a column in a recordset, specified by column number.
<b>GetColNum()</b>	Returns the number of a column in the recordset, specified by column name.
<b>GetCurRow()</b>	Returns the row number of the current row in the recordset.
<b>GetRowCount()</b>	Returns the number of rows in the recordset.
<b>GetRowCountAdvice()</b>	For recordsets that use incremental fetching, returns the estimated number of rows that will be in the recordset when all rows are fetched. Applies only when the <b>RowFetchMode</b> property of the bound container associated with the recordset is set to "Fetch Count First".
<b>GetSession()</b>	Returns the session object associated with the recordset.
<b>InsertRow()</b>	Inserts a new row into the recordset before the position of the current row.
<b>LockRow()</b>	Attempts to acquire a data lock on the current row.
<b>ReadColFromFile()</b>	Reads a value into the recordset from a file.
<b>SetColVal()</b>	Assigns a value to the specified column for the current row in the recordset.
<b>SetCurRow()</b>	Changes the current row to the specified row number in the recordset.
<b>WriteColToFile()</b>	Writes a value from the recordset into a file.

### **Recordset-Related Properties of Containers**

<b>Property</b>	<b>Description</b>
<b>DefaultCondition</b>	A SQL condition used to restrict the rows in the recordset. The <b>DefaultCondition</b> is used whenever the recordset is requested, except when requested using the <b>QueryWhere()</b> method.
<b>HasExtraRow</b>	Determines whether a "blank row" is displayed after the last row in the recordset. The blank row does not belong to the recordset—it simply provides a convenient way for the user to enter a new row of data into the recordset.

<b>Property</b>	<b>Description</b>
<b>OrderBy</b>	The name of the column or columns by which the rows in the recordset should be ordered. Rows are retrieved in ascending order by default; you can specify descending order by including the word "DESC" after the column names.
<b>RecordSource</b>	The name of the record source table or view.
<b>RecSrcAddCols</b>	The names of additional database columns associated with the container but not associated with controls in the container. Multiple columns are separated by commas.
<b>RecSrcMaxMem</b>	The maximum amount of record source data to be stored in local memory before temporary disk files are created.
<b>RecSrcSession</b>	The database session "containing" the record source.
<b>RowFetchMode</b>	Determines how rows are fetched from the database into the recordset when the recordset is requeried.

**Recordset-Related Methods of Containers**

<b>Method</b>	<b>Description</b>
<b>CommitForm()</b>	Commits the current transaction for all bound containers located on the same form as the container for which <b>CommitForm()</b> is called.
<b>DeleteRow()</b>	Deletes the current row from the container's recordset.
<b>GetRecordset()</b>	Returns a handle to the recordset object associated with the container.
<b>GetRowStat()</b>	Indicates the status of the current row of the recordset (whether the row is locked and whether any of the row's data have been changed).
<b>GoNxtLine()</b>	Moves the current row of the container's recordset forward one row.
<b>GoNxtPage()</b>	Moves the current row of the container's recordset forward one "page" of rows. A page is the number of rows displayed on the container at one time. For containers such as forms, embedded forms, and user-defined classes, a page is one row.
<b>GoPos()</b>	Moves the current row of the container's recordset to a numerically specified row.
<b>GoPrvLine()</b>	Moves the current row of the container's recordset back one row.
<b>GoPrvPage()</b>	Moves the current row of the container's recordset back one "page" of rows.

---

<b>Method</b>	<b>Description</b>
<b>InsertRow()</b>	Inserts a new row into the container's recordset before the position of the current row.
<b>OnQuery()</b>	Called after the container's recordset is requeried.
<b>Query()</b>	Requeries the container's recordset.
<b>QueryWhere()</b>	Requeries the container's recordset with a specified SQL condition.
<b>RevertRow()</b>	Discards changes made to the current row in the container's recordset. Only changes made since the row most recently became the current row are discarded.
<b>RollbackForm()</b>	Rolls back the current transaction for all bound containers located on the same form as the container for which <b>RollbackForm()</b> is called.

#### **Recordset-Related Properties of Controls**

<b>Property</b>	<b>Description</b>
<b>CompareOnLock</b>	Determines whether Oracle Power Objects checks the control's value against the database value it represents before locking the currently displayed row.
<b>DataSource</b>	The column of the record source table or view from which the control's value is derived.
<b>ScrollWithRow</b>	For unbound controls, determines whether the control has a separate value for each row of the recordset associated with the control's container. If <b>ScrollWithRow</b> is True, the control forms a "hidden" column of the recordset.
<b>Value</b>	The value of the control. If the control is bound to the database, the control's <b>Value</b> property is derived from the <b>DataSource</b> column of the record source table or view.

#### **Recordset-Related Methods of Controls**

<b>Method</b>	<b>Description</b>
<b>GetBindColumn()</b>	Returns the number of the recordset column associated with the control.
<b>RevertValue()</b>	Discards changes made to the control that currently has the focus. Only changes made since the control's value was last set are discarded.

# 18

---

## Defining Master-Detail Relationships

This chapter covers the following topics:

Overview .....	18.2
Defining Master and Detail Containers .....	18.4
Integrity Checks .....	18.6
Options for Displaying Master and Detail Records .....	18.8

---

## Overview

Oracle Power Objects is designed to simplify the task of creating *master-detail relationships* among records. In this kind of relationship, rows in one recordset (details) are associated with rows in another recordset (a master), so that rows appearing in the detail recordset only appear when there are matching rows in the master recordset. To make a master-detail relationship work, columns in the master and detail recordsets must contain matching values.

Master-detail relationships are also known as *one-to-many relationships*, although some master records can have as few as one associated detail record, or none.

These relationships are not defined within the database. On the contrary, a properly *normalized* relational database does not group records in this fashion, to avoid limiting your flexibility in relating information from different tables and views.

## Master-Detail Relationships and Joins

When the application queries master and detail records, the query is called a *join*. For the join to work, the values in the column of one record source must match some values in a column in the other record source. The column in the master recordset used for the join is the *primary key*, the corresponding column in the detail recordset is the *foreign key*.

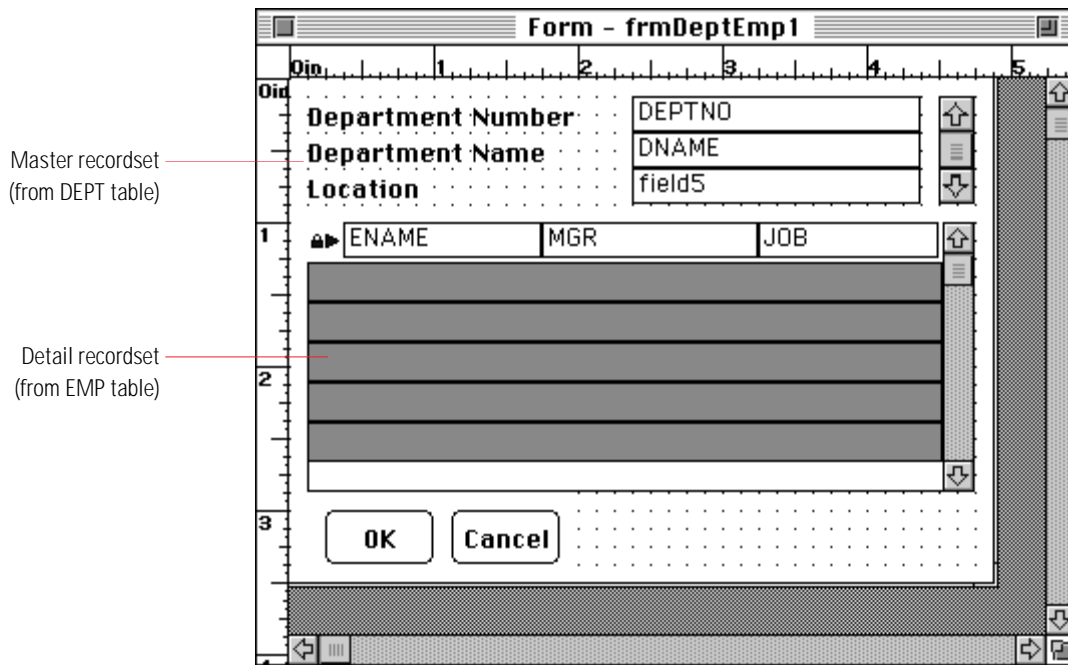
For example, the DEPT table has the column DEPTNO, as does the EMP table. To query employee records by department, you would use DEPT as the master table, and EMP as the detail table. DEPTNO would be the primary key in DEPT and the foreign key in EMP.

You can specify a join as part of an SQL query. For information on the syntax needed to perform SQL queries, see Chapter 9, “Structured Query Language (SQL)”, as well as the online help topics covering the SQL Language. Oracle Power Objects is designed to simplify this task by automating many aspects of the master-detail join.

## Automated Joins in Oracle Power Objects

When you write a SQL query, you must carefully enter the conditions for the join to ensure that the query returns a set of master and detail records associated in the way you want to view them. In many development environments, you would then write code to maintain the relationship, so that detail records are always associated with their master records.

In Oracle Power Objects, you do not have to write any SQL code to create a join; instead, to create the master-detail relationship, you set the properties of the containers holding master and detail records. Behind the scenes, Oracle Power Objects builds a SQL query that specifies the join, and then queries the master and detail records. These records are kept in separate recordsets, associated with the containers designed to display master and detail records.



The primary key and foreign keys are defined by different properties of the container displaying detail records, as are most other aspects of the join. As with other properties, you set the properties relevant to master-detail relationships through the Property sheet. For more information, see the section "Defining Master and Detail Containers" on page 18.4.

## Referential Integrity

Maintaining the logical relationship between master and detail records is known as *referential integrity*. There are many ways of enforcing referential integrity, and Oracle Power Objects gives you the ability to define these different degrees of referential integrity when you define master and detail containers.

For example, you can instruct Oracle Power Objects to prevent the user from deleting a master record if it has any detail records associated with it. In the case of the DEPT and EMP recordsets, you can prevent the user from deleting a department record as long as employees are assigned to the

---

department. Alternatively, you could immediately delete all employee records when the user deletes the department record. To establish these types of referential integrity, you set a property on the container displaying the detail records; Oracle Power Objects then takes care of the work of enforcing this relationship at run time.

## Defining Master and Detail Containers

After you define the record source (a table or view) for the containers displaying master and detail records, you then set four properties to relate these recordsets. These properties are always set on the container displaying the detail records, whether the container is part of the form displaying master records, or is a separate form altogether.

The following table summarizes the properties needed to establish a master-detail relationship.

<b>Property</b>	<b>Description</b>
<b>LinkDetailColumn</b>	Defines the foreign key for the master-detail relationship (that is, the column in the detail recordset used for the join).
<b>LinkMasterColumn</b>	Defines the primary key for the master-detail relationship (that is, the column in the master recordset used for the join).
<b>LinkMasterForm</b>	Identifies the container displaying master records. The container can be part of the same form displaying detail records, or it can be a separate form altogether.
<b>LinkPrimaryKey</b>	Identifies the container whose recordset contains the primary key for the master-detail relationship. For more information on primary keys, see the section “Setting the Primary Key” on page 18.6.



For example, the “Departments” form shown here displays records from the DEPT and EMP tables. The record of each department is displayed in the main form; employee records associated with each department are shown in a repeater display. Therefore, the container holding the master records is the form, and the detail records are displayed in the repeater display.

ENAME	JOB	MGR	SAL
JONES	MANAGER	7839	\$2,975.00
SCOTT	ANALYST	7566	\$3,000.00
FORD	ANALYST	7566	\$3,000.00
ADAMS	CLERK	7788	\$1,100.00
SMITH	CLERK	7902	\$800.00

Both the DEPT and EMP tables have DEPTNO columns in them, holding the department number for each department and employee. Therefore, to establish the master-detail relationship, the following properties are set on the *repeater display*.

Property	Value
LinkDetailColumn	DEPTNO
LinkMasterColumn	DEPTNO
LinkMasterForm	Departments
LinkPrimaryKey	On Master
RecordSource	EMP

In addition, the **RecordSource** property of the *form* is set to DEPT.

You can use any bound containers to establish a master-detail relationship, as long as it is possible to create a join between the container’s recordsets. For more information, see the section “Other Options for Displaying Master-Detail Relationships” on page 18.11.

---

## Setting the Primary Key

The **LinkPrimaryKey** property identifies the container in which the primary key is set. As with other properties relevant to master-detail relationships, you define this property for the container displaying detail records.

Most commonly, you will set the primary key in the container displaying master records. However, occasionally you may want to define the primary key value in the container displaying detail records.

The **LinkPrimaryKey** property has two settings:

<b>Value</b>	<b>Description</b>
<i>Here (on detail)</i>	The control holding primary key values appears in the container displaying detail records.
<i>On Master</i>	The control holding primary key values appears in the container displaying master records, as designated through the <b>LinkMasterForm</b> property.

## Integrity Checks

Referential integrity is the degree to which the application maintains the logical connection between master and detail records. For example, to ensure that line items in an invoice are never left in the system after the invoice itself is deleted, you must establish some kind of referential integrity between the detail records (the line items) and the master record (the invoice).

When detail records are no longer associated with master records, they are *orphaned*. For example, if you delete an invoice, but leave the line item records in the database, the line items become orphaned.

Two further properties of the detail container, **LinkMasterUpd** and **LinkMasterDel**, determine the degree of referential integrity enforced when the user modifies or deletes master records.

### LinkMasterUpd Property

The **LinkMasterUpd** property enforces referential integrity when the user modifies the primary key for a master record. You set this property on the container displaying detail records.

When the primary key changes, there is a risk that the foreign key value for the detail records will no longer match it. Therefore, the **LinkMasterUpd** property has the following settings:

<b>Setting</b>	<b>Description</b>
<i>Refuse When Children Present</i>	Prevents the user from changing the primary key value as long as associated detail records exist.

<b>Setting</b>	<b>Description</b>
<i>Update Cascade</i>	Modifies the foreign key value for all detail records before flushing changes to the database.
<i>Orphan Details</i>	Allows the detail records to be orphaned by not changing their foreign key values.

The application enforces the type of referential integrity defined through the **LinkMasterUpd** property when the user attempts to change the primary key value.

### LinkMasterDel Property

The **LinkMasterDel** property enforces referential integrity when the user tries to delete a master record. You set this property on the container displaying detail records, and it has the following settings:

<b>Setting</b>	<b>Description</b>
<i>Refuse When Children Present</i>	Prevents the user from deleting the master record as long as associated detail records exist.
<i>Delete Cascade</i>	Deletes all detail records associated with the master record.
<i>Orphan Details</i>	Allows the detail records to be orphaned when the user deletes the master record.

The application enforces the type of referential integrity defined through **LinkMasterDel** when the user attempts to delete a record.

---

## Options for Displaying Master and Detail Records

Oracle Power Objects affords great flexibility in defining the containers displaying master and detail records. These containers can be part of the same form, as in the form shown below. In this case, the master records from DEPT appear on the form, while the detail records from EMP appear in the repeater display.

The screenshot shows a form window titled "Departments". At the top, there are three input fields: "DEPTNO" with the value "10", "DNAME" with the value "ACCOUNTING", and "LOC" with the value "NEW YORK". Below these fields is a horizontal separator with left and right arrow buttons. Underneath is a table with three columns and four rows. The first three rows contain data: (CLARK, MANAGER, \$2,450.00), (KING, PRESIDENT, \$5,000.00), and (MILLER, CLERK, \$1,300.00). The fourth row is empty. To the right of the table is a vertical scrollbar with up and down arrow buttons.

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK

CLARK	MANAGER	\$2,450.00
KING	PRESIDENT	\$5,000.00
MILLER	CLERK	\$1,300.00

Alternatively, the master records from DEPT could appear in the repeater display, while the detail records could appear on the form:

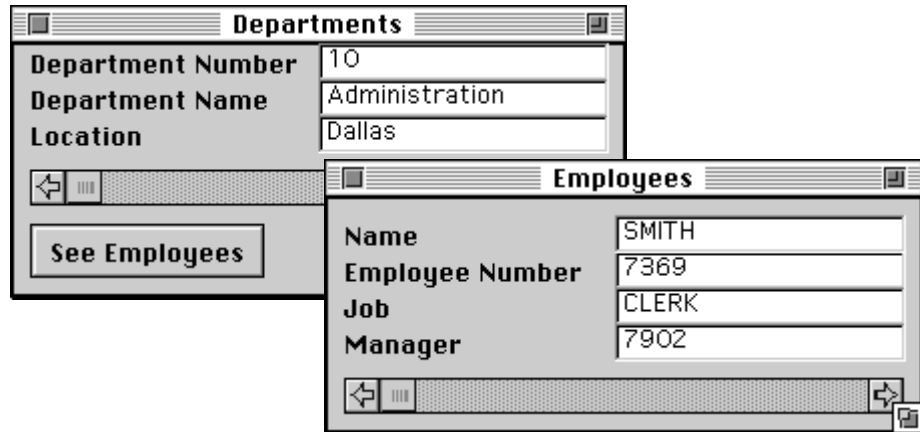
The screenshot shows a form window titled "Departments". The top section is a table with four rows, each representing a department. The first three rows have a left-pointing triangle, and the fourth row has a right-pointing triangle. The data is: (10, ACCOUNTING, NEW YORK), (20, RESEARCH, DALLAS), (30, SALES, CHICAGO), and (40, OPERATIONS, BOSTON). To the right of the table is a vertical scrollbar with up and down arrow buttons. Below the table are three input fields: "EMPNO" with the value "7499", "ENAME" with the value "ALLEN", and "JOB" with the value "SALESMAN". Below these are three more input fields: "MGR" with the value "7698", "SAL" with the value "1600", and "COMM" with the value "300". To the right of these fields is a vertical scrollbar with up and down arrow buttons.

▷ 10	ACCOUNTING	NEW YORK
▷ 20	RESEARCH	DALLAS
▷ 30	SALES	CHICAGO
▷ 40	OPERATIONS	BOSTON

EMPNO	ENAME	JOB
7499	ALLEN	SALESMAN
MGR	SAL	COMM
7698	1600	300

As a third option, you could display detail records on an entirely different form, as shown below. In this case, the employee records are visible only when the user presses the Employees pushbutton on the DEPT form, opening the separate EMP form:



Most commonly, detail records appear in either a repeater display or an embedded form within a form, and the master records appear in the form itself. However, in some cases you might want to reverse this situation, or place the detail records on a separate form. Additionally, you may also want to create multiple levels of master-detail relationships through an Oracle Power Objects application.

### Placing the Detail Records on the Form

Sometimes, it makes sense to display master records in a repeater display or embedded form, and display master records on the form itself. For example, because of the memory overhead required to display graphics, you might want to create a graphics browser that places the categories of graphics in a repeater display, while displaying the graphic on a form.

In this case, you must set the properties relevant to master-detail relationships on the form, not on the repeater display. In the case of the graphics browser, you would set the following properties of the form:

Property	Setting
LinkDetailColumn	PICTYPE
LinkMasterColumn	PICTYPE
LinkMasterForm	repPicTypes
LinkPrimaryKey	On Master

---

In this example, the column PICTYPE appears in both the master and detail recordsets, where it defines the type of graphic.

## Placing Detail Records on a Separate Form

In some cases, you may not be able to fit both master and detail records easily on a form, or it may not be necessary for the user to view master and detail records at the same time. In these cases, you can create two forms, one for master records and the other for detail records.

In this case, you set the **LinkMasterForm** property of the form displaying detail records to the name of the form displaying master records. You set the other relevant properties (**LinkDetailColumn** and **LinkMasterColumn**) in the same way as in other cases, specifying the primary and foreign keys for the master-detail relationship.

## Creating a Drilldown Form

You can define several levels of a master-detail relationship on the same form or across multiple forms. Commonly, developers create “drill-down” forms that let the user view a second set of detail records determined by the first set of detail records. In such a case, the developer has created two master-detail relationships, in which the first set of detail records are the masters for the second set of details.

For example, suppose you want add a second repeater display to the DEPT/EMP form that lets users view a selected employee’s history, as stored in the EMPHIST table:

Review Employee Histories		
RESEARCH	DALLAS	
▶ JONES	MANAGER	2975
▶ SCOTT	ANALYST	3000
▶ FORD	ANALYST	3000
▶ ADAMS	CLERK	1100
▶ SMITH	CLERK	800

Employee History	
▶ 10/3/94	Hired at company
▶ 10/21/94	Caught bullet meant for CEO
▶ 11/15/94	Married CEO's daughter
▶ 11/20/94	Divorced after rocky honeymoon
▶ 1/1/95	Vowed to begin anew

In this case, the repeater holding the employee history records designates the other repeater display (the one displaying employee records) as its master. To do this, you would set the following properties on the second repeater display:

<b>Property</b>	<b>Setting</b>
<b>LinkDetailColumn</b>	ENAME
<b>LinkMasterColumn</b>	ENAME
<b>LinkMasterForm</b>	repEmployees
<b>RecordSource</b>	EMPHIST

When the user selects an employee record from the first repeater, that employee's record then appears in the second repeater.

### Other Options for Displaying Master-Detail Relationships

The techniques discussed in this chapter for displaying master-detail records are by no means the only ones available to you. Both the object-oriented design and automated master-detail features of Oracle Power Objects provide further options for displaying master-detail relationships. As long as you set the **LinkMasterForm**, **LinkMasterColumn**, **LinkDetailColumn**, and **LinkPrimaryKey** properties correctly, you can use any bound containers to define master-detail relationships, including multilevel master-detail relationships.





# 19

---

## Using Constraints to Enforce Business Rules

This chapter covers the following topics:

Overview of Constraints . . . . .	19.2
Constraints in the Database . . . . .	19.2
Types of Database Constraints . . . . .	19.3
Defining Database Constraints . . . . .	19.9
Removing Database Constraints . . . . .	19.12
Constraints in the Application . . . . .	19.13
Using Database and Application Constraints Together . . . . .	19.28

---

## Overview of Constraints

*Constraints* (also called *integrity constraints*) restrict the ways in which users can add to, modify, or delete your application's data. Constraints are typically used to enforce *business rules*—guidelines that your organization follows to protect and standardize information.

For example, you might use a constraint to ensure that every employee is assigned to a department, or to ensure that users do not accidentally enter negative values for a product's price.

You can define constraints in two locations:

**In the database.** Constraints in the database are associated with the definitions of table objects. For example, a table can have a constraint that requires each value in a column to be unique.

**In the application.** Constraints in the application are associated with the application objects that form the interface to the information. For example, a text field can have a constraint that requires all values entered into it to be greater than 20.

This chapter describes the kinds of constraints you can enter in both locations. It also discusses the comparative advantages of using different constraint locations. Since many applications will use constraints in both locations, this chapter also provides an in-depth example that incorporates both types of constraints.

## Constraints in the Database

A database constraint is a declaratively defined rule restricting the values that can be entered into a column or set of columns in a table. Database constraints are said to be *declaratively defined* because you define constraints as part of the table structure when you create or modify it. Once you have associated a constraint with a table, the constraint is always enforced unless you explicitly remove or disable the constraint.

Constraints located in the database have the following advantages:

- **Centralization.** A database constraint can be defined once and be used automatically by all clients accessing the database. Defining the constraint in the database relieves you from having to add the same constraints to every form that uses the information. Also, when you need to update the constraint, you can make your changes in a single location.
- **Security.** Database constraints always apply no matter what data access tool is used. In contrast, constraints defined in your application could be violated by someone using a different application or tool that can connect to the same tables (for example, Oracle SQL\*Plus).
- **Simplicity.** Database constraints are easy to define and require little or no coding.

Database constraints are associated only with tables. You cannot associate constraints directly with views, but a view is subject to any constraints associated with its base tables. You associate constraints with tables in two ways: with a single column or with the entire table.

Not all databases support table-level constraints (Oracle7 Servers and Blaze databases support table constraints). Table constraints are more flexible than column constraints for the following reasons:

- They can apply to multiple columns at once
- They can be added after the table has been created

## Types of Database Constraints

The types of constraints that you can associate with a table vary depending on the database in which the table is stored. The constraint categories described in the following sections are supported by most relational databases.

### Not Null Constraints

A Not Null constraint prevents the user from entering a value of Null into a column of a table. A Not Null constraint always applies to a single column.

You use Not Null constraints to ensure that values are always entered for important columns. For example, you might use a Not Null constraint to ensure that every employee in your database has a corresponding salary.

### Unique Constraints

A Unique constraint prevents the user from entering duplicate values into a column or a set of columns. A Unique constraint can be enabled for a single column or for a combination of columns. A Unique constraint enabled for a combination of columns is sometimes called a *composite Unique constraint*.

You use Unique constraints to ensure that column values are not duplicated in your table. For example, you might use a Unique constraint to ensure that every product in your database has a different part number.

A Unique constraint by itself does not prevent the user from entering multiple null values—a null in a column always satisfies a Unique constraint. To prevent the entry of null values in a column with a Unique constraint, you must also add a Not Null constraint to the column.

Many databases (including Blaze databases, Oracle7 Servers, and SQL Server databases) enforce Unique constraints using an index object. In some databases (such as Oracle7 Servers), the index object is created automatically when you create the Unique constraint. In other databases (such as SQL Server databases), you create the Unique constraint by creating the index.

Because Unique constraints are often enforced using indexes, Unique constraints are frequently subject to the implementation-specific limitations of indexes. For example, composite Unique constraints are limited to 16 columns on an Oracle7 Server.

For information about indexes, see the section "Views" on page 8.16.

---

## Primary Key Constraints

A Primary Key constraint ensures that each row in the table is uniquely identified by the value in the primary key column or set of columns. A primary key constraint combines the features of a Unique constraint and a Not Null constraint.

Generally, you should include a Primary Key constraint in every table you create. Having a primary key can significantly improve the speed of accessing rows of the table. For databases that do not have automatic ROWIDs (such as SQL Server), Oracle Power Objects requires a primary key to identify single rows for locking and updating purposes.

A Primary Key constraint is also used to enforce *referential integrity* when master-detail relationships are defined in the database. Enforcing referential integrity entails maintaining the correspondence between the master table and a detail table. To maintain referential integrity, Primary Key constraints are often used in combination with Foreign Key constraints, described in the section "Foreign Key Constraints" on page 19.7.

In a master-detail relationship, the *primary key* is the column in the master table that holds a unique identifier for each row in the table. A foreign key column in a detail table uses the primary key value to specify a join between the rows, as illustrated in the following figure:

Table - DEPT (MLDATA)						
Column Name	Datatype	Size	Prec	Not Null	Unique	
DEPTNO	NUMBER			✓		
DNAME	VARCHAR2	14				
LOC						

Table - EMP (MLDATA)						
Column Name	Datatype	Size	Prec	Not Null	Unique	
EMPNO	NUMBER	4		✓		
ENAME	VARCHAR2	10				
JOB	VARCHAR2	9				
MGR	NUMBER	4				
HIREDATE	DATE					
SAL	NUMBER	7	2			
COMM	NUMBER	7	2			
DEPTNO	NUMBER	2				

The following rules are frequently used in maintaining referential integrity between a master table and a detail table:

- A detail row cannot be inserted until the master row exists.
- A master row cannot be deleted without deleting all detail rows first.
- If a primary key value in a master row is changed, all foreign key values that refer to that primary key value must be updated as well.

For some databases (such as Oracle7 Servers), you can maintain full referential integrity by simply creating Primary Key and Foreign Key constraints. In other databases (such as SQL Server), you must define your referential integrity processing separately (typically in a trigger). However, in either case, you must define Primary Key and Foreign Key constraints in order to enforce referential integrity rules in the database.

➤ **Note:** You can also enforce referential integrity rules in your application, as described in the section “Dropping Constraints in the Table Editor Window” on page 19.12. Application-based referential integrity does *not* require you to specify Primary Key and Foreign Key constraints, although it is often useful to have this information stored in the database even if it is not used there. Regardless of where you choose to enforce referential integrity, you should always define a Primary Key constraint for every table in your database.

A Primary Key constraint can be enabled for a single column or for a combination of columns (called a *composite key*, a *multi-column key* or a *multi-segmented key*). A composite key constraint ensures that each row has a unique combination of values in its key columns.

For example, the following diagram shows a composite primary key constraint on the EMP table:

Table - EMP (MLDATA)						
Column Name	Datatype	Size	Prec	Not Null	Unique	
EMPNO	NUMBER	4		✓		
ENAME	VARCHAR2	10				
JOB	VARCHAR2	9				
MGR	NUMBER	4				
HIREDATE	DATE	11				
SAL	NUMBER	7	2			
COMM	NUMBER	7	2			
DEPTNO	NUMBER	2				

The following row cannot be added to the table, because it violates the Primary Key constraint:

This row violates the constraint

Table - EMP (MLDATA)								
EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO	
7839	KING	PRESIDENT		17-NOV-81	5000		10	
7844	TURNER	SALESMAN	7698	08-SEP-81	1500	0	30	
7876	ADAMS	CLERK	7788	12-JAN-83	1100		20	
7900	JAMES	CLERK	7698	03-DEC-81	950		30	
7902	FORD	ANALYST	7566	03-DEC-81	3000		20	
7934	MILLER	CLERK	7782	23-JAN-82	1300		10	
7934	MILLER							

However, the following row can be added to the table, because the *combination* of ENAME and EMPNO is unique:

This row does not violate the constraint

Table - EMP (MLDATA)								
EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO	
7839	KING	PRESIDENT		17-NOV-81	5000		10	
7844	TURNER	SALESMAN	7698	08-SEP-81	1500	0	30	
7876	ADAMS	CLERK	7788	12-JAN-83	1100		20	
7900	JAMES	CLERK	7698	03-DEC-81	950		30	
7902	FORD	ANALYST	7566	03-DEC-81	3000		20	
7934	MILLER	CLERK	7782	23-JAN-82	1300		10	
7935	MILLER							

For information about indexes, see the section "Views" on page 8.16.

Many databases (including Oracle7 Servers and SQL Server databases) enforce Primary Key constraints using an index object. In some databases (such as Oracle7 Servers), the index object is created automatically when you create the Primary Key constraint. In other databases (such as SQL Server databases), you must create the index before you create the Primary Key constraint.

When Primary Key constraints are enforced using indexes, Primary Key constraints are subject to the limitations of indexes. For example, composite Primary Key constraints are limited to 16 columns on an Oracle7 Server.

## Foreign Key Constraints

A Foreign Key constraint ensures that each value entered into the column is already present in a different column (usually in another table). Foreign Key constraints are typically used to maintain referential integrity when master-detail relationships are defined in the database. Foreign Key constraints are always used in conjunction with Primary Key constraints (described in the previous section).

In a master-detail relationship, the *foreign key* is the column in the detail table that holds an identifier for a row in the master table. A value in a foreign key column is equal to a value in a primary key column in another table.

In a *one-to-one* relationship, each row in the detail table corresponds to a unique row in the master table. In a *one-to-many* relationship, any number of rows in the detail table can correspond to the same row in the master table.

For information about how Foreign Key constraints are used to enforce referential integrity in the database, see the section “Primary Key Constraints” on page 19.4.

A Foreign Key constraint can be enabled for a single column or for a combination of columns (called a *composite key*, a *multi-column key* or a *multi-segmented key*). A composite key constraint allows you to specify a join to a composite Primary Key in another table.

## Check Constraints

A Check constraint (also called a *rule*) ensures that only values matching specified criteria can be entered in the column or set of columns. A Check constraint compares the values to be entered with a specific condition.

You use a Check constraint to enforce business rules that require comparison or calculation. For example, you might use a Check constraint to ensure that all salaries in your database are greater than zero, or to ensure that all employee names are entered in upper case.

The capabilities of a Check constraint vary from database to database.

For an Oracle7 Server or a Blaze database, a Check constraint can refer to values in other columns. For example, you can create an Oracle7 constraint ensuring that the value in one column does not exceed a value in another column. However, a Check constraint cannot include a subquery or use certain SQL functions (such as SYSDATE and USER). Oracle7 Servers and Blaze databases allow you to associate Check constraints with both individual table columns and with entire table objects.

For SQL Server, a Check constraint (a rule) can include operators and values, but cannot reference any other column or database object. SQL Server databases allow you to associate rules with individual table columns and with user-defined datatypes (any column of that datatype in any database object automatically uses the rule).

---

## List of Supported Constraints

The following table indicates which types of constraints are supported by which types of databases:

<b>Constraint Type</b>	<b>Blaze</b>	<b>Oracle7 Server</b>	<b>SQL Server</b>
Not Null	Yes	Yes	Yes
Unique	Yes	Yes	Yes
Primary Key	Yes	Yes	Yes
Foreign Key	Yes	Yes	Yes
Check	Yes	Yes	Yes

## Other Database Integrity Checks

Some databases provide additional techniques for ensuring data integrity and enforcing business rules. Two common techniques use *triggers* and *stored procedures*.

**Triggers** are subprograms associated with database objects. Triggers can perform complex processing when the table's rows are inserted, deleted, or updated.

**Stored procedures** are subprograms that are stored in the database as separate objects. Unlike triggers, stored procedures are not intrinsically associated with the definition of a table, although many kinds of database objects can be accessed from within the procedure. Stored procedures can perform complex tasks when specifically called. Stored procedures can be called from application code or from another trigger or stored procedure.

Triggers and stored procedures are often used to enforce rules that cannot easily be defined in other ways. For example, a trigger can automatically copy into an auditing table rows deleted from another table. A stored procedure can perform queries and subqueries on other tables to enforce complex relationships. Some databases use triggers and stored procedures to provide unique column values or to enforce referential integrity between tables.

Triggers and stored procedures usually consist of SQL statements combined with additional programming constructs (such as variables and flow-of-control statements). The specific syntax of triggers and stored procedures varies greatly from database to database.

For information about defining and using triggers and stored procedures, see the documentation accompanying your database.



## Defining Database Constraints

You can specify a constraint in two ways:

- By specifying values in the Table Editor window.
- By issuing a SQL command (or, for SQL Server, executing a system procedure) with the appropriate clauses.

These techniques are described below.

### Defining Constraints in the Table Editor Window

You can define the following types of column constraints in the Table Editor window:

- Not Null
- Unique
- Primary Key

To specify other types of constraints, you must use a SQL command (or a system procedure), as described later in this chapter.

#### ☆ To define a Not Null constraint in the Table Editor window:

1 Click on the Not Null column so that a check mark appears.



2 Save the table definition by choosing the **File-Save** menu command or clicking on the Save button.

#### ☆ To define a Unique constraint in the Table Editor window:

1 Click on the Unique column so that a check mark appears.



2 Save the table definition by choosing the **File-Save** menu command or clicking on the Save button.

#### ☆ To define a Primary Key constraint in the Table Editor window:

1 Select the column for which you want to enable the constraint.

You can select the column by using the keyboard, or by clicking anywhere in the desired row.

2 Click on the Primary Key tool.

3 To define a composite primary key, repeat steps 1 and 2 for additional columns in the key.



4 Save the table definition by choosing the **File-Save** menu command or clicking on the Save button.

---

## Defining Constraints Using SQL Commands or System Procedures

You can define any type of constraint by issuing a SQL command (or, for SQL Server, executing a system procedure). This section lists the commands used to define constraints, along with a simple example of each command. For a complete reference to the commands described, see the SQL language reference manual provided with your database.

### Blaze Database Constraints

Type of Constraint	SQL Commands	Example
Not Null	CREATE TABLE	EXEC SQL CREATE TABLE nn_tab & (id NUMBER NOT NULL)
	ALTER TABLE	EXEC SQL ALTER TABLE nn_tab2 & (id NUMBER NOT NULL)
Unique	CREATE TABLE	EXEC SQL CREATE TABLE unq_tab & (id NUMBER UNIQUE)
	ALTER TABLE	EXEC SQL ALTER TABLE unq_tab2 & ADD UNIQUE(id)
Primary Key	CREATE TABLE	EXEC SQL CREATE TABLE pk_tab & (id NUMBER PRIMARY KEY)
	ALTER TABLE	EXEC SQL ALTER TABLE pk_tab2 & ADD PRIMARY KEY (id)

### Oracle7 Server Constraints

Type of Constraint	SQL Commands	Example
Not Null	CREATE TABLE	EXEC SQL CREATE TABLE nn_tab & (id NUMBER NOT NULL)
	ALTER TABLE	EXEC SQL ALTER TABLE nn_tab2 & (id NUMBER NOT NULL)
Unique	CREATE TABLE	EXEC SQL CREATE TABLE unq_tab & (id NUMBER UNIQUE)
	ALTER TABLE	EXEC SQL ALTER TABLE unq_tab2 & ADD UNIQUE (id)

Type of Constraint	SQL Commands	Example
Primary Key	CREATE TABLE	EXEC SQL CREATE TABLE pk_tab & (id NUMBER PRIMARY KEY)
	ALTER TABLE	EXEC SQL ALTER TABLE pk_tab2 & ADD PRIMARY KEY (id)
Foreign Key	CREATE TABLE	EXEC SQL CREATE TABLE fk_tab & (id NUMBER PRIMARY KEY, & pk_tab_id NUMBER & REFERENCES pk_tab.id)
	ALTER TABLE	EXEC SQL ALTER TABLE fk_tab2 & ADD (FOREIGN KEY(pk_tab_id) & REFERENCES pk_tab.id)
Check	CREATE TABLE	EXEC SQL CREATE TABLE chk_tab & (id NUMBER CHECK (id > 0))
	ALTER TABLE	EXEC SQL ALTER TABLE chk_tab2 & ADD CHECK (id > 0)

**SQL Server Constraints**

Type of Constraint	SQL Commands/ System Procedures	Example
Not Null	CREATE TABLE	EXEC SQL CREATE TABLE nn_tab & (id INT NOT NULL)
Unique	CREATE INDEX	EXEC SQL CREATE UNIQUE & INDEX unq_tab_id_index & ON unq_tab (id)
Primary Key	CREATE INDEX + sp_primarykey	EXEC SQL CREATE UNIQUE & CLUSTERED INDEX pk_tab_id_index & ON pk_tab (id) EXEC SQL sp_primarykey pk_tab, id
Foreign Key	sp_foreignkey	EXEC SQL sp_foreignkey fk_tab, pk_tab_id
Check	CREATE RULE + sp_bindrule	EXEC SQL CREATE RULE & chk_tab_id_rule AS @id > 0 EXEC SQL sp_bindrule & chk_tab_id_rule, 'chk_tab.id'

---

## Removing Database Constraints

You can remove a constraint in two ways: by disabling the constraint or by dropping it.

**Disabling** a constraint means that the constraint is still stored as part of the table's definition, but it is not enforced. Constraints are sometimes disabled to improve speed when importing a large number of rows into a table. Support for disabling constraints varies from database to database.

**Dropping** a constraint means that the constraint is permanently removed from the table's definition.

Once a constraint has been disabled or dropped, the constraint is no longer enforced. A constraint can be re-enabled or re-created only if no rows have been added to the table that violate the constraint.

You can remove a constraint in the following ways:

- By specifying values in the Table Editor window.
- By issuing a SQL command (or, for SQL Server, executing a system procedure) with the appropriate clauses.

### Dropping Constraints in the Table Editor Window

You can drop the following types of column constraints in the Table Editor window:

- Not Null
- Unique
- Primary Key

To drop other types of constraints, or to disable any kind of constraint, you must use a SQL command or a system procedure.

#### ☆ To drop a Not Null constraint in the Table Editor window:

1 Click on the Not Null column so that the check mark disappears.



2 Save the table definition by choosing the **File-Save** menu command or clicking on the Save button.

#### ☆ To drop a Unique constraint in the Table Editor window:

1 Click on the Unique column so that the check mark disappears.



2 Save the table definition by choosing the **File-Save** menu command or clicking on the Save button.

☆ **To drop a Primary Key constraint in the Table Editor window:**

- 1 Select the column for which you want to drop the constraint.  
You can select the column by using the keyboard, or by clicking anywhere in the desired row.
- 2 Click on the Primary Key tool to remove the key icon from the selected column.
- 3 To drop a composite primary key, repeat steps 1 and 2 for all additional columns in the key.
- 4 Save the table definition by choosing the **File-Save** menu command or clicking on the Save button.



### Dropping or Disabling Constraints Using SQL Commands or System Procedures

You can drop or disable any type of constraint by issuing a SQL command or (for SQL Server) executing a system procedure. For a discussion of the appropriate commands and related issues, see the SQL language reference manual provided with your database.

## Constraints in the Application

Oracle Power Objects provides many ways to enforce business rules on the client (that is, within the application). Some require method code, but many do not. You enforce constraints on the client in the following cases:

- **Immediate Response** - You want the application to respond immediately to changes entered by the user.
- **Reduced Network Traffic** - You want to relieve network traffic by performing as many checks as possible on the client before flushing changes to the database.
- **Reduced Server Burden** - You want to remove some of the server's processing burdens by moving some constraints to the client.
- **Interactivity** - You can prompt the user to add or change data.
- **Multi-Database Constraints** - Since recordsets can be queried from multiple databases, represented by several different sessions, you can enforce the same business rules across all of these databases.

Client-enforced constraints act on the beginning of a transaction, when it is initiated on the front end of the database application. In Oracle Power Objects, a transaction spans several application layers, from the user interface through the recordset, the session, and finally the database itself. Therefore, you can intercept the transaction at several points, applying a constraint at the appropriate time.

---

In this program architecture, you can enforce business rules from the client when:

- The value in a control changes, either manually (by some action taken by the user) or programmatically.
- A row in a recordset is inserted, deleted, or updated.
- Modifications are made to a master or detail recordset in a master-detail relationship.
- The application tries to flush changes to the database.

Each of these types of constraints is described below.

## Control-Level Constraints

An Oracle Power Objects application can enforce many types of business rules within an individual control. For example, you can limit the text entered into a text field in several ways:

- By constraining the length of strings entered in the control.
- By applying a format mask to data entered in the control.
- By performing a programmatic check on the data entered before the user moves the focus out of the control.

This section summarizes the techniques for creating these control-level constraints. In some cases, you create the constraint by setting a property of the control; in others, you must write method code to enforce the business rule.

### Constraints on Datatype and DataSize

Some of the simplest constraints can be created through the **Datatype** and **DataSize** properties of the control. **Datatype** limits the kind of data (*String*, *Date*, *Long*, or *Double*) that can be entered or displayed in the control.

**DataSize** limits the number of bytes allocated for holding a control's data in memory. In addition, **DataSize** commonly limits the number of characters that can appear in a control with the *String* datatype. The number entered for this property is the number of characters that can appear in a text string displayed in the control. If the control queries a string from a column in a table or view that is longer than the **DataSize** property permits, the application truncates the end of string to fit the specified **DataSize**.

#### ☆ To set the Datatype of a control:

- 1 From within the control's Property sheet, click on the **Datatype** property.
- 2 From the scrolling list that appears, select the desired datatype.

☆ **To set the DataSize of the control:**

- 1 From within the control's Property sheet, click on the **DataSize** property.
- 2 Enter a non-negative numeric value for the **DataSize** of the control.

**Read-Only Values**

Often, you simply want to prevent the user from entering a value into a control. For example, in the Order Entry form shown below, the entry date is generated automatically whenever the user creates a new order, but you do not want the user to change this date. In the same form, though the user can select a SHIP FROM location, the details of this shipping site are not editable.



In these cases, you can use the **ReadOnly** and **Enabled** properties of a control to limit user access. When the **ReadOnly** property is set to TRUE, or when the **Enabled** property is set to FALSE, the user cannot move the focus into the control, making it impossible for the user to interact with it.

The difference between controls disabled using the **ReadOnly** and **Enabled** properties is their appearance. The text in a control whose **ReadOnly** property is set to TRUE appears normal. However, the text in a control whose **Enabled** property is set to FALSE is gray. This is a common graphical cue for a disabled control, and should be used when you want to inform the user that the control is disabled.

---

Both the **ReadOnly** and **Enabled** properties can be set programmatically. For example, the following method code disables a **Save** pushbutton, called `btnSave`, on a form if there is no value entered in a key field:

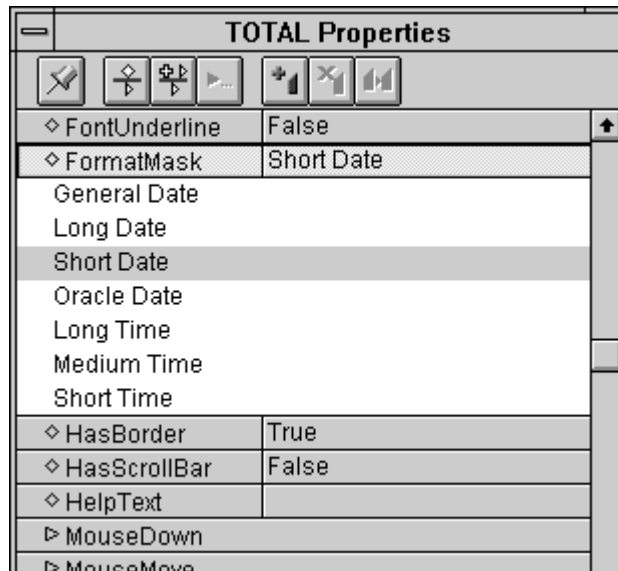
```
IF ISNULL(fldItemCode.Value) THEN
    btnSave.Enabled = FALSE
END IF
```

### Format Masks

For information about format mask characters, see the section "Format Masks" on page 10.31.

In Oracle Power Objects, a format mask determines how data are displayed, but does not constrain the values that the user can enter. The format mask is a string of characters specifying the format for displaying data. Each datatype has its own set of allowable format mask characters.

The format mask represents a constraint only in that it may suggest the format for entering data. For example, when you apply the standard format mask *Short Date*, date information appears in the *m/d/y* format (for example, 3/11/95). However, the user can still enter date information in other formats (for example, 11-March-95); the control then translates the information to fit the format mask.





### Counter Fields

Text fields can be configured to produce a unique value, or counter, whenever the user inserts a new record. This functionality duplicates on the front end what a sequence does in the database, creating a unique value for a record's primary key. If your Oracle Power Objects application is running against a database that does not support sequences, counter fields provide the only automated means for generating unique key values.

You can create three kinds of counters:

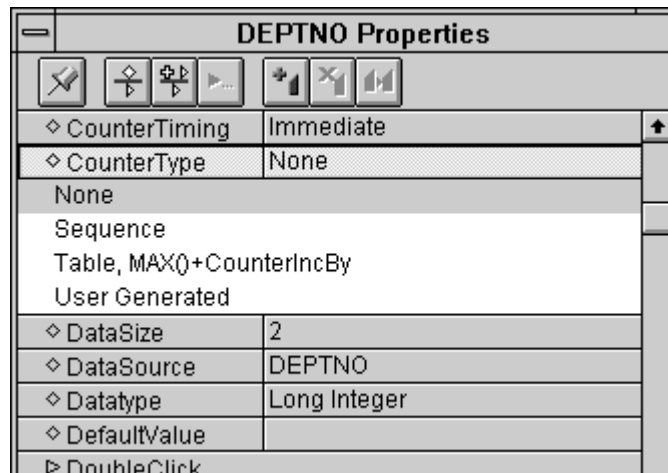
Counter Type	Description
Sequence-Generated	The application uses a sequence object in a database to generate a new value for a counter. To use a sequence-generated counter, set the <b>CounterType</b> property of the text field to "Sequence", and identify the table used for generating the counter in the <b>CounterSeq</b> property.
Automated	After reading the highest value set for a counter in the column to which the text field is bound, the application uses the value entered for the <b>CounterIncBy</b> property to increment each new counter. The application locks the table long enough to generate the new counter value.
User-Defined	The developer enters an algorithm for defining a new counter as part of the <b>CounterGenKey()</b> method (see below).

The best kind of counter depends on your database platform. For databases that support sequences, a sequence-generated counter is preferred. For other databases, an automated counter is fast, but less reliable than a user-defined counter. Additionally, the automated counter locks the table until the end of the transaction, so other users cannot generate a key value until the transaction is committed or rolled back. The disadvantage of a user-defined counter is that it can require significant coding and custom database setup.

Four properties of a text field can be used to define a counter. The first, **CounterType**, determines the way in which the counter is generated.

☆ **To set the CounterType property:**

- 1 Click on the **CounterType** section of the text field's Property sheet.
- 2 Select the type of counter from the scrolling list that then appears.



Counter Type	Value for CounterType Property
No counter	None
Sequence-Generated	Sequence
Automated	Table, MAX( )+CounterIncBy
User-Defined	User-Generated

If you have the **CounterType** property set to *User*, the application calls the **CounterGenKey()** method. Since this method has no default processing, you must add method code to **CounterGenKey()** to define the algorithm for generating a new counter value.

The following table summarizes the other three important properties relevant to generating counter values:

Property	Description
<b>CounterIncBy</b>	Determines the increment between counters.
<b>CounterSeq</b>	Identifies the sequence used for generating counters. This property required when the <b>CounterType</b> is set to <i>Sequence</i> .

Property	Description
CounterTiming	Determines when the counter is generated. Possible settings include Immediate (when the user inserts a new record) and Deferred (when the application flushes the new record to the database).

### Validate() Method

The most important control-level constraints are normally defined through the **Validate()** method. You enter method code to check whether data entered by the user meet your criteria. The **Validate()** method is a function that can return TRUE or FALSE:

- If the return value is TRUE, the value is accepted and the user can move the focus out of the control.
- If the return value is FALSE, the value is rejected, and the focus cannot leave the control until the user enters a valid value.

Method code added to **Validate()** can have the method return TRUE, indicating that the validation was successful. The user can continue working with other parts of the application.

If **Validate()** returns FALSE, the focus cannot leave the control until the user enters a valid value. In addition, if an error message has been entered for the **ValidateMsg** property, this text appears in a message box when the validation fails.

The application triggers **Validate()** on a control after (1) its **Value** property has changed, and (2) one of the following events occurs:

- The user tries to tab out of the control.
- The user clicks anywhere outside the control.
- The user presses the Enter key after editing the contents of a text field or combo box.
- The user makes a new selection in a list box, combo box, or popup list.
- The user selects the control, in the case of a radio button or check box.

The default processing of **Validate()** is for the method to return TRUE. Therefore, any method code added to **Validate()** causes it to return 0 (that is, FALSE), unless you enter the following line somewhere in the method code:

```
Validate = TRUE
```

If you call the method **RevertValue()** on the same control after **Validate()** returns FALSE, the value displayed in the control reverts to the one displayed before the user entered a rejected value. The call to **RevertValue()** must occur within the method code for **Validate()**.

Oracle Power Objects also includes a row-level validation method, **ValidateRow()**, as a standard method of bindable containers. For more information on **ValidateRow()**, see the section “**ValidateRow()**” on page 19.20.

---

To illustrate how to use `Validate()`, the following method code ensures that a discount is not a negative number, and that it does not exceed 100% of the value of the item. The method code would appear in the `Validate()` method of the control in which the user enters the discount amount. Aside from performing the check, the code also modifies the `ValidateMsg` property to display a different error message, depending on the reasons why the validation failed.

```
SELECT CASE Self.Value
CASE 0 TO 100
    Validate = TRUE
CASE IS > 100
    Validate = FALSE
    Self.ValidateMessage = "You entered a discount > 100%"
CASE IS < 0
    Validate = FALSE
    Self.ValidateMessage = "You entered a negative discount"
END SELECT
```

For information about `SQLLOOKUP`, see the section "The `SQLLOOKUP` Function" on page 9.21.

To illustrate further, the following method code checks to see if a user ID entered into a text field exists in the `USERS` table. The method code uses the `SQLLOOKUP` function to perform this test.

```
IF NVL(SQLLOOKUP("select count(*) from USERS where" & &
"USER_NAME like " + newval), 0 > 0 THEN
    Validate = TRUE
ELSE
    MSGBOX "User does not exist"
    Validate = FALSE
END IF
```

## Row-Level Constraints

You can enforce constraints in several ways when the application tries to perform a row-level operation (adding, deleting, or modifying a record). The following table summarizes the standard methods of bound containers that you can use to enforce row-level constraints:

<b>Method</b>	<b>When to Use</b>
<code>ValidateRow()</code>	Inserting or modifying a row
<code>InsertRow()</code>	Inserting a row
<code>DeleteRow()</code>	Deleting a row

### `ValidateRow()`

The `ValidateRow()` method is triggered whenever the user or the application tries to commit changes, or when the user tries to move to another row. `ValidateRow()` is a standard property of a bindable container, not of a control.

You use **ValidateRow()** instead of **Validate()** when you must check on dependencies between values in the recordset, or when you want to make sure that some values in the record are not null. For example, you may want to ensure that an employee's commission is not larger than his or her salary. In this case, the constraint is enforced at the level of the entire record, and should therefore be enforced when the user tries to save modifications to it.

If **ValidateRow()** returns FALSE, the application displays whatever error message is defined for the **ValidateRowMsg** property of the container.

The default processing of **ValidateRow()** returns TRUE. Therefore, any method code added to **ValidateRow()** causes the method to return 0 (that is, FALSE), unless you add the following statement to the code:

```
ValidateRow = TRUE
```

You can call the **RevertRow()** method when **ValidateRow()** returns FALSE, so that all controls within the container display their original values, before the user entered changes.

To illustrate, the following method code checks to make sure that the entry date of an invoice is not later than its apply date. The values for these two dates appear in the text fields fldEntryDate and fldApplyDate. You would enter this code in the **ValidateRow()** method of the container in which these text fields appear.

```
IF fldApplyDate.Value < fldEntryDate.Value THEN
    ValidateRow = FALSE
    Self.RevertRow()
ELSE
    ValidateRow = TRUE
END IF
```

### InsertRow() and DeleteRow()

Although **ValidateRow()** is often used to enforce business rules at the row level, you can also use **DeleteRow()** and **InsertRow()** for the same purpose when deleting or adding a row. Like **ValidateRow()**, both methods are part of any bound container.

As their names imply, the default processing is inserting a new record in the recordset for **InsertRow()**, and deleting a the current row for **DeleteRow()**. Therefore, any method code added to these methods can override the insertion or deletion of the row. However, you can allow the insertion or deletion by adding the statement `Inherited.DeleteRow()` or `Inherited.InsertRow()` to the appropriate method. If you are using these methods to perform these checks, you should add the `Inherited.method_name()` statement only if the user has satisfied the necessary criteria.

---

For example, the following method code appears in `DeleteRow()`, and prevents the user from deleting an customer record if the customer has an outstanding balance due.

```
IF fldBalanceDue.Value <> 0 THEN
    MsgBox ("You cannot delete a customer with an " & &
        "outstanding balance", 16, "Error!")
ELSE
    Inherited.DeleteRow()
END IF
```

Any method code appearing before the `Inherited.DeleteRow()` statement executes before the actual deletion, while method code appearing after the statement executes after the deletion. Similarly, code appearing before the `Inherited.InsertRow()` statement executes before the insertion, while code appearing after the statement executes after the insertion.

### Other Methods

Three container-level methods are always triggered when you insert, delete, or modify a row. Two of these methods are triggered immediately before the actual recordset operation, while the other is triggered after the operation (add, delete, or update) has occurred.

The names of these three methods are `PrexxxCheck()`, `Prexxx()`, and `Postxxx()`, where `xxx` is the name of the recordset operation (Insert, Delete, or Update). For example, inserting a record triggers methods `PreInsertCheck()`, `PreInsert()`, and `PostInsert()`.

The following table describes these three kinds of methods:

Method	Description
<code>PrexxxCheck()</code>	Triggered before the application receives notification that the recordset operation should proceed. <code>PrexxxCheck()</code> methods return either TRUE if the operation should proceed, or FALSE if it should not. The default processing of <code>PrexxxCheck()</code> methods returns TRUE, so any method code that does not include an <code>Inherited.PrexxxCheck()</code> statement prevents the recordset operation from occurring.
<code>Prexxx()</code>	Triggered after the application receives notification that the recordset operation should proceed, but immediately before the operation occurs. Unlike <code>PrexxxCheck()</code> , <code>Prexxx()</code> methods cannot prevent the recordset operation from occurring.
<code>Postxxx()</code>	Triggered after the operation is completed.

In the cases of insertions and deletions, `InsertRow()` and `DeleteRow()` execute first. Part of their default processing is to call their corresponding `PrexxxCheck()`, `Prexxx()`, and `Postxxx()` methods.

When you delete a record, the following methods are called:

Method	Action
<b>DeleteRow()</b>	If <b>PreDeleteCheck()</b> returns TRUE, deletes the row and calls <b>PreDelete()</b> and <b>PostDelete()</b> .
<b>PreDeleteCheck()</b>	Determines whether the deletion should occur.
<b>PreDelete()</b>	Performs any processing to occur immediately before the deletion, and calls <b>PostDelete()</b> .
<b>PostDelete()</b>	Performs processing after the deletion. Execution then returns to <b>DeleteRow()</b> .

The purpose of this method calling sequence is to give you fine control over the timing of certain tasks, including the enforcement of a constraint. To use our previous example, if you wanted to prevent the user from deleting customer records with an outstanding balance, you could add the necessary method code to either **DeleteRow()** or **PreDeleteCheck()**, depending on how close to the actual deletion you wanted to place the validation routine.

Since **DeleteRow()** and **InsertRow()** call their associated **PrexxxCheck()**, **Prexxx()**, and **Postxxx()** methods as part of their default processing, these three methods are triggered before the application returns execution to **DeleteRow()**. This means, for example, that any method code appearing after the statement `Inherited.DeleteRow()` in the **DeleteRow()** method executes only after **PreDeleteCheck()**, **PreDelete()**, and **PostDelete()** have finished execution.

## Master-Detail Constraints

For information about properties needed to establish a master-detail relationship, see Chapter 18, "Defining Master-Detail Relationships".

In addition to constraints imposed on a single bound container, you can enforce constraints on master and detail containers through several means. This chapter assumes that you are familiar with these properties and with the common techniques for representing master-detail relationships in an application.

As explained in Chapter 18, "Defining Master-Detail Relationships", two properties of the container holding detail records, **LinkMasterColumn** and **LinkDetailColumn**, specify the primary key and foreign key columns, respectively. A third property of the container, **LinkMasterForm**, identifies the bound container displaying the master records in the master-detail relationship.

Some techniques for enforcing master-detail constraints are automated in Oracle Power Objects through the Record Manager. This portion of the application (1) maintains recordsets, and (2) enforces both entity integrity and referential integrity.

- Entity integrity** is enforced through the assignment of a primary key to a record source. Therefore, if the application queries records from the DEPT table, the Record Manager knows that the DEPTNO column is the primary key column. When the user adds or modifies the record, therefore, the application ensures that a non-null value is entered for DEPTNO.

- 
- **Referential integrity** is enforced by specifying the primary key and foreign key columns for joined recordsets. The properties **LinkMasterUpd** and **LinkMasterDel** determine the kind of referential integrity maintained for a container.

Once you have specified these properties, the application automatically enforces referential integrity whenever the user inserts, deletes, or modifies a master or detail row.

### Master Rows

Three referential integrity concerns arise when the user edits a master recordset:

- **Inserting a row** - Will associated detail rows have foreign key values that match the primary key value for the master row?
- **Updating a row** - If the user modifies the primary key value, will the detail rows have their foreign key values similarly updated, or will they be *orphaned* (that is, no longer associated with the master row)?
- **Deleting a row** - If the user deletes a master row, will the associated detail records also be deleted, or will they be orphaned?

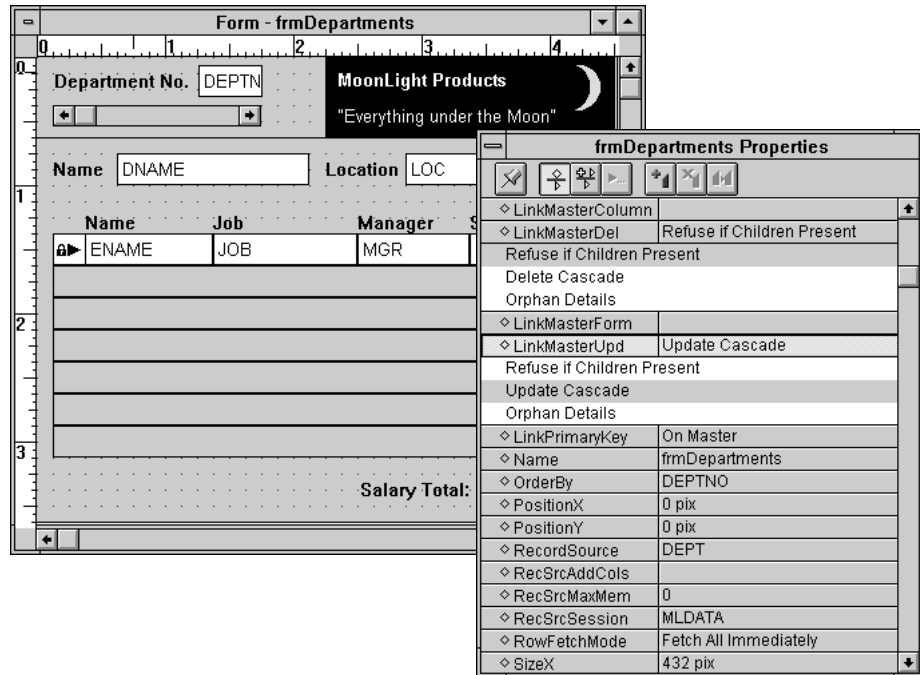
Depending on the business rules you wish to apply to master and detail records, you may answer these three questions differently. For example, if you want to prevent any detail records from being orphaned, you may want to make it impossible to delete the master row as long as any corresponding detail rows exist.

In Oracle Power Objects, the following properties determine the kind of referential integrity you wish to enforce when the user inserts, deletes, or updates a master record.

Property	Description
<b>LinkMasterDel</b>	Determines whether the user can delete a master record, if associated detail records exist. You set the <b>LinkMasterDel</b> property on the bound container displaying detail records. Settings for this property include <i>Refuse if Children Present</i> (prevent the deletion if there are detail records), <i>Delete Cascade</i> (delete all detail records when the master record is deleted) and <i>Orphan Details</i> (let the detail records be orphaned when the master record is deleted).
<b>LinkMasterUpd</b>	Determines whether the user can update the primary key value of a master record, if associated detail records exist. Again, you set this property on the bound container displaying detail records. Settings for <b>LinkMasterUpd</b> include <i>Refuse if Children Present</i> (prevent the update if there are detail records), <i>Update Cascade</i> (update the foreign key values of all detail records to match the new primary key) and <i>Orphan Details</i> (let the detail records be orphaned when the primary key value changes).



For example, the following figure shows these two properties being set for a form that will enforce referential integrity rules for a master-detail relationship:



### Detail Rows

When the user edits a detail recordset, two concerns arise:

- **Inserting a new record** - Will the new row have a foreign key value that corresponds to the appropriate primary key value in the master recordset?
- **Updating the foreign key** - If the user changes the foreign key value in a detail row, will the record be orphaned?

In Oracle Power Objects, the application does not automatically prevent the second problem, in the same way that you can prevent orphaning through the **LinkMasterDel** and **LinkMasterUpd** properties. However, you can write method code associated with the **Validate()** method of the control holding the foreign key value to prevent the detail record from being orphaned. An even simpler solution is to disable this control, or not display it on the detail recordset's associated container.

When you insert a record, the application automatically assigns the correct foreign key value to a new row, *even if no control associated with that column appears in the detail recordset's associated container*. As discussed in Chapter 17, "Binding a Container to a Record Source", an application

---

always queries the columns needed for a join as part of the master and detail recordsets associated with bound containers. For example, if you have a list of employees appearing in a repeater display, each employee record needs a department number that corresponds to a department number in the master recordset. However, in the form shown below, no bound control connected to the DEPTNO column appears in the repeater display.

In this case, the application applies the proper DEPTNO value to the new employee record, even though no control is bound to DEPTNO in the repeater display.

Behind the scenes, Oracle Power Objects maintains a column in the detail recordset for the foreign key, even if it is not bound to a control appearing in the recordset's associated container. In the example, the DEPTNO column is always part of the repeater display's recordset, even though no associated bound control appears in this container. Although the user cannot access this additional column, you can access it programmatically through several recordset methods (for example, `GetColVal()`, `GetColName()`, etc.).

## Session-Level Constraints

Earlier sections have described how to enforce constraints at the level of bound controls and bound containers, where transactions are initiated. This section summarizes points where you can control transactions at the level of a bound container or a session.

Using several standard methods, you can control whether the work pending within a particular session is committed or rolled back. These methods can operate on an individual session, or on all sessions represented within the same bound container.

<b>Method</b>	<b>Object</b>	<b>Description</b>
<code>GetRecordset()</code>	Container	Identifies the recordset of a bound container, using the syntax <code>container.GetRecordset()</code> .
<code>GetSession()</code>	Recordset	Returns an object reference to a session. You must call this method on a recordset object.
<code>IsWorkPending()</code>	Session	Indicates whether work is pending within a particular session (that is, the session has an uncommitted transaction).
<code>CommitWork()</code>	Session	Flushes changes pending within a session to the database and issues the SQL command COMMIT to commit the transaction.
<code>RollbackWork()</code>	Session	Flushes any deferred database operations associated with the session, and then issues the SQL command ROLLBACK.

Method	Object	Description
<b>CommitForm()</b>	Container	Flushes work pending in all sessions represented within the same container to the database, then issues the SQL command COMMIT to the database in each session. If other bound containers appear within the container, <b>CommitForm()</b> commits the work for all of them, even if they connect to separate sessions.
<b>RollbackForm()</b>	Container	Issues the SQL command ROLLBACK through all these sessions to the database.

To illustrate, the following code sample determines whether work is pending within a particular session represented on a form, by calling the **IsWorkPending()** method. Additionally, if work is pending within this session, the code then commits work pending in all sessions represented within the form, by calling the **CommitForm()** method.

For example, the following method gives the user the option to roll back all work within a bound container.

```

CONST BTN_OK = 1
IF MSGBOX("Roll back all work?", 33, "Attention") = BTN_OK
THEN
    Self.RollbackForm()
END IF
    
```

➤ **Note:** In the current release of Oracle Power Objects, two-phase commit has not been implemented. If you want to write all changes to the database, you should call **CommitWork()** on each session, instead of **CommitForm()** on the bound container.

## Other Client-Enforced Constraints

The list of client-enforced constraints described in this section is by no means comprehensive. These are, however, the most common and important means for enforcing business rules on the client. Whatever techniques you use to enforce constraints, keep the following in mind:

- You can always interrupt the default processing of a standard method by adding method code to it. For example, if you want to prevent certain users from opening a form, you can add the necessary method code to the **OpenWindow()** method of the form. In this code, you would then call the default processing (that is, opening the form) only if certain criteria are met.
- Many methods call each other in a specific sequence. As described earlier in this chapter, the **DeleteRow()** method calls **PreDeleteCheck()**, which in turn calls **PreDelete()**, which then calls **PostDelete()**. By preventing any of these methods from being called, you also stop the other methods later in the calling sequence from being triggered.

- 
- Remember the distinction between the local recordset and the records stored in the database. Many operations can be performed within the local recordset, performing the necessary checks before flushing the changes to the database.
  - You can add user-defined method and properties to enforce many constraints on the client. For example, if you create a user-defined property for specifying a security level needed to open a form, you can then control user access to forms by adding this property to all forms in an application.

## Using Database and Application Constraints Together

The *Getting Started with Oracle Power Objects* manual uses a demo application, “MoonLightDemo” (called “MLDEMO” in Windows), and a sample Blaze Database, “MoonLight Database” (called “MLDB” in Windows), to illustrate many development fundamentals. Among other things, the “MoonLightDemo” application and “MoonLight Database” demonstrate how to use server-based and client-based constraints in Oracle Power Objects.

This section describes how the demo application enforces constraints both in the application and in the database, and explains why each constraint appears on the client or the server.

### Server-Enforced Constraints

The following constraints are enforced within the tables defined in the “MoonLight Database”.

#### Creating Primary Keys

*What:* The primary key constraint appears on the ORDERS and ORDER\_ITEMS tables.

*Why:* The developer wanted to prevent data duplication by generating a unique ID for every order, and every line item in the order. When a column is designated as a primary key, data entered into it must be both unique and not null.

In addition, using primary keys makes it easier for the developer to establish master-detail relationships. Every detail record can be associated with only one master record, uniquely identified by the value in its primary key column.

### Ensuring that Values Always Appear in Some Columns

*What:* The NOT NULL constraint appears on several columns in the ORDERS table.

*Why:* The application also enforces this constraint, preventing the user from entering a new order that does not include data for these columns. However, the developer wanted to ensure that every row in ORDERS had data in these columns, regardless of the application used to add or update records in ORDERS. For sake of consistency, therefore, the developer added the NOT NULL constraint to these columns.

### Ensuring a Unique Name for Each Country

*What:* The NAME column in the COUNTRIES table has the UNIQUE constraint.

*Why:* The developer added this constraint to ensure that every country had a unique name associated with it in this table. Although each country has a unique numeric ID in the COUNTRIES table, the UNIQUE constraint was necessary so that the same name would not be accidentally entered for two countries. For example, if this UNIQUE constraint did not exist, users could enter "China" for both Taiwan and the People's Republic of China.

## Client-Enforced Constraints

The follow constraints are enforced within the "MoonLightDemo" application.

### Ensuring Values are Entered for Columns

*What:* In the frmOrders form, the user cannot save a new record until there are values entered for the entry date, ship date, representative, and company.

*Why:* Although the NOT NULL constraint already exists for the corresponding columns in the ORDERS table, the developer added this client-enforced constraint to ensure that the same business rule was enforced on the client. This may seem like duplication, but if network or server performance is an issue, it might be more efficient for the check to be performed once before sending the transaction to the server.

The following code appears in the `ValidateRow()` method of the frmOrders form to perform this check:

```
'Initialize the return value of ValidateRow()
ValidateRow = FALSE

'Determine whether any mandatory values were not entered
IF ISNULL(order_date.value) THEN
    MSGBOX "You must enter a valid order date."
ELSEIF ISNULL(ship_date.value) THEN
```

---

```

        MSGBOX "You must enter a valid ship date."
ELSEIF ISNULL(sales_rep.value) THEN
    MSGBOX "You must choose a sales representative."
ELSEIF ISNULL(company.value) THEN
    MSGBOX "You must identify the recipient company."

'Determine whether the ship date precedes the order date
ELSEIF ship_date.value < order_date.value THEN
    MSGBOX "The ship date cannot precede the order date."

'If the row met all the preceding criteria, it is valid
'and can be written to the recordset.
ELSE
    ValidateRow = TRUE
END IF

```

### Preventing a Ship Date Earlier Than an Apply Date

*What:* In the same form, the user cannot enter a ship date earlier than an apply date.

*Why:* Unless you write a trigger to compare dates on the server, this constraint must be enforced on the client. Again, it is more efficient from a client/server standpoint to perform this kind of preliminary check on the client, before sending the transaction across the network to the server.

The following code also appears in the `ValidateRow()` method of the `frmOrders` form, and is excerpted from the larger code sample shown above.

```

'Determine whether the ship date precedes the order date
ELSEIF ship_date.value < order_date.value THEN
    MSGBOX "The ship date cannot precede the order date."

```

### Preventing the User from Entering a Negative Quantity

*What:* In the Quantity text field of the `frmOrderItems` form, the user cannot enter a negative number for the quantity of an item.

*Why:* Again, it is more efficient to perform this check on the client, at the time the user enters the data. Otherwise, the user would be forced to reenter the transaction after the server rejected it.

The following code appears in the `Validate()` method of the field, where it enforces the constraint:

```

'If the new value is negative, do not accept it.
IF newval < 0 THEN
    msgBox "You cannot enter a negative quantity."
    Validate = FALSE
ELSE
    Validate = TRUE
END IF

```

### Preventing the User from Entering an Invalid Discount

*What:* In the Discount text field of the frmOrderItems form, the user cannot enter a negative discount, or one larger than 100%.

*Why:* This example further illustrates how to use the **Validate()** method to enforce a business rule as the user enters data, to improve the efficiency of the database application and the immediacy of the response.

The following method code appears in the **Validate()** method of the text field, where it enforces the constraint:

```
'If the new value is negative, do not accept it.
IF newval < 0 THEN
    MsgBox "You cannot enter a negative discount."
    Validate = FALSE
ELSE

'If the user typed a value of 1 or greater, convert it into
'a percentage
    IF newval >= 1 THEN
        newval = newval/100
        self.value = newval
    END IF

'If the user entered a value over 100%, do not accept it.
    IF newval > 1 THEN
        MsgBox "You cannot have a discount over 100%."
        Validate = FALSE
    ELSE
        Validate = TRUE
    END IF
END IF
```

### Managing Orders and Their Line Items

*What:* The application enforces referential integrity between order records and line item records in the following ways:

- If the user deletes an order, all of its associated line items are also deleted.
- The application prevents the user from changing the order ID of an order if it has associated line items.

---

*Why:* This kind of master-detail relationship is best maintained on the client, where the user will be deleting or modifying records. To enforce this constraint, the developer has set two properties of the frmOrders form to prevent the deletion:

<b>Property</b>	<b>Setting</b>
<b>LinkMasterDel</b>	<i>Delete Cascade</i> (delete all detail records when the master is deleted)
<b>LinkMasterUpd</b>	<i>Refuse If Children Present</i> (no updates to the primary key are possible if associated detail records exist)

### Creating Unique IDs for Orders and Line Items

*What:* The application generates a unique ID for every order and every line item.

*Why:* The application automates the task of creating a unique identifier for each record, so that problems do not later arise from data duplication or primary key/foreign key mismatches. The developer has created an automated counter by setting the following properties:

<b>Property</b>	<b>Setting</b>
<b>CounterType</b>	<i>Table, MAX()+CounterIncBy</i> (specifies an automated counter)
<b>CounterIncBy</b>	<i>1</i> (indicates that each new ID is one greater than the largest in use)
<b>CounterTiming</b>	<i>Immediate</i> (directs the application to generate a counter when the user creates a new record)
<b>CounterSeq</b>	<i>ORDER_ID_SEQ</i> (identifies the sequence in the “MoonLight Database” used for defining sequences).

For a complete explanation of these properties and of counters, see the section “Counter Fields” on page 19.17.



# A

---

## Suggested Coding Standards

This Appendix covers the following topics:

Overview .....	A.2
Text Conventions .....	A.2
Commenting Code .....	A.3
Declaration and Initialization .....	A.4
Constants .....	A.5
Naming Conventions .....	A.5
Object References .....	A.8

---

## Overview

This appendix presents a set of guidelines for use in developing Oracle Power Objects applications. You can follow these coding standards as presented, or modify them to suit your needs.

Following these standards can help improve your application in the following areas:

- Consistency
- Readability
- Usability

These guidelines are followed whenever possible in the code examples and sample applications that accompany Oracle Power Objects.

## Text Conventions

These text conventions can help improve the readability of your code.

### Keywords

The following text conventions apply to keywords in Oracle Basic method code:

<b>Type of keyword</b>	<b>Convention</b>	<b>Examples</b>
Oracle Basic commands, functions, and operators	Upper case	DIM MSGBOX NEW
Oracle Basic constants	Upper case	TRUE
Properties	Mixed case	TextJustVert ColorFill
Methods	Mixed case, followed by parentheses	Click() OpenModal(TRUE)
Oracle Basic Datatypes	Init cap	Long Object
Object Names	Lower case prefix, followed by mixed case	frmOrders btnOK

## Indentation

Normally, you should indent each level of code with a tab. However, when the number of levels makes it difficult to view the code within the code window, you can use space characters instead of tab characters to indent code levels.

## Line Length

Whenever possible, a line of method code should not exceed 60-80 characters. You can break up long lines of code by using the line continuation character (&) as described in the section “Line Continuation” on page 5.7.

Possible exceptions include:

- Calls to methods or DLL procedures that have many arguments.
- Declarations of DLL procedures that require a long line of text.
- Lines of SQL text in an EXEC SQL command or SQLLOOKUP function.

## Code Sections

Functionally distinct sections of code can be separated visually with a blank line.

## Commenting Code

As described in the section “Commenting Code” on page 5.8, comments can help improve the clarity and readability of your code. Comments can be included in two ways:

- For short comments that apply only to a single line of code, you can include the text to the right of the code to be commented.
- For longer comments, or comments that apply to more than one line of code, you can include the text on one or more separate lines above the code to be commented.

You should include code comments in the following locations:

- **At the beginning of the method** to describe what the method does.
- **At the beginning of each functionally distinct section of code** to describe what happens in the section.
- **When declaring a variable or constant** to describe its purpose, unless the purpose is immediately obvious.
- **When referring to a user-defined property or method** to describe the purpose of the property or method.

---

## Example

The following method code demonstrates the suggested use of comments:

```
Sub Click()  
    DIM vNumRows AS Long    ' The number of rows in the recordset  
  
    ' Requery the container, applying the condition entered by  
    ' the user in fldMyCond  
    Container.QueryWhere(fldMyCond.Value)  
  
    ' Get the number of rows in the recordset  
    vNumRows = Container.GetRecordSet().GetRowCount()  
  
    ' Loop through all the records in the recordset to build a  
    ' string, which is then applied to the popEmpNameList's  
    ' Translation property. The popup list then displays all the  
    ' employee names queried for the form.  
    DIM vTransStr AS String ' The new contents of the popup list  
    DIM vNameStr AS String  ' Holds the name of each employee  
    FOR x = 1 TO vNumRows  
  
        ' Read the employee name into vNameStr  
        vNameStr = Container.GetRecordSet().GetColVal("NAME")  
  
        ' Add the employee name to vTransStr. Note that the name  
        ' must be added twice, separated by an equal sign (=),  
        ' to satisfy the conventions of the Translation property  
        vTransStr = vTransStr & vNameStr & "=" & vNameStr  
  
    NEXT x  
  
    ' Assign the string to the Translation property  
    ' of popEmpNameList.  
    popEmpNameList.Translation = vTransStr
```

## Declaration and Initialization

By declaring and initializing variables and constants in a consistent location, you make it easier for readers to locate and, if necessary, modify this information. The following guidelines are suggested:

- **Global** variables and constants should be declared in the (Declarations) section of the application.

Global variables should be initialized in the **Initialize()** method of the application.

- **Local variables and constants** should usually be declared explicitly at the top of the method code.

For variables and constants that are used only in one section of method code, you can place the variable declaration at the beginning of the section in which the variable is used.

Variables to be used as “counters” (for example, in a FOR... NEXT loop) do not need to be explicitly declared.

## Constants

When possible, you should use constants in place of numeric values.

Oracle Basic includes predefined constants, described in Appendix B. These constants are used to represent the values of list-type properties, to represent the command codes of built-in menu commands, and to represent possible values for the parameters of methods such as `TBInsertButton()`.

You should also use constants when you need to reference numeric values throughout method code. For example, you could define the constants `STANDARD_WIDTH` and `EXPANDED_WIDTH` to be two standard widths of a form, in pixels. You could then use one of these constants in a line of code that sets the width of a form.

## Naming Conventions

By using a set of standard naming conventions for objects, variables, and user-defined property and method names, you improve the readability of your method code.

### Object Names

The following table summarizes recommended prefixes for each type of object.

<b>Object Type</b>	<b>Prefix</b>	<b>Example</b>
Applications (variable names)*	app	appMyApp
Bitmaps	bmp	bmpCompanyHQ
Charts	cht	chtRevenue
Check boxes	chk	chkOnHold
Embedded forms	emb	embCustomerInfo
Forms	frm	frmMain

---

<b>Object Type</b>	<b>Prefix</b>	<b>Example</b>
Horizontal scroll bars	hsb	hsbBrowser
Lines	lin	linClockHand
List boxes	lst	lstCountries
Menus	mnu	mnuEdit
Menu bars	mbr	mbrCustomMenu
OCX controls	ocx	ocxMediaControl
OLE objects	ole	oleWordDoc
Picture boxes	pic	picCompanyLogo
Popup lists	pop	popPaymentType
Pushbuttons	btn	btnCancel
Radio buttons	rad	radSendUSMail
Radio button frames	rbf	rbfShippingOptions
Recordsets	rec	recCustRecords
Repeater display	rep	repLineItems
Reports	rpt	rptWinners
Report groups	grp	grpDeptInfo
Static text objects	lbl	lblCustomerLabel
Status lines	sln	slnAppStatus
Text fields	fld	fldFirstName
Toolbars	tbr	tbrCustomToolbar
User-defined classes	cls	clsDBControls
Vertical scroll bars	vsb	vsbBrowser

\* In these cases, the prefix should be used only when assigning the application or session object to an object-style variable. For example:

```

DIM sesMySession AS Object
sesMySession = TESTSESS

```

## Subclass Names

When naming a user-defined class that is a subclass of another user-defined class, it is often desirable to indicate the relationship between master class and subclass in the object name.

To indicate this relationship, you can give the subclass the same name as the master class, adding a suffix that indicates the specific purpose of the subclass. The suffix should be separated from the rest of the name by an underscore character (\_).

For example, you might have a master class "clsAddress" that displays a set of address fields. You might then have several subclasses of this class that customize the label text to different languages.

You would give the subclass containing Spanish labels the following name:

```
clsAddress_Spanish
```

You would give the subclass containing Japanese labels the following name:

```
clsAddress_Japanese
```

## Variable Names

Variable names should be descriptive of their function. An exception to this guideline is variables used as counters, which are commonly given single-letter names such as *x* or *i*. (for example, FOR *x* = 1 TO 100).

You can use single-letter prefixes to indicate the scope of a variable (global or local). The following prefixes are recommended:

Type of variable	Prefix	Example
Global	g	gAppStatus
Local	v	vNumRows

➤ **Note:** Variables that store references to objects (such as recordsets, toolbars, and status lines) are sometimes clearer if named using the object prefixes described the section "Object Names" on page A.5. For example, a variable storing a reference to the "Edit" menu might be called `mnuEdit` rather than `vEditMenu`.

If you want the names of your variables to contain information about their datatype, you can use the native Oracle Basic datatype suffixes.

---

## User-Defined Property and Method Names

As with variable and constant names, the name of a user-defined property or method should be descriptive of its function.

You can use a three-letter prefix to distinguish user-defined properties and methods from standard properties and methods. The following prefixes are recommended:

Type of characteristic	Prefix	Example
User-defined property	udp	udpSecurityLevel
User-defined method	udm	udmCalculateTax

You may want to use the letter “p” as a prefix for the parameters of a user-defined method. This prefix will help distinguish parameter variables from other variables in the application. Using this convention, a user-defined method in the property sheet may look like this:

```
Sub udmDoTaxCalc(pTaxType as Long, pGrossAmt as Long)
```

## Object References

### Relative References

Relative references are described in the section “Relative References” on page 3.25.

In many cases, a relative object reference is preferable to an absolute reference.

- Your method code is less likely to break when you change the name of an object.
- Relative references indicate the relationship between objects in the containment hierarchy.
- Method code containing relative references is easier to reuse, because it does not have to be customized for each location where it is used.

For example, the following line of method code refers to an object by name:

```
vNumRows = frmMainForm.GetRecordSet().GetRowCount()
```

You could make this line of code more generic by replacing it with the following method code:

```
vNumRows = Self.GetContainer().GetRecordSet().GetRowCount()
```

### Recordsets

When referring to a recordset object, it is not always necessary to store a reference to the recordset in an Oracle Basic variable. If the recordset is referenced only once from within the method code, you can use the following syntax to refer to a characteristic of the recordset:

```
bound_container.GetRecordSet().property_or_method
```



For example, the following two code fragments return the number of rows in a container. However, the second code sample requires fewer lines of method code.

```
DIM recSet AS Object
DIM vNumRows AS Long
recSet = Self.GetContainer().GetRecordSet()
vNumRows = recSet.GetRowCount()
```

Alternative syntax:

```
DIM vNumRows AS Long
vNumRows = Self.GetContainer().GetRecordSet().GetRowCount()
```



# *B*

---

## List of Properties and Methods

This Appendix covers the following topics:

Overview .....	B.2
Standard Properties .....	B.2
Standard Methods .....	B.9

---

## Overview

This appendix lists all standard properties and methods in Oracle Power Objects. For a full description of any property or method, see the item's description in the On-Line Help.

## Standard Properties

The following table lists all standard properties in Oracle Power Objects. Properties marked with an "R" are readable at run time; properties marked with a "W" are writable at run time.

<b>Property</b>	<b>R</b>	<b>W</b>	<b>Description</b>
<b>Bitmap</b>	R	W	Determines which bitmap, if any, is displayed in an application object capable of displaying bitmaps.
<b>BitmapTiled</b>	R	W	Determines whether a bitmap displayed on an application object is tiled or not across the face of the object in which it appears.
<b>ChartAutoFormat</b>	R	W	Determines whether the chart control automatically sizes sections of the chart.
<b>ChartGap</b>	R	W	Determines the width (in pixels) between groups of bars in a horizontal bar or vertical bar chart.
<b>ChartLabelStyle</b>	R	W	Determines the labels appearing adjacent to each bar in a vertical or horizontal bar chart.
<b>ChartLegendHAlign</b>	R	W	Determines the horizontal position of the legend appearing for a chart.
<b>ChartLegendVAlign</b>	R	W	Determines the vertical position of the legend appearing for a chart.
<b>ChartLineStyle</b>	R	W	Determines the style of lines displayed in the chart control
<b>ChartMaxVal</b>	R	W	Determines the maximum value for the Y axis of a vertical bar, horizontal bar, or line graph.
<b>ChartMinVal</b>	R	W	Determines the maximum value for the Y axis of a vertical bar, horizontal bar, or line graph.
<b>ChartOverlap</b>	R	W	Determines the amount of overlap (in pixels) between multiple bars appearing for the same record in a vertical or horizontal bar chart.
<b>ChartPieCircle</b>	R	W	Determines whether a pie chart appears as a circle, or as an oval sized to fit within the vertical and horizontal dimensions of the chart control.

<b>Property</b>	<b>R</b>	<b>W</b>	<b>Description</b>
<b>ChartRowCount</b>	R	—	Determines the maximum number of records displays in a chart.
<b>ChartShowGrid</b>	R	W	Determines whether a grid appears within a chart.
<b>ChartShowLegend</b>	R	W	Determines whether a chart has a legend, providing a guide to each component of the chart.
<b>ChartStacked</b>	R	W	Determines whether a vertical or horizontal bar chart displays different sets of associated information as stacked sections of the same bar, or as separate bars.
<b>ChartStyle</b>	R	W	Determines the type of chart or graph to be displayed in a chart control.
<b>ChartXCol</b>	R	—	Identifies the column used to plot the X coordinates of a chart or graph.
<b>ChartYCols</b>	R	—	Identifies the columns used to plot the Y coordinates of a chart or graph. For vertical and horizontal bar charts, you can specify multiple columns, each displayed as a separate bar.
<b>ColorBrdr</b>	R	W	Determines the color of the border surrounding an object.
<b>ColorFill</b>	R	W	Determines an object's background color.
<b>ColorText</b>	R	W	Determines the color of text that appears within an object.
<b>CompareOnLock</b>	—	—	Determines whether the application checks to see if a record has been changed between (1) the time it was queried and (2) the time when the application attempts to lock the record.
<b>ConnectType</b>	R	—	Specifies how and when a database session object becomes active (establishes a connection to its database) during run time mode.
<b>ControlType</b>	R	—	Determines the class of an application object other than a session, recordset, or the application itself. In other words, <b>ControlType</b> identifies the class of an object than can be visible within the application (that is, containers, static objects, and controls).
<b>CounterIncBy</b>	R	—	For a text field object whose <b>CounterType</b> property is set to "Table, MAX( )+CounterIncBy", determines the amount added to the current column maximum to generate the new value.
<b>CounterSeq</b>	R	—	For a text field object whose <b>CounterType</b> property is set to "Sequence", designates the sequence object from which new values are selected.
<b>CounterTiming</b>	R	—	Determines when a new unique value is generated for a "counter" text field object.

---

<b>Property</b>	<b>R</b>	<b>W</b>	<b>Description</b>
<b>CounterType</b>	R	—	Determines whether a text field object is a “counter” field, which automatically receives a unique value when a new row is inserted into the field’s container. This unique value can be generated by one of several techniques:
<b>DataSize</b>	R	—	Sets aside a number of bytes for the value held in a control, if the control uses the string datatype.
<b>DataSource</b>	R	—	Determines how the application populates a control with data.
<b>Datatype</b>	R	—	Determines the type of data that can be stored in the control.
<b>DefaultButton</b>	R	—	Indicates which pushbutton on a form or user-defined class is the default button.
<b>DefaultCondition</b>	R	W	Determines the range of records queried for a bound container, by setting the WHERE clause of the query that fetches the container’s recordset.
<b>DefaultSession</b>	—	—	Identifies the default session for an application object.
<b>DefaultValue</b>	R	W	Assigns a default value to a control.
<b>DesignConnect</b>	R	—	Contains the connect string used to make a database session object active when the developer double-clicks on the Connector control in a Database Session window.
<b>DesignRunConnect</b>	R	—	Contains the connect string used to make a database session object active from design run time mode.
<b>Direction</b>	R	W	Determines the direction of a line.
<b>DrawStyle</b>	—	—	Determines whether the control uses the sculpted, 3-D look and feel.
<b>Enabled</b>	R	W	Determines whether the user can interact with the control.
<b>FirstChild</b>	R	—	Indicates the first object contained by the current object.
<b>FirstPgFtr</b>	—	—	Determines whether the page footer area should appear on the first page of a report when the report is previewed or printed.
<b>FirstPgHdr</b>	—	—	Indicates whether the page header area should appear on the first page of a report when the report is previewed or printed.
<b>FontBold</b>	R	W	Determines whether the text appearing in the control is boldfaced.
<b>FontItalic</b>	R	W	Determines whether the text appearing in the control is italic.

<b>Property</b>	<b>R</b>	<b>W</b>	<b>Description</b>
<b>FontName</b>	R	W	Determines the font used for any text appearing within an object.
<b>FontSize</b>	R	W	Determines the point size for any text that appears within the control.
<b>FontUnderline</b>	R	W	Determines whether the text appearing in the control is underlined.
<b>FormatMask</b>	R	W	Determines how data appears in a control is formatted when displayed.
<b>GroupCol</b>	—	—	Determines the database column used to group records in a report.
<b>HasBorder</b>	R	W	Indicates whether a control has a border.
<b>HasExtraRow</b>	R	W	Determines whether a bound container displays a blank row for data entry after the last row containing data.
<b>HasScrollBar</b>	R	—	Determines whether the control has a vertical scroll bar on its right side.
<b>HelpText</b>	R	W	Defines the summary help associated with an object.
<b>HelpTextVisible</b>	R	W	Controls whether the system automatically displays Summary help information in the Summary panel. <b>HelpTextVisible</b> is True by default; to disable the display of Summary help, set <b>HelpTextVisible</b> to False.
<b>IsDismissBtn</b>	R	W	Determines whether a pushbutton can, by default, close a modal form or user-defined class.
<b>Label</b>	R	W	Determines the label that appears on or next to a control, for a particular menu or menu command, within a static text object, or on the title bar of a form, report, or user-defined class.
<b>LastPgFtr</b>	—	—	Indicates whether the page footer area should appear on the last page of a report when the report is previewed or printed.
<b>LinkDetailColumn</b>	R	—	Identifies the name of a column used to link a detail recordset to a master recordset, when a master-detail relationship exists.
<b>LinkMasterColumn</b>	R	—	Identifies a column used to link a master recordset to a detail recordset.
<b>LinkMasterDel</b>	R	—	Determines what happens to detail (foreign key) records when you delete a master (primary key) record.

---

<b>Property</b>	<b>R</b>	<b>W</b>	<b>Description</b>
<b>LinkMasterForm</b>	R	—	Identifies the name of the form, class, report, or repeater display that contains the master records in a master-detail relationship.
<b>LinkMasterUpd</b>	R	—	Determines what happens to detail (foreign key) records when you update a master (primary key) record in a master-detail relationship.
<b>LinkPrimaryKey</b>	R	—	Indicates whether the control holding primary key values is located on the master container or the detail container in a master-detail relationship.
<b>MultiLine</b>	—	—	Controls whether a text field can display multiple lines of text.
<b>Name</b>	R	—	Assigns a name to an object for later reference in properties and methods. For database objects, defines the SQL name of the object.
<b>ObjectType</b>	R	—	Determines the class of an object.
<b>OrderBy</b>	R	W	Determines the order of rows in a recordset when the rows are queried from a database.
<b>PageOnBreak</b>	—	—	Indicates whether the report should begin a new page whenever it begins a new report group when the report is previewed or printed.
<b>PositionX</b>	R	W	Determines the horizontal position, measured in pixels, of an object, from the left edge of its container.
<b>PositionY</b>	R	W	Determines the vertical position, measured in pixels, of an object, from the top edge of its container.
<b>ReadOnly</b>	R	W	Determines whether the user can enter a value into a control.
<b>RecordSource</b>	R	—	Names the record source (a table or view) for a bound container. The <b>RecordSource</b> and <b>RecSrcSession</b> properties define a recordset object, which maps to a table or view through a session.
<b>RecSrcSession</b>	—	—	Names the session through which the bound container accesses its record source (a table or view). The session must be defined as an object in Oracle Power Objects.
<b>RecSrcAddCols</b>	—	—	The names of additional database columns associated with the container but not associated with controls in the container. Multiple columns are separated by commas.

---



<b>Property</b>	<b>R</b>	<b>W</b>	<b>Description</b>
<b>RecSrcMaxMem</b>	R	—	Sets the limit on the amount of memory available to a recordset before the application writes it to a temporary file on the client system.
<b>RowFetchMode</b>	R	W	Determines the amount or records queried for a bound container, and when the records are queried.
<b>RunConnect</b>	R	W	Contains the connect string used to make a database session object active from standalone run time mode.
<b>ScrollAmtLine</b>	R	W	Determines the number of records the user moves through when pressing the arrow at either end of a scroll bar set up as a database navigation tool.
<b>ScrollAmtPage</b>	R	W	Determines the number of records moved when the user clicks on the body of a scroll bar, if the scroll bar is set up as a database navigation tool.
<b>ScrollMax</b>	R	W	Assigns a value to the top end of a scroll bar's scroll range.
<b>ScrollMin</b>	R	W	Assigns a value to the bottom end of a scroll bar's scroll range.
<b>ScrollObj</b>	R	W	Determines which container, if any, has its current record changed when you use the scroll bar.
<b>ScrollPos</b>	R	W	Indicates the current position of the thumb on a scroll bar.
<b>ScrollWithRow</b>	R	—	Determines whether an unbound control in a bound container holds one value for each row in the recordset, or contains a single constant value for all rows in the recordset.
<b>SizeX</b>	R	W	Sets the width of an object, as measured in pixels.
<b>SizeY</b>	R	W	Sets the height of an object, as measured in pixels.
<b>TabEnabled</b>	R	W	Determines whether the control is part of the tab order on a container.
<b>TabOrder</b>	R	W	Determines the position of the object in the tab order on a container.
<b>TextJustHoriz</b>	R	W	Determines the horizontal alignment of text within a control.
<b>TextJustVert</b>	R	W	Determines the vertical alignment of text within a control.
<b>Translation</b>	R	W	Determines the mapping between a control's display value and its internal value.

---

<b>Property</b>	<b>R</b>	<b>W</b>	<b>Description</b>
<b>ValidateRowMsg</b>	R	W	Sets the message to be displayed when the <b>ValidateRow()</b> method returns <b>False</b> (that is, a row-level validation failed), indicating that changes to the current record were not saved to the database.
<b>ValidateMsg</b>	R	W	Sets the message to be displayed when the <b>Validate()</b> method indicates that the validation fails by returning <b>False</b> .
<b>Value</b>	R	W	The current value held in a control.
<b>ValueList</b>	R	W	Determines the contents of the popup list appearing next to a combo box.
<b>ValueOff</b>	R	W	Assigns a value to a check box when it is unchecked, or a radio button when it is not selected.
<b>ValueOn</b>	R	W	Assigns a value to a check box when it is checked, or a radio button when it is selected.
<b>Visible</b>	R	W	Determines whether an object is visible at run time.
<b>WinInitPos</b>	R	W	Determines the position of a window when first displayed.
<b>WinPositionX</b>	R	W	Specifies the current horizontal position of a form.
<b>WinPositionY</b>	R	W	Specifies the current vertical position of a form.
<b>WinSizeX</b>	R	W	Determines the horizontal size of the window surrounding a form or report.
<b>WinSizeY</b>	R	W	Determines the vertical size of the window surrounding a form or report.
<b>WindowStyle</b>	R	W	Determines the appearance and behavior of the window surrounding a form.

## Standard Methods

<b>Method</b>	<b>Description</b>
<b>AddColumn()</b>	Adds a column to an unbound recordset.
<b>AppendMenu()</b>	Appends a menu to the end of the menu bar. The menu is inserted before any system default menus that appear on the menu bar (such as the Help menu in Windows).
<b>AppendMenuItem()</b>	Appends an item to the end of the menu. You must specify the item's label, a command code, a help context, and a keyboard equivalent.
<b>CanPasteFromClipboard()</b>	Returns True if an OLE data object is in the Clipboard, or False if there is not.
<b>ChgCurrentRec()</b>	Called to move the pointer to a different record in a container's recordset.
<b>ChildClick()</b>	Sent to a container when the user clicks on an object within that container.
<b>ChildDbClick()</b>	Triggered on a container when the user double-clicks on an object within the container.
<b>ChildPostChange()</b>	Triggered on a container when the internal value of any control within the container changes.
<b>ChildPreChange()</b>	Called before the user edits any control within a container.
<b>ClearMenuBar()</b>	Removes all menus from the menu bar. However, the menu objects are not deleted from the system.
<b>ClearStatusLine()</b>	Deletes all panels from the status line except the Summary panel, which cannot be deleted.
<b>ClearToolbar()</b>	Deletes all buttons from the toolbar.
<b>Click()</b>	Called when a user presses the mouse button (the left button, for Windows) while the cursor is above an object.
<b>CloseApp()</b>	Closes an application.
<b>CloseWindow()</b>	Removes a form from memory. Called automatically when the user closes the form.
<b>CommitForm()</b>	Commits all pending transactions associated with a bound form, including inserts, deletions, and updates. The transaction set includes transactions entered in containers within the form, such as repeater displays and embedded forms.

---

<b>Method</b>	<b>Description</b>
<b>CommitWork( )</b>	Commits the current transactions associated with an active database session object.
<b>Connect( )</b>	Makes a database session object active by establishing a connection to a database.
<b>CopyColFrom( )</b>	Copies the source column of the source recordset object to the destination column in this recordset object.
<b>CopyToClipboard( )</b>	Copies the OLE data object stored in the OLE control to the clipboard.
<b>CounterGenKey( )</b>	Determines how unique values are generated for a text field object whose <b>CounterType</b> property is set to "User Generated". This determination is made through method code added to <b>CounterGenKey( )</b> .
<b>DefaultMenuBar( )</b>	Initializes a menu bar with the system default menus and application default menus appropriate to the form or report. This method deletes any existing menus before initializing the menu bar.
<b>DefaultToolbar( )</b>	Initializes a toolbar with the application default buttons appropriate to the form or report. This method deletes any existing buttons before initializing the toolbar.
<b>DeleteAllMenus( )</b>	Removes all menus from the menu bar and deletes the menu objects.
<b>DeleteMenuItem( )</b>	Deletes an item from a specified position in the menu.
<b>DeleteRow( )</b>	Called to delete the current row in a container's recordset.
<b>DeleteStatusPanel( )</b>	Deletes a panel from a specified position in the status line. The Summary panel cannot be deleted.
<b>Disconnect( )</b>	Makes a database session object inactive by terminating the connection to a database.
<b>DismissModal( )</b>	Called to dismiss an application-modal or system-modal form, removing it from the screen (but not from memory).
<b>DoCommand( )</b>	Performs an action when the user clicks a toolbar button or selects a menu command.
<b>DoubleClick( )</b>	Called when the user double-clicks on an object.
<b>FetchAllRows( )</b>	Fetches all rows in a database into a recordset.
<b>FetchToRow( )</b>	Fetches rows in a database, up to and including a specified row, and includes them in a recordset.

<b>Method</b>	<b>Description</b>
<b>FocusEntering( )</b>	Called when the focus moves into a control.
<b>FocusLeaving( )</b>	Called when the focus moves out of a control.
<b>ForceUpdate( )</b>	Instructs the application to redraw a form or report.
<b>GetBindColumn( )</b>	Returns the column number in a bound container's recordset of the column to which a control is bound.
<b>GetColCount( )</b>	Returns the number of columns in the record set.
<b>GetColName( )</b>	Returns the name of the column in a recordset whose column number is passed to the method. Columns are incremented from 1.
<b>GetColNum( )</b>	Returns the column number of a column in a recordset whose column name is passed to the method. Column numbers are incremented from 1.
<b>GetColVal( )</b>	Returns the value for the current record in the recordset held in the specified column.
<b>GetContainer( )</b>	Returns an object reference to the container in which the application object appears.
<b>GetCurRow( )</b>	Returns the current row in the record set.
<b>GetFirstForm( )</b>	Returns an object reference to the first form or report found in an application.
<b>GetFocus( )</b>	Returns an object reference to the application object that currently has the focus.
<b>GetFormByName( )</b>	Returns an object reference to a form or report, identified by name.
<b>GetItemCount( )</b>	Returns a count of all items in the menu, including both commands and separator lines.
<b>GetMenu( )</b>	Returns a reference to a specified menu object in the menu bar.
<b>GetMenuBar( )</b>	Returns a reference to the menu bar associated with the form or report.
<b>GetMenuCount( )</b>	Returns a count of all menus in the menu bar.
<b>GetMenuItem( )</b>	Returns a specified piece of information about a specified item in the menu. You can get the item's label, the command code, the help context, or the keyboard equivalent.

---

<b>Method</b>	<b>Description</b>
<b>GetNextForm()</b>	Returns an object reference to the next form or report found in an application.
<b>GetRecordset()</b>	Identifies a recordset object associated with a bound container or bound list control. After assigning the recordset to an object-type variable, you can read and modify the contents of the recordset programmatically, and you can access the session associated with the recordset.
<b>GetRowCount()</b>	Returns the number of rows in a recordset.
<b>GetRowCountAdvice()</b>	Returns an estimated number of rows that the application will query from the database.
<b>GetRowStat()</b>	Called to detect the status of the current row in a recordset.
<b>GetSession()</b>	Returns a reference to the database session object associated with a recordset object (normally associated with a bound container).
<b>GetStatCount()</b>	Returns a count of all panels in the status line.
<b>GetStatPanel()</b>	Returns a specified piece of information about a specified panel in the status line. You can get the text currently displayed in the panel, the panel's width, the command code, or the message strings associated with the panel's status.
<b>GetStatusLine()</b>	Returns a reference to the status line associated with the form or report.
<b>GetToolBar()</b>	Returns a reference to the toolbar associated with a form or report.
<b>GetTopContainer()</b>	Returns an object reference to the top-level container in the object containment hierarchy (that is, the form, report, or user-defined class in which the object resides).
<b>GetWindowHandle()</b>	Returns an operating system handle to a window in an Oracle Power Objects application.
<b>GoNxtLine()</b>	Moves the current row of the container's recordset forward one row.
<b>GoNxtPage()</b>	Moves the current row of the container's recordset forward one "page" of rows. A page is the number of rows displayed on the container at one time. For containers such as forms, embedded forms, and user-defined classes, a page is one row.
<b>GoPos()</b>	Moves the current row of the container's recordset to a numerically specified row.
<b>GoPrvLine()</b>	Moves the current row of the container's recordset back one row.

<b>Method</b>	<b>Description</b>
<b>GoPrvPage()</b>	Moves the current row of the container's recordset back one "page" of rows.
<b>HideWindow()</b>	Called to hide a form, but keep it in memory.
<b>Initialize()</b>	Called when the application launches, as well as when a single form or report runs in Design Run-time mode.
<b>InitializeWindow()</b>	Called to create an associated menu bar, toolbar, and status line when the form or report is loaded into memory.
<b>InsertMenu()</b>	Inserts a menu at a specified position in the menu bar.
<b>InsertMenuItem()</b>	Inserts an item at a specified position in the menu. You must specify the item's label, a command code, a help context, and a keyboard equivalent.
<b>InsertRow()</b>	Called to insert a new row into a recordset.
<b>InsertStatusPanel()</b>	Inserts a panel at a specified position in the status line. You must specify the panel's width and the maximum message length that can be displayed in the panel.
<b>IsConnected()</b>	Indicates whether a database session object is currently active (connected to a database).
<b>IsWorkPending()</b>	Indicates whether any changes have been made to the recordset objects associated with a database session since the recordsets were fetched from the database.
<b>LastWindowClosed()</b>	Called when the user closes the last form in the application, but before the application itself closes.
<b>LinkChgCurrentRec()</b>	Called when a linked master recordset changes its current record, to update the rows appearing in the detail recordset. Triggered on the bound container holding detail records.
<b>LinkPostDelete()</b>	Called after you have deleted a master record in a master-detail relationship. Triggered on the bound container holding detail records.
<b>LinkPostInsert()</b>	Called after you insert a new record in a master-detail relationship. Triggered on the bound container holding detail records.
<b>LinkPostUpdate()</b>	Called after you edit a master record in a master-detail relationship. Triggered on the bound container holding detail records.

---

<b>Method</b>	<b>Description</b>
<b>LinkPreDelete()</b>	Called before you delete a master record in a master-detail relationship. Triggered on the bound container holding detail records.
<b>LinkPreInsert()</b>	Called when the user inserts a master record in a master-detail relationship. Triggered on the bound container holding detail records.
<b>LinkPreUpdate()</b>	Called when you edit a master record in a master-detail relationship. Triggered on the bound container holding detail records.
<b>LinkPreDeleteCheck()</b>	Called to perform a check before deleting a master record in a master-detail relationship. Triggered on the bound container holding detail records.
<b>LinkPreInsertCheck()</b>	Called to perform a check before inserting a master record in a master-detail relationship. Triggered on the bound container holding detail records.
<b>LinkPreUpdateCheck()</b>	Called to perform a check before updating a master record in a master-detail relationship. Triggered on the bound container holding detail records.
<b>LockRow()</b>	Called to lock the current row.
<b>MouseDown()</b>	Called when the user presses the mouse button within the area of an object.
<b>MouseMove()</b>	Called when the user moves the mouse while the mouse button is depressed. <b>MouseMove()</b> is triggered every time the cursor moves one pixel across the object where the method is triggered.
<b>MouseUp()</b>	Called when the user releases the mouse button. This method is triggered on the same object where <b>MouseDown()</b> was originally triggered.
<b>NextControl()</b>	Cycles programmatically through all the objects within a container.
<b>OMAMsgRecvd()</b>	Called when the application receives a message from Oracle Mobile Agents.
<b>OMAShutdown()</b>	Called to shut down the Oracle Mobile Agents message manager.
<b>OnLoad()</b>	Called when the application loads a form or report into memory, or when the application opens. The <b>OnLoad()</b> method is triggered on a container and all objects within it, when the container is loaded into memory.



<b>Method</b>	<b>Description</b>
<b>OnQuery()</b>	Called after the application performs a query for a bound container. During <b>OnQuery()</b> , all bound controls on the container refresh to reflect the new values returned from the query.
<b>OpenModal()</b>	Called when opening a modal form.
<b>OpenPreview()</b>	Called to open a form or report to see how it will look when printed (that is, displays the container in Print Preview mode).
<b>OpenPrint()</b>	Called to print a form or report.
<b>OpenWindow()</b>	Called to load a form into memory and display it in the screen.
<b>PasteFromClipboard()</b>	Pastes an OLE data object from the clipboard into the OLE control.
<b>PostChange()</b>	Called when the user changes the value assigned to an application object.
<b>PostDelete()</b>	Called immediately after a record is deleted.
<b>PostInsert()</b>	Called after a record is inserted.
<b>PostUpdate()</b>	Called after a record is updated.
<b>PreChange()</b>	Called before the user begins editing a record.
<b>PreDelete()</b>	Called before a record is deleted, but after the decision to delete the record has been made.
<b>PreInsert()</b>	Called when a record is inserted.
<b>PreUpdate()</b>	Called when a record is updated.
<b>PreDeleteCheck()</b>	Called before the decision to delete a record has been made.
<b>PreInsertCheck()</b>	Called before the decision to insert a record has been made.
<b>PreUpdateCheck()</b>	Called before the decision to update a record has been made.
<b>Query()</b>	Called to fetch records from the database to populate a recordset object. Recordset objects are normally associated with a bound container or a list control.
<b>QueryWhere()</b>	Queries records for a bound container or translation control, applying a condition to the query. The condition passed to <b>QueryWhere()</b> replaces the <b>DefaultCondition</b> set for the container or control.
<b>QueryMasters()</b>	Recursively queries all containers holding master records.

---

<b>Method</b>	<b>Description</b>
<b>ReadColFromFile( )</b>	Reads a value from a file and writes it into a recordset. The value can be of any valid datatype.
<b>RemoveMenu( )</b>	Removes a menu from a specified position in the menu bar. However, the menu object is not deleted.
<b>RevertRow( )</b>	When a validation fails, restores the original values for all columns within a row before the record was updated.
<b>RevertValue( )</b>	When a validation fails, replaces the value entered with the original value in the control.
<b>RollbackForm( )</b>	Called to roll back all pending transaction associated with a bound container, including transactions initiated in other bound containers within the container.
<b>RollbackWork( )</b>	Rolls back the current transaction associated with an active session object, discarding any changes that have been made by the user since the transaction began.
<b>SetColVal( )</b>	Assigns a new value to the specified column. The change is applied to the current record in the recordset.
<b>SetCurRow( )</b>	Sets the current row within the recordset to a specified row number.
<b>SetCursor( )</b>	Changes the appearance of the cursor.
<b>SetFocus( )</b>	Moves the focus to a control.
<b>SetMenuBar( )</b>	Assigns a custom menu bar to a form or report.
<b>SetMenuItem( )</b>	Modifies a specified piece of information about a specified item in the menu. You can set the item's label, the command code, the help context, or the keyboard equivalent.
<b>SetQuery( )</b>	Defines a query that binds a recordset to a record source (a table or view).
<b>SetStatDispList( )</b>	Designates a status line panel to be updated automatically. You must specify the panel's command code and the message strings associated with the panel's status.
<b>SetStatusPanelMsg( )</b>	Sets the text displayed in a specified panel of the status line.
<b>SetStatusLine( )</b>	Assigns a custom status line to a form or report.
<b>SetToolBar( )</b>	Assigns a custom toolbar to a form or report.

<b>Method</b>	<b>Description</b>
<b>ShowWindow()</b>	Called to display a hidden window (that is, a window held in memory but not visible).
<b>SysDefaultMenuBar()</b>	Initializes a menu bar with the system default menus. This method deletes any existing menus before initializing the menu bar.
<b>SysDefaultStatusLine()</b>	Initializes a status line with the system default panels. This method deletes any existing panels before initializing the status line.
<b>TBAppendButton()</b>	Appends a button to the end of the toolbar. You must specify the button's command code, a bitmap, a button style, and a help context.
<b>TBDeleteButton()</b>	Deletes a button from a specified position in the toolbar.
<b>TBGetButton()</b>	Returns a specified piece of information about a specified button in the toolbar. You can get the button's command code, the bitmap, the button style, or the help context.
<b>TBGetCount()</b>	Returns a count of all buttons in the toolbar, including separator buttons.
<b>TBInsertButton()</b>	Inserts a button at a specified position in the toolbar. You must specify the button's command code, a bitmap, a button style, and a help context.
<b>TBSetButton()</b>	Modifies a specified piece of information about a specified button in the toolbar. You can set the button's command code, the bitmap, the button style, or the help context.
<b>TestCommand()</b>	Checks to see whether the user has selected a menu command or pushed a toolbar button.
<b>UpdateList()</b>	Called to refresh the contents of a list when the recordset of a list box, combo box, or popup list changes.
<b>Validate()</b>	Called to enforce business rules on data in a control.
<b>ValidateRow()</b>	Called to enforce business rules on an inserted or updated record.
<b>WriteColToFile()</b>	Writes a value to a file from a recordset. The value can be of any valid datatype.



# C

---

## Constants and Reserved Words

This Appendix covers the following topics:

Oracle Basic Constants .....	C.2
Oracle Basic Reserved Words .....	C.4

---

## Oracle Basic Constants

CMD_ABOUT	DATATYPE_DATE
CMD_APPQUERY	DATATYPE_DOUBLE
CMD_CLEAR	DATATYPE_FLOAT
CMD_CLOSE	DATATYPE_INTEGER
CMD_COMMIT	DATATYPE_LONG_INTEGER
CMD_COPY	DATATYPE_STRING
CMD_CUT	DIRECTION_LOWER_LEFT_TO_UPPER_RIGHT
CMD_DELETETROW	DIRECTION_UPPER_LEFT_TO_LOWER_RIGHT
CMD_FIRSTUSERCOMMAND	
CMD_FULLPAGE	FALSE
CMD_HELP	
CMD_HELPONHELP	LINKMASTERDEL_CASCADE
CMD_INSERTOBJECT	LINKMASTERDEL_ORPHAN
CMD_INSERTROW	LINKMASTERDEL_REFUSE
CMD_NEW	LINKMASTERUPD_CASCADE
CMD_NEWWINDOW	LINKMASTERUPD_ORPHAN
CMD_NEXTPAGE	LINKMASTERUPD_REFUSE
CMD_OPEN	
CMD_PASTE	MENUPART_ACCEL
CMD_PASTESPECIAL	MENUPART_COMMAND
CMD_PREVPAGE	MENUPART_HELP
CMD_PRINT	MENUPART_LABEL
CMD_PRINTPREVIEW	
CMD_PRINTSETUP	PRIMARYKEY_HERE
CMD_QBF	PRIMARYKEY_ON_MASTER
CMD_QFFRUN	
CMD_QUIT	RECDTY_DATE
CMD_REDO	RECDTY_DOUBLE
CMD_ROLLBACK	RECDTY_INTEGER
CMD_RUNSTOP	RECDTY_LONG
CMD_SAVE	RECDTY_STRING
CMD_SAVEAS	ROWFETCHMODE_FETCH_ALL_IMMEDIATELY
CMD_UNDO	ROWFETCHMODE_FETCH_AS_NEEDED
COUNTERTIMING_DEFERRED	ROWFETCHMODE_FETCH_COUNT_FIRST
COUNTERTIMING_IMMEDIATE	
COUNTERTYPE_NONE	STATUSLINEPART_COMMAND
COUNTERTYPE_SEQUENCE	STATUSLINEPART_MSG_CHECKED
COUNTERTYPE_TABLE_MAX_COUNTERINCBY	STATUSLINEPART_MSG_DISABLED
COUNTERTYPE_USER_GENERATED	STATUSLINEPART_MSG_DISABLED_CHECKED
	STATUSLINEPART_MSG_ENABLED
	STATUSLINEPART_TEXT
	STATUSLINEPART_WIDTH

*Continued on Next Page*

TESTCOMMAND\_CHECKED  
TESTCOMMAND\_DISABLED  
TESTCOMMAND\_DISABLED\_CHECKED  
TESTCOMMAND\_ENABLED  
TEXTJUSTHORIZ\_CENTER  
TEXTJUSTHORIZ\_LEFT  
TEXTJUSTHORIZ\_RIGHT  
TEXTJUSTVERT\_BOTTOM  
TEXTJUSTVERT\_CENTER  
TEXTJUSTVERT\_TOP  
TOOLBARPART\_BITMAP  
TOOLBARPART\_COMMAND  
TOOLBARPART\_HELP  
TOOLBARPART\_STYLE  
TOOLBARSTYLE\_PUSHBTN  
TOOLBARSTYLE\_SEPARATOR  
TOOLBARSTYLE\_TOGGLE  
TRUE

WINDOWSTYLE\_ALTERNATE\_DIALOG  
WINDOWSTYLE\_DOCUMENT\_WITHOUT\_MAXIMIZE  
WINDOWSTYLE\_FIXED  
WINDOWSTYLE\_MOVABLE\_DIALOG  
WINDOWSTYLE\_PALETTE  
WINDOWSTYLE\_PALETTE\_WITH\_CLOSE\_BOX  
WINDOWSTYLE\_PLAIN\_DIALOG  
WINDOWSTYLE\_STANDARD\_DIALOG  
WINDOWSTYLE\_STANDARD\_DOCUMENT

---

## Oracle Basic Reserved Words

ABS	ELSE	KILL	PAGENUM	SYD
ACCESS	ELSEIF		PMT	SYSDATE
ALIAS	END	LBOUND	PPMT	SYSTEMNAME
AND	ENVIRON	LCASE	PRESERVE	
APPEND	EOF	LEFT	PRINT	TAB
AS	EQV	LEFTB	PRIVATE	TAN
ASC	ERASE	LET	PUT	THEN
ATN	ERL	LIB	PV	TIME
	ERR	LINE		TIMER
BEEP	ERROR	LOC	RANDOM	TIMESERIAL
BINARY	EXEC	LOCK	RANDOMIZE	TIMEVALUE
BYVAL	EXIT	LOF	RATE	TO
	EXP	LOG	READ	TRIM
CALL		LONG	REDIM	
CASE	FIX	LOOP	RESET	UBOUND
CDBL	FOR	LTRIM	RESUME	UCASE
CHDIR	FORMAT		RETURN	UNLOCK
CHDRIVE	FREEFILE	MID	RIGHT	UNTIL
CHOOSE	FUNCTION	MIDB	RIGHTB	
CHR	FV	MINUTE	RMDIR	VAL
CINT		MIRR	RND	VARIANT
CLNG	GET	MKDIR	RTRIM	VARTYPE
CLOSE	GLOBAL	MOD		
CONST	GO	MONTH	SECOND	WEEKDAY
COS	GOSUB	MSGBOX	SELECT	WEND
CSNG	GOTO		SGN	WHILE
CSTR		NEW	SHARED	WIDTH
CURDIR	HEX	NEXT	SIN	WRITE
CURRENCY	HOUR	NOT	SINGLE	
CVDATE		NOW	SLN	XOR
	IF	NPER	SPACE	
DATE	IIF	NPV	SPC	YEAR
DATEADD	IMP	NULL	SQLERRCLASS	
DATEDIFF	INPUTBOX	NVL	SQLERRCODE	
DATEPART	INSTR		SQLERRTEXT	
DATESERIAL	INSTRB	OBJECT	SQLLOOKUP	
DATEVALUE	INT	OCT	SQLROWCOUNT	
DAY	INTEGER	ON	SQL	
DDB	IPMT	OPEN	STATIC	
DECLARE	IRR	OR	STEP	
DELETE	IS	OUTPUT	STOP	
DIM	ISDATE		STR	
DO	ISNULL		SUB	
DOUBLE	ISNUMERIC		SWITCH	



# Index

---

## A

- Aggregation functions
  - using Oracle Basic 10.11
  - using the DataSource property 10.11
- Alias 8.29
- ALTER privilege 8.5
- Application constraints. *See* Constraints, application
- Application development 1.2, 1.4, 1.8 to 1.12
- Application objects 3.2, 3.4, 10.1 to 10.39
  - binding 17.1 to 17.34
  - binding graphically 17.4
  - binding manually 17.8
  - bitmap 3.4, 10.5
  - categories of 2.8, 10.4
  - class 3.4
  - connecting to a database with EXEC SQL 17.4
  - connecting to a database with SQLLOOKUP 17.4
  - container 10.4
  - control 10.4
  - copying 2.11
  - creating 2.10, 2.37, 10.3
  - deleting 2.10
  - editing properties and methods 2.9
  - embedded form 10.8
  - form 3.4, 10.7, 11.1 to 11.20, 14.46, 14.47
  - limitations in reports 12.6
  - methods 14.46, 14.48
  - moving 2.11, 2.37
  - names 3.21
  - OLE data object 3.4, 10.5
  - opening 2.11
  - repeater display 10.9
  - report 3.4, 10.8, 12.1 to 12.20
  - report. *See also* Reports
  - resizing 2.37
  - static 10.4
  - user-defined class 10.5
- Application window 2.3, 2.8 to 2.13
  - tasks performed in 2.9
- Application-modal forms 11.11
- Applications 3.3, 3.9
  - exporting to flat files 2.12
  - filenames 10.2
  - importing from earlier versions 2.12
  - naming rules 10.3
  - opening 3.9

- Arguments
  - of methods 5.3, 5.7
- Arithmetic operators 4.13
- Arrays 4.9

## B

- Binary operators 9.10
- Bind variable arrays 9.18
- Bind variables 9.16
  - and the SQLLOOKUP function 9.22
  - datatypes 9.17
  - declaration of 9.17
  - where used 9.16
- Bindable containers 1.6, 8.23
- Bindable controls. *See* Controls, bindable
- Binding 17.2
  - controls to columns in reports 12.8
- Bitmaps 2.12, 3.4, 10.5
  - exporting to a .BMP file 2.13
  - importing to an application 2.12
  - viewing 2.13
- Bitwise operations 4.15
- Blaze databases 6.20 to 6.21, 7.1 to 7.8
- Bound containers 1.6, 17.3
  - and recordsets 17.9
  - and report group headers 12.16
  - associating 17.3
- Bound controls 1.6
- Breakpoints
  - removing 5.13
  - setting 5.12
- Browser window 2.2
- Business rules. *See* Constraints
- Buttons
  - See* Pushbuttons
  - See* Toolbar buttons

## C

- Calling methods 3.20
- Chart controls 3.6, 10.16
- Charts
  - in report groups 12.18
  - in report records 12.19
  - in reports 12.17

- Check boxes 3.6, 10.19
  - and database tables 10.20
  - and null values 10.20
- Check constraints. *See* Constraints, check
- Class Designer window 2.15
- Classes 1.12, 3.4, 3.10, 13.1 to 13.12
  - adding instances of 13.10
  - adding objects to 13.10
  - as containers 13.8
  - developing 13.9
  - editing an instance of 13.7
  - master. *See* Master class
  - reinhiering properties and methods of 13.8
  - standard. *See* Standard classes
  - user-defined. *See* User-defined classes
- Client constraints. *See* Constraints, application
- Columns
  - binding to an existing control 17.6
  - changing attributes 8.13
  - constraints 8.7
  - Datatype field 8.13
  - foreign key 8.24
  - Name field 8.13
  - Not Null field 8.14
  - Precision field 8.13
  - primary key 8.24
  - Size field 8.13
  - Unique field 8.14
- Combo boxes 3.6, 10.22
  - and foreign tables 10.22
- Command code
  - for menu commands 14.9
  - for toolbar buttons 14.24
- Commands 4.21, 9.13
  - categories of 4.21
- Comments
  - in method code 5.8
- Comparison operators. *See* Operators, comparison
- Conditions 9.12
- Connector control 2.19
- Constants. *See* Symbolic constants
- Constraints 8.9, 19.1 to 19.32
  - and format masks 19.16
  - and the InsertRow() and DeleteRow() methods 19.20
  - and the Validate() method 19.19
  - application 19.13, 19.27, 19.29 to 19.32
  - at the control level 19.14 to 19.20
  - at the row level 19.20 to 19.23
  - at the session level 19.26 to 19.27
  - check 19.7
  - database 19.2, 19.28
  - database and application together 19.28
  - database, defining 19.9 to 19.11
  - database, removing 19.12 to 19.13
  - dataseize 19.14
  - datatype 19.14
  - foreign key 19.7
  - master-detail 18.6, 19.23
  - Not Null 19.3
  - primary key 19.4
  - read-only 19.15
  - server. *See* Constraints, database
  - stored procedures 19.8
  - triggers 19.8
  - unique 19.3
- Containers 3.2, 3.7, 3.8 to 3.12, 10.4, 10.6
  - and record sources 17.3
  - and tab order 10.38
  - application 3.9
  - bindable 8.16, 10.6, 17.3
  - binding graphically 17.4
  - binding manually 17.8
  - binding to a record source 17.1 to 17.34
  - binding to a table or view 17.5
  - binding to individual columns 17.6
  - binding to multiple record sources 17.3
  - bound 1.6
  - categories of 10.7
  - class 3.10
  - defined 3.3
  - embedded form 3.7, 10.8
  - form 3.10, 10.7, 11.1 to 11.20
  - library 3.9
  - properties and methods of bindable 17.31 to 17.34
  - radio button frame 3.7
  - recordset-related methods of 17.33
  - recordset-related properties of 17.32
  - repeater display 3.7, 10.9
  - repeater panel 3.7
  - report 3.10, 10.8
  - report group 3.7
  - session 3.10

---

- user-defined class 10.20
- Containment hierarchy 1.4
- Controls 3.2, 3.6, 10.4, 10.11 to 10.31
  - and format masks 10.31
  - and record sources 17.3
  - bindable 10.12, 17.4
  - categories of 10.16
  - chart 3.6, 10.16
  - check box 3.6, 10.19
  - combo box 3.6, 10.22
  - current row pointer 3.7, 10.23
  - display value 10.14
  - enabling and disabling 10.39
  - internal value 10.14
  - list 10.14, 10.15
  - list box 3.6, 10.24
  - OCX 3.6
  - OLE data objects 3.6, 10.24
  - picture 3.6, 10.25
  - popup list 3.6, 10.25
  - pushbutton 3.6, 10.26
  - radio button 3.6, 10.26
  - recordset-related methods of 17.34
  - recordset-related properties of 17.34
  - scrollbar 3.6, 10.28
  - sequence of 10.38
  - text field 3.6, 10.29
  - user-defined class 10.20
- Conversions
  - datatype 4.22
  - numeric 4.23
  - operator 4.22
- Counter fields. *See* Counters
- Counters 19.17
  - and sequences. *See also* Sequences
- Current row 17.13
- Current row pointers 3.7, 10.23

## D

- Data
  - transferring 17.18
- Data Definition Language (DDL) 8.5, 8.30
  - commands 9.13
- Data definition operations 8.4
- Data locks 17.25

- Data Manipulation Language (DML) 8.5, 8.30
  - commands 9.13
- Data manipulation operations 8.5
- Data model 1.10
- Data modeling. *See* Normalization
- Database connection 1.4
- Database Editor window 2.2
- Database engine 8.2
- Database objects 1.4, 3.2, 3.3, 8.1 to 8.32
  - categories of 8.3
  - copying 2.20, 8.30
  - deleting 8.31
  - index 3.4, 8.4
  - named in SQL statements 9.7
  - names 3.20
  - sequence 3.4, 8.4
  - session 8.3
  - synonym 3.4, 8.4
  - table 3.3, 8.3
  - view 1.5, 3.3, 8.3
- Database Session window 2.18
  - tasks performed in 2.19
- Database sessions. *See* Sessions
- Database tables. *See* Tables
- Database views. *See* Views
- Databases 1.4 to 1.5, 6.1 to 6.24
  - external 8.4
  - fetching rows from 17.14
  - querying 17.14
  - requering 17.21
  - See also* Blaze databases
  - See also* Oracle7 Servers
  - See also* SQL Server databases
- Datatype field 8.13
- Datatypes 9.3
  - and suffixes 4.7
  - ANSI 9.6
  - Blaze 9.4
  - conversions 4.22
  - Date 4.3
  - Double 4.3
  - in different databases 9.6
  - in Oracle Basic 4.3
  - Integer 4.3
  - Long integer 4.3
  - null 4.3, 9.7

- Object 4.3
  - of properties 3.15
  - of values 4.4
  - Oracle7 9.3
  - Single 4.3
  - SQL Server 9.5
  - String 4.3
  - Variant 4.3
  - Date and time literals 4.5, 9.9
  - Date datatype 4.3
  - Date operators. *See* Operators, date
  - Debugger (Expressions) window 5.11
    - interrogating values through 5.11
  - Debugger (Main) window 5.10
  - Debugging
    - interrogating values 5.11
    - moving to a point in method code 5.14
    - removing a breakpoint 5.13
    - reports 12.12
    - setting a breakpoint 5.12
    - setting a watchpoint 5.13
  - Default processing 1.2, 3.19
    - of methods 5.6
  - DELETE privilege 8.5
  - Designer objects 3.2, 3.4, 3.11
    - line 3.5
    - names 3.21
    - oval 3.5
    - rectangle 3.5
    - static 3.5
    - static text 3.5
  - Designer window 2.3
    - opening 2.8, 2.14
  - Detail rows 19.25
  - Development
    - application 1.2, 1.4, 1.8 to 1.12
    - object-oriented 1.2 to 1.4
  - Dialog boxes
    - modal 10.26, 11.12
  - Double datatype 4.3
  - Drill-down forms. *See* Master-detail relationships
  - Dynamic link libraries (DLLs) 15.2, 15.10 to 15.14
    - calling procedures 15.12
    - declaring procedures 15.10
    - development considerations 15.14
    - flexibility in 15.12
    - passing arguments 15.13
- E**
- Embedded forms 3.7, 10.8
  - Events
    - defined 3.18
  - EXEC SQL command 9.15
    - and result information 9.19
    - and sessions 9.18
  - Expressions 4.22, 9.11
    - evaluation of 4.22
    - self-modifying 5.9
  - External databases 8.4
- F**
- Fields (database). *See* Columns
  - File objects 3.2, 3.3
    - application 10.2
    - creating 2.7
    - deleting 2.8
    - names 3.21
  - Filename extensions 10.2
  - Filenames 3.21
  - Floating toolbar. *See* Object palette
  - Flushing. *See* Recordsets, sending changes to the database
  - Foreign Key columns 8.24
  - Foreign keys
    - and joins 18.3
  - Form Designer window 2.15
  - Form Run-Time toolbar 2.17
  - Format mask characters 10.34 to 10.37
    - dates 10.35
    - numbers 10.34
    - strings 10.35
  - Format masks 10.15, 10.31
    - and constraints 19.16
    - setting 10.37
    - standard 10.32
    - user-defined 10.33
  - Forms 3.4, 3.10, 10.7, 11.1 to 11.20
    - adding objects to 11.6
    - application-modal 11.11
    - behavior of 11.13, 11.19
    - copying 11.6

---

- creating 11.3
- cutting and pasting 11.6
- deleting 11.5
- methods 14.46, 14.47
- modal 11.11, 11.12
- printing 11.14
- running 2.16
- system-modal 11.11
- testing 11.8 to 11.11
- window styles of 11.13

- Functions 3.19, 9.11
  - adding to an object 5.5
  - aggregation. *See* Aggregation functions.
  - built-in 4.17
  - user-defined, declaring 5.4

## G

- Graphics
  - See* Charts
  - See* OLE data objects
  - See* Pictures
- Graphs. *See* Charts

## H

- Hierarchical containment structure 1.4
- Hierarchical names 3.23
  - full 3.24
  - partial 3.24
  - syntax 3.24

## I

- Indexes 3.4, 8.4, 8.24 to 8.25
  - creating 8.25
  - using 8.25
- Inheritance 3.29
  - object hierarchy 3.31
  - reinherting object characteristics 3.31
- In-memory objects 3.2, 3.7
  - creating with the NEW operator 14.2
  - menus 14.2, 14.2 to 14.20
  - properties and methods of 14.44 to 14.49
  - status lines 14.2, 14.34 to 14.44

- toolbars 14.2, 14.20 to 14.33
- INSERT privilege 8.5
- Instances
  - and inheritance 3.29
- Integer datatype 4.3
- Integrity checks. *See* Constraints
- Integrity constraints. *See* Constraints
- Interface
  - of Oracle Power Objects 2.1 to 2.37

## J

- Joins 18.2
  - automated 18.2
  - primary and foreign keys 18.3

## K

- Keywords. *See* Commands

## L

- Libraries 1.12, 3.3, 3.9
  - opening 3.9
- Library window 2.3, 2.26
  - referring to bitmaps 2.26
- Lines 3.5, 10.30
- List boxes 3.6, 10.24
- List controls 10.14, 10.15
- Literals 4.4, 9.8, 9.9
  - date and time 4.5, 9.9
  - numeric 4.4, 9.8
  - text 4.5, 9.9
- Logical operators. *See* Operators, logical
- Long integer datatype 4.3

## M

- Main window 2.2, 2.5
  - adding file objects to 2.7
  - creating file objects in 2.7
  - removing file objects from 2.8
  - tasks performed in 2.7
- Master class 13.2
  - object references to 13.9

- reinhiering properties and methods of 13.8
  - Master objects 3.30
  - Master rows 19.24
  - Master-detail relationships 8.9, 18.1 to 18.11
    - and joins. *See* Joins
    - defining containers 18.4
    - displaying 18.11
    - displaying detail records 18.9, 18.10
    - displaying master records 18.9
    - drill-down forms 18.10
    - in reports 12.15
    - properties for defining 18.4
  - Menu bars 3.8
    - adding menus to 14.12
    - associating with windows 14.13
    - creating 14.5, 14.18
    - initializing 14.5
    - methods 14.44
  - Menu commands
    - checked 14.4
    - disabled 14.4
    - help context 14.10
    - IDs 14.9
    - keyboard equivalents 14.10
  - Menu items
    - deleting 14.11
    - examining 14.11
    - executing code for 14.16
    - keyboard equivalents 14.4
    - modifying 14.11
    - separator lines 14.4
    - setting status of 14.14
  - Menus 3.8, 14.2, 14.3 to 14.20
    - adding items to 14.7
    - adding to menu bars 14.12
    - application default 14.3, 14.6
    - custom 14.3, 14.7, 14.14
    - deleting 14.12
    - examining 14.12
    - labels 14.8
    - methods 14.45
    - modifying 14.12
    - properties 14.45
    - system default 14.3, 14.6
  - Method code 3.19, 5.1 to 5.14
    - and default processing 5.6
    - commenting 5.8
    - creating standards 5.8
    - debugging 5.9
    - debugging. *See also* Debugging
    - opening and closing the code window 2.33
    - viewing 5.11
    - writing 5.5, 5.7
  - Methods 2.33 to 2.34, 3.13, 3.17 to 3.20, 5.1 to 5.14
    - adding to an object 5.5
    - and default processing 5.6
    - as functions. *See* Functions
    - as subroutines. *See* Subroutines
    - calling 3.20, 5.3
    - categories of 3.18
    - code. *See* Method code
    - creating 5.4
    - defined 1.2, 3.13
    - of objects 4.23
    - of OCX controls 15.17
    - overridden 3.31
    - passing arguments to 5.3
    - reinhiering 13.8
    - self-invoking 5.9
    - standard 1.2
    - syntax 3.13
    - triggering, by calling 5.2
    - triggering, through an event 5.2
    - user-defined 5.4
    - with shared names 5.7
  - Modularity 1.3
- ## N
- Name Field 8.13
  - NEW operator 14.2
  - Normalization 1.5, 8.8
  - Not Null field 8.14
  - Null datatype 4.3
  - Nulls 4.6, 9.7, 9.9
  - Numeric literals 4.4, 9.8
- ## O
- Object classes 3.29
  - Object containment hierarchy 3.12
    - and object references 3.25

- 
- object names 3.23
  - Object datatype 4.3
  - Object inheritance hierarchy 3.31, 13.3
  - Object linking and embedding (OLE). *See* OLE data objects
  - Object methods. *See* Methods, of objects
  - Object names 3.20 to 3.25
    - applications 3.21
    - changing 3.22
    - default 3.20
    - designer objects 3.21
    - hierarchical 3.23
    - of database objects 3.20
    - of files 3.21
    - rules concerning 3.20
  - Object operators. *See* Operators, object
  - Object palette 2.3, 2.36
    - creating an object with 2.37
    - creating OLE data objects 15.6
  - Object privileges 8.5
  - Object properties. *See* Properties, of objects
  - Object references 3.25
    - by name 3.25
    - by object datatype 3.25
    - by relative position 3.25
    - relative 3.25
    - restrictions on 3.28
    - syntax 3.28
  - Object-oriented development 1.2 to 1.4
  - Objects 3.1 to 3.34
    - application 3.2, 3.3
    - categories of 3.2
    - container 3.2
    - controls 3.2
    - database 3.2, 3.3 to 3.4, 8.1 to 8.32
    - designer 3.2, 3.11
    - file 3.2, 3.3
    - in-memory 3.2, 3.7
    - library 3.3
    - menu bar 3.8
    - menus 3.8
    - recordset 3.7
    - references to. *See* Object references
    - renaming 3.22
    - session 3.3
    - static 3.2
    - status line 3.8
    - toolbar 3.8
    - top-level 3.12
  - OCX controls 3.6, 15.2, 15.14 to 15.18
    - adding to a container 15.16
    - creating 15.15
    - importing 15.16
    - methods 15.17
    - properties 15.17
  - OLE controls 3.6, 15.5
    - binding 15.8
    - editing 15.8
    - running 15.8
  - OLE data objects 3.4, 10.5, 10.24, 15.2, 15.2 to 15.9
    - and files 15.8
    - and OLE controls. *See* OLE controls
    - and the Clipboard 15.9
    - classes of 15.5
    - creating 2.10, 15.6
    - embedded 15.6
    - in Oracle Power Objects 15.3
    - in reports 12.20
    - linked, creating 15.7
    - linking and embedding 15.4
    - pasting 15.7
    - technology overview 15.3
    - unbound 15.6
  - One-to-many relationships. *See* Master-detail relationships
  - Operators 4.12, 9.10
    - arithmetic 4.13, 9.10
    - binary 9.10
    - bitwise 4.15, 9.11
    - categories of 9.10
    - character 9.10
    - comparison 4.14, 9.10
    - conversions 4.22
    - date 4.16
    - logical 4.14
    - object 4.17
    - precedence of 4.13, 9.10
    - set 9.11
    - string 4.14
    - unary 9.10
  - Oracle Basic 4.1 to 4.23
    - aggregation functions. *See* Aggregation functions
    - built-in functions. *See* Functions, built-in
    - commands. *See* Commands



- components 4.2
- datatypes 4.3
- expressions 4.22
- operators. *See* Operators
- values 4.3
- variables. *See* Variables
- Oracle Power Objects
  - Application window. *See* Application window
  - Browser window. *See* Browser window
  - Database Editor windows. *See* Table Editor window
  - Designer windows. *See* Class Designer window
  - desktop 2.2
  - extensions 15.1 to 15.18
  - extensions. *See also* Dynamic link libraries (DLLs)
  - extensions. *See also* OCX controls
  - extensions. *See also* OLE data objects
  - interface 2.2
  - launching 2.4
  - Library window. *See* Library window
  - Main window. *See* Main window
  - Object palette. *See* Object palette
  - Property sheet. *See* Property sheet
  - Run-Time Debugger. *See* Run-Time Debugger
  - Session window. *See* Session window
- Oracle7 Servers 6.21 to 6.22
- Ovals 3.5, 10.30

## P

- Parent 3.12
- Picture controls 3.6, 10.25
- Pictures
  - in reports 12.20
- PL/SQL 9.14
- Popup lists 3.6, 10.25
- Precision field 8.13
- Primary Key columns 8.24
- Primary Key constraint 8.11
  - counters 19.17
- Primary key constraint
  - composite 8.11
- Primary keys 8.9
  - and joins 18.3
  - creating 19.28
  - setting 18.6
  - specifying 8.11

- Print preview. *See* Reports, previewing
- Privileges
  - ALTER 8.5
  - DELETE 8.5
  - INSERT 8.5
  - SELECT 8.5
  - UPDATE 8.5
- Procedural extensions 9.14
  - executing 9.20
  - PL/SQL 9.14
  - Transact-SQL 9.14
- Processing
  - default 3.19
- Properties 2.30 to 2.32, 3.14 to 3.17
  - categories of 3.14
  - datatypes of 3.15
  - defined 1.2, 3.13
  - of objects 4.23
  - of OCX controls 15.17
  - overridden 3.31
  - reinherting 13.8
  - setting 2.30
  - setting in multiple objects 2.32
  - setting on the Property sheet 3.15
  - setting with Oracle Basic 3.16
  - setting with the Debugger 3.17
  - standard 3.13
  - syntax 3.13
- Property sheet 2.3, 2.27
  - opening multiple 2.29
  - sections of 2.28
  - tasks performed through 2.27
  - viewing 2.29
- Property sheet buttons
  - Add Property 2.29
  - Delete Property 2.29
  - Group by Creator 2.28
  - Group by Type 2.28
  - Reinherit 2.29
  - Track Object 2.28
- Pushbuttons 3.6, 10.26
  - and modal dialog boxes 10.26

## Q

- Query by Form (QBF) 11.14 to 11.18, 11.20

---

- and master-detail relationships 11.17
- clearing conditions 11.17
- entering conditions 11.16
- syntax 11.18

## R

- Radio button frames. *See* Radio button groups
- Radio button groups 3.7, 10.26, 10.28, 17.4
  - creating 10.27
  - values 10.27
- Radio buttons 3.6, 10.26, 17.4
  - See also* Radio button groups
  - values 10.27
- Record sources
  - and containers 17.3
  - and controls 17.3
  - binding a container to 17.1 to 17.34
  - See also* Tables
- Record sources. *See also* Views
- Records (database). *See* Rows
- Recordsets 1.6, 3.7, 17.9 to 17.34
  - accessing programmatically 17.16
  - adding a new row 17.14, 17.23
  - and bound containers 17.9
  - and data transfer 17.18
  - and reports 12.7
  - bound 17.29
  - copying values between 17.18
  - current row 17.13
  - deleting 17.31
  - deleting a row 17.23
  - editing data in 17.23
  - fetching rows from 17.14
  - joining 18.2
  - master and detail 18.1 to 18.11
  - methods of 17.31 to 17.32
  - requering 17.21
  - sending changes to the database 17.24
  - shared 17.19
  - standalone 17.27
  - storage of rows 17.16
  - structure 17.10
  - unbound 17.28
- Rectangles 3.5, 10.31
- Recursive methods. *See* Methods, self-invoking
- Referential integrity. *See* Constraints
- Reinheriting object characteristics 3.31
- Relative reference 3.25
- Repeater displays 3.7, 10.9
  - and aggregate values 10.11
  - current row pointer 10.23
  - repositioning 10.10
  - resizing 10.10
- Repeater panels 3.7
- Report Designer window 2.16
- Report groups 3.7, 12.10
  - creating 12.11
  - defining 12.10
  - group-by column 12.10
- Report Run-Time toolbar 2.17
- Reports 3.4, 3.10, 10.8, 12.1 to 12.20
  - adding charts to 12.17
  - adding objects to 12.6
  - and recordsets 12.7
  - binding controls to columns 12.8
  - binding to a table or view 12.4
  - creating 12.3
  - defining filters 12.7
  - designing 12.4 to 12.6
  - detail 12.3
  - fonts 12.19
  - group footer 12.3
  - group header 12.3, 12.15
  - master-detail relationships in 12.15
  - methods 14.47
  - moving objects in 12.6
  - page footer 12.3
  - page header 12.3
  - page width 12.19
  - populating controls on 12.7
  - populating controls with derived values 12.9
  - populating controls with SQLLOOKUP 12.10
  - previewing 12.13
  - printing 12.13
  - printing headers and footers 12.14
  - printing, standard methods for 12.13
  - report footer 12.3
  - report groups 12.10
  - report header 12.3
  - resizing areas of 12.5
  - running 2.16

- starting a new page 12.14
  - testing 12.12
  - using derived values in 12.9
  - Rows
    - detail 19.25
    - master 19.24
  - Run-Time Debugger 2.3, 5.9
- S**
- Schemas 8.3
  - Scrollbars 3.6, 10.28
    - and database browsing 10.29
    - position of thumb 10.28
  - SELECT privilege 8.5
  - Sequences 1.5, 3.4, 8.4, 8.26 to 8.28
    - and counters. *See* Counters
    - associating with a bound container 8.28
    - beginning number 8.26
    - creating 8.28
    - cycling 8.26
    - in SQL statements 8.28
    - increment 8.26
    - maximum value 8.26
    - minimum value 8.26
    - name 8.26
  - Server constraints. *See* Constraints, database
  - Session objects 1.4
  - Session window 2.3
  - Sessions 1.4 to 1.5, 3.3, 3.10, 6.3 to 6.19, 8.3
    - activating at design time 2.19
    - associating with a schema 2.19
    - closing 2.19
    - directing statements to 9.18
    - opening 3.10
  - Siblings 3.12
  - Single datatype 4.3
  - Size field 8.13
  - SQL components
    - commands 9.3
    - conditions 9.3
    - expressions 9.3
    - functions 9.3
    - literals 9.2
    - objects 9.2
    - operators 9.2
      - procedural extensions 9.3
      - values 9.2
  - SQL Server databases 6.22 to 6.24
  - SQL statements
    - and tables 8.16
    - evaluation of 9.21
  - SQL. *See* Structured Query Language
  - SQLLOOKUP function 9.21, 10.12
    - and bind variables 9.22
    - and report group headers 12.15
    - derived values 9.22
    - lookup fields 9.22
    - to populate report controls 12.10
  - Standalone recordsets. *See* Recordsets, standalone
  - Standard classes 13.2
  - Standard methods 1.2
    - for printing reports 12.13
  - Static objects 3.2, 3.5, 10.4, 10.11, 10.30 to 10.31
    - line 10.30
    - oval 10.30
    - rectangle 10.31
    - static text 10.31
  - Static text 3.5, 10.31
  - Status lines 3.8, 14.2, 14.34 to 14.44
    - adding panels to 14.36
    - associating with windows 14.38
    - creating 14.35, 14.42
    - creating, overview 14.34
    - custom panels 14.34
    - deleting panels 14.36
    - initializing 14.35
    - methods 14.48
    - modifying panels 14.36
    - properties 14.48
    - summary panel 14.34
    - summary panel, disabling 14.42
    - system default panels 14.34
    - updating panels 14.39
  - Stored procedures 19.8
  - Stored queries. *See* Views
  - String datatype 4.3
  - String operators. *See* Operators, string
  - Structured Query Language (SQL) 9.1 to 9.23
    - bind variable arrays. *See* Bind variable arrays
    - bind variables. *See* Bind variables
    - categories of operators 9.10

---

- commands 9.13
- components. *See* SQL Components
- conditions 9.12
- EXEC SQL command. *See* EXEC SQL command
- executing SQL statements 9.15
- expressions 9.11
- functions 9.11
- SQLLOOKUP function. *See* SQLLOOKUP function
- syntax for databases 9.8
- values and datatypes 9.3

Subclasses 13.12

- and inheritance 3.29
- creating 13.12

Subroutines 3.19

- adding to an object 5.5
- user-defined, declaring 5.4

Summary help 14.34

Symbolic constants 4.10

- Boolean 4.11
- declaring 4.10
- names 4.10
- predefined 4.11
- property value 4.11

Synonyms 3.4, 8.4, 8.29

- and Data Definition Language (DDL) 8.30
- and Data Manipulation Language (DML) 8.30
- creating 8.30

Syntax

- method references 3.13
- property references 3.13

System-modal forms 11.11

## T

Tab order 10.38

Table Browser window 2.23, 8.8, 8.14

- opening 2.23

Table definition. *See* Tables, structure

Table Editor window 2.20

- creating a new table through 2.22
- Datatype field 2.21
- editing data in 2.23
- Expand button 2.22
- Name field 2.21
- Not Null field 2.21
- opening 2.22
- Precision field 2.21
- Primary Key indicator 2.21
- Primary Key tool 2.21
- Selector button 2.21
- Size field 2.21
- Unique field 2.21

Tables 3.3, 8.3, 8.6 to 8.16

- adding columns 8.10, 8.12
- binding to a container object 8.16
- constraints 8.7
- creating 2.20, 2.22, 8.8 to 8.12
- creating a translation list 8.16
- data 8.7
- deleting rows 8.15
- editing 2.23
- editing data 8.14, 8.15
- editing structure 8.12
- inserting rows 8.14
- issuing a SQL statement 8.16
- names 8.7
- rows 8.8
- saving 8.12, 8.16
- structure 8.6

Text fields 3.6, 10.29

- multiple lines 10.29

Text literals 4.5, 9.9

Toolbar buttons 14.21

- bitmaps on 14.25
- deleting 14.25
- disabled 14.21
- examining 14.25
- executing code for 14.30
- help context 14.25
- IDs 14.24
- method code for 14.28
- modifying 14.25
- setting status of 14.28
- style 14.25
- toggle 14.21

Toolbars 3.8, 14.2, 14.20 to 14.33

- adding buttons to 14.23
- application default 14.21
- associating with windows 14.27
- creating 14.21, 14.22, 14.31
- custom 14.21
- initializing 14.22

- methods 14.46
- separator areas 14.21
- Transaction processing 1.7
- Transaction Processing Language (TPL)
  - commands 9.13, 9.14
- Transact-SQL 9.14
- Translation fields 8.23
- Translation lists 8.16
- Triggers 19.8

## U

- Unary operators 9.10
- Unique field 8.14
- UPDATE privilege 8.5
- User Properties window 2.34
  - arguments 2.35
  - datatype 2.35
  - deleting user-defined methods and properties from 2.36
  - name 2.34
  - opening 2.34
  - type 2.34
- User-defined classes 10.5, 10.20, 13.2, 13.10
  - creating 13.9
- User-defined methods 2.34
  - adding to an object 2.35
  - creating 2.35
  - deleting from an object 2.35
- User-defined properties 2.34
  - adding to an object 2.35
  - creating 2.35
  - deleting from an object 2.35

## V

- Validation 10.14, 10.39
- Values 9.3
- Variables
  - and symbolic constants. *See* Symbolic constants
  - arrays 4.9
  - declaring 4.7
  - default values 4.8
  - global 4.8
  - in Oracle Basic 4.6
  - local 4.9
  - names 4.6

- scope of 4.8
- Variant datatype 4.3
- View Browser window 2.25
  - opening 2.25
- View definition. *See* Views, structure
- View Editor window 2.24, 8.17
  - Column List area 2.24
  - Condition field 2.25
  - creating a view through 2.25
  - Display field 2.25
  - Heading field 2.25
  - Join line 2.24
  - Name field 2.25
  - opening 2.25
  - Or field 2.25
  - Table field 2.24
  - Table List area 2.24
- Views 1.5, 3.3, 8.3, 8.16 to 8.23
  - adding base tables 8.19, 8.21
  - adding columns 8.20, 8.21
  - binding to a container 8.23
  - choosing base tables for 8.19
  - creating 2.20, 8.18
  - creating a translation field 8.23
  - data 8.18
  - deleting columns 8.22
  - deleting conditions 8.22
  - editing columns 8.23
  - editing conditions 8.23
  - editing data 8.23
  - editing structure 8.21
  - issuing a SQL statement 8.23
  - joining base tables 8.19
  - removing base tables 8.19, 8.21
  - saving 8.21
  - structure 8.17

## W

- Watchpoints
  - setting 5.13
- Windows API 15.13



## Reader's Comment Form

Oracle Power Objects User's Guide  
Part No. A25660-5  
July 1995

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have any suggestions for improvement, please indicate the topic, chapter, and page number below:

---

---

---

---

---

Please send your comments to:

Oracle Power Objects Documentation Manager  
Oracle Corporation  
500 Oracle Parkway  
Redwood Shores, CA 94065  
(415) 506-7000  
FAX: (415) 506-7200

If you would like a reply, please give your name, address, and telephone number below:

---

---

---

Thank you for helping us improve our documentation.

