

Oracle8i

Application Developer's Guide - Object-Relational Features

Release 2 (8.1.6)

December 1999

Part No. A76976-01

ORACLE®

Part No. A76976-01

Copyright © 1996, 1999, Oracle Corporation. All rights reserved.

Primary Author: John Russell

Contributing Authors: S. Banerjee, V. Krishnamurthy, M. Krishnaprasad, G. Lee, S. Muralidhar, D. Raphaely, R. Urbano

Contributors: S. Krishnaswamy, M. Morsi, R. Murthy, K. Osinski, E. Rohwedder, N. Shariatpanahy

Graphic Designer: V. Moore

The Programs (which include both the software and documentation) contain proprietary information of Oracle Corporation; they are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright, patent, and other intellectual and industrial property laws. Reverse engineering, disassembly, or decompilation of the Programs is prohibited.

The information contained 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. Except as may be expressly permitted in your license agreement for these Programs, no part of these Programs may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Oracle Corporation.

If the Programs are delivered to the U.S. Government or anyone licensing or using the programs on behalf of the U.S. Government, the following notice is applicable:

Restricted Rights Notice Programs delivered subject to the DOD FAR Supplement are "commercial computer software" and use, duplication, and disclosure of the Programs, including documentation, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement. Otherwise, Programs delivered subject to the Federal Acquisition Regulations are "restricted computer software" and use, duplication, and disclosure of the Programs shall be subject to the restrictions in FAR 52.227-19, Commercial Computer Software - Restricted Rights (June, 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

The Programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be the licensee's responsibility to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use of such applications if the Programs are used for such purposes, and Oracle Corporation disclaims liability for any damages caused by such use of the Programs.

Oracle is a registered trademark, and Pro*Ada, Pro*COBOL, Pro*FORTRAN, SQL*Loader, SQL*Net, SQL*Plus, Designer/2000, Developer/2000, Net8, Oracle Call Interface, Oracle7, Oracle8, Oracle8i, Oracle Forms, Oracle Parallel Server, PL/SQL, Pro*C, Pro*C/C++ and Trusted Oracle are trademarks or registered trademarks of Oracle Corporation. All other company or product names mentioned are used for identification purposes only and may be trademarks of their respective owners.

Contents

Send Us Your Comments	xi
Preface.....	xiii
1 Introduction to Oracle Objects	
The Nuts and Bolts of Oracle Objects	1-1
The Object-Relational Model	1-1
Object Types	1-2
Objects	1-3
Methods	1-3
Object Tables	1-5
Object Views.....	1-6
REF Datatype.....	1-6
Collections	1-8
Inheritance	1-10
An Example of an Object-Oriented Model.....	1-10
2 Managing Oracle Objects	
Using Object Types and References.....	2-1
Null Objects and Attributes	2-2
Default Values for Objects and Collections	2-3
Constraints for Object Tables.....	2-3
Indexes for Object Tables and Nested Tables.....	2-4
Triggers for Object Tables	2-5

Rules for REF Columns and Attributes	2-5
Name Resolution.....	2-6
Method Calls without Arguments	2-8
Using Collections	2-8
Querying Collections.....	2-8
Collection Unnesting.....	2-9
DML on Collections.....	2-9
Privileges on Object Types and Their Methods	2-10
System Privileges	2-10
Schema Object Privileges.....	2-11
Using Types in New Types or Tables	2-11
Example.....	2-11
Privileges on Type Access and Object Access	2-12
Dependencies and Incomplete Types	2-14
Completing Incomplete Types.....	2-15
Type Dependencies of Tables	2-15
Import/Export/Load of Object Types	2-15

3 Object Support in Oracle Programmatic Environments

SQL	3-1
PL/SQL	3-2
Oracle Call Interface (OCI)	3-2
Associative Access in OCI Programs	3-3
Navigational Access in OCI Programs	3-4
Object Cache	3-4
Building an OCI Program that Manipulates Objects.....	3-5
Pro*C/C++	3-6
Associative Access in Pro*C/C++	3-6
Navigational Access in Pro*C/C++	3-6
Converting Between Oracle Types and C Types	3-7
Oracle Type Translator (OTT).....	3-8
Oracle Objects For OLE (for Visual Basic, Excel, ActiveX, Active Server Pages)	3-9
Representing Objects in Visual Basic (OraObject)	3-10
Representing REFs in Visual Basic (OraRef).....	3-10
Representing VARRAYs and Tables in Visual Basic (OraCollection).....	3-11

Java: JDBC, Oracle SQLJ, and JPublisher	3-12
JDBC Access to Oracle Object Data.....	3-12
SQLJ Access to Oracle Object Data	3-12
Using JPublisher to Create Java Classes for JDBC and SQLJ Programs	3-13
 4 Applying an Object Model to Relational Data	
Why to Use Object Views.....	4-2
Defining Object Views.....	4-3
Using Object Views in Applications.....	4-4
Nesting Objects in Object Views.....	4-4
Identifying Null Objects in Object Views.....	4-5
Using Nested Tables and Varrays in Object Views.....	4-6
Specifying Object Identifiers for Object Views.....	4-7
Creating References to View Objects	4-9
Modelling Inverse Relationships with Object Views	4-10
Updating Object Views	4-11
Updating Nested Table Columns in Views	4-11
Using INSTEAD-OF Triggers to Control Mutating and Validation	4-11
Applying the Object Model to Remote Tables	4-12
Defining Complex Relationships in Object Views	4-13
Tables and Types to Demonstrate Circular View References	4-14
Creating Object Views with Circular References.....	4-16
 5 Design Considerations for Oracle Objects	
Representing Objects as Columns or Rows	5-1
Column Object Storage	5-2
Row Object Storage in Object Tables	5-7
Performance of Object Comparisons.....	5-7
Storage Considerations for Object Identifiers (OIDs).....	5-8
Storage Size of REFs.....	5-9
Integrity Constraints for REF Columns.....	5-9
Performance and Storage Considerations for Scoped REFs.....	5-9
Indexing Scoped REFs	5-10
Speeding up Object Access using the WITH ROWID Option.....	5-11
Viewing Object Data in Relational Form with Unnesting Queries.....	5-12

Storage Considerations for Varrays.....	5-13
Performance of Varrays vs. Nested Tables	5-14
Nested Tables.....	5-14
Nested Table Storage.....	5-14
Nested Table Indexes	5-17
Nested Table Locators.....	5-18
Optimizing Set Membership Queries	5-18
DML Operations on Nested Tables.....	5-19
Nesting Collections within other Collections	5-20
Choosing a Language for Method Functions	5-26
Static Methods	5-28
Writing Reusable Code using Invoker Rights	5-29
Function-Based Indexes on the Return Values of Type Methods.....	5-30
New Object Format in Release 8.1	5-31
Replicating Object Tables and Columns.....	5-31
Consequences of the Oracle Inheritance Implementation	5-32
Simulating Inheritance	5-32
Constraints on Objects.....	5-37
Type Evolution.....	5-38
Performance Tuning	5-38
Parallel Queries with Oracle Objects	5-38

6 Advanced Topics for Oracle Objects

Storage of Objects.....	6-1
Leaf-Level Attributes.....	6-1
How Row Objects are Split Across Columns	6-1
Hidden Columns for Tables with Column Objects	6-2
REFs	6-2
Internal Layout of Nested Tables	6-3
Internal Layout of VARRAYs	6-3
Object Identifiers	6-3
OCI Tips and Techniques for Objects.....	6-4
Initializing an OCI Program in Object Mode.....	6-4
Creating a New Object	6-4
Updating an Object.....	6-5

Deleting an Object	6-5
Controlling Object Cache Size	6-5
Retrieving Objects into the Client Cache (Pinning)	6-6
How to Choose the Locking Technique	6-8
Flushing an Object from the Object Cache.....	6-9
Pre-Fetching Related Objects (Complex Object Retrieval)	6-9
Demonstration of OCI and Oracle Objects	6-11
Using the OCI Object Cache with View Objects	6-11
Partitioning Tables that Contain Oracle Objects	6-14
Parallel Query with Object Views.....	6-15
How Locators Improve the Performance of Nested Tables	6-15

7 Frequently Asked Questions about Programming with Oracle Objects

General Questions about Oracle Objects	7-2
Are the object-relational features a separate option?	7-2
What are the design goals of Oracle8i Object-Relational & Extensibility technologies? ...	7-2
What are the key features in Oracle8i Object-Relational Technology?.....	7-2
What are the new Object-Relational features in Oracle8i?	7-5
Object Types	7-6
What is structured data?.....	7-6
Where are the user-defined types, user-defined functions, and abstract data types?.....	7-6
What is an object type?	7-6
Why are object types useful?	7-7
How is object data stored and managed in Oracle8i?	7-7
Is inheritance supported in Oracle8i?	7-7
Object Methods	7-8
What language can I use to write my object methods?.....	7-8
How do I decide between using PL/SQL and Java for my object methods?	7-8
When should I use external procedures?	7-8
What are definer and invoker rights?	7-9
Object References	7-9
What is an object reference?.....	7-9
When should I use object references? How are they different from foreign keys?	7-9
Can I construct object references based on primary keys?.....	7-10
What is a scoped REF and when should I use it?	7-10

Can I manipulate objects using object references in PL/SQL and Java?	7-10
Collections	7-10
What kinds of collections are supported by Oracle8i?	7-10
How do I decide between using varrays and nested tables for modeling collections?....	7-11
Do Oracle8i Objects support collections within collections?	7-11
What is a collection locator?	7-11
What is collection unnesting?	7-11
Object Views	7-12
What are the differences between object views and object tables?	7-12
Are object views updateable?	7-12
Object Cache	7-12
Why do we need the object cache?	7-12
Does the object cache support object locking?	7-13
Large Objects (LOBs)	7-13
How can I manage large objects using Oracle8i?	7-13
User-Defined Operators	7-14
What is a user-defined operator?	7-14
Why are user-defined operators useful?	7-14

8 A Sample Application using Object-Relational Features

Introduction	8-2
A Purchase Order Example	8-3
Implementing the Application Under The Relational Model	8-4
Entities and Relationships	8-5
Creating Tables Under the Relational Model	8-5
Inserting Values Under the Relational Model	8-8
Querying Data Under The Relational Model	8-9
Updating Data Under The Relational Model	8-10
Deleting Data Under The Relational Model	8-10
Limitations of a Purely Relational Model	8-11
The Evolution of the Object-Relational Database System.....	8-12
Implementing the Application Under The Object-Relational Model	8-13
Defining Types	8-14
Method Definitions.....	8-21
Creating Object Tables	8-23

Object Datatypes as a Template for Object Tables.....	8-25
Object Identifiers and References.....	8-26
Object Tables with Embedded Objects.....	8-26
Manipulating Objects Through Java	8-39
Using oracle.sql.* Classes (Weak Typing).....	8-39
Using Strong Typing (SQLData or CustomDatum)	8-41
Manipulating Objects with Oracle Objects for OLE.....	8-45
Selecting Data.....	8-45
Inserting Data.....	8-46
Updating Data.....	8-48
Calling a Method Function.....	8-50

Index

Send Us Your Comments

Oracle8i Application Developer's Guide - Object-Relational Features, Release 2 (8.1.6)

Part No. A76976-01

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?
- Do you need more information? If so, where?
- 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 other suggestions for improvement, please indicate the chapter, section, and page number (if available). You can send comments to us in the following ways:

- E-mail - infodev@us.oracle.com
- FAX - (650) 506-7228 Attn: Oracle Server Documentation
- Postal service:
Oracle Corporation
Server Documentation Manager
500 Oracle Parkway
Redwood Shores, CA 94065
USA

If you would like a reply, please give your name, address, and telephone number below.

If you have problems with the software, please contact your local Oracle Support Services.

Preface

Application Developer's Guide - Object-Relational Features describes how to write applications that use the object-relational features of the Oracle Server, Release 2 (8.1.6). Information in this guide applies to versions of the Oracle Server that run on all platforms, and does not include system-specific information.

The Preface includes the following sections:

- [Information in This Guide](#)
- [Audience](#)
- [Feature Coverage and Availability](#)
- [Other Guides](#)
- [How This Book Is Organized](#)
- [Conventions Used in This Guide](#)
- [Your Comments Are Welcome](#)

Information in This Guide

As an application developer, you are probably interested in features that help to write reusable code and to accurately model the application domain in the database schema. This Guide describes a set of application development features that address these subjects. You should already understand how to develop database applications; it might help to have some background in an object-oriented language such as C++ or Java.

Audience

The *Application Developer's Guide - Object-Relational Features* is intended for programmers developing new applications or converting existing applications to run in the Oracle environment. The object-relational features are often used in multimedia, Geographic Information Systems (GIS), and similar applications that deal with complex data. The object views feature can be valuable when writing new applications on top of an existing relational schema.

This guide assumes that you have a working knowledge of application programming, and that you are familiar with the use of Structured Query Language (SQL) to access information in relational database systems.

Feature Coverage and Availability

The *Application Developer's Guide - Object-Relational Features* contains information that describes the features and functionality of the Oracle8i and the Oracle8i Enterprise Edition products. Oracle8i and Oracle8i Enterprise Edition have the same basic features. The core object-relational features covered in this book are part of the Oracle8i server. Some other advanced features are available only with the Enterprise Edition, and some of these, such as the Data Cartridges, are optional.

For information about the differences between Oracle8i and the Oracle8i Enterprise Edition and the features and options that are available to you, see *Getting to Know Oracle8i*.

Other Guides

Use the *PL/SQL User's Guide and Reference* to learn PL/SQL and to get a complete description of this high-level programming language, which is Oracle Corporation's procedural extension to SQL.

For general information about developing applications, see the *Oracle8i Application Developer's Guide - Fundamentals*.

To use Oracle's object-relational features through Java, you should also refer to *Oracle8i JDBC Developer's Guide and Reference* and *Oracle8i Java Stored Procedures Developer's Guide*.

The Oracle Call Interface (OCI) is described in *Oracle Call Interface Programmer's Guide*.

You can use the OCI to build third-generation language (3GL) applications that access the Oracle Server.

Oracle Corporation also provides the Pro* series of precompilers, which allow you to embed SQL and PL/SQL in your application programs. If you write 3GL application programs in Ada, C, C++, COBOL, or FORTRAN that incorporate embedded SQL, then refer to the corresponding precompiler manual. For example, if you program in C or C++, then refer to the *Pro*C/C++ Precompiler Programmer's Guide*.

Oracle Developer/2000 is a cooperative development environment that provides several tools including a form builder, reporting tools, and a debugging environment for PL/SQL. If you use Developer/2000, then refer to the appropriate Oracle Tools documentation.

For SQL information, see the *Oracle8i SQL Reference* and *Oracle8i Administrator's Guide*. For basic Oracle concepts, see *Oracle8i Concepts*.

How This Book Is Organized

The *Application Developer's Guide - Object-Relational Features* contains the following chapters. This section includes a brief summary of what you will find in each chapter.

Chapter 1, "Introduction to Oracle Objects"

Introduces the background concepts and terminology that you need to work with Oracle objects.

Chapter 2, "Managing Oracle Objects"

Explains how to perform essential operations with objects and object types.

Chapter 3, "Object Support in Oracle Programmatic Environments"

Summarizes the object-relational features in SQL and PL/SQL; Oracle Call Interface (OCI); Pro*C/C++; Oracle Objects For OLE; and Java, JDBC, and Oracle SQLJ. The information in this chapter is high level, for education and planning. The following chapters explain how to use the object-relational features in greater detail.

Chapter 4, "Applying an Object Model to Relational Data"

Explains object views, which allow you to develop object-oriented applications without changing the underlying relational schema.

Chapter 5, "Design Considerations for Oracle Objects"

Explains the implementation and performance characteristics of Oracle's object-relational model.

Chapter 6, "Advanced Topics for Oracle Objects"

Discusses features that you might need to manage storage and performance as you scale up an object-oriented application.

Chapter 7, "Frequently Asked Questions about Programming with Oracle Objects"

Provides helpful hints for people getting started with object-oriented programming, or coming to Oracle with a background in some other database system or object-oriented language.

Chapter 8, "A Sample Application using Object-Relational Features"

Demonstrates how a relational program can be rewritten as an object-oriented one, schema and all.

Conventions Used in This Guide

The following notational and text formatting conventions are used in this guide:

[]

Square brackets indicate that the enclosed item is optional. Do not type the brackets.

{ }

Braces enclose items of which only one is required.

|

A vertical bar separates items within braces, and may also be used to indicate that multiple values are passed to a function parameter.

...

In code fragments, an ellipsis means that code not relevant to the discussion has been omitted.

`font change`

SQL or C code examples are shown in monospaced font.

italics

Italics are used for OCI parameters, OCI routines names, file names, and data fields.

UPPERCASE

Uppercase is used for SQL keywords, like `SELECT` or `UPDATE`.

This guide uses special text formatting to draw the reader's attention to some information. A paragraph that is indented and begins with a bold text label may have special meaning. The following paragraphs describe the different types of information that are flagged this way.

Note: The "Note" flag indicates that you should pay particular attention to the information to avoid a common problem or to increase understanding of a concept.

Warning: An item marked as "Warning" indicates something that an OCI programmer must be careful to do, or not do, in order for an application to work correctly.

See Also: Text marked "See Also" points you to another section of this guide, or to other documentation, for additional information about the topic being discussed.

Your Comments Are Welcome

We value and appreciate your comment as an Oracle user and reader of our manuals. As we write, revise, and evaluate our documentation, your opinions are the most important feedback we receive.

You can send comments and suggestions about this manual to the Information Development department at the following e-mail address:

`infodev@us.oracle.com`

If you prefer, then you can send letters or faxes containing your comments to the following address:

Server Technologies Documentation Manager

Oracle Corporation

500 Oracle Parkway

Redwood Shores, CA 94065

Fax: (650) 506-7200

Introduction to Oracle Objects

If you have programmed in C++, Java, Perl, or other such modern languages, then you have probably encountered the idea of object-oriented programming. Oracle provides a number of object-oriented features that let you transfer your design and problem-solving skills from those languages to database application development.

If you are a long-time database programmer, you may have been frustrated by the lack of complex types, the need to "flatten" hierarchical data structures into tables with numerous primary and foreign keys, and the amount of application logic that must be copied and adapted to handle each new situation. Oracle's object-oriented features are intended to help solve each of these problems.

Before reading this book, you should be familiar with database application development. You should be able to use one or more programming languages together with DDL and DML to do all the usual database operations.

If you are just beginning to use Oracle's object-relational features, and you still have questions after reading this chapter, refer to [Chapter 7, "Frequently Asked Questions about Programming with Oracle Objects"](#).

The Nuts and Bolts of Oracle Objects

Here is a brief overview of the concepts and terminology you need to work with Oracle's object-relational features.

The Object-Relational Model

The object-relational model is an evolutionary way to introduce object-oriented features to the database without giving up the existing relational features that are used in existing applications. As you read further, you will see how object-oriented

features are integrated into Oracle 8i without compromising the good features of the past.

Object-Relational Database Systems versus Third-Generation Languages

Why not create object-oriented applications using a third-generation language (3GL), without the database at all?

First, an RDBMS provides functionality that you can build upon, instead of reinventing.

Second, one of the problems of information management using 3GLs is that they are not persistent; or, if they are persistent, then they sacrifice security to get the necessary performance by locating the application logic and the data logic in the same address space. Neither trade-off is acceptable to users of an RDBMS, for whom both persistence and security are basic requirements.

This leaves the application developer working under the relational model with the problem of simulating complex types by some form of mapping into SQL. Apart from the many person-hours required, this approach involves serious problems of implementation. You must:

- Translate from application logic into data logic on 'write', and then
- Perform the reverse process on 'read' (and vice versa).

This involves heavy traffic between the client and server address spaces, leading to slower performance. If client and server are on different machines, network roundtrips may add considerable overhead.

Object-relational (O-R) technology solves these problems, as you will see throughout this book.

Object Types

Object types are abstractions of the real-world entities—for example, purchase orders—that application programs deal with. They are analogous to Java and C++ classes.

You can think of an object type as a template, and an object as a structure that matches the template. Object types can represent many different data structures; a few examples are line items, images, and spatial data.

Object types are schema objects, subject to the same kinds of administrative control as other schema objects (see [Chapter 2, "Managing Oracle Objects"](#)).

You can use object types to map an object model directly to a database schema, instead of flattening the model to relational tables and columns. They let you bring related pieces of data together in a single unit, and to store the behavior of data along with the data itself. Application code can retrieve and manipulate the data as objects.

An object type is a schema object with three kinds of components:

- A *name*, which identifies the object type uniquely within that schema.
- *Attributes*, which model the structure and state of the real-world entity. Attributes can be built-in types or object types.
- *Methods*, functions or procedures that implement operations that mimic ones you can perform on the real-world entity.

Objects

When you create a variable of an object type, the result is an object. The object has attributes and methods based on its type. Because the object is a concrete thing, you can assign values to its attributes and call its methods.

Methods

Methods of an object type are functions or procedures that are called by the application to model the behavior of the objects. Methods that are written in PL/SQL or Java are stored in the database, which is preferable for data-intensive procedures and short procedures that are called frequently. Procedures in other languages, such as C, are stored externally, which is preferable for computationally intensive procedures that are called less frequently.

The methods of an object type broadly fall into three categories: Member, Static, and Comparison.

A member method is a function or a procedure that always has an implicit `SELF` parameter, and thus can work with the attributes of a specific object. You invoke it using the "dot" notation `OBJECT_VARIABLE.METHOD()`. Member methods are useful for writing observer or mutator methods, where the operation affects a specific object and you do not need to pass in any parameters.

A static method is a function or a procedure that does not have an implicit `SELF` parameter. Such methods may be invoked by qualifying the method with the type name, as in `TYPE_NAME.METHOD()`. Static methods are useful for procedures that work with global data rather than the object state, or functions that return the same value regardless of the object.

Comparison methods compare instances of objects, to allow sorting and IF/THEN logic.

In the example, PURCHASE_ORDER has a method named GET_VALUE. Each purchase order object has its own GET_VALUE method. For example, if X_OBJ and Y_OBJ are PL/SQL variables that hold purchase order objects and W_NUM and Z_NUM are variables that hold numbers, the following two statements can retrieve values from the objects:

```
w_num = x_obj.get_value();
z_num = y_obj.get_value();
```

Both objects, being of the same type, have the GET_VALUE method. The method call does not need any parameters, because it operates on its own set of data: the attributes of X_OBJ and Y_OBJ. In this *selfish style* of method invocation, the method uses the appropriate set of data depending upon the object for which it is called.

Constructor Methods Every object type has a system-defined *constructor method*, that is, a method that makes a new object and sets up the values of its attributes. The name of the constructor method is the name of the object type. Its parameters have the names and types of the object type's attributes. The constructor method is a function. It returns the new object as its value.

For example, the expression

```
purchase_order(
  1000376,
  person ("John Smith", "1-800-555-1212"),
  NULL )
```

represents a purchase order object with the following attributes:

```
id          1000376
contact     person("John Smith", "1-800-555-1212")
lineitems   NULL
```

The expression `person ("John Smith", "1-800-555-1212")` is an invocation of the constructor function for the object type PERSON. The object that it returns becomes the contact attribute of the purchase order.

See ["Null Objects and Attributes"](#) on page 2-2 for a discussion of null objects and null attributes.

Comparison Methods Oracle has facilities for comparing two data items of a given built-in type and determining whether one is greater than, equal to, or less than the

other. To compare two items of a user-defined type, the creator of the type must define an order relationship for the type using *map* methods or *order* methods.

Map methods produce a single value of a built-in type that can be used for comparisons and sorting. For example, if you define an object type called **RECTANGLE**, the map method **AREA** can multiply its **HEIGHT** and **WIDTH** attributes and return the answer. Oracle can then compare two rectangles by comparing their areas.

Order methods are more general. They use their own internal logic to compare two objects of a given object type and return a value that encodes the order relationship: -1 if the first is smaller, 0 if they are equal, and 1 if the first is larger.

For example, an order method can allow you to sort a set of addresses based on their distance from a fixed point, or some other operation more complicated than comparing individual values.

In defining an object type, you can specify either a map method or an order method for it, but not both. If an object type has no comparison method, Oracle cannot determine a greater-than or less-than relationship between two objects of that type. It can, however, attempt to determine whether two objects of the type are equal.

Oracle compares two objects of a type that lacks a comparison method by comparing corresponding attributes:

- If all the attributes are non-null and equal, the objects are considered equal.
- If there is an attribute for which the two objects have unequal non-null values, the objects are considered unequal.
- Otherwise, the objects are considered unequal.

Because the system can perform scalar value comparisons very efficiently, coupled with the fact that calling a user-defined function is slower than calling a kernel implemented function, sorting objects using the **ORDER** method is relatively slow compared to sorting the mapped scalar values returned by the **MAP** function.

Object Tables

An *object table* is a special kind of table in which each row represents an object.

For example, the following statement defines an object table for objects of the **PERSON** type:

```
CREATE TABLE person_table OF person;
```

Oracle allows you to view this table in two ways:

- A single-column table in which each row is a PERSON object, allowing you to perform object-oriented operations.
- A multi-column table in which each attribute of the object type PERSON, namely NAME and PHONE, occupies a column, allowing you to perform relational operations.

For example, you can execute the following instructions:

```
INSERT INTO person_table VALUES (  
    "John Smith",  
    "1-800-555-1212" );  
  
SELECT VALUE(p) FROM person_table p  
    WHERE p.name = "John Smith";
```

The first instruction inserts a PERSON object into PERSON_TABLE as a multi-column table. The second selects from PERSON_TABLE as a single column table.

Row Objects and Column Objects Objects that occupy complete rows in object tables are called *row objects*. Objects that occupy table columns in a larger row, or are attributes of other objects, are called *column objects*.

Object Views

An object view (see [Chapter 4, "Applying an Object Model to Relational Data"](#)) is a way to access relational data using object-relational features. It lets you develop object-oriented applications without changing the underlying relational schema.

REF Datatype

A REF is a logical "pointer" to a row object. It is an Oracle built-in datatype. REFs and collections of REFs model associations among objects--particularly many-to-one relationships--thus reducing the need for foreign keys. REFs provide an easy mechanism for navigating between objects. You can use the dot notation to follow the pointers. Oracle does joins for you when needed, and in some cases can avoid doing joins.

You can use a REF to examine or update the object it refers to. You can also use a REF to obtain a copy of the object it refers to. You can change a REF so that it points to a different object of the same object type, or assign it a null value.

Scoped REFS In declaring a column type, collection element, or object type attribute to be a REF, you can constrain it to contain only references to a specified object table. Such a REF is called a *scoped REF*. Scoped REFS require less storage space and allow more efficient access than unscoped REFS.

Dangling REFS It is possible for the object identified by a REF to become unavailable—through either deletion of the object or a change in privileges. Such a REF is called *dangling*. Oracle SQL provides a predicate (called IS DANGLING) to allow testing REFS for this condition.

Dereferencing REFS Accessing the object referred to by a REF is called *dereferencing* the REF. Oracle provides the Deref operator to do this.

Dereferencing a dangling REF returns a null object.

Oracle also provides *implicit dereferencing* of REFS. For example, consider the following:

```
CREATE TYPE person AS OBJECT (  
    name    VARCHAR2(30),  
    manager REF person );
```

If X represents an object of type PERSON, then the SQL expression:

```
x.manager.name;
```

follows the pointer from the person X to another person, X's manager, and retrieves the manager's name. (Following the REF like this is allowed in SQL, but not in PL/SQL.)

Obtaining REFS You can obtain a REF to a row object by selecting the object from its object table and applying the REF operator. For example, you can obtain a REF to the purchase order with identification number 1000376 as follows:

```
DECLARE OrderRef REF to purchase_order;  
  
SELECT REF(po) INTO OrderRef  
FROM purchase_order_table po  
WHERE po.id = 1000376;
```

The query must return exactly one row.

For more on storage of objects and REFS, see ["Using Collections"](#) on page 2-8.

Collections

For modelling one-to-many relationships, Oracle supports two *collection* datatypes: varrays and nested tables. For example, a purchase order has an arbitrary number of line items, so you may want to put the line items into a collection.

- Varrays have a maximum number of elements, although you can change the upper bound. The order of elements is defined. Varrays are stored as opaque objects (that is, raw or BLOB).
- Nested tables can have any number of elements, and you can select, insert, delete, and so on the same as with regular tables. The order of the elements is not defined. Nested tables are stored in a storage table with every element mapping to a row in the storage table.

If you need to loop through the elements in order, store only a fixed number of items, or retrieve and manipulate the entire collection as a value, then use varrays.

If you need to run efficient queries on collections, handle arbitrary numbers of elements, or do mass insert/update/delete operations, then use nested tables. If the collections are very large and you want to retrieve only subsets, you can model the collection as a nested table and retrieve a locator for the result set.

For example, a purchase order object may have a nested table of line items, while a rectangle object may contain a varray with 4 coordinates.

Creating a VARRAY or Nested Table

You can make an object of a collection type by calling its constructor method. The name of the constructor method is the name of the type, and its argument is a comma-separated list of the new collection's elements.

Calling the constructor method with an empty list creates an empty collection of that type. An empty collection is different from a null collection.

VARRAYs

An *array* is an ordered set of data *elements*. All elements of a given array are of the same datatype. Each element has an *index*, which is a number corresponding to the element's position in the array.

The number of elements in an array is the *size* of the array. Oracle allows arrays to be of variable size, which is why they are called VARRAYs. You must specify a maximum size when you declare the array type.

For example, the following statement declares an array type:

```
CREATE TYPE prices AS VARRAY(10) OF NUMBER(12,2);
```

The VARRAYs of type PRICES have no more than ten elements, each of datatype NUMBER(12,2).

Creating an array type does not allocate space. It defines a datatype, which you can use as

- The datatype of a column of a relational table.
- An object type attribute.
- The type of a PL/SQL variable, parameter, or function return value.

A VARRAY is normally stored in line, that is, in the same tablespace as the other data in its row. If it is sufficiently large, Oracle stores it as a BLOB (see ["Import/Export/Load of Object Types"](#) on page 2-15).

Additional Information: For more information on VARRAYs, see ["Storage Considerations for Varrays"](#) on page 5-13.

Nested Tables

A *nested table* is an unordered set of data *elements*, all of the same datatype. It has a single column, and the type of that column is a built-in type or an object type. If the column in a nested table is an object type, the table can also be viewed as a multi-column table, with a column for each attribute of the object type.

For example, in the purchase order example, the following statement declares the table type used for the nested tables of line items:

```
CREATE TYPE lineitem_table AS TABLE OF lineitem;
```

A table type definition does not allocate space. It defines a type, which you can use as

- The datatype of a column of a relational table.
- An object type attribute.
- A PL/SQL variable, parameter, or function return type.

When a table type appears as the type of a column in a relational table or as an attribute of the underlying object type of an object table, Oracle stores all of the nested table data in a single table, which it associates with the enclosing relational or object table. For example, the following statement defines an object table for the object type PURCHASE_ORDER:

```
CREATE TABLE purchase_order_table OF purchase_order
  NESTED TABLE lineitems STORE AS lineitems_table;
```

The second line specifies `LINEITEMS_TABLE` as the storage table for the `LINEITEMS` attributes of all of the `PURCHASE_ORDER` objects in `PURCHASE_ORDER_TABLE`.

A convenient way to access the elements of a nested table individually is to use a nested cursor.

Additional Information: See *Oracle8i SQL Reference* for information about nested cursors, and see "[Nested Tables](#)" on page 5-14 for more information on using nested tables.

Inheritance

Inheritance is a technique used in object-oriented development to create objects that contain generalized attributes and behavior for groups of related objects. The more general object types are referred to as a super-types. The specialized object types that "inherit" from the super-types are called subtypes.

A common case of inheritance is that of `Person` and `Employee`. The set of people includes employees and non-employees. The more general case, `Person`, is the super-type and the special case, `Employee`, the sub-type. Another example could involve a `Vehicle` as super-type and `Car` and `Truck` as its subtypes.

An Example of an Object-Oriented Model

Here is an example of how you might define a set of object types.

The object types are `PERSON`, `LINEITEM`, `LINEITEM_TABLE`, and `PURCHASE_ORDER`.

`NAME`, `PHONE`, `ITEM_NAME`, and so on are attributes of their respective object types. The attribute `CONTACT` is an object, and the attribute `LINEITEMS` is a nested table.

`LINEITEM_TABLE` is a table in which each row is an object of type `LINEITEM`.

```
CREATE TYPE person AS OBJECT (
  name      VARCHAR2(30),
  phone     VARCHAR2(20) );

CREATE TYPE lineitem AS OBJECT (
  item_name VARCHAR2(30),
```

```
quantity    NUMBER,
unit_price  NUMBER(12,2) );

CREATE TYPE lineitem_table AS TABLE OF lineitem;

CREATE TYPE purchase_order AS OBJECT (
    id        NUMBER,
    contact    person,
    lineitems  lineitem_table,

    MEMBER FUNCTION
    get_value  RETURN NUMBER );
```

This is a simplified example. It does not show how to specify the body of the method GET_VALUE, which you do with the CREATE OR REPLACE TYPE BODY statement.

Defining an object type does not allocate any storage.

You can use LINEITEM, PERSON, or PURCHASE_ORDER in SQL statements in most of the same places you can use types like NUMBER or VARCHAR2.

For example, you might define a relational table to keep track of your contacts:

```
CREATE TABLE contacts (
    contact    person
    date       DATE );
```

The CONTACT table is a relational table with an object type defining one of its columns. Objects that occupy columns of relational tables are called *column objects* (see ["Row Objects and Column Objects"](#) on page 1-6).

Managing Oracle Objects

This chapter explains how Oracle objects work in combination with the rest of the database, and how to perform DML and DDL operations on them. It contains the following major sections:

- [Using Object Types and References](#)
- [Using Collections](#)
- [Privileges on Object Types and Their Methods](#)
- [Dependencies and Incomplete Types](#)
- [Import/Export/Load of Object Types](#)

Using Object Types and References

This section describes object types and references, including:

- [Null Objects and Attributes](#)
- [Default Values for Objects and Collections](#)
- [Constraints for Object Tables](#)
- [Indexes for Object Tables and Nested Tables](#)
- [Triggers for Object Tables](#)
- [Rules for REF Columns and Attributes](#)
- [Name Resolution](#)

Null Objects and Attributes

Many things associated with objects can be null: a table column, object, object attribute, collection, or collection element. This means that the item is initialized to NULL or is not initialized. Usually, the value of the item is not yet known but might become available later.

An object whose value is NULL is called *atomically null*. In addition, attributes of an object can be null. These two uses of nulls are different. When all the attributes of an object are null, you can still change the attributes and call methods. When an object is atomically null, you cannot do very much with it at all.

For example, consider the CONTACTS table defined as follows:

```
CREATE TYPE person AS OBJECT (  
    name      VARCHAR2(30),  
    phone     VARCHAR2(20) );  
  
CREATE TABLE contacts (  
    contact    person  
    date       DATE );
```

The statement

```
INSERT INTO contacts VALUES (  
    person (NULL, NULL),  
    '24 Jun 1997' );
```

gives a different result from

```
INSERT INTO contacts VALUES (  
    NULL,  
    '24 Jun 1997' );
```

In both cases, Oracle allocates space in CONTACTS for a new row and sets its DATE column to the value given. In the first case, Oracle allocates space for an object in the PERSON column and sets each of its attributes to NULL. In the second case, it sets the PERSON column to NULL and does not allocate space for an object.

In some cases, you can omit checks for null values. A table row or row object cannot be null. A nested table of objects cannot contain an element whose value is NULL.

A nested table or array can be null, so you do need to handle that condition. A null collection is different from an empty one, that is, a collection containing no elements.

Default Values for Objects and Collections

When you declare a table column to be of an object type or collection type, you can include a `DEFAULT` clause. This provides a value to use in cases where you do not explicitly specify a value for the column. The default clause must contain a *literal invocation* of the constructor method for that object or collection.

A *literal invocation* of a constructor method is a call to the constructor method in which any arguments are either literals, or further literal invocations of constructor methods. No variables or functions are allowed.

For example, consider the following statements:

```
CREATE TYPE person AS OBJECT (
  id      NUMBER
  name    VARCHAR2(30),
  address VARCHAR2(30) );

CREATE TYPE people AS TABLE OF person;
```

The following is a literal invocation of the constructor method for the nested table type `PEOPLE`:

```
people ( person(1, 'John Smith', '5 Cherry Lane'),
         person(2, 'Diane Smith', NULL) )
```

The following example shows how to use literal invocations of constructor methods to specify defaults:

```
CREATE TABLE department (
  d_no   CHAR(5) PRIMARY KEY,
  d_name CHAR(20),
  d_mgr  person DEFAULT person(1, 'John Doe', NULL),
  d_emps people DEFAULT people() )
NESTED TABLE d_emps STORE AS d_emps_tab;
```

Note that the term `PEOPLE()` is a literal invocation of the constructor method for an empty `PEOPLE` table.

Constraints for Object Tables

You can define constraints on an object table just as you can on other tables.

You can define constraints on the leaf-level scalar attributes of a column object, with the exception of `REFs` that are not scoped.

The following examples illustrate the possibilities.

The first example places a primary key constraint on the SSNO column of the object table PERSON_EXTENT:

```
CREATE TYPE location (  
    building_no NUMBER,  
    city          VARCHAR2(40) );  
  
CREATE TYPE person (  
    ssno          NUMBER,  
    name          VARCHAR2(100),  
    address       VARCHAR2(100),  
    office        location );  
  
CREATE TABLE person_extent OF person (  
    ssno          PRIMARY KEY );
```

The DEPARTMENT table in the next example has a column whose type is the object type LOCATION defined in the previous example. The example defines constraints on scalar attributes of the LOCATION objects that appear in the DEPT_LOC column of the table.

```
CREATE TABLE department (  
    deptno        CHAR(5) PRIMARY KEY,  
    dept_name     CHAR(20),  
    dept_mgr      person,  
    dept_loc      location,  
    CONSTRAINT    dept_loc_cons1  
        UNIQUE (dept_loc.building_no, dept_loc.city),  
    CONSTRAINT    dept_loc_cons2  
        CHECK (dept_loc.city IS NOT NULL) );
```

Indexes for Object Tables and Nested Tables

You can define indexes on an object table or on the storage table for a nested table column or attribute, just as you can on other tables.

You can define indexes on leaf-level scalar attributes of column objects, as shown in the following example. You can only define indexes on REF attributes or columns if the REF is scoped.

Here, DEPT_ADDR is a column object, and CITY is a leaf-level scalar attribute of DEPT_ADDR that we want to index:

```
CREATE TABLE department (  
    deptno        CHAR(5) PRIMARY KEY,  
    dept_name     CHAR(20),  
    dept_mgr      person,  
    dept_loc      location,  
    dept_addr     DEPT_ADDR,  
    CONSTRAINT    dept_addr_idx  
        INDEX (dept_addr.city)
```

```

deptno      CHAR(5) PRIMARY KEY,
dept_name   CHAR(20),
dept_addr   address );

CREATE INDEX i_dept_addr1
            ON department (dept_addr.city);

```

Wherever Oracle expects a column name in an index definition, you can also specify a scalar attribute of an object column.

Triggers for Object Tables

You can define triggers on an object table just as you can on other tables. You cannot define a trigger on the storage table for a nested table column or attribute.

You cannot modify LOB values in a trigger body. Otherwise, there are no special restrictions on using object types with triggers.

The following example defines a trigger on the PERSON_EXTENT table defined in an earlier section:

```

CREATE TABLE movement (
    ssno      NUMBER,
    old_office location,
    new_office location );

CREATE TRIGGER trig1
BEFORE UPDATE
    OF office
    ON person_extent
FOR EACH ROW
    WHEN new.office.city = 'REDWOOD SHORES'
BEGIN
    IF :new.office.building_no = 600 THEN
        INSERT INTO movement (ssno, old_office, new_office)
            VALUES (:old.ssno, :old.office, :new.office);
    END IF;
END;

```

Rules for REF Columns and Attributes

In Oracle, a REF column or attribute can be unconstrained or constrained using a SCOPE clause or a referential constraint clause. When a REF column is unconstrained, it may store object references to row objects contained in any object table of the corresponding object type.

Oracle does not ensure that the object references stored in such columns point to valid and existing row objects. Therefore, REF columns may contain object references that do not point to any existing row object. Such REF values are referred to as *dangling references*. Currently, Oracle does not permit storing object references that contain a primary-key based object identifier in unconstrained REF columns.

A REF column may be constrained to be scoped to a specific object table. All the REF values stored in a column with a SCOPE constraint point at row objects of the table specified in the SCOPE clause. The REF values may, however, be dangling.

A REF column may be constrained with a REFERENTIAL constraint similar to the specification for foreign keys. The rules for referential constraints apply to such columns. That is, the object reference stored in these columns must point to a valid and existing row object in the specified object table.

UNIQUE or PRIMARY KEY constraints cannot be specified for REF columns. However, you can specify NOT NULL constraints for such columns.

Name Resolution

Oracle SQL lets you omit table names in some relational operations. For example, if ASSIGNMENT is a column in PROJECTS and TASK is a column in DEPTS, you can write:

```
SELECT *  
FROM projects  
WHERE EXISTS  
    (SELECT * FROM depts  
     WHERE assignment = task);
```

Oracle determines which table each column belongs to.

You can qualify the column names with table names or table aliases to make things more maintainable:

```
SELECT * FROM projects WHERE EXISTS  
    (SELECT * FROM depts WHERE projects.assignment = depts.task);  
  
SELECT * FROM projects pj WHERE EXISTS  
    (SELECT * FROM depts dp WHERE pj.assignment = dp.task);
```

In some cases, object-relational features require you to specify the table aliases.

When Table Aliases are Required

Using unqualified names can lead to problems. If you add an `ASSIGNMENT` column to the second table (`DEPTS`) and forget to change the query, Oracle automatically recompiles the query and the new version uses the `ASSIGNMENT` column from the `DEPTS` table. This situation is called *inner capture*.

To avoid inner capture and similar misinterpretations of the intended meanings of SQL statements, Oracle requires you to use table aliases to qualify references to methods or attributes of objects.

The same requirement applies to attribute references via REFs. This requirement is called the *capture avoidance rule*.

For example, consider the following statements:

```
CREATE TYPE person AS OBJECT (ssno VARCHAR(20));
CREATE TABLE ptab1 OF person;
CREATE TABLE ptab2 (c1 person);
```

These define an object type `PERSON` and two tables. The first is an object table for objects of type `PERSON`. The second has a single column of type `PERSON`.

Now consider the following queries:

```
SELECT      ssno FROM ptab1   ; --Correct
SELECT    c1.ssno FROM ptab2   ; --Wrong
SELECT p.c1.ssno FROM ptab2 p ; --Correct
```

- In the first `SELECT` statement, `SSNO` is the name of a column of `PTAB1`. Because this is considered a relational query, no further qualification is required.
- In the second `SELECT` statement, `SSNO` is the name of an attribute of the `PERSON` object in the column named `C1`. This reference requires a table alias, as shown in the third `SELECT` statement.

You must qualify references to object attributes with table aliases rather than table names, even if the table names are further qualified by schema names.

For example, the following expression tries to refer to the `SCOTT` schema, `PROJECTS` table, `ASSIGNMENT` column, and `DUEDATE` attribute of that column. But it is not allowed because `PROJECTS` is a table name, not an alias.

```
scott.projects.assignment.duedate
```

Table aliases should be unique throughout a query and should not be the same as schema names that could legally appear in the query.

Note: We recommend that you define table aliases in all UPDATE, DELETE, and SELECT statements and subqueries and use them to qualify column references, whether or not the columns contain object types.

Method Calls without Arguments

Methods are functions or subroutines. The proper syntax for invoking them uses parentheses following the method name to enclose any calling arguments. In order to avoid ambiguities, Oracle requires empty parentheses for method calls that do not have arguments.

For example, if TB is a table with column C of object type T, and T has a method m that does not take arguments, the following query illustrates the correct syntax:

```
SELECT p.c.m() FROM tb p;
```

This differs from the rules for PL/SQL functions and procedures, where the parentheses are optional for calls that have no arguments.

Using Collections

This section describes the use of collections, including:

- [Querying Collections](#)
- [Collection Unnesting](#)
- [DML on Collections](#)

Querying Collections

In Oracle8i, a collection column may be queried using the TABLE expression. For example, a nested table column (PROJECTS) of the table (EMPLOYEES) can be queried as follows:

```
SELECT * FROM TABLE(SELECT t.projects FROM employees t WHERE t.eno = 1000);
```

```
SELECT t.eno, CURSOR(SELECT * FROM TABLE(t.projects)) FROM employees t;
```

The TABLE expression can be used to query any collection value expression, including transient values such as variables and parameters.

Note: The **TABLE** expression takes the place of **THE** subquery introduced in a previous release. The **THE** subquery expression will eventually be deprecated.

Collection Unnesting

Many tools and applications are not equipped to deal with collection types, and require a flattened view of the data. In order to use these tools to view Oracle collection data, you must have to unnest or flatten the collection attribute of a row into one or more relational rows. You do this by joining the rows of the nested table with the row that contains the nested table.

Consider the following object-relational schema, where we define a type that contains a nested table, and specify the name by which we will access the nested table:

```
CREATE TYPE emp_set_t IS NESTED TABLE OF emp_t;
CREATE TYPE dept_t(deptno NUMBER, emps emp_set_t);
CREATE TABLE depts OF dept_t NESTED TABLE emps STORE AS depts_emps;
```

The following query unnests the data in the **EMPS** column with respect to the **DEPT** table by augmenting every row of **EMPS** with its parent **DEPTS** row:

```
SELECT d.deptno, e.* FROM depts d, TABLE(d.emps) e;
```

Oracle8i also supports the following syntax to produce outer-join results:

```
SELECT d.*, e.* FROM depts d, TABLE(d.emps)(+) e;
```

The **(+)** indicates that the dependent join between **DEPTS** and **D.EMPS** should be **NULL**-augmented. That is, there will be rows of **DEPTS** in the output for which **D.EMPS** is **NULL** or empty, with **NULL** values for columns corresponding to **D.EMPS**.

DML on Collections

Oracle supports the following DML operations on nested table columns:

- Inserts and updates that provide a new value for the entire collection
- Piecewise Updates
 - Inserting new elements into the collection
 - Deleting elements from the collection

- DROP ANY TYPE allows you to drop named types in any schema.
- EXECUTE ANY TYPE allows you to use and reference named types in any schema.

The CONNECT and RESOURCE roles include the CREATE TYPE system privilege. The DBA role includes all of the above privileges.

Schema Object Privileges

The only schema object privilege that applies to object types is EXECUTE.

EXECUTE on a object type allows you to use the type to:

- Define a table.
- Define a column in a relational table.
- Declare a variable or parameter of the named type.

EXECUTE lets you invoke the type's methods, including the constructor.

Method execution and the associated permissions are the same as for stored PL/SQL procedures.

Using Types in New Types or Tables

In addition to the permissions detailed in the previous sections, you need specific privileges to:

- Create types or tables that use types created by other users.
- Grant use of your new types or tables to other users.

You must have the EXECUTE ANY TYPE system privilege, or you must have the EXECUTE object privilege for any type you use in defining a new type or table. You must have received these privileges explicitly, not through roles.

If you intend to grant access to your new type or table to other users, you must have either the required EXECUTE object privileges with the GRANT option or the EXECUTE ANY TYPE system privilege with the option WITH ADMIN OPTION. You must have received these privileges explicitly, not through roles.

Example

Assume that three users exist with the CONNECT and RESOURCE roles: USER1, USER2, and USER3.

USER1 performs the following DDL in the USER1 schema:

```
CREATE TYPE type1 AS OBJECT ( attr1 NUMBER );  
CREATE TYPE type2 AS OBJECT ( attr2 NUMBER );  
GRANT EXECUTE ON type1 TO user2;  
GRANT EXECUTE ON type2 TO user2 WITH GRANT OPTION;
```

USER2 performs the following DDL in the USER2 schema:

```
CREATE TABLE tab1 OF user1.type1;  
CREATE TYPE type3 AS OBJECT ( attr3 user1.type2 );  
CREATE TABLE tab2 (col1 user1.type2 );
```

The following statements succeed, because USER2 has EXECUTE on USER1's TYPE2 with the GRANT option:

```
GRANT EXECUTE ON type3 TO user3;  
GRANT SELECT on tab2 TO user3;
```

However, the following grant fails, because USER2 does not have EXECUTE on USER1.TYPE1 with the GRANT option:

```
GRANT SELECT ON tab1 TO user3;
```

USER3 can successfully perform the following actions:

```
CREATE TYPE type4 AS OBJECT (attr4 user2.type3);  
CREATE TABLE tab3 OF type4;
```

Privileges on Type Access and Object Access

While object types only make use of EXECUTE privilege, object tables use all the same privileges as relational tables:

- SELECT lets you access an object and its attributes from the table.
- UPDATE lets you modify attributes of objects in the table.
- INSERT lets you add new objects to the table.
- DELETE lets you delete objects from the table.

Similar table and column privileges regulate the use of table columns of object types.

Selecting columns of an object table does not require privileges on the type of the object table. Selecting the entire row object, however, does.

Consider the following schema:

```
CREATE TYPE emp_type as object (
    eno    NUMBER,
    ename  CHAR(31),
    eaddr  addr_t );
```

```
CREATE TABLE emp OF emp_type;
```

and the following two queries:

```
SELECT VALUE(e) FROM emp e;
SELECT eno, ename FROM emp;
```

For either query, Oracle checks the user's SELECT privilege for the EMP table. For the first query, the user needs to obtain the EMP_TYPE type information to interpret the data. When the query accesses the EMP_TYPE type, Oracle checks the user's EXECUTE privilege.

Execution of the second query, however, does not involve named types, so Oracle does not check type privileges.

Additionally, using the schema from the previous section, USER3 can perform the following queries:

```
SELECT tab1.col1.attr2 from user2.tab1 tab1;
SELECT t.attr4.attr3.attr2 FROM tab3 t;
```

Note that in both selects by USER3, USER3 does not have explicit privileges on the underlying types, but the statement succeeds because the type and table owners have the necessary privileges with the GRANT option.

Oracle checks privileges on the following requests, and returns an error if the requestor does not have the privilege for the action:

- Pinning an object in the object cache using its REF value causes Oracle to check SELECT privilege on the object table containing the object and EXECUTE privilege on the object type. (For more information about the OCI object cache, see ["OCI Tips and Techniques for Objects"](#) on page 6-4.)
- Modifying an existing object or flushing an object from the object cache, causes Oracle to check UPDATE privilege on the destination object table. Flushing a new object causes Oracle to check INSERT privilege on the destination object table.
- Deleting an object causes Oracle to check DELETE privilege on the destination table.

- Invoking a method causes Oracle to check EXECUTE privilege on the corresponding object type.

Oracle does not provide column level privileges for object tables.

Dependencies and Incomplete Types

Types can depend upon each other for their definitions. For example, you might want to define object types EMPLOYEE and DEPARTMENT in such a way that one attribute of EMPLOYEE is the department the employee belongs to and one attribute of DEPARTMENT is the employee who manages the department.

Types that depend on each other in this way, either directly or via intermediate types, are called *mutually dependent*. A diagram of mutually dependent types, with arrows representing the dependencies, always reveals a path of arrows starting and ending at one of the types.

To define such a cyclic dependency, you must use REFs for at least one branch of the cycle.

For example, you can define the following types:

```
CREATE TYPE department;  
  
CREATE TYPE employee AS OBJECT (  
    name    VARCHAR2(30),  
    dept    REF department,  
    supv    REF employee );  
  
CREATE TYPE emp_list AS TABLE OF employee;  
  
CREATE TYPE department AS OBJECT (  
    name    VARCHAR2(30),  
    mgr     REF employee,  
    staff   emp_list );
```

This is a legal set of mutually dependent types and a legal sequence of SQL DDL statements. Oracle compiles it without errors. The first statement:

```
CREATE TYPE department;
```

is optional. It makes the compilation proceed without errors. It establishes DEPARTMENT as an *incomplete object type*. A REF to an incomplete object type compiles without error, so the compilation of EMPLOYEE proceeds.

When Oracle reaches the last statement, which completes the definition of DEPARTMENT, all of the components of DEPARTMENT have compiled successfully, so the compilation finishes without errors.

Without the optional declaration of DEPARTMENT as an incomplete type, EMPLOYEE compiles with errors. Oracle then automatically adds EMPLOYEE to its library of schema objects as an incomplete object type. This makes the declarations of EMP_LIST and DEPARTMENT compile without errors. When EMPLOYEE is recompiled after EMP_LIST and DEPARTMENT are complete, EMPLOYEE compiles without errors and becomes a complete object type.

Completing Incomplete Types

Once you have declared an incomplete object type, you must complete it as an object type. You cannot, for example, declare it to be a table type or an array type. The only alternative is to drop the type.

This is also true if Oracle has made the type an incomplete object type for you—as it did when EMPLOYEE failed to compile in the previous section.

Type Dependencies of Tables

The SQL commands REVOKE and DROP TYPE return an error and abort if the type referred to in the command has tables or other types that depend on it.

The FORCE option with either of these commands overrides that behavior. The command succeeds and the affected tables or types become invalid.

If a table contains data that relies on a type definition for access, any change to the type causes the table's data to become inaccessible. This happens if privileges required by the type are revoked or if the type or a type it depends on is dropped. The table then becomes invalid and cannot be accessed.

A table that is invalid because of missing privileges automatically becomes valid and accessible if the required privileges are re-granted.

A table that is invalid because a type it depends on has been dropped can never be accessed again. The only permissible action is to drop the table.

Import/Export/Load of Object Types

The Export and Import utilities move data into and out of Oracle databases. They also back up or archive data and aid migration to different releases of the Oracle RDBMS.

Export and Import support object types. Export writes object type definitions and all of the associated data to the dump file. Import then re-creates these items from the dump file.

The SQL*Loader supports loading row objects, column objects and objects with collections and references. In Oracle8i, only conventional path loading is supported for objects.

An alternative to conventional path loading is to first load the data into relational tables using direct path loading, and then create the object tables and tables with column objects using `CREATE TABLE...AS SELECT` commands. However, with this approach you need enough space to hold as much as twice the actual data.

See Also: *Oracle8i Utilities* for information about exporting, importing, and loading Oracle objects.

Object Support in Oracle Programmatic Environments

In Oracle8i, you can create object types with SQL data definition language (DDL) commands, and you can manipulate objects with SQL data manipulation language (DML) commands. Object support is built into Oracle's application programming environments:

- [SQL](#)
- [PL/SQL](#)
- [Oracle Call Interface \(OCI\)](#)
- [Pro*C/C++](#)
- [Oracle Type Translator \(OTT\)](#)
- [Oracle Objects For OLE \(for Visual Basic, Excel, ActiveX, Active Server Pages\)](#)
- [Java: JDBC, Oracle SQLJ, and JPublisher](#)

SQL

Oracle SQL DDL provides the following support for object types:

- Defining object types, nested tables, and arrays
- Specifying privileges
- Specifying table columns of user-defined types
- Creating object tables

Oracle SQL DML provides the following support for object types:

- Querying and updating objects and collections
- Manipulating REFs

See Also: For a complete description of Oracle SQL syntax, see *Oracle8i SQL Reference*.

PL/SQL

PL/SQL allows you to use the SQL features that support object types within functions and procedures.

The parameters and variables of PL/SQL functions and procedures can be of object types.

You can implement the methods associated with object types in PL/SQL. These methods (functions and procedures) reside on the server as part of a user's schema.

See Also: For a complete description of PL/SQL, see *PL/SQL User's Guide and Reference*.

Oracle Call Interface (OCI)

OCI is a set of C library functions that applications can use to manipulate data and schemas in an Oracle database. OCI supports both traditional 3GL and object-oriented techniques for database access, as explained in the following sections.

An important component of OCI is a set of calls to manage a workspace called the object cache. The *object cache* is a memory block on the client side that allows programs to store entire objects and to navigate among them without additional round trips to the server.

The object cache is completely under the control and management of the application programs using it. The Oracle server has no access to it. The application programs using it must maintain data coherency with the server and protect the workspace against simultaneous conflicting access.

OCI provides functions to

- Access objects on the server using SQL.
- Access, manipulate and manage objects in the object cache by traversing pointers or REFs.
- Convert Oracle dates, strings and numbers to C data types.

- Manage the size of the object cache's memory.

OCI improves concurrency by allowing individual objects to be locked. It improves performance by supporting complex object retrieval.

OCI developers can use the object type translator to generate the C datatypes corresponding to a Oracle object types.

See Also: *Oracle Call Interface Programmer's Guide* for more information about using objects with OCI.

Associative Access in OCI Programs

Traditionally, 3GL programs manipulate data stored in a relational database by executing SQL statements and PL/SQL procedures. Data is usually manipulated on the server without incurring the cost of transporting the data to the client(s). OCI supports this *associative* access to objects by providing an API for executing SQL statements that manipulate object data. Specifically, OCI allows you to:

- Execute SQL statements that manipulate object data and object type schema information
- Pass object instances, object references (REFs), and collections as input variables in SQL statements
- Return object instances, REFs, and collections as output of SQL statement fetches
- Describe the properties of SQL statements that return object instances, REFs, and collections
- Describe and execute PL/SQL procedures or functions with object parameters or results
- Synchronize object and relational functionality through enhanced commit and rollback functions

See Also: ["Associative Access in Pro*C/C++"](#) on page 3-6

Navigational Access in OCI Programs

In the object-oriented programming paradigm, applications model their real-world entities as a set of inter-related objects that form graphs of objects. The relationships between objects are implemented as references. An application processes objects by starting at some initial set of objects, using the references in these initial objects to traverse the remaining objects, and performing computations on each object. OCI provides an API for this style of access to objects, known as *navigational* access. Specifically, OCI allows you to:

- Cache objects in memory on the client machine
- De-reference an object reference and pin the corresponding object in the object cache. The pinned object is transparently mapped in the host language representation.
- Notify the cache when the pinned object is no longer needed
- Fetch a graph of related objects from the database into the client cache in one call
- Lock objects
- Create, update, and delete objects in the cache
- Flush changes made to objects in the client cache to the database

See Also: ["Navigational Access in Pro*C/C++"](#) on page 3-6

Object Cache

To support high-performance navigational access of objects, OCI runtime provides an object cache for caching objects in memory. The object cache supports references (REFs) to database objects in the object cache, the database objects can be identified (that is, pinned) through their references. Applications do not need to allocate or free memory when database objects are loaded into the cache, because the object cache provides transparent and efficient memory management for database objects.

Also, when database objects are loaded into the cache, they are transparently mapped into the host language representation. For example, in the C programming language, the database object is mapped to its corresponding C structure. The object cache maintains the association between the object copy in the cache and the corresponding database object. Upon transaction commit, changes made to the object copy in the cache are propagated automatically to the database.

The object cache maintains a fast look-up table for mapping `REFs` to objects. When an application de-references a `REF` and the corresponding object is not yet cached in the object cache, the object cache automatically sends a request to the server to fetch the object from the database and load it into the object cache. Subsequent de-references of the same `REF` are faster because they become local cache access and do not incur network round-trips. To notify the object cache that an application is accessing an object in the cache, the application pins the object; when it is finished with the object, it unpins it. The object cache maintains a pin count for each object in the cache. The count is incremented upon a pin call and decremented upon an unpin call. When the pin count goes to zero, it means the object is no longer needed by the application. The object cache uses a least-recently used (LRU) algorithm to manage the size of the cache. When the cache reaches the maximum size, the LRU algorithm frees candidate objects with a pin count of zero.

Building an OCI Program that Manipulates Objects

When you build an OCI program that manipulates objects, you must complete the following general steps:

1. Define the object types that correspond to the application objects.
2. Execute the SQL DDL statements to populate the database with the necessary object types.
3. Represent the object types in the host language format.

For example, to manipulate instances of the object types in a C program, you must represent these types in the C host language format. You can do this by representing the object types as C *structs*. You can use a tool provided by Oracle called the Object Type Translator (OTT) to generate the C mapping of the object types. The OTT puts the equivalent C structs in header (*.h) files. You include these *.h files in the *.c files containing the C functions that implement the application.

4. Construct the application executable by compiling and linking the application's *.c files with the OCI library.

See Also: ["OCI Tips and Techniques for Objects"](#) on page 6-4

Pro*C/C++

The Oracle Pro*C/C++ precompiler allows programmers to use user-defined datatypes in C and C++ programs.

Pro*C developers can use the Object Type Translator to map Oracle object types and collections into C datatypes to be used in the Pro*C application.

Pro*C provides compile time type checking of object types and collections and automatic type conversion from database types to C datatypes.

Pro*C includes an EXEC SQL syntax to create and destroy objects and offers two ways to access objects in the server:

- SQL statements and PL/SQL functions or procedures embedded in Pro*C programs.
- An interface to the object cache (described under "[Oracle Call Interface \(OCI\)](#)" on page 3-2), where objects can be accessed by traversing pointers, then modified and updated on the server.

Additional Information: For a complete description of the Pro*C precompiler, see *Pro*C/C++ Precompiler Programmer's Guide*.

Associative Access in Pro*C/C++

For background information on associative access, see "[Associative Access in OCI Programs](#)" on page 3-3.

Pro*C/C++ offers the following capabilities for associative access to objects:

- Support for transient copies of objects allocated in the object cache
- Support for transient copies of objects referenced as input host variables in embedded SQL INSERT, UPDATE, and DELETE statements, or in the WHERE clause of a SELECT statement
- Support for transient copies of objects referenced as output host variables in embedded SQL SELECT and FETCH statements
- Support for ANSI dynamic SQL statements that reference object types through the DESCRIBE statement, to get the object's type and schema information

Navigational Access in Pro*C/C++

For background information on navigational access, see "[Navigational Access in OCI Programs](#)" on page 3-4.

Pro*C/C++ offers the following capabilities to support a more object-oriented interface to objects:

- Support for de-referencing, pinning, and optionally locking an object in the object cache using an embedded SQL `OBJECT Deref` statement
- Allowing a Pro*C/C++ user to inform the object cache when an object has been updated or deleted, or when it is no longer needed, using embedded SQL `OBJECT UPDATE`, `OBJECT DELETE`, and `OBJECT RELEASE` statements
- Support for creating new referenceable objects in the object cache using an embedded SQL `OBJECT CREATE` statement
- Support for flushing changes made in the object cache to the server with an embedded SQL `OBJECT FLUSH` statement

Converting Between Oracle Types and C Types

The C representation for objects that is generated by the Oracle Type Translator (OTT) uses OCI types whose internal details are hidden, such as `OCIString` and `OCINumber` for scalar attributes. Collection types and object references are similarly represented using `OCITable`, `OCIArray`, and `OCIRef` types. While using these "opaque" types insulates you from changes to their internal formats, using such types in a C or C++ application is cumbersome. Pro*C/C++ provides the following ease-of-use enhancements to simplify use of OCI types in C and C++ applications:

- Object attributes can be retrieved and implicitly converted to C types with the embedded SQL `OBJECT GET` statement.
- Object attributes can be set and converted from C types with the embedded SQL `OBJECT SET` statement.
- Collections can be mapped to a host array with the embedded SQL `COLLECTION GET` statement. Furthermore, if the collection is comprised of scalar types, then the OCI types can be implicitly converted to a compatible C type.
- Host arrays can be used to update the elements of a collection with the embedded SQL `COLLECTION SET` statement. As with the `COLLECTION GET` statement, if the collection is comprised of scalar types, C types are implicitly converted to OCI types.

Oracle Type Translator (OTT)

The Oracle type translator (OTT) is a program that automatically generates C language structure declarations corresponding to object types. OTT makes it easier to use the Pro*C precompiler and the OCI server access package.

Additional Information: For complete information about OTT, see *Oracle Call Interface Programmer's Guide* and *Pro*C/C++ Precompiler Programmer's Guide*.

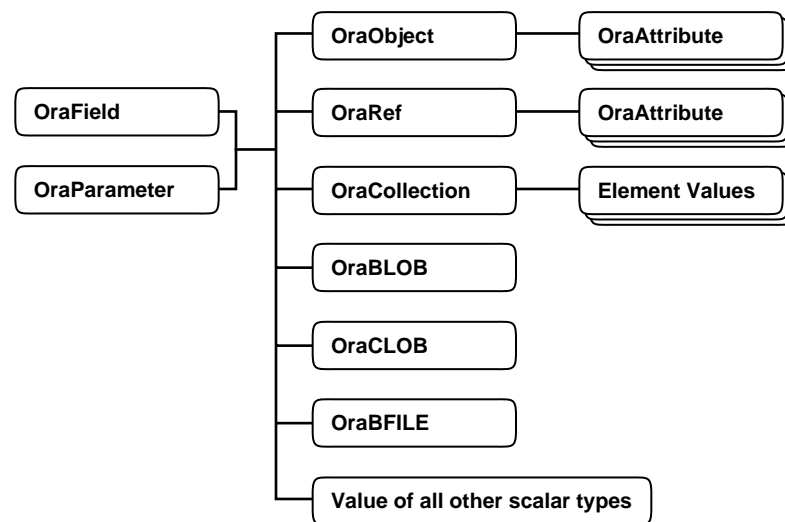
Oracle Objects For OLE (for Visual Basic, Excel, ActiveX, Active Server Pages)

Oracle Objects for OLE (OO4O) provides full support for accessing and manipulating instances of `REFs`, value instances, variable-length arrays (`VARRAYS`), and nested tables in an Oracle database server.

See Also: OO4O online help for detailed information about using OO4O with Oracle objects.

[Figure 3–1](#) illustrates the containment hierarchy for value instances of all types in OO4O.

Figure 3–1 Supported Oracle Datatypes



Instances of these types can be fetched from the database or passed as input or output variables to SQL statements and PL/SQL blocks, including stored procedures and functions. All instances are mapped to COM Automation Interfaces that provide methods for dynamic attribute access and manipulation. These interfaces may be obtained from:

- The value property of an `OraField` object in a Dynaset

- The value property of an OraParameter object used as an input or an output parameter in SQL Statements or PL/SQL blocks
- An attribute of an object (REF)
- An element in a collection (varray or a nested table)

Representing Objects in Visual Basic (OraObject)

The OraObject interface is a representation of an Oracle embedded object or a value instance. It contains a collection interface (OraAttributes) for accessing and manipulating (updating and inserting) individual attributes of a value instance. Individual attributes of an OraAttributes collection interface can be accessed by using a subscript or the name of the attribute.

The following Visual Basic example illustrates how to access attributes of the Address object in the person_tab table:

```
Dim Address OraObject
Set Person = OraDatabase.CreateDynaset("select * from person_tab", 0&)
Set Address = Person.Fields("Addr").Value
msgbox Address.Zip
msgbox Address.City
```

Representing REFs in Visual Basic (OraRef)

The OraRef interface represents an Oracle object reference (REF) as well as referenceable objects in client applications. The object attributes are accessed in the same manner as attributes of an object represented by the OraObject interface. OraRef is derived from an OraObject interface via the containment mechanism in COM. REF objects are updated and deleted independent of the context they originated from, such as Dynasets. The OraRef interface also encapsulates the functionality for navigating through graphs of objects utilizing the Complex Object Retrieval Capability (COR) in OCI, described in ["Pre-Fetching Related Objects \(Complex Object Retrieval\)"](#) on page 6-9.

Representing VARRAYs and Tables in Visual Basic (OraCollection)

The OraCollection interface provides methods for accessing and manipulating Oracle collection types, namely variable-length arrays (VARRAYs) and nested tables in OO4O. Elements contained in a collection are accessed by subscripts.

The following Visual Basic example illustrates how to access attributes of the EnameList object from the department table:

```
Dim EnameList OraCollection
Set Person = OraDatabase.CreateDynaset("select * from department", 0&)
set EnameList = Department.Fields("Enames").Value
'access all elements of the EnameList VArray
for I=1 to I=EnameList.Size
    msgbox EnameList(I)
Next I
```

Java: JDBC, Oracle SQLJ, and JPublisher

Java has emerged as a powerful, modern object-oriented language that provides developers with a simple, efficient, portable, and safe application development platform. Oracle provides two ways to integrate Oracle object features with Java: JDBC and Oracle SQLJ. The following sections provide more information about these environments.

JDBC Access to Oracle Object Data

JDBC (Java Database Connectivity) is a set of Java interfaces to the Oracle server. Oracle provides tight integration between objects and JDBC. You can map SQL types to Java classes with considerable flexibility.

Oracle's JDBC:

- Allows access to objects and collection types (defined in the database) in Java programs through dynamic SQL.
- Translates types defined in the database into Java classes through default or customizable mappings.

Version 2.0 of the JDBC specification supports Object-Relational constructs such as user-defined (Object) types. JDBC materializes Oracle objects as instances of particular Java classes. Using JDBC to access Oracle objects involves creating the Java classes for the Oracle objects and populating these classes. You can either:

- Let JDBC materialize the object as a `STRUCT`. In this case, JDBC creates the classes for the attributes and populates them for you.
- Personally specify the mappings between Oracle objects and Java classes; that is, customize your Java classes for object data. The driver then populates the customized Java classes that you specify, which imposes a set of constraints on the Java classes. To satisfy these constraints, you can choose to define your classes according to either the `SQLData` interface or the `CustomDatum` interface.

Additional Information: For complete information about JDBC, see the *Oracle8i JDBC Developer's Guide and Reference*.

SQLJ Access to Oracle Object Data

SQLJ provides access to server objects using SQL statements embedded in the Java code:

- You can use user-defined types in Java programs.
- You can use JPublisher to map Oracle object and collection types into Java classes to be used in the application.
- The object types and collections in the SQL statements are checked at compile time.

Additional Information: For complete information about SQLJ, see the *Oracle8i Java Developer's Guide*.

Choosing a Data Mapping Strategy

Oracle SQLJ supports either strongly typed or weakly typed Java representations of object types, reference types (REFs), and collection types (VARARRAYs and nested tables) to be used in iterators or host expressions.

Strongly typed representations use a *custom Java class* that corresponds to a particular object type, reference type, or collection type and must implement the interface `oracle.sql.CustomDatum`. The Oracle JPublisher utility can automatically generate such custom Java classes.

Weakly typed representations use the class `oracle.sql.STRUCT` (for objects), `oracle.sql.REF` (for references), or `oracle.sql.ARRAY` (for collections).

See Also: ["Manipulating Objects Through Java"](#) on page 8-39 for sample code showing both techniques.

Using JPublisher to Create Java Classes for JDBC and SQLJ Programs

Oracle lets you map Oracle object types, reference types, and collection types to Java classes and preserve all the benefits of strong typing. You can:

- Use JPublisher to automatically generate custom Java classes and use those classes without any change.
- Subclass the classes produced by JPublisher to create your own specialized Java classes.
- Manually code custom Java classes without using JPublisher, provided that the classes meet the requirements stated in the *Oracle8i SQLJ Developer's Guide and Reference*.

We recommend that you use JPublisher, and subclass when the generated classes do not do everything you need.

What JPublisher Produces

When you run JPublisher for a user-defined object type, it automatically creates the following:

- A custom object class to act as a type definition to correspond to your Oracle object type

This class includes getter and setter methods for each attribute. The method names are of the form `getFoo()` and `setFoo()` for attribute `foo`.

Also, you can optionally instruct JPublisher to generate wrapper methods in your class that invoke the associated Oracle object methods executing in the server.

- A related custom reference class for object references to your Oracle object type

This class includes a `getValue()` method that returns an instance of your custom object class, and a `setValue()` method that updates an object value in the database, taking as input an instance of the custom object class.

When you run JPublisher for a user-defined collection type, it automatically creates the following:

- A custom collection class to act as a type definition to correspond to your Oracle collection type

This class includes overloaded `getArray()` and `setArray()` methods to retrieve or update a collection as a whole, a `getElement()` method and `setElement()` method to retrieve or update individual elements of a collection, and additional utility methods.

JPublisher-produced custom Java classes in any of these categories implement the `CustomDatum` interface, the `CustomDatumFactory` interface, and the `getFactory()` method.

See Also: The *Oracle8i JPublisher User's Guide* for more information about using JPublisher.

Applying an Object Model to Relational Data

This chapter shows how to write object-oriented applications without changing the underlying structure of your relational data:

- [Why to Use Object Views](#)
- [Defining Object Views](#)
- [Using Object Views in Applications](#)
- [Nesting Objects in Object Views](#)
- [Identifying Null Objects in Object Views](#)
- [Using Nested Tables and Varrays in Object Views](#)
- [Specifying Object Identifiers for Object Views](#)
- [Creating References to View Objects](#)
- [Modelling Inverse Relationships with Object Views](#)
- [Updating Object Views](#)
- [Applying the Object Model to Remote Tables](#)
- [Defining Complex Relationships in Object Views](#)

Why to Use Object Views

Just as a view is a virtual table, an *object view* is a virtual object table. Each row in the view is an object: you can call its methods, access its attributes using the dot notation, and create a REF that points to it.

Object views are useful in prototyping or transitioning to object-oriented applications, because the data in the view can be taken from relational tables and accessed as if the table were defined as an object table. You can run object-oriented applications without converting existing tables to a different physical structure.

Object views provide the same features as traditional views, applied to object data. For example, you might provide an object view of an employee table that doesn't have attributes containing sensitive data and doesn't have a deletion method.

Using object views can lead to better performance. Relational data that make up a row of an object view traverse the network as a unit, potentially saving many round trips.

You can fetch relational data into the client-side object cache and map it into C structs or C++ or Java classes, so 3GL applications can manipulate it just like native classes. You can also use object-oriented features like complex object retrieval with relational data.

- By synthesizing objects from relational data, you can query the data in new ways. You can view data from multiple tables by using object de-referencing instead of writing complex joins with multiple tables.
- Since the objects in the view are processed within the server, not on the client, this can result in significantly fewer SQL statements and much less network traffic.
- The object data from object views can be pinned and used in the client side object cache. When you retrieve these synthesized objects in the object cache by means of specialized object-retrieval mechanisms, you reduce network traffic.
- You gain great flexibility when you create an object model within a view in that you can continue to develop the model. If you need to alter an object type, you can simply replace the invalidated views with a new definition.
- Using objects in views does not place any restrictions on the characteristics of the underlying storage mechanisms. By the same token, you are not limited by the restrictions of current technology. For example, you can synthesize objects from relational tables which are parallelized and partitioned.
- You can create different complex data models from the same underlying data.

See Also:

- *Oracle8i SQL Reference* for a complete description of SQL syntax and usage.
 - *PL/SQL User's Guide and Reference* for a complete discussion of PL/SQL capabilities
 - *Oracle8i Java Stored Procedures Developer's Guide* for a complete discussion of Java.
 - *Oracle Call Interface Programmer's Guide* for a complete discussion of those facilities.
-

Defining Object Views

The procedure for defining an object view is:

1. Define an object type, where each attribute of the type corresponds to an existing column in a relational table.
2. Write a query that specifies how to extract the data from relational tables. Specify the columns in the same order as the attributes in the object type.
3. Specify a unique value, based on attributes of the underlying data, to serve as an object identifier, which allows you to create pointers (REFs) to the objects in the view. You can often use an existing primary key.

If you want to be able to update an object view, you may have to take another step, if the attributes of the object type do not correspond exactly to columns in existing tables:

4. Write an INSTEAD OF trigger procedure (see ["Updating Object Views"](#) on page 4-11) for Oracle to execute whenever an application program tries to update data in the object view.

After these steps, you can use an object view just like an object table.

For example, the following SQL statements define an object view, where each row in the view is an object of type EMPLOYEE_T:

```
CREATE TABLE emp_table (
    empnum    NUMBER (5),
    ename     VARCHAR2 (20),
    salary    NUMBER (9, 2),
    job       VARCHAR2 (20) );
```

```
CREATE TYPE employee_t (
```

```
empno    NUMBER (5),
ename    VARCHAR2 (20),
salary   NUMBER (9, 2),
job       VARCHAR2 (20) );

CREATE VIEW emp_view1 OF employee_t
WITH OBJECT IDENTIFIER (empno) AS
  SELECT  e.empnum, e.ename, e.salary, e.job
  FROM    emp_table e
  WHERE   job = 'Developer';
```

To access the data from the EMPNUM column of the relational table, you would access the EMPNO attribute of the object type.

Using Object Views in Applications

Data in the rows of an object view may come from more than one table, but the object still traverses the network in one operation. When the instance is in the client side object cache, it appears to the programmer as a C or C++ structure or as a PL/SQL object variable. You can manipulate it like any other native structure.

You can refer to object views in SQL statements the same way you refer to an object table. For example, object views can appear in a SELECT list, in an UPDATE-SET clause, or in a WHERE clause.

You can also define object views on object views.

You can access object view data on the client side using the same OCI calls you use for objects from object tables. For example, you can use *OCIObjectPin()* for pinning a REF and *OCIObjectFlush()* for flushing an object to the server. When you update or flush to the server an object in an object view, Oracle updates the object view.

Additional Information: See *Oracle Call Interface Programmer's Guide* for more information about OCI calls.

Nesting Objects in Object Views

If one of the attributes of an object type is itself an object type, you must extract suitable column objects from the relational data. You can either select them from a column object that already exists in the relational table, or synthesize them from a set of relational columns using the appropriate type constructor.

For example, consider the department table `dept`:

```
CREATE TABLE dept
(
    deptno      NUMBER PRIMARY KEY,
    deptname    VARCHAR2(20),
    deptstreet  VARCHAR2(20),
    deptcity    VARCHAR2(10),
    deptstate   CHAR(2),
    deptzip     VARCHAR2(10)
);
```

You might want to create an object view where the addresses are objects inside the department objects. That would allow you to define reusable methods for address objects, and use them for all kinds of addresses.

1. Create the type for the address object:

```
CREATE TYPE address_t AS OBJECT
(
    street  VARCHAR2(20),
    city    VARCHAR2(10),
    state   CHAR(2),
    zip     VARCHAR2(10)
);
/
```

2. Create the view containing the department number, name and address. The address is constructed from several columns of the relational table.

```
CREATE VIEW dept_view AS
    SELECT d.deptno, d.deptname,
           address_t(d.deptstreet,d.deptcity,d.deptstate,d.deptzip) AS
deptaddr
    FROM   dept d;
```

Identifying Null Objects in Object Views

Because the constructor for an object never returns a null, none of the address objects in the above view can ever be null, even if the city, street, and so on columns in the relational table are all null. The relational table has no column that specifies whether the department address is null. If we define a convention so that a null `deptstreet` column indicates that the whole address is null, then we can capture

the logic using the DECODE function, or some other function, to return either a null or the constructed object:

```
CREATE VIEW dept_view AS
  SELECT d.deptno, d.deptname,
         DECODE(d.deptstreet, NULL, NULL,
               address_t(d.deptstreet, d.deptcity, d.deptstate, d.deptzip)) AS
deptaddr
  FROM dept d;
```

Using such a technique makes it impossible to directly update the department address through the view, because it does not correspond directly to a column in the relational table. Instead, we would define an INSTEAD-OF trigger over the view to handle updates to this column.

Using Nested Tables and Varrays in Object Views

Collections, both nested tables and VARRAYs, can be columns in views. You can select these collections from underlying collection columns or you can synthesize them using subqueries. The CAST-MULTISET operator provides a way of synthesizing such collections.

Taking the previous example as our starting point, we represent each employee in an emp relational table with following structure:

```
CREATE TABLE emp
(
  empno    NUMBER PRIMARY KEY,
  empname  VARCHAR2(20),
  salary   NUMBER,
  deptno   NUMBER REFERENCES dept(deptno)
);
```

Using this relational table, we can construct a dept_view with the department number, name, address and a collection of employees belonging to the department.

1. Define a employee type and a nested table type for the employee type:

```
CREATE TYPE employee_t AS OBJECT
(
  eno NUMBER,
  ename VARCHAR2(20),
  salary NUMBER
);
```

```
CREATE TYPE employee_list_t AS TABLE OF employee_t;
```

2. The dept_view can now be defined:

```
CREATE VIEW dept_view AS
  SELECT d.deptno, d.deptname,
         address_t(d.deptstreet,d.deptcity,d.deptstate,d.deptzip) AS deptaddr,
         CAST( MULTISET (
                   SELECT e.empno, e.empname, e.salary
                   FROM emp e
                   WHERE e.deptno = d.deptno)
              AS employee_list_t)
         AS emp_list
  FROM   dept d;
```

The `SELECT` subquery inside the `CAST-MULTISSET` block selects the list of employees that belong to the current department. The `MULTISSET` keyword indicates that this is a list as opposed to a singleton value. The `CAST` operator casts the result set into the appropriate type, in this case to the `employee_list_t` collection type.

A query on this view could give us the list of departments, with each department row containing the department number, name, the address object and a collection of employees belonging to the department.

Specifying Object Identifiers for Object Views

You can construct pointers (REFs) to the row objects in an object view. Since the view data is not stored persistently, you must specify a set of distinct values to be used as object identifiers. The notion of object identifiers allows the objects in object views to be referenced and pinned in the object cache.

If the view is based on an object table or an object view, then there is already an object identifier associated with each row and you can reuse them. Either omit the `WITH OBJECT IDENTIFIER` clause, or specify `WITH OBJECT IDENTIFIER DEFAULT`.

However, if the row object is synthesized from relational data, you must choose some other set of values.

Oracle lets you specify object identifiers based on the primary key. The set of unique keys that identify the row object is turned into an identifier for the object. These

values must be unique within the rows selected out of the view, since duplicates would lead to problems during navigation through object references.

The object view created with the `WITH OBJECT IDENTIFIER` clause has an object identifier derived from the primary key. If the `WITH OBJECT IDENTIFIER DEFAULT` clause is specified, the object identifier is either system generated or primary key based, depending on the underlying table or view definition.

Continuing with our department example, we can create a `dept_view` object view that uses the department number as the object identifier:

Define the object type for the row, in this case the `dept_t` department type:

```
CREATE TYPE dept_t AS OBJECT
(
    dno          NUMBER,
    dname        VARCHAR2(20),
    deptaddr     address_t,
    emplist      employee_list_t
);
```

Because the underlying relational table has `deptno` as the primary key, each department row has a unique department number. In the view, the `deptno` column becomes the `dno` attribute of the object type. Once we know that `dno` is unique within the view objects, we can specify it as the object identifier:

```
CREATE VIEW dept_view OF dept_t WITH OBJECT IDENTIFIER(dno)
AS SELECT d.deptno, d.deptname,
        address_t(d.deptstreet,d.deptcity,d.deptstate,d.deptzip),
        CAST( MULTISSET (
                SELECT e.empno, e.empname, e.salary
                FROM emp e
                WHERE e.deptno = d.deptno)
        AS employee_list_t)
FROM    dept d;
```

See Also: [Object Identifiers](#) on page 6-3

Creating References to View Objects

In the example we have been developing, each object selected out of the `dept_view` view has a unique object identifier derived from the department number value. In the relational case, the foreign key `deptno` in the `emp` employee table matches the `deptno` primary key value in the `dept` department table. We used the primary key value for creating the object identifier in the `dept_view`. This allows us to use the foreign key value in the `emp_view` in creating a reference to the primary key value in `dept_view`.

We accomplish this by using `MAKE_REF` operator to synthesize a primary key object reference. This takes the view or table name to which the reference points and a list of foreign key values to create the object identifier portion of the reference that will match with a particular object in the referenced view.

In order to create an `emp_view` view which has the employee's number, name, salary and a reference to the department in which she works, we need first to create the employee type `emp_t` and then the view based on that type

```
CREATE TYPE emp_t AS OBJECT
(
  eno      NUMBER,
  ename    VARCHAR2(20),
  salary   NUMBER,
  deptref  REF dept_t
);

CREATE VIEW emp_view OF emp_t WITH OBJECT IDENTIFIER(enno)
AS SELECT e.empno, e.empname, e.salary,
         MAKE_REF(dept_view, e.deptno)
FROM emp e;
```

The `deptref` column in the view holds the department reference. We write the following simple query to determine all employees whose department is located in the city of San Francisco:

```
SELECT e.eno, e.salary, e.deptref.dno
FROM emp_view e
WHERE e.deptref.deptaddr.city = 'San Francisco';
```

Note that we could also have used the `REF` modifier to get the reference to the `dept_view` objects:

```
CREATE VIEW emp_view OF emp_t WITH OBJECT IDENTIFIER(enno)
AS SELECT e.empno, e.empname, e.salary, REF(d)
```

```
FROM emp e, dept_view d
WHERE e.deptno = d.dno;
```

In this case we join the `dept_view` and the `emp` table on the `deptno` key. The advantage of using `MAKE_REF` operator instead of the `REF` modifier is that in using the former, we can create circular references. For example, we can create employee view to have a reference to the department in which she works, and the department view can have a list of references to the employees who work in that department.

Note that if the object view has a primary key based object identifier, the reference to such a view is primary key based. On the other hand, a reference to a view with system generated object identifier will be a system generated object reference. This difference is only relevant when you create object instances in the OCI object cache and need to get the reference to the newly created objects. This is explained in a later section.

As with synthesized objects, we can also select persistently stored references as view columns and use them seamlessly in queries. However, the object references to view objects cannot be stored persistently.

Modelling Inverse Relationships with Object Views

Views with objects can be used to model inverse relationships.

One-to-One Relationships

One-to-one relationships can be modeled with inverse object references. For example, let us say that each employee has a particular computer on her desk, and that the computer belongs to that employee only. A relational model would capture this using foreign keys either from the computer table to the employee table, or in the reverse direction. Using views, we can model the objects so that we have an object reference from the employee to the computer object and also have a reference from the computer object to the employee.

One-to-Many and One-to-Many Relationships

One-to-many relationships (or many-to-many relationships) can be modeled either by using object references or by embedding the objects. One-to-many relationship can be modeled by having a collection of objects or object references. The many-to-one side of the relationship can be modeled using object references.

Consider the department-employee case. In the underlying relational model, we have the foreign key in the employee table. Using collections in views, we can model the relationship between departments and employees. The department view

can have a collection of employees, and the employee view can have a reference to the department (or inline the department values). This gives us both the forward relation (from employee to department) and the inverse relation (department to list of employees). The department view can also have a collection of references to employee objects instead of embedding the employee objects.

Updating Object Views

You can update, insert, and delete the data in an object view using the same SQL DML you use for object tables. Oracle updates the base tables of the object view if there is no ambiguity.

A view is not updatable if its view query contains joins, set operators, aggregate functions, GROUP BY, or DISTINCT. If a view query contains pseudocolumns or expressions, the corresponding view columns are not updatable. Object views often involve joins.

To overcome these obstacles Oracle provides *INSTEAD OF triggers*. They are called INSTEAD OF triggers because Oracle executes the trigger body instead of the actual DML statement.

INSTEAD OF triggers provide a transparent way to update object views or relational views. You write the same SQL DML (INSERT, DELETE, and UPDATE) statements as for an object table. Oracle invokes the appropriate trigger instead of the SQL statement, and the actions specified in the trigger body take place.

Updating Nested Table Columns in Views

A nested table can be modified by inserting new elements and updating or deleting existing elements. Nested table columns that are virtual or synthesized, as in a view, are not usually updatable. To overcome this, Oracle allows INSTEAD OF triggers to be created on these columns.

The INSTEAD OF trigger defined on a nested table column (of a view) is fired when the column is modified. Note that if the entire collection is replaced (by an update of the parent row), the INSTEAD OF trigger on the nested table column is not fired.

Using INSTEAD-OF Triggers to Control Mutating and Validation

INSTEAD-OF triggers provide a way of updating complex views that otherwise could not be updated. They can also be used to enforce constraints, check privileges and validate the DML. Using these triggers, you can control mutation of the objects

created though an object view that might be caused by inserting, updating and deleting.

For instance, suppose we wanted to enforce the condition that the number of employees in a department cannot exceed 10. To enforce this, we can write an `INSTEAD-OF` trigger for the employee view. The trigger is not needed for doing the DML since the view can be updated, but we need it to enforce the constraint.

We implement the trigger by means of the following code:

```
CREATE TRIGGER emp_instr INSTEAD OF INSERT on emp_view
FOR EACH ROW
DECLARE
    dept_var dept_t;
    emp_count integer;
BEGIN
    -- Enforce the constraint..!
    -- First get the department number from the reference
    UTL_REF.SELECT_OBJECT(:NEW.deptref,dept_var);

    SELECT COUNT(*) INTO emp_count
    FROM emp
    WHERE deptno = dept_var.dno;

    IF emp_count < 9 THEN
        -- let us do the insert
        INSERT INTO emp VALUES (:NEW.eno,:NEW.ename,:NEW.salary,dept_var.dno);
    END IF;
END;
```

Applying the Object Model to Remote Tables

Although you cannot directly access remote tables as object tables, object views let you access remote tables as if they were object tables.

Consider a company with two branches — one in Washington D.C., and another in Chicago. Each site has an employee table. The headquarters in Washington has a department table with the list of all the departments. To get a total view of the entire organization, we can create views over the individual remote tables and then a overall view of the organization.

First, we create an object view for each employee table:

```
CREATE VIEW emp_washington_view (eno,ename,salary)
AS SELECT e.empno, e.empname, e.salary
```



```
FROM emp@washington_link e;

CREATE VIEW emp_chicago_view
AS SELECT e.eno, e.name, e.salary
FROM emp_tab@chicago_link e;
```

We can now create the global view:

```
CREATE VIEW orgnzn_view OF dept_t WITH OBJECT IDENTIFIER (dno)
AS SELECT d.deptno, d.deptname,
        address_t(d.deptstreet,d.deptcity,d.deptstate,d.deptzip),
        CAST( MULTISET (
                SELECT e.eno, e.ename, e.salary
                FROM emp_washington_view e)
        AS employee_list_t)
FROM dept d
WHERE d.deptcity = 'Washington'
UNION ALL
SELECT d.deptno, d.deptname,
        address_t(d.deptstreet,d.deptcity,d.deptstate,d.deptzip),
        CAST( MULTISET (
                SELECT e.eno, e.name, e.salary
                FROM emp_chicago_view e)
        AS employee_list_t)
FROM dept d
WHERE d.deptcity = 'Chicago';
```

This view has the list of all employees for each department. We use UNION ALL since we cannot have two employees working in more than one department. If we had to deal with that eventuality, we could use a UNION of the rows. However, one caveat in using the UNION operator is that we need to introduce an ORDER BY operator within the CAST-MULTISSET expressions so that the comparison of two collections is performed properly.

Defining Complex Relationships in Object Views

You can define circular references in object views using the MAKE_REF operator: view_A can refer to view_B which in turn can refer to view_A. This allows an object view to synthesize a complex structure such as a graph from relational data.

For example, in the case of the department and employee, the department object currently includes a list of employees. To conserve space, we may want to put references to the employee objects inside the department object, instead of

materializing all the employees within the department object. We can construct ("pin") the references to employee objects, and later follow the references using the dot notation to extract employee information.

Because the employee object already has a reference to the department in which the employee works, an object view over this model contains circular references between the department view and the employee view.

You can create circular references between object views in two different ways.

Method 1: Create Both Views

1. Create view A without any reference to view B.
2. Create view B, which includes a reference to view A.
3. Replace view A with a new definition that includes the reference to view B.

Method 2:

1. Create view A with the reference to view B using the `FORCE` keyword.
2. Create view B with reference to view A. When view A is used, it is validated and re-compiled.

Method 2 has fewer steps, but the `FORCE` keyword may hide errors in the view creation. You need to query the `USER_ERRORS` catalog view to see if there were any errors during the view creation. Use this method only if you are sure that there are no errors in the view creation statement.

Also, if errors prevent the views from being recompiled upon use, you must recompile them manually using the `ALTER VIEW COMPILE` command.

We will see the implementation for both the methods.

Tables and Types to Demonstrate Circular View References

First, we set up some relational tables and associated object types. Although the tables contain some objects, they are not object tables. To access the data objects, we will create object views later.

The `emp` table stores the employee information:

```
CREATE TABLE emp
(
    empno    NUMBER PRIMARY KEY,
    empname  VARCHAR2(20),
    salary   NUMBER,
```

```
    deptno    NUMBER
);
```

The *emp_t* type contains a reference to the department. We need a dummy department type so that the *emp_t* type creation succeeds.

```
CREATE TYPE dept_t;
/
```

The employee type includes a reference to the department:

```
CREATE TYPE emp_t AS OBJECT
(
    eno NUMBER,
    ename VARCHAR2(20),
    salary NUMBER,
    deptref REF dept_t
);
/
```

We represent the list of references to employees as a nested table:

```
CREATE TYPE employee_list_ref_t AS TABLE OF REF emp_t;
/
```

The department table is a typical relational table:

```
CREATE TABLE dept
(
    deptno          NUMBER PRIMARY KEY,
    deptname        VARCHAR2(20),
    deptstreet      VARCHAR2(20),
    deptcity        VARCHAR2(10),
    deptstate       CHAR(2),
    deptzip         VARCHAR2(10)
);
```

To create object views, we need object types that map to columns from the relational tables:

```
CREATE TYPE address_t AS OBJECT
(
    street          VARCHAR2(20),
    city            VARCHAR2(10),
    state           CHAR(2),
    zip             VARCHAR2(10)
);
```

```
);  
/
```

We earlier created an incomplete type; now we fill in its definition:

```
CREATE OR REPLACE TYPE dept_t AS OBJECT  
(  
    dno          NUMBER,  
    dname        VARCHAR2(20),  
    deptaddr     address_t,  
    empreflist   employee_list_ref_t  
);  
/
```

Creating Object Views with Circular References

Now that we have the underlying relational table definitions, we create the object views on top of them.

Method 1: Views compiled now.

We first create the employee view with a null in the *deptref* column. Later, we will turn that column into a reference.

```
CREATE VIEW emp_view OF emp_t WITH OBJECT IDENTIFIER(eno)  
    AS SELECT e.empno, e.empname, e.salary,  
              NULL  
    FROM emp e;
```

Next, we create the department view, which includes references to the employee objects.

```
CREATE VIEW dept_view OF dept_t WITH OBJECT IDENTIFIER(dno)  
    AS SELECT d.deptno, d.deptname,  
              address_t(d.deptstreet,d.deptcity,d.deptstate,d.deptzip),  
              CAST( MULTISSET (  
                  SELECT MAKE_REF(emp_view, e.empno)  
                  FROM emp e  
                  WHERE e.deptno = d.deptno)  
              AS employee_list_ref_t)  
    FROM dept d;
```

We create a list of references to employee objects, instead of including the entire employee object. We now re-create the employee view with the reference to the department view.

```
CREATE OR REPLACE VIEW emp_view OF emp_t WITH OBJECT IDENTIFIER(eno)
  AS SELECT e.empno, e.empname, e.salary,
           MAKE_REF(dept_view, e.deptno)
  FROM emp e;
```

This creates the views.

Method 2: Views compiled upon use.

If we are sure that the view creation statement has no syntax errors, we can use the **FORCE** keyword to force the creation of the first view without the other view being present.

First, we create an employee view that includes a reference to the department view, which does not exist at this point. This view cannot be queried until the department view is created properly.

```
CREATE FORCE VIEW emp_view OF emp_t WITH OBJECT IDENTIFIER(eno)
  AS SELECT e.empno, e.empname, e.salary,
           MAKE_REF(dept_view, e.deptno)
  FROM emp e;
```

Next, we create a department view that includes references to the employee objects. We do not have to use the **FORCE** keyword here, since **emp_view** already exists.

```
CREATE VIEW dept_view OF dept_t WITH OBJECT IDENTIFIER(dno)
  AS SELECT d.deptno, d.deptname,
           address_t(d.deptstreet,d.deptcity,d.deptstate,d.deptzip),
           CAST( MULTISSET (
               SELECT MAKE_REF(emp_view, e.empno)
               FROM emp e
               WHERE e.deptno = d.deptno)
           AS employee_list_ref_t)
  FROM dept d;
```

This allows us to query the department view, getting the employee object by de-referencing the employee reference from the nested table **empreflist**:

```
SELECT Deref(e.COLUMN_VALUE)
FROM TABLE( SELECT e.empreflist FROM dept_view e WHERE e.dno = 100) e;
```

`COLUMN_VALUE` is a special name that represents the scalar value in a scalar nested table. In this case, `COLUMN_VALUE` denotes the reference to the employee objects in the nested table `empreflist`.

We can also access only the employee number of all those employees whose name begins with “John”.

```
SELECT e.COLUMN_VALUE.eno
FROM TABLE(SELECT e.empreflist FROM dept_view e WHERE e.dno = 100) e
WHERE e.COLUMN_VALUE.ename like 'John%';
```

To get a tabular output, unnest the list of references by joining the department table with the items in its nested table:

```
SELECT d.dno, e.COLUMN_VALUE.eno, e.COLUMN_VALUE.ename
FROM dept_view d, TABLE(d.empreflist) e
WHERE e.COLUMN_VALUE.ename like 'John%'
AND d.dno = 100;
```

Finally, we can rewrite the above query to use the `emp_view` instead of the `dept_view` to show how you can navigate from one view to the other:

```
SELECT e.deptref.dno, Deref(f.COLUMN_VALUE)
FROM emp_view e, TABLE(e.deptref.empreflist) f
WHERE e.deptref.dno = 100
AND f.COLUMN_VALUE.ename like 'John%';
```

Design Considerations for Oracle Objects

This chapter explains the implementation and performance characteristics of Oracle's object-relational model. Use this information to map a logical data model into an Oracle physical implementation, and when developing applications that use object-oriented features.

This chapter covers the following topics:

- [Representing Objects as Columns or Rows](#)
- [Storage Considerations for Object Identifiers \(OIDs\)](#)
- [Viewing Object Data in Relational Form with Unnesting Queries](#)
- [Choosing a Language for Method Functions](#)

You should be familiar with the basic concepts behind Oracle objects before you read this chapter.

See Also: *Oracle8i Concepts* for conceptual information about Oracle objects, and see *Oracle8i SQL Reference* for information about the SQL syntax for using Oracle objects.

Representing Objects as Columns or Rows

You can store objects in columns of relational tables as column objects, or in object tables as row objects. Objects that have meaning outside of the relational database object in which they are contained, or objects that are shared among more than one relational database object, should be made referenceable as row objects. That is, such objects should be stored in an object table instead of in a column of a relational table.

For example, an object of object type `CUSTOMER` has meaning outside of any particular purchase order, and should be referenceable; therefore, `CUSTOMER` objects should be stored as row objects in an object table. An object of object type `ADDRESS`, however, has little meaning outside of a particular purchase order and can be one attribute within a purchase order; therefore, `ADDRESS` objects should be stored as column objects in columns of relational tables or object tables. So, `ADDRESS` might be a column object in the `CUSTOMER` row object.

Column Object Storage

The storage of a column object is the same as the storage of an equivalent set of scalar columns that collectively make up the object. The only difference is that there is the additional overhead of maintaining the atomic null values of the object and its embedded object attributes. These values are called *null indicators* because, for every column object, a null indicator specifies whether the column object is null and whether each of its embedded object attributes is null. However, null indicators do not specify whether the scalar attributes of a column object are null. Oracle uses a different method to determine whether scalar attributes are null.

Consider a table that holds the identification number, name, address, and phone numbers of people within an organization. You can create three different object types to hold the name, address, and phone number. First, to create the `name_objtyp` object type, enter the following SQL statement:

```
CREATE TYPE name_objtyp AS OBJECT (  
  first      VARCHAR2(15),  
  middle     VARCHAR2(15),  
  last       VARCHAR2(15));
```

Figure 5–1 Object Relational Representation for the `name_objtyp` Type

Type NAME_OBJTYP		
FIRST	MIDDLE	LAST
Text VARCHAR2(15)	Text VARCHAR2(15)	Text VARCHAR2(15)

Next, to create the address_objtyp object type, enter the following SQL statement:

```
CREATE TYPE address_objtyp AS OBJECT (  
    street      VARCHAR2(200),  
    city        VARCHAR2(200),  
    state       CHAR(2),  
    zipcode     VARCHAR2(20));
```

Figure 5–2 Object Relational Representation of the address_objtyp Type

Type ADDRESS_OBJTYP			
STREET	CITY	STATE	ZIP
Text VARCHAR2(200)	Text VARCHAR2(200)	Text CHAR(2)	Number VARCHAR2(20)

Finally, to create the phone_objtyp object type, enter the following SQL statement:

```
CREATE TYPE phone_objtyp AS OBJECT (  
    location    VARCHAR2(15),  
    num         VARCHAR2(14));
```

Figure 5–3 Object Relational Representation of the phone_objtyp Type

Type PHONE_OBJTYP	
LOCATION	NUM
Text VARCHAR2(15)	Number VARCHAR2(14)

Because each person may have more than one phone number, create a nested table type `phone_ntabtyp` based on the `phone_objtyp` object type:

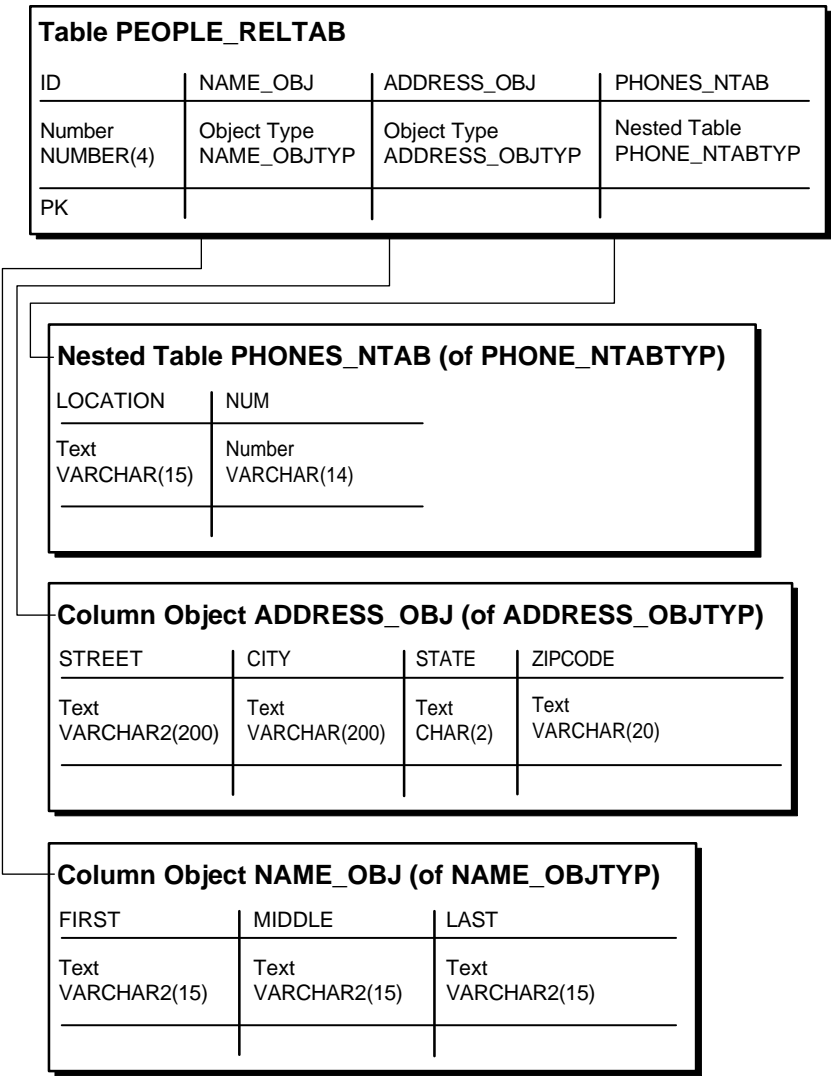
```
CREATE TYPE phone_ntabtyp AS TABLE OF phone_objtyp;
```

See Also: ["Nested Tables"](#) on page 5-14 for more information about nested tables.

Once all of these object types are in place, you can create a table to hold the information about the people in the organization with the following SQL statement:

```
CREATE TABLE people_reltab (  
    id          NUMBER(4)    CONSTRAINT pk_people_reltab PRIMARY KEY,  
    name_obj    name_objtyp,  
    address_obj address_objtyp,  
    phones_ntab phone_ntabtyp)  
NESTED TABLE phones_ntab STORE AS phone_store_ntab;
```

Figure 5–4 Representation of the people_reltab Relational Table



The `people_reltab` table has three column objects: `name_obj`, `address_obj`, and `phones_ntab`. The `phones_ntab` column object is also a nested table.

Note: The `name_obj` object, `address_obj` object, `phones_ntab` nested table, and `people_reltab` table are used in examples throughout this chapter.

The storage for each object stored in the `people_reltab` table is the same as that of the attributes of the object. For example, the storage required for a `name_obj` object is the same as the storage for the `first`, `middle`, and `last` attributes combined, except for the null indicator overhead.

If the `COMPATIBLE` parameter is set to 8.1.0 or higher, the null indicator for an object and its embedded object attributes occupy one bit each. Thus, an object with n embedded object attributes (including objects at all levels of nesting) has a storage overhead of $\text{CEIL}(n/8)$ bytes. In the `people_reltab` table, for example, the overhead of the null information for each row is one byte because it translates to $\text{CEIL}(3/8)$ or $\text{CEIL}(.37)$, which rounds up to one byte. In this case, there are three objects in each row: `name_obj`, `address_obj`, and `phones_ntab`.

If, however, the `COMPATIBLE` parameter is set to a value below 8.1.0, such as 8.0.0, the storage is determined by the following calculation:

$$\text{CEIL}(n/8) + 6$$

Here, n is the total number of all attributes (scalar and object) within the object. Therefore, in the `people_reltab` table, for example, the overhead of the null information for each row is seven bytes because it translates to the following calculation:

$$\text{CEIL}(4/8) + 6 = 7$$

$\text{CEIL}(4/8)$ is $\text{CEIL}(.5)$, which rounds up to one byte. In this case, there are three objects in each row and one scalar.

Therefore, the storage overhead and performance of manipulating a column object is similar to that of the equivalent set of scalar columns. The storage for collection attributes are described in the ["Viewing Object Data in Relational Form with Unnesting Queries"](#) section on page 5-12.

See Also: *Oracle8i SQL Reference* for more information about `CEIL`.

Row Object Storage in Object Tables

Row objects are stored in *object tables*. An object table is a special kind of table that holds objects and provides a relational view of the attributes of those objects. An object table is logically and physically similar to a relational table whose column types correspond to the top level attributes of the object type stored in the object table. The key difference is that an object table can optionally contain an additional *object identifier* (OID) column and index.

Object Identifier (OID) Storage and OID Index By default, Oracle assigns every row object a unique, immutable object identifier, called an OID. An OID allows the corresponding row object to be referred to from other objects or from relational tables. A built-in datatype called a `REF` represents such references. A `REF` encapsulates a reference to a row object of a specified object type.

By default, an object table contains a system-generated OID column, so that each row object is assigned a globally unique OID. This OID column is automatically indexed for efficient OID-based lookups. The OID column is the equivalent of having an extra 16-byte primary key column.

Primary-Key Based OIDs If a primary key column is available, you can avoid the storage and performance overhead of maintaining the 16-byte OID column and its index. Instead of using the system-generated OIDs, you can use a `CREATE TABLE` statement to specify that the system use the primary key column(s) as the OIDs of the objects in the table. Therefore, you can use existing columns as the OIDs of the objects or use application generated OIDs that are smaller than the 16-byte globally unique OIDs generated by Oracle.

Performance of Object Comparisons

You can compare objects by invoking the *map* or *order* methods defined on the object type. A map method converts objects into scalar values while preserving the ordering of the objects. Mapping objects into scalar values, if it can be done, is preferred because it allows the system to efficiently order objects once they are mapped.

The way objects are mapped has significant performance implications when sorting is required on the objects for `ORDER BY` or `GROUP BY` processing because an object

may need to be compared to other objects many times, and it is much more efficient if the objects can be mapped to scalar values first. If the comparison semantics are extremely complex, or if the objects cannot be mapped into scalar values for comparison, you can define an order method that, given two objects, returns the ordering determined by the object implementor. Order methods are not as efficient as map methods, so performance may suffer if you use order methods. In any one object type, you can implement either map or order methods, but not both.

Once again, consider an object type `ADDRESS` consisting of four character attributes: `STREET`, `CITY`, `STATE`, and `ZIPCODE`. Here, the most efficient comparison method is a map method because each object can be converted easily into scalar values. For example, you might define a map method that orders all of the objects by state.

On the other hand, suppose you want to compare binary objects, such as images. In this case, the comparison semantics may be too complex to use a map method; if so, you can use an order method to perform comparisons. For example, you could create an order method that compares images according to brightness or the number of pixels in each image.

If an object type does not have either a map or order method, only equality comparisons are allowed on objects of that type. In this case, Oracle performs the comparison by doing a field-by-field comparison of the corresponding object attributes, in the order they are defined. If the comparison fails at any point, a `FALSE` value is returned. If the comparison matches at every point, a `TRUE` value is returned. However, if an object has a collection of LOB attributes, then Oracle does not compare the object on a field-by-field basis. Such objects must have a map or order method to perform comparisons.

Storage Considerations for Object Identifiers (OIDs)

`REFs` use object identifiers (OIDs) to point to objects. You can use either system-generated OIDs or primary-key based OIDs. The differences between these types of OIDs are outlined in "[Row Object Storage in Object Tables](#)" on page 5-7. If you use system-generated OIDs for an object table, Oracle maintains an index on the column that stores these OIDs. The index requires storage space, and each row object has a system-generated OID, which requires an extra 16 bytes of storage per row.

You can avoid these added storage requirements by using the primary key for the object identifiers, instead of system-generated OIDs. You can enforce referential integrity on columns that store references to these row objects in a way similar to foreign keys in relational tables.

However, if each primary key value requires more than 16 bytes of storage and you have a large number of `REFs`, using the primary key might require more space than system-generated OIDs because each `REF` is the size of the primary key. In addition, each primary-key based OID is locally (but not necessarily globally) unique. If you require a globally unique identifier, you must ensure that the primary key is globally unique or use system-generated OIDs.

Storage Size of REFS

A `REF` contains the following three logical components:

- OID of the object referenced. A system-generated OID is 16 bytes long. The size of a primary-key based OID depends on the size of the primary key column(s).
- OID of the table or view containing the object referenced, which is 16 bytes long.
- Rowid hint, which is 10 bytes long.

Integrity Constraints for REF Columns

Referential integrity constraints on `REF` columns ensure that there is a row object for the `REF`. Referential integrity constraints on `REFs` create the same relationship as specifying a primary key/foreign key relationship on relational data. In general, you should use referential integrity constraints wherever possible because they are the only way to ensure that the row object for the `REF` exists. However, you cannot specify referential integrity constraints on `REFs` that are in nested tables.

Performance and Storage Considerations for Scoped REFS

A *scoped `REF`* is constrained to contain only references to a specified object table. You can specify a *scoped `REF`* when you declare a column type, collection element, or object type attribute to be a `REF`. In general, you should use *scoped `REFs`* whenever possible instead of *unscoped `REFs`* because *scoped `REFs`* are stored more efficiently. *Scoped `REFs`* are stored on disk as just the OID, so each *scoped `REF`* is 16 bytes long. In addition to the smaller size, the optimizer often can optimize queries that dereference a *scoped `REF`* into efficient joins. This optimization is not possible for *unscoped `REFs`* because the optimizer cannot determine the containing table(s) for *unscoped `REFs`* at query optimization time.

However, unlike referential integrity constraints, *scoped `REFs`* do not ensure that the referenced row object exists; they only ensure that the referenced object table

exists. Therefore, if you specify a scoped REF to a row object and then delete the row object, the scoped REF becomes a dangling REF because the referenced object no longer exists.

Note: Referential integrity constraints are scoped implicitly.

Unscoped REFS are useful if the application design requires that the objects referenced be scattered in multiple tables. Because rowid hints are ignored for scoped REFS, you should use unscoped REFS if the performance gain of the rowid hint, as explained below in the "[Speeding up Object Access using the WITH ROWID Option](#)" section, outweighs the benefits of the storage saving and query optimization of using scoped REFS.

Indexing Scoped REFs

You can build indexes on scoped REF columns using the CREATE INDEX command. Then, you can use the index to efficiently evaluate queries that dereference the scoped REFS. Such queries are turned into joins implicitly. For certain types of queries, Oracle can use an index on the scoped REF column to evaluate the join efficiently.

For example, suppose the object type `address_objtyp` is used to create an object table named `address_objtab`:

```
CREATE TABLE address_objtab OF address_objtyp ;
```

Then, a `people_reltab2` table can be created that has the same definition as the `people_reltab` table discussed in "[Column Object Storage](#)" on page 5-2, except that a REF is used for the address:

```
CREATE TABLE people_reltab2 (  
    id          NUMBER(4)    CONSTRAINT pk_people_reltab2 PRIMARY KEY,  
    name_obj    name_objtyp,  
    address_ref REF address_objtyp SCOPE IS address_objtab,  
    phones_ntab phones_ntabtyp)  
    NESTED TABLE phones_ntab STORE AS phone_store_ntab2 ;
```

Now, an index can be created on the `address_ref` column:

```
CREATE INDEX address_ref_idx ON people_reltab2 (address_ref) ;
```

The following query dereferences the `address_ref`:


```
SELECT id FROM people_reltab2 p
WHERE p.address_ref.state = 'CA' ;
```

When this query is executed, the `address_ref_idx` index is used to efficiently evaluate it. Here, `address_ref` is a scoped REF column that stores references to addresses stored in the `address_objtab` object table. Oracle implicitly transforms the above query into a query with a join:

```
SELECT p.id FROM people_reltab2 p, address_objtab a
WHERE p.address_ref = ref(a) AND a.state = 'CA' ;
```

Oracle's optimizer might create a plan to perform a nested-loops join with `address_objtab` as the outer table and look up matching addresses using the index on the `address_ref` scoped REF column.

Speeding up Object Access using the WITH ROWID Option

If the `WITH ROWID` option is specified for a REF column, Oracle maintains the rowid of the object referenced in the REF. Then, Oracle can find the object referenced directly using the rowid contained in the REF, without the need to fetch the rowid from the OID index. Therefore, you use the `WITH ROWID` option to specify a rowid hint. Maintaining the rowid requires more storage space because the rowid adds 16 bytes to the storage requirements of the REF.

Bypassing the OID index search improves the performance of REF traversal (navigational access) in applications. The actual performance gain may vary from application to application depending on the following factors:

- How large the OID indexes are.
- Whether the OID indexes are cached in the buffer cache.
- How many REF traversals an application does.

The `WITH ROWID` option is only a hint because, when you use this option, Oracle checks the OID of the row object with the OID in the REF. If the two OIDs do not match, Oracle uses the OID index instead. The rowid hint is not supported for scoped REFs, for REFs with referential integrity constraints, or for primary key-based REFs.

Viewing Object Data in Relational Form with Unnesting Queries

An unnesting query on a collection allows the data to be viewed in a flat (relational) form. You can execute unnesting queries on both nested tables and varrays. This section contains examples of unnesting queries.

Nested tables can be unnested for queries using the `TABLE` syntax, as in the following example:

```
SELECT p.name_obj, n.num
      FROM people_reltab p, TABLE(p.phones_ntab) n ;
```

Here, `phones_ntab` specifies the attributes of the `phones_ntab` nested table. To ensure that the parent rows with no children rows also are retrieved, use the outer join syntax as follows:

```
SELECT p.name_obj, n.num
      FROM people_reltab p, TABLE(p.phones_ntab) (+) n ;
```

In the first case, if the query does not refer to any columns from the parent table (other than the nested table column in the `FROM` clause), the query is optimized to execute only against the storage table.

You can also use the `TABLE` syntax to query varrays. For example, suppose the `phones_ntab` nested table is instead a varray named `phones_var`. In this case, you still can use the `TABLE` syntax to query the varray, as in the following example:

```
SELECT p.name_obj, n.num
      FROM people_reltab p, TABLE(p.phones_var) n ;
```

The unnesting query syntax is the same for varrays and nested tables.

Using Procedures and Functions in Unnesting Queries

You can create procedures and functions that you can then execute to perform unnesting queries. For example, you can create a function called `home_phones()` that returns only the phone numbers where `location` is 'home'. To create the `home_phones()` function, you enter code similar to the following:

```
CREATE OR REPLACE FUNCTION home_phones(allphones IN phone_ntabtyp)
      RETURN phone_ntabtyp IS
    homephones phone_ntabtyp := phone_ntabtyp();
    indx1      number;
    indx2      number := 0;
BEGIN
    FOR indx1 IN 1..allphones.count LOOP
```

```
IF
    allphones(indx1).location = 'home'
THEN
    homephones.extend;    -- extend the local collection
    indx2 := indx2 + 1;   -- extend the local collection
    homephones(indx2) := allphones(indx1);
END IF;
END LOOP;

RETURN homephones;
END;
/
```

Now, to query for a list of people and their home phone numbers, enter the following:

```
SELECT p.name_obj, n.num
FROM   people_reltab p, table(
        CAST(home_phones(p.phones_ntab) AS phone_ntabtyp)) n ;
```

To query for a list of people and their home phone numbers, including those people who do not have a home phone number listed, enter the following:

```
SELECT p.name_obj, n.num
FROM   people_reltab p,
        TABLE(CAST(home_phones(p.phones_ntab) AS phone_ntabtyp))(+) n ;
```

See Also: *Oracle8i SQL Reference* for more information about using the TABLE syntax.

Storage Considerations for Varrays

The size of a stored varray depends only on the current count of the number of elements in the varray and not on the maximum number of elements that it can hold. The storage of varrays incurs some overhead, such as null information. Therefore, the size of the varray stored may be slightly greater than the size of the elements multiplied by the count.

Varrays are stored in columns either as raw values or BLOBs. Oracle decides how to store the varray when the varray is defined, based on the maximum possible size of the varray computed using the LIMIT of the declared varray. If the size exceeds approximately 4000 bytes, then the varray is stored in BLOBs. Otherwise, the varray is stored in the column itself as a raw value. In addition, Oracle supports inline

LOBs; therefore, elements that fit in the first 4000 bytes of a large varray (with some bytes reserved for the LOB locator) are stored in the column of the row itself.

Performance of Varrays vs. Nested Tables

If the entire collection is manipulated as a single unit in the application, varrays perform much better than nested tables. The varray is stored "packed" and requires no joins to retrieve the data, unlike nested tables.

Varray Querying

The unnesting syntax can be used to access varray columns similar to the way it is used to access nested tables.

See Also: ["Viewing Object Data in Relational Form with Unnesting Queries"](#) on page 5-12 for more information.

Varray Updates

Piece-wise updates of a varray value are not supported. Thus, when a varray is updated, the entire old collection is replaced by the new collection.

Nested Tables

The following sections contain design considerations for using nested tables.

Nested Table Storage

Oracle stores the rows of a nested table in a separate storage table. A system generated `NESTED_TABLE_ID`, which is 16 bytes in length, correlates the parent row with the rows in its corresponding storage table.

[Figure 5-5](#) shows how the storage table works. The storage table contains each value for each nested table in a nested table column. Each value occupies one row in the storage table. The storage table uses the `NESTED_TABLE_ID` to track the nested table for each value. So, in [Figure 5-5](#), all of the values that belong to nested table A are identified, all of the values that belong to nested table B are identified, etc.

Figure 5–5 Nested Table Storage

DATA1	DATA2	DATA3	DATA4	NT_DATA
...	A
...	B
...	C
...	D
...	E

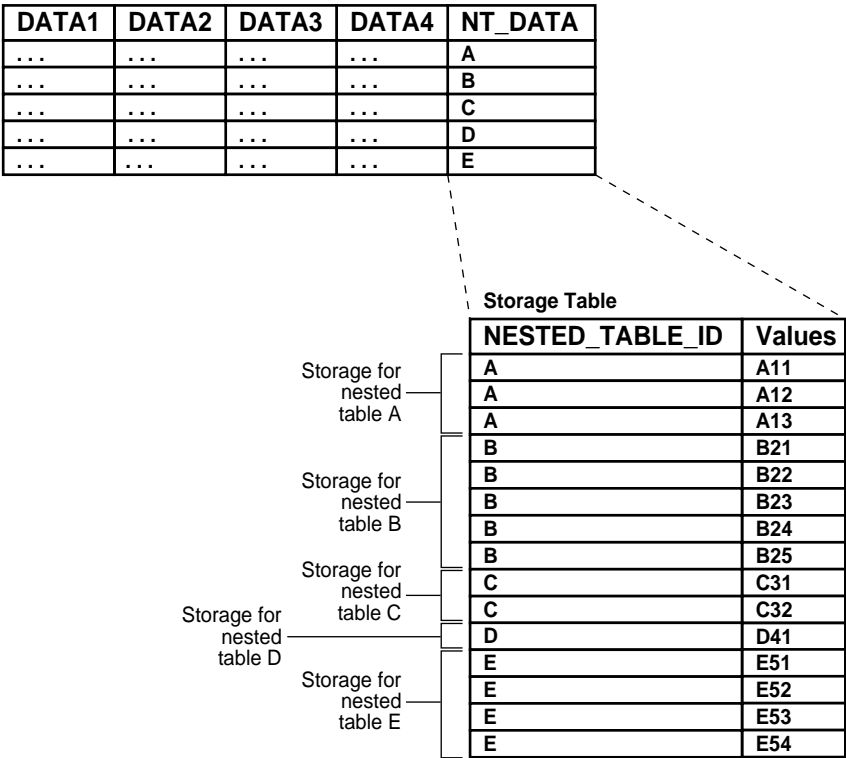
Storage Table	
NESTED_TABLE_ID	Values
B	B21
B	B22
C	C33
A	A11
E	E51
B	B25
E	E52
A	A12
E	E54
B	B23
C	C32
A	A13
D	D41
B	B24
E	E53

Nested Table in an Index-Organized Table (IOT)

If a nested table has a primary key, you can organize the nested table as an index-organized table (IOT). If the `NESTED_TABLE_ID` column is a prefix of the primary key for a given parent row, Oracle physically clusters its children rows together. So, when a parent row is accessed, all its children rows can be efficiently retrieved. When only parent rows are accessed, efficiency is maintained because the children rows are not inter-mixed with the parent rows.

Figure 5–6 shows how the storage table works when the nested table is in an IOT. The storage table groups the values for each nested table within a nested table column. In Figure 5–6, for each nested table in the `NT_DATA` column of the parent table, the data is grouped in the storage table. So, all of the values in nested table A are grouped together, all of the values in nested table B are grouped together, etc.

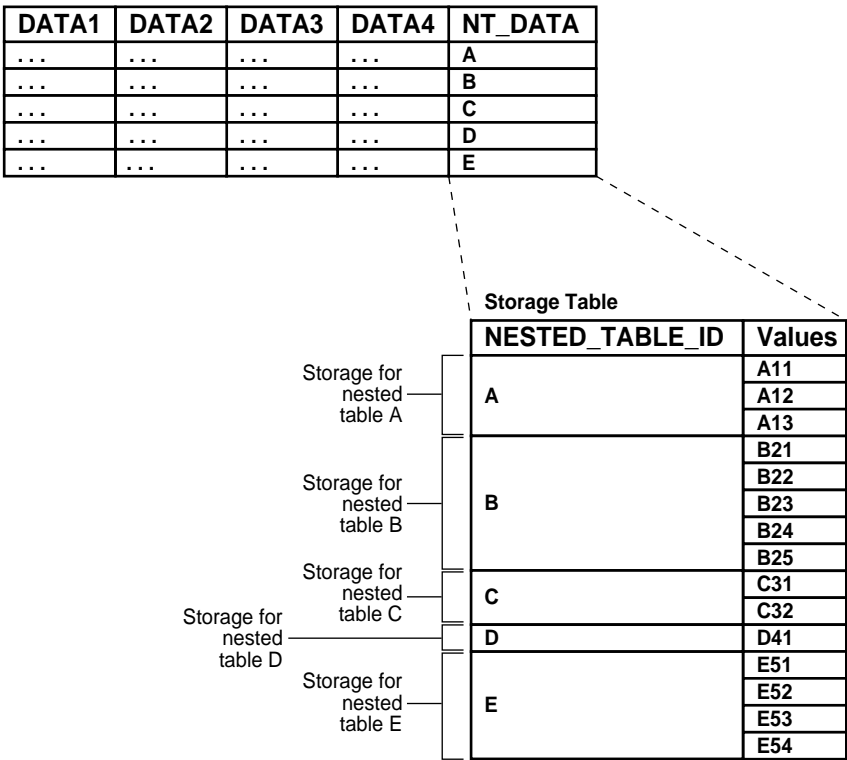
Figure 5–6 Nested Table in IOT Storage



In addition, the COMPRESS clause enables prefix compression on the IOT rows. It factors out the key of the parent in every child row. That is, the parent key is not repeated in every child row, thus providing significant storage savings.

In other words, if you specify nested table compression using the COMPRESS clause, the amount of space required for the storage table is reduced because the NESTED_TABLE_ID is not repeated for each value in a group. Instead, the NESTED_TABLE_ID is stored only once per group, as illustrated in [Figure 5–7](#).

Figure 5–7 Nested Table in IOT Storage with Compression



In general, Oracle Corporation recommends that nested tables be stored in an IOT with the `NESTED_TABLE_ID` column as a prefix of the primary key. Further, prefix compression should be enabled on the IOT. However, if you usually do not retrieve the nested table as a unit and you do not want to cluster the child rows, do not store the nested table in an IOT and do not specify compression.

Nested Table Indexes

For nested tables stored in heap tables (as opposed to IOTs), you should create an index on the `NESTED_TABLE_ID` column of the storage table. The index on the corresponding ID column of the parent table is created by Oracle automatically when the table is created. Creating an index on the `NESTED_TABLE_ID` column enables Oracle to access the child rows of the nested table more efficiently, because

Oracle must perform a join between the parent table and the nested table using the `NESTED_TABLE_ID` column.

Nested Table Locators

For large child-sets, the parent row and a locator to the child-set can be returned so that the children rows can be accessed on demand; the child-sets also can be filtered. Using nested table locators allows you to avoid unnecessary transporting of children rows for every parent.

You can perform either one of the following actions to access the children rows using the nested table locator:

- Call the OCI collection functions. This action occurs implicitly when you access the elements of the collection in the client-side code, such as *OCIColl** functions. The entire collection is retrieved implicitly on the first access.

See Also: *Oracle Call Interface Programmer's Guide* for more information about OCI collection functions.

- Use SQL to retrieve the rows corresponding to the nested table. This action is described in "[The Object Table PurchaseOrder_objtab](#)" on page 8-27.

Optimizing Set Membership Queries

Set membership queries are useful when you want to search for a specific item in a nested table. For example, the following query tests the membership in a child-set; specifically, whether the location `home` is in the nested table `phones_ntab`, which is in the parent table `people_reltab`:

```
SELECT * FROM people_reltab p
WHERE 'home' IN (SELECT location FROM TABLE(p.phones_ntab)) ;
```

Oracle can execute a query that tests the membership in a child-set more efficiently by transforming it internally into a semi-join. However, this optimization only happens if the `ALWAYS_SEMI_JOIN` initialization parameter is set. If you want to perform semi-joins, the valid values for this parameter are `MERGE` and `HASH`; these parameter values indicate which join method to use.

Note: In the example above, `home` and `location` are child set elements. If the child set elements are object types, they must have a `map` or `order` method to perform a set membership query.

DML Operations on Nested Tables

You can perform DML operations on nested tables. Rows can be inserted into or deleted from a nested table, and existing rows can be updated, by using the appropriate SQL command against the nested table. In these operations, the nested table is identified by a `TABLE` subquery. The following example inserts a new person into the `people_reltab` table, including phone numbers into the `phones_ntab` nested table:

```
INSERT INTO people_reltab values (
    0001,
    name_objtyp(
        'john', 'william', 'foster'),
    address_objtyp(
        '111 Maple Road', 'Fairfax', 'VA', '22033'),
    phone_ntabtyp(
        phone_objtyp('home', '650.331.1222'),
        phone_objtyp('work', '650.945.4389')) ;
```

The following example inserts a phone number into the nested table `phones_ntab` for an existing person in the `people_reltab` table whose identification number is 0001:

```
INSERT INTO TABLE(SELECT p.phones_ntab FROM people_reltab p WHERE p.id = '0001')
VALUES ('cell', '650.331.9337') ;
```

To drop a particular nested table, you can set the nested table column in the parent row to `NULL`, as in the following example:

```
UPDATE people_reltab SET phones_ntab = NULL WHERE id = '0001' ;
```

Once you drop a nested table, you cannot insert values into it until you recreate it. To recreate the nested table in the `phones_ntab` nested table column object for the person whose identification number is 0001, enter the following SQL statement:

```
UPDATE people_reltab SET phones_ntab = phone_ntabtyp() WHERE id = '0001' ;
```

You also can insert values into the nested table as you recreate it:

```
UPDATE people_reltab
```

```
SET phones_ntab = phone_ntabtyp(phone_objtyp('home', '650.331.1222'))
WHERE id = '0001' ;
```

DML operations on a nested table lock the parent row. Therefore, only one modification at a time can be made to the data in a particular nested table, even if the modifications are on different rows in the nested table. However, if only part of the data in your nested table must support simultaneous modifications, while other data in the nested table does not require this support, you should consider using `REFs` to the data that requires simultaneous modifications.

For example, if you have an application that processes purchase orders, you might include customer information and line items in the purchase orders. In this case, the customer information does not change often and so you do not need to support simultaneous modifications for this data. Line items, on the other hand, might change very often. To support simultaneous updates on line items that are in the same purchase order, you can store the line items in a separate object table and reference them with `REFs` in the nested table.

Nesting Collections within other Collections

An attribute of a collection cannot be a collection type (either varray or nested table). In other words, you cannot have collections within collections. Oracle allows only one level of direct nesting of collections. However, an attribute of a collection can be a reference to an object that has a collection attribute. Thus, you can have multiple levels of collections indirectly by using `REFs`.

For example, suppose you want to create a new object type called `person_objtyp` using the object types described in ["Column Object Storage"](#) on page 5-2, which are `name_objtyp`, `address_objtyp`, and `phone_ntabtyp`. Remember that the `phone_ntabtyp` object type is a nested table because each person may have more than one phone number.

To create the `person_objtyp` object type, issue the following SQL statement:

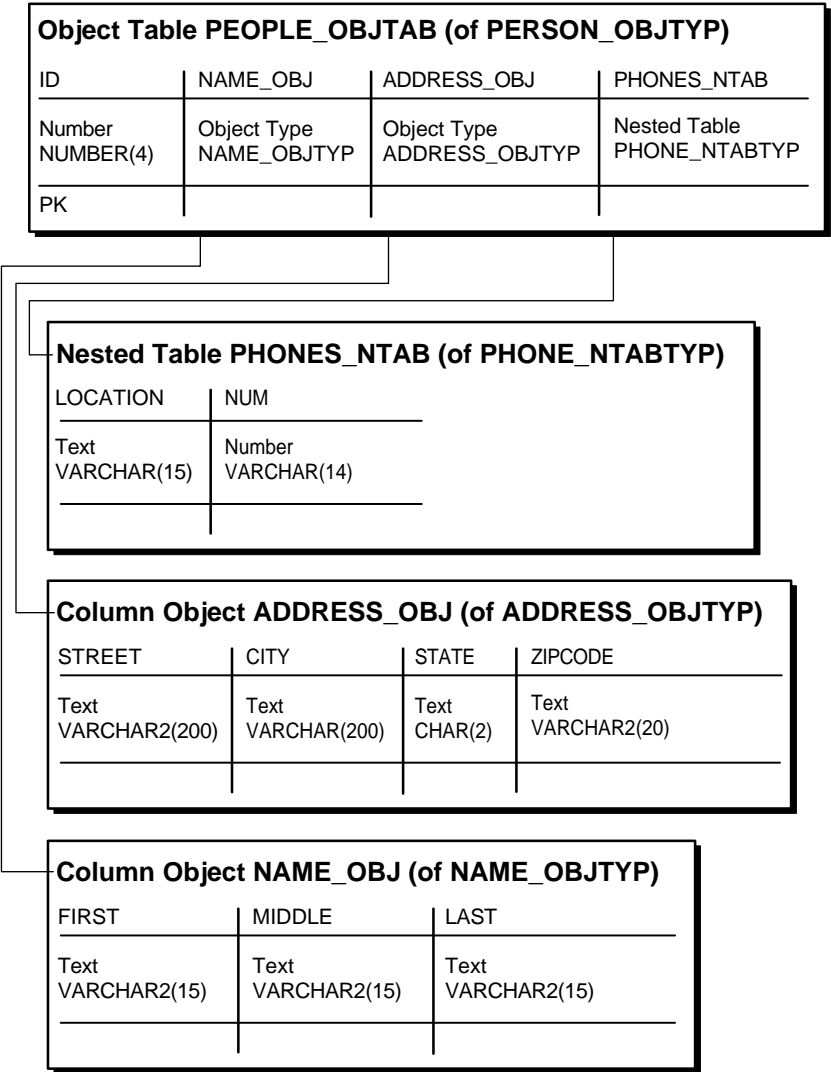
```
CREATE TYPE person_objtyp AS OBJECT (
    id          NUMBER(4),
    name_obj    name_objtyp,
    address_obj address_objtyp,
    phones_ntab phone_ntabtyp);
```

To create an object table called `people_objtab` of `person_objtyp` object type, issue the following SQL statement:

```
CREATE TABLE people_objtab OF person_objtyp (id PRIMARY KEY)
  NESTED TABLE phones_ntab STORE AS phones_store_ntab ;
```

The `people_objtab` table has the same attributes as the `people_reltab` table discussed in "[Column Object Storage](#)" on page 5-2. The difference is that the `people_objtab` is an object table with row objects, while the `people_reltab` table is a relational table with three column objects.

Figure 5–8 Object Relational Representation of the people_objtab Object Table



Now you can reference the row objects in the `people_objtab` object table from other tables. For example, suppose you want to create a `projects_objtab` table that contains:

- A project identification number for each project.
- The title of each project.
- The project lead for each project.
- A description of each project.
- Nested table collection of the team of people assigned to each project.

You can use `REFs` to the `people_objtab` for the project leads, and you can use a nested table collection of `REFs` for the team. To begin, create a nested table object type called `personref_ntabtyp` based on the `person_objtyp` object type:

```
CREATE TYPE personref_ntabtyp AS TABLE OF REF person_objtyp;
```

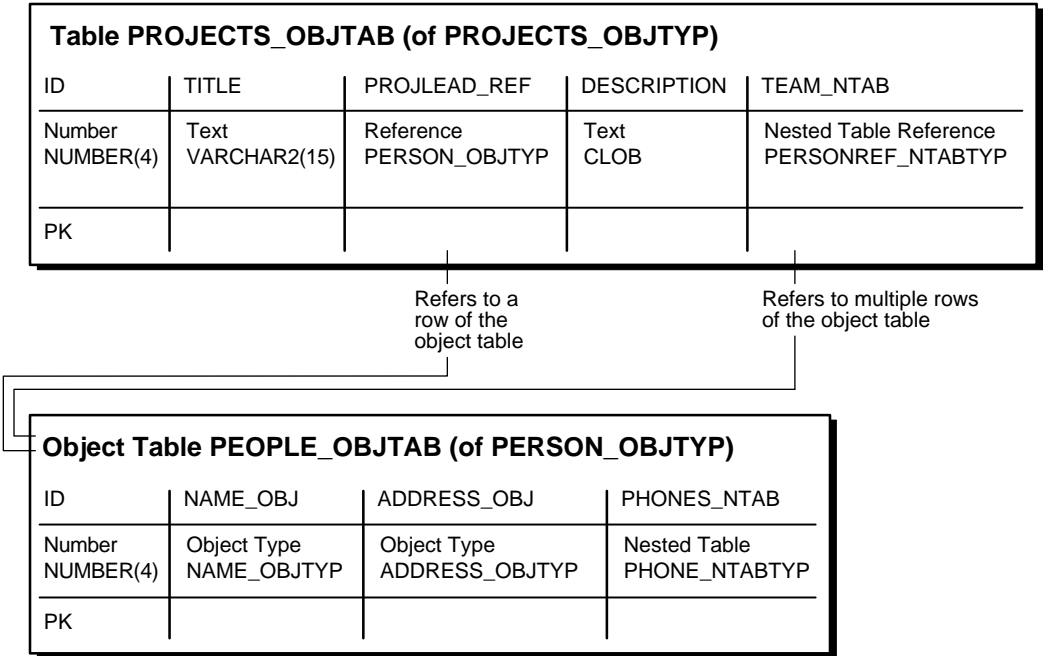
Now you are ready to create the object table `projects_objtab`. First, create the object type `projects_objtyp` by issuing the following SQL statement:

```
CREATE TYPE projects_objtyp AS OBJECT (  
    id            NUMBER(4),  
    title         VARCHAR2(15),  
    proglead_ref  REF person_objtyp,  
    description   CLOB,  
    team_ntab     personref_ntabtyp);
```

Next, create the object table `projects_objtab` based on the `projects_objtyp`:

```
CREATE TABLE projects_objtab OF projects_objtyp (id PRIMARY KEY)  
    NESTED TABLE team_ntab STORE AS team_store_ntab ;
```

Figure 5–9 Object Relational Representation of the projects_objtab Object Table



Once the people_objtab object table and the projects_objtab object table are in place, you indirectly have a nested collection. That is, the projects_objtab table contains a nested table collection of REFs that point to the people in the people_objtab table, and the people in the people_objtab table have a nested table collection of phone numbers.

You can insert values into the people_objtab table in the following way:

```
INSERT INTO people_objtab VALUES (
    0001,
    name_objtyp('JOHN', 'JACOB', 'SCHMIDT'),
    address_objtyp('1252 Maple Road', 'Fairfax', 'VA', '22033'),
    phone_ntabtyp(
        phone_objtyp('home', '650.339.9922'),
        phone_objtyp('work', '510.563.8792')));
```

```

INSERT INTO people_objtab VALUES (
    0002,
    name_objtyp('MARY', 'ELLEN', 'MILLER'),
    address_objtyp('33 Spruce Street', 'McKees Rocks', 'PA', '15136'),
    phone_ntabtyp(
        phone_objtyp('home', '415.642.6722'),
        phone_objtyp('work', '650.891.7766')));

```

```

INSERT INTO people_objtab VALUES (
    0003,
    name_objtyp('SARAH', 'MARIE', 'SINGER'),
    address_objtyp('525 Pine Avenue', 'San Mateo', 'CA', '94403'),
    phone_ntabtyp(
        phone_objtyp('home', '510.804.4378'),
        phone_objtyp('work', '650.345.9232'),
        phone_objtyp('cell', '650.854.9233')));

```

Then, you can insert into the projects_objtab relational table by selecting from the people_objtab object table using a REF operator, as in the following examples:

```

INSERT INTO projects_objtab VALUES (
    1101,
    'Demo Product',
    (SELECT REF(p) FROM people_objtab p WHERE id = 0001),
    'Demo the product, show all the great features.',
    personref_ntabtyp(
        (SELECT REF(p) FROM people_objtab p WHERE id = 0001),
        (SELECT REF(p) FROM people_objtab p WHERE id = 0002),
        (SELECT REF(p) FROM people_objtab p WHERE id = 0003)));

```

```

INSERT INTO projects_objtab VALUES (
    1102,
    'Create PRODDB',
    (SELECT REF(p) FROM people_objtab p WHERE id = 0002),
    'Create a database of our products.',
    personref_ntabtyp(
        (SELECT REF(p) FROM people_objtab p WHERE id = 0002),
        (SELECT REF(p) FROM people_objtab p WHERE id = 0003)));

```

Note: This example uses nested tables to store `REFs`, but you also can store `REFs` in `varrays`. That is, you can have a `varray` of `REFs`.

Choosing a Language for Method Functions

Method functions can be implemented in any of the languages supported by Oracle, such as PL/SQL, Java, or C. Consider the following factors when you choose the language for a particular application:

- Ease of use
- SQL calls
- Speed of execution
- Same/different address space

In general, if the application performs intense computations, C is preferable, but if the application performs a relatively large number of database calls, PL/SQL or Java is preferable.

A method implemented in C executes in a separate process from the server using external routines. In contrast, a method implemented in Java or PL/SQL executes in the same process as the server.

Method Implementation Example

The example described in this section involves an object type whose methods are implemented in different languages. In the example, the object type `ImageType` has an `ID` attribute, which is a `NUMBER` that uniquely identifies it, and an `IMG` attribute, which is a `BLOB` that stores the raw image. The object type `ImageType` has the following methods:

- The method `get_name()` fetches the name of the image by looking it up in the database. This method is implemented in PL/SQL.
- The method `rotate()` rotates the image. This method is implemented in C.
- The method `clear()` returns a new image of the specified color. This method is implemented in Java.

For implementing a method in C, a `LIBRARY` object must be defined to point to the library that contains the external C routines. For implementing a method implemented in Java, this example assumes that the Java class with the method has been compiled and uploaded into Oracle.

Here is the object type specification and its methods:

```
CREATE TYPE ImageType AS OBJECT (
    id    NUMBER,
    img   BLOB,
    MEMBER FUNCTION get_name() return VARCHAR2,
    MEMBER FUNCTION rotate() return BLOB,
    STATIC FUNCTION clear(color NUMBER) return BLOB
);

CREATE TYPE BODY ImageType AS
    MEMBER FUNCTION get_name() RETURN VARCHAR2
    AS
    imgname VARCHAR2(100);
    BEGIN
        SELECT name INTO imgname FROM imgtab WHERE imgid = id;
        RETURN imgname;
    END;

    MEMBER FUNCTION rotate() RETURN BLOB
    AS LANGUAGE C
    NAME "Crotate"
    LIBRARY myCfuncs;

    STATIC FUNCTION clear(color NUMBER) RETURN BLOB
    AS LANGUAGE JAVA
    NAME 'myJavaClass.clear(color oracle.sql.NUMBER) RETURN oracle.sql.BLOB';

END;
/
```

Restriction: Type methods can be mapped only to static Java methods.

See Also:

- *Oracle8i Java Stored Procedures Developer's Guide* for more information.
 - [Chapter 3, "Object Support in Oracle Programmatic Environments"](#) for more information about choosing a language.
-
-

Static Methods

Static methods differ from member methods in that the `SELF` value is not passed in as the first parameter. Methods in which the value of `SELF` is not relevant should be implemented as static methods. Static methods can be used for user-defined constructors.

The following example is a constructor-like method that constructs an instance of the type based on the explicit input parameters and inserts the instance into the specified table:

```
CREATE OR REPLACE TYPE atype AS OBJECT(a1 NUMBER,
    STATIC PROCEDURE newa (
        p1          NUMBER,
        tabname     VARCHAR2,
        schname     VARCHAR2));

CREATE OR REPLACE TYPE BODY atype AS
    STATIC PROCEDURE newa (p1 NUMBER, tabname VARCHAR2, schname VARCHAR2)
    IS
        sqlstmt VARCHAR2(100);
    BEGIN
        sqlstmt := 'INSERT INTO ' || schname || '.' || tabname || ' VALUES (atype(:1))';
        EXECUTE IMMEDIATE sqlstmt USING p1;
    END;
END;
/

CREATE TABLE atab OF atype;
BEGIN
    atype.newa(1, 'atab', 'scott');
END;
```

Writing Reusable Code using Invoker Rights

To create generic object types that can be used in any schema, you must define the type to use invoker-rights, through the `AUTHID CURRENT_USER` option of `CREATE OR REPLACE TYPE`. In general, use invoker-rights when both of the following conditions are true:

- There are type methods that access and manipulate data.
- Users who did not define these type methods must use them.

For example, you can grant user SARA execute privileges on type `atype` created by SCOTT in ["Static Methods"](#) on page 5-28, and then create table `atab` based on the type:

```
GRANT EXECUTE ON atype TO SARA ;
CONNECT SARA/TPK101 ;
CREATE TABLE atab OF scott.atype ;
```

Now, suppose user SARA tries to use `atype` in the following statement:

```
BEGIN
    scott.atype.new(1, 'atab', 'SARA'); -- raises an error
END;
/
```

This statement raises an error because the definer of the type (SCOTT) does not have the privileges required to perform the insert in the `new` procedure. You can avoid this error by defining `atype` using invoker-rights. Here, you first drop the `atab` table in both schemas and recreate `atype` using invoker-rights:

```
DROP TABLE atab ;
CONNECT SCOTT/TIGER ;
DROP TABLE atab ;

CREATE OR REPLACE TYPE atype AUTHID CURRENT_USER AS OBJECT(a1 NUMBER,
    STATIC PROCEDURE new(p1 NUMBER, tabname VARCHAR2, schname VARCHAR2));
```

```
CREATE OR REPLACE TYPE BODY atype AS
  STATIC PROCEDURE newa(p1 NUMBER, tabname VARCHAR2, schname VARCHAR2)
  IS
    sqlstmt VARCHAR2(100);
  BEGIN
    sqlstmt := 'INSERT INTO '||schname||'.'||tabname||' VALUES
      (scott.atype(:1))';
    EXECUTE IMMEDIATE sqlstmt USING p1;
  END;
END;
/
```

Now, if user SARA tries to use `atype` again, the statement executes successfully:

```
GRANT EXECUTE ON atype TO SARA ;
CONNECT SARA/TPK101 ;
CREATE TABLE atab OF scott.atype;

BEGIN
  scott.atype.newa(1, 'atab', 'SARA'); -- executes successfully
END;
/
```

The statement is successful this time because the procedure is executed under the privileges of the invoker (SARA), not the definer (SCOTT).

Function-Based Indexes on the Return Values of Type Methods

You can create function-based indexes on the return values of type methods. The following example creates a function-based index on the method `afun()` of the type `atype2`:

```
CREATE TYPE atype2 AS OBJECT
(
  a NUMBER,
  MEMBER FUNCTION afun RETURN NUMBER DETERMINISTIC
);

CREATE OR REPLACE TYPE BODY atype2 IS
  MEMBER FUNCTION afun RETURN NUMBER IS
  BEGIN
    RETURN self.a * 100;
  END;
END;
/
```

```
CREATE TABLE atab2 OF atype2 ;  
CREATE INDEX atab2_afun_idx ON atab2 x (x.afun()) ;
```

For some methods, you can use function-based indexes to improve the performance of method invocation in SQL.

Restriction: You cannot create an index on a type method that takes as input LOB, REF, nested table, or varray arguments, or on any object type that contains such attributes.

See Also: *Oracle8i SQL Reference* for detailed information about using function-based indexes.

New Object Format in Release 8.1

In release 8.1, objects are stored in a new format that uses less storage space and has better performance characteristics than the previous format. The performance also is improved due to a more efficient transport protocol. If the `COMPATIBLE` parameter is set to 8.1.0 or higher, all the new objects you create are automatically stored and transported in the release 8.1 format.

In order to convert the objects created in a release 8.0 database to the release 8.1 format, complete following steps:

1. Recreate the tables using a `CREATE TABLE...AS SELECT...` statement.
2. Export/import the data in the tables.

See Also: *Oracle8i Migration* for more information about compatibility and the `COMPATIBLE` initialization parameter.

Replicating Object Tables and Columns

Replication of object columns and object tables is not yet supported. If replication is a requirement, then you can use object views and store the application objects in relational tables, which can be replicated. Using object views, both the object model and the data to be replicated can be preserved in the database.

Consequences of the Oracle Inheritance Implementation

Inheritance can imply various levels of encapsulation for super-types. In cases where the super-type should not be exposed to other objects, a subtype should contain the methods and attributes necessary to make the super-type invisible. To understand the implementation consequences of the inheritance, it is also important to remember that Oracle8i is a strongly-typed system. A strongly-typed system requires that the type of an attribute is declared when the attribute is declared. Only values of the declared type may be stored in the attribute. For example, the Oracle8i collections are strongly-typed. Oracle8i does not allow the implementation of heterogeneous collections (collections of multiple types).

Simulating Inheritance

The Oracle type model does not support inheritance directly. However, you can map your current Oracle object types to Java classes and then leverage the inheritance features native to Java.

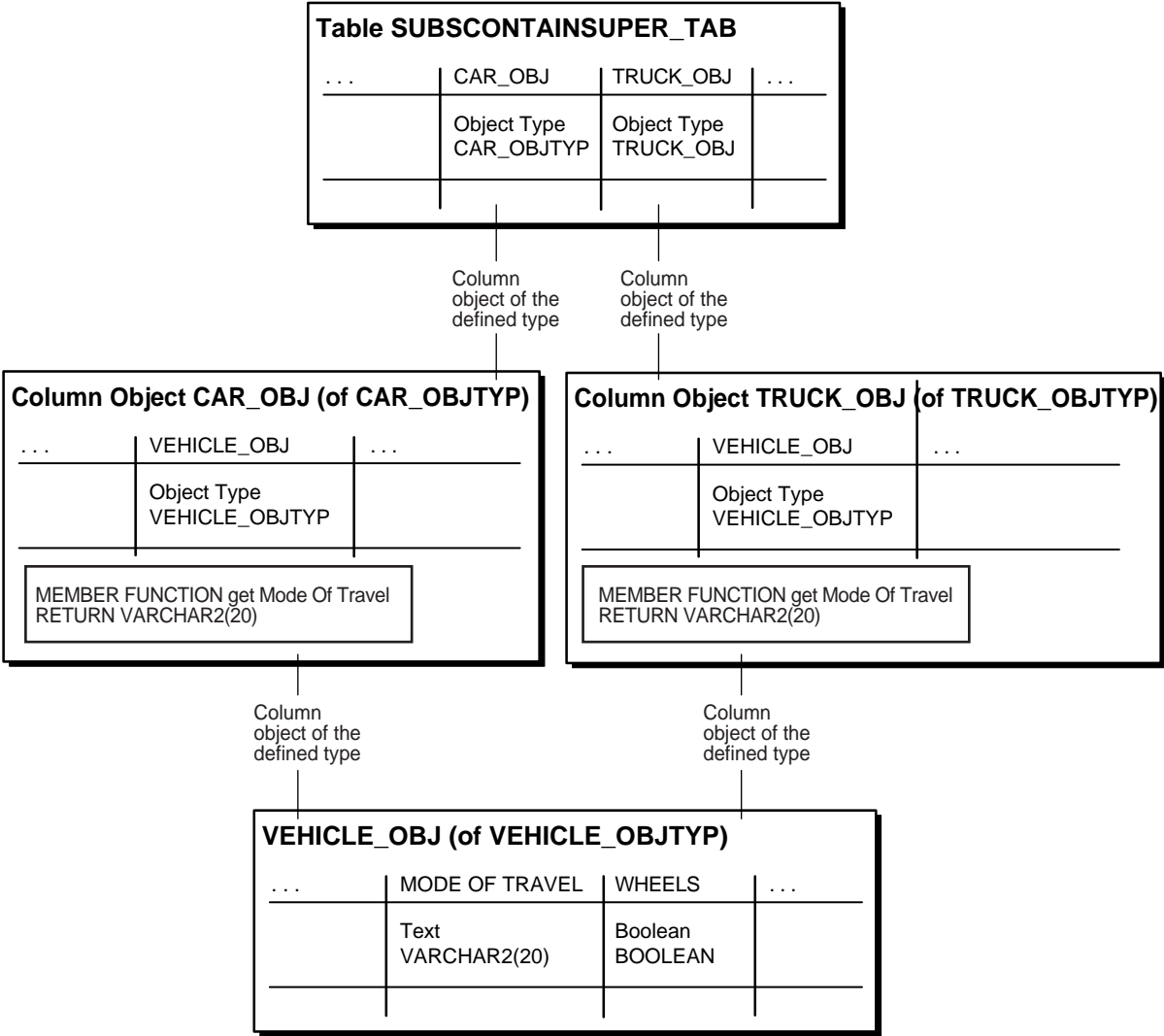
See Also: *Oracle8i JDBC Developer's Guide and Reference* and *Oracle8i SQLJ Developer's Guide and Reference* for more information about mapping Oracle objects to Java classes.

In addition, inheritance can be simulated in Oracle. For example, you can use one of the following techniques to simulate inheritance:

- Subtype Contains Super-type
- Super-type Contains or References All Subtypes
- Dual Subtype / Super-type Reference

Subtype Contains Super-type

Figure 5–10 Object-Relational Schema — Subtype Contains Super-type



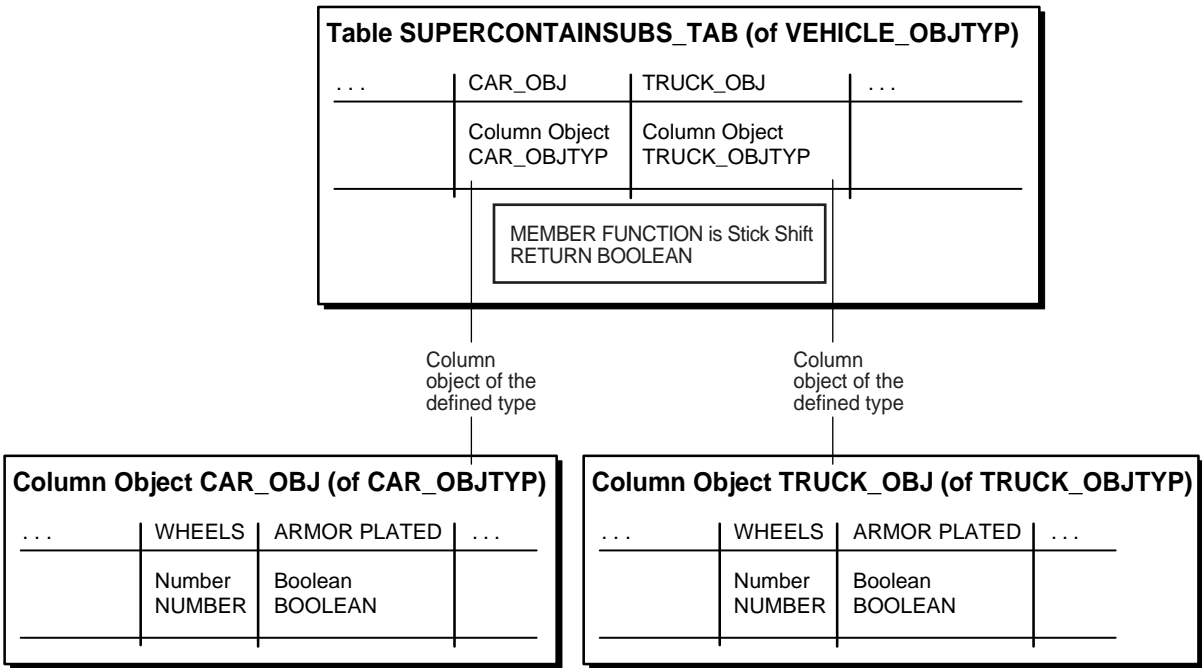
The Subtype Contains Super-type technique hides the implementation of the abstractions/generalizations for a subtype. Each of the subtypes are exposed to other types in the object model. The super-types are not exposed to other types. To simulate inheritance, the super-type in the design object model is created as an object type. The subtype is also created as an object type. The super-type is defined as an embedded attribute in the subtype. All of the methods that can be executed for the subtype and its super-type must be defined in the subtype.

The Subtype Contains Super-type technique is used when each subtype has specific relationships to other objects in the object model. For example, a super-type of Customer may have subtypes of Private Customer and Corporate Customer. Private Customers have relationships with the Personal Banking objects, while Corporate Customers have relationships with the Commercial Banking objects. In this environment, the Customer super-type is not visible to the rest of the object model.

In the Vehicle-Car/Truck example, the Vehicle (super-type) is embedded in the sub-types Car and Truck.

Super-type Contains All Subtypes

Figure 5–11 Object-Relational Schema — Super-type Contains All Subtypes



The *Super-type Contains All Subtypes* technique hides the implementation of the subtypes and only exposes the super-type. To simulate inheritance, all of the subtypes for a given super-type in the design object model are created as object types. The super-type is created as an object type as well. The super-type declares an attribute for each subtype. The super-type also declares the constraints to enforce the one-and-only-one rules for the subtype attributes. All of the methods that can be executed for the subtype must be defined in the super-type.

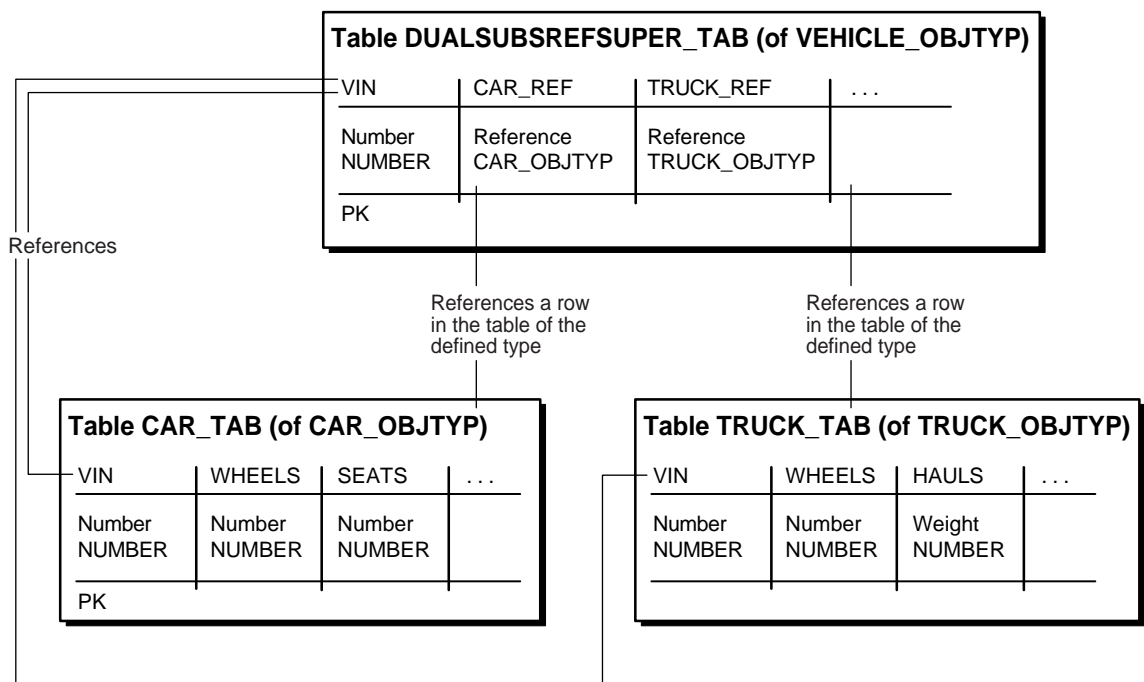
The *Super-type Contains All Subtypes* technique is used when objects have relationships with other objects that are predominately one-to-many in multiplicity. For example, a Customer can have many Accounts and a Bank can have many Accounts. The many relationships require a collection for each subtype if the *Subtype Contains Super-type* technique is used. If the Account is a super-type and Checking and Savings are subtypes, both Bank and Customer must implement a collection of Checking and Savings (4 collections). Adding a new account

subtype requires that both `Customer` and `Bank` add the collection to support the new account subtype (2 collections per addition). Using the *Super-type Contains All Subtypes* technique means that `Customer` and `Bank` have a collection of `Account`. Adding a subtype to `Accounts` means that only account changes.

In the case of the `Vehicle-Car/Truck`, the `Vehicle` is created with `Car` and `Truck` as embedded attributes of `Vehicle`.

Dual Subtype / Super-type Reference

Figure 5–12 Object-Relational Schema — Dual Subtype / Super-type Reference



In cases where the super-type is involved in multiple object-relationships with many for a multiplicity and the subtypes have specific relationships in the object model, the implementation of inheritance is a combination of the two inheritance techniques. The super-type is implemented as an object type. Each subtype is implemented as an object type. The super-type implements a referenced attribute for each subtype (zero referenced relationship). The super-type also implements an

or-association for the group of subtype attributes. Each subtype implements a referenced attribute for the super-type (one referenced relationship). In this way, both the super-type and sub-type are visible to the rest of the object model.

In the case of the Vehicle-Car/Truck, the Vehicle is created as a type. The Car and Truck are created as types. The Vehicle type implements a reference to both Car and Truck, with the or-constraint on the Car and Truck attributes. The Car implements an attribute that references Vehicle. The Truck implements an attribute that references Vehicle.

Constraints on Objects

Oracle does not support constraints and defaults in type specifications. However, you can specify the constraints and defaults when creating the tables:

```
CREATE OR REPLACE TYPE customer_type AS OBJECT(  
    cust_id INTEGER);
```

```
CREATE OR REPLACE TYPE department_type AS OBJECT(  
    deptno INTEGER);
```

```
CREATE TABLE customer_tab OF customer_type (  
    cust_id default 1 NOT NULL);
```

```
CREATE TABLE department_tab OF department_type (  
    deptno PRIMARY KEY);
```

```
CREATE TABLE customer_tab1 (  
    cust customer_type DEFAULT customer_type(1)  
    CHECK (cust.cust_id IS NOT NULL),  
    some_other_column VARCHAR2(32));
```

Type Evolution

You cannot change the definitions of types that have dependent data (in the form of column and/or row objects). However, you can modify tables with column objects by dropping and adding columns in a way similar to regular relational tables.

You cannot change tables containing row objects by dropping, adding, or modifying columns. If you need to modify tables containing row objects, a workaround is to:

1. Copy the table data into a temporary table, or export the table data.
2. Drop the table.
3. Recreate the type with the new definition.
4. Recreate the table.
5. Copy in the relevant data from temporary table, or import the data.

If type evolution is a requirement and this workaround is not acceptable, you should use object views defined over relational tables, instead of column objects or row objects. You can then change the definitions of object types and views.

Performance Tuning

See *Oracle8i Tuning* for details on measuring and tuning the performance of your application. In particular, some of the key performance factors are the following:

- `ANALYZE` command to collect statistics.
- `tkprof` to profile execution of SQL commands.
- `EXPLAIN PLAN` to generate the query plans.

Parallel Queries with Oracle Objects

Oracle8i lets you perform parallel queries with objects, when you follow these rules:

- To make queries involving joins and sorts parallel (using the `ORDER BY`, `GROUP BY`, and `SET` operations), a `MAP` function is required. In the absence of a `MAP` function, the query automatically becomes serial.
- Parallel queries on nested tables are not supported. Even if there are parallel hints or parallel attributes for the table, the query is serial.
- Parallel DML and parallel DDL are not supported with objects. DML and DDL are always performed in serial.

Advanced Topics for Oracle Objects

The other chapters in this book discuss the topics that you need to get started with Oracle objects. The topics in this chapter are of interest once you start applying object-relational techniques to large-scale applications or complex schemas.

If the terms in this chapter are unfamiliar to you, or you are not sure what their significance is, refer to [Chapter 1, "Introduction to Oracle Objects"](#) and [Chapter 7, "Frequently Asked Questions about Programming with Oracle Objects"](#).

Storage of Objects

Oracle automatically maps the complex structure of object types into the simple rectangular structure of tables.

Leaf-Level Attributes

An object type is like a tree structure, where the branches represent the attributes. Attributes that are objects sprout subbranches for their own attributes.

Ultimately, each branch ends at an attribute that is a built-in type (such as NUMBER, VARCHAR2, or REF) or a collection type (such as VARRAY or nested table). Each of these *leaf-level attributes* of the original object type is stored in a table column.

The leaf-level attributes that are not collection types are called the *leaf-level scalar attributes* of the object type.

How Row Objects are Split Across Columns

In an object table, Oracle stores the data for every leaf-level scalar or REF attribute in a separate column. Each VARRAY is also stored in a column, unless it is too large

(see ["Internal Layout of VARRAYs"](#) on page 6-3). Oracle stores leaf-level attributes of table types in separate tables associated with the object table. You must declare these tables as part of the object table declaration (see ["Internal Layout of Nested Tables"](#) on page 6-3).

When you retrieve or change attributes of objects in an object table, Oracle performs the corresponding operations on the columns of the table. Accessing the value of the object itself produces a copy of the object, by invoking the default constructor for the type, using the columns of the object table as arguments.

Oracle stores the system-generated object identifier in a hidden column. Oracle uses the object identifier to construct REFs to the object.

Hidden Columns for Tables with Column Objects

When a table is defined with a column of an object type, Oracle adds hidden columns to the table for the object type's leaf-level attributes. Each column object also has a corresponding hidden column to store the NULL information of the object (that is, the atomic nulls of the top-level and the nested objects).

REFs

When Oracle constructs a REF to a row object, the constructed REF is made up of the object identifier, some metadata of the object table, and, optionally, the ROWID.

The size of a REF in a column of REF type depends on the storage properties associated with the column. For example, if the column is declared as a REF WITH ROWID, Oracle stores the ROWID in the REF column. The ROWID hint is ignored for object references in constrained REF columns.

If column is declared as a REF with a SCOPE clause, the column is made smaller by omitting the object table metadata and the ROWID. A scoped REF is 16 bytes long.

If the object identifier is primary-key based, Oracle may create one or more internal columns to store the values of the primary key depending on how many columns comprise the primary key.

Note: When a REF column references row objects whose object identifiers are derived from primary keys, we refer to it as a *primary-key-based REF* or *pkREF*. Columns containing pkREFs must be scoped or have a referential constraint.

Internal Layout of Nested Tables

The rows of a nested table are stored in a separate storage table. Each nested table column has a single associated storage table, not one for each row. The storage table holds all the elements for all of the nested tables in that column. The storage table has a hidden `NESTED_TABLE_ID` column with a system-generated value that lets Oracle map the nested table elements back to the appropriate row.

You can speed up queries that retrieve entire collections by making the storage table index-organized. Include the `ORGANIZATION INDEX` clause inside the `STORE AS` clause.

A nested table type can contain objects or scalars:

- If the elements are objects, the storage table is like an object table: the top-level attributes of the object type become the columns of the storage table. But because a nested table row has no object identifier column, you cannot construct REFs to objects in a nested table.
- If the elements are scalars, the storage table contains a single column called `COLUMN_VALUE` that contains the scalar values.

For more information, see [Nested Table Storage](#) on page 5-14.

Internal Layout of VARRAYs

All the elements of a VARRAY are stored in a single column. Depending upon the size of the array, it may be stored inline or in a BLOB. See [Storage Considerations for Varrays](#) on page 5-13 for details.

Object Identifiers

Every row object in an object table has an associated logical object identifier (OID). By default, Oracle assigns each row object a unique system-generated OID, 16 bytes in length. Oracle provides no documentation of or access to the internal structure of object identifiers. This structure can change at any time.

The OID column of an object table is a hidden column. Once it is set up, you can ignore it and focus instead on fetching and navigating objects through object references.

The OID for a row object uniquely identifies it in an object table. Oracle implicitly creates and maintains an index on the OID column of an object table. In a distributed and replicated environment, the system-generated unique identifier lets Oracle identify objects unambiguously .

Primary-key Based Object Identifiers In less demanding environments, where globally unique system-generated identifiers are not required, it may be inefficient to store sixteen extra bytes with each object and maintain an index on it. A space-saving technique is to reuse the primary key value of a row object as its object identifier.

Primary-key based identifiers also make it faster and easier to loading data into an object table. By contrast, system-generated object identifiers need to be remapped using some user-specified keys, especially when references to them are also stored persistently.

OCI Tips and Techniques for Objects

The following sections introduce tips and techniques for using OCI effectively by showing common operations performed by an OCI program that uses objects.

Initializing an OCI Program in Object Mode

To enable object manipulation, the OCI program must be initialized in *object mode*. The following OCI code initializes a program in object mode:

```
err = OCIInitialize(OCI_OBJECT, 0, 0, 0, 0);
```

When the program is initialized in object mode, the object cache is initialized. Memory for the cache is not allocated at this time; instead, it is allocated only on demand.

Creating a New Object

The *OCIObjectNew()* function creates transient or persistent objects. A transient object's lifetime is the duration of the session in which it was created. A persistent object is an object that is stored in an object table in the database. The *OCIObjectNew()* function returns a pointer to the object created in the cache, and the application should initialize the new object by setting the attribute values directly. The object is not created in the database yet; it will be created and stored in the database when it is flushed from the cache.

When *OCIObjectNew()* creates an object in the cache, it sets all the attributes to `NULL`. The attribute null indicator information is recorded in the parallel null indicator structure. If the application sets the attribute values, but fails to set the null indicator information in the parallel null structure, then upon object flush the object attributes will be set to `NULL` in the database.

In Oracle8i, if you want to set all of the attributes to NOT NULL during object creation instead, you can use the `OCI_OBJECT_NEW_NOTNULL` attribute of the environment handle using the `OCIAttrSet()` function. When set, this attribute creates a non-null object. That is, all the attributes are set to default values provided by Oracle and their null status information in the parallel null indicator structure is set to NOT NULL. Using this attribute eliminates the additional step of changing the indicator structure. You cannot change the default values provided by Oracle. Instead, you can populate the object with your own default values immediately after object creation.

When `OCIObjectNew()` is used to create a persistent object, the caller must identify the database table into which the newly created object is to be inserted. The caller identifies the table using a *table object*. Given the schema name and table name, the `OCIObjectPinTable()` function returns a pointer to the table object. Each call to `OCIObjectPinTable()` results in a call to the server to fetch the table object information. The call to the server happens even if the required table object has been previously pinned in the cache. When the application is creating multiple objects to be inserted into the same database table, Oracle Corporation recommends that the table object be pinned once and the pointer to the table object be saved for future use. Doing so improves performance of the application.

Updating an Object

Before you can update an object, the object must be pinned in the cache. After pinning the object, the application can update the desired attributes directly. You must make a call to the `OCIObjectMarkUpdate()` function to indicate that the object has been updated. Objects which have been marked as updated are placed in a dirty list and are flushed to the server upon cache flush or when the transaction is committed.

Deleting an Object

You can delete an object by calling the `OCIObjectMarkDelete()` function or the `OCIObjectMarkDeleteByRef()` function.

Controlling Object Cache Size

You can control the size of the object cache by using the following two OCI environment handle attributes:

- `OCI_ATTR_CACHE_MAX_SIZE` controls the maximum cache size
- `OCI_ATTR_CACHE_OPT_SIZE` controls the optimal cache size

You can get or set these OCI attributes using the *OCIAttrGet()* or *OCIAttrSet()* functions. Whenever memory is allocated in the cache, a check is made to determine whether the maximum cache size has been reached. If the maximum cache size has been reached, the cache automatically frees (ages out) the least-recently used objects with a pin count of zero. The cache continues freeing such objects until memory usage in the cache reaches the optimal size, or until it runs out of objects eligible for freeing. The object cache does not limit cache growth to the maximum cache size. The servicing of the memory allocation request could cause the cache to grow beyond the specified maximum cache size. The above two parameters allow the application to control the frequency of object aging from the cache.

Retrieving Objects into the Client Cache (Pinning)

Pinning is the process of retrieving an object from the server to the client cache, laying it in memory, providing a pointer to it for an application to manipulate, and marking the object as being in use. The *OCIObjectPin()* function de-references the given REF and pins the corresponding object in the cache. A pointer to the pinned object is returned to the caller and this pointer is valid as long as the object is pinned in the cache. This pointer *should not be used* after the object is unpinned because the object may have aged out and therefore may no longer be in the object cache.

The following are examples of *OCIObjectPin()* and *OCIObjectUnpin()* calls:

```
status = OCIObjectPin(envh, errh, empRef, (OCIComplexObject*)0,
                    OCI_PIN_RECENT, OCI_DURATION_TRANSACTION,
                    OCI_LOCK_NONE, (dvoid**)&emp);
/* manipulate emp object */
status = OCIObjectUnpin(envh, errh, emp);
```

The *empRef* parameter passed in the pin call specifies the REF to the desired employee object. A pointer to the employee object in the cache is returned via the *emp* parameter.

You can use the *OCIObjectPinArray()* function to pin an array of objects in one call. This function de-references an array of REFs and pins the corresponding objects in the cache. Objects that are not already cached in the cache are retrieved from the server in one network round-trip. Therefore, calling *OCIObjectPinArray()* to pin an array of objects improves application performance. Also, the array of objects you are pinning can be of different types.

Specifying which Version of an Object to Retrieve

When pinning an object, you can use the pin option argument to specify whether the recent version, latest version, or any version of the object is desired. The valid options are explained in more detail in the following list:

- The `OCI_PIN_RECENT` pin option instructs the object cache to return the object that is loaded into the cache in the current transaction; in other words, if the object was loaded prior to the current transaction, the object cache needs to refresh it with the latest version from the database. Succeeding pins of the object within the same transaction would return the cached copy and would not result in database access. In most cases, you should use this pin option.
- The `OCI_PIN_LATEST` pin option instructs the object cache to always get the latest copy of the object. If the object is already in the cache and not-locked, the object copy is refreshed with the latest copy from the database. On the other hand, if the object in the cache is locked, Oracle assumes that it is the latest copy, and the cached copy is returned. You should use this option for applications that must display the most recent copy of the object, such as applications that display stock quotes, current account balance, etc.
- The `OCI_PIN_ANY` pin option instructs the object cache to fetch the object in the most efficient manner; the version of the returned object does not matter. The pin any option is appropriate for objects which do not change often, such as product information, parts information, etc. The pin any option also is appropriate for read-only objects.

Specifying How Long to Keep the Object Pinned

When pinning an object, you can specify the duration for which the object is pinned in the cache. When the duration expires, the object is unpinned automatically from the cache. The application should not use the object pointer after the object's pin duration has ended. An object can be unpinned prior to the expiration of its duration by explicitly calling the *OCIObjectUnpin()* function. Oracle supports two pre-defined pin durations:

- The session pin duration (`OCI_DURATION_SESSION`) lifetime is the duration of the database connection. Objects that are required in the cache at all times across transactions should be pinned with session duration.
- The transaction pin duration (`OCI_DURATION_TRANS`) lifetime is the duration of the database transaction. That is, the duration ends when the transaction is rolled back or committed.

Specifying Whether to Lock the Object on the Server

When pinning an object, the caller can specify whether the object should be locked via *lock options*. When an object is locked, a server-side lock is acquired, which prevents any other user from modifying the object. The lock is released when the transaction commits or rolls back. The following list describes the available lock options:

- The `OCI_LOCK_NONE` lock option instructs the cache to pin the object without locking.
- The `OCI_LOCK_X` lock option instructs the cache to pin the object only after acquiring a lock. If the object is currently locked by another user, the pin call with this option waits until it can acquire the lock before returning to the caller. Using the `OCI_LOCK_X` lock option is equivalent to executing a `SELECT FOR UPDATE` statement.
- The `OCI_LOCK_X_NOWAIT` lock option instructs the cache to pin the object only after acquiring a lock. Unlike the `OCI_LOCK_X` option, the pin call with `OCI_LOCK_X_NOWAIT` option will not wait if the object is currently locked by another user. Using the `OCI_LOCK_X_NOWAIT` lock option is equivalent to executing a `SELECT FOR UPDATE WITH NOWAIT` statement.

How to Choose the Locking Technique

Depending upon how frequently objects are updated, you can choose which locking options from the previous section to use.

If objects are updated frequently, you can use the *pessimistic locking scheme*. This scheme presumes that contention for update access is frequent. Objects are locked before the object in the cache is modified, ensuring that no other user can modify the object until the transaction owning the lock performs a commit or rollback. The object can be locked at the time of pin by choosing the appropriate locking options. An object that was not locked at the time of pin also can be locked by the function `OCIObjectLock()`. A new locking function, `OCIObjectLockNoWait()`, has been added in Oracle8i. As the name indicates, this function does not wait to acquire the lock if another user holds a lock on the object.

If objects are updated infrequently, you can use the *optimistic locking scheme*. This scheme presumes that contention for update access is rare. Objects are fetched and modified in the cache without acquiring a lock. A lock is acquired only when the object is flushed to the server. Optimistic locking allows for a higher degree of concurrent access than pessimistic locking. To use optimistic locking effectively, the Oracle8i object cache detects if an object is changed by any other user since it was

fetched into the cache. By turning on the *object change detection mode*, object modifications are made persistent only if the object has not been changed by any other user since it was fetched into the cache. This mode is activated by setting `OCI_OBJECT_DETECTCHANGE` attribute of the environment handle using the `OCIAttrSet()` function.

Flushing an Object from the Object Cache

Changes made to the objects in the object cache are not sent to the database until the object cache is flushed. The `OCICacheFlush()` function flushes all changes in a single network round-trip between the client and the server. The changes may involve insertion of new objects into the appropriate object tables, updating objects in object tables, and deletion of objects from object tables. If the application commits a transaction by calling the `OCITransCommit()` function, the object cache automatically performs a cache flush prior to committing the transaction.

Pre-Fetching Related Objects (Complex Object Retrieval)

Complex Object Retrieval (COR) can significantly improve the performance of applications that manipulate graphs of objects. COR allows applications to pre-fetch a set of related objects in one network round-trip, thereby improving performance. When pinning the root object(s) using `OCIObjectPin()` or `OCIObjectPinArray()`, you can specify the related objects to be pre-fetched along with the root. The pre-fetched objects are not pinned in the cache; instead, they are put in the LRU list. Subsequent pin calls on these objects result in a cache hit, thereby avoiding a round-trip to the server.

The application specifies the set of related objects to be pre-fetched by providing the following information:

- A `REF` to the root object
- One or more pairs of object type and depth information to specify the content and boundary of objects to be pre-fetched. The type information indicates which `REF` attributes should be de-referenced and which resulting object should be pre-fetched. The depth defines the boundary of objects pre-fetched. The depth level is the shortest number of references that need to be traversed from the root object to a related object.

For example, consider a purchase order system with the following properties:

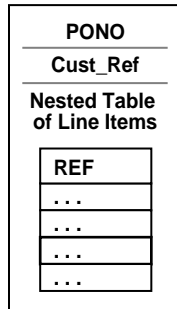
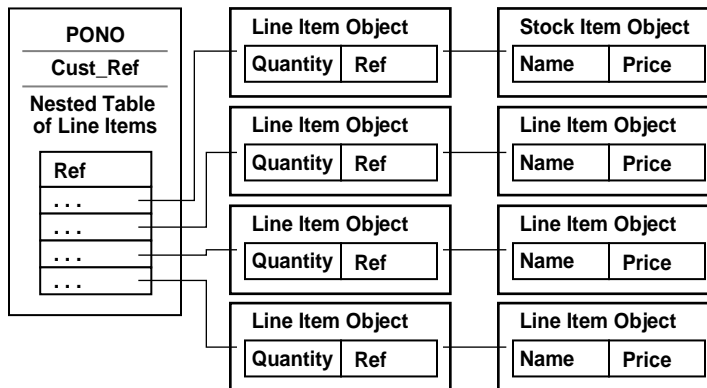
- Each purchase order object includes a purchase order number, a `REF` to a customer object, and a collection of `REFs` that point to line item objects.

- Each customer object includes information about the customer, such as the customer's name and address.
- Each line item object includes a reference to a stock item and the quantity of the order.
- Each stock item object includes the name of the item, its price, and other information about the item.

Suppose you want to calculate the total cost of a particular purchase order. To maximize efficiency, you want to fetch only the objects necessary for the calculation from the server to the client cache, and you want to fetch these objects with the least number of calls to the server possible.

If you do not use COR, your application must make several server calls to retrieve all of the necessary objects. However, if you use COR, you can specify the objects that you want to retrieve and exclude other objects that are not required. To calculate the total cost of a purchase order, you need the purchase order object, the related line item objects, and the related stock item objects, but you do not need the customer objects.

Therefore, as shown in [Figure 6–1](#), COR enables you to retrieve the required information for the calculation in the most efficient way possible. When pinning the purchase order object without COR, only that object is retrieved. When pinning it with COR, the purchase order and the related line item objects and stock item objects are retrieved. However, the related customer object is not retrieved because it is not required for the calculation.

Figure 6–1 Difference Between Retrieving an Object Without COR and With COR**Pinning of Purchase Order Object without COR****Pinning of Purchase Order Object with COR**

Demonstration of OCI and Oracle Objects

For a demonstration of how to use OCI with Oracle objects, see the `cdemocor1.c` file in `$ORACLE_HOME/rdbms/demo`.

Using the OCI Object Cache with View Objects

We can pin and navigate objects synthesized from object views in the OCI Object Cache similar to the way we do this with object tables. We can also create new view objects, update them, delete them and flush them from the cache. The flush performs the appropriate DML on the view (such as insert for newly created objects

and updates for any attribute changes). This fires any `INSTEAD-OF` triggers on the view and stores the object persistently.

There is a minor difference between the two approaches with regard to getting the reference to a newly created instance in the object cache.

In the case of object views with primary key based reference, the attributes that make up the identifier for the object need to be initialized before the `OCIObjectGetObjectRef` call can be called on the object to get the object reference. For example, to create a new object in the OCI Object cache for the purchase order object, we need to take the following steps:

```
.. /* Initialize all the settings including creating a connection, getting a
   environment handle etc. We do not check for error conditions to make
   the example easier to read. */
OCIType *purchaseOrder_tdo = (OCIType *) 0; /* This is the type object for the
   purchase order */
dvoid * purchaseOrder_viewobj = (dvoid *) 0; /* This is the view object */

/* The purchaseOrder struct is a structure that is defined to have the same
   attributes as that of PurchaseOrder_objtyp type. This can be created by the
   user or generated automatically using the OTT generator. */
purchaseOrder_struct *purchaseOrder_obj;

/* This is the null structure corresponding to the purchase order object's
   attributes */
purchaseOrder_nullstruct *purchaseOrder_nullobj;

/* This is the variable containing the purchase order number that we need to
   create */
int PONo = 1003;

/* This is the reference to the purchase order object */
OCIRef *purchaseOrder_ref = (OCIRef *)0;

/* Pin the object type first */
OCITypeByName( envhp, errhp, svchp,
               (CONST text *) "", (ub4) strlen( "" ) ,
               (CONST text *) "PURCHASEORDER_OBJTYP" ,
               (ub4) strlen("PURCHASEORDER_OBJTYP"),
               (CONST char *) 0, (ub4)0,
               OCI_DURATION_SESSION, OCI_TYPEGET_ALL,
               &purchaseOrder_tdo);

/* Pin the table object - in this case it is the purchase order view */
OCIObjectPinObjectTable(envhp, errhp, svchp, (CONST text *) "",
```



```
(ub4) strlen( "" ),
(CONST text *) "PURCHASEORDER_OBJVIEW",
(ub4 ) strlen("PURCHASEORDER_OBJVIEW"),
(CONST OCIRef *) NULL,
OCI_DURATION_SESSION,
&purchaseOrder_viewobj);

/* Now create a new object in the cache. This is a purchase order object */
OCIObjectNew(envhp, errhp, svchp, OCI_TYPECODE_OBJECT, purchaseOrder_tdo,
             purchaseOrder_viewobj, OCI_DURATION_DEFAULT, FALSE,
             (dvoid **) &purchaseOrder_obj);

/* Now we can initialize this object, and use it as a regular object. But before
getting the reference to this object we need to initialize the PONO attribute of
the object which makes up its object identifier in the view */

/* Initialize the null identifiers */
OCIObjectGetInd( envhp, errhp, purchaseOrder_obj, purchaseOrder_nullobj);

purchaseOrder_nullobj->purchaseOrder = OCI_IND_NOTNULL;
purchaseOrder_nullobj->PONO = OCI_IND_NOTNULL;

/* This sets the PONO attribute */
OCINumberFromInt( errhp, (CONST dvoid *) &PoNo, sizeof(PoNo), OCI_NUMBER_SIGNED,
                  &( purchaseOrder_obj->PONO));

/* Create an object reference */
OCIObjectNew( envhp, errhp, svchp, OCI_TYPECODE_REF, (OCITYPE *) 0,
             (dvoid *) 0, (dvoid *) 0, OCI_DURATION_DEFAULT, TRUE,
             (dvoid **) &purchaseOrder_ref);

/* Now get the reference to the newly created object */
OCIObjectGetObjectRef(envhp, errhp, (dvoid *) purchaseOrder_obj, purchaseOrder_
ref);

/* This reference may be used in the rest of the program .... */
...
/* We can flush the changes to the disk and the newly instantiated purchase
order object in the object cache will become permanent. In the case of the
purchase order object, the insert will fire the INSTEAD-OF trigger defined over
the purchase order view to do the actual processing */

OCICacheFlush( envhp, errhp, svchp, (dvoid *) 0, 0, (OCIRef **) 0);
...
```

Partitioning Tables that Contain Oracle Objects

Partitioning addresses the key problem of supporting very large tables and indexes by allowing you to decompose them into smaller and more manageable pieces called partitions. Oracle8i extends your partitioning capabilities by letting you partition tables that contain objects, REFS, varrays, and nested tables. Varrays stored in LOBs are equipartitioned in a way similar to LOBs.

The following example partitions the purchase order table along zip codes (ToZip), which is an attribute of the ShipToAddr embedded column object. For the purposes of this example, the LineItemList nested table was made a varray to illustrate storage for the partitioned varray.

Restriction: Nested tables are allowed in tables that are partitioned; however, the storage table associated with the nested table is not partitioned.

Assuming that the LineItemList is defined as a varray:

```
CREATE TYPE LineItemList_vartyp AS varray(10000) OF LineItem_objtyp;
```

```
CREATE TYPE PurchaseOrder_typ AS OBJECT (  
    PONo                NUMBER,  
    Cust_ref            REF Customer_objtyp,  
    OrderDate           DATE,  
    ShipDate            DATE,  
    OrderForm           BLOB,  
    LineItemList         LineItemList_vartyp,  
    ShipToAddr          Address_objtyp,  

```

```
    MAP MEMBER FUNCTION  
        ret_value RETURN NUMBER,  

```

```
    MEMBER FUNCTION  
        total_value RETURN NUMBER  
);
```

```
CREATE TABLE PurchaseOrders_tab OF PurchaseOrder_typ  
    LOB (OrderForm) STORE AS (NOCACHE LOGGING)  
    PARTITION BY RANGE (ShipToAddr.zip)  
        (PARTITION PurOrderZone1_part  
            VALUES LESS THAN ('59999')  
            LOB (OrderForm) STORE AS (
```

```

        storage (INITIAL 10 MINEXTENTS 10 MAXEXTENTS 100))
    VARRAY LineItemList store as LOB (
        storage (INITIAL 10 MINEXTENTS 10 MAXEXTENTS 100)),
    PARTITION PurOrderZone6_part
        VALUES LESS THAN ('79999')
        LOB (OrderForm) store as (
            storage (INITIAL 10 MINEXTENTS 10 MAXEXTENTS 100))
        VARRAY LineItemList store as LOB (
            storage (INITIAL 10 MINEXTENTS 10 MAXEXTENTS 100)),
    PARTITION PurOrderZoneO_part
        VALUES LESS THAN ('99999')
        LOB (OrderForm) store as (
            storage (INITIAL 10 MINEXTENTS 10 MAXEXTENTS 100))
        VARRAY LineItemList store as LOB (
            storage (INITIAL 10 MINEXTENTS 10 MAXEXTENTS 100));

```

Parallel Query with Object Views

Parallel query is supported on the objects synthesized from views.

To execute queries involving joins and sorts (using the `ORDER BY`, `GROUP BY`, and `SET` operations) in parallel, a `MAP` function is needed. In the absence of a `MAP` function, the query automatically becomes serial.

Parallel queries on nested table columns are not supported. Even in the presence of parallel hints or parallel attributes for the view, the query will be serial if it involves the nested table column.

Parallel DML is not supported on views with `INSTEAD-OF` trigger. However, the individual statements within the trigger may be parallelized.

How Locators Improve the Performance of Nested Tables

In Oracle8i, the collection typed value does not map directly to a native type or structure in languages such as C++ and Java. An application using those languages must access the contents of a collection through Oracle interfaces, such as OCI.

Generally, when the client accesses a nested table explicitly or implicitly (by fetching the containing object), Oracle returns the entire collection value to the client process. For performance reasons, a client may wish to delay or avoid retrieving the entire contents of the collection. Oracle handles this case for you by using a locator instead of the real nested table value. When you really access the contents of the collection, they are automatically transferred to the client.

A nested table locator is like a handle to the collection value. It attempts to preserve the value or copy semantics of the nested table by containing the database snapshot as of its time of retrieval. The snapshot helps the database retrieve the correct instantiation of the nested table value at a later time when the collection elements are fetched using the locator. The locator is scoped to a session and cannot be used across sessions. Since database snapshots are used, it is possible to get a "snapshot too old" error if there is a high update rate on the nested table. Unlike a LOB locator, the nested table locator is truly a locator and cannot be used to modify the collection instance.

Frequently Asked Questions about Programming with Oracle Objects

Here are some questions and answers that new users often have about Oracle's object-relational features:

- [General Questions about Oracle Objects](#)
- [Object Types](#)
- [Object Methods](#)
- [Object References](#)
- [Collections](#)
- [Object Views](#)
- [Object Cache](#)
- [Large Objects \(LOBs\)](#)
- [User-Defined Operators](#)

You can use this chapter as introductory information, or refer here if you still have questions after reading the rest of the book.

General Questions about Oracle Objects

Are the object-relational features a separate option?

Not anymore. As of Version 8.1, they are part of the base server product.

What are the design goals of Oracle8i Object-Relational & Extensibility technologies?

The design goals of Oracle8i Objects and Extensibility technologies are to:

- Provide users with the ability to model their business objects in the database by enhancing the type system to support user-defined types. These types are meant to closely model application objects and are treated as built-in types, such as number and character, by the database server.
- Provide an infrastructure to facilitate object-based access to data stored in an Oracle database and minimize the potential mismatch between the data model used in an application and the data model supported by a database.
- Provide built-in support for new data types needed in multi-media, financial and spatial applications.
- Provide a framework for database extensibility so that new multimedia and complex data types can be supported and managed natively in the database. This framework provides the infrastructure needed to allow extensions of the data server by third parties via data cartridges.

This book talks about the object-relational technologies. For details about extensibility, see *Oracle8 Data Cartridge Developer's Guide*.

What are the key features in Oracle8i Object-Relational Technology?

Building on the standard features of the Oracle data server, the Oracle8i Objects is an additional set of features that enable the creation and manipulation of user-defined object types:

Object Type System

Oracle8i provides the capability to define new types in the database via a new meta-model, simply called the Oracle8i Type System. This type system extends the Oracle relational types to allow users to define types and methods that model objects. Oracle8i stores the meta-data for user-defined types in a schema that is available to SQL, PL/SQL, Java and other published interfaces. A user can create a

new object type by using any built-in database types, and/or any known object types, object references and collection types.

Object Views

In addition to natively storing object data in the server, Oracle8i allows the creation of an object abstraction over existing relational data via the object view mechanism. Objects belonging to an object view are accessed in the same manner as row objects via SQL or other call interfaces. To support such data access, Oracle8i server materializes objects of user-defined types from data stored in relational schemas and tables. This view mechanism supports the development of new object-oriented applications without having to modify existing database schemas.

SQL Object Extensions

To support the new features of the Objects Option, extensions have been made to SQL -- including new DDL -- to create, alter, or drop object types; to store object types in tables; and to create, alter, or drop object views. There are DML and query extensions to support object types, references, and collections.

PL/SQL Object Extensions

PL/SQL is Oracle's database programming language that is tightly integrated with SQL. With the addition of user-defined types and other SQL types introduced in Oracle8i, PL/SQL has been enhanced to operate on user-defined types seamlessly. Thus, application developers can use PL/SQL to implement logic and operations on user-defined types that execute in the database server.

Java Support for Oracle8i Objects

Oracle8i's Java VM is tightly integrated with the RDBMS and supports access to Oracle8i Objects via object extensions to JDBC (dynamic SQL) and SQLJ (static SQL). Thus, application developers can use the Java to implement logic and operations on user-defined types that execute in the database server.

External Procedures

Database functions, procedures, or member methods of an object type can be implemented in PL/SQL, Java, or C as external procedures. External procedures are best suited for tasks that are more quickly or easily done in a low-level language such as C, which is more efficient at machine-precision calculation. External procedures are always run in a safe mode outside the address space of the RDBMS server.

Object Type Translator

The Object Type Translator (OTT) available with the Objects Option provides client-side mappings to object type schema by generating header files containing Java classes and C structs and indicators, using schema information from the Oracle data dictionary. These generated header files can be used in host-language applications for transparent access to database objects.

Client-Side Cache

Oracle8i provides an object cache for efficient access to persistent objects stored in the database. Copies of objects can be brought into the object cache. Once the data has been cached in the client, the application can traverse through these at memory speed. Any changes made to objects in the cache can be committed to the database by using the object extensions to Oracle Call Interface programmatic interfaces.

Complex Object Retrieval

Oracle8i also provides support for efficient complex object retrieval. That is, a single request to fetch an object from the server can be used to retrieve other objects, which are connected to the object being fetched via object references, in a single round-trip between the client and the server. Such functionality facilitates the fetching and manipulation of a set of related objects as a single unit.

OCI Object Extensions

Oracle8i Oracle Call Interface provides a comprehensive application programming interface for application and tool developers seeking to use the object capabilities of Oracle8. Oracle Call Interface provides a run-time environment with functions to connect to an Oracle8 server and control transactions that access objects in the server. It allows application developers to access and manipulate objects and their attributes in the client-side object cache either "navigationally" by traversing a graph of inter-connected objects or "associatively" by specifying the nature of the data via declarative SQL DML. Oracle Call Interface also provides a number of functions for accessing meta-data information at run-time about object types defined in the server. Such a set of functions facilitates dynamic access to the object meta-data and the actual object data stored in the database.

PRO*C/C++ Object Extensions

Oracle8i Pro*C precompiler provides an embedded SQL application programming interface and offers a higher level of abstraction than Oracle Call Interface. Like Oracle Call Interface, the Pro*C precompiler allows application developers to use the Oracle8i client-side object cache and the Object Type Translator Utility. Pro*C

supports the use of "C" bind variables for Oracle8i object types. Furthermore, Pro*C provides new simplified syntax to allocate and free objects of SQL types and access them by either SQL DML, or via the "navigational" interface. Thus, it provides application developers many benefits, including compile-time type checking of (client-side) bind variables against the schema in the server, automatic mapping of object data in an Oracle8i server to program bind variables in the client, and simple ways to manage and manipulate database objects in the client process.

OO4O Object Extensions

Oracle8i Oracle Objects For OLE (OO4O) is a set of COM Automation interfaces/objects for connecting to Oracle8i database servers, executing queries and managing the results. Automation interfaces in OO4O provide easy and efficient access to features that are unique to Oracle8i, and can be used from virtually any programming or scripting language that supports the Microsoft COM Automation technology. This includes Visual Basic, Visual C++, VBA in Excel, VBScript and JavaScript in IIS Active Server Pages.

Integration with Relational Functionality

Oracle8i Objects continues to support standard relational database functionality such as queries (SELECT...FROM...WHERE), fast commits, backup and recovery, scalable connectivity, row-level locking, read consistency, partitioned tables, parallel queries, parallel server, export/import, loader etc.

What are the new Object-Relational features in Oracle8i?

Oracle8i provides the foundation for modeling complex objects. Here is a list of object-relational features, with the new 8i features in italics:

Type System

Scalars, LOBs, Objects, References, Collections

Execution Environment

PL/SQL methods, External Procedures, *Java Methods*

Query Processing

Triggers, Constraints, Object Views, *User-Defined Operators*

Data Indexing

Sorted, Hash, Bitmap, Index-Organized Tables, *Extensible Indexing*

Query Optimization

Object Query Optimization, *Extensible Optimizer*

Operational Completeness

Object Support in Export/Import, *Loader, Parallel Query, Partitioning*

Language Interfaces

Object Support in OCI, C++ (ODD), Pro*C, *JDBC, OO4O*

Object Types

What is structured data?

The SQL 92 standard defines the 19 atomic datatypes that are used in most database programming. We refer to these kinds of data as "simple structured".

Oracle Objects introduces the ideas of REFs and collections. We refer to these kinds of data as "complex structured".

LOBs provide another way to store information. We refer to them as "unstructured".

Where are the user-defined types, user-defined functions, and abstract data types?

The Oracle equivalent of a user-defined type or an abstract data type is an object type.

The Oracle equivalent of a user-defined function is an object type method.

We use these terms because their semantics are different from the common industry usage. For example, an Oracle object can be null, while an object of an abstract data type cannot.

What is an object type?

Oracle8i supports a form of user-defined data types called object types. Object types are abstractions of real-world entities. An object type is a schema object with the following components:

- A name, which identifies the object type uniquely within a schema
- Attributes, which model the structure and state of the real-world entity
- Methods, which implement the behavior of the real-world entity

Why are object types useful?

An object type is similar to the class mechanism supported by C++ and Java. Object reusability provides faster and more efficient database application development. Object support makes it easier to model complex, real-world business entities and logic. By supporting object types natively in the database, Oracle8i eliminates the impedance mismatch between object-oriented programming languages and the database, and relieves application developers from the need to write a mapping layer between client-side objects and database objects. Object abstraction and encapsulation also make it easier to understand and maintain applications, an important consideration in enterprise database application development.

How is object data stored and managed in Oracle8i?

Objects are managed natively by the data server. Object types can be used as the type of a column (column objects) or as type of each row in an object table (row objects). When used as column objects, object types serve as classical relational domains. Each row object has a unique identity, called an object identifier (OID).

Objects are first-class citizens and are fully integrated with the database components. They can be indexed and partitioned. For example, queries involving objects can be parallelized and are optimized by the cost-based optimizer using statistics.

By building on the proven foundation of the Oracle8i data server, objects are managed with the same reliability, availability, and scalability as relational data.

Is inheritance supported in Oracle8i?

Not directly. Because each language has its own semantics for inheritance -- for example, single inheritance versus multiple inheritance -- Oracle uses techniques that mimic the inheritance behavior of each language.

Oracle8i provides support for client-side inheritance via its C++ and Java mappings. For C++, use the Object Modelling Option of Oracle Designer to produce DDL and C++ code based on diagrams in the Universal Modelling Language (UML). For Java, use the "custom datum" feature of the Oracle JDBC driver.

Server-side method inheritance is provided in Java by the Oracle8i Java VM. Oracle8i does not currently support inheritance in SQL, the ability to store and query instances of a type and its subtypes.

Object Methods

What language can I use to write my object methods?

Methods can be implemented in PL/SQL, Java, C or C++. C & C++ support is via the external procedure functionality in Oracle8i, whereas PL/SQL and Java methods run within the address space of the server. De-coupling of the specification of a method in SQL from its implementation provides a uniform way to invoke methods on object types, even though these object types can be implemented in various programming languages. Oracle8i provides a safe and secure environment for invoking PL/SQL methods, Java methods, and external C procedures from the server. Programming errors in user methods will not crash the server or corrupt the database, thus ensuring the reliability and availability of the server in a mission critical environment.

How do I decide between using PL/SQL and Java for my object methods?

With Oracle8i, PL/SQL and Java can be used interchangeably as a server programming language. PL/SQL is a seamless extension of SQL and is the language of choice for doing SQL intensive operations in the database. Java is the emerging language of choice for writing portable applications that run in multiple tiers, including the database server.

When should I use external procedures?

External procedures are typically used for compute intensive operations that are best written in a low-level language such as C. External procedures are also useful for invoking routines in some existing libraries that cannot be easily rewritten in Java or PL/SQL to run in the data server.

The IPC (inter-process communication) overhead of invoking an external procedure is an order of magnitude higher than that of invoking PL/SQL or Java procedure. However, the overhead of invoking an external procedure become insignificant if the computation done in the external procedure is complex and is in the order of tens of thousands of instructions.

What are definer and invoker rights?

The distinction between definer and invoker rights applies to more than just objects. You may find invoker rights especially useful for object-oriented programs because they typically contain reusable modules.

An object method can be executed with the privileges of its owner (definer rights) or with the privileges of the current user (invoker rights), based on the method definition. Invoker rights are useful for writing reusable objects because users of these objects do not have to grant access privileges to their tables to the implementor of the objects. Definer rights are useful when the as part of the object implementation, the object methods need to access some meta-data maintained by the object implementor. Methods that access the meta-data are executed using the definer rights so that the object implementor does not have to expose the proprietary meta-data to the users.

Object References

What is an object reference?

An object reference (REF) uniquely identify a row object stored in an object table or an object constructed from an object view. Typically, a REF value is comprised of the object's unique identifier, the unique identifier associated with the object table, and the ROWID of the row in the object table in which the object is stored. The optional ROWID is used as a hint to provide fast access to the object.

When should I use object references? How are they different from foreign keys?

Object references, like foreign keys, are useful in modeling complex relationships. Object references are more flexible than foreign keys for modeling complex relationships because:

- Object references are strongly typed and this provides better compile-time type checking
- One-to-many relationships can be modeled using a collection of object references
- Application can easily navigate and retrieve objects using object references without having to construct SQL statements
- REF navigation in SQL avoids the need to do complicated multi-table joins

- Object references allow applications to retrieve objects connected by REFs in a single request to the server

Can I construct object references based on primary keys?

Yes, object references can be constructed based on foreign keys to reference objects in:

- Object views: When constructing objects from relational tables using an object view, the OIDs of the constructed objects are typically based on the primary keys on the underlying relational tables.
- Object tables with primary key-based OIDs: When defining an object table, Oracle8i provides the option of specifying the primary keys as the OIDs of the row objects instead of using the system generated OIDs.

What is a scoped REF and when should I use it?

In general, a column may contain references to objects of a particular declared type regardless of the object table(s) in which the objects are stored. However, a REF type column may be scoped (constrained) to only contain references to objects from a specified object table. One should use scoped REFs whenever possible because scoped REFs are smaller in size than regular REFs on disk because the system does not have to store the table identifier with the scoped REFs. Also, queries containing navigation of scoped REFs can be optimized into joins when appropriate.

Can I manipulate objects using object references in PL/SQL and Java?

Yes, both PL/SQL and Java support object references. In PL/SQL, an object can be retrieved and updated using the UTL_REF package given its object references. In Java, object references are mapped to reference classes with get and set methods to retrieve and update the objects.

Collections

What kinds of collections are supported by Oracle8i?

Oracle8i supports two types of collections: varying arrays (varrays) and nested tables. Attributes of object types and columns of tables can be of collection types. By using varying arrays and nested tables, applications can model one-to-many and many-to-many relationships natively in their database schema.

How do I decide between using varrays and nested tables for modeling collections?

Varrays are useful when you need to maintain ordering of your collection elements. Varrays are very efficient when you always manipulate the entire collection as a unit, and that you don't require querying on individual elements of the collections. Varrays are stored inline with the containing row if it is small and automatically stored as a LOBs by the system when its size is beyond a certain threshold.

Nested tables are useful when there is no ordering among the collection elements and that efficient querying on individual elements are important. Elements of a collection type column are stored in a separate table, similar to parent-child tables in a relational schema.

Do Oracle8i Objects support collections within collections?

A collection cannot contain another collection attribute. However, nested collections can be modeled using a reference to an object which has a collection attribute. Therefore applications can model nested collections with REF indirection.

What is a collection locator?

Collection locators allow applications to retrieve large collections without materializing the collections in memory. This allows for efficient transfer of large collections across interfaces. A collection will be transparently materialized when the application first accesses its elements. Also, applications can query and retrieve subsets of the collection using its locator.

The specification of retrieval of collection locators is done in CREATE and ALTER TABLE DDL. Since access to a collection is encapsulated, applications will use the same interface to retrieve a nested table specified to be returned as a locator as one specified to be returned as a value.

What is collection unnesting?

Collection unnesting allows applications to efficiently query over a set of collections in some specified rows, similar to query on the child rows in a relational schema for some specified parent rows. Collection unnesting allows applications the flexibility to view one-to-many relationships in the collection form or in the flat parent-child form.

Object Views

What are the differences between object views and object tables?

Like the similarity between relational views and tables, an object view has properties similar to an object table:

- It contains objects in rows. The columns of the view map to top-level attributes of the object type.
- Each object has an identifier associated with it. The identifier is specified by the view definer; in most cases, the primary key of the base table serves as the identifier.

Are object views updateable?

It is easy to update an object view where every attribute maps back to a real column in a table. For views that derive some attributes by more complex techniques, such as CAST-MULTISET, INSTEAD-OF triggers can be used to do the updates. When such a view is updated (or inserted into or deleted from), rather than attempting to implicitly modify any base tables, the system simply invokes the INSTEAD-OF trigger specified for the view. You can encapsulate whatever update semantics you want in the trigger body.

Object Cache

Why do we need the object cache?

The object cache gives applications the following benefits:

- Transparent mapping of database objects to host language objects in memory.
- Transparent, efficient memory management for persistent objects. Applications do not have to worry about allocation of memory for accessing database objects.
- Transactional semantics for client-side objects. Modified persistent objects in the object cache can be propagated (flushed) to the database in a single round-trip between the client and the server.
- Navigational object access. Object cache allows for navigational style of object access. Using OCI's object functions, objects can be fetched into the object cache by pinning object REFs. Navigational style of object access may be more

suitable when operating on a graph of objects that are inter-connected via object REFs.

- Complex object retrieval. That is, a single request to fetch an object from the server can be used to retrieve other objects, which are connected to the object being fetched via REFs, in a single round-trip between the client and the server.

Does the object cache support object locking?

The object cache supports both a pessimistic locking scheme and an optimistic locking scheme.

- In the pessimistic locking scheme, objects are locked up-front in the server prior to modifying the object in the cache. This ensures no other user can modify the object until the transaction owning the lock commits/rollbacks.
- In the optimistic locking scheme, an object is fetched and modified in the cache without acquiring a lock. The lock is acquired only when the object is flushed to the server. Optimistic locking allows for a higher degree of concurrent access than pessimistic locking. To use optimistic locking effectively, the object cache provides the ability for detecting if an object was changed by any other user since it was fetched into the cache. By turning on the "object change detection mode", object modifications will be made persistent if the nobody else has changed the object since it was fetched into the cache.

Large Objects (LOBs)

How can I manage large objects using Oracle8i?

Support for multimedia data types like text, images, audio, and video requires robust support for binary and character data. The data in these domains tends to be large and requires direct access to different pieces of the binary data. To address this need, Oracle8i provides significantly improved support for large-scale binary and character data. It introduces the Large Object type (LOB) which can be used to store large, domain-specific data from various domains, including images, audio files, text and spatial data.

Oracle8i supports three kinds of large data objects: binary, character-based, and file-based. In addition to providing the ability to create LOBs, Oracle8i server provides several other improvements in managing binary data. These improvements can be summarized as follows:

- Support for defining more than one LOB column in a table

- Random, piece-wise access to LOB data
- Support for transferring LOB data as a single stream
- Support for disabling logging and/or caching for LOB data
- Support for transparently moving LOBs from "in-line" row storage to "out-of-line" storage in another segment or even another tablespace

For more information about LOBs, see *Oracle8i Application Developer's Guide - Large Objects (LOBs)*.

User-Defined Operators

What is a user-defined operator?

Oracle8i allows developers of object-oriented applications to extend the list of built-in relational operators (for example, +, -, /, *, LIKE) with domain specific operators (for example, Contains, Within_Distance, Similar) called user-defined operators. A user-defined operator can be used anywhere built-in operators can be used, for example, in the select list or the where clause. Similar to built-in operators, a user-defined operator may support arguments of different types, and that it may be evaluated using an index.

For more information about user-defined operators, see CREATE OPERATOR in the *Oracle8i SQL Reference*, and the *Oracle8 Data Cartridge Developer's Guide*.

Why are user-defined operators useful?

Similar to built-in operators, user-defined operators allow efficient content-based querying and sorting on object data. For example, to find a resume containing certain qualifications, one may specify the Contains operator as part of the SQL where clause. The optimizer may choose to use a Text index on the resume column to perform the query efficiently, similar to using a B-tree index to evaluate a relational operator.

A Sample Application using Object-Relational Features

This chapter has an extended example of how to use user-defined datatypes (Oracle objects). The example shows how a relational model might be transformed into an object-relational model that better represents the real-world entities that are managed by an application.

This chapter contains the following sections:

- [Introduction](#)
- [A Purchase Order Example](#)
 - [Implementing the Application Under The Relational Model](#)
 - [Implementing the Application Under The Object-Relational Model](#)
- [Manipulating Objects Through Java](#)
- [Manipulating Objects with Oracle Objects for OLE](#)

Introduction

User-defined types are schema objects in which users formalize the data structures and operations that appear in their applications.

The example in this chapter illustrates the most important aspects of defining and using user-defined types. One important aspect of using user-defined types is creating methods that perform operations on objects. In the example, definitions of object type methods use the PL/SQL language. Other aspects of using user-defined types, such as defining a type, use SQL.

PL/SQL and Java provide additional capabilities beyond those illustrated in this chapter, especially in the area of accessing and manipulating the elements of collections.

Client applications that use the Oracle Call Interface (OCI), Pro*C/C++, or Oracle Objects for OLE (OO4O) can take advantage of its extensive facilities for accessing objects and collections, and manipulating them on clients.

See Also:

- *Oracle8i Concepts* for an introduction to user-defined types and instructions on how to use them.
 - *Oracle8i SQL Reference* for a complete description of SQL syntax and usage for user-defined types.
 - *PL/SQL User's Guide and Reference* for a complete discussion of PL/SQL capabilities
 - *Oracle8i Java Stored Procedures Developer's Guide* for a complete discussion of Java.
 - *Oracle Call Interface Programmer's Guide*,
 - *Pro*C/C++ Precompiler Programmer's Guide*
 - *Oracle Objects for OLE/ActiveX Programmer's Guide*
-
-

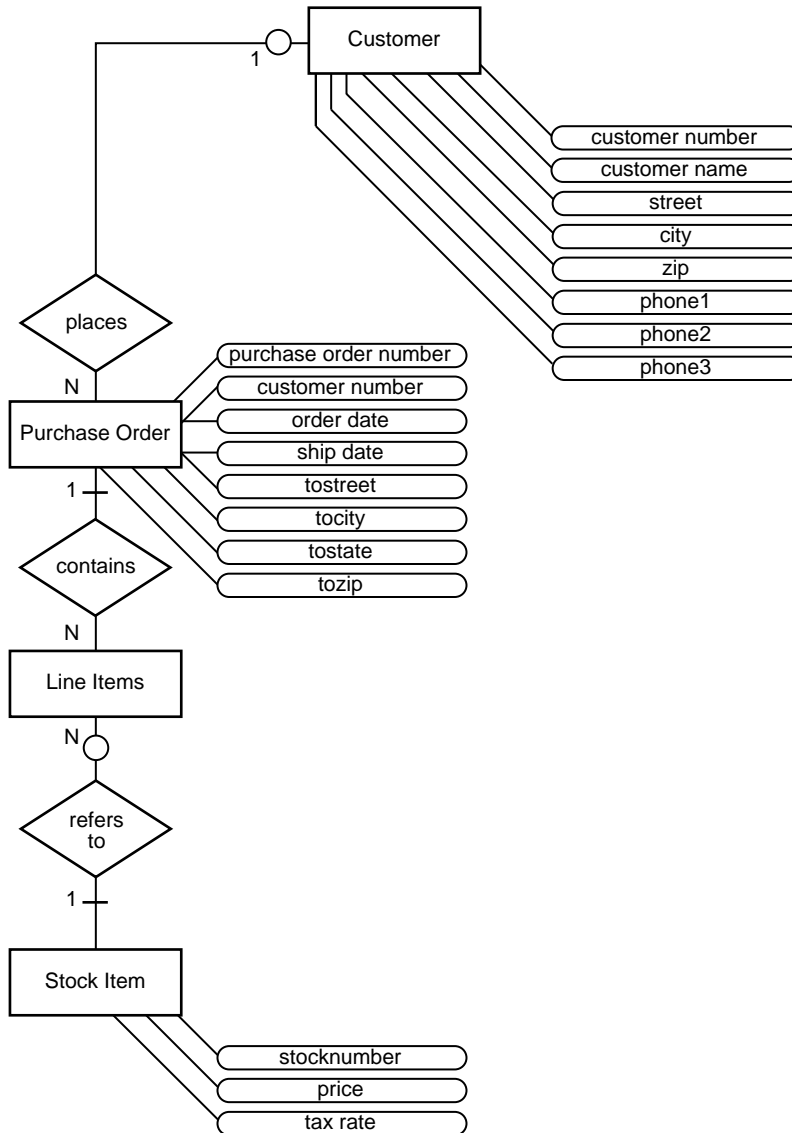
A Purchase Order Example

This example is based on a typical business activity: managing customer orders. We demonstrate how the application might evolve from relational to object-relational, and how you could write it from scratch using a pure object-oriented approach.

- First, we show how to implement the schema — **Implementing the Application Under The Relational Model** — using only Oracle's built-in datatypes. You can build an object-oriented application on top of this relational schema using object views, as described in [Chapter 4](#)
- **Implementing the Application Under The Object-Relational Model** uses Oracle's object types to represent the entities and relationships of the application domain. It uses object tables to hold the underlying data, and encapsulates the behavior of objects in method functions.

Implementing the Application Under The Relational Model

Figure 8–1 Entity-Relationship Diagram for Purchase Order Application



Entities and Relationships

The basic entities in this example are:

- Customers
- The stock of products for sale
- Purchase orders

As you can see from [Figure 8–1](#), a customer has contact information, so that the address and set of telephone numbers is exclusive to that customer. The application does not allow different customers to be associated with the same address or telephone numbers. If a customer changes her address, the previous address ceases to exist. If someone ceases to be a customer, the associated address disappears.

A customer has a one-to-many relationship with a purchase order: a customer can place many orders, but a given purchase order is placed by one customer. Because a customer can be defined before they place an order, the relationship is optional rather than mandatory.

Similarly, a purchase order has a many-to-many relationship with a stock item. Because this relationship does not show which stock items appear on which purchase orders, the entity-relationship has the notion of a line item. A purchase order must contain one or more line items. Each line item is associated only with one purchase order.

The relationship between line item and stock item is that a stock item can appear on zero, one, or many line items, but each line item refers to exactly one stock item.

Creating Tables Under the Relational Model

The relational approach normalizes everything into tables. The table names are `Customer_reltab`, `PurchaseOrder_reltab`, and `Stock_reltab`.

Each part of an address becomes a column in the `Customer_reltab` table.

Structuring telephone numbers as columns sets an arbitrary limit on the number of telephone numbers a customer can have.

The relational approach separates line items from their purchase orders and puts each into its own table, named `PurchaseOrder_reltab` and `LineItems_reltab`. As depicted in [Figure 8-1](#), a line item has a relationship to both a purchase order and a stock item. These are implemented as columns in `LineItems_reltab` table with foreign keys to `PurchaseOrder_reltab` and `Stock_reltab`.

Note: We have adopted a convention in this section of adding the suffix `_reltab` to the names of relational tables. Such a self-describing notation can make your code easier to maintain.

You may find it useful to make distinctions between tables (`_tab`) and types (`_typ`). But you can choose any names you want; one of the main advantages of object-relational methods is that the names of software entities can closely model real-world objects.

The relational approach results in the following tables:

Customer_reltab

The `Customer_reltab` table has the following definition:

```
CREATE TABLE Customer_reltab (  
  CustNo          NUMBER NOT NULL,  
  CustName        VARCHAR2(200) NOT NULL,  
  Street          VARCHAR2(200) NOT NULL,  
  City            VARCHAR2(200) NOT NULL,  
  State           CHAR(2) NOT NULL,  
  Zip             VARCHAR2(20) NOT NULL,  
  Phone1          VARCHAR2(20),  
  Phone2          VARCHAR2(20),  
  Phone3          VARCHAR2(20),  
  PRIMARY KEY (CustNo)  
);
```

This table, `Customer_reltab`, stores all the information about customers, which means that it fully contains information that is intrinsic to the customer (defined with the `NOT NULL` constraint) and information that is not as essential. According to this definition of the table, the application requires that every customer have a shipping address.

Our Entity-Relationship (E-R) diagram showed a customer placing an order, but the table does not make allowance for any relationship between the customer and the

purchase order. This suggests that the relationship must be managed by the purchase order.

PurchaseOrder_reltab

The PurchaseOrder_reltab table has the following definition:

```
CREATE TABLE PurchaseOrder_reltab (
  PONo      NUMBER, /* purchase order no */
  Custno    NUMBER references Customer_reltab, /* Foreign KEY referencing
                                                customer */
  OrderDate DATE, /* date of order */
  ShipDate  DATE, /* date to be shipped */
  ToStreet  VARCHAR2(200), /* shipto address */
  ToCity    VARCHAR2(200),
  ToState   CHAR(2),
  ToZip     VARCHAR2(20),
  PRIMARY KEY(PONo)
) ;
```

As expected, PurchaseOrder_reltab manages the relationship between the customer and the purchase order by means of the foreign key (FK) column CustNo, which references the CustNo key of the PurchaseOrder_reltab. Because the table makes no allowance for the relationship between the purchase order and its line items, the list of line items must handle this.

LineItems_reltab

The LineItems_reltab table has the following definition:

```
CREATE TABLE LineItems_reltab (
  LineItemNo  NUMBER,
  PONo        NUMBER REFERENCES PurchaseOrder_reltab,
  StockNo     NUMBER REFERENCES Stock_reltab,
  Quantity    NUMBER,
  Discount    NUMBER,
  PRIMARY KEY (PONo, LineItemNo)
) ;
```

Note: The Stock_reltab table, describe in "[Stock_reltab](#)" on page 8-8, must be created before the LineItems_reltab table.

The table name is in the plural form LineItems_reltab to emphasize to someone reading the code that the table holds a collection of line items.

As shown in the E-R diagram, the list of line items has relationships with both the purchase order and the stock item. These relationships are managed by `LineItems_reltab` by means of two foreign key columns:

- `PONo`, which references the `PONo` column in `PurchaseOrder_reltab`
- `StockNo`, which references the `StockNo` column in `Stock_reltab`

Stock_reltab

The `Stock_reltab` table has the following definition:

```
CREATE TABLE Stock_reltab (  
    StockNo      NUMBER PRIMARY KEY,  
    Price        NUMBER,  
    TaxRate      NUMBER  
);
```

Inserting Values Under the Relational Model

In our application, statements like these insert data into the tables:

Establish Inventory

```
INSERT INTO Stock_reltab VALUES(1004, 6750.00, 2) ;  
INSERT INTO Stock_reltab VALUES(1011, 4500.23, 2) ;  
INSERT INTO Stock_reltab VALUES(1534, 2234.00, 2) ;  
INSERT INTO Stock_reltab VALUES(1535, 3456.23, 2) ;
```

Register Customers

```
INSERT INTO Customer_reltab  
VALUES (1, 'Jean Nance', '2 Avocet Drive',  
        'Redwood Shores', 'CA', '95054',  
        '415-555-1212', NULL, NULL) ;  
  
INSERT INTO Customer_reltab  
VALUES (2, 'John Nike', '323 College Drive',  
        'Edison', 'NJ', '08820',  
        '609-555-1212', '201-555-1212', NULL) ;
```

Place Orders

```
INSERT INTO PurchaseOrder_reltab  
VALUES (1001, 1, SYSDATE, '10-MAY-1997',  
        NULL, NULL, NULL, NULL) ;
```

```
INSERT INTO PurchaseOrder_reltab
VALUES (2001, 2, SYSDATE, '20-MAY-1997',
       '55 Madison Ave', 'Madison', 'WI', '53715') ;
```

Detail Line Items

```
INSERT INTO LineItems_reltab VALUES(01, 1001, 1534, 12, 0) ;
INSERT INTO LineItems_reltab VALUES(02, 1001, 1535, 10, 10) ;
INSERT INTO LineItems_reltab VALUES(01, 2001, 1004, 1, 0) ;
INSERT INTO LineItems_reltab VALUES(02, 2001, 1011, 2, 1) ;
```

Querying Data Under The Relational Model

The application can execute queries like these:

Get Customer and Line Item Data for a Specific Purchase Order

```
SELECT  C.CustNo, C.CustName, C.Street, C.City, C.State,
        C.Zip, C.phone1, C.phone2, C.phone3,
        P.PONo, P.OrderDate,
        L.StockNo, L.LineItemNo, L.Quantity, L.Discount
FROM    Customer_reltab C,
        PurchaseOrder_reltab P,
        LineItems_reltab L
WHERE   C.CustNo = P.CustNo
AND     P.PONo = L.PONo
AND     P.PONo = 1001 ;
```

Get the Total Value of Purchase Orders

```
SELECT      P.PONo, SUM(S.Price * L.Quantity)
FROM        PurchaseOrder_reltab P,
            LineItems_reltab L,
            Stock_reltab S
WHERE       P.PONo = L.PONo
AND        L.StockNo = S.StockNo
GROUP BY   P.PONo ;
```

Get the Purchase Order and Line Item Data for those LineItems that Use a Stock Item Identified by a Specific Stock Number

```
SELECT      P.PONo, P.CustNo,
            L.StockNo, L.LineItemNo, L.Quantity, L.Discount
FROM        PurchaseOrder_reltab P,
```

```
                LineItems_reltab      L
WHERE    P.PONo = L.PONo
AND      L.StockNo = 1004 ;
```

Updating Data Under The Relational Model

The application can execute statements like these to update the data:

Update the Quantity for Purchase Order 1001 and Stock Item 1534

```
UPDATE LineItems_reltab
SET      Quantity = 20
WHERE    PONo      = 1001
AND      StockNo   = 1534 ;
```

Deleting Data Under The Relational Model

The application can execute statements like these to delete data:

Delete Purchase Order 1001

```
DELETE
FROM    LineItems_reltab
WHERE    PONo = 1001 ;

DELETE
FROM    PurchaseOrder_reltab
WHERE    PONo = 1001 ;
```

Limitations of a Purely Relational Model

The Relational Database Management System (RDBMS) is a very powerful and efficient form of information management. Why then should you even consider another approach? If you examine the application as developed under the relational model in comparison to the real world of the application domain, then certain shortcomings become evident.

Limitation in Encapsulating Data (Structure) with Operations (Behavior)

Database tables are excellent for modeling a structure of relationships, but they fail to capture the way that objects in the real world are naturally bundled with operations on the data. For example, when you operate on a purchase order in the real world, you expect to be able to sum the line items to find the total cost to the customer. Similarly, you expect that you should be able to retrieve information about the customer who placed the order — such as name, reference number, address, and so on. More complexly, you may want to determine the customer's buying history and payment pattern.

An RDBMS provides very sophisticated structures for storing and retrieving data, but each application developer must craft the operations needed for each application. This means that you must recode operations often, even though they may be very similar to operations already coded for applications within the same enterprise.

Limitation in Dealing with Composition

Relational tables do not capture compositions. For example, an address may be a composite of number, street, city, state, and zip code, but in a relational table, the notion of an address as a structure composed of the individual columns is not captured.

Limitation in Dealing with Aggregation

Relational tables have difficulty dealing with complex part-whole relationships. A piston and an engine have the same status as columns in the `Stock_reltab`, but there is no easy way to describe the fact that pistons are part of engines, except by creating multiple tables with primary key-foreign key relationships. Similarly, there is no easy way to implement the complex interrelationships between collections.

Limitation in Dealing with Generalization-Specialization

There is no easy way to capture the relationship of generalization-specification (inheritance). If we abstract the base requirements of a purchase order and write code to capture the relationships, then there is no way to develop purchase orders that use this code and then further specialize it for different domains. Instead, we will have duplicated the code in every implementation of a purchase order.

The Evolution of the Object-Relational Database System

So why not create applications using a third-generation language (3GL)?

First, an RDBMS provides functionality that would take millions of person-hours to replicate.

Second, one of the problems of information management using 3GLs is that they are not persistent; or, if they are persistent, then they sacrifice security to obtain the necessary performance by way of locating the application logic and the data logic in the same address space. Neither trade-off is acceptable to users of an RDBMS, for whom both persistence and security are basic requirements.

This leaves the application developer working under the relational model with the problem of simulating complex types by some form of mapping into SQL. Apart from the many person-hours required, this approach involves serious problems of implementation. You must:

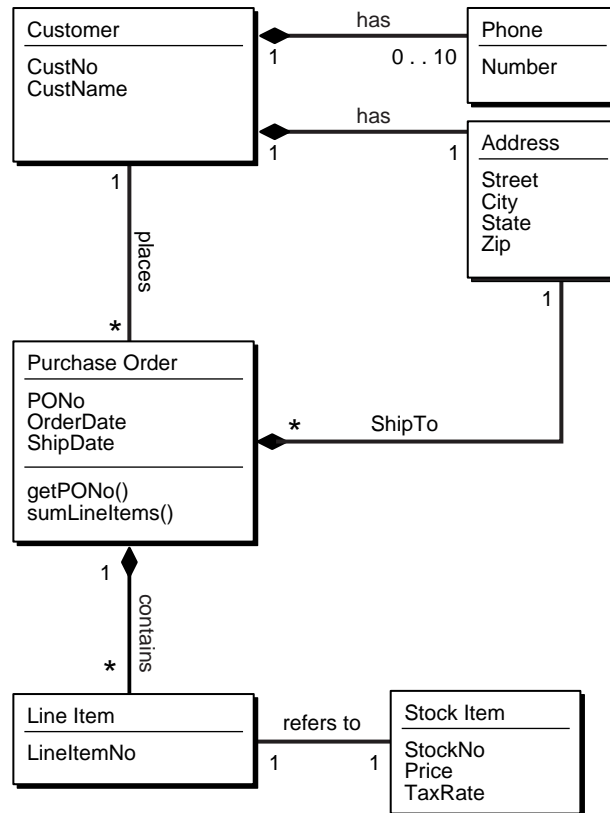
- Translate from application logic into data logic on 'write', and then
- Perform the reverse process on 'read' (and vice versa).

Obviously, there is heavy traffic back and forth between the client address space and that of the server, with the accompanying decrement in performance. And, if client and server are on different machines, then the toll on performance from network roundtrips may be considerable.

Object-relational (O-R) technology solves these problems. This chapter and the following chapter present examples that implement this new functionality.

Implementing the Application Under The Object-Relational Model

Figure 8–2 Class Diagram for Purchase Order Application



The object-relational (O-R) approach begins with the same entity relationships as in ["Entities and Relationships"](#) on page 8-5. Viewing these from the object-oriented perspective, as in the class diagram above, allows us to translate more of the real-world structure into the database schema.

Rather than breaking up addresses or multiple phone numbers into unrelated columns in relational tables, the O-R approach defines types to represent them. Rather than breaking line items out into a separate table, the O-R approach allows them to stay with their respective purchase orders as nested tables.

The main entities — customers, stock, and purchase orders — become objects. Object references express the relationships between them. Collection types model their multi-valued attributes.

There are two approaches to an object-relational implementation:

- Create and populate object tables.
- Use object views to represent virtual object tables from existing relational data.

The remainder of this chapter develops the O-R schema and shows how to implement it with object tables. [Chapter 4, "Applying an Object Model to Relational Data"](#) implements the same schema with object views.

Defining Types

The following statements set the stage by defining incomplete object types:

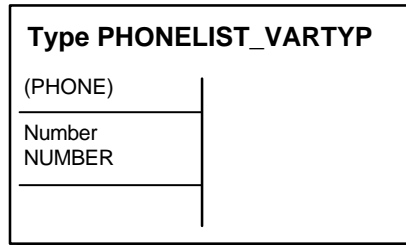
```
CREATE TYPE StockItem_objtyp;  
CREATE TYPE LineItem_objtyp;  
CREATE TYPE PurchaseOrder_objtyp;
```

The incomplete definitions notify Oracle that full definitions are coming later. Oracle can compile other types that refer to these incomplete types. Incomplete type declarations are like forward declarations in C and other programming languages.

The following statement defines an array type:

```
CREATE TYPE PhoneList_vartyp AS VARRAY(10) OF VARCHAR2(20);
```

Figure 8–3 Object Relational Representation of PhoneList_vartyp Type



The preceding statement defines the type `PhoneList_vartyp`. Any data unit of type `PhoneList_vartyp` is a varray of up to 10 telephone numbers, each represented by a data item of type `VARCHAR2`.

A list of phone numbers could occupy a varray or a nested table. In this case, the list is the set of contact phone numbers for a single customer. A varray is a better choice than a nested table for the following reasons:

- The order of the numbers might be important: varrays are ordered while nested tables are unordered.
- The number of phone numbers for a specific customer is small. Varrays force you to specify a maximum number of elements (10 in this case) in advance. They use storage more efficiently than nested tables, which have no special size limitations.
- There is no reason to query the phone number list, so the nested table format offers no benefit.

In general, if ordering and bounds are not important design considerations, then designers can use the following rule of thumb for deciding between varrays and nested tables: If you need to query the collection, then use nested tables; if you intend to retrieve the collection as a whole, then use varrays.

See Also: [Chapter 5, "Design Considerations for Oracle Objects"](#) for more information about the design considerations for varrays and nested tables.

The following statement defines the object type `Address_objtyp` to represent addresses:

```
CREATE TYPE Address_objtyp AS OBJECT (  
    Street      VARCHAR2(200),  
    City        VARCHAR2(200),  
    State       CHAR(2),  
    Zip         VARCHAR2(20)  
)  
/
```

Figure 8–4 Object Relational Representation of `Address_objtyp` Type

Type ADDRESS_OBJTYP			
STREET	CITY	STATE	ZIP
Text VARCHAR2(200)	Text VARCHAR2(200)	Text CHAR(2)	Number VARCHAR2(20)

All of the attributes of an address are character strings, representing the usual parts of a simplified mailing address.

The following statement defines the object type `Customer_objtyp`, which uses other user-defined types as building blocks.

```
CREATE TYPE Customer_objtyp AS OBJECT (  
    CustNo      NUMBER,  
    CustName    VARCHAR2(200),  
    Address_obj Address_objtyp,  
    PhoneList_var PhoneList_vartyp,  
  
    ORDER MEMBER FUNCTION  
        compareCustOrders(x IN Customer_objtyp) RETURN INTEGER  
)  
/
```

Instances of the type `Customer_objtyp` are objects that represent blocks of information about specific customers. The attributes of a `Customer_objtyp` object are a number, a character string, an `Address_objtyp` object, and a varray of type `PhoneList_vartyp`.

Every `Customer_objtyp` object also has an associated order method, one of the two types of comparison methods. Whenever Oracle needs to compare two `Customer_objtyp` objects, it invokes the `compareCustOrders` method to do so.

Note: The PL/SQL to implement the comparison method appears in "[The `compareCustOrders` Method](#)" on page 8-22.

The two types of comparison methods are map methods and order methods. This application uses one of each for purposes of illustration.

An `ORDER` method must be called for every two objects being compared, whereas a `MAP` method is called once per object. In general, when sorting a set of objects, the number of times an `ORDER` method is called is more than the number of times a `MAP` method would be called.

See Also: ■

- [Chapter 5, "Design Considerations for Oracle Objects"](#) for more information about design considerations for `ORDER` and `MAP` methods.
 - *PL/SQL User's Guide and Reference* for details about how to use pragma declarations.
-

The following statement completes the definition of the incomplete object type `LineItem_objtyp` declared at the beginning of this section.

```
CREATE TYPE LineItem_objtyp AS OBJECT (  
    LineItemNo    NUMBER,  
    Stock_ref     REF StockItem_objtyp,  
    Quantity      NUMBER,  
    Discount      NUMBER  
)  
/
```

Figure 8–5 Object Relational Representation of `LineItem_objtyp` Type

Type LINEITEM_OBJTYP			
LINEITEMNO	STOCK_REF	QUANTITY	DISCOUNT
Number NUMBER	Reference STOCKITEM_OBJTYP	Number NUMBER	Number NUMBER

Instances of type `LineItem_objtyp` are objects that represent line items. They have three numeric attributes and one `REF` attribute. The `LineItem_objtyp` models the line item entity and includes an object reference to the corresponding stock object.

The following statement defines the nested table type `LineItemList_ntabtyp`, which will represent an arbitrary set of line items inside a purchase order:

```
CREATE TYPE LineItemList_ntabtyp AS TABLE OF LineItem_objtyp  
/
```

A data unit of this type is a nested table, each row of which contains an object of type `LineItem_objtyp`. A nested table of line items is a better choice to represent the multivalued line item list than a varray of `LineItem_objtyp` objects, because:

- Most applications will need to query the contents of line items. This is an inefficient operation for varrays because their storage representation is not the same as the table representation.
- If an application needs to index on line item data, this can be done with nested tables but not with varrays.

- The order of line items is usually unimportant; the line item number can identify an order when necessary.
- There is no practical upper bound on the number of line items on a purchase order. Using a varray requires specifying an arbitrary upper bound on the number of elements.

The following statement completes the definition of the incomplete object type `PurchaseOrder_objtyp` declared at the beginning of this section:

```
CREATE TYPE PurchaseOrder_objtyp AUTHID CURRENT_USER AS OBJECT (  
    PONO                NUMBER,  
    Cust_ref            REF Customer_objtyp,  
    OrderDate           DATE,  
    ShipDate            DATE,  
    LineItemList_ntab   LineItemList_ntabtyp,  
    ShipToAddr_obj      Address_objtyp,  
  
    MAP MEMBER FUNCTION  
        getPONO RETURN NUMBER,  
  
    MEMBER FUNCTION  
        sumLineItems RETURN NUMBER  
);  
/
```

Figure 8–6 Object Relational Representation of the *PurchaseOrder_objtyp*

Type PURCHASEORDER_OBJTYP					
PONO	CUST_REF	ORDERDATE	SHIPDATE	LINEITEMLIST_NTAB	SHIPTOADDR_OBJ
Number NUMBER	Reference CUSTOMER_ OBJTYP	Date DATE	Date DATE	Nested Table LINEITEMLIST_ NTABTYP	Object Type ADDRESS_ OBJTYP
PK	FK				
<div>MEMBER FUNCTION getPONO RETURN NUMBER MEMBER FUNCTION SumLineItems RETURN NUMBER</div>					

The preceding statement defines the object type `PurchaseOrder_objtyp`. Instances of this type are objects representing purchase orders. They have six attributes, including a REF to `Customer_objtyp`, an `Address_objtyp` object, and a nested table of type `LineItemList_ntabtyp`, which is based on type `LineItem_objtyp`.

Objects of type `PurchaseOrder_objtyp` have two methods: `getPONo` and `sumLineItems`. One, `getPONo`, is a MAP method, one of the two kinds of comparison methods. A MAP method returns the relative position of a given record within the order of records within the object. So, whenever Oracle needs to compare two `PurchaseOrder_objtyp` objects, it implicitly calls the `getPONo` method to do so.

The two pragma declarations provide information to PL/SQL about what sort of access the two methods need to the database.

The statement does not include the actual PL/SQL programs implementing the methods `getPONo` and `sumLineItems`. That appears in "Method Definitions" on page 8-21.

The following statement completes the definition of `StockItem_objtyp`, the last of the three incomplete object types declared at the beginning of this section.

```
CREATE TYPE StockItem_objtyp AS OBJECT (  
    StockNo    NUMBER,  
    Price      NUMBER,  
    TaxRate    NUMBER  
)  
/
```

Figure 8–7 Object Relational Representation of the `StockItem_objtyp`

Type STOCKITEM_OBJTYP		
STOCKNO	PRICE	TAXRATE
Number NUMBER	Number NUMBER	Number NUMBER
PK		

Instances of type `StockItem_objtyp` are objects representing the stock items that customers order. They have three numeric attributes.

Method Definitions

This section shows how to specify the methods of the `PurchaseOrder_objtyp` and `Customer_objtyp` object types. The following statement defines the body of the `PurchaseOrder_objtyp` object type (the PL/SQL programs that implement its methods):

```
CREATE OR REPLACE TYPE BODY PurchaseOrder_objtyp AS

MAP MEMBER FUNCTION getPONo RETURN NUMBER is
BEGIN
    RETURN PONo;
END;

MEMBER FUNCTION sumLineItems RETURN NUMBER is
    i            INTEGER;
    StockVal     StockItem_objtyp;
    Total        NUMBER := 0;

BEGIN
    FOR i in 1..SELF.LineItemList_ntab.COUNT LOOP
        UTL_REF.SELECT_OBJECT(LineItemList_ntab(i).Stock_ref,StockVal);
        Total := Total + SELF.LineItemList_ntab(i).Quantity * StockVal.Price;
    END LOOP;
    RETURN Total;
END;
END;
/
```

The getPONo Method

The `getPONo` method is simple; use it to return the purchase order number of its associated `PurchaseOrder_objtyp` object. Such "get" methods allow you to avoid reworking code that uses the object if its internal representation changes.

The sumLineItems Method

The `sumLineItems` method uses a number of object-relational features:

- As already noted, the basic function of the `sumLineItems` method is to return the sum of the values of the line items of its associated `PurchaseOrder_objtyp` object. The keyword `SELF`, which is implicitly created as a parameter to every function, lets you refer to that object.
- The keyword `COUNT` gives the count of the number of elements in a PL/SQL table or array. Here, in combination with `LOOP`, the application iterates through

all the elements in the collection — in this case, the items of the purchase order. In this way `SELF.LineItemList_ntab.COUNT` counts the number of elements in the nested table that match the `LineItemList_ntab` attribute of the `PurchaseOrder_objtyp` object, here represented by `SELF`.

- A method from package `UTL_REF` is used in the implementation. The `UTL_REF` methods are necessary because Oracle does not support implicit dereferencing of refs within PL/SQL programs. The `UTL_REF` package provides methods that operate on object references. Here, the `SELECT_OBJECT` method is called to obtain the `StockItem_objtyp` object corresponding to the `Stock_ref`.
- The `AUTHID CURRENT_USER` syntax specifies that the `PurchaseOrder_objtyp` is defined using invoker-rights: the methods are executed under the rights of the current user, not under the rights of the user who defined the type.
- The PL/SQL variable `StockVal` is of type `StockItem_objtyp`. The `UTL_REF.SELECT_OBJECT` sets it to the object whose reference is the following:

```
(LineItemList_ntab(i).Stock_ref)
```

This object is the actual stock item referred to in the currently selected line item.

- Having retrieved the stock item in question, the next step is to compute its cost. The program refers to the stock item's cost as `StockVal.Price`, the `Price` attribute of the `StockItem_objtyp` object. But to compute the cost of the item, you also need to know the quantity of items ordered. In the application, the term `LineItemList_ntab(i).Quantity` represents the `Quantity` attribute of the currently selected `LineItem_objtyp` object.

The remainder of the method program is a loop that sums the extended values of the line items, and the method returns the total as its value.

The compareCustOrders Method

The following statement defines the `compareCustOrders` method of the `Customer_objtyp` object type.

```
CREATE OR REPLACE TYPE BODY Customer_objtyp AS
  ORDER MEMBER FUNCTION
  compareCustOrders (x IN Customer_objtyp) RETURN INTEGER IS
  BEGIN
    RETURN CustNo - x.CustNo;
  END;
END;
/
```


As mentioned earlier, the order method `compareCustOrders` operation compares information about two customer orders. It takes another `Customer_objtyp` object as an input argument and returns the difference of the two `CustNo` numbers. The return value is:

- a negative number, if its own object has a smaller value of `CustNo`
- a positive number, if its own object has a larger value of `CustNo`
- zero, if the two objects have the same value of `CustNo`—in which case it is referring to itself.

Whether the return value is positive, negative, or zero signifies the relative order of the customer numbers. For example, perhaps lower numbers are created earlier in time than higher numbers. If either of the input arguments (`SELF` and `explicit`) to an `ORDER` method is `NULL`, Oracle does not call the `ORDER` method and simply treats the result as `NULL`.

This completes the definition of the user-defined types used in the purchase order application. None of the declarations creates tables or reserves data storage space.

Creating Object Tables

To this point, the example is the same whether you plan to create and populate object tables or implement the application with object views on top of the relational tables that appear in ["Implementing the Application Under The Relational Model"](#) on page 8-4. The remainder of this chapter continues the example using object tables. [Chapter 4, "Applying an Object Model to Relational Data"](#), picks up from this point and continues the example with object views.

Generally, you can think of the relationship between the "objects" and "object tables" in the following way:

- Classes, which represent entities, map to object tables
- Attributes map to columns
- Objects map to rows

Viewed in this way, each object table is an implicit type whose objects (specific rows) each have the same attributes (column values). The creation of explicit user-defined datatypes and object tables introduces a new level of functionality.

The Object Table `Customer_objtab`

The following statement defines an object table `Customer_objtab` to hold objects of type `Customer_objtyp`:

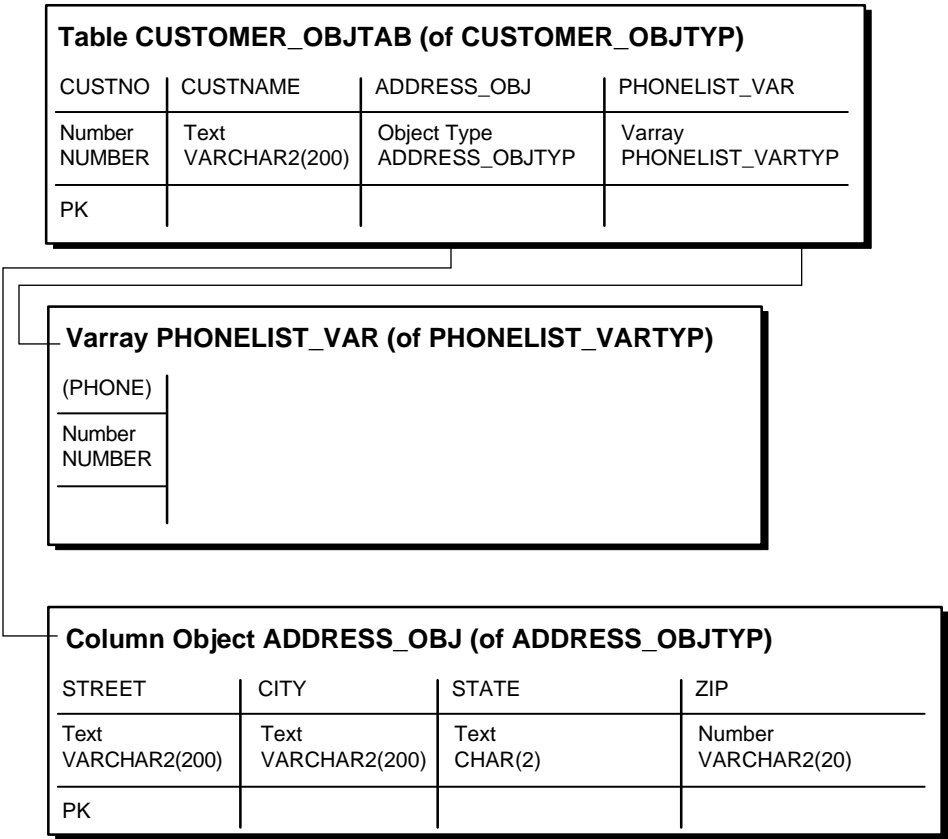
```
CREATE TABLE Customer_objtab OF Customer_objtyp (CustNo PRIMARY KEY)
  OBJECT ID PRIMARY KEY ;
```

As you can see, the term "OF" makes the create statement different for object tables as opposed to relational tables. We earlier defined the attributes of `Customer_objtyp` objects as:

<code>CustNo</code>	<code>NUMBER</code>
<code>CustName</code>	<code>VARCHAR2(200)</code>
<code>Address_obj</code>	<code>Address_objtyp</code>
<code>PhoneList_var</code>	<code>PhoneList_vartyp</code>

This means that the object table `Customer_objtab` has columns of `CustNo`, `CustName`, `Address_obj`, and `PhoneList_var`, and that each row is an object of type `Customer_objtyp`. As you will see, this notion of *row object* offers a significant advance in functionality.

Figure 8–8 Object Relational Representation of Table Customer_objtab



Object Datatypes as a Template for Object Tables

Because there is a type `Customer_objtyp`, you could create numerous object tables of the same type. For example, you could create an object table `Customer_objtab2` also of type `Customer_objtyp`. Without this ability, you would need to define each table individually.

You can introduce variations when creating multiple tables. The statement that created `Customer_objtab` defined a primary key constraint on the `CustNo` column. This constraint applies only to this object table. Another object table of the same type might not have this constraint.

Object Identifiers and References

`Customer_objtab` contains customer objects, represented as row objects. Oracle allows row objects to be referenceable, meaning that other row objects or relational rows may reference a row object using its object identifier (OID). For example, a purchase order row object may reference a customer row object using its object reference. The object reference is an opaque system-generated value represented by the type `REF` and is composed of the row object's unique OID.

Oracle requires every row object to have a unique OID. You may specify the unique OID value to be system-generated or specify the row object's primary key to serve as its unique OID. You indicate this when you execute the `CREATE TABLE` statement by specifying `OBJECT ID PRIMARY KEY` or `OBJECT ID SYSTEM GENERATED`, the latter serving as the default. The choice of primary key as the object identifier may be more efficient in cases where the primary key value is smaller than the default 16 byte system-generated identifier. For our example, the choice of primary key as the row object identifier has been made.

Object Tables with Embedded Objects

Examining the definition of `Customer_objtab`, you can see that the `Address_obj` column contains `Address_objtyp` objects. In other words, an object type may have attributes that are themselves object types. These embedded objects represent composite or structured values, and are also referred to as column objects. They differ from row objects because they are not referenceable and can be `NULL`.

`Address_objtyp` objects have attributes of built-in types, which means that they are leaf-level scalar attributes of `Customer_objtyp`. Oracle creates columns for `Address_objtyp` objects and their attributes in the object table `Customer_objtab`. You can refer to these columns using the dot notation. For example, if you want to build an index on the `Zip` column, then you can refer to it as `Address.Zip`.

The `PhoneList` column contains varrays of type `PhoneList_vartyp`. We defined each object of type `PhoneList_vartyp` as a varray of up to 10 telephone numbers, each represented by a data item of type `VARCHAR2`:

```
CREATE TYPE PhoneList_vartyp AS VARRAY(10) OF VARCHAR2(20)
/
```

Because each varray of type `PhoneList_vartyp` can contain no more than 200 characters (10 x 20), plus a small amount of overhead, Oracle stores the varray as a single data unit in the `PhoneList_var` column. Oracle stores varrays that exceed 4000 bytes in "inline" BLOBs, which means that a portion of the varray value could potentially be stored outside the table.

The Object Table `Stock_objtab`

The next statement creates an object table for `StockItem_objtyp` objects:

```
CREATE TABLE Stock_objtab OF StockItem_objtyp (StockNo PRIMARY KEY)
      OBJECT ID PRIMARY KEY ;
```

Each row of the table is a `StockItem_objtyp` object having three numeric attributes:

```
StockNo    NUMBER
Price      NUMBER
TaxRate    NUMBER
```

Oracle assigns a column for each attribute, and the `CREATE TABLE` statement places a primary key constraint on the `StockNo` column, and specifies that the primary key be used as the row object's identifier.

The Object Table `PurchaseOrder_objtab`

The next statement defines an object table for `PurchaseOrder_objtyp` objects:

```
CREATE TABLE PurchaseOrder_objtab OF PurchaseOrder_objtyp ( /* Line 1 */
      PRIMARY KEY (PONo), /* Line 2 */
      FOREIGN KEY (Cust_ref) REFERENCES Customer_objtab) /* Line 3 */
      OBJECT ID PRIMARY KEY /* Line 4 */
      NESTED TABLE LineItemList_ntab STORE AS PoLine_ntab ( /* Line 5 */
          (PRIMARY KEY(NESTED_TABLE_ID, LineItemNo)) /* Line 6 */
          ORGANIZATION INDEX COMPRESS) /* Line 7 */
      RETURN AS LOCATOR /* Line 8 */
/
```

The preceding `CREATE TABLE` statement creates the `PurchaseOrder_objtab` object table. The significance of each line is as follows:

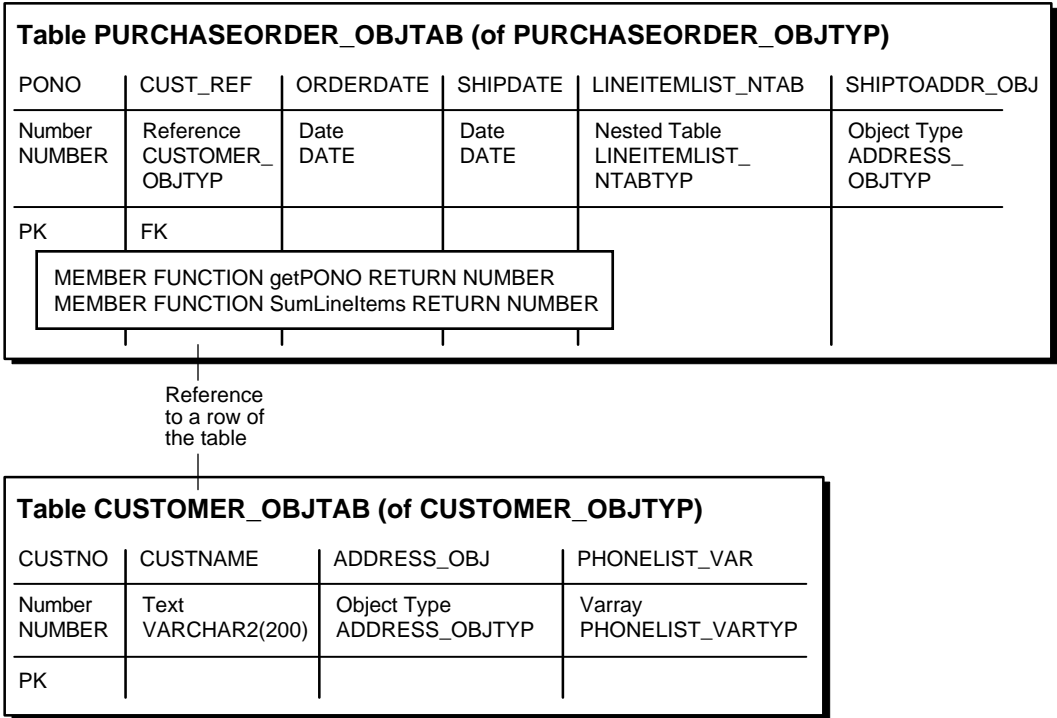
Line 1:

```
CREATE TABLE PurchaseOrder_objtab OF PurchaseOrder_objtyp (
```

This line indicates that each row of the table is a PurchaseOrder_objtyp object. Attributes of PurchaseOrder_objtyp objects are:

PONo	NUMBER
Cust_ref	REF Customer_objtyp
OrderDate	DATE
ShipDate	DATE
LineItemList_ntab	LineItemList_ntabtyp
ShipToAddr_obj	Address_objtyp

Figure 8–9 Object Relational Representation of Table PurchaseOrder_objtab



Line 2:

PRIMARY KEY (PONo),

This line specifies that the PONo attribute is the primary key for the table.

Line 3:

```
FOREIGN KEY (Cust_ref) REFERENCES Customer_objtab)
```

This line specifies a referential constraint on the `Cust_ref` column. This referential constraint is similar to those specified for relational tables. When there is no constraint, the `REF` column allows you to reference any row object. However, in this case, the `Cust_ref` REFS can refer only to row objects in the `Customer_objtab` object table.

Line 4:

```
OBJECT ID PRIMARY KEY
```

This line indicates that the primary key of the `PurchaseOrder_objtab` object table be used as the row's OID.

Line 5 - 8:

```
NESTED TABLE ListItemList_ntab STORE AS PoLine_ntab (
    (PRIMARY KEY(NESTED_TABLE_ID, ListItemNo))
    ORGANIZATION INDEX COMPRESS)
RETURN AS LOCATOR
```

These lines pertain to the storage specification and properties of the nested table column, `ListItemList_ntab`. The rows of a nested table are stored in a separate storage table. This storage table is not directly queryable by the user but can be referenced in DDL statements for maintenance purposes. A hidden column in the storage table, called the `NESTED_TABLE_ID`, matches the rows with their corresponding parent row. All the elements in the nested table belonging to a particular parent have the same `NESTED_TABLE_ID` value. For example, all the elements of the nested table of a given row of `PurchaseOrder_objtab` have the same value of `NESTED_TABLE_ID`. The nested table elements that belong to a different row of `PurchaseOrder_objtab` have a different value of `NESTED_TABLE_ID`.

In the `CREATE TABLE` example above, Line 5 indicates that the rows of `ListItemList_ntab` nested table are to be stored in a separate table (referred to as the storage table) named `PoLine_ntab`. The `STORE AS` clause also allows you to specify the constraint and storage specification for the storage table. In this example, Line 7 indicates that the storage table is an index-organized table (IOT). In general, storing nested table rows in an IOT is beneficial, because it provides clustering of rows belonging to the same parent. The specification of `COMPRESS` on the IOT saves storage space because, if you do not specify `COMPRESS`, the `NESTED_`

TABLE_ID part of the IOT's key is repeated for every row of a parent row object. If, however, you specify COMPRESS, the NESTED_TABLE_ID is stored only once for each row of a parent row object.

The SCOPE FOR constraint on a REF is not allowed in a CREATE TABLE statement. Therefore, to specify that Stock_ref can reference only the object table Stock_objtab, issue the following ALTER TABLE statement on the PoLine_ntab storage table:

```
ALTER TABLE PoLine_ntab
  ADD (SCOPE FOR (Stock_ref) IS stock_objtab) ;
```

Note that this statement is executed on the storage table, not the parent table.

See Also: ["Nested Table Storage"](#) on page 5-14 for information about the benefits of organizing a nested table as an IOT and specifying nested table compression, and for more information about nested table storage.

In Line 6, the specification of NESTED_TABLE_ID and LineItemNo attribute as the primary key for the storage table serves two purposes: first, it serves as the key for the IOT; second, it enforces uniqueness of a column (LineItemNo) of a nested table within each row of the parent table. By including the LineItemNo column in the key, the statement ensures that the LineItemNo column contains distinct values within each purchase order.

Line 8 indicates that the nested table, LineItemList_ntab, is returned in the locator form when retrieved. If you do not specify LOCATOR, the default is VALUE, which indicates that the entire nested table is returned instead of just a locator to the nested table. When the nested table collection contains many elements, it may not be very efficient to return the entire nested table whenever the containing row object or the column is selected.

Specifying that the nested table's locator is returned enables Oracle to send to the client only a locator to the actual collection value. An application can find whether a fetched nested table is in the locator or value form by calling the OCICollIsLocator or UTL_COLL.IS_LOCATOR interfaces. Once you know that the locator has been returned, the application can query using the locator to fetch only the desired subset of row elements in the nested table. This locator-based retrieval of the nested table rows is based on the original statement's snapshot, to preserve the value or copy semantics of the nested table. That is, when the locator is

used to fetch a subset of row elements in the nested table, the nested table snapshot reflects the nested table when the locator was first retrieved.

Recall the implementation of the `sumLineItems` method of `PurchaseOrder_objtyp` in "[Method Definitions](#)" on page 8-21. That implementation assumed that the `LineItemList_ntab` nested table would be returned as a `VALUE`. In order to handle large nested tables more efficiently, and to take advantage of the fact that the nested table in the `PurchaseOrder_objtab` is returned as a locator, the `sumLineItems` method must be rewritten as follows:

```
CREATE OR REPLACE TYPE BODY PurchaseOrder_objtyp AS

    MAP MEMBER FUNCTION getPONo RETURN NUMBER IS
    BEGIN
        RETURN PONo;
    END;

    MEMBER FUNCTION sumLineItems RETURN NUMBER IS
        i          INTEGER;
        StockVal    StockItem_objtyp;
        Total       NUMBER := 0;

    BEGIN
        IF (UTL_COLL.IS_LOCATOR(LineItemList_ntab)) -- check for locator
        THEN
            SELECT SUM(L.Quantity * L.Stock_ref.Price) INTO Total
            FROM   TABLE(CAST(LineItemList_ntab AS LineItemList_ntabtyp)) L;
        ELSE
            FOR i in 1..SELF.LineItemList_ntab.COUNT LOOP
                UTL_REF.SELECT_OBJECT(LineItemList_ntab(i).Stock_ref,StockVal);
                Total := Total + SELF.LineItemList_ntab(i).Quantity *
                                                                    StockVal.Price;
            END LOOP;
        END IF;
        RETURN Total;
    END;
END;
/
```

The rewritten `sumLineItems` method checks whether the nested table attribute, `LineItemList_ntab`, is returned as a locator using the `UTL_COLL.IS_LOCATOR` function. When the condition evaluates to `TRUE`, the nested table locator is queried using the `TABLE` expression.

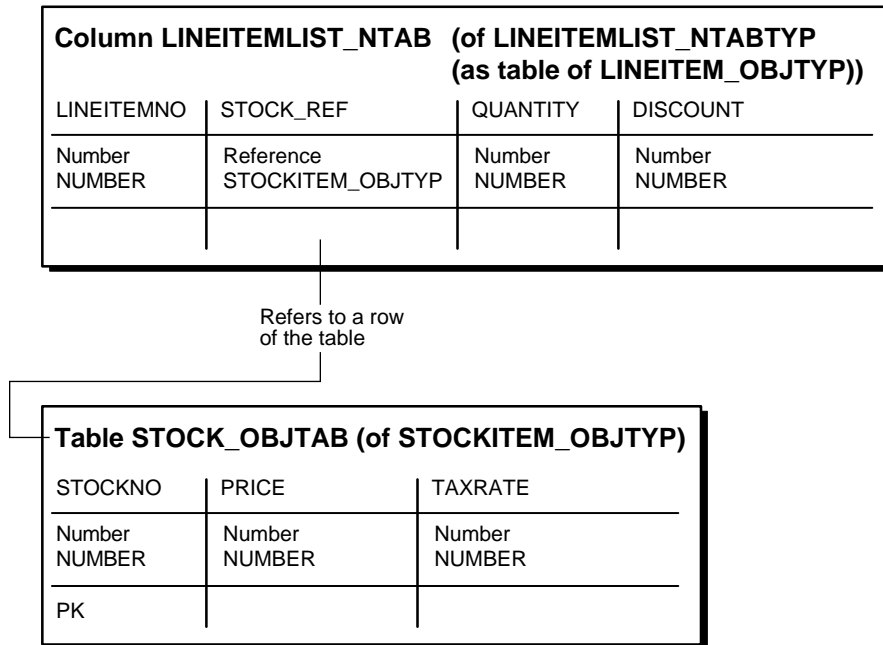
Note: The `CAST` expression is currently required in such `TABLE` expressions to communicate to the SQL compilation engine the actual type of the collection attribute (or parameter or variable) so that it can successfully compile the query.

The querying of the nested table locator results in a more efficient processing of the large line item list of a purchase order. The previous code that iterates over the `LineItemList_ntab` is kept to deal with the case where the nested table is returned as a `VALUE`.

After the table is created, the following `ALTER TABLE` statement is issued:

```
ALTER TABLE PoLine_ntab
  ADD (SCOPE FOR (Stock_ref) IS stock_objtab);
```

This statement specifies that the `Stock_ref` column of the nested table is scoped to `Stock_objtab`. This indicates that the values stored in this column must be references to row objects in `Stock_objtab`. The `SCOPE` constraint is different from the referential constraint, because the `SCOPE` constraint has no implication on the referenced object. For example, any referenced row object in `Stock_objtab` may be deleted, even if it is referenced in the `Stock_ref` column of the nested table. Such a deletion renders the corresponding reference in the nested table a `DANGLING REF`.

Figure 8–10 Object Relational Representation of Nested Table LineItemList_ntab

Oracle does not support referential constraint specification for storage tables. In this situation, specifying the `SCOPE` clause for a `REF` column is useful. In general, specifying scope or referential constraints for `REF` columns has a few benefits:

- It saves storage space because it allows Oracle to store just the row object's unique identifier as the `REF` value in the column.
- It enables an index to be created on the storage table's `REF` column.
- It allows Oracle to rewrite queries containing dereferences of these `REFs` as joins involving the referenced table.

At this point, all of the tables for the purchase order application are in place. The next section shows how to operate on these tables.

Figure 8–11 Object Relational Representation of Table `PurchaseOrder_objtab`

Table PURCHASEORDER_OBJTAB (of PURCHASEORDER_OBJTYP)					
PONO	CUST_REF	ORDERDATE	SHIPDATE	LINEITEMLIST_NTAB	SHIPTOADDR_OBJ
Number NUMBER	Reference CUSTOMER_ OBJTYP	Date DATE	Date DATE	Nested Table LINEITEMLIST_ NTABTYP	Object Type ADDRESS_ OBJTYP
PK	FK				
<div> <div>MEMBER FUNCTION getPONO RETURN NUMBER</div> <div>MEMBER FUNCTION SumLineItems RETURNNUMBER</div> </div>					

Column Object
of the defined type

Column Object SHIPTOADDR_OBJ (of ADDR_OBJTYP)			
STREET	CITY	STATE	ZIP
Text VARCHAR2(200)	Text VARCHAR2(200)	Text CHAR(2)	Number VARCHAR2(20)

Inserting Values

Here is how to insert the same data into the object tables as we did earlier for relational tables. Notice how some of the values are actually calls to the constructors for object types.

Stock_objtab

```
INSERT INTO Stock_objtab VALUES(1004, 6750.00, 2) ;
INSERT INTO Stock_objtab VALUES(1011, 4500.23, 2) ;
INSERT INTO Stock_objtab VALUES(1534, 2234.00, 2) ;
INSERT INTO Stock_objtab VALUES(1535, 3456.23, 2) ;
```

Customer_objtab

```
INSERT INTO Customer_objtab
VALUES (
    1, 'Jean Nance',
    Address_objtyp('2 Avocet Drive', 'Redwood Shores', 'CA', '95054'),
    PhoneList_vartyp('415-555-1212')
) ;

INSERT INTO Customer_objtab
VALUES (
    2, 'John Nike',
    Address_objtyp('323 College Drive', 'Edison', 'NJ', '08820'),
    PhoneList_vartyp('609-555-1212','201-555-1212')
) ;
```

PurchaseOrder_objtab

```
INSERT INTO PurchaseOrder_objtab
SELECT 1001, REF(C),
       SYSDATE, '10-MAY-1999',
       LineItemList_ntabtyp(),
       NULL
FROM   Customer_objtab C
WHERE  C.CustNo = 1 ;
```

The preceding statement constructs a PurchaseOrder_objtyp object with the following attributes:

PONo	1001
Cust_ref	REF to customer number 1
OrderDate	SYSDATE
ShipDate	10-MAY-1999
LineItemList_ntab	an empty LineItem_ntabtyp
ShipToAddr_obj	NULL

The statement uses a query to construct a REF to the row object in the Customer_objtab object table that has a CustNo value of 1.

The following statement uses a `TABLE` expression to identify the nested table as the target for the insertion, namely the nested table in the `LineItemList_ntab` column of the row object in the `PurchaseOrder_objtab` table that has a `PONo` value of 1001.

Note: Oracle release 8.0 supports the "flattened subquery" or "THE (subquery)" expression to identify the nested table. This construct is deprecated in release 8.1 in favor of the `TABLE` expression illustrated below.

```
INSERT INTO TABLE (  
  SELECT  P.LineItemList_ntab  
    FROM  PurchaseOrder_objtab P  
   WHERE  P.PONo = 1001  
)  
SELECT  01, REF(S), 12, 0  
  FROM  Stock_objtab S  
 WHERE  S.StockNo = 1534 ;
```

The preceding statement inserts a line item into the nested table identified by the `TABLE` expression. The inserted line item contains a `REF` to the row object with a `StockNo` value of 1534 in the object table `Stock_objtab`.

The following statements follow the same pattern as the previous ones:

```
INSERT INTO PurchaseOrder_objtab  
  SELECT  2001, REF(C),  
          SYSDATE, '20-MAY-1997',  
          LineItemList_ntabtyp(),  
          Address_objtyp('55 Madison Ave', 'Madison', 'WI', '53715')  
  FROM  Customer_objtab C  
 WHERE  C.CustNo = 2 ;  
  
INSERT INTO TABLE (  
  SELECT  P.LineItemList_ntab  
    FROM  PurchaseOrder_objtab P  
   WHERE  P.PONo = 1001  
)  
SELECT  02, REF(S), 10, 10  
  FROM  Stock_objtab S  
 WHERE  S.StockNo = 1535 ;
```

```

INSERT INTO TABLE (
  SELECT P.LineItemList_ntab
    FROM PurchaseOrder_objtab P
   WHERE P.PONo = 2001
)
SELECT 10, REF(S), 1, 0
  FROM Stock_objtab S
  WHERE S.StockNo = 1004 ;

INSERT INTO TABLE (
  SELECT P.LineItemList_ntab
    FROM PurchaseOrder_objtab P
   WHERE P.PONo = 2001
)
VALUES(11, (SELECT REF(S)
  FROM Stock_objtab S
  WHERE S.StockNo = 1011), 2, 1) ;

```

Querying

The following query statement implicitly invokes a comparison method. It shows how Oracle orders objects of type `PurchaseOrder_objtyp` using that type's comparison method:

```

SELECT p.PONo
  FROM PurchaseOrder_objtab p
 ORDER BY VALUE(p) ;

```

Oracle invokes the map method `getPONo` for each `PurchaseOrder_objtyp` object in the selection. Because that method returns the object's `PONo` attribute, the selection produces a list of purchase order numbers in ascending numerical order.

The following queries correspond to the queries executed under the relational model.

Customer and Line Item Data for Purchase Order 1001

```

SELECT DEREf(p.Cust_ref), p.ShipToAddr_obj, p.PONo,
       p.OrderDate, LineItemList_ntab
  FROM PurchaseOrder_objtab p
  WHERE p.PONo = 1001 ;

```

Total Value of Each Purchase Order

```
SELECT  p.PONo, p.sumLineItems()  
FROM    PurchaseOrder_objtab p ;
```

Purchase Order and Line Item Data Involving Stock Item 1004

```
SELECT  po.PONo, po.Cust_ref.CustNo,  
        CURSOR (  
            SELECT  *  
            FROM    TABLE (po.LineItemList_ntab) L  
            WHERE   L.Stock_ref.StockNo = 1004  
        )  
FROM    PurchaseOrder_objtab po ;
```

The above query returns a nested cursor for the set of `LineItem_obj` objects selected from the nested table. The application can fetch from the nested cursor to get the individual `LineItem_obj` objects. The above query can also be expressed by unnesting the nested set with respect to the outer result:

```
SELECT  po.PONo, po.Cust_ref.CustNo, L.*  
FROM    PurchaseOrder_objtab po, TABLE (po.LineItemList_ntab) L  
WHERE   L.Stock_ref.StockNo = 1004 ;
```

The above query returns the result set as a "flattened" form (or First Normal Form). This type of query is useful when accessing Oracle collection columns from relational tools and APIs, such as ODBC. In the above unnesting example, only the rows of the `PurchaseOrder_objtab` object table that has any `LineItemList_ntab` rows are returned. To fetch all rows of the `PurchaseOrder_objtab` table, regardless of the presence of any rows in their corresponding `LineItemList_ntab`, then the (+) operator is required:

```
SELECT  po.PONo, po.Cust_ref.CustNo, L.*  
FROM    PurchaseOrder_objtab po, TABLE (po.LineItemList_ntab) (+) L  
WHERE   L.Stock_ref.StockNo = 1004 ;
```

Average Discount across all Line Items of all Purchase Orders

This request requires querying the rows of all nested tables, `LineItemList_ntab`, of all `PurchaseOrder_objtab` rows. Again, unnesting is required:

```
SELECT  AVG(L.DISCOUNT)  
FROM    PurchaseOrder_objtab po, TABLE (po.LineItemList_ntab) L ;
```


Deleting

The following example has the same effect as the two deletions needed in the relational case (see ["Deleting Data Under The Relational Model"](#) on page 8-10). In this case, Oracle automatically deletes all line items belonging to the deleted purchase order. The relational case requires a separate step.

Delete Purchase Order 1001

```
DELETE
FROM   PurchaseOrder_objtab
WHERE  PONO = 1001 ;
```

Manipulating Objects Through Java

Using the schema that we have already defined for the purchase order example, we can manipulate objects within the database through the Java Database Connectivity (JDBC) API or by using embedded SQL with SQLJ. Although we use JDBC in this example, the coding for both is similar, and you can use either technique for object-oriented programs.

The first decision you do have to make is how closely you want to map the object types in the database to Java classes. The following sections show the two choices.

Using oracle.sql.* Classes (Weak Typing)

In this example:

- We map the data types and objects from the customer table to predefined object classes provided in the ORACLE.SQL package.
- We create only a single class with all the application logic, not one class for each object type.
- We treat the objects as the generic type `oracle.sql.STRUCT`, collection types as `oracle.sql.ARRAY`, and the scalar values as predefined types such as `oracle.sql.NUMBER`.
- We dynamically retrieve the attributes from the `STRUCT` class, pulling them into a single array. We must know the internal details of the class, such as that the first attribute is a number, and cast each element of the array into an object of the right type.

This technique lets us essentially write a procedural Java program that can easily interact with a particular class, as long as the definition of that class stays the same.

```
import java.sql.*;
import oracle.sql.*;

public class DefaultMappingDemo
{
    public static void main(String[] args)
    {
        System.out.println("*** JAVA OBJECTS DEMO ***");

        try {
            DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());

            Connection conn = DriverManager.getConnection
                ("jdbc:oracle:thin:@stpc90.us.oracle.com:1521:stpc90",
                 "scott", "tiger") ;

            Statement stmt = conn.createStatement();

            ResultSet rs = stmt.executeQuery
                ("select value(c) from CUSTOMER_TAB c order by value(c)");

            while (rs.next ())
            {

                // retrieve the STRUCT
                oracle.sql.STRUCT cust_struct = (STRUCT)rs.getObject(1);

                oracle.sql.Datum cust_attrs[] = cust_struct.getOracleAttributes();
                oracle.sql.NUMBER num = (NUMBER)cust_attrs[0];

                // string attribute in Object
                oracle.sql.CHAR name = (CHAR) cust_attrs[1];

                // embedded object
                oracle.sql.STRUCT address_struct = (STRUCT)cust_attrs[2];
                oracle.sql.Datum address_attrs[] = address_struct.getOracleAttributes();
                oracle.sql.CHAR street = (CHAR) address_attrs[0];
                oracle.sql.CHAR city   = (CHAR) address_attrs[1];
                oracle.sql.CHAR state  = (CHAR) address_attrs[2];
                oracle.sql.CHAR zip    = (CHAR) address_attrs[3];

                System.out.println("Number: " + num.stringValue() + ", Name: " + name +
```

```

        ", Address: " + street + ", " + city + ", " + state +
        ", " + zip);
    //embedded array
    oracle.sql.ARRAY phone_list = (ARRAY)cust_attrs[3];
    }
    rs.close();
    stmt.close();
}
catch (SQLException exn)
{
    System.out.println("SQLException: "+exn);
}
}
}
}

```

Using Strong Typing (SQLData or CustomDatum)

If you want to model the database object types using multiple Java classes, you can construct a strongly typed model. The classes all implement some common behavior to do the underlying database operations. Now, you have another choice: do you want to model the classes on the JDBC 2.0 API (the `SQLData` interface) or on Oracle's API (the `CustomDatum` interface)?

The `SQLData` interface is standards-based and potentially offers portability between different database systems. The `CustomDatum` interface is derived from JDBC, but offers additional enhancements; it can encapsulate REFs, collection types, and other object-oriented features not supported by JDBC.

You can generate wrapper classes for either interface by using JPublisher with different options.

Generating Wrapper Classes with JPublisher

In the strongly typed model, we need a Java class for each object type in the schema. The easiest way to get these classes is to let Oracle read the type definitions from the database and generate the Java code for us. To do this, we can use the following file as input to the JPublisher tool:

```

SQL SCOTT."ADDRESS_OBJTYP" AS JAddress
SQL SCOTT."CUSTOMER_OBJTYP" AS JCustomerInfo
SQL SCOTT."LINEITEMLIST_NTABTYP" AS JLineItemList

```

```
SQL SCOTT."LINEITEM_OBJTYP" AS JLineItem
SQL SCOTT."PHONELIST_VARTYP" AS JPhoneList
SQL SCOTT."PURCHASEORDER_OBJTYP" AS JPurchaseOrder
SQL SCOTT."STOCKITEM_OBJTYP" AS JStockInfo
```

How to Use the Wrapper Classes

The wrapper classes all look much like the one below, JCustomer which corresponds to the CUSTOMER_INFO_T type in the database schema. For our example, we would also need the JAddress wrapper class because one of the attributes of JCustomer is a JAddress object.

You can read or write instances of this type using regular Java I/O streams. To implement additional member functions, you can subclass JCustomer, so that your code is preserved whenever that class is regenerated.

```
import java.sql.*;
import oracle.jdbc2.*;
import oracle.sql.*;

public class JCustomer implements SQLData
{
    private String sql_type;
    public int custNo;
    public String custName;
    public JAddress address;
    public Array phoneList;

    public String getSQLTypeName() throws SQLException { return sql_type; }

    public void readSQL (SQLInput stream, String typeName) throws SQLException
    {
        sql_type = typeName;
        custNo   = stream.readInt();
        custName = stream.readString();
        address  = (JAddress) stream.readObject();
        phoneList= stream.readArray();
    }
    public void writeSQL (SQLOutput stream) throws SQLException
    {
        stream.writeInt(custNo);
        stream.writeString(custName);
        stream.writeObject(address);
    }
}
```

```

        stream.writeArray(phoneList);
    }
}

```

In this example, we do not show member functions being derived from the method functions of the database type. Calling such member functions causes traffic as object data is passed back and forth to the database server, and you must follow certain conventions for input and output parameters. For information on this subject, see *Oracle8i SQLJ Developer's Guide and Reference* (Objects and Collections) and *Oracle8i JDBC Developer's Guide and Reference* (Working with Oracle Object Types).

Sample Program Using the SQLData Interface

In the following program:

- We work with the JCustomer and JAddress classes that are produced by JPublisher. JAddress is needed because it is the type for one of JCustomer's attributes.
- We let the database know which Java classes correspond to which SQL object types. For example, JCustomer corresponds to CUSTOMER_INFO_T. That information allows Oracle to substitute object data into SQL statements such as the INSERT in the example.
- Once we cast an object from the result set to JCustomer, we can access its data and functions as with any other Java class.
- We update the object in Java, then substitute the Java object into an SQL statement that updates the database.

```

import java.sql.*;
import oracle.sql.*;
import oracle.jdbc.driver.*;
import oracle.jdbc2.*;
import java.util.*;

public class SQLDataDemo
{
    public static void main(String[] args) throws Exception, SQLException
    {
        System.out.println("*** JAVA OBJECTS DEMO : USING SQLData INTERFACE ***");

        DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
    }
}

```

```
        OracleConnection conn = (OracleConnection) DriverManager.getConnection
            ("jdbc:oracle:thin:@stpc90.us.oracle.com:1521:stpc90",
            "scott", "tiger");

        Statement stmt = conn.createStatement();

        //put an entry in the typemap
        try
        {
            Dictionary map = conn.getTypeMap();

            map.put("CUSTOMER_INFO_T", Class.forName("JCustomer"));
            map.put("ADDRESS_T", Class.forName("JAddress"));
        }
        catch (ClassNotFoundException exn) { }

        ResultSet rs = stmt.executeQuery("select VALUE (p) from CUSTOMER_TAB p");

        while (rs.next())
        {
            //retrieve the object using standard API
            JCustomer jc = (JCustomer) rs.getObject(1);
            int custNo = jc.custNo;
            String custName = jc.custName;

            jc.custName = "Geoff Lee";
            PreparedStatement pstmt = conn.prepareStatement
                ("INSERT INTO CUSTOMER_TAB VALUES (?)");
            pstmt.setObject(1, jc);
            pstmt.executeUpdate();

            rs.close();
            stmt.close();
        }
    }
}
```

Manipulating Objects with Oracle Objects for OLE

On Windows systems, you can use Oracle Objects for OLE (OO4O) to write object-oriented database programs in Visual Basic or other environments that support the COM protocol, such as Excel.

The following examples all begin with a similar header section that connects to the database, then each shows how to perform a different operation on object data.

Selecting Data

Here is an event handler for a button that performs a SELECT operation.

- We get a set of rows from the database, each row containing some relational columns and some columns that are objects.
- Using the name of the CUSTREF column, we retrieve its value, which is an object.
- Then we can use the dot notation to access the attributes of the object. We define the variable as a generic object type, OraObject. After it is instantiated with a real object, it takes on the properties of the corresponding object type.

```
Private Sub obj_select_Click()
    Dim OO4OSession As OraSession
    Dim InvDB As OraDatabase
    Dim PurchaseOrder As OraDynaset
    Dim CustomerInfo As OraRef
    Dim LineItemsList As OraCollection
    Dim LineItem As OraObject
    Dim ShipToAddr As OraObject
    Dim StockInfo As OraRef
    Dim CustomerAddr As OraObject

    'Create the OraSession Object.
    Set OO4OSession = CreateObject("OracleInProcServer.XOraSession")

    'Create the OraDatabase Object by opening a connection to Oracle.
    Set InvDB = OO4OSession.OpenDatabase("exampledb", "scott/tiger", 0&)

    'Select from purchase_tab
    Set PurchaseOrder = InvDB.CreateDynaset("select * from purchase_tab", 0&)

    'Get the custref attribute from PurchaseOrder
    Set CustomerInfo = PurchaseOrder.Fields("custref").Value
```

```
' Accessing attributes CustomerInfo object

'Display custno,custname,phonelist attributes of CustomerInfo
MsgBox CustomerInfo.custno
MsgBox CustomerInfo.custname

'Get address and phonelist attributes of CustomerInfo
Set CustomerAddr = CustomerInfo.Address

'Display all the attributes of CustomerAddr
MsgBox CustomerAddr.Street
MsgBox CustomerAddr.State
MsgBox CustomerAddr.Zip

' Accessing elements of LineItemsList Object

'Get line_item_list attribute from PurchaseOrder
Set LineItemsList = PurchaseOrder.Fields("line_item_list").Value

'Get LineItem object element from LineItemList collection
Set LineItem = LineItemsList(1)

'Display lineitemno,quantity,discount attributes
MsgBox LineItem.lineitemno
MsgBox LineItem.quantity
MsgBox LineItem.discount

'Access stockref attribute of LineItem
Set StockInfo = LineItem.Stockref

'Display stockno,cost,tax_code of StockInfo
MsgBox StockInfo.stockno
MsgBox StockInfo.cost
MsgBox StockInfo.tax_code

End Sub
```

Inserting Data

Here is a program that retrieves a set of rows from the database, then adds a new row.

- We create some objects of the appropriate object types.

- We populate the objects with sample values.
- We create a new row for the purchase order table, and fill in the values for its columns. The columns that are not objects can be set directly. The columns that are objects must be set using the VALUE field.

```

Dim OO4OSession As OraSession
Dim InvDB As OraDatabase
Dim PurchaseOrder As OraDynaset
Dim CustomerInfo As OraRef
Dim LineItemsList As OraCollection
Dim LineItem As OraObject
Dim ShipToAddr As OraObject
Dim StockInfo As OraRef
Dim CustomerAddr As OraObject

'Create the OraSession Object.
Set OO4OSession = CreateObject("OracleInProcServer.XOraSession")

'Create the OraDatabase Object by opening a connection to Oracle.
Set InvDB = OO4OSession.OpenDatabase("exampledb", "scott/tiger", 0&)

'Select from purchase_tab
Set PurchaseOrder = InvDB.CreateDynaset("select * from purchase_tab", 0&)

' Step 1 - Creating CustomerInfo ref object

'select a ref from customer_tab for custono 2
Set CustomerDyn = InvDB.CreateDynaset("select REF(C) from customer_tab c
where c.custno = 2", 0&)

'get the CustomerInfo ref object
Set CustomerInfo = CustomerDyn.Fields(0).Value

' Step 2 - Creating LineItemsList object

' Create a new line_items_list object
Set LineItemsList = InvDB.CreateOraObject("line_item_list_t")

' Create a new line_items object
Set LineItem = InvDB.CreateOraObject("line_item_t")

'set attributes of LineItem object
LineItem.lineitemno = 2
LineItem.quantity = 15

```

```
LineItem.discount = 30
LineItem.Stockref = Null

'set the LineItem to first element of LineItemList
LineItemsList(1) = LineItem

' Step 3 - Creating ShipToAddr object

' create a shiptoaddr object
Set ShipToAddr = InvDB.CreateOraObject("address_t")

'set the attributes of ShipToAddr Object
ShipToAddr.city = "Belmont"
ShipToAddr.Street = "Continental way"
ShipToAddr.Zip = "94002"
ShipToAddr.State = "CA"

' Start the AddNew operation on PurchaseOrder dynaset

PurchaseOrder.AddNew

PurchaseOrder.Fields("pono").Value = 1002
PurchaseOrder.Fields("orderdate").Value = "5/15/99"
PurchaseOrder.Fields("shipdate").Value = "6/15/99"

'set the custref field to CustomerInfo object created in step1
PurchaseOrder.Fields("custref").Value = CustomerInfo

'set the line_item_list field to LineItemslist object created in step2
PurchaseOrder.Fields("line_item_list").Value = LineItemsList

'set the shiptoaddr field to ShipToAddr object created in step3
PurchaseOrder.Fields("shiptoaddr").Value = ShipToAddr

' Call the update method on Purchaseorder Dynaset which inserts a new row
' in purchase_tab table

PurchaseOrder.Update
```

Updating Data

Here is a program that retrieves some rows from the database, then updates a specific one.

- We select the purchase order using a query that returns a single row.
- We get individual data items to manipulate from other tables and from the original purchase order.
- We lock the purchase order row for updating, and put in the new values.

```

Dim OO4OSession As OraSession
Dim InvDB As OraDatabase
Dim PurchaseOrder As OraDynaset
Dim CustomerInfo As OraRef
Dim LineItemList As OraCollection
Dim LineItem As OraObject
Dim ShipToAddr As OraObject
Dim StockInfo As OraRef
Dim CustomerAddr As OraObject

'Create the OraSession Object.
Set OO4OSession = CreateObject("OracleInProcServer.XOraSession")

'Create the OraDatabase Object by opening a connection to Oracle.
Set InvDB = OO4OSession.OpenDatabase("exampledb", "scott/tiger", 0&)

'Select from purchase_tab for pono 1002
Set PurchaseOrder = InvDB.CreateDynaset("select * from purchase_tab where
pono = 1002", 0&)

'Create a StockInfo from stock_tab for stockno 1535
Set StockDyn = InvDB.CreateDynaset("select REF(s) from stock_tab s where
s.stockno = 1535", 0&)
Set StockInfo = StockDyn.Fields(0).Value

'Get line_item_list attribute from PurchaseOrder
Set LineItemList = PurchaseOrder.Fields("line_item_list").Value

'Get LineItem object element from LineItemList collection
Set LineItem = LineItemList(1)

'Start the edit operation on PurchaseOrder dynaset
PurchaseOrder.Edit

' Set the StockInfo object created in Step1 to stockref attribute
' of LineItem
LineItem.Stockref = StockInfo
PurchaseOrder.Update

```

Calling a Method Function

Here is a program that retrieves a purchase order, and calls its member function `TOTAL_VALUE` to sum the cost of the line items that are part of the purchase order.

- We select one row from the purchase order table. Notice we select the `VALUE` so that the result comes back as an object.
- We get a pointer to the purchase order object (the zero'th column of the result row). Later this pointer is passed to a PL/SQL stored procedure, to simulate the `SELF` pointer in Java or C++ methods.
- We build a list of parameters corresponding to the implicit self parameter and the return value of the method function. For each, we specify the bind variable, its value, its mode, and its type.
- We call the stored procedure corresponding to the method function, storing the result in the `TOTALVALUE` bind variable.
- To use the result, we retrieve the return value from the parameter list.

```
Dim OO4OSession As OraSession
Dim InvDB As OraDatabase
Dim PurchaseOrderObj As OraDynaset

'Create the OraSession Object.
Set OO4OSession = CreateObject("OracleInProcServer.XOraSession")

'Create the OraDatabase Object by opening a connection to Oracle.
Set InvDB = OO4OSession.OpenDatabase("exampdb", "scott/tiger", 0&)

'Select from purchase_tab
Set PurchaseOrderDyn = InvDB.CreateDynaset("select VALUE(p) from
purchase_tab p where p.pono = 1001", 0&)

'Get the PurchaseOrderObj
Set PurchaseOrderObj = PurchaseOrderDyn.Fields(0).Value

'Create a OraParameter object for purchase_order_t object and set it to
PurchaseOrder
InvDB.Parameters.Add "PURCHASEORDER", PurchaseOrderObj, ORAPARM_BOTH,
ORATYPE_OBJECT, "PURCHASE_ORDER_T"

'Create a parameter for total_value return
InvDB.Parameters.Add "TOTALVALUE", "", ORAPARM_OUTPUT

'Execute a member method
```

```
InvDB.ExecutesQL ("BEGIN :TOTALVALUE :=  
PURCHASE_ORDER_T.TOTAL_VALUE(:PURCHASEORDER); END;")
```

```
'Display the totalvalue  
MsgBox InvDB.Parameters("TOTALVALUE").Value
```

Index

A

Active Server Pages, 3-9
ActiveX, 3-9
ADMIN OPTION
 with EXECUTE ANY TYPE, 2-11
ALTER ANY TYPE privilege, 2-10
 See also privileges
arrays, 8-26
 size of VARRAYs, 1-8
 variable (VARRAYs), 1-8
ASP, 3-9
atomic nulls, 2-2
attributes
 leaf-level, 6-1
 leaf-level scalar, 6-1
 of object types, 1-3

B

bind variables
 user-defined types, 3-2

C

caches
 object cache, 2-13, 3-2, 3-6
 object views, 4-4
capture avoidance rule, 2-7
collections
 nested tables, 1-9
 nesting, 5-20
 querying, 5-12
 variable arrays (VARRAYs), 1-8

column objects
 vs. row objects, 5-1
columns
 column names
 qualifying in queries, 2-7
 column objects, 1-6
 indexes, 2-4
 qualifying in queries, 2-6
 comparison methods, 1-4, 8-20
 compilation of object types, 2-14
 complex object retrieval
 for Oracle Call Interface, 6-9
COMPRESS clause
 nested tables, 5-16
CONNECT role
 user-defined types, 2-11
constraints, 8-25
 object tables, 2-3
 on Oracle objects, 5-37
 REFs, 5-9
 SCOPE FOR constraint, 8-30, 8-32
constructor methods, 1-4, 6-2
 literal invocation of, 2-3
COUNT attribute of collection types, 8-21
CREATE ANY TYPE privilege, 2-10
 See also privileges
CREATE INDEX statement
 object types, 2-5
CREATE TABLE statement
 examples
 column objects, 1-11, 2-7
 nested tables, 1-10
 object tables, 1-5, 1-10, 2-4, 2-7
CREATE TRIGGER statement

- examples
 - object tables, 2-5
- CREATE TYPE command
 - nested tables, 1-9, 1-11
- CREATE TYPE privilege, 2-10
 - See also* privileges
- CREATE TYPE statement
 - incomplete types, 2-14
 - nested tables, 2-3
 - object types, 1-10, 2-2, 2-3, 2-7, 8-14
 - object views, 4-3
 - varrays, 1-9, 8-15
- CREATE VIEW statement
 - examples
 - object views, 4-4

D

- dangling REFs, 1-7
- database administrators (DBAs)
 - DBA role, 2-11
- datatypes
 - array types, 1-8
 - nested tables, 1-9
 - object types, 1-2
- DBA role
 - user-defined types, 2-11
- default values
 - user-defined types, 2-3
- DELETE privilege for object tables, 2-12, 2-13
- dependencies
 - object type definitions, 2-14, 2-15
- dereferencing, 1-7, 8-22
 - implicit, 1-7, 8-22
- dot notation, 1-3
- DROP ANY TYPE privilege, 2-11
 - See also* privileges
- DROP TYPE statement
 - FORCE option, 2-15
- dump files
 - Export and Import, 2-16

E

- Excel, 3-9

- EXECUTE ANY TYPE privilege, 2-11
 - See also* privileges
- EXECUTE privilege
 - user-defined types, 2-11
 - See also* privileges
- EXECUTE user-defined type, 2-11
- Export utility
 - user-defined types, 2-16

F

- FAQ
 - for Oracle objects, 7-1
- files
 - Export and Import dump file, 2-16
- FORCE option
 - object type dependencies, 2-15
- foreign keys
 - representing many-to-one entity relationship with, 8-7
- frequently asked questions
 - about Oracle objects, 7-1
- function-based indexes
 - returning values of type methods, 5-30

G

- GRANT option for EXECUTE privilege, 2-11
- granting
 - execute user-defined type, 2-11

I

- implicit dereferencing, 1-7, 8-22
- Import utility
 - user-defined types, 2-16
- incomplete object types, 2-14, 8-14
- indexes
 - on REFs, 2-4
 - user-defined types, 2-4
- index-organized tables
 - storing nested tables as, 5-15
- inheritance, 1-10
 - dual subtype/super-type reference, 5-36
 - subtype contains super-type, 5-33

- super-type contains all subtypes, 5-35
- inner capture, 2-7
- INSERT privilege for object tables, 2-12, 2-13
- INSTEAD OF triggers
 - nested tables, 4-11
- invoker-rights
 - object types, 5-29

J

Java

- Oracle JDBC and Oracle objects, 3-12
- Oracle SQLJ and Oracle objects, 3-12
- with Oracle objects, 3-12

JDBC

- See Oracle JDBC

K

keys

- foreign keys, 8-7

L

- leaf-level attributes, 6-1
- leaf-level scalar attributes, 6-1
- literal invocation
 - constructor methods, 2-3
- locators, 8-30
 - returning nested tables as, 5-18
- locks
 - object level locking, 3-3

M

- map methods, 1-5, 5-7, 8-17
- methods, 1-3
 - choosing a language for, 5-26
 - comparison, 8-20
 - comparison methods, 1-4
 - constructor methods, 1-4
 - literal invocation, 2-3
 - function-based indexes, 5-30
 - map, 5-7, 8-17
 - of object types, 1-3, 8-21

- constructor methods, 6-2
- execution privilege for, 2-11
- map methods, 1-5
- order methods, 1-5
- PL/SQL, 3-2
 - selfish style of invocation, 1-4
 - use of empty parentheses with, 2-8
- order, 5-7, 8-17, 8-23
- static, 5-28

N

- nested tables, 1-9, 5-14
 - COMPRESS clause, 5-16
 - creating indexes on, 5-17
 - DML operations on, 5-19
 - in an index-organized table, 5-15
 - indexes, 2-4
 - INSTEAD OF triggers, 4-11
 - querying, 8-18
 - returning as locators, 5-18, 8-30
 - storage, 5-14, 8-29
 - uniqueness in, 8-30
 - updating in views, 4-11
 - vs VARRAY, 8-18
 - vs varrays, 8-15
- NESTED_TABLE_ID, 5-17, 8-29
- nulls
 - atomic, 2-2
 - object types, 2-2

O

- object cache
 - flushing an object, 6-9
 - object views, 4-4
 - OCI, 3-2
 - privileges, 2-13
 - Pro*C, 3-6
- object identifiers, 8-26
 - for object types, 6-2
 - primary-key based, 5-7
 - REFs, 5-8
 - storage, 5-7
 - WITH OBJECT IDENTIFIER clause, 4-4

- object tables, 1-5, 5-7, 8-23
 - constraints, 2-3
 - deleting values, 8-39
 - indexes, 2-4
 - inserting values, 8-34
 - querying, 8-37
 - row objects, 1-6
 - triggers, 2-5
 - virtual object tables, 4-2
- object types, 1-2
 - attributes of, 1-3
 - column objects, 1-6
 - indexes, 2-4
 - column objects vs. row objects, 5-1
 - comparison methods for, 1-4, 8-20
 - constructor methods for, 1-4, 6-2
 - incomplete, 2-14, 8-14
 - invoker-rights, 5-29
 - locking in cache, 3-3
 - methods of, 1-3, 8-21
 - method calls, 2-8
 - PL/SQL, 3-2
 - mutually dependent, 2-14
 - Oracle type translator, 3-8
 - purchase order example, 1-10
 - row objects, 1-6
 - use of table aliases, 2-7
- object views, 4-1 to 4-18
 - advantages of, 4-2
 - defining, 4-3
 - nested tables, 4-11
 - updating through INSTEAD OF triggers, 4-11
- object-relational model, 8-1
 - comparing objects, 5-7
 - constraints, 5-37
 - design considerations, 5-1
 - embedded objects, 8-26
 - implementing with object tables, 8-14
 - inheritance, 1-10
 - limitations of relational model, 8-11
 - methods, 1-3
 - new object format, 5-31
 - partitioning, 6-14
 - programmatically environments for, 3-1 to 3-12
 - replication, 5-31
 - type evolution, 5-38
- objects
 - collection objects, 4-6
 - in columns, 4-4
 - object references, 4-9
 - row objects and object identifiers, 4-6
- OCI
 - associative access, 3-3
 - complex object retrieval (COR), 6-9
 - creating a new object, 6-4
 - deleting an object, 6-5
 - for Oracle objects
 - building a program, 3-5
 - initializing object manipulation, 6-4
 - lock options, 6-8
 - navigational access, 3-4
 - object cache, 3-4, 6-11
 - flushing an object, 6-9
 - OCIObjectFlush, 4-4
 - OCIObjectPin, 4-4
 - pinning and unpinning objects, 6-6
 - updating an object, 6-5
- OIDs
 - See object identifiers
- Oracle Call Interface
 - controlling object cache size, 6-5
- Oracle JDBC
 - accessing Oracle object data, 3-12
- Oracle objects
 - See object-relational model
- Oracle Objects for OLE
 - OraCollection interface, 3-11
 - OraObject interface, 3-10
 - OraRef interface, 3-10
- Oracle SQLJ
 - creating custom Java classes, 3-13
 - JPublisher, 3-13
 - support for Oracle objects, 3-12
- Oracle type translator (OTT), 3-8
- OraCollection interface, 3-11
- OraObject interface, 3-10
- OraRef interface, 3-10
- order methods, 1-5, 5-7, 8-17, 8-23
- OTT, 3-8

P

- parallel query
 - restrictions for Oracle objects, 5-38
- parentheses, use of in method calls, 2-8
- partitioning
 - tables containing Oracle objects, 6-14
- pkREFs, 6-2
- PL/SQL
 - bind variables
 - user-defined types, 3-2
 - object views, 4-4
 - user-defined datatypes, 3-2
- pragma RESTRICT REFERENCES, 8-20
- primary-key-based REFs, 6-2
- privileges
 - system
 - user-defined types, 2-10
 - user-defined types
 - acquired by role, 2-11
 - ALTER ANY TYPE, 2-10
 - checked when pinning, 2-13
 - column level for object tables, 2-14
 - CREATE ANY TYPE, 2-10
 - CREATE TYPE, 2-10
 - DELETE, 2-12, 2-13
 - DROP ANY TYPE, 2-11
 - EXECUTE, 2-11
 - EXECUTE ANY TYPE, 2-11
 - EXECUTE ANY TYPE with ADMIN OPTION, 2-11
 - EXECUTE with GRANT option, 2-11
 - INSERT, 2-12, 2-13
 - SELECT, 2-12, 2-13
 - system privileges, 2-10
 - UPDATE, 2-12, 2-13
 - using, 2-11, 2-15
- Pro*C/C++
 - associative access, 3-6
 - converting between Oracle and C types, 3-7
 - navigational access, 3-6
 - user-defined datatypes, 3-2
- programmatic environments
 - for Oracle objects, 3-1 to 3-12

Q

- queries
 - set membership, 5-18
 - unnesting, 5-12
 - varrays, 5-14

R

- REFs, 1-6
 - constraints on, 5-9
 - constructing from object identifiers, 6-2
 - dangling, 1-7
 - dereferencing of, 1-7, 8-22
 - for rows of object views, 4-3
 - implicit dereferencing of, 1-7, 8-22
 - indexes on, 2-4
 - indexing, 5-10
 - mutually dependent types, 2-14
 - object identifiers, 8-26
 - pinning, 2-13, 4-4
 - scoped, 1-7, 5-9, 6-2
 - size of, 6-2
 - storage, 5-9
 - use of table aliases, 2-7
 - WITH ROWID option, 5-11
- RESOURCE role
 - user-defined types, 2-11
- returning nested tables as, 8-30
- REVOKE command
 - object types and dependencies, 2-15
- REVOKE statement
 - FORCE option, 2-15
- roles
 - CONNECT role, 2-11
 - DBA role, 2-11
 - RESOURCE role, 2-11
- row objects, 1-6
 - storage, 5-7
- rows
 - row objects, 1-6

S

- schema names
 - qualifying column names, 2-7

schemas

- user-defined datatypes, 3-2

- user-defined types, 1-2

SCOPE FOR constraint, 8-30, 8-32

scoped REFs, 1-7, 6-2

SELECT privilege for object tables, 2-12, 2-13

selfish style of method invocation, 1-4

SQL

- user-defined datatypes, 3-1

- embedded SQL, 3-6

- OCI, 3-2

SQLJ

- See* Oracle SQLJ

storage

- nested tables, 6-3

- object tables, 6-1

- REFs, 6-2

STORE AS clause, 8-29

system privileges

- ADMIN OPTION, 2-11

- user-defined types, 2-10

- See also* privileges

T

TABLE syntax, 5-12

tables

- nested tables, 1-9

- indexes, 2-4

- object

- See* object tables

- object tables, 1-5

- constraints, 2-3

- indexes, 2-4

- triggers, 2-5

- virtual, 4-2

- qualifying column names, 2-6, 2-7

triggers

- INSTEAD OF triggers

- object views and, 4-11

- user-defined types, 2-5

type evolution, 5-38

types

- See* datatypes, object types

U

unnesting queries, 5-12

UPDATE privilege for object tables, 2-12, 2-13

updates

- object views, 4-11

user-defined datatypes, 2-1 to 2-16

- collections

- nested tables, 1-9

- variable arrays (VARRAYs), 1-8

- Export and Import, 2-15

- incomplete types, 2-14

- object types, 1-2

- use of table aliases, 2-7

- privileges, 2-10

- See also* object-relational model

- storage, 6-1

V

variables

- bind variables

- user-defined types, 3-2

- object variables, 4-4

VARRAY

- vs nested tables, 8-18

varrays, 1-8

- accessing, 5-14

- querying, 5-14

- See also* arrays, collections

- storage, 5-13

- updating, 5-14

- vs nested tables, 8-15

views

- See also* object views

- updatability, 4-11

Visual Basic, 3-9

W

WITH OBJECT IDENTIFIER clause, 4-4