Oracle® Database

Object-Relational Developer's Guide

11*g* Release 1 (11.1)

B28371-01

July 2007

Oracle Database Object-Relational Developer’s Guide 11*g* Release 1 (11.1)

B28371-01

Copyright © 1996, 2007, Oracle. All rights reserved.

Contributors: Shashaanka Agrawal, Geeta Arora, Eric Belden, Janis Greenberg, Chandrasekharan Iyer, Geoff Lee, Anand Manikutty, Valarie Moore, Magdi Morsi, Helen Yeh, Adiel Yoaz, Qin Yu

The Programs (which include both the software and documentation) contain proprietary information; 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, except to the extent required to obtain interoperability with other independently created software or as specified by law, 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. This document is not warranted to be 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.

If the Programs are delivered to the United States Government or anyone licensing or using the Programs on behalf of the United States Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the Programs, including documentation and technical data, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement, and, to the extent applicable, the additional rights set forth in FAR 52.227-19, Commercial Computer Software--Restricted Rights (June 1987). Oracle USA, Inc., 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 we disclaim liability for any damages caused by such use of the Programs.

Oracle, JD Edwards, PeopleSoft, and Siebel are registered trademarks of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

The Programs may provide links to Web sites and access to content, products, and services from third parties. Oracle is not responsible for the availability of, or any content provided on, third-party Web sites. You bear all risks associated with the use of such content. If you choose to purchase any products or services from a third party, the relationship is directly between you and the third party. Oracle is not responsible for: (a) the quality of third-party products or services; or (b) fulfilling any of the terms of the agreement with the third party, including delivery of products or services and warranty obligations related to purchased products or services. Oracle is not responsible for any loss or damage of any sort that you may incur from dealing with any third party.

Contents

4 Using PL/SQL With Object Types

5 Object Support in Oracle Programming Environments

6 Applying an Object Model to Relational Data

7 Managing Oracle Objects

8 Advanced Topics for Oracle Objects

9 Design Considerations for Oracle Objects

A Sample Application Using Object-Relational Features

List of Examples

Preface

Oracle Database Object-Relational Developer's Guide describes how to use the object-relational features of the Oracle Database, 11*g* release 1 (11.1). Information in this guide applies to versions of the Oracle Database that run on all platforms, and does not include system-specific information.

- Audience
- Documentation Accessibility
- Related Documents
- Conventions

Audience

Oracle Database Object-Relational Developer's Guide 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 content management, data warehousing, data/information integration, and similar applications that deal with complex structured data. The object views feature can be valuable when writing new C++, Java, or XML 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.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at

`http://www.oracle.com/accessibility/`

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

TTY Access to Oracle Support Services

Oracle provides dedicated Text Telephone (TTY) access to Oracle Support Services within the United States of America 24 hours a day, seven days a week. For TTY support, call 800.446.2398.

Related Documents

For more information, see these Oracle resources:

- *Oracle Database SecureFiles and Large Objects Developer's Guide* for information on Large Objects (LOBs)
- *Oracle Database Advanced Application Developer's Guide* for general information about developing applications
- *Oracle Database PL/SQL User's Guide and Reference* for information on PL/SQL, the procedural language extension to Oracle SQL
- *Oracle XML DB Developer's Guide* and *Oracle XML Developer's Kit Programmer's Guide* for information about developing applications with XML
- *Oracle Database JDBC Developer's Guide and Reference* and *Oracle Database Java Developer's Guide* to use Oracle object-relational features through Java
- *Oracle Call Interface Programmer's Guide* and *Oracle C++ Call Interface Programmer's Guide* for information on using the Oracle Call Interface (OCI) and Oracle C++ Call Interface to build third-generation language (3GL) applications that access the Oracle Server
- *Pro*C/C++ Programmer's Guide* for information on Oracle's Pro* series of precompilers, which allow you to embed SQL and PL/SQL in 3GL application programs written in Ada, C, C++, COBOL, or FORTRAN
- *Oracle Database SQL Language Reference* and *Oracle Database Administrator's Guide* for information on SQL
- *Oracle Database Concepts* for information on basic Oracle concepts

Many of the examples in this book use the sample schemas, which are installed by default when you select the Basic Installation option with an Oracle Database installation. Refer to *Oracle Database Sample Schemas* for information on how these schemas were created and how you can use them yourself.

Printed documentation is available for sale in the Oracle Store at

`http://oraclestore.oracle.com/`

To download free release notes, installation documentation, white papers, or other collateral, please visit the Oracle Technology Network (OTN). You must register online before using OTN; registration is free and can be done at

`http://www.oracle.com/technology/contact/welcome.html`

If you already have a username and password for OTN, then you can go directly to the documentation section of the OTN Web site at

`http://www.oracle.com/technology/documentation/index.html`

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

What’s New in Object-Relational Features?

This section describes the new object-relational features of Oracle 11*g* release 1 (11.1). New features information from previous releases is also retained to help those users upgrading to the current release.

The following sections describe the new features in Oracle Object-Relational Features:

- Oracle Database 11g Release 1 (11.1) New Features in Object-Relational Features
- Oracle Database 10g Release 1 (10.1) New Features in Object-Relational Features

Oracle Database 11g Release 1 (11.1) New Features in Object-Relational Features

Support for Generalized Invocation

Generalized invocation syntax is now supported. Therefore, a member method in a subtype can statically invoke (dispatch) a member method in any supertype in the supertype hierarchy of the current subtype, including the subtype's immediate supertype.

See "Generalized Invocation" on page 2-16.

Oracle Database 10*g* Release 1 (10.1) New Features in Object-Relational Features

New object-relational features for Oracle 10*g* release 1 (10.1) include the following collection enhancements:

Nested Table and Varray Storage

New functionality for nested table and varray storage, including the evolution of varray size and specification of a tablespace when storing nested tables. See "Creating Collection Data Types" on page 3-1.

Nested Table Comparisons and ANSI SQL Multiset Operations

New functionality for nested table comparisons and ANSI SQL multiset operations for nested tables. See "Operations on Collection Data Types" on page 3-10.

1

Introduction to Oracle Objects

This chapter describes the advantages and key features of the Oracle object-relational model. The chapter contains these topics:

- About Oracle Objects
- Advantages of Objects
- Key Features of the Object-Relational Model

About Oracle Objects

Oracle object types are user-defined types that make it possible to model real-world entities such as customers and purchase orders as objects in the database.

Oracle object technology is a layer of abstraction built on Oracle relational technology. New object types can be created from any built-in database types and any previously created object types, object references, and collection types. Metadata for user-defined types is stored in a schema that is available to SQL, PL/SQL, Java, and other published interfaces. Object datatypes make it easier to work with complex data, such as images, audio, and video.

Object types and related object-oriented features such as variable-length arrays and nested tables provide higher-level ways to organize and access data in the database. Underneath the object layer, data is still stored in columns and tables, but you are able to work with the data in terms of the real-world entities, such as customers and purchase orders, that make the data meaningful. Instead of thinking in terms of columns and tables when you query the database, you can simply select a customer.

Internally, statements about objects are still basically statements about relational tables and columns, and you can continue to work with relational data types and store data in relational tables as before. But now you have the option to take advantage of object-oriented features too. You can begin to use object-oriented features while continuing to work with most of your data relationally, or you can go over to an object-oriented approach entirely. For instance, you can define some object data types and store the objects in columns in relational tables, which enables you to extend the system built-in types with user-defined ones. You can also create object views of existing relational data to represent and access this data according to an object model. Or you can store object data in object tables, where each row is an object.

Advantages of Objects

In general, the object-type model is similar to the class mechanism found in C++ and Java. Like classes, objects make it easier to model complex, real-world business entities and logic, and the reusability of objects makes it possible to develop database applications faster and more efficiently. By natively supporting object types in the database, Oracle enables application developers to directly access the data structures used by their applications. No mapping layer is required between client-side objects and the relational database columns and tables that contain the data. Object abstraction and the encapsulation of object behaviors also make applications easier to understand and maintain.

Other advantages that objects offer over a purely relational approach are listed here.

Objects Can Encapsulate Operations Along with Data

Database tables contain only data. Objects can include the ability to perform operations that are likely to be needed on that data. Thus a purchase order object might include a method to sum the cost of all the items purchased. Or a customer object might have methods to return the customer's buying history and payment pattern. An application can simply call the methods to retrieve the information.

Objects Are Efficient

Using object types makes for greater efficiency:

- Object types and their methods are stored with the data in the database, so they are available for any application to use. Developers can benefit from work that is already done and do not need to re-create similar structures in every application.
- You can fetch and manipulate a set of related objects as a single unit. A single request to fetch an object from the server can retrieve other objects that are connected to it. For example, when you select a customer object, you get the customer's name, phone, and the multiple parts of the address in a single round-trip between the client and the server. When you reference a column of a SQL object type, you retrieve the whole object.

Objects Can Represent Part-Whole Relationships

In a relational system, it is awkward to represent complex part-whole relationships. A piston and an engine have the same status in a table for stock items. To represent pistons as parts of engines, you must create complicated schemas of multiple tables with primary key-foreign key relationships. Object types, on the other hand, give you a rich vocabulary for describing part-whole relationships. An object can have other objects as attributes, and the attribute objects can have their own object attributes too. An entire parts-list hierarchy can be built up in this way from interlocking object types.

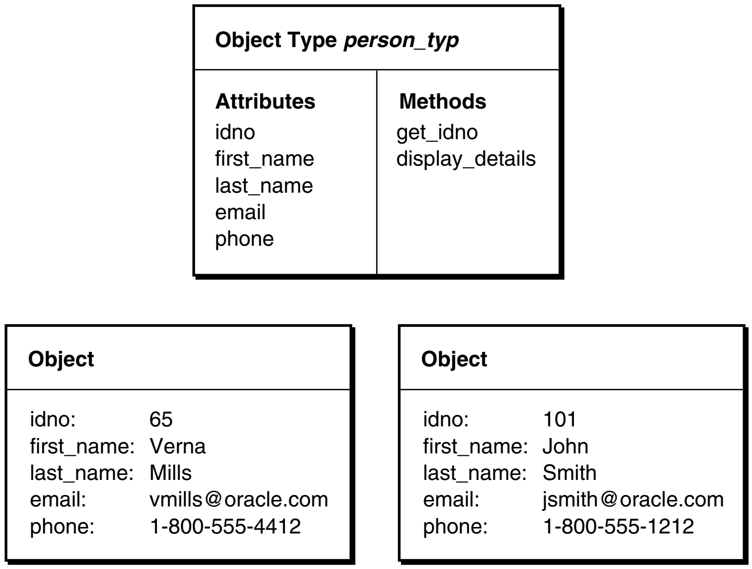
Key Features of the Object-Relational Model

Oracle implements the object-type system as an extension of the relational model. The object-type interface 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, cluster database, export and import, and loader. SQL*Plus and various programmatic interfaces to Oracle, including PL/SQL, Java, Oracle Call Interface, Pro*C/C++, and OO4O, have been enhanced with new extensions to support objects. The result is an object-relational model that offers the intuitiveness and economy of an object interface while preserving the high concurrency and throughput of a relational database.

Core Database Key Features

This section lists the key features and concepts of the object-relational model that are related to the database. Figure 1–1 shows an object type and instances of the object.

Figure 1–1 An Object Type and Object Instances



Object Types

An object type is a kind of data type. You can use it in the same ways that you use more familiar data types such as NUMBER or VARCHAR2. For example, you can specify an object type as the data type of a column in a relational table, and you can declare variables of an object type. You use a variable of an object type to contain a value of that object type. The value of an object type is an instance of that type. An object instance is also called an object.

You use the CREATE TYPE statement to define object types. Example 1–1 shows how to create an object type named person_typ. In the example, an object specification and object body are defined. For information on the CREATE TYPE SQL statement and on the CREATE TYPE BODY SQL statement, see *Oracle Database SQL Language Reference*.

Example 1–1 Creating the person_typ Object Type

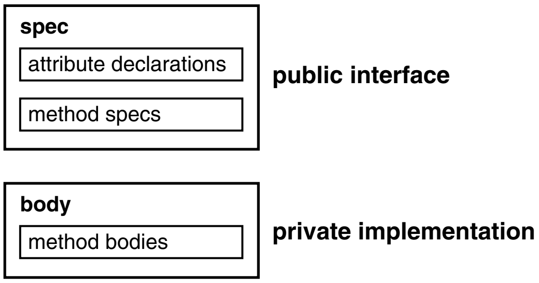
```
CREATE TYPE person_typ AS OBJECT (
  idno           NUMBER,
  first_name     VARCHAR2(20),
  last_name      VARCHAR2(25),
  email          VARCHAR2(25),
  phone          VARCHAR2(20),
  MAP MEMBER FUNCTION get_idno RETURN NUMBER,
  MEMBER PROCEDURE display_details ( SELF IN OUT NOCOPY person_typ ));
/

CREATE TYPE BODY person_typ AS
  MAP MEMBER FUNCTION get_idno RETURN NUMBER IS
  BEGIN
    RETURN idno;
  END;
  MEMBER PROCEDURE display_details ( SELF IN OUT NOCOPY person_typ ) IS
  BEGIN
    -- use the PUT_LINE procedure of the DBMS_OUTPUT package to display details
    DBMS_OUTPUT.PUT_LINE(TO_CHAR(idno) || ' ' || first_name || ' ' || last_name);
    DBMS_OUTPUT.PUT_LINE(email || ' '  || phone);
  END;
END;
```

/

Object types have some important differences from the more familiar data types that are native to a relational database:

- A set of object types does not come ready-made with the database. Instead, you define the specific object types you want by extending built-in types with user-defined ones as shown in Example 1–1.
- Object types are composed of parts, called attributes and methods, illustrated in Figure 1–2.
  - Attributes hold the data about an object's features of interest. For example, a `student` object type might have `name`, `major`, and `graduation` date attributes. An attribute has a declared data type which can in turn be another object type. Taken together, the attributes of an object instance contain that object's data.
  - Methods are procedures or functions provided to enable applications to perform useful operations on the attributes of the object type. Methods are an optional element of an object type. They define the behavior of objects of that type and determine what (if anything) that type of object can do.

Figure 1–2 Object Attributes and Methods



Object types are less generic than native data types. In fact, this is one of their major virtues. You can define object types to model the actual structure of the real-world entities, such as customers and purchase orders, that application programs deal with. You can capture the structural interrelationships of objects and their attributes instead of flattening this structure into a two-dimensional, purely relational schema of tables and columns. This can make it easier and more intuitive to manage the data for these entities. In this respect object types are like Java and C++ classes.

You can think of an object type as a blueprint or template which defines structure and behavior. An instantiation of the object type creates an object built according to the template. Object types are database schema objects, subject to the same kinds of administrative control as other schema objects.

With object types, you can store related pieces of data in a unit, along with the behaviors defined for that data. Application code can then retrieve and manipulate these units as objects. See Chapter 7, "Managing Oracle Objects".

Objects

When you create a variable of an object type, you create an instance of the type and the result is an object. An object has the attributes and methods defined for its type. Because an object instance is a concrete thing, you can assign values to its attributes and call its methods.

In Example 1–1, the CREATE TYPE statement defines the object type person_typ.

The indented elements idno, name, and phone in the CREATE TYPE statements are attributes. Each has a data type declared for it. These are simplified examples and do not show how to specify member methods.

Defining an object type does not allocate any storage. After they are defined, object types can be used 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, as shown in Example 1–2.

Example 1–2 Creating the contacts Table with an Object Type Column

```
CREATE TABLE contacts (
  contact         person_typ,
  contact_date    DATE );

INSERT INTO contacts VALUES (
  person_typ (65, 'Verna', 'Mills', 'vmills@oracle.com', '1-800-555-4412'),
 '24 Jun 2003' );
```

The contacts table is a relational table with an object type as the data type of one of its columns. Objects that occupy columns of relational tables are called column objects. See "Objects and Table Storage" on page 1-8.

Object Methods

Methods are functions or procedures that you can declare in an object type definition to implement behavior that you want objects of that type to perform. For example, a method is declared in Example 1–1 to allow comparisons between person_typ objects.

The general kinds of methods that can be declared in a type definition are:

- Member
- Static
- Constructor

A principal use of methods is to provide access to the data of an object. You can define methods for operations that an application is likely to want to perform on the data so that the application does not have to code these operations itself. To perform the operation, an application calls the appropriate method on the appropriate object.

In Example 1–3, the SQL statement uses the get_idno() method to display the Id number of persons in the contacts table:

Example 1–3 Using the get_idno Object Method

```
SELECT c.contact.get_idno() FROM contacts c;
```

You can also define static methods to compare object instances and to perform operations that do not use any particular object's data but instead are global to an object type.

A constructor method is implicitly defined for every object type, unless this default constructor is over-written with a user-defined constructor. A constructor method is called on a type to construct or create an object instance of the type. See "Object Methods" on page 2-7.

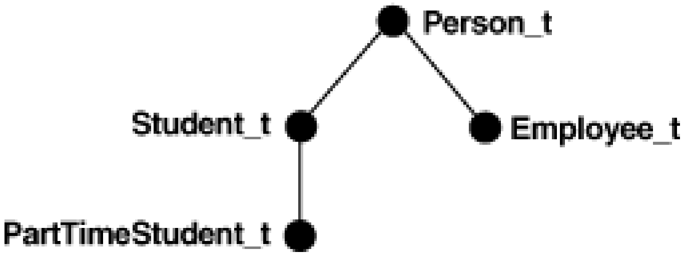
Type Inheritance

Type inheritance adds to the usefulness of objects by enabling you to create type hierarchies by defining successive levels of increasingly specialized subtypes that derive from a common ancestor object type, which is called a supertype of the derived types. Derived subtypes inherit the features of the parent object type but extend the parent type definition. The specialized types can add new attributes or methods, or redefine methods inherited from the parent. The resulting type hierarchy provides a higher level of abstraction for managing the complexity of an application model.

For example, specialized types of persons, such as a student type or a part-time student type with additional attributes or methods, might be derived from a general person object type. See "Inheritance in SQL Object Types" on page 2-12.

Appendix 1–3 illustrates two subtypes, `Student_t` and `Employee_t`, created under `Person_t`.

Figure 1–3 A Type Hierarchy



Type Evolution

Using an `ALTER TYPE` statement, you can modify, or evolve, an existing object type to make the following changes:

- Add and drop attributes
- Add and drop methods
- Modify a numeric attribute to increase its length, precision, or scale
- Modify a varying length character attribute to increase its length
- Change a type's `FINAL` and `INSTANTIABLE` properties

Dependencies of a type to be altered are checked, using essentially the same validations applied for a `CREATE TYPE` statement. If a type or any of its dependent types fails the type validations, the `ALTER TYPE` statement rolls back.

Metadata for all tables and columns that use an altered type are updated for the new type definition so that data can be stored in them in the new format. Existing data can be converted to the new format either all at once or piecemeal, as it is updated. In either case, data is always presented in the new type definition even if it is still stored in the format of the older one.

Object Tables

An object table is a special kind of table in which each row represents an object. In Example 1–4, the statement creates an object table for `person_typ` objects.

Example 1–4 Creating the person_obj_table Object Table

```
CREATE TABLE person_obj_table OF person_typ;
```

You can view this table in two ways:

- As a single-column table, in which each row is a person_typ object, allowing you to perform object-oriented operations.
- As a multi-column table, in which each attribute of the object type person_typ such as idno, name, and phone, occupies a column, allowing you to perform relational operations.

Example 1–5 illustrates several operations on an object table.

Example 1–5 Operations on the person_obj_table Object Table

```
INSERT INTO person_obj_table VALUES (
       person_typ(101, 'John', 'Smith', 'jsmith@oracle.com', '1-800-555-1212') );

SELECT VALUE(p) FROM person_obj_table p
        WHERE p.last_name = 'Smith';

DECLARE
  person person_typ;
BEGIN -- PL/SQL block for selecting a person and displaying details
  SELECT VALUE(p) INTO person FROM person_obj_table p WHERE p.idno = 101;
  person.display_details();
END;
/
```

The first SQL statement in Example 1–5 inserts a person_typ object into person_obj_table, treating person_obj_table as a multi-column table.

The second SQL statement selects from person_obj_table as a single-column table, using the VALUE function to return rows as object instances. See "VALUE" on page 2-36 for information on the VALUE function.

The PL/SQL block in Example 1–5 selects a specific person and executes a member function of person_typ to display details about the specified person. For more information on the use of PL/SQL with objects, see Chapter 4, "Using PL/SQL With Object Types".

Object Identifiers

By default, every row object in an object table has an associated logical object identifier (OID) that uniquely identifies it in an object table. In a distributed and replicated environment, the system-generated unique identifier lets Oracle identify objects unambiguously.

The object identifier column of an object table is a hidden column. Although the object identifier value in itself is not very meaningful to an object-relational application, Oracle uses this value to construct object references to the row objects. Applications need to be concerned with only object references that are used for fetching and navigating objects.

The purpose of the object identifier for a row object is to uniquely identify it in an object table. To do this Oracle implicitly creates and maintains an index on the object identifier column of an object table. The system-generated unique identifier has many advantages, among which are the unambiguous identification of objects in a distributed and replicated environment.

See "Storage Considerations for Object Identifiers (OIDs)" on page 9-4 for information on Object Identifiers and using REFs to OIDs.

Primary-Key Based Object Identifiers For applications that do not require the functionality provided by globally unique system-generated identifiers, storing 16 extra bytes with each object and maintaining an index on it may not be efficient. Oracle allows the option of specifying the primary key value of a row object as the object identifier for the row object.

Primary-key based identifiers also have the advantage of enabling a more efficient and easier loading of the 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.

Objects and Table Storage

Objects can be stored in tables in two ways:

- Objects stored alone

  Objects that are stored as complete rows in object tables are called row objects.

- Objects stored with other table data

  Objects that are stored as columns of a table in a larger row, or are attributes of other objects, are called column objects.

Object Views

An object view is a way to access relational data using object-relational features. It lets you develop object-oriented applications without changing the underlying relational schema.

Oracle allows the creation of an object abstraction over existing relational data through the object view mechanism. You access objects that belong to an object view in the same way that you access row objects in an object table. Oracle also supports materialized view objects of user-defined types from data stored in relational schemas and tables. By using object views, you can develop object-oriented applications without having to modify existing relational database schemas.

Object views also let you exploit the polymorphism that a type hierarchy makes possible. A polymorphic expression can take a value of the expression's declared type or any of that type's subtypes. If you construct a hierarchy of object views that mirrors some or all of the structure of a type hierarchy, you can query any view in the hierarchy to access data at just the level of specialization you are interested in. If you query an object view that has subviews, you can get back polymorphic data—rows for both the type of the view and for its subtypes. See Chapter 6, "Applying an Object Model to Relational Data".

References

A `REF` is a logical pointer to a row object that is constructed from the object identifier (OID) of the referenced object and is an Oracle built-in data type. `REF`s and collections of `REF`s model associations among objects, particularly many-to-one relationships, thus reducing the need for foreign keys. `REF`s 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 the object it refers to. You can change a `REF` so that it points to a different object of the same object type hierarchy or assign it a null value.

Example 1–6 illustrates a simple use of a `REF`.

Example 1–6 Using a REF to the emp_person_typ Object

```
CREATE TYPE emp_person_typ AS OBJECT (
  name     VARCHAR2(30),
  manager  REF emp_person_typ );
/
CREATE TABLE emp_person_obj_table OF emp_person_typ;

INSERT INTO emp_person_obj_table VALUES (
   emp_person_typ ('John Smith', NULL));

INSERT INTO emp_person_obj_table
  SELECT emp_person_typ ('Bob Jones', REF(e))
    FROM emp_person_obj_table e
    WHERE e.name = 'John Smith';
```

See "Rules for REF Columns and Attributes" on page 2-5 and "Design Considerations for REFs" on page 9-6.

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 REF types require less storage space and allow more efficient access than unscoped REF types.

Example 1–7 shows REF column contact_ref scoped to person_obj_table which is an object table of type person_typ.

Example 1–7 Creating the contacts_ref Table Using a Scoped REF

```
CREATE TABLE contacts_ref (
  contact_ref   REF person_typ SCOPE IS person_obj_table,
  contact_date  DATE );
```

To insert a row in the table, you could issue the following:

```
INSERT INTO contacts_ref
  SELECT REF(p), '26 Jun 2003'
    FROM person_obj_table p
    WHERE p.idno = 101;
```

A REF can be scoped to an object table of the declared type (person_typ in the example) or of any subtype of the declared type. If scoped to an object table of a subtype, the REF column is effectively constrained to hold references only to instances of the subtype (and its subtypes, if any) in the table. See "Inheritance in SQL Object Types" on page 2-12.

Dangling REFs

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

Dangling REFs can be avoided by defining referential integrity constraints. See "Rules for REF Columns and Attributes" on page 2-5.

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, as shown in Example 1–8.

Example 1–8 Using DEREF to Dereference a REF

```
SELECT DEREF(c.contact_ref), c.contact_date FROM contacts_ref c;
```

Oracle also provides implicit dereferencing of REFs. For example, to access the manager's name for an employee, you can use a SQL expression similar to the following:

```
SELECT e.name, e.manager.name FROM emp_person_obj_table e
  WHERE e.name = 'Bob Jones';
```

In the example, e.manager.name follows the pointer from the person'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. You can obtain a REF to the person with idno equal to 1 as shown in Example 1–9.

Example 1–9 Obtaining a REF to a Row Object

```
DECLARE
  person_ref REF person_typ;
BEGIN
  SELECT REF(p) INTO person_ref
    FROM person_obj_table p
    WHERE p.idno = 101;
END;
/
```

The query must return exactly one row. See "Storage Size of REFs" on page 9-6.

Collections

For modeling multi-valued attributes and many-to-many relationships, Oracle supports two collection data types: varrays and nested tables. Collection types can be used anywhere other data types can be used. You can have object attributes of a collection type in addition to columns of a collection type. For example, you might give a purchase order object type a nested table attribute to hold the collection of line items for a given purchase order.

You use the CREATE TYPE statement to define collection types. In Example 1–10, the CREATE TYPE statements define the object types people_typ and dept_persons_typ.

Example 1–10 Creating the people_typ Collection Data Type

```
CREATE TYPE people_typ AS TABLE OF person_typ;
/

CREATE TYPE dept_persons_typ AS OBJECT (
  dept_no    CHAR(5),
  dept_name  CHAR(20),
  dept_mgr   person_typ,
  dept_emps  people_typ);
/
```

In this simplified example, `people_typ` is a collection type, specifically a nested table type. The `dept_persons_typ` object type has an attribute `dept.emps` of this type. Each row in the `dept.emps` nested table is an object of type `person_typ` which was defined in Example 1–1 on page 1-3. See "Creating Collection Data Types" on page 3-1.

Language Binding Features

This section lists the key features of the object-relational model that are related to languages and application programming interfaces (APIs).

SQL Object Extensions

To support the new object-related features, SQL extensions, including new DDL, have been added 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. See"SQL" on page 5-1.

PL/SQL Object Extensions

PL/SQL is an Oracle database programming language that is tightly integrated with SQL. With the addition of object types and other SQL types, PL/SQL has been enhanced to operate on object types seamlessly. Thus, application developers can use PL/SQL to implement logic and operations on user-defined types that execute in the database server. See "PL/SQL" on page 5-2.

Java Support for Oracle Objects

Oracle Java VM is tightly integrated with the RDBMS and supports access to Oracle Objects through object extensions to Java Database Connectivity (JDBC), which provides dynamic SQL, and SQLJ, which provides static SQL. Thus, application developers can use Java to implement logic and operations on object types that execute in the database. With Oracle, you can now also create SQL types mapped to existing Java classes to provide persistent storage for Java objects using SQLJ object types where all the methods are implemented in their corresponding Java classes. See "Java Object Storage" on page 5-12.

See Also: *Oracle Database JDBC Developer's Guide and Reference*

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. Generic external procedures can be written that declare one or more parameters to be of a system-defined generic type. The generic type permits a procedure that uses it to work with data of any built-in or user-defined type.

Object Type Translator/JPublisher

Object Type Translator (OTT) and Oracle JPublisher provide client-side mappings to object type schemas by using schema information from the Oracle data dictionary to generate header files containing Java classes and C structures and indicators. These generated header files can be used in host-language applications for transparent access to database objects.

Client-Side Cache

Oracle 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.

Oracle Call Interface and Oracle C++ Call Interface

Oracle Call Interface (OCI) and Oracle C++ Call Interface provide a comprehensive application programming interface for application and tool developers seeking to use the object capabilities of Oracle. Oracle Call Interface provides a run-time environment with functions to connect to an Oracle 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 through declarative SQL DML. Oracle Call Interface also provides a number of functions for accessing metadata about object types defined in the server at run-time. Such a set of functions facilitates dynamic access to the object metadata and the actual object data stored in the database. See "Oracle Call Interface (OCI)" on page 5-2 and "Oracle C++ Call Interface (OCCI)" on page 5-6.

Pro*C/C++ Object Extensions

The Oracle Pro*C/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/C++ precompiler allows application developers to use the Oracle client-side object cache and the Object Type Translator Utility. Pro*C/C++ supports the use of C bind variables for Oracle object types. Furthermore, Pro*C/C++ provides simplified syntax to allocate and free objects of SQL types and access them by either SQL DML, or through 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 Oracle server to program bind variables in the client, and simple ways to manage and manipulate database objects in the client process. See "Oracle Call Interface (OCI)" on page 5-2.

OO4O Object Extensions

Oracle Objects For OLE (OO4O) is a set of COM Automation interfaces/objects for connecting to Oracle database servers, executing queries and managing the results. Automation interfaces in OO4O provide easy and efficient access to Oracle features 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. See "Oracle Objects For OLE (OO4O)" on page 5-7.

2 Basic Components of Oracle Objects

This chapter provides basic information about working with objects. It explains what object types and subprograms are, and describes how to create and work with a hierarchy of object types that are derived from a shared root type and are connected by inheritance.

This chapter contains these topics:

- SQL Object Types and References
- Object Methods
- Inheritance in SQL Object Types
- Functions and Operators Useful with Objects

SQL Object Types and References

This section describes SQL object types and references, including:

- Null Objects and Attributes
- Character Length Semantics
- Constraints for Object Tables
- Indexes for Object Tables
- Triggers for Object Tables
- Rules for REF Columns and Attributes
- Name Resolution
- Restriction on Using User-Defined Types with a Remote Database

You can create a SQL object type with the `CREATE TYPE` statement. An example of creating an object type is shown in Example 2–1 on page 2-2. For information on the `CREATE TYPE` SQL statement, see *Oracle Database SQL Language Reference*. For information on the `CREATE TYPE BODY` SQL statement, see *Oracle Database SQL Language Reference*.

Null Objects and Attributes

A table column, object attribute, collection, or collection element is `NULL` if it has been initialized to `NULL` or has not been initialized at all. Usually, a `NULL` value is replaced by an actual value later on.

An object whose value is `NULL` is called atomically null. An atomically null object is different from one that simply happens to have null values for all its attributes. When all the attributes of an object are null, these attributes can still be changed, and the object's subprograms or methods can be called. With an atomically null object, you can do neither of these things. In Example 2–1, consider the contacts table which contains the person_typ object type.

Example 2–1 Inserting NULLs for Objects in a Table

```
CREATE TYPE person_typ AS OBJECT (
  idno          NUMBER,
  name          VARCHAR2(30),
  phone         VARCHAR2(20),
  MAP MEMBER FUNCTION get_idno RETURN NUMBER,
  MEMBER PROCEDURE display_details ( SELF IN OUT NOCOPY person_typ ) );
/

CREATE TYPE BODY person_typ AS
  MAP MEMBER FUNCTION get_idno RETURN NUMBER IS
  BEGIN
    RETURN idno;
  END;
  MEMBER PROCEDURE display_details ( SELF IN OUT NOCOPY person_typ ) IS
  BEGIN
    -- use the PUT_LINE procedure of the DBMS_OUTPUT package to display details
    DBMS_OUTPUT.PUT_LINE(TO_CHAR(idno) || ' - '  || name || ' - '  || phone);
  END;
END;
/
CREATE TABLE contacts (
  contact       person_typ,
  contact_date  DATE );

INSERT INTO contacts VALUES (
  person_typ (NULL, NULL, NULL), '24 Jun 2003' );

INSERT INTO contacts VALUES (
  NULL, '24 Jun 2003' );
```

The two INSERT statements give two different results. In both cases, Oracle allocates space in contacts for a new row and sets its DATE column to the value given. But in the first case, Oracle allocates space for an object in the contact column and sets each of the object's attributes to NULL. In the second case, Oracle sets the person_typ field itself 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.

Character Length Semantics

Lengths for character types CHAR and VARCHAR2 may be specified as a number of characters, instead of bytes, in object attributes and collections even if some of the characters consist of multiple bytes.

To specify character-denominated lengths for CHAR and VARCHAR2 attributes, you add a qualifier char to the length specification.

Like CHAR and VARCHAR2, NCHAR and NVARCHAR2 may also be used as attribute types in objects and collections. These types are always implicitly measured in terms of characters, so no char qualifier is used.

For example, the following statement creates an object with both a character-length VARCHAR2 attribute and an NCHAR attribute:

Example 2–2 Creating the employee_typ Object Using a char Qualifier

```
CREATE TYPE employee_typ AS OBJECT (
  name        VARCHAR2(30 char),
  language    NCHAR(10),
  phone       VARCHAR2(20) );
/
```

For CHAR and VARCHAR2 attributes whose length is specified without a char qualifier, the default unit of measure characters or bytes is determined by whether the NLS_LENGTH_SEMANTICS initialization parameter is set to CHAR or BYTE.

> **See Also:** *Oracle Database Globalization Support Guide* for information on character length semantics

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.

Example 2–3 and Example 2–4 illustrate the possibilities.

Example 2–3 places a PRIMARY KEY constraint on the office_id column of the object table office_typ.

Example 2–3 Creating the office_tab Object Table with a Constraint

```
CREATE TYPE location_typ AS OBJECT (
  building_no  NUMBER,
  city         VARCHAR2(40) );
/

CREATE TYPE office_typ AS OBJECT (
  office_id    VARCHAR(10),
  office_loc   location_typ,
  occupant     person_typ );
/

CREATE TABLE office_tab OF office_typ (
             office_id      PRIMARY KEY );
```

The department_mgrs table in Example 2–4 has a column whose type is the object type location_typ defined in Example 2–3. The example defines constraints on scalar attributes of the location_typ objects that appear in the dept_loc column of the table.

Example 2–4 Creating the department_mgrs Table with Multiple Constraints

```
CREATE TABLE department_mgrs (
  dept_no     NUMBER PRIMARY KEY,
  dept_name   CHAR(20),
  dept_mgr    person_typ,
  dept_loc    location_typ,
  CONSTRAINT  dept_loc_cons1
      UNIQUE (dept_loc.building_no, dept_loc.city),
  CONSTRAINT  dept_loc_cons2
       CHECK (dept_loc.city IS NOT NULL) );

INSERT INTO department_mgrs VALUES
          ( 101, 'Physical Sciences',
           person_typ(65,'Vrinda Mills', '1-800-555-4412'),
           location_typ(300, 'Palo Alto'));
```

Indexes for Object 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. For an example of an index on a nested table, see Example 3–4 on page 3-4.

You can define indexes on leaf-level scalar attributes of column objects, as shown in Example 2–5. 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.

Example 2–5   Creating an Index on an Object Type in a Table

```
CREATE TABLE department_loc (
  dept_no     NUMBER PRIMARY KEY,
  dept_name   CHAR(20),
  dept_addr   location_typ );

CREATE INDEX  i_dept_addr1
          ON  department_loc (dept_addr.city);

INSERT INTO department_loc VALUES
          ( 101, 'Physical Sciences',
           location_typ(300, 'Palo Alto'));
INSERT INTO department_loc VALUES
          ( 104, 'Life Sciences',
           location_typ(400, 'Menlo Park'));
INSERT INTO department_loc VALUES
          ( 103, 'Biological Sciences',
           location_typ(500, 'Redwood Shores'));
```

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.

Example 2–6 defines a trigger on the office_tab table defined in "Constraints for Object Tables" on page 2-3.

Example 2–6   Creating a Trigger on Objects in a Table

```
CREATE TABLE movement (
     idno           NUMBER,
     old_office     location_typ,
     new_office     location_typ );

CREATE TRIGGER trigger1
  BEFORE UPDATE
             OF  office_loc
             ON  office_tab
   FOR EACH ROW
           WHEN  (new.office_loc.city = 'Redwood Shores')
   BEGIN
     IF :new.office_loc.building_no = 600 THEN
      INSERT INTO movement (idno, old_office, new_office)
       VALUES (:old.occupant.idno, :old.office_loc, :new.office_loc);
     END IF;
   END;
/

INSERT INTO movement VALUES
   ( 101, location_typ(300, 'Palo Alto'),
     location_typ(400, 'Menlo Park'));
```

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.

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.

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 qualifying table names in some relational operations. For example, if dept_addr is a column in the department_loc table and old_office is a column in the movement table, you can use the following:

```
SELECT * FROM department_loc WHERE EXISTS
  (SELECT * FROM movement WHERE dept_addr = old_office);
```

Oracle determines which table each column belongs to.

Using the dot notation, you can qualify the column names with table names or table aliases to make things more maintainable. For example:

Example 2–7 Using the Dot Notation for Name Resolution

```
SELECT * FROM department_loc WHERE EXISTS
  (SELECT * FROM movement WHERE department_loc.dept_addr = movement.old_office);

SELECT * FROM department_loc d WHERE EXISTS
  (SELECT * FROM movement m WHERE d.dept_addr = m.old_office);
```

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 depts and forget to change the query, Oracle automatically recompiles the query so that the inner SELECT uses the assignment column from the depts table. This situation is called inner capture.

To avoid inner capture and similar problems resolving references, Oracle requires you to use a table alias to qualify any dot-notational reference to subprograms or attributes of objects. Use of a table alias is optional when referencing top-level attributes of an object table directly, without using the dot notation.

For example, the following statements define two tables that contain the person_typ object type. person_obj_table is an object table for objects of type person_typ, and contacts is a relational table that contains a column of an object type.

The following queries show some correct and incorrect ways to reference attribute idno:

```
SELECT idno FROM person_obj_table;         --Correct
SELECT contact.idno FROM contacts;         --Illegal
SELECT contacts.contact.idno FROM contacts; --Illegal
SELECT p.contact.idno FROM contacts p;     --Correct
```

- In the first SELECT statement, idno is the name of a column of person_obj_table. It references this top-level attribute directly, without using the dot notation, so no table alias is required.
- In the second SELECT statement, idno is the name of an attribute of the person_typ object in the column named contact. This reference uses the dot notation and so requires a table alias, as shown in the fourth SELECT statement.
- The third SELECT uses the table name itself to qualify this the reference. This is incorrect; a table alias is required.

You must qualify a reference to an object attribute or subprogram with a table alias rather than a table name even if the table name is itself qualified by a schema name.

For example, the following expression tries to refer to the HR schema, department_loc table, dept_addr column, and city attribute of that column. But the expression is incorrect because department_loc is a table name, not an alias.

```
HR.department_loc.dept_addr.city
```

The same requirement applies to attribute references that use REFs.

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

Note: Oracle recommends 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.

Restriction on Using User-Defined Types with a Remote Database

User-defined types (specifically, types declared with a SQL CREATE TYPE statement, as opposed to types declared within a PL/SQL package) are currently useful only within a single database. You cannot use a database link to do any of the following:

- Connect to a remote database to select, insert, or update a user-defined type or an object REF on a remote table

  You can use the CREATE TYPE statement with the optional keyword OID to create a user-specified object identifier (OID) that allows an object type to be used in multiple databases. See the discussion on assigning an OID to an object type in the *Oracle Database Data Cartridge Developer's Guide*.

- Use database links within PL/SQL code to declare a local variable of a remote user-defined type
- Convey a user-defined type argument or return value in a PL/SQL remote procedure call.

Object Methods

Subprograms, or methods, are functions or procedures that you can declare in an object type definition to implement behavior that you want objects of that type to perform. An application calls the subprograms to invoke the behavior.

For example, you might declare a function get_sum() to get a purchase order object to return the total cost of its line items. The following line of code calls such a function for purchase order po and returns the amount into sum_line_items.

```
sum_line_items = po.get_sum();
```

In SQL, the parentheses are required for all subprogram calls. Unlike with PL/SQL functions and procedures, SQL requires parentheses for subprogram calls that do not have arguments.

Subprograms, or methods, can be written in PL/SQL or virtually any other programming language. Methods written in PL/SQL or Java are stored in the database. Methods written in other languages, such as C, are stored externally.

These types of methods are described in this section:

- Member Methods
- Methods for Comparing Objects
- Static Methods
- Constructor Methods
- External Implemented Methods

Member Methods

Member methods are the means by which an application gains access to an object instance's data. You define a member method in the object type for each operation that you want an object of that type to be able to perform. For example, the method `get_sum()` that sums the total cost of a purchase order's line items operates on the data of a particular purchase order and is a member method.

Member methods have a built-in parameter named `SELF` that denotes the object instance on which the method is currently being invoked. Member methods can reference the attributes and methods of `SELF` without a qualifier. This makes it simpler to write member methods. In Example 2–8 the code shows a method declaration that takes advantage of `SELF` to omit qualification of the attributes `hgt`, `len`, and `wth`.

Example 2–8 Creating a Member Method

```
-- Creating a Member Method example, not sample schema
CREATE TYPE solid_typ AS OBJECT (
  len    INTEGER,
  wth    INTEGER,
  hgt    INTEGER,
  MEMBER FUNCTION surface RETURN INTEGER,
  MEMBER FUNCTION volume RETURN INTEGER,
  MEMBER PROCEDURE display (SELF IN OUT NOCOPY solid_typ) );
/

CREATE TYPE BODY solid_typ AS
  MEMBER FUNCTION volume RETURN INTEGER IS
  BEGIN
    RETURN len * wth * hgt;
 -- RETURN SELF.len * SELF.wth * SELF.hgt; -- equivalent to previous line
  END;
  MEMBER FUNCTION surface RETURN INTEGER IS
  BEGIN -- not necessary to include SELF prefix in following line
    RETURN 2 * (len * wth + len * hgt + wth * hgt);
  END;
  MEMBER PROCEDURE display (SELF IN OUT NOCOPY solid_typ) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Length: ' || len || ' - '  || 'Width: ' || wth
                          || ' - '  || 'Height: ' || hgt);
    DBMS_OUTPUT.PUT_LINE('Volume: ' || volume || ' - ' || 'Surface area: '
                          || surface);
  END;
END;
/

CREATE TABLE solids of solid_typ;
INSERT INTO solids VALUES(10, 10, 10);
INSERT INTO solids VALUES(3, 4, 5);
SELECT * FROM solids;
SELECT s.volume(), s.surface() FROM solids s WHERE s.len = 10;
DECLARE
  solid solid_typ;
BEGIN -- PL/SQL block for selecting a solid and displaying details
  SELECT VALUE(s) INTO solid FROM solids s WHERE s.len = 10;
  solid.display();
END;
/
DROP TABLE solids;
DROP TYPE solid_typ FORCE;
```

`SELF` does not need to be explicitly declared, although it can be. It is always the first parameter passed to the method.

- In member functions, if SELF is not declared, its parameter mode defaults to IN.
- In member procedures, if SELF is not declared, its parameter mode defaults to IN OUT. The default behavior does not include the NOCOPY compiler hint.

You invoke a member method using dot notation, for example, object_variable.method(). This notation specifies the object on which to invoke the method, then the method to call. Any parameters must be placed inside the required parentheses. See also "Using SELF IN OUT NOCOPY with Member Procedures" on page 9-21.

Methods for Comparing Objects

The values of a scalar data type such as CHAR or REAL have a predefined order, which allows them to be compared. But an object type, such as a person_typ, which can have multiple attributes of various data types, has no predefined axis of comparison. To be able to compare and order variables of an object type, you must specify a basis for comparing them. Two special kinds of member methods can be defined for doing this: map methods and order methods.

Map Methods

A map method is an optional kind of method that provides a basis for comparing objects by mapping object instances to one of the scalar types DATE, NUMBER, VARCHAR2 or to an ANSI SQL type such as CHARACTER or REAL. With a map method, you can order any number of objects by calling each object's map method once to map that object to a position on the axis used for the comparison, such as a number or date. Example 2–1 on page 2-2 contains a simple map method.

From the standpoint of writing one, a map method is simply a parameter-less member function that uses the MAP keyword and returns one of the data types just listed. What makes a map method special is that, if an object type defines one, the method is called automatically to evaluate such comparisons as obj_1 > obj_2 and comparisons implied by the DISTINCT, GROUP BY, UNION, and ORDER BY clauses which require sorting by rows.

Where obj_1 and obj_2 are two object variables that can be compared using a map method map(), the comparison:

```
obj_1 > obj_2
```

is equivalent to:

```
obj_1.map() > obj_2.map()
```

And similarly for other relational operators besides the greater than (>) operator.

The following example defines a map method area() that provides a basis for comparing rectangle objects by their area:

Example 2–9 Creating a Map Method

```
CREATE TYPE rectangle_typ AS OBJECT (
  len NUMBER,
  wid NUMBER,
  MAP MEMBER FUNCTION area RETURN NUMBER);
/

CREATE TYPE BODY rectangle_typ AS
  MAP MEMBER FUNCTION area RETURN NUMBER IS
  BEGIN
```

```
     RETURN len * wid;
  END area;
END;
/
```

An object type can declare, at most, one map method or one order method. A subtype can declare a map method only if its root supertype declares one. See "Equal and Not Equal Comparisons" on page 3-14 for the use of map methods when comparing collections that contain object types.

Order Methods

Order methods make direct object-to-object comparisons. Unlike map methods, they cannot map any number of objects to an external axis. They simply tell you that the current object is less than, equal to, or greater than the other object that it is being compared to, with respect to the criterion used by the method.

An order method is a function with one declared parameter for another object of the same type. The method must be written to return either a negative number, zero, or a positive number. The return signifies that the object picked out by the SELF parameter is respectively less than, equal to, or greater than the other parameter's object.

As with map methods, an order method, if one is defined, is called automatically whenever two objects of that type need to be compared.

Order methods are useful where comparison semantics may be too complex to use a map method. For example, to compare binary objects such as images, you might create an order method to compare the images by their brightness or number of pixels.

An object type can declare at most one order method or one map method. Only a type that is not derived from another type can declare an order method; a subtype cannot define one.

Example 2–10 shows an order method that compares locations by building number:

Example 2–10 Creating an Order Method

```
CREATE TYPE location_typ AS OBJECT (
  building_no  NUMBER,
  city         VARCHAR2(40),
  ORDER MEMBER FUNCTION match (l location_typ) RETURN INTEGER );
/
CREATE TYPE BODY location_typ AS
  ORDER MEMBER FUNCTION match (l location_typ) RETURN INTEGER IS
  BEGIN
    IF building_no < l.building_no THEN
      RETURN -1;               -- any negative number will do
    ELSIF building_no > l.building_no THEN
      RETURN 1;                -- any positive number will do
    ELSE
      RETURN 0;
    END IF;
  END;
END;
/
```

Guidelines for Comparison Methods

A map method maps object values into scalar values and can order multiple values by their position on the scalar axis. An order method directly compares values for two particular objects.

You can declare a map method or an order method but not both. If you declare a method of either type, you can compare objects in SQL and procedural statements. However, if you declare neither method, you can compare objects only in SQL statements and only for equality or inequality. Two objects of the same type are considered equal only if the values of their corresponding attributes are equal.

When sorting or merging a large number of objects, use a map method. One call maps all the objects into scalars, then sorts the scalars. An order method is less efficient because it must be called repeatedly (it can compare only two objects at a time). See "Performance of Object Comparisons" on page 9-5.

Comparison Methods in Type Hierarchies

In a type hierarchy, where definitions of specialized types are derived from definitions of more general types, only the root type—the most basic type, from which all other types are derived—can define an order method. If the root type does not define one, its subtypes cannot define one either.

If the root type specifies a map method, any of its subtypes can define a map method that overrides the map method of the root type. But if the root type does not specify a map method, no subtype can specify one either.

So if the root type does not specify either a map or an order method, none of the subtypes can specify either a map or order method. See "Inheritance in SQL Object Types" on page 2-12 and "Inheriting, Overloading, and Overriding Methods" on page 2-19.

Static Methods

Static methods are invoked on the object type, not its instances. You use a static method for operations that are global to the type and do not need to reference the data of a particular object instance. A static method has no SELF parameter.

You invoke a static method by using the dot notation to qualify the method call with the name of the object type, such as:

```
type_name.method()
```

See "Static Methods" on page 9-20 for information on design considerations.

Constructor Methods

Every object type has a constructor method implicitly defined for it by the system. A constructor method is a function that returns a new instance of the user-defined type and sets up the values of its attributes. The system implicitly defines a constructor function called the attribute value constructor for all object types that have attributes.

Consider the person_typ object type that is defined in Example 2–1 on page 2-2. The name of the constructor method is simply the name of the object type, as shown in the following:

```
person_typ (1, 'John Smith', '1-800-555-1212'),
```

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. For example:

```
CREATE TABLE people_tab OF person_typ;

INSERT INTO people_tab VALUES (
      person_typ(101, 'John Smith', '1-800-555-1212') );
```

You can also define constructor functions of your own called user-defined constructors to create and initialize objects of such types. Attribute value constructors are convenient to use because they already exist, but user-defined constructors have some important advantages with respect to type evolution. See "Advantages of User-Defined Constructors" on page 8-15 for information on user-defined constructors and their advantages. See "Constructor Methods for Collections" on page 3-1 for information on user-defined constructors for collections.

External Implemented Methods

You can use PL/SQL to invoke external subprograms that have been written in other languages. This provides access to the strengths and capabilities of those languages.

> **See Also:** *Oracle Database PL/SQL User's Guide and Reference* for information on external implemented methods

Inheritance in SQL Object Types

Object types enable you to model the real-world entities such as customers and purchase orders that your application works with. But this is just the first step in exploiting the capabilities of objects. With objects, you cannot only model an entity such as a customer, you can also define different specialized types of customers in a type hierarchy under the original type. You can then perform operations on a hierarchy and have each type implement and execute the operation in a special way.

A type hierarchy is a family tree of object types. It consists of a parent base type, called a supertype, and one or more levels of child object types, called subtypes, derived from the parent.

Subtypes in a hierarchy are connected to their supertypes by inheritance. Therefore, subtypes automatically acquire the attributes and methods of their parent type. Also, subtypes automatically acquire any changes made to these attributes or methods in the parent: any attributes or methods updated in a supertype are updated in subtypes as well.

A subtype becomes a specialized version of the parent type by adding new attributes and methods to the set inherited from the parent or by redefining inherited methods. Redefining an inherited methods gives a subtype its own way of executing the method. Add to this that an object instance of a subtype can generally be substituted for an object instance of any of its supertypes in code, and you have polymorphism.

Polymorphism is the ability of a value in code to contain a value of either a certain declared type or any of a range of the declared type's subtypes. A method called on whatever value occupies the slot may execute differently depending on the value's type because the various types might implement the method differently.

The topics described in this section are:

- Types and Subtypes
- FINAL and NOT FINAL Types and Methods
- Creating Subtypes With Overriding Methods
- NOT INSTANTIABLE Types and Methods
- Inheriting, Overloading, and Overriding Methods
- Dynamic Method Dispatch

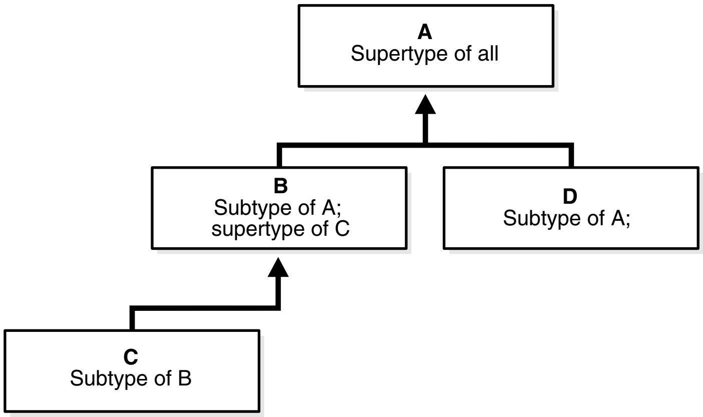
- Substituting Types in a Type Hierarchy
- Column and Row Substitutability
- Creating Subtypes After Creating Substitutable Columns
- Dropping Subtypes After Creating Substitutable Columns
- Turning Off Substitutability in a New Table
- Constraining Substitutability
- Modifying Substitutability
- Restrictions on Modifying Substitutability
- Assignments Across Types
- Comparisons of Objects, REF Variables, and Collections

Types and Subtypes

A subtype can be derived from a supertype either directly or indirectly through intervening levels of other subtypes. A subtype can directly derive only from a single supertype: it cannot derive jointly from more than one. A supertype can have multiple sibling subtypes, but a subtype can have at most one direct parent supertype. In other words, Oracle supports only single inheritance, not multiple inheritance.

A subtype is derived from a supertype by defining a specialized variant of the supertype. For example, from a `person_typ` object type you might derive the specialized types `student_typ` and `employee_typ`. Each of these subtypes is still a `person_typ`, but a special kind of person. What makes a subtype special and distinguishes it from its parent supertype is some change made in the subtype to the attributes or methods that the subtype received from its parent.

Figure 2–1 Subtypes



An object type's attributes and methods make the type what it is: they are its essential, defining features. If a `person_typ` object type has the three attributes `idno`, `name`, and `phone` and the method `get_idno()`, then any object type that is derived from `person_typ` will have these same three attributes and a method `get_idno()`. A subtype is a special case of its parent type, not a totally different kind of thing. As such, it shares with its parent type the features that make the general type what it is.

You can specialize the attributes or methods of a subtype in these ways:

- Add new attributes that its parent supertype does not have.

For example, you might specialize student_typ as a special kind of person_typ by adding to its definition an attribute for major. A subtype cannot drop or change the type of an attribute it inherited from its parent; it can only add new attributes.

- Add entirely new methods that the parent does not have.
- Change the implementation of some of the methods a subtype inherits from its parent so that the subtype's version executes different code from the parent's.

  For example, a shape object type might define a method calculate_area(). Two subtypes of shape, rectilinear_shape, and circular_shape, might each implement this method in a different way. See "Inheriting, Overloading, and Overriding Methods" on page 2-19.

Attributes and methods that a subtype gets from its parent type are said to be inherited. This means more than just that the attributes and methods are patterned on the parent's when the subtype is defined. With object types, the inheritance link remains live. Any changes made later to the parent type's attributes or methods are also inherited, thus, the changes are reflected in the subtype as well. Unless a subtype reimplements an inherited method, it always contains the same core set of attributes and methods that are in the parent type, plus any attributes and methods that it adds.

Remember, a child type is not a different type from its parent; it is a particular kind of that type. If the general definition of person_typ ever changes, the definition of student_typ changes also.

The inheritance relationship that holds between a supertype and its subtypes is the source of both much of the power of objects and much of their complexity. It is a very powerful feature to be able to change a method in a supertype and have the change take effect in all the subtypes downstream just by recompiling. But this same capability means that you have to think about such things as whether you want to allow a type to be specialized or a method to be redefined. Similarly, it is a powerful feature for a table or column to be able to contain any type in a hierarchy, but then you must decide whether to allow this in a particular case, and you may need to constrain DML statements and queries so that they pick out from the type hierarchy just the range of types that you want. The following sections address these aspects of working with objects.

FINAL and NOT FINAL Types and Methods

The definition of an object type determines whether subtypes can be derived from that type. To permit subtypes, the object type must be defined as not final. This is done by including the NOT FINAL keyword in its type declaration, as shown in Example 2–11.

Example 2–11 Creating the person_typ Object Type as NOT FINAL

```
CREATE TYPE person_typ AS OBJECT (
   idno           NUMBER,
   name           VARCHAR2(30),
   phone          VARCHAR2(20))
NOT FINAL;
/
```

The preceding statement declares person_typ to be a not final type so that subtypes of person_typ can be defined. By default, an object type is declared as final and subtypes cannot be derived from it.

You can change a final type to a not final type and vice versa with an ALTER TYPE statement. For example, the following statement changes person_typ to a final type:

```
ALTER TYPE person_typ FINAL;
```

You can alter a type from NOT FINAL to FINAL only if the target type has no subtypes.

Methods can also be declared to be final or not final. If a method is declared to be final, subtypes cannot override it by providing their own implementation. Unlike types, methods are not final by default and must be explicitly declared to be final.

Example 2–12 creates a not final type containing a final member function.

Example 2–12 Creating an Object Type as NOT FINAL with a FINAL Member Function

```
CREATE TYPE person_typ AS OBJECT (
   idno           NUMBER,
   name           VARCHAR2(30),
   phone          VARCHAR2(20),
   FINAL MAP MEMBER FUNCTION get_idno RETURN NUMBER)
NOT FINAL;
/
```

See "Redefining Methods" on page 2-20.

Creating Subtypes With Overriding Methods

You can create a subtype using a CREATE TYPE statement that specifies the immediate parent of the subtype with an UNDER clause.

Creating a Parent or Supertype Object

The creation of the parent or supertype person_typ object is provided in Example 2–13. Subtype definitions for this object are provided in Example 2–14, Example 2–17, and Example 2–18.

Note the methods that are created in the supertype body of Example 2–13. In the subtype examples, the show() function of the parent type is overridden to specifications for each subtype.

Example 2–13 Creating the Parent or Supertype person_typ Object

```
CREATE  TYPE person_typ AS OBJECT (
 idno           NUMBER,
 name           VARCHAR2(30),
 phone          VARCHAR2(20),
 MAP MEMBER FUNCTION get_idno RETURN NUMBER,
 MEMBER FUNCTION show RETURN VARCHAR2)
 NOT FINAL;
/

CREATE TYPE BODY person_typ AS
 MAP MEMBER FUNCTION get_idno RETURN NUMBER IS
 BEGIN
   RETURN idno;
 END;
-- function that can be overriden by subtypes
 MEMBER FUNCTION show RETURN VARCHAR2 IS
 BEGIN
   RETURN 'Id: ' || TO_CHAR(idno) || ', Name: ' || name;
 END;

END;
/
```

Creating a Subtype Object

A subtype inherits the following:

- All the attributes declared in or inherited by the supertype.
- Any methods declared in or inherited by supertype.

In Example 2–14, the student_typ object is a subtype of person_typ and inherits all the attributes declared in or inherited by person_typ and any methods inherited by or declared in person_typ.

The statement that defines student_typ specializes person_typ by adding two new attributes. New attributes declared in a subtype must have names that are different from the names of any attributes or methods declared in any of its supertypes, higher up in its type hierarchy.

Generalized Invocation

Generalized invocation provides a mechanism to invoke a method of a supertype or a parent type, rather than the specific member method, using the following syntax:

```
(SELF AS person_typ).show
```

The student_typ show method first calls the person_typ show method to do the common actions and then does its own specific action, which is to append '--Major:' to the value returned by the person_typ show method. This way, overriding subtype methods can call corresponding overriding parent type methods to do the common actions before doing their own specific actions.

Example 2–14 Creating a student_typ Subtype Using the UNDER Clause

```
CREATE  TYPE student_typ UNDER person_typ (
   dept_id NUMBER,
   major VARCHAR2(30),
   OVERRIDING MEMBER FUNCTION show RETURN VARCHAR2)
   NOT FINAL;
/

CREATE  TYPE BODY student_typ AS
 OVERRIDING MEMBER FUNCTION show RETURN VARCHAR2 IS
 BEGIN
    RETURN (self AS person_typ).show || ' -- Major: ' || major ;
 END;

END;
/
```

Methods are invoked just like normal member methods, except that the type name after AS should be the type name of the parent type of the type that the expression evalutes to.

In Example 2–15, there is an implicit SELF argument just like the implicit self argument of a normal member method invocation. In this case, it invokes the person_typ show method rather than the specific student_typ show method.

Example 2–15 Using Generalized Invocation

```
DECLARE
 myvar student_typ := student_typ(100, 'Sam', '6505556666', 100, 'Math');
 name VARCHAR2(100);
BEGIN
```

```
 name := (myvar AS person_typ).show; --Generalized invocation
END;
/
```

Generalized expression, like member method invocation, is also supported when a method is invoked with an explicit self argument.

Example 2–16 Using Generalized Expression

```
DECLARE
 myvar student_typ := student_typ(100, 'Sam', '6505556666', 100, 'Math');
 name VARCHAR2(100);
BEGIN
 name := person_typ.show(myvar AS person_typ)); -- Generalized expression
END;
/
```

Note: Constructor methods cannot be invoked using this syntax. Also, the type name that appears after AS in this syntax should be one of the parent types of the type of the expression for which method is being invoked.

This syntax can only be used to invoke corresponding overriding member methods of the parent types.

Multiple Subtypes

A type can have multiple child subtypes, and these can also have subtypes. Example 2–17 creates another subtype employee_typ under person_typ.

Example 2–17 Creating an employee_typ Subtype Using the UNDER Clause

```
CREATE TYPE employee_typ UNDER person_typ (
    emp_id NUMBER,
    mgr VARCHAR2(30),
    OVERRIDING MEMBER FUNCTION show RETURN VARCHAR2);
/

CREATE TYPE BODY employee_typ AS
  OVERRIDING MEMBER FUNCTION show RETURN VARCHAR2 IS
  BEGIN
    RETURN ( (SELF AS person_typ).show|| ' -- Employee Id: '
           || TO_CHAR(emp_id) || ', Manager: ' || mgr ;
  END;

END;
/
```

A subtype can be defined under another subtype. Again, the new subtype inherits all the attributes and methods that its parent type has, both declared and inherited. Example 2–18 defines a new subtype part_time_student_typ under student_typ. The new subtype inherits all the attributes and methods of student_typ and adds another attribute.

Example 2–18 Creating a part_time_student_typ Subtype Using the UNDER Clause

```
CREATE TYPE part_time_student_typ UNDER student_typ (
  number_hours NUMBER,
  OVERRIDING MEMBER FUNCTION show RETURN VARCHAR2);
/

CREATE TYPE BODY part_time_student_typ AS
  OVERRIDING MEMBER FUNCTION show RETURN VARCHAR2 IS
  BEGIN
    RETURN ( (SELF AS person_typ).show|| ' -- Major: ' || major ||
           ', Hours: ' || TO_CHAR(number_hours);
  END;

END;
/
```

Creating a Table that Contains Supertype and Subtype Objects

You can create a table that contains the supertype and subtypes and populate the table as shown with the `person_obj_table` in Example 2–19.

Example 2–19 Inserting Values into Substitutable Rows of an Object Table

```
CREATE TABLE person_obj_table OF person_typ;

INSERT INTO person_obj_table
  VALUES (person_typ(12, 'Bob Jones', '111-555-1212'));

INSERT INTO person_obj_table
  VALUES (student_typ(51, 'Joe Lane', '1-800-555-1312', 12, 'HISTORY'));

INSERT INTO person_obj_table
  VALUES (employee_typ(55, 'Jane Smith', '1-800-555-7765',
                      100, 'Jennifer Nelson'));

INSERT INTO person_obj_table
  VALUES (part_time_student_typ(52, 'Kim Patel', '1-800-555-1232', 14,
         'PHYSICS', 20));
```

You can call the `show()` function for the supertype and subtypes in the table with the following:

```
SELECT p.show() FROM person_obj_table p;
```

With the table populated as illustrated in Example 2–19, the output is similar to:

Id: 12, Name: Bob Jones
Id: 51, Name: Joe Lane -- Major: HISTORY
Id: 55, Name: Jane Smith -- Employee Id: 100, Manager: Jennifer Nelson
Id: 52, Name: Kim Patel -- Major: PHYSICS, Hours: 20

Note that data that the `show()` method displays depends on whether the object is a supertype or subtype, and if the `show()` method of the subtype is overridden. For example, Bob Jones is a `person_typ`, that is, an supertype. Only his name and Id are displayed. For Joe Lane, a `student_typ`, his name and Id are provided by the `show()` function of the supertype, and his major is provided by the overridden `show()` function of the subtype.

NOT INSTANTIABLE Types and Methods

A type can be declared to be `NOT INSTANTIABLE`. If a type is not instantiable, there is no constructor (default or user-defined) for it, and you cannot instantiate instances of that type (objects, in other words). You might use this option with types that you intend to use solely as supertypes of specialized subtypes that you do instantiate. The following pseudocode provides an example.

```
CREATE TYPE address_typ AS OBJECT(...)
    NOT INSTANTIABLE NOT FINAL;
CREATE TYPE USaddress_typ UNDER address_typ(...);
CREATE TYPE Intladdress_typ UNDER address_typ(...);
```

A method can also be declared to be not instantiable. Use this option when you want to declare a method in a type without implementing the method there. A type that contains a non-instantiable method must itself be declared not instantiable, as shown in Example 2–20.

Example 2–20 Creating an Object Type that is NOT INSTANTIABLE

```
CREATE TYPE person_typ AS OBJECT (
  idno           NUMBER,
  name           VARCHAR2(30),
  phone          VARCHAR2(20),
  NOT INSTANTIABLE MEMBER FUNCTION get_idno RETURN NUMBER)
  NOT INSTANTIABLE NOT FINAL;
/
```

A non-instantiable method serves as a placeholder. You might define a non-instantiable method when you expect every subtype to override the method in a different way. In such a case, there is no point in defining the method in the supertype.

If a subtype does not provide an implementation for every inherited non-instantiable method, the subtype itself, like the supertype, must be declared not instantiable. A non-instantiable subtype can be defined under an instantiable supertype.

You can alter an instantiable type to a non-instantiable type and vice versa with an ALTER TYPE statement. In the following example, the ALTER TYPE statement makes person_typ instantiable:

Example 2–21 Altering an Object Type to INSTANTIABLE

```
CREATE TYPE person_typ AS OBJECT (
  idno           NUMBER,
  name           VARCHAR2(30),
  phone          VARCHAR2(20))
  NOT INSTANTIABLE NOT FINAL;
/
ALTER TYPE person_typ INSTANTIABLE;
```

You can alter an instantiable type to a non-instantiable type only if the type has no columns, views, tables, or instances that reference that type, either directly, or indirectly through another type or subtype.

You cannot declare a non-instantiable type to be FINAL, which would be pointless anyway.

Inheriting, Overloading, and Overriding Methods

A subtype automatically inherits all methods (both member and static methods) declared in or inherited by its supertype.

A subtype can redefine methods it inherits, and it can also add new methods. It can even add new methods that have the same names as methods it inherits, such that the subtype ends up containing more than one method with the same name.

Giving a type multiple methods with the same name is called method overloading. Redefining an inherited method to customize its behavior in a subtype is either overriding, in the case of member methods, or hiding, in the case of static methods.

See the examples in "Creating Subtypes With Overriding Methods" on page 2-15 and Example 8–7 on page 8-16.

Overloading Methods

Overloading is useful when you want to provide a variety of ways of doing something. For example, a shape object might overload a `draw()` method with another `draw()` method that adds a text label to the drawing and contains an argument for the label's text.

When a type has several methods with the same name, the compiler uses the methods' signatures to tell them apart. A method's signature is a sort of structural profile. It consists of the method's name and the number, types, and order of the method's formal parameters, including the implicit `self` parameter. Methods that have the same name, but different signatures are called overloads when they exist in the same user-defined type.

In the following pseudocode, Subtype `MySubType_typ` creates an overload of `draw()`:

```
CREATE TYPE MyType_typ AS OBJECT (...,
  MEMBER PROCEDURE draw(x NUMBER), ...) NOT FINAL;

CREATE TYPE MySubType_typ UNDER MyType_typ (...,
  MEMBER PROCEDURE draw(x VARCHAR2(20)),
  STATIC FUNCTION bar(...)...
  ...);
```

`MySubType_typ` contains two versions of `draw()`. One is an inherited version with a `NUMBER` parameter and the other has a `VARCHAR2` parameter.

Redefining Methods

Overriding and hiding redefine an inherited method to make it do something different in the subtype. For example, a subtype `circular_shape` derived from a `shape` supertype might override a member method `calculate_area()` to customize it specifically for calculating the area of a circle. For examples of overriding methods, see "Creating Subtypes With Overriding Methods" on page 2-15.

Redefining a method is called overriding when the method that is redefined is a member method; redefining is called hiding when the redefined method is a static method. Overriding and hiding are similar in that, in either case, the version of the method redefined in the subtype eclipses an inherited version of the same name and signature such that the new version is executed instead of the inherited one whenever an instance of the subtype invokes the method. If the subtype itself has subtypes, these inherit the redefined method instead of the original version.

However, with overriding, the system relies on type information contained in the member method's implicit self argument to dynamically choose the correct version of the method to execute. With hiding, the correct version can be identified at compile time, and dynamic dispatch is not necessary. See "Dynamic Method Dispatch" on page 2-23.

It is possible that a supertype may contain overloads of a method that is redefined in a subtype. Overloads of a method all have the same name, so the compiler uses the signature of the subtype's method to identify the particular version in the supertype that is superseded. This means that, to override or hide a method, you must preserve its signature.

A subtype that overrides a member method must signal the override with the `OVERRIDING` keyword in the type definition. No such special keyword is required when a subtype hides a static method.

For example, in the following pseudocode, the subtype signals that it is overriding method `Print()`:

```
CREATE TYPE MyType_typ AS OBJECT (...,
  MEMBER PROCEDURE Print(),
  FINAL MEMBER FUNCTION function_mytype(x NUMBER)...
) NOT FINAL;

CREATE TYPE MySubType_typ UNDER MyType_typ (...,
  OVERRIDING MEMBER PROCEDURE Print(),
...);
```

As with new methods, you supply the declaration for a method that hides or overrides in a `CREATE TYPE BODY` statement.

Restrictions on Overriding Methods

The following are restrictions on overriding methods:

- You can override only methods that are not declared to be final in the supertype.
- Order methods may appear only in the root type of a type hierarchy: they may not be redefined (overridden) in subtypes.
- A static method in a subtype may not redefine a member method in the supertype.
- A member method in a subtype may not redefine a static method in the supertype.
- If a method being overridden provides default values for any parameters, then the overriding method must provide the same default values for the same parameters.

How Overloading Works with Inheritance

The overloading algorithm allows substituting a subtype value for a formal parameter that is a supertype. This capability is known as substitutability. If more than one instance of an overloaded procedure matches the procedure call, the following rules apply to determine which procedure is called:

If the only difference in the signatures of the overloaded procedures is that some parameters are object types from the same supertype-subtype hierarchy, the closest match is used. The closest match is one where all the parameters are at least as close as any other overloaded instance, as determined by the depth of inheritance between the subtype and supertype, and at least one parameter is closer.

A semantic error occurs when two overloaded instances match, and some argument types are closer in one overloaded procedure to the actual arguments than in any other instance.

A semantic error also occurs if some parameters are different in their position within the object type hierarchy, and other parameters are of different data types so that an implicit conversion would be necessary.

For example, create a type hierarchy with three levels and then declare two overloaded instances of a function, where the only difference in argument types is their position in this type hierarchy, as shown in Example 2–22. We declare a variable of type `final_t`, then call the overloaded function. The instance of the function that is executed is the one that accepts a `sub_t` parameter, because that type is closer to `final_t` in the hierarchy than `super_t` is.

Example 2–22 Resolving PL/SQL Functions With Inheritance

```
CREATE OR REPLACE TYPE super_t AS OBJECT
  (n NUMBER) NOT final;
/
CREATE OR REPLACE TYPE sub_t UNDER super_t
  (n2 NUMBER) NOT final;
/
CREATE OR REPLACE TYPE final_t UNDER sub_t
  (n3 NUMBER);
/
CREATE OR REPLACE PACKAGE p IS
   FUNCTION func (arg super_t) RETURN NUMBER;
   FUNCTION func (arg sub_t) RETURN NUMBER;
END;
/
CREATE OR REPLACE PACKAGE BODY p IS
   FUNCTION func (arg super_t) RETURN NUMBER IS BEGIN RETURN 1; END;
   FUNCTION func (arg sub_t) RETURN NUMBER IS BEGIN RETURN 2; END;
END;
/

DECLARE
  v final_t := final_t(1,2,3);
BEGIN
  DBMS_OUTPUT.PUT_LINE(p.func(v));  -- prints 2
END;
/
```

In Example 2–22, the choice of which instance to call is made at compile time. In Example 2–23, this choice is made dynamically. We declare `v` as an instance of `super_t`, but because we assign a value of `sub_t` to it, the appropriate instance of the function is called. This feature is known as dynamic dispatch.

Example 2–23 Resolving PL/SQL Functions With Inheritance Dynamically

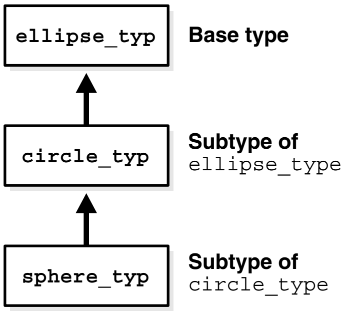
```
CREATE TYPE super_t AS OBJECT
  (n NUMBER, MEMBER FUNCTION func RETURN NUMBER) NOT final;
/
CREATE TYPE BODY super_t AS
 MEMBER FUNCTION func RETURN NUMBER IS BEGIN RETURN 1; END; END;
/
CREATE OR REPLACE TYPE sub_t UNDER super_t
  (n2 NUMBER,
   OVERRIDING MEMBER FUNCTION func RETURN NUMBER) NOT final;
/
CREATE TYPE BODY sub_t AS
 OVERRIDING MEMBER FUNCTION func RETURN NUMBER IS BEGIN RETURN 2; END; END;
/
CREATE OR REPLACE TYPE final_t UNDER sub_t
  (n3 NUMBER);
/
```

```
DECLARE
  v super_t := final_t(1,2,3);
BEGIN
  DBMS_OUTPUT.PUT_LINE(v.func); -- prints 2
END;
/
```

Dynamic Method Dispatch

As a result of method overriding, a type hierarchy can define multiple implementations of the same method. For example, in a hierarchy of the types `ellipse_typ`, `circle_typ`, `sphere_typ`, each type might define a method `calculate_area()` differently.

Figure 2–2 Hierarchy of Types



When such a method is invoked, the type of the object instance that invokes it is used to determine which implementation of the method to use. The call is then dispatched to that implementation for execution. This process of selecting a method implementation is called virtual or dynamic method dispatch because it is done at run time, not at compile time.

A method call is dispatched to the nearest implementation, working back up the inheritance hierarchy from the current or specified type. If the call invokes a member method of an object instance, the type of that instance is the current type, and the implementation defined or inherited by that type is used. If the call invokes a static method of a type, the implementation defined or inherited by that specified type is used.

For example, if `c1` is an object instance of `circle_typ`, `c1.proc()` looks first for an implementation of `proc()` defined in `circle_typ`. If none is found, it looks up the supertype chain for an implementation in `ellipse_typ`. The fact that `sphere_typ` also defines an implementation is irrelevant because the type hierarchy is searched only upwards, toward the top. Subtypes of the current type are not searched.

Similarly, a call to a static method `circle_typ.bar()` looks first in `circle_typ` and then, if necessary, in the supertype(s) of `circle_typ`. The subtype `sphere_typ` is not searched.

> **See Also:** *Oracle Database PL/SQL User's Guide and Reference* for information on how subprograms calls are resolved and the dynamic dispatch feature

Substituting Types in a Type Hierarchy

In a type hierarchy, the subtypes are variant kinds of the root, base type. For example, a student_typ type and an employee_typ are kinds of a person_typ. The base type includes these other types.

When you work with types in a type hierarchy, sometimes you want to work at the most general level and, for example, select or update all persons. But sometimes you want to select or update only students, or only persons who are not students.

The (polymorphic) ability to select all persons and get back not only objects whose declared type is person_typ but also objects whose declared subtype is student_typ or employee_typ is called substitutability. A supertype is substitutable if one of its subtypes can substitute or stand in for it in a variable or column whose declared type is the supertype.

In general, types are substitutable. Object attributes, collection elements and REFs are substitutable. An attribute defined as a REF, type, or collection of type person_typ can hold a REF to, an instance of, or instances of an instance of person_typ or an instance of any subtype of person_typ.

This is what you would expect, given that a subtype is, after all, just a specialized kind of any of its supertypes. Formally, though, a subtype is a type in its own right: it is not the same type as its supertype. A column that holds all persons, including all persons who are students and all persons who are employees, actually holds data of multiple types.

Substitutability comes into play in attributes, columns, and rows (of an object view or object table) declared to be an object type, a REF to an object type, or a collection type.

In principle, object attributes, collection elements and REFs are always substitutable: there is no syntax at the level of the type definition to constrain their substitutability to some subtype. You can, however, turn off or constrain substitutability at the storage level, for specific tables and columns. See "Turning Off Substitutability in a New Table" on page 2-27 and "Constraining Substitutability" on page 2-28.

Column and Row Substitutability

Object type columns are substitutable, and so are object-type rows in object tables and views. In other words, a column or row defined to be of type t can contain instances of t and any of its subtypes.

For example, consider the person_typ type hierarchy introduced in "Creating Subtypes With Overriding Methods" on page 2-15. An object table of person_typ can contain rows of all three types. You insert an instance of a given type using the constructor for that type in the VALUES clause of the INSERT statement as shown in Example 2–19 on page 2-18.

Similarly, in a relational table or view, a substitutable column of type person_typ can contain instances of all three types. The following example inserts a person, a student, and a part-time student in the person_typ column contact:

Example 2–24   Inserting Values into Substitutable Columns of a Table

```
CREATE TABLE contacts (
  contact         person_typ,
  contact_date    DATE );

INSERT INTO contacts
  VALUES (person_typ (12, 'Bob Jones', '111-555-1212'), '24 Jun 2003' );
```

```
INSERT INTO contacts
  VALUES (student_typ(51, 'Joe Lane', '1-800-555-1312', 12, 'HISTORY'),
         '24 Jun 2003' );

INSERT INTO contacts
  VALUES (part_time_student_typ(52, 'Kim Patel', '1-800-555-1232', 14,
          'PHYSICS', 20), '24 Jun 2003' );
```

A newly created subtype can be stored in any substitutable tables and columns of its supertype, including tables and columns that existed before the subtype was created.

Attributes in general can be accessed using the dot notation. Attributes of a subtype of a row or column's declared type can be accessed with the TREAT function. For example:

```
SELECT TREAT(contact AS student_typ).major FROM contacts;
```

See "TREAT" on page 2-35.

Using OBJECT_VALUE and OBJECT_ID with Substitutable Rows

The OBJECT_VALUE and OBJECT_ID pseudocolumns allow you to access and identify the value and OID of a substitutable row in an object table as shown in Example 2–25.

Example 2–25 Using OBJECT_VALUE and OBJECT_ID

```
CREATE TABLE person_obj_table OF person_typ;

INSERT INTO person_obj_table
  VALUES (person_typ(20, 'Bob Jones', '111-555-1212'));

SELECT p.object_id, p.object_value FROM person_obj_table p;
```

Subtypes Having Supertype Attributes

A subtype can have an attribute whose type is the type of a supertype. For example:

Example 2–26 Creating a Subtype with a Supertype Attribute

```
CREATE TYPE student_typ UNDER person_typ (
    dept_id   NUMBER,
    major     VARCHAR2(30),
    advisor   person_typ);
/
```

However, columns of such types are not substitutable. Similarly, a subtype ST can have a collection attribute whose element type is one of ST's supertypes, but, again, columns of such types are not substitutable. For example, if student_typ had a nested table or varray of person_typ, the student_typ column would not be substitutable.

You can, however, define substitutable columns of subtypes that have REF attributes that reference supertypes. For example, the composite_category_typ subtype shown in Example 2–27 contains the subcategory_ref_list nested table. This table contains subcategory_ref_list_typ which are REFs to category_typ. The subtype was created as follows:

Example 2–27 Defining Columns of Subtypes that have REF Attributes

```
-- not to be executed
```

```
CREATE TYPE subcategory_ref_list_typ
  AS TABLE OF REF category_typ;
/

CREATE TYPE composite_category_typ
  UNDER category_typ
    (
      subcategory_ref_list subcategory_ref_list_typ
...
```

See "Turning Off Substitutability in a New Table" on page 2-27.

REF Columns and Attributes

REF columns and attributes are substitutable in both views and tables. For example, in either a view or a table, a column declared to be REF person_typ can hold references to instances of person_typ or any of its subtypes.

Collection Elements

Collection elements are substitutable in both views and tables. For example, a nested table of person_typ can contain object instances of person_typ or any of its subtypes.

Creating Subtypes After Creating Substitutable Columns

If you create a subtype, any table that already has substitutable columns of the supertype is automatically enabled to store the new subtype as well. This means that your options for creating subtypes are affected by the existence of such tables. If such a table exists, you can only create subtypes that are substitutable, that is, subtypes that Oracle can enable that table to store.

The following example shows an attempt to create a subtype student_typ under person_typ.

Example 2–28 Creating a Subtype After Creating Substitutable Columns

```
CREATE TYPE person_typ AS OBJECT (
  idno           NUMBER,
  name           VARCHAR2(30),
  phone          VARCHAR2(20))
  NOT FINAL;
/

CREATE TYPE employee_typ UNDER person_typ (
    emp_id NUMBER,
    mgr VARCHAR2(30));
/

CREATE TABLE person_obj_table (p person_typ);
```

The following statement fails because student_typ has a supertype attribute, and table person_obj_table has a substitutable column p of the supertype.

```
CREATE TYPE student_typ UNDER person_typ ( -- incorrect CREATE subtype
    advisor person_typ);
/
```

The following attempt succeeds. This version of the student_typ subtype is substitutable. Oracle automatically enables table person_obj_table to store instances of this new type.

```
CREATE TYPE student_typ UNDER person_typ (
    dept_id NUMBER,
    major VARCHAR2(30));
/
INSERT INTO person_obj_table
  VALUES (student_typ(51, 'Joe Lane', '1-800-555-1312', 12, 'HISTORY'));
```

Dropping Subtypes After Creating Substitutable Columns

You can drop a subtype with the `VALIDATE` option only if no instances of the subtype are stored in any substitutable column of the supertype.

For example, the following statement fails because an instance of `student_typ` is stored in substitutable column `p` of table `person_obj_table`:

```
DROP TYPE student_typ VALIDATE -- incorrect DROP TYPE ;
```

To drop the type, first delete any of its instances in substitutable columns of the supertype:

```
DELETE FROM person_obj_table WHERE p IS OF (student_typ);

DROP TYPE student_typ VALIDATE;
```

Turning Off Substitutability in a New Table

When creating a table, you can turn off all substitutability on a column or attribute, including embedded attributes and collections nested to any level, with the clause `NOT SUBSTITUTABLE AT ALL LEVELS`.

In the following example, the clause confines column `office` of a relational table to storing only `office_typ` instances and disallows any subtype instances:

Example 2–29 Turning off Substitutability When Creating a Table

```
CREATE TYPE office_typ AS OBJECT (
  office_id    VARCHAR(10),
  location     location_typ,
  occupant     person_typ )
  NOT FINAL;
/

CREATE TABLE dept_office (
  dept_no      NUMBER,
  office       office_typ)
  COLUMN office NOT SUBSTITUTABLE AT ALL LEVELS;
```

With object tables, the clause can be applied to the table as a whole, like this:

```
CREATE TABLE office_tab OF office_typ
  NOT SUBSTITUTABLE AT ALL LEVELS;
```

Alternatively, the clause can also be applied to turn off substitutability in a particular column that is, for a particular attribute of the object type of the table:

```
CREATE TABLE office_tab OF office_typ
  COLUMN occupant NOT SUBSTITUTABLE AT ALL LEVELS;
```

You can specify that the element type of a collection is not substitutable using syntax like the following:

```
CREATE TABLE people_tab (
    people_column people_typ )
    NESTED TABLE people_column
      NOT SUBSTITUTABLE AT ALL LEVELS STORE AS people_column_nt;
```

There is no mechanism to turn off substitutability for REF columns.

Constraining Substitutability

You can impose a constraint that limits the range of subtypes permitted in an object column or attribute to a particular subtype in the declared type's hierarchy. You do this using an IS OF *type* constraint.

For example, the following statement creates a table of office_typ in which occupants are constrained to just those persons who are employees:

Example 2–30 Constraining Substitutability When Creating a Table

```
CREATE TABLE office_tab OF office_typ
  COLUMN occupant IS OF (ONLY employee_typ);
```

Although the type office_typ allows authors to be of type person_typ, the column declaration imposes a constraint to store only instances of employee_typ.

You can only use the IS OF *type* operator to constrain row and column objects to a single subtype (not several), and you must use the ONLY keyword, as in the preceding example.

You can use either IS OF *type* or NOT SUBSTITUTABLE AT ALL LEVELS to constrain an object column, but you cannot use both.

Modifying Substitutability

In an existing table, you can change an object column from SUBSTITUTABLE to NOT SUBSTITUTABLE (or from NOT SUBSTITUTABLE to SUBSTITUTABLE) by using an ALTER TABLE statement. To do so, you specify the clause [NOT] SUBSTITUTABLE AT ALL LEVELS for the particular column.

You can modify substitutability only for a specific column; you cannot modify substitutability for an object table as a whole.

The following statement makes column office substitutable:

Example 2–31 Modifying Substitutability in a Table

```
ALTER TABLE dept_office
  MODIFY COLUMN office SUBSTITUTABLE AT ALL LEVELS;
```

The following statement makes the column not substitutable. Notice that it also uses the FORCE keyword. This keyword causes any hidden columns containing typeid information or data for subtype attributes to be dropped:

```
ALTER TABLE  dept_office
  MODIFY COLUMN office NOT SUBSTITUTABLE AT ALL LEVELS FORCE;
```

If the FORCE keyword is not used when a column is made not substitutable, the column and all attributes of the type must be FINAL or the ALTER TABLE statement will fail.

A VARRAY column can be modified from SUBSTITUTABLE to NOT SUBSTITUTABLE only if the element type of the varray is final itself and has no embedded types (in its attributes or in their attributes, and so on) that are not final.

See "Hidden Columns for Substitutable Columns and Tables" on page 8-2 for more information about hidden columns for typeids and subtype attributes.

Restrictions on Modifying Substitutability

You can change the substitutability of only one column at a time with an ALTER TABLE statement. To change substitutability for multiple columns, you must issue multiple statements.

In an object table, you can modify substitutability for a column only if substitutability was not explicitly set at the table level, for the entire table, when the table was created.

For example, the following attempt to modify substitutability for column address succeeds because substitutability has not been explicitly turned on or off at the table level in the CREATE TABLE statement:

```
CREATE TABLE office_tab OF office_typ;

ALTER TABLE office_tab
  MODIFY COLUMN occupant NOT SUBSTITUTABLE AT ALL LEVELS FORCE;
```

However, in the following example, substitutability is explicitly set at the table level, so the attempt to modify the setting for column address fails:

```
CREATE TABLE office_tab OF office_typ
  NOT SUBSTITUTABLE AT ALL LEVELS;

/* Following SQL statement generates an error: */
ALTER TABLE office_tab
  MODIFY COLUMN occupant SUBSTITUTABLE AT ALL LEVELS FORCE  -- incorrect ALTER;
```

A column whose substitutability is already constrained by an IS OF *type* operator cannot have its substitutability modified with a [NOT] SUBSTITUTABLE AT ALL LEVELS clause. See "Constraining Substitutability" on page 2-28 for information about IS OF *type*.

Assignments Across Types

The assignment rules described in this section apply to INSERT/UPDATE statements, the RETURNING clause, function parameters, and PL/SQL variables.

Objects and REFs to Objects

Substitutability is the ability of a subtype to stand in for one of its supertypes. An attempt to perform a substitution in the other direction, to substitute a supertype for a subtype, raises an error at compile time.

An assignment of a source of type source_typ to a target of type target_typ must be of one of the following two patterns:

- Case 1: source_typ and target_typ are the same type
- Case 2: source_typ is a subtype of target_typ (widening)

Case 2 illustrates widening. Widening is an assignment in which the declared type of the source is more specific than the declared type of the target. For example, assigning an employee instance to a variable of person type.

Intuitively, the idea here is that you are regarding an employee as a person. An employee is a more narrowly defined, specialized kind of person, so you can put an employee in a slot meant for a person if you do not mind ignoring whatever extra specialization makes that person an employee. All employees are persons, so a widening assignment always works.

To illustrate widening, suppose that you have the following table:

```
TABLE T(pers_col person_typ, emp_col employee_typ,
        stu_col student_typ)
```

The following assignments show widening. The assignments are valid unless `perscol` has been defined to be not substitutable.

```
UPDATE T set pers_col = emp_col;
```

The following is a PL/SQL example:

```
DECLARE
  var1 person_typ;
  var2 employee_typ;
BEGIN
  var2 := employee_typ(55, 'Jane Smith', '1-800-555-7765', 100, 'Jennifer
Nelson');
  var1 := var2;
END;
/
```

Besides widening, there is also narrowing. Narrowing is the reverse of widening. It involves regarding a more general, less specialized type of thing, such as a person, as a more narrowly defined type of thing, such as an employee. Not all persons are employees, so a particular assignment like this works only if the person in question actually happens to be an employee.

To do a narrowing assignment, you must use the `TREAT` function to test that the source instance of the more general declared type is in fact an instance of the more specialized target type and can therefore be operated on as such. The TREAT function does a runtime check to confirm this and returns NULL if the source value the person in question is not of the target type or one of its subtypes.

For example, the following `UPDATE` statement sets values of `person_typ` in column `perscol` into column `empcol` of `employee_typ`. For each value in `perscol`, the assignment succeeds only if that person is also an employee. If person George is not an employee, `TREAT` returns `NULL`, and the assignment returns `NULL`.

```
UPDATE T set emp_col = TREAT(pers_col AS employee_typ);
```

The following statement attempts to do a narrowing assignment without explicitly changing the declared type of the source value. The statement will return an error:

```
UPDATE T set emp_col = pers_col;
```

See "TREAT" on page 2-35.

Collection Assignments

In assignments of expressions of a collection type, the source and target must be of the same declared type. Neither widening nor narrowing is permitted. However, a subtype value can be assigned to a supertype collection. For example, suppose we have the following collection types:

```
CREATE TYPE person_set AS TABLE OF person_typ;
/

CREATE TYPE student_set AS TABLE OF student_typ;
/
```

Expressions of these different collection types cannot be assigned to each other, but a collection element of `student_typ` can be assigned to a collection of `PersonSet` type:

```
DECLARE
  var1 person_set;
  var2 student_set;
  elem1 person_typ;
  elem2 student_typ;
BEGIN
--  var1 := var2;   /* ILLEGAL - collections not of same type */
  var1 := person_set (elem1, elem2);   /* LEGAL : Element is of subtype */
END;
/
```

Comparisons of Objects, REF Variables, and Collections

This section discusses the comparison operators used in SQL conditions.

> **See Also:** *Oracle Database SQL Language Reference* for information about using SQL conditions

Comparing Object Instances

Two object instances can be compared if, and only if, they are both of the same declared type, or one is a subtype of the other.

Map methods and order methods provide the mechanism for comparing objects. You optionally define one or the other of these in an object type to specify the basis on which you want objects of that type to be compared. If a method of either sort is defined, it is called automatically whenever objects of that type or one of its subtypes need to be compared.

If a type does not define either a map method or an order method, object variables of that type can be compared only in SQL statements and only for equality or inequality. Two objects of the same type count as equal only if the values of their corresponding attributes are equal. See "Methods for Comparing Objects" on page 2-9.

Comparing REF Variables

Two `REF` variables can be compared if, and only if, the targets that they reference are both of the same declared type, or one is a subtype of the other.

Functions and Operators Useful with Objects

Several functions and operators are particularly useful for working with objects and references to objects:

- CAST
- CURSOR
- DEREF
- IS OF type
- REF
- SYS_TYPEID

- TABLE()
- TREAT
- VALUE

Examples are given throughout this book.

In PL/SQL the VALUE, REF and DEREF functions can appear only in a SQL statement. For information about SQL functions, see *Oracle Database SQL Language Reference*.

CAST

CAST converts one built-in data type or collection-typed value into another built-in data type or collection-typed value. For example:

Example 2–32 Using the CAST Function

```
CREATE TYPE person_list_typ AS TABLE OF person_typ;
/

SELECT CAST(COLLECT(contact) AS person_list_typ)
  FROM contacts;
```

For more information about the SQL CAST function, see *Oracle Database SQL Language Reference*.

CURSOR

A CURSOR expression returns a nested cursor. This form of expression is equivalent to the PL/SQL REF CURSOR and can be passed as a REF CURSOR argument to a function.

For more information about the SQL CURSOR expression, see *Oracle Database SQL Language Reference*.

DEREF

The DEREF function in a SQL statement returns the object instance corresponding to a REF. The object instance returned by DEREF may be of the declared type of the REF or any of its subtypes.

For example, the following statement returns person_typ objects from the table contact_ref.

Example 2–33 Using the DEREF Function

```
SELECT DEREF(c.contact_ref), c.contact_date
  FROM contacts_ref c;
```

See "Dereferencing REFs" on page 1-10. For more information about the SQL DEREF function, see *Oracle Database SQL Language Reference*.

IS OF *type*

The IS OF *type* predicate tests object instances for the level of specialization of their type.

For example, the following query retrieves all student instances (including any subtypes of students) stored in the person_obj_table table.

Example 2–34 Using the IS OF type Operator to Query Value of a Subtype

```
SELECT VALUE(p)
  FROM person_obj_table p
WHERE VALUE(p) IS OF (student_typ);
```

For any object that is not of a specified subtype, or a subtype of a specified subtype, `IS OF` returns `FALSE`. Subtypes of a specified subtype are just more specialized versions of the specified subtype. If you want to exclude such subtypes, you can use the `ONLY` keyword. This keyword causes `IS OF` to return `FALSE` for all types except the specified types.

In the following example, the statement tests objects in object table `person_obj_table`, which contains persons, employees, and students, and returns `REF`s just to objects of the two specified person subtypes `employee_typ`, `student_typ`, and their subtypes, if any:

```
SELECT REF(p)
  FROM person_obj_table p
WHERE VALUE(p) IS OF (employee_typ, student_typ);
```

Here is a similar example in PL/SQL. The code does something if the person is an employee or student:

```
DECLARE
  var person_typ;
BEGIN
  var := employee_typ(55, 'Jane Smith', '1-800-555-7765', 100, 'Jennifer Nelson');
  IF var IS OF (employee_typ, student_typ) THEN
     DBMS_OUTPUT.PUT_LINE('Var is an employee_typ or student_typ object.');
  ELSE
     DBMS_OUTPUT.PUT_LINE('Var is not an employee_typ or student_typ object.');
  END IF;
END;
/
```

The following statement returns only students whose most specific or specialized type is `student_typ`. If the table or view contains any objects of a subtype of `student_typ`, such as `part_time_student_typ`, these are excluded. The example uses the `TREAT` function to convert objects that are students to `student_typ` from the declared type of the view, `person_typ`:

```
SELECT TREAT(VALUE(p) AS student_typ)
  FROM person_obj_table p
WHERE VALUE(p) IS OF(ONLY student_typ);
```

To test the type of the object that a `REF` points to, you can use the `DEREF` function to dereference the `REF` before testing with the `IS OF` *`type`* predicate.

For example, if `contact_ref` is declared to be `REF person_typ`, you can get just the rows for students as follows:

```
SELECT *
  FROM contacts_ref
WHERE DEREF(contact_ref) IS OF (student_typ);
```

For more information about the SQL `IS OF` *`type`* condition, see *Oracle Database SQL Language Reference*.

REF

The REF function in a SQL statement takes as an argument a correlation name for an object table or view and returns a reference (a REF) to an object instance from that table or view. The REF function may return references to objects of the declared type of the table, view, or any of its subtypes. For example, the following statement returns the references to all persons, including references to students and employees, whose idno attribute is 12:

Example 2–35   Using the REF Function

```
SELECT REF(p)
  FROM person_obj_table p
  WHERE p.idno = 12;
```

For more information about the SQL REF function, see *Oracle Database SQL Language Reference*.

SYS_TYPEID

The SYS_TYPEID function can be used in a query to return the typeid of the most specific type of the object instance passed to the function as an argument.

The most specific type of an object instance is the type to which the instance belongs that is farthest removed from the root type. For example, if Tim is a part-time student, he is also a student and a person, but his most specific type is part-time student.

The function returns the typeids from the hidden type discriminant column that is associated with every substitutable column. The function returns a null typeid for a final, root type.

The syntax of the function is:

```
SYS_TYPEID(object_type_value)
```

Function SYS_TYPEID may be used only with arguments of an object type. Its primary purpose is to make it possible to build an index on a hidden type discriminant column.

All types that do belong to a type hierarchy are assigned a non-null typeid that is unique within the type hierarchy. Types that do not belong to a type hierarchy have a null typeid.

Every type except a final root type belongs to a type hierarchy. A final root type has no types related to it by inheritance:

- It cannot have subtypes derived from it because it is final
- It is not itself derived from some other type because it is a root type, so it does not have any supertypes.

See "Hidden Columns for Substitutable Columns and Tables" on page 8-2 for more information about type discriminant columns.

For an example of SYS_TYPEID, consider the substitutable object table person_obj_table, of person_typ. person_typ is the root type of a hierarchy that has student_typ as a subtype and part_time_student_typ as a subtype of student_typ. See Example 2–19 on page 2-18.

The following query uses SYS_TYPEID. It gets the name attribute and typeid of the object instances in the person_obj_table table. Each of the instances is of a different type:

Example 2–36 Using the SYS_TYPEID Function

```
SELECT name, SYS_TYPEID(VALUE(p)) typeid FROM person_obj_table p;
```

See "Hidden Columns for Substitutable Columns and Tables" on page 8-2 for information about the type discriminant and other hidden columns. For more information about the SQL SYS TYPEID function, see *Oracle Database SQL Language Reference*.

TABLE()

Table functions are functions that produce a collection of rows, a nested table or a varray, that can be queried like a physical database table or assigned to a PL/SQL collection variable. You can use a table function like the name of a database table, in the FROM clause of a query, or like a column name in the SELECT list of a query.

A table function can take a collection of rows as input. An input collection parameter can be either a collection type, such as a VARRAY or a PL/SQL table, or a REF CURSOR.

Use PIPELINED to instruct Oracle to return the results of a table function iteratively. A table function returns a nested table or varray collection type. You query table functions by using the TABLE keyword before the function name in the FROM clause of the query.

For information on TABLE() functions, see *Oracle Database Data Cartridge Developer's Guide* and *Oracle Database PL/SQL User's Guide and Reference*.

TREAT

The TREAT function does a runtime check to confirm that an expression can be operated on as if it were of a different specified type in the hierarchy normally, a subtype of the expression s declared type. In other words, the function attempts to treat a supertype instance as a subtype instance to treat a person as a student, for example. Whether this can be done in a given case depends on whether the person in question actually is a student (or student subtype, such as a part-time student). If the person is a student, then the person is returned as a student, with the additional attributes and methods that a student may have. If the person happens not to be a student, TREAT returns NULL in SQL.

The two main uses of TREAT are:

- In narrowing assignments, to modify the type of an expression so that the expression can be assigned to a variable of a more specialized type in the hierarchy: in other words, to set a supertype value into a subtype.
- To access attributes or methods of a subtype of the declared type of a row or column

A substitutable object table or column of type T has a hidden column for every attribute of every subtype of T. These hidden columns are not listed by a DESCRIBE statement, but they contain subtype attribute data. TREAT enables you to access these columns.

The following example shows TREAT used in an assignment where a column of person type is set into a column of employee type. For each row in perscol, TREAT returns an employee type or NULL, depending on whether the given person happens to be an employee.

```
UPDATE T set empcol = TREAT(perscol AS employee_typ);
```

In the next example, TREAT returns all (and only) student_typ instances from person_obj_table of type person_typ, a supertype of student_typ. The statement uses TREAT to modify the type of p from person_typ to student_typ.

Example 2–37 Using the TREAT Function to Return a Specific Subtype in a Query

```
SELECT TREAT(VALUE(p) AS student_typ)
  FROM person_obj_table p;
```

For each p, the TREAT modification succeeds only if the most specific or specialized type of the value of p is student_typ or one of its subtypes. If p is a person who is not a student, or if p is NULL, TREAT returns NULL in SQL or, in PL/SQL, raises an exception.

You can also use TREAT to modify the declared type of a REF expression. For example:

```
SELECT TREAT(REF(p) AS REF student_typ)
  FROM person_obj_table p;
```

The previous example returns REFs to all student_typ instances. In SQL it returns NULL REFs for all person instances that are not students, and in PL/SQL it raises an exception.

Perhaps the most important use of TREAT is to access attributes or methods of a subtype of a row or column's declared type. The following query retrieves the major attribute of all persons, students and part-time students, who have this attribute. NULL is returned for persons who are not students:

Example 2–38 Using the TREAT Function to Access Attributes of a Specific Subtype

```
SELECT name, TREAT(VALUE(p) AS student_typ).major major
  FROM person_obj_table p;
```

The following query will not work because major is an attribute of student_typ but not of person_typ, the declared type of table persons:

```
SELECT name, VALUE(p).major major FROM person_obj_table p -- incorrect;
```

The following is a PL/SQL example:

```
DECLARE
  var person_typ;
BEGIN
  var := employee_typ(55, 'Jane Smith', '1-800-555-7765', 100, 'Jennifer Nelson');
  DBMS_OUTPUT.PUT_LINE(TREAT(var AS employee_typ).mgr);
END;
/
```

See "Assignments Across Types" on page 2-29 for information on using TREAT in assignments. For more information about the SQL TREAT function, see *Oracle Database SQL Language Reference*.

VALUE

In a SQL statement, the VALUE function takes as its argument a correlation variable (table alias) for an object table or object view and returns object instances corresponding to rows of the table or view. The VALUE function may return instances of the declared type of the row or any of its subtypes. InExample 2–39 the query returns all persons, including students and employees, from table person_obj_table of person_typ.

Example 2–39 Using the VALUE Function

```
SELECT VALUE(p) FROM person_obj_table p;
```

To retrieve only part time students, that is, instances whose most specific type is `part_time_student_typ`, use the `ONLY` keyword to confine the selection:

```
SELECT VALUE(p) FROM person_obj_table p
  WHERE VALUE(p) IS OF (ONLY part_time_student_typ);
```

In the following example, `VALUE` is used to update a object instance in an object table:

```
UPDATE person_obj_table p
   SET VALUE(p) = person_typ(12, 'Bob Jones', '1-800-555-1243')
   WHERE p.idno = 12;
```

See also Example 3–21, "Using VALUE to Update a Nested Table" on page 3-13. For more information about the SQL `VALUE` function, see *Oracle Database SQL Language Reference*.

3

Support for Collection Data Types

This chapter provides basic information about working with varrays and nested table collection data types. It explains how to create and manage collection data types.

This chapter contains these topics:

- Creating Collection Data Types
- Operations on Collection Data Types

Creating Collection Data Types

Oracle supports the varray and nested table collection data types.

- A varray is an ordered collection of elements
- A nested table can have any number of elements

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

If you need to run efficient queries on a collection, handle arbitrary numbers of elements, or perform mass insert, update, or delete operations, then use a nested table. See "Design Considerations for Collections" on page 9-8.

Creating an Instance of a VARRAY or Nested Table

You create an instance of a collection type in the same way that you create an instance of any other object type, namely, by calling the type's constructor method. The name of a type's constructor method is simply the name of the type. You specify the elements of the collection as a comma-delimited list of arguments to the method.

Calling a constructor method with an empty list creates an empty collection of that type. Note that an empty collection is an actual collection that happens to be empty; it is not the same as a null collection. See "Design Considerations for Nested Tables" on page 9-10 for more information on using nested tables.

Constructor Methods for Collections

Example 3–1 shows how a literal invocation of the constructor method is used in a SQL statement to insert values into a nested table type `people_typ`.

Example 3–1 Using the Constructor Method to Insert Values into a Nested Table

```
CREATE TYPE people_typ AS TABLE OF person_typ;
/
```

```
CREATE TABLE people_tab (
    group_no NUMBER,
    people_column people_typ )
    NESTED TABLE people_column STORE AS people_column_nt;

INSERT INTO people_tab VALUES (
            100,
            people_typ( person_typ(1, 'John Smith', '1-800-555-1212'),
                        person_typ(2, 'Diane Smith', NULL)));
```

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.

Example 3–2 shows how to use literal invocations of constructor methods to specify defaults for person_typ and people_typ:

Example 3–2 Creating the department_persons Table Using the DEFAULT Clause

```
CREATE TABLE department_persons (
  dept_no    NUMBER PRIMARY KEY,
  dept_name  CHAR(20),
  dept_mgr   person_typ DEFAULT person_typ(10,'John Doe',NULL),
  dept_emps  people_typ DEFAULT people_typ() )
  NESTED TABLE dept_emps STORE AS dept_emps_tab;

INSERT INTO department_persons VALUES
   ( 101, 'Physical Sciences', person_typ(65,'Vrinda Mills', '1-800-555-4412'),
           people_typ( person_typ(1, 'John Smith', '1-800-555-1212'),
                        person_typ(2, 'Diane Smith', NULL) ) );
INSERT INTO department_persons VALUES
  ( 104, 'Life Sciences', person_typ(70,'James Hall', '1-800-555-4621'),
    people_typ() );
```

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

Varrays

A varray is an ordered set of data elements. All elements of a given varray are of the same data type or a subtype of the declared one. Each element has an index, which is a number corresponding to the element's position in the array. The index number is used to access a specific element.

When you define a varray, you specify the maximum number of elements it can contain, although you can change this number later. 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.

The following statement creates an array type email_list_arr that has no more than ten elements, each of data type VARCHAR2(80).

```
CREATE TYPE email_list_arr AS VARRAY(10) OF VARCHAR2(80);
/
```

In Example 3–3, a VARRAY type is created that is an array of an object type. The phone_varray_typ VARRAY type is used as a data type for a column in the dept_phone_list table. The INSERT statements show how to insert values into phone_varray_typ in the table.

Example 3–3 Creating and Populating a VARRAY Data Type

```
CREATE TYPE phone_typ AS OBJECT (
    country_code   VARCHAR2(2),
    area_code      VARCHAR2(3),
    ph_number      VARCHAR2(7));
/
CREATE TYPE phone_varray_typ AS VARRAY(5) OF phone_typ;
/
CREATE TABLE dept_phone_list (
  dept_no NUMBER(5),
  phone_list phone_varray_typ);

INSERT INTO dept_phone_list VALUES (
   100,
   phone_varray_typ( phone_typ ('01', '650', '5061111'),
                     phone_typ ('01', '650', '5062222'),
                     phone_typ ('01', '650', '5062525')));
```

Creating an array type, as with a SQL object type, does not allocate space. It defines a data type, which you can use as:

- The data type 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 "Storage Considerations for Varrays" on page 9-9.

You can create a VARRAY type of XMLType or of a LOB type for procedural purposes, such as in PL/SQL or in view queries. However, database storage of a varray of those types is not supported. This means that you cannot create an object table or an object type column of a varray type of XMLType or of a LOB type.

> **See Also:** *Oracle Database SQL Language Reference* for information and examples on the STORE AS LOB clause of the CREATE TABLE statement

Nested Tables

A nested table is an unordered set of data elements, all of the same data type. No maximum is specified in the definition of the table and the order of the elements is not preserved. You select, insert, delete, and update in a nested table just as you do with ordinary tables using the TABLE expression.

Elements of a nested table are actually stored in a separate storage table that contains a column that identifies the parent table row or object to which each element belongs. A nested table 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.

In Example 3–4 on page 3-4, the table type used for the nested tables is declared with the CREATE TYPE ... IS TABLE OF statement. A table type definition does not allocate space. It defines a type, which you can use as:

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

When a column in a relational table is of nested table type, Oracle stores the nested table data for all rows of the relational table in the same storage table. Similarly, with an object table of a type that has a nested table attribute, Oracle stores nested table data for all object instances in a single storage table associated with the object table. See Figure 9–2, "Nested Table Storage" on page 9-11.

In Example 3–4, the `NESTED TABLE` clause specifies the storage name for the nested table. The storage name is used when creating an index on a nested table. The example uses `person_typ` defined in Example 2–1 on page 2-2 and people_typ defined in Example 3–1 on page 3-1.

Example 3–4 Creating and Populating Simple Nested Tables

```
CREATE TABLE students (
   graduation DATE,
   math_majors people_typ,
   chem_majors people_typ,
   physics_majors people_typ)
  NESTED TABLE math_majors STORE AS math_majors_nt
  NESTED TABLE chem_majors STORE AS chem_majors_nt
  NESTED TABLE physics_majors STORE AS physics_majors_nt;

CREATE INDEX math_idno_idx ON math_majors_nt(idno);
CREATE INDEX chem_idno_idx ON chem_majors_nt(idno);
CREATE INDEX physics_idno_idx ON physics_majors_nt(idno);

INSERT INTO students (graduation) VALUES ('01-JUN-03');
UPDATE students
  SET math_majors =
        people_typ (person_typ(12, 'Bob Jones', '111-555-1212'),
                    person_typ(31, 'Sarah Chen', '111-555-2212'),
                    person_typ(45, 'Chris Woods', '111-555-1213')),
      chem_majors =
        people_typ (person_typ(51, 'Joe Lane', '111-555-1312'),
                    person_typ(31, 'Sarah Chen', '111-555-2212'),
                    person_typ(52, 'Kim Patel', '111-555-1232')),
   physics_majors =
        people_typ (person_typ(12, 'Bob Jones', '111-555-1212'),
                    person_typ(45, 'Chris Woods', '111-555-1213'))
WHERE graduation = '01-JUN-03';
```

A convenient way to access the elements of a nested table individually is to use a nested cursor or the `TABLE` function. See "Querying Collections" on page 3-10.

Specifying a Tablespace When Storing a Nested Table

A nested table can be stored in a different tablespace than its parent table. In Example 3–5, the nested table is stored in the `system` tablespace:

Example 3–5 Specifying a Different Tablespace for Storing a Nested Table

```
CREATE TABLE people_tab (
    people_column people_typ )
    NESTED TABLE people_column STORE AS people_column_nt (TABLESPACE system);
```

If the `TABLESPACE` clause is not specified, then the storage table of the nested table is created in the tablespace where the parent table is created. For multilevel nested tables, Oracle creates the child table in the same tablespace as its immediate preceding parent table.

The user can issue ALTER TABLE MOVE statement to move a table to a different tablespace. If the user issues ALTER TABLE MOVE statement on a table with nested table columns, it only moves parent table, no action is taken on the nested table's storage tables. If the user wants to move a nested table s storage table to a different tablespace, issue ALTER TABLE MOVE on the storage table. For example:

```
ALTER TABLE people_tab MOVE TABLESPACE system;
ALTER TABLE people_column_nt MOVE TABLESPACE example;
```

Now the people_tab table is in system tablespace and nested table is stored in the example tablespace.

Varray Storage

Multilevel varrays are stored in one of two ways, depending on whether the varray is a varray of varrays or a varray of nested tables.

- In a varray of varrays, the entire varray is stored inline in the row unless it is larger than approximately 4000 bytes or LOB storage is explicitly specified.
- In a varray of nested tables, the entire varray is stored in a LOB, with only the LOB locator stored in the row. There is no storage table associated with nested table elements of a varray. The entire nested table collection is stored inside the varray.

You can explicitly specify LOB storage for varrays. The following example does this for the varray elements of a nested table. As Example 3–6 also shows, you can use the COLUMN_VALUE keyword with varrays as well as nested tables.

Example 3–6 Specifying LOB Storage for a VARRAY Elements of a Nested Table

```
CREATE TYPE email_list_arr AS VARRAY(10) OF VARCHAR2(80);
/

CREATE TYPE email_list_typ AS TABLE OF email_list_arr;
/

CREATE TABLE dept_email_list (
  dept_no NUMBER,
  email_addrs email_list_typ)
  NESTED TABLE email_addrs STORE AS email_addrs_nt
  (VARRAY COLUMN_VALUE STORE AS LOB dept_emails_lob);
```

Example 3–7 shows explicit LOB storage specified for a varray of varray type.

Example 3–7 Specifying LOB Storage for a VARRAY Type

```
CREATE TYPE email_list_typ2 AS OBJECT (
    section_no   NUMBER,
    emails       email_list_arr);
/

CREATE TYPE email_varray_typ AS VARRAY(5) OF email_list_typ2;
/

CREATE TABLE dept_email_list2 (
  dept_no NUMBER,
  email_addrs email_varray_typ)
  VARRAY email_addrs STORE AS LOB dept_emails_lob2;
```

See "Storage Considerations for Varrays" on page 9-9. See also *Oracle Database SecureFiles and Large Objects Developer's Guide*.

Increasing the Size and Precision of VARRAYs and Nested Tables

When the element type of a VARRAY type is a variable character or RAW type or a numeric type, you can increase the size of the variable character or RAW type or increase the precision of the numeric type. A new type version is generated for the VARRAY type. The same changes can be applied to nested table types.

Options like INVALIDATE and CASCADE are provided to either invalidate all dependent objects or propagate the change to its type and table dependents.

Example 3–8 is illustrates the use ALTER TYPE to increase the size of a VARRAY and a nested table element type.

Example 3–8 Increasing the Size of an Element Type in a VARRAY and Nested Table

```
CREATE TYPE email_list_arr AS VARRAY(10) OF VARCHAR2(80);
/
ALTER TYPE email_list_arr MODIFY ELEMENT TYPE VARCHAR2(100) CASCADE;

CREATE TYPE email_list_tab AS TABLE OF VARCHAR2(30);
/
ALTER TYPE email_list_tab MODIFY ELEMENT TYPE VARCHAR2(40) CASCADE;
```

Increasing VARRAY Limit Size

The ALTER TYPE ... MODIFY LIMIT syntax allows increasing the number of elements of a VARRAY type. If the number of elements of the VARRAY type is increased, a new type version is generated for the VARRAY type and this is maintained as part of the history of the type changes.

Options like INVALIDATE and CASCADE are provided to either invalidate all dependent objects or propagate the change to its type and table dependents.

Example 3–9 Increasing the VARRAY Limit Size

```
CREATE TYPE email_list_arr AS VARRAY(10) OF VARCHAR2(80);
/
CREATE TYPE email_list_typ AS OBJECT (
    section_no   NUMBER,
    emails       email_list_arr);
/

CREATE TYPE email_varray_typ AS VARRAY(5) OF email_list_typ;
/

ALTER TYPE email_varray_typ MODIFY LIMIT 100 INVALIDATE;
```

When a VARRAY type is altered, changes are propagated to the dependent tables. See "Propagating VARRAY Size Change" on page 9-10.

Creating a Varray Containing LOB References

In Example 3–10, email_addrs of type email_list_typ already exists in table dept_email_list as shown in the SQL examples in "Varray Storage" on page 3-5.

To create a varray of LOB references, first define a VARRAY type of type REF email_list_typ2. Next define a column of the array type in dept_email_list3.

Example 3–10 Creating a VARRY Containing LOB References

```
CREATE TYPE ref_email_varray_typ AS VARRAY(5) OF REF email_list_typ;
/

CREATE TABLE dept_email_list3 (
  dept_no NUMBER,
  email_addrs ref_email_varray_typ)
  VARRAY email_addrs STORE AS LOB dept_emails_lob3;
```

Multilevel Collection Types

Multilevel collection types are collection types whose elements are themselves directly or indirectly another collection type. Possible multilevel collection types are:

- Nested table of nested table type
- Nested table of varray type
- Varray of nested table type
- Varray of varray type
- Nested table or varray of a user-defined type that has an attribute that is a nested table or varray type

Like ordinary, single-level collection types, multilevel collection types can be used as columns in a relational table or with object attributes in an object table.

Nested Table Storage Tables for Multilevel Collection Types

A nested table type column or object table attribute requires a storage table where rows for all nested tables in the column are stored. With a multilevel nested table collection of nested tables, the inner set of nested tables also requires a storage table just as the outer set does. You specify one by appending a second nested-table storage clause.

Example 3–11 creates a multilevel collection type that is a nested table of nested tables. The example models a system of corporate regions in which each region has a nested table collection of the countries, and each country has a nested table collection of its locations. This example is based on the regions, countries, and locations tables of the Oracle HR sample schema.

In Example 3–11, the SQL statements create a table region_tab that contains a column countries whose type is a multilevel collection. This multilevel collection is a nested table of an object type that has a nested table attribute locations. Separate nested table clauses are provided for the outer countries nested table and for the inner locations one.

Example 3–11 Multilevel Nested Table Storage

```
CREATE TYPE location_typ AS OBJECT (
  location_id      NUMBER(4),
  street_address   VARCHAR2(40),
  postal_code      VARCHAR2(12),
  city             VARCHAR2(30),
  state_province   VARCHAR2(25));
/
```

```
CREATE TYPE nt_location_typ AS TABLE OF location_typ;
/

CREATE TYPE country_typ AS OBJECT (
  country_id     CHAR(2),
  country_name   VARCHAR2(40),
  locations      nt_location_typ);
/

CREATE TYPE nt_country_typ AS TABLE OF country_typ;
/

CREATE TABLE region_tab (
  region_id     NUMBER,
  region_name   VARCHAR2(25),
  countries     nt_country_typ)
  NESTED TABLE countries STORE AS nt_countries_tab
   (NESTED TABLE locations STORE AS nt_locations_tab);
```

In Example 3–11 you can refer to the inner `locations` nested table by name because this nested table is a named attribute of an object. However, if the inner nested table is not an attribute, it has no name. The keyword `COLUMN_VALUE` is provided for this case; you use it in place of a name for an inner nested table as shown in Example 3–12.

Example 3–12 Multilevel Nested Table Storage Using the COLUMN_VALUE Keyword

```
CREATE TYPE inner_table AS TABLE OF NUMBER;
/
CREATE TYPE outer_table AS TABLE OF inner_table;
/
CREATE TABLE tab1 (
  col1 NUMBER,
  col2 outer_table)
NESTED TABLE col2 STORE AS col2_ntab
  (NESTED TABLE COLUMN_VALUE STORE AS cv_ntab);
```

Physical attributes for the storage tables can be specified in the nested table clause, as shown in Example 3–13.

Example 3–13 Specifying Physical Attributes for Nested Table Storage

```
CREATE TABLE region_tab (
  region_id     NUMBER,
  region_name   VARCHAR2(25),
  countries     nt_country_typ)
  NESTED TABLE countries STORE AS nt_countries_tab (
   (PRIMARY KEY (NESTED_TABLE_ID, country_id))
   ORGANIZATION INDEX COMPRESS
   NESTED TABLE locations STORE AS nt_locations_tab);
```

Every nested table storage table contains a column, referenceable by `NESTED_TABLE_ID`, that keys rows in the storage table to the associated row in the parent table. A parent table that is itself a nested table has two system-supplied ID columns: one, referenceable by `NESTED_TABLE_ID`, that keys its rows back to rows in its own parent table, and one hidden column referenced by the `NESTED_TABLE_ID` column in its nested table children.

In Example 3–13, nested table `countries` is made an index-organized table (IOT) by adding the `ORGANIZATION INDEX` clause and assigning the nested table a primary key in which the first column is NESTED_TABLE_ID. This column contains the ID of the row in the parent table with which a storage table row is associated. Specifying a primary key with NESTED_TABLE_ID as the first column and index-organizing the table cause Oracle to physically cluster all the nested table rows that belong to the same parent row, for more efficient access.

Each nested table needs its own table storage clause, so you must have as many nested table storage clauses as you have levels of nested tables in a collection. See "Nested Table Storage" on page 9-10.

Assignment and Comparison of Multilevel Collections

As with single-level collections, both the source and the target must be of the same declared data type in assignments of multilevel collections.

Only items whose data types are nested table collection types, including multilevel collection types, can be compared. See "Comparisons of Collections" on page 3-14.

Constructors for Multilevel Collections

Like single-level collection types, multilevel collection types are created by calling the respective type's constructor method. Like the constructor methods for other object types, a constructor for a multilevel collection type is a system-defined function that has the same name as the type and returns a new instance of it—in this case, a new multilevel collection. Constructor parameters have the names and types of the object type's attributes.

Example 3–14 shows the constructor call for the multilevel collection type nt_country_typ. This type is a nested table of countries, each of which contains a nested table of locations as an attribute. The constructor for the outer nested table calls the country_typ constructor for each country to be created; each country constructor calls the constructor for the locations nested table type to create its nested table of locations; and the locations nested table type constructor calls the location_typ constructor for each location instance to be created.

Example 3–14 Using Constructors for Multilevel Collections

```
INSERT INTO region_tab
VALUES(1, 'Europe', nt_country_typ(
  country_typ( 'IT', 'Italy', nt_location_typ (
    location_typ(1000, '1297 Via Cola di Rie','00989','Roma', ''),
    location_typ(1100, '93091 Calle della Testa','10934','Venice','') )
    ),
  country_typ( 'CH', 'Switzerland', nt_location_typ (
    location_typ(2900, '20 Rue des Corps-Saints', '1730', 'Geneva', 'Geneve'),
    location_typ(3000, 'Murtenstrasse 921', '3095', 'Bern', 'BE') )
    ),
  country_typ( 'UK', 'United Kingdom', nt_location_typ (
    location_typ(2400, '8204 Arthur St', '', 'London', 'London'),
    location_typ(2500, 'Magdalen Centre, The Oxford Science Park', 'OX9 9ZB',
                 'Oxford', 'Oxford'),
    location_typ(2600, '9702 Chester Road', '09629850293', 'Stretford',
                 'Manchester') )
      )
  )
);
```

Operations on Collection Data Types

This section describes the operations on collection data types.

Querying Collections

There are two general ways to query a table that contains a column or attribute of a collection type. One way returns the collections nested in the result rows that contain them. The other way distributes or unnests collections such that each collection element appears on a row by itself.

Nesting Results of Collection Queries

The following queries use the department_persons table shown in Example 3–2 on page 3-2. The column dept_emps is a nested table collection of person_typ type. The dept_emps collection column appears in the SELECT list like an ordinary, scalar column. Querying a collection column in the SELECT list like this nests the elements of the collection in the result row with which the collection is associated.

In Example 3–15 the query retrieves the nested collection of employees.

Example 3–15 Nesting Results of Collection Queries

```
SELECT d.dept_emps
  FROM department_persons d;

DEPT_EMPS(IDNO, NAME, PHONE)
-------------------------------------------------------------
PEOPLE_TYP(PERSON_TYP(1, 'John Smith', '1-800-555-1212'),
PERSON_TYP(2, 'Diane Smith', '1-800-555-1243'))
```

The results are also nested if an object-type column in the SELECT list contains a collection attribute, even if that collection is not explicitly listed in the SELECT list itself. For example, the query SELECT * FROM department_persons would produce a nested result.

Unnesting Results of Collection Queries

Not all tools or applications are able to deal with results in a nested format. To view Oracle collection data using tools that require a conventional format, you must unnest, or flatten, the collection attribute of a row into one or more relational rows. You can do this by using a TABLE expression with the collection. A TABLE expression enables you to query a collection in the FROM clause like a table. In effect, you join the nested table with the row that contains the nested table.

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

As in Example 3–15, the query in Example 3–16 retrieves the collection of employees, but the collection is unnested.

Example 3–16 Unnesting Results of Collection Queries

```
SELECT e.*
  FROM department_persons d, TABLE(d.dept_emps) e;

IDNO       NAME                           PHONE
---------- ------------------------------ ---------------
         1 John Smith                     1-800-555-1212
         2 Diane Smith                    1-800-555-1243
```

As shown in Example 3–16, a TABLE expression can have its own table alias. In the example, a table alias for the TABLE expression appears in the SELECT list to select columns returned by the TABLE expression.

The TABLE expression uses another table alias to specify the table that contains the collection column that the TABLE expression references. The expression TABLE(d.dept_emps) specifies the department_persons table as containing the dept_emps collection column. A TABLE expression can use the table alias of any table appearing to the left of it in a FROM clause to reference a column of that table. This way of referencing collection columns is called left correlation.

In the example, the department_persons table is listed in the FROM clause solely to provide a table alias for the TABLE expression to use. No columns from the department_persons table other than the column referenced by the TABLE expression appear in the result

The following example produces rows only for departments that have employees.

```
SELECT d.dept_no, e.*
  FROM department_persons d, TABLE(d.dept_emps) e;
```

To get rows for departments that have no employees, you can use outer-join syntax:

```
SELECT d.dept_no, e.*
  FROM department_persons d, TABLE(d.dept_emps) (+) e;
```

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

Unnesting Queries Containing Table Expression Subqueries

The examples in "Unnesting Results of Collection Queries" on page 3-10 show a TABLE expression that contains the name of a collection. Alternatively, a TABLE expression can contain a subquery of a collection.

Example 3–17 returns the collection of employees whose department number is 101.

Example 3–17 Using a Table Expression Containing a Subquery of a Collection

```
SELECT *
  FROM TABLE(SELECT d.dept_emps
             FROM department_persons d
             WHERE d.dept_no = 101);
```

There are these restrictions on using a subquery in a TABLE expression:

- The subquery must return a collection type
- The SELECT list of the subquery must contain exactly one item
- The subquery must return only a single collection; it cannot return collections for multiple rows. For example, the subquery SELECT dept_emps FROM department_persons succeeds in a TABLE expression only if table department_persons contains just a single row. If the table contains more than one row, the subquery produces an error.

Example 3–18 shows a TABLE expression used in the FROM clause of a SELECT embedded in a CURSOR expression.

Example 3–18 Using a Table Expression in a CURSOR Expression

```
SELECT d.dept_no, CURSOR(SELECT * FROM TABLE(d.dept_emps))
  FROM department_persons d
   WHERE d.dept_no = 101;
```

Unnesting Queries with Multilevel Collections

Unnesting queries can be used with multilevel collections, too, for both varrays and nested tables. Example 3–19 shows an unnesting query on a multilevel nested table collection of nested tables. From a table `region_tab` in which each region has a nested table of countries and each country has a nested table of locations, the query returns the names of all regions, countries, and locations.

Example 3–19 Unnesting Queries with Multilevel Collections Using the TABLE Function

```
SELECT r.region_name, c.country_name, l.location_id
  FROM region_tab r, TABLE(r.countries) c, TABLE(c.locations) l;

-- the following query is optimized to run against the locations table
SELECT l.location_id, l.city
  FROM region_tab r, TABLE(r.countries) c, TABLE(c.locations) l;
```

Because no columns of the base table `region_tab` appear in the second `SELECT` list, the query is optimized to run directly against the `locations` storage table.

Outer-join syntax can also be used with queries of multilevel collections. See "Viewing Object Data in Relational Form with Unnesting Queries" on page 9-8.

Performing DML Operations 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
  - Updating elements of the collection.

Oracle does not support piecewise updates on `VARRAY` columns. However, `VARRAY` columns can be inserted into or updated as an atomic unit.

For piecewise updates of nested table columns, the DML statement identifies the nested table value to be operated on by using the `TABLE` expression.

The DML statements in Example 3–20 demonstrate piecewise operations on nested table columns.

Example 3–20 Piecewise Operations on Collections

```
INSERT INTO TABLE(SELECT d.dept_emps
                  FROM department_persons d
                  WHERE d.dept_no = 101)
   VALUES (5, 'Kevin Taylor', '1-800-555-6212');

UPDATE TABLE(SELECT d.dept_emps
                  FROM department_persons d
                  WHERE d.dept_no = 101) e
   SET VALUE(e) = person_typ(5, 'Kevin Taylor', '1-800-555-6233')
   WHERE e.idno = 5;

DELETE FROM TABLE(SELECT d.dept_emps
                  FROM department_persons d
                  WHERE d.dept_no = 101) e
   WHERE e.idno = 5;
```

Example 3–21 shows VALUE used to return object instance rows for updating:

Example 3–21 Using VALUE to Update a Nested Table

```
UPDATE TABLE(SELECT d.dept_emps FROM department_persons d
               WHERE  d.dept_no = 101) p
   SET VALUE(p) = person_typ(2, 'Diane Smith', '1-800-555-1243')
   WHERE p.idno = 2;
```

Performing DML on Multilevel Collections

For multilevel nested table collections, DML can be done atomically, on the collection as a whole, or piecewise, on selected elements. For multilevel varray collections, DML operations can be done only atomically.

Collections as Atomic Data Items The section "Constructors for Multilevel Collections" on page 3-9 shows an example of inserting an entire multilevel collection with an INSERT statement. Multilevel collections can also be updated atomically with an UPDATE statement. For example, suppose v_country is a variable declared to be of the countries nested table type nt_country_typ. Example 3–22 updates region_tab by setting the countries collection as a unit to the value of v_country.

Example 3–22 Using UPDATE to Insert an Entire Multilevel Collection

```
INSERT INTO region_tab (region_id, region_name) VALUES(2, 'Americas');

DECLARE
  v_country nt_country_typ;
BEGIN
  v_country :=  nt_country_typ( country_typ(
   'US', 'United States of America', nt_location_typ (
   location_typ( 1500,'2011 Interiors Blvd','99236','San Francisco','California'),
   location_typ(1600,'2007 Zagora St','50090','South Brunswick','New Jersey'))));
  UPDATE region_tab r
    SET r.countries = v_country WHERE r.region_id = 2;
END;
/
```

Piecewise Operations on Nested Tables Piecewise DML is possible only on nested tables, not on varrays.

Example 3–23 shows a piecewise insert operation on the countries nested table of nested tables. The example inserts a new country, complete with its own nested table of location_typ:

Example 3–23 Piecewise INSERT on a Multilevel Collection

```
INSERT INTO TABLE( SELECT countries FROM region_tab r WHERE r.region_id = 2)
  VALUES ( 'CA', 'Canada', nt_location_typ(
       location_typ(1800, '147 Spadina Ave', 'M5V 2L7', 'Toronto', 'Ontario')));
```

Example 3–24 performs a piecewise insert into an inner nested table to add a location for a country. Like the preceding, this example uses a TABLE expression containing a subquery that selects the inner nested table to specify the target for the insert.

Example 3–24 Piecewise INSERT into an Inner Nested Table

```
INSERT INTO TABLE( SELECT c.locations
  FROM TABLE( SELECT r.countries FROM region_tab r WHERE r.region_id = 2) c
  WHERE c.country_id = 'US')
  VALUES (1700, '2004 Lakeview Rd', '98199', 'Seattle', 'Washington');

SELECT r.region_name, c.country_name, l.location_id
  FROM region_tab r, TABLE(r.countries) c, TABLE(c.locations) l;
```

Comparisons of Collections

The conditions listed in this section allow comparisons of nested tables. There is no mechanism for comparing varrays. The SQL examples in this section use the nested tables created in Example 3–4 on page 3-4.

Equal and Not Equal Comparisons

The equal (=) and not equal (<>) conditions determine whether the input nested tables are identical or not, returning the result as a Boolean value.

Two nested tables are equal if they have the same named type, have the same cardinality, and their elements are equal. Elements are equal depending on whether they are equal by the elements own equality definitions, except for object types which require a map method.

Example 3–25 Using an Equality Comparison with Nested Tables

```
SELECT p.name
  FROM students, TABLE(physics_majors) p
WHERE math_majors = physics_majors;
```

In Example 3–25, the nested tables contain person_typ objects which have an associated map method. See Example 2–1 on page 2-2.

In Comparisons

The IN condition checks whether a nested table is in a list of nested tables, returning the result as a Boolean value. NULL is returned if the nested table is a null nested table.

Example 3–26 Using an IN Comparison with Nested Tables

```
SELECT p.idno, p.name
  FROM students, TABLE(physics_majors) p
WHERE physics_majors IN (math_majors, chem_majors);
```

Subset of Multiset Comparison

The SUBMULTISET [OF] condition checks whether a nested table is a subset of a another nested table, returning the result as a Boolean value. The OF keyword is optional and does not change the functionality of SUBMULTISET.

This operator is implemented only for nested tables because this is a multiset function only.

Example 3–27 Testing the SUBMULTISET OF Condition on a Nested Table

```
SELECT p.idno, p.name
  FROM students, TABLE(physics_majors) p
WHERE physics_majors SUBMULTISET OF math_majors;
```

Member of a Nested Table Comparison

The MEMBER [OF] or NOT MEMBER [OF] condition tests whether an element is a member of a nested table, returning the result as a Boolean value. The OF keyword is optional and has no effect on the output.

Example 3–28 Using MEMBER OF on a Nested Table

```
SELECT graduation
  FROM students
WHERE person_typ(12, 'Bob Jones', '1-800-555-1212') MEMBER OF math_majors;
```

In Example 3–28, person_typ (12, 'Bob Jones', '1-800-555-1212') is an element of the same type as the elements of the nested table math_majors.

Empty Comparison

The IS [NOT] EMPTY condition checks whether a given nested table is empty or not empty, regardless of whether any of the elements are NULL. If a NULL is given for the nested table, the result is NULL. The result is returned as a Boolean value.

Example 3–29 Using IS NOT on a Nested Table

```
SELECT p.idno, p.name
  FROM students, TABLE(physics_majors) p
WHERE physics_majors IS NOT EMPTY;
```

Set Comparison

The IS [NOT] A SET condition checks whether a given nested table is composed of unique elements, returning a Boolean value.

Example 3–30 Using IS A SET on a Nested Table

```
SELECT p.idno, p.name
  FROM students, TABLE(physics_majors) p
WHERE physics_majors IS A SET;
```

Multisets Operations

This section describes multiset operations with nested tables. For a description of additional operations, see "Comparisons of Objects, REF Variables, and Collections" on page 2-31. The SQL examples in this section use the nested tables created in Example 3–4 on page 3-4.

For more information about using operators with nested tables, see *Oracle Database SQL Language Reference*.

CARDINALITY

The CARDINALITY function returns the number of elements in a varray or nested table. The return type is NUMBER. If the varray or nested table is a null collection, NULL is returned.

Example 3–31 Determining the CARDINALITY of a Nested Table

```
SELECT CARDINALITY(math_majors)
  FROM students;
```

For more information about the CARDINALITY function, see *Oracle Database SQL Language Reference*.

COLLECT

The COLLECT function is an aggregate function which would create a multiset from a set of elements. The function would take a column of the element type as input and create a multiset from rows selected. To get the results of this function you must use it within a CAST function to specify the output type of COLLECT. See "CAST" on page 2-32 for an example of the COLLECT function.

For more information about the COLLECT function, see *Oracle Database SQL Language Reference*.

MULTISET EXCEPT

The MULTISET EXCEPT operator inputs two nested tables and returns a nested table whose elements are in the first nested table but not in the second nested table. The input nested tables and the output nested table are all type name equivalent.

The ALL or DISTINCT options can be used with the operator. The default is ALL.

- With the ALL option, for ntab1 MULTISET EXCEPT ALL ntab2, all elements in ntab1 other than those in ntab2 would be part of the result. If a particular element occurs *m* times in ntab1 and *n* times in ntab2, the result will have (*m* - *n*) occurrences of the element if *m* is greater than *n* otherwise 0 occurrences of the element.
- With the DISTINCT option, any element that is present in ntab1 which is also present in ntab2 would be eliminated, irrespective of the number of occurrences.

Example 3–32 Using the MULTISET EXCEPT Operation on Nested Tables

```
SELECT math_majors MULTISET EXCEPT physics_majors
  FROM students
WHERE graduation = '01-JUN-03';
```

For more information about the MULTISET EXCEPT operator, see *Oracle Database SQL Language Reference*.

MULTISET INTERSECTION

The MULTISET INTERSECT operator returns a nested table whose values are common in the two input nested tables. The input nested tables and the output nested table are all type name equivalent.

There are two options associated with the operator: ALL or DISTINCT. The default is ALL. With the ALL option, if a particular value occurs *m* times in ntab1 and *n* times in ntab2, the result would contain the element MIN(*m*, *n*) times. With the DISTINCT option the duplicates from the result would be eliminated, including duplicates of NULL values if they exist.

Example 3–33 Using the MULTISET INTERSECT Operation on Nested Tables

```
SELECT math_majors MULTISET INTERSECT physics_majors
  FROM students
```

```
WHERE graduation = '01-JUN-03';
```

For more information about the MULTISET INTERSECT operator, see *Oracle Database SQL Language Reference*.

MULTISET UNION

The MULTISET UNION operator returns a nested table whose values are those of the two input nested tables. The input nested tables and the output nested table are all type name equivalent.

There are two options associated with the operator: ALL or DISTINCT. The default is ALL. With the ALL option, all elements that are in ntab1 and ntab2 would be part of the result, including all copies of NULLs. If a particular element occurs *m* times in ntab1 and *n* times in ntab2, the result would contain the element (*m* + *n*) times. With the DISTINCT option the duplicates from the result are eliminated, including duplicates of NULL values if they exist.

Example 3–34 Using the MULTISET UNION Operation on Nested Tables

```
SELECT math_majors MULTISET UNION DISTINCT physics_majors
  FROM students
WHERE graduation = '01-JUN-03';

PEOPLE_TYP(PERSON_TYP(12, 'Bob Jones', '1-800-555-1212'),
       PERSON_TYP(31, 'Sarah Chen', '1-800-555-2212'),
       PERSON_TYP(45, 'Chris Woods', '1-800-555-1213'))

SELECT math_majors MULTISET UNION ALL physics_majors
  FROM students
WHERE graduation = '01-JUN-03';

PEOPLE_TYP(PERSON_TYP(12, 'Bob Jones', '1-800-555-1212'),
    PERSON_TYP(31, 'Sarah Chen', '1-800-555-2212'),
    PERSON_TYP(45, 'Chris Woods', '1-800-555-1213'),
    PERSON_TYP(12, 'Bob Jones', '1-800-555-1212'),
    PERSON_TYP(45, 'Chris Woods', '1-800-555-1213'))
```

For more information about the MULTISET UNION operator, see *Oracle Database SQL Language Reference*.

POWERMULTISET

The POWERMULTISET function generates all non-empty submultisets from a given multiset. The input to the POWERMULTISET function could be any expression which evaluates to a multiset. The limit on the cardinality of the multiset argument is 32.

Example 3–35 Using the POWERMULTISET Operation on Multiset

```
SELECT * FROM TABLE(POWERMULTISET( people_typ (
           person_typ(12, 'Bob Jones', '1-800-555-1212'),
           person_typ(31, 'Sarah Chen', '1-800-555-2212'),
           person_typ(45, 'Chris Woods', '1-800-555-1213'))));
```

For more information about the POWERMULTISET function, see *Oracle Database SQL Language Reference*.

POWERMULTISET_BY_CARDINALITY

The POWERMULTISET_BY_CARDINALITY function returns all non-empty submultisets of a nested table of the specified cardinality. The output would be rows of nested tables.

POWERMULTISET_BY_CARDINALITY(x, l) is equivalent to TABLE(POWERMULTISET(x)) p where CARDINALITY(value(p)) = l, where *x* is a multiset and l is the specified cardinality.

The first input parameter to the POWERMULTISET_BY_CARDINALITY could be any expression which evaluates to a nested table. The length parameter should be a positive integer, otherwise an error will be returned. The limit on the cardinality of the nested table argument is 32.

Example 3–36 Using the POWERMULTISET_BY_CARDINALITY Function

```
SELECT * FROM TABLE(POWERMULTISET_BY_CARDINALITY( people_typ (
           person_typ(12, 'Bob Jones', '1-800-555-1212'),
           person_typ(31, 'Sarah Chen', '1-800-555-2212'),
           person_typ(45, 'Chris Woods', '1-800-555-1213')),2));
```

For more information about the POWERMULTISET_BY_CARDINALITY function, see *Oracle Database SQL Language Reference*.

SET

The SET function converts a nested table into a set by eliminating duplicates, and returns a nested table whose elements are DISTINCT from one another. The nested table returned is of the same named type as the input nested table.

Example 3–37 Using the SET Function on a Nested Table

```
SELECT SET(physics_majors)
  FROM students
WHERE graduation = '01-JUN-03';
```

For more information about the SET function, see *Oracle Database SQL Language Reference*.

4

Using PL/SQL With Object Types

This chapter describes how to use object types with PL/SQL

This chapter contains these topics:

Declaring and Initializing Objects in PL/SQL

Using object types in a PL/SQL block, subprogram, or package is a two-step process.

1. You must define object types using the SQL statement `CREATE TYPE`, in SQL*Plus or other similar programs.

   For information on the `CREATE TYPE` and `CREATE TYPE BODY` SQL statements, see *Oracle Database SQL Language Reference*.

   After an object type is defined and installed in the schema, you can use it in any PL/SQL block, subprogram, or package.

2. In PL/SQL, you then declare a variable whose data type is the ADT that you just defined.

Objects or ADTs follow the usual scope and instantiation rules.

Defining Object Types

Example 4–1 shows how to create an object type, object body type, and a table of object types. Then, the next example shows how to declare a variable of that data type in PL/SQL.

Example 4–1 Working With Object Types

```
CREATE TYPE address_typ AS OBJECT (
   street          VARCHAR2(30),
   city            VARCHAR2(20),
   state           CHAR(2),
   postal_code     VARCHAR2(6) );
/
CREATE TYPE employee_typ AS OBJECT (
  employee_id       NUMBER(6),
  first_name        VARCHAR2(20),
  last_name         VARCHAR2(25),
  email             VARCHAR2(25),
  phone_number      VARCHAR2(20),
  hire_date         DATE,
  job_id            VARCHAR2(10),
  salary            NUMBER(8,2),
  commission_pct    NUMBER(2,2),
  manager_id        NUMBER(6),
  department_id     NUMBER(4),
  address           address_typ,
  MAP MEMBER FUNCTION get_idno RETURN NUMBER,
  MEMBER PROCEDURE display_address ( SELF IN OUT NOCOPY employee_typ ) );
/
CREATE TYPE BODY employee_typ AS
  MAP MEMBER FUNCTION get_idno RETURN NUMBER IS
  BEGIN
    RETURN employee_id;
  END;
  MEMBER PROCEDURE display_address ( SELF IN OUT NOCOPY employee_typ ) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(first_name || ' '  || last_name);
    DBMS_OUTPUT.PUT_LINE(address.street);
    DBMS_OUTPUT.PUT_LINE(address.city || ', '  || address.state || ' ' ||
                         address.postal_code);
  END;
END;
/
CREATE TABLE employee_tab OF employee_typ;
```

Declaring Objects in a PL/SQL Block

You can use objects or ADTs wherever built-in types such as CHAR or NUMBER can be used. In Example 4–2, you declare object emp of type employee_typ. Then, you call the constructor for object type employee_typ to initialize the object.

Example 4–2 Declaring Objects in a PL/SQL Block

```
DECLARE
  emp employee_typ; -- emp is atomically null
BEGIN
-- call the constructor for employee_typ
  emp := employee_typ(315, 'Francis', 'Logan', 'FLOGAN',
        '555.777.2222', '01-MAY-04', 'SA_MAN', 11000, .15, 101, 110,
         address_typ('376 Mission', 'San Francisco', 'CA', '94222'));
  DBMS_OUTPUT.PUT_LINE(emp.first_name || ' ' || emp.last_name); -- display details
  emp.display_address();  -- call object method to display details
END;
/
```

The formal parameter of a PL/SQL subprogram may have data type of ADT. Therefore, you can pass objects to stored subprograms and from one subprogram to another. In the next example, you use object type employee_typ to specify the datatype of a formal parameter:

```
PROCEDURE open_acct (new_acct IN OUT employee_typ) IS ...
```

In the following example, you use object type employee_typ to specify the return type of a function:

```
FUNCTION get_acct (acct_id IN NUMBER) RETURN employee_typ IS ...
```

How PL/SQL Treats Uninitialized Objects

ADTs, just like collections are atomically null, until you initialize the object by calling the constructor for its object type. That is, the object itself is null, not just its attributes.

Comparing a null object with any other object always yields NULL. Also, if you assign an atomically null object to another object, the other object becomes atomically null (and must be reinitialized). Likewise, if you assign the non-value NULL to an object, the object becomes atomically null.

In an expression, attributes of an uninitialized object evaluate to NULL. When applied to an uninitialized object or its attributes, the IS NULL comparison operator yields TRUE.

Example 4–3 illustrates null objects and objects with null attributes.

Example 4–3 Null Objects in a PL/SQL Block

```
DECLARE
  emp employee_typ; -- emp is atomically null
BEGIN
  IF emp IS NULL THEN DBMS_OUTPUT.PUT_LINE('emp is NULL #1'); END IF;
  IF emp.employee_id IS NULL THEN
     DBMS_OUTPUT.PUT_LINE('emp.employee_id is NULL #1');
  END IF;
  emp.employee_id := 330;
  IF emp IS NULL THEN DBMS_OUTPUT.PUT_LINE('emp is NULL #2'); END IF;
  IF emp.employee_id IS NULL THEN
    DBMS_OUTPUT.PUT_LINE('emp.employee_id is NULL #2');
  END IF;
  emp := employee_typ(NULL, NULL, NULL, NULL,
        NULL, NULL, NULL, NULL, NULL, NULL, NULL,
         address_typ(NULL, NULL, NULL, NULL));
  -- emp := NULL; -- this would have made the following IF statement TRUE
  IF emp IS NULL THEN DBMS_OUTPUT.PUT_LINE('emp is NULL #3'); END IF;
  IF emp.employee_id IS NULL THEN
    DBMS_OUTPUT.PUT_LINE('emp.employee_id is NULL #3');
  END IF;
EXCEPTION
   WHEN ACCESS_INTO_NULL THEN
     DBMS_OUTPUT.PUT_LINE('Cannot assign value to NULL object');
END;
/
```

The output is:

```
emp is NULL #1
emp.employee_id is NULL #1
emp is NULL #2
emp.employee_id is NULL #3
```

Calls to methods of an uninitialized object raise the predefined exception NULL_SELF_DISPATCH. When passed as arguments to IN parameters, attributes of an uninitialized object evaluate to NULL. When passed as arguments to OUT or IN OUT parameters, they raise an exception if you try to write to them.

Manipulating Objects in PL/SQL

This section describes how to manipulate object attributes and methods in PL/SQL.

Accessing Object Attributes With Dot Notation

You refer to an attribute by name. To access or change the value of an attribute, you use dot notation. Attribute names can be chained, which lets you access the attributes of a nested object type. For example:

Example 4–4 Accessing Object Attributes

```
DECLARE
  emp employee_typ;
BEGIN
  emp := employee_typ(315, 'Francis', 'Logan', 'FLOGAN',
        '555.777.2222', '01-MAY-04', 'SA_MAN', 11000, .15, 101, 110,
         address_typ('376 Mission', 'San Francisco', 'CA', '94222'));
  DBMS_OUTPUT.PUT_LINE(emp.first_name || ' '  || emp.last_name);
  DBMS_OUTPUT.PUT_LINE(emp.address.street);
  DBMS_OUTPUT.PUT_LINE(emp.address.city || ', '  ||emp. address.state || ' ' ||
                     emp.address.postal_code);
END;
/
```

Calling Object Constructors and Methods

Calls to a constructor are allowed wherever function calls are allowed. Like all functions, a constructor is called as part of an expression, as shown in Example 4–4 on page 4-4 and Example 4–5.

Example 4–5 Inserting Rows in an Object Table

```
DECLARE
  emp employee_typ;
BEGIN
  INSERT INTO employee_tab VALUES (employee_typ(310, 'Evers', 'Boston', 'EBOSTON',
   '555.111.2222', '01-AUG-04', 'SA_REP', 9000, .15, 101, 110,
    address_typ('123 Main', 'San Francisco', 'CA', '94111')) );
  INSERT INTO employee_tab VALUES (employee_typ(320, 'Martha', 'Dunn', 'MDUNN',
    '555.111.3333', '30-SEP-04', 'AC_MGR', 12500, 0, 101, 110,
    address_typ('123 Broadway', 'Redwood City', 'CA', '94065')) );
END;
/
```

When you pass parameters to a constructor, the call assigns initial values to the attributes of the object being instantiated. When you call the default constructor to fill in all attribute values, you must supply a parameter for every attribute; unlike constants and variables, attributes cannot have default values. You can call a constructor using named notation instead of positional notation.

Like packaged subprograms, methods are called using dot notation. In Example 4–6, the `display_address` method is called to display attributes of an object. Note the use of the `VALUE` function which returns the value of an object. `VALUE` takes as its argument a correlation variable. In this context, a correlation variable is a row variable or table alias associated with a row in an object table.

Example 4–6 Accessing Object Methods

```
DECLARE
  emp employee_typ;
BEGIN
  SELECT VALUE(e) INTO emp FROM employee_tab e WHERE e.employee_id = 310;
  emp.display_address();
```

```
END;
/
```

In SQL statements, calls to a parameterless method require an empty parameter list. In procedural statements, an empty parameter list is optional unless you chain calls, in which case it is required for all but the last call. You cannot chain additional method calls to the right of a procedure call because a procedure is called as a statement, not as part of an expression. Also, if you chain two function calls, the first function must return an object that can be passed to the second function.

For static methods, calls use the notation *type_name.method_name* rather than specifying an instance of the type.

When you call a method using an instance of a subtype, the actual method that is executed depends on the exact declarations in the type hierarchy. If the subtype overrides the method that it inherits from its supertype, the call uses the subtype's implementation. Or, if the subtype does not override the method, the call uses the supertype's implementation. This capability is known as dynamic method dispatch.

Updating and Deleting Objects

From inside a PL/SQL block you can modify and delete rows in an object table.

Example 4–7   Updating and Deleting Rows in an Object Table

```
DECLARE
  emp employee_typ;
BEGIN
  INSERT INTO employee_tab VALUES (employee_typ(370, 'Robert', 'Myers', 'RMYERS',
   '555.111.2277', '07-NOV-04', 'SA_REP', 8800, .12, 101, 110,
    address_typ('540 Fillmore', 'San Francisco', 'CA', '94011')) );
  UPDATE employee_tab e SET e.address.street = '1040 California'
     WHERE e.employee_id = 370;
  DELETE FROM employee_tab e WHERE e.employee_id = 310;
END;
/
```

Manipulating Objects Through Ref Modifiers

You can retrieve refs using the function REF, which takes as its argument a correlation variable.

Example 4–8   Updating Rows in an Object Table With a REF Modifier

```
DECLARE
  emp         employee_typ;
  emp_ref REF employee_typ;
BEGIN
  SELECT REF(e) INTO emp_ref FROM employee_tab e WHERE e.employee_id = 370;
  UPDATE employee_tab e
    SET e.address = address_typ('8701 College', 'Oakland', 'CA', '94321')
    WHERE REF(e) = emp_ref;
END;
/
```

You can declare refs as variables, parameters, fields, or attributes. You can use refs as input or output variables in SQL data manipulation statements.

You cannot navigate through refs in PLSQL. For example, the assignment in Example 4–9 using a ref is not allowed. Instead, use the function DEREF or make calls to the package UTL_REF to access the object. For information on the REF function, see *Oracle Database SQL Language Reference*.

Example 4–9 Using DEREF in a SELECT INTO Statement

```
DECLARE
  emp          employee_typ;
  emp_ref   REF employee_typ;
  emp_name      VARCHAR2(50);
BEGIN
  SELECT REF(e) INTO emp_ref FROM employee_tab e WHERE e.employee_id = 370;
-- the following assignment raises an error, not allowed in PL/SQL
-- emp_name := emp_ref.first_name || ' ' || emp_ref.last_name;
-- emp := DEREF(emp_ref); not allowed, cannot use DEREF in procedural statements
  SELECT DEREF(emp_ref) INTO emp FROM DUAL; -- use dummy table DUAL
  emp_name := emp.first_name || ' ' || emp.last_name;
  DBMS_OUTPUT.PUT_LINE(emp_name);
END;
/
```

For detailed information on the DEREF function, see *Oracle Database SQL Language Reference*.

Defining SQL Types Equivalent to PL/SQL Collection Types

To store nested tables and varrays inside database tables, you must also declare SQL types using the CREATE TYPE statement. The SQL types can be used as columns or as attributes of SQL object types.

For more information on object types, see "Object Types" on page 1-3.

You can declare equivalent types within PL/SQL, or use the SQL type name in a PL/SQL variable declaration.

Example 4–10 shows how you might declare a nested table in SQL, and use it as an attribute of an object type.

Example 4–10 Declaring a Nested Table in SQL

```
CREATE TYPE CourseList AS TABLE OF VARCHAR2(10)  -- define type
/
CREATE TYPE student AS OBJECT (  -- create object
   id_num  INTEGER(4),
   name    VARCHAR2(25),
   address VARCHAR2(35),
   status  CHAR(2),
   courses CourseList);  -- declare nested table as attribute
/
CREATE TABLE sophomores of student
  NESTED TABLE courses STORE AS courses_nt;
```

The identifier courses represents an entire nested table. Each element of courses stores the name of a college course such as 'Math 1020'.

Example 4–11 creates a database column that stores varrays. Each varray element contains a VARCHAR2.

Example 4–11 Creating a Table with a Varray Column

```
-- Each project has a 16-character code name.
-- We will store up to 50 projects at a time in a database column.
```

```
CREATE TYPE ProjectList AS VARRAY(50) OF VARCHAR2(16);
/
CREATE TABLE dept_projects (  -- create database table
   dept_id  NUMBER(2),
   name     VARCHAR2(15),
   budget   NUMBER(11,2),
-- Each department can have up to 50 projects.
   projects ProjectList);
```

In Example 4–12, you insert a row into database table `dept_projects`. The varray constructor `ProjectList()` provides a value for column `projects`.

Example 4–12 Varray Constructor Within a SQL Statement

```
BEGIN
  INSERT INTO dept_projects
    VALUES(60, 'Security', 750400,
      ProjectList('New Badges', 'Track Computers', 'Check Exits'));
END;
/
```

In Example 4–13, you insert several scalar values and a `CourseList` nested table into the `sophomores` table.

Example 4–13 Nested Table Constructor Within a SQL Statement

```
CREATE TABLE sophomores of student
  NESTED TABLE courses STORE AS courses_nt;
BEGIN
   INSERT INTO sophomores
      VALUES (5035, 'Janet Alvarez', '122 Broad St', 'FT',
         CourseList('Econ 2010', 'Acct 3401', 'Mgmt 3100'));
END;
/
```

Manipulating Individual Collection Elements with SQL

By default, SQL operations store and retrieve whole collections rather than individual elements. To manipulate the individual elements of a collection with SQL, use the `TABLE` operator. The `TABLE` operator uses a subquery to extract the varray or nested table, so that the `INSERT`, `UPDATE`, or `DELETE` statement applies to the nested table rather than the top-level table.

To perform DML operations on a PL/SQL nested table, use the operators `TABLE` and `CAST`. This way, you can do set operations on nested tables using SQL notation, without actually storing the nested tables in the database.

The operands of `CAST` are PL/SQL collection variable and a SQL collection type (created by the `CREATE TYPE` statement). `CAST` converts the PL/SQL collection to the SQL type.

Example 4–14 Performing Operations on PL/SQL Nested Tables With CAST

```
CREATE TYPE Course AS OBJECT
           (course_no  NUMBER,
            title      VARCHAR2(64),
            credits    NUMBER);
/
CREATE TYPE CourseList AS TABLE OF course;
```

```
/

-- create department table
CREATE TABLE department (
   name     VARCHAR2(20),
   director VARCHAR2(20),
   office   VARCHAR2(20),
   courses  CourseList)
   NESTED TABLE courses STORE AS courses_tab;

INSERT INTO department VALUES ('English', 'June Johnson', '491C',
                 CourseList(Course(1002, 'Expository Writing', 4),
                 Course(2020, 'Film and Literature', 4),
                 Course(4210, '20th-Century Poetry', 4),
                 Course(4725, 'Advanced Workshop in Poetry', 4)));

DECLARE
   revised CourseList :=
      CourseList(Course(1002, 'Expository Writing', 3),
                 Course(2020, 'Film and Literature', 4),
                 Course(4210, '20th-Century Poetry', 4),
                 Course(4725, 'Advanced Workshop in Poetry', 5));
   num_changed INTEGER;
BEGIN
   SELECT COUNT(*) INTO num_changed
      FROM TABLE(CAST(revised AS CourseList)) new,
      TABLE(SELECT courses FROM department
         WHERE name = 'English') old
      WHERE new.course_no = old.course_no AND
         (new.title != old.title OR new.credits != old.credits);
   DBMS_OUTPUT.PUT_LINE(num_changed);
END;
/
```

Using PL/SQL Collections with SQL Object Types

Collections let you manipulate complex datatypes within PL/SQL. Your program can compute subscripts to process specific elements in memory, and use SQL to store the results in database tables.

In SQL*Plus, you can create SQL object types whose definitions correspond to PL/SQL nested tables and varrays, as shown in Example 4–15. Each item in column dept_names is a nested table that will store the department names for a specific region. The NESTED TABLE clause is required whenever a database table has a nested table column. The clause identifies the nested table and names a system-generated store table, in which Oracle stores the nested table data.

Within PL/SQL, you can manipulate the nested table by looping through its elements, using methods such as TRIM or EXTEND, and updating some or all of the elements. Afterwards, you can store the updated table in the database again. You can insert table rows containing nested tables, update rows to replace its nested table, and select nested tables into PL/SQL variables. You cannot update or delete individual nested table elements directly with SQL; you have to select the nested table from the table, change it in PL/SQL, then update the table to include the new nested table.

Example 4–15 Using INSERT, UPDATE, DELETE, and SELECT Statements With Nested Tables

```
CREATE TYPE dnames_tab AS TABLE OF VARCHAR2(30);
/
CREATE TABLE depts (region VARCHAR2(25), dept_names dnames_tab)
   NESTED TABLE dept_names STORE AS dnames_nt;
BEGIN
   INSERT INTO depts VALUES('Europe', dnames_tab('Shipping','Sales','Finance'));
   INSERT INTO depts VALUES('Americas', dnames_tab('Sales','Finance','Shipping'));
   INSERT INTO depts VALUES('Asia', dnames_tab('Finance','Payroll'));
   COMMIT;
END;
/
DECLARE
-- Type declaration is not needed, because PL/SQL can access the SQL object type
-- TYPE dnames_tab IS TABLE OF VARCHAR2(30); not needed
-- Declare a variable that can hold a set of department names
   v_dnames dnames_tab;
-- Declare a record that can hold a row from the table
-- One of the record fields is a set of department names
   v_depts depts%ROWTYPE;
   new_dnames dnames_tab;
BEGIN
-- Look up a region and query just the associated department names
   SELECT dept_names INTO v_dnames FROM depts WHERE region = 'Europe';
   FOR i IN v_dnames.FIRST .. v_dnames.LAST
   LOOP
      DBMS_OUTPUT.PUT_LINE('Department names: ' || v_dnames(i));
   END LOOP;
-- Look up a region and query the entire row
   SELECT * INTO v_depts FROM depts WHERE region = 'Asia';
-- Now dept_names is a field in a record, so we access it with dot notation
   FOR i IN v_depts.dept_names.FIRST .. v_depts.dept_names.LAST LOOP
-- Because we have all the table columns in the record, we can refer to region
     DBMS_OUTPUT.PUT_LINE(v_depts.region || ' dept_names = ' ||
                          v_depts.dept_names(i));
   END LOOP;
-- We can replace a set of department names with a new collection
-- in an UPDATE statement
   new_dnames := dnames_tab('Sales','Payroll','Shipping');
   UPDATE depts SET dept_names = new_dnames WHERE region = 'Europe';
-- Or we can modify the original collection and use it in the UPDATE.
-- We'll add a new final element and fill in a value
   v_depts.dept_names.EXTEND(1);
   v_depts.dept_names(v_depts.dept_names.COUNT) := 'Finance';
   UPDATE depts SET dept_names = v_depts.dept_names
     WHERE region = v_depts.region;
-- We can even treat the nested table column like a real table and
-- insert, update, or delete elements. The TABLE operator makes the statement
-- apply to the nested table produced by the subquery.
   INSERT INTO TABLE(SELECT dept_names FROM depts WHERE region = 'Asia')
     VALUES('Sales');
   DELETE FROM TABLE(SELECT dept_names FROM depts WHERE region = 'Asia')
      WHERE column_value = 'Payroll';
   UPDATE TABLE(SELECT dept_names FROM depts WHERE region = 'Americas')
      SET column_value = 'Payroll' WHERE column_value = 'Finance';
   COMMIT;
END;
/
```

Example 4–16 shows how you can manipulate SQL varray object types with PL/SQL statements. In this example, varrays are transferred between PL/SQL variables and SQL tables. You can insert table rows containing varrays, update a row to replace its varray, and select varrays into PL/SQL variables. You cannot update or delete individual varray elements directly with SQL; you have to select the varray from the table, change it in PL/SQL, then update the table to include the new varray.

Example 4–16 Using INSERT, UPDATE, DELETE, and SELECT Statements With Varrays

```
-- By using a varray, we put an upper limit on the number of elements
-- and ensure they always come back in the same order
CREATE TYPE dnames_var IS VARRAY(7) OF VARCHAR2(30);
/
CREATE TABLE depts (region VARCHAR2(25), dept_names dnames_var);
BEGIN
   INSERT INTO depts VALUES('Europe', dnames_var('Shipping','Sales','Finance'));
   INSERT INTO depts VALUES('Americas', dnames_var('Sales','Finance','Shipping'));
   INSERT INTO depts
     VALUES('Asia', dnames_var('Finance','Payroll','Shipping','Sales'));
   COMMIT;
END;
/
DECLARE
   new_dnames dnames_var := dnames_var('Benefits', 'Advertising', 'Contracting',
                                       'Executive', 'Marketing');
   some_dnames dnames_var;
BEGIN
   UPDATE depts SET dept_names  = new_dnames WHERE region = 'Europe';
   COMMIT;
   SELECT dept_names INTO some_dnames FROM depts WHERE region = 'Europe';
   FOR i IN some_dnames.FIRST .. some_dnames.LAST
   LOOP
      DBMS_OUTPUT.PUT_LINE('dept_names = ' || some_dnames(i));
   END LOOP;
END;
/
```

In Example 4–17, PL/SQL BULK COLLECT is used with a multilevel collection that includes an object type.

Example 4–17 Using BULK COLLECT with Nested Tables

```
CREATE TYPE dnames_var IS VARRAY(7) OF VARCHAR2(30);
/
CREATE TABLE depts (region VARCHAR2(25), dept_names dnames_var);
BEGIN
   INSERT INTO depts VALUES('Europe', dnames_var('Shipping','Sales','Finance'));
   INSERT INTO depts VALUES('Americas', dnames_var('Sales','Finance','Shipping'));
   INSERT INTO depts
     VALUES('Asia', dnames_var('Finance','Payroll','Shipping','Sales'));
   COMMIT;
END;
/
DECLARE
   TYPE dnames_tab IS TABLE OF dnames_var;
   v_depts dnames_tab;
BEGIN
    SELECT dept_names BULK COLLECT INTO v_depts FROM depts;
    DBMS_OUTPUT.PUT_LINE(v_depts.COUNT); -- prints 3
END;
```

```
/
```

Using Dynamic SQL With Objects

Example 4–18 illustrates the use of objects and collections with dynamic SQL. First, define object type person_typ and VARRAY type hobbies_var, then write a package that uses these types.

Example 4–18 TEAMS Package Using Dynamic SQL for Object Types and Collections

```
CREATE TYPE person_typ AS OBJECT (name VARCHAR2(25), age NUMBER);
/
CREATE TYPE hobbies_var AS VARRAY(10) OF VARCHAR2(25);
/
CREATE OR REPLACE PACKAGE teams
   AUTHID CURRENT_USER AS
   PROCEDURE create_table (tab_name VARCHAR2);
   PROCEDURE insert_row (tab_name VARCHAR2, p person_typ, h hobbies_var);
   PROCEDURE print_table (tab_name VARCHAR2);
END;
/
CREATE OR REPLACE PACKAGE BODY teams AS
   PROCEDURE create_table (tab_name VARCHAR2) IS
   BEGIN
      EXECUTE IMMEDIATE 'CREATE TABLE ' || tab_name ||
                        ' (pers person_typ, hobbs hobbies_var)';
   END;
   PROCEDURE insert_row (
      tab_name VARCHAR2,
      p person_typ,
      h hobbies_var) IS
   BEGIN
      EXECUTE IMMEDIATE 'INSERT INTO ' || tab_name ||
         ' VALUES (:1, :2)' USING p, h;
   END;
   PROCEDURE print_table (tab_name VARCHAR2) IS
      TYPE  refcurtyp IS REF CURSOR;
      v_cur refcurtyp;
      p     person_typ;
      h     hobbies_var;
   BEGIN
      OPEN v_cur FOR 'SELECT pers, hobbs FROM ' || tab_name;
      LOOP
         FETCH v_cur INTO p, h;
         EXIT WHEN v_cur%NOTFOUND;
         -- print attributes of 'p' and elements of 'h'
         DBMS_OUTPUT.PUT_LINE('Name: ' || p.name || ' - Age: ' || p.age);
         FOR i IN h.FIRST..h.LAST
         LOOP
           DBMS_OUTPUT.PUT_LINE('Hobby(' || i || '): ' || h(i));
         END LOOP;
      END LOOP;
      CLOSE v_cur;
   END;
END;
/
```

From an anonymous block, you might call the procedures in package TEAMS:

Example 4–19 Calling Procedures from the TEAMS Package

```
DECLARE
   team_name VARCHAR2(15);
BEGIN
   team_name := 'Notables';
   TEAMS.create_table(team_name);
   TEAMS.insert_row(team_name, person_typ('John', 31),
      hobbies_var('skiing', 'coin collecting', 'tennis'));
   TEAMS.insert_row(team_name, person_typ('Mary', 28),
      hobbies_var('golf', 'quilting', 'rock climbing', 'fencing'));
   TEAMS.print_table(team_name);
END;
/
```

5

Object Support in Oracle Programming Environments

In an Oracle database, 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 application programming environments.

This chapter discusses the following topics:

- SQL
- PL/SQL
- Oracle Call Interface (OCI)
- Pro*C/C++
- Oracle C++ Call Interface (OCCI)
- Oracle Objects For OLE (OO4O)
- Java: JDBC, Oracle SQLJ, JPublisher, and SQLJ Object Types
- XML

SQL

Oracle SQL DDL provides the following support for object types:

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

Oracle SQL DML provides the following support for object types:

- Querying and updating objects and collections
- Manipulating `REF`s

> **See Also:** For a complete description of Oracle SQL syntax, see *Oracle Database SQL Language Reference*

PL/SQL

Object types and subtypes can be used in PL/SQL procedures and functions in most places where built-in types can appear.

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 the *Oracle Database 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 `REF`s.
- 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 data types 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 enables 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 "Associative Access in Pro*C/C++" on page 5-5.

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 enables 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 "Navigational Access in Pro*C/C++" on page 5-5.

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:** *Oracle Call Interface Programmer's Guide* for tips and techniques for using OCI program effectively with objects

Defining User-Defined Constructors in C

When defining a user-defined constructor in C, you must specify SELF (and you may optionally specify SELF TDO) in the PARAMETERS clause. On entering the C function, the attributes of the C structure that the object maps to are all initialized to NULL. The value returned by the function is mapped to an instance of the user-defined type. Example 5–1 shows how to define a user-defined constructor in C.

Example 5–1 Defining a User-Defined Constructor in C

```
CREATE LIBRARY person_lib TRUSTED AS STATIC
/

CREATE TYPE person AS OBJECT
  (  name VARCHAR2(30),
     CONSTRUCTOR FUNCTION person(SELF IN OUT NOCOPY person, name VARCHAR2)
         RETURN SELF AS RESULT);
/

CREATE TYPE BODY person IS
    CONSTRUCTOR FUNCTION person(SELF IN OUT NOCOPY person, name VARCHAR2)
         RETURN SELF AS RESULT
    IS EXTERNAL NAME "cons_person_typ" LIBRARY person_lib WITH CONTEXT
    PARAMETERS(context, SELF, name OCIString, name INDICATOR sb4);
END;
/
```

The SELF parameter is mapped like an IN parameter, so in the case of a NOT FINAL type, it is mapped to (dvoid *), not (dvoid **).

The return value's TDO must match the TDO of SELF and is therefore implicit. The return value can never be null, so the return indicator is implicit as well.

Pro*C/C++

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

Pro*C developers can use the Object Type Translator to map Oracle object types and collections into C data types 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 data types.

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 5-2), where objects can be accessed by traversing pointers, then modified and updated on the server.

> **See Also:** For a complete description of the Pro*C precompiler, see *Pro*C/C++ Programmer's Guide.*

Associative Access in Pro*C/C++

For background information on associative access, see "Associative Access in OCI Programs" on page 5-2.

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 5-3.

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.

> **See Also:** For complete information about OTT, see *Oracle Call Interface Programmer's Guide* and *Pro*C/C++ Programmer's Guide*.

Oracle C++ Call Interface (OCCI)

The Oracle C++ Call Interface (OCCI) is a C++ API that enables you to use the object-oriented features, native classes, and methods of the C++ programing language to access the Oracle database.

The OCCI interface is modeled on the JDBC interface and, like the JDBC interface, is easy to use. OCCI itself is built on top of OCI and provides the power and performance of OCI using an object-oriented paradigm.

OCI is a C API to the Oracle database. It supports the entire Oracle feature set and provides efficient access to both relational and object data, but it can be challenging to use—particularly if you want to work with complex, object data types. Object types are not natively supported in C, and simulating them in C is not easy. OCCI addresses this by providing a simpler, object-oriented interface to the functionality of OCI. It does this by defining a set of wrappers for OCI. By working with these higher-level abstractions, developers can avail themselves of the underlying power of OCI to manipulate objects in the server through an object-oriented interface that is significantly easier to program.

The Oracle C++ Call Interface, OCCI, can be roughly divided into three sets of functionalities, namely:

- Associative relational access
- Associative object access
- Navigational access

OCCI Associative Relational and Object Interfaces

The associative relational API and object classes provide SQL access to the database. Through these interfaces, SQL is executed on the server to create, manipulate, and fetch object or relational data. Applications can access any data type on the server, including the following:

- Large objects
- Objects/Structured types
- Arrays
- References

The OCCI Navigational Interface

The navigational interface is a C++ interface that lets you seamlessly access and modify object-relational data in the form of C++ objects without using SQL. The C++ objects are transparently accessed and stored in the database as needed.

With the OCCI navigational interface, you can retrieve an object and navigate through references from that object to other objects. Server objects are materialized as C++ class instances in the application cache.

An application can use OCCI object navigational calls to perform the following functions on the server's objects:

- Create, access, lock, delete, and flush objects
- Get references to the objects and navigate through them

> **See Also:** *Oracle C++ Call Interface Programmer's Guide* for a complete account of how to build applications with the Oracle C++ API

Oracle Objects For OLE (OO4O)

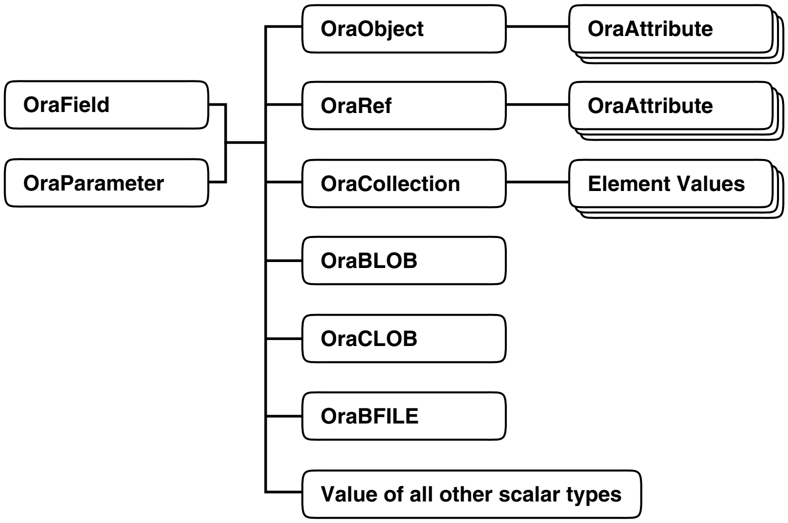
Oracle Objects for OLE (OO4O) provides full support for accessing and manipulating instances of `REF`s, value instances, variable-length arrays (`VARRAY`s), and nested tables in an Oracle database server.

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, ActiveX, and Active Server Pages.

> **See Also:** The "OO4O Automation Server Reference" section of the Oracle Objects for OLE online help or *Oracle Objects for OLE Developer's Guide* online documentation for detailed information and examples on using OO4O with Oracle objects

Figure 5–1 illustrates the containment hierarchy for value instances of all types in OO4O.

Figure 5–1 Supported Oracle Data Types



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 by means of 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.

> **See Also:** *Oracle Call Interface Programmer's Guide* for tips and techniques for using OCI program effectively with objects

Representing VARRAYs and Nested 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
'The following loop accesses all elements of
'the EnameList VArray
For I=1 to I=EnameList.Size
    Msgbox EnameList(I)
Next I
```

Java: JDBC, Oracle SQLJ, JPublisher, and SQLJ Object Types

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. These interfaces enable you both to access SQL data from Java and to provide persistent database storage for Java objects.

For an example of using Java APIs with Oracle objects, see the Oracle by Example Series available on the Oracle Technology Network (OTN) Web site:

http://otn.oracle.com/products/oracle9i/htdocs/9iobe/OBE9i-Public/obe-dev/html/objects/objects.htm

You can use the followings steps to navigate to the Oracle objects module in the Oracle by Example Series.

- Go to http://www.oracle.com/technology/
- Select **Oracle Database** under **Products** from the menu on the left side of the page
- Select **Oracle9i Database Release 1** under **Previous Releases** on the right side of the page
- Under **Technical Information**, select **Oracle9i by Example Series**

- Select **Build Application Components** from the menu on the left side of the page
- Select the **Using Objects to Build an Online Product Catalog** example

You can also search for **Using Objects to Build an Online Product Catalog** on the OTN Web site at `http://www.oracle.com/technology/`.

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 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.
- Manually 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 `ORAData` interface.

See Also: For complete information about JDBC, see the *Oracle Database 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.

See Also: For complete information about SQLJ, see the *Oracle Database 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 (`REF`s), and collection types (varrays 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, `REF` type, or collection type and must implement the interface `oracle.sql.ORAData`. 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).

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 if the classes meet the requirements stated in the *Oracle Database JPublisher User's Guide*.

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

What JPublisher Produces for a User-Defined Object Type

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 `getXxx()` and `setXxx()` for attribute `xxx`.

  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 `ORAData` interface and the `getFactory()` method.

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

Java Object Storage

JPublisher enables you to construct Java classes that map to existing SQL types. You can then access the SQL types from a Java application using JDBC.

You can also go in the other direction. That is, you can create SQL types that map to existing Java classes. This capability enables you to provide persistent storage for Java objects. Such SQL types are called SQL types of Language Java, or SQLJ object types. They can be used as the type of an object, an attribute, a column, or a row in an object table. You can navigationally access objects of such types—Java objects—through either object references or foreign keys, and you can query and manipulate such objects from SQL.

You create SQLJ types with a `CREATE TYPE` statement as you do other user-defined SQL types. For SQLJ types, two special elements are added to the `CREATE TYPE` statement:

- An `EXTERNAL NAME` phrase, used to identify the Java counterpart for each SQLJ attribute and method and the Java class corresponding to the SQLJ type itself
- A `USING` clause, to specify how the SQLJ type is to be represented to the server. The `USING` clause specifies the interface used to retrieve a SQLJ type and the kind of storage.

For example:

Example 5–2 Mapping SQL Types to Java Classes

```
CREATE TYPE full_address AS OBJECT (a NUMBER);
/

CREATE OR REPLACE TYPE person_t AS OBJECT
  EXTERNAL NAME 'Person' LANGUAGE JAVA
  USING SQLData (
    ss_no NUMBER (9) EXTERNAL NAME 'socialSecurityNo',
    name varchar(100) EXTERNAL NAME 'name',
    address full_address EXTERNAL NAME 'addrs',
    birth_date date EXTERNAL NAME 'birthDate',
    MEMBER FUNCTION age  RETURN NUMBER EXTERNAL NAME 'age () return int',
    MEMBER FUNCTION addressf RETURN full_address
      EXTERNAL NAME 'get_address () return long_address',
    STATIC function createf RETURN person_t EXTERNAL NAME 'create ()
         return Person',
    STATIC function createf (name VARCHAR2, addrs full_address, bDate DATE)
      RETURN person_t EXTERNAL NAME 'create (java.lang.String, Long_address,
      oracle.sql.date) return Person',
    ORDER member FUNCTION compare (in_person person_t) RETURN NUMBER
      EXTERNAL NAME 'isSame (Person) return int')
/
```

SQLJ types use the corresponding Java class as the body of the type; you do not specify a type body in SQL to contain implementations of the type's methods as you do with ordinary object types.

Representing SQLJ Types to the Server

How a SQLJ type is represented to the server and stored depends on the interfaces implemented by the corresponding Java class. Currently, Oracle supports a representation of SQLJ types only for Java classes that implement a `SQLData` or `ORAData` interface. These are represented to the server and are accessible through

SQL. A representation for Java classes that implement the java.io.Serializable interface is not currently supported.

In a SQL representation, the attributes of the type are stored in columns like attributes of ordinary object types. With this representation, all attributes are public because objects are accessed and manipulated through SQL statements, but you can use triggers and constraints to ensure the consistency of the object data.

For a SQL representation, the USING clause must specify either SQLData or ORAData, and the corresponding Java class must implement one of those interfaces. The EXTERNAL NAME clause for attributes is optional.

Creating SQLJ Object Types

The SQL statements to create SQLJ types and specify their mappings to Java are placed in a file called a **deployment descriptor**. Related SQL constraints and privileges are also specified in this file. The types are created when the file is executed.

Below is an overview of the process of creating SQL versions of Java types/classes:

1. Design the Java types.
2. Generate the Java classes.
3. Create the SQLJ object type statements.
4. Construct the JAR file. This is a single file that contains all the classes needed.
5. Using the loadjava utility, install the Java classes defined in the JAR file.
6. Execute the statements to create the SQLJ object types.

Additional Notes About Mapping

The following are additional notes to consider when mapping of Java classes to SQL types:

- You can map a SQLJ static function to a user-defined constructor in the Java class. The return value of this function is of the user-defined type in which the function is locally defined.
- Java static variables are mapped to SQLJ static methods that return the value of the corresponding static variable identified by EXTERNAL NAME. The EXTERNAL NAME clause for an attribute is optional with a SQLData or ORAData representation.
- Every attribute in a SQLJ type of a SQL representation must map to a Java field, but not every Java field must be mapped to a corresponding SQLJ attribute: you can omit Java fields from the mapping.
- You can omit classes: you can map a SQLJ type to a non-root class in a Java class hierarchy without also mapping SQLJ types to the root class and intervening superclasses. Doing this enables you to hide the superclasses while still including attributes and methods inherited from them.

  However, you must preserve the structural correspondence between nodes in a class hierarchy and their counterparts in a SQLJ type hierarchy. In other words, for two Java classes j_A and j_B that are related through inheritance and are mapped to two SQL types s_A and s_B, respectively, there must be exactly one corresponding node on the inheritance path from s_A to s_B for each node on the inheritance path from j_A to j_B.
- You can map a Java class to multiple SQLJ types as long as you do not violate the restriction in the preceding paragraph. In other words, no two SQLJ types mapped to the same Java class can have a common supertype ancestor.

- If all Java classes are not mapped to SQLJ types, it is possible that an attribute of a SQLJ object type might be set to an object of an unmapped Java class. Specifically, to a class occurring above or below the class to which the attribute is mapped in an inheritance hierarchy. If the object's class is a superclass of the attribute's type/class, an error is raised. If it is a subclass of the attribute's type/class, the object is mapped to the most specific type in its hierarchy for which a SQL mapping exists

> **See Also:** The *Oracle Database JPublisher User's Guide* for JPublisher examples of object mapping

Evolving SQLJ Types

The ALTER TYPE statement enables you to evolve a type by, for example, adding or dropping attributes or methods.

When a SQLJ type is evolved, an additional validation is performed to check the mapping between the class and the type. If the class and the evolved type match, the type is marked valid. Otherwise, the type is marked as pending validation.

Being marked as pending validation is not the same as being marked invalid. A type that is pending validation can still be manipulated with ALTER TYPE and GRANT statements, for example.

If a type that has a SQL representation is marked as pending evaluation, you can still access tables of that type using any DML or SELECT statement that does not require a method invocation.

You cannot, however, execute DML or SELECT statements on tables of a type that has a serializable representation and has been marked as pending validation. Data of a serializable type can be accessed only navigationally, through method invocations. These are not possible with a type that is pending validation. However, you can still re-evolve the type until it passes validation.

See "Type Evolution" on page 8-5.

Constraints

For SQLJ types having a SQL representation, the same constraints can be defined as for ordinary object types.

Constraints are defined on tables, not on types, and are defined at the column level. The following constraints are supported for SQLJ types having a SQL representation:

- Unique constraints
- Primary Key
- Check constraints
- NOT NULL constraints on attributes
- Referential constraints

The IS OF TYPE constraint on column substitutability is supported, too, for SQLJ types having a SQL representation. See "Constraining Substitutability" on page 2-28.

Querying SQLJ Objects

SQLJ types can be queried just like ordinary SQL object types. Methods called in a SELECT statement must not attempt to change attribute values.

Inserting Java Objects

Inserting a row in a table containing a column of a SQLJ type requires a call to the type's constructor function to create a Java object of that type.

The implicit, system-generated constructor can be used, or a static function can be defined that maps to a user-defined constructor in the Java class.

Updating SQLJ Objects

SQLJ objects can be updated either by using an `UPDATE` statement to modify the value of one or more attributes, or by invoking a method that updates the attributes and returns `SELF`—that is, returns the object itself with the changes made.

For example, suppose that `raise()` is a member function that increments the `salary` field/attribute by a specified amount and returns `SELF`. The following statement gives every employee in the object table `employee_objtab` a raise of `1000`:

```
UPDATE employee_objtab SET c=c.raise(1000);
```

A column of a SQLJ type can be set to `NULL` or to another column using the same syntax as for ordinary object types. For example, the following statement assigns column `d` to column `c`:

```
UPDATE employee_reltab SET c=d;
```

Defining User-Defined Constructors in Java

When you implement a user-defined constructor in Java, the string supplied as the implementing routine must correspond to a static function. For the return type of the function, specify the Java type mapped to the SQL type.

Example 5–3 is an example of a type declaration that involves a user-defined constructor implemented in Java.

Example 5–3 Defining a User-Defined Constructor in Java

```
CREATE TYPE person1_typ AS OBJECT
 EXTERNAL NAME 'pkg1.J_Person' LANGUAGE JAVA
 USING SQLData(
  name VARCHAR2(30),
  age NUMBER,
  CONSTRUCTOR FUNCTION person1_typ(SELF IN OUT NOCOPY person1_typ, name VARCHAR2,
                                   age NUMBER) RETURN SELF AS RESULT
  AS LANGUAGE JAVA
    NAME 'pkg1.J_Person.J_Person(java.lang.String, int) return J_Person')
/
```

XML

`XMLType` views wrap existing relational and object-relational data in XML formats. These views are similar to object views. Each row of an `XMLType` view corresponds to an `XMLType` instance. The object identifier for uniquely identifying each row in the view can be created using an expression such as `extract()` on the `XMLType` value.

> **See Also:** *Oracle XML DB Developer's Guide* for information and examples on using XML with Oracle objects

6

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.

The chapter contains these topics:

Why 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 can be used like relational views to present only the data that you want users to see. For example, you might create an object view that presents selected data from an employee table but omits sensitive data about salaries.

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 structures 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.
- Because 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:

- *Oracle Database SQL Language Reference* for a complete description of SQL syntax and usage.
- *Oracle Database PL/SQL User's Guide and Reference* for a complete discussion of PL/SQL capabilities
- *Oracle Database Java 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 enables 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:

Write an INSTEAD OF trigger procedure for Oracle to execute whenever an application program tries to update data in the object view. See "Updating Object Views" on page 6-9.

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:

Example 6–1 Creating an Object View

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

CREATE TYPE employee_t AS OBJECT (
    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. The instance appears in the client side object cache 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 in 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.

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

Nesting Objects in Object Views

An object type can have other object types nested in it as attributes.

If the object type on which an object view is based has an attribute that itself is an object type, then you must provide column objects for this attribute as part of the process of creating the object view. If column objects of the attribute type already exist in a relational table, you can simply select them; otherwise, you must synthesize the object instances from underlying relational data just as you synthesize the principal object instances of the view. You synthesize, or create, these objects by calling the respective object type's constructor method to create the object instances, and you populate their attributes with data from relational columns that you specify in the constructor.

For example, consider the department table `dept` in Example 6–2. 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.

First, create the types for the address and department objects, then create the view containing the department number, name and address. The `address` objects are constructed from columns of the relational table.

Example 6–2 Creating a View with Nested Object Types

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

CREATE TYPE address_t AS OBJECT (
   street   VARCHAR2(20),
    city    VARCHAR2(10),
    state   CHAR(2),
    zip     VARCHAR2(10));
/
CREATE TYPE dept_t AS OBJECT (
   deptno     NUMBER,
   deptname   VARCHAR2(20),
   address    address_t );
/

CREATE VIEW dept_view OF dept_t WITH OBJECT IDENTIFIER (deptno) 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 preceding 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:

Example 6–3 Identifying Null Objects in an Object View

```
CREATE OR REPLACE 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.

Single-Level Collections in Object Views

Using Example 6–2 as our starting point, we represent each employee in an emp relational table that has the following structure in Example 6–4. 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.

First, define a nested table type for the employee type employee_t. Next, define a department type having a department number, name, address, and a nested table of employees. Finally, define the object view dept_view.

Example 6–4 Creating a View with a Single-Level Collection

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

CREATE TYPE employee_list_t AS TABLE OF employee_t;
/
CREATE TYPE dept_t AS OBJECT (
    deptno     NUMBER,
    deptname   VARCHAR2(20),
    address    address_t,
    emp_list   employee_list_t);
/
CREATE VIEW dept_view OF dept_t WITH OBJECT IDENTIFIER (deptno) 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, e.job
                           FROM emp e
                           WHERE e.deptno = d.deptno)
                        AS employee_list_t)
                   AS emp_list
   FROM dept d;
```

The SELECT subquery inside the CAST-MULTISET block selects the list of employees that belong to the current department. The MULTISET 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 nested table 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.

Multilevel Collections in Object Views

Multilevel collections and single-level collections are created and used in object views in the same way. The only difference is that, for a multilevel collection, you must create an additional level of collections.

Example 6–5 builds an object view containing a multilevel collection. The view is based on flat relational tables that contain no collections. As a preliminary to building the object view, the example creates the object and collection types it uses. An object type (for example, `emp_t`) is defined to correspond to each relational table, with attributes whose types correspond to the types of the respective table columns. In addition, the employee type has a nested table (attribute) of projects, and the department type has a nested table (attribute) of employees. The latter nested table is a multilevel collection. The `CAST-MULTISET` operator is used in the `CREATE VIEW` statement to build the collections.

Example 6–5 Creating a View with Multilevel Collections

```
CREATE TABLE depts
  ( deptno     NUMBER,
    deptname   VARCHAR2(20));

CREATE TABLE emps
  ( ename VARCHAR2(20),
    salary     NUMBER,
    deptname   VARCHAR2(20));

CREATE TABLE projects
  ( projname   VARCHAR2(20),
    mgr        VARCHAR2(20));

CREATE TYPE project_t AS OBJECT
  ( projname   VARCHAR2(20),
    mgr        VARCHAR2(20));
/
CREATE TYPE nt_project_t AS TABLE OF project_t;
/
CREATE TYPE emp_t AS OBJECT
(  ename      VARCHAR2(20),
   salary     NUMBER,
   deptname   VARCHAR2(20),
   projects   nt_project_t );
/
CREATE TYPE nt_emp_t AS TABLE OF emp_t;
/
CREATE TYPE depts_t AS OBJECT
  ( deptno     NUMBER,
    deptname   VARCHAR2(20),
    emps       nt_emp_t );
/
CREATE VIEW v_depts OF depts_t WITH OBJECT IDENTIFIER (deptno) AS
  SELECT d.deptno, d.deptname,
    CAST(MULTISET(SELECT e.ename, e.salary, e.deptname,
        CAST(MULTISET(SELECT p.projname, p.mgr
          FROM projects p
          WHERE p.mgr = e.ename)
        AS nt_project_t)
      FROM emps e
      WHERE e.deptname = d.deptname)
    AS nt_emp_t)
  FROM depts d;
```

Specifying Object Identifiers for Object Views

You can construct pointers (REFs) to the row objects in an object view. Because 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, because 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.

For example, note the definition of the object type dept_t and the object view dept_view described in "Single-Level Collections in Object Views" on page 6-5.

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 deptno attribute of the object type. Once we know that deptno is unique within the view objects, we can specify it as the object identifier.

See "Storage Considerations for Object Identifiers (OIDs)" on page 9-4.

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 as shown in Example 6–6.

Example 6–6 Creating a Reference to Objects in a View

```
CREATE TYPE emp_t AS OBJECT (
  empno    NUMBER,
  ename    VARCHAR2(20),
  salary   NUMBER,
  deptref  REF dept_t);
/
CREATE OR REPLACE VIEW emp_view OF emp_t WITH OBJECT IDENTIFIER(empno)
   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. The following simple query retrieves all employees whose department is located in the city of San Francisco:

```
SELECT e.empno, e.salary, e.deptref.deptno
  FROM emp_view e
 WHERE e.deptref.address.city = 'San Francisco';
```

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

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

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 Many-to-One 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 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 directly updatable if its view query contains joins, set operators, aggregate functions, or GROUP BY or DISTINCT clauses. Also, individual columns of a view are not directly updatable if they are based on pseudocolumns or expression in the view query.

If a view is not directly updatable, you can still update it indirectly using INSTEAD OF triggers. To do so, you define an INSTEAD OF trigger for each kind of DML statement you want to execute on the view. In the INSTEAD OF trigger, you code the operations that must take place on the underlying tables of the view to accomplish the desired change in the view. Then, when you issue a DML statement for which you have defined an INSTEAD OF trigger, Oracle transparently runs the associated trigger. See "Using INSTEAD OF Triggers to Control Mutating and Validation" on page 6-10 for an example of an INSTEAD OF trigger.

Something you want to be careful of: In an object view hierarchy, UPDATE and DELETE statements operate polymorphically just as SELECT statements do: the set of rows picked out by an UPDATE or DELETE statement on a view implicitly includes qualifying rows in any subviews of the specified view as well. See "Object View Hierarchies" on page 6-14 for a discussion of object view hierarchy and examples defining Student_v and Employee_v.

For example, the following statement, which deletes all persons from Person_v, also deletes all students from Student_v and all employees from the Employee_v view.

```
DELETE FROM Person_v;
```

To exclude subviews and restrict the affected rows just to those in the view actually specified, use the ONLY keyword. For example, the following statement updates only persons and not employees or students.

```
UPDATE ONLY(Person_v) SET address = ...
```

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 because the view can be updated, but we need it to enforce the constraint.

We implement the trigger by means of the SQL statements in Example 6–7.

Example 6–7 Creating INSTEAD OF Triggers on a View

```
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.deptno;
  IF emp_count < 9 THEN
     -- Do the insert
     INSERT INTO emp (empno, empname, salary, deptno)
        VALUES (:NEW.empno, :NEW.ename, :NEW.salary, dept_var.deptno);
  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.

In Example 6–8, we begin by creating an object view for each employee table. Then we can create the global view.

Example 6–8 Creating an Object View to Access Remote Tables

```
CREATE VIEW emp_washington_view (eno, ename, salary, job)
   AS SELECT e.empno, e.empname, e.salary, e.job
          FROM emp@washington e;
```

```
CREATE VIEW emp_chicago_view (eno, ename, salary, job)
   AS SELECT e.empno, e.empname, e.salary, e.job
          FROM emp@chicago e;

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

This view has the list of all employees for each department. We use `UNION ALL` because we cannot have two employees working in more than one department.

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: Re-create First View After Creating Second View

  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: Create First View Using FORCE Keyword

  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: Re-create First View After Creating Second View

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( MULTISET (
                           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: Create First View Using FORCE Keyword

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 OR REPLACE 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, because emp_view already exists.

```
CREATE OR REPLACE 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( MULTISET (
                           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 preceding 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%';
```

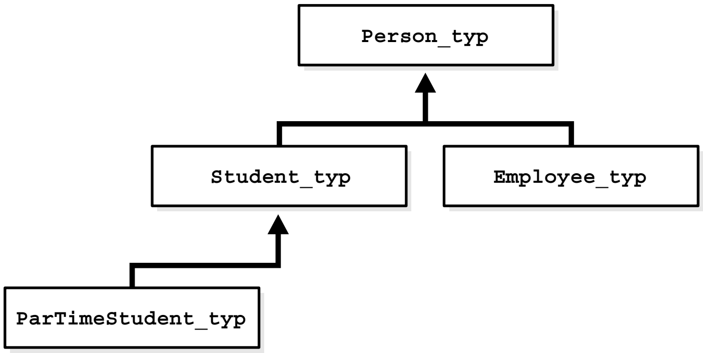
Object View Hierarchies

An object view hierarchy is a set of object views each of which is based on a different type in a type hierarchy. Subviews in a view hierarchy are created under a superview, analogously to the way subtypes in a type hierarchy are created under a supertype.

Each object view in a view hierarchy is populated with objects of a single type, but queries on a given view implicitly address its subviews as well. Thus an object view hierarchy gives you a simple way to frame queries that can return a polymorphic set of objects of a given level of specialization or greater.

For example, suppose you have the following type hierarchy, with person_typ as the root:

Figure 6–1 Object View Hierarchy



If you have created an object view hierarchy based on this type hierarchy, with an object view built on each type, you can query the object view that corresponds to the level of specialization you are interested in. For instance, you can query the view of `student_typ` to get a result set that contains only students, including part-time students.

You can base the root view of an object view hierarchy on any type in a type hierarchy: you do not need to start the object view hierarchy at the root type. Nor do you need to extend an object view hierarchy to every leaf of a type hierarchy or cover every branch. However, you cannot skip intervening subtypes in the line of descent. Any subview must be based on a direct subtype of the type of its direct superview.

Just as a type can have multiple sibling subtypes, an object view can have multiple sibling subviews. But a subview based on a given type can participate in only one object view hierarchy: two different object view hierarchies cannot each have a subview based on the same subtype.

A subview inherits the object identifier (OID) from its superview. An OID cannot be explicitly specified in any subview.

A root view can explicitly specify an object identifier using the `WITH OBJECT ID` clause. If the OID is system-generated or the clause is not specified in the root view, then subviews can be created only if the root view is based on a table or view that also uses a system generated OID.

The query underlying a view determines whether the view is updatable. For a view to be updatable, its query must contain no joins, set operators, aggregate functions, `GROUP BY`, `DISTINCT`, pseudocolumns, or expressions. The same applies to subviews.

If a view is not updatable, you can define `INSTEAD OF` triggers to perform appropriate DML actions. Note that `INSTEAD OF` triggers are not inherited by subviews.

All views in a view hierarchy must be in the same schema.

Note: You can create views of types that are non-instantiable. A non-instantiable type cannot have instances, so ordinarily there would be no point in creating an object view of such a type. However, a non-instantiable type can have subtypes that *are* instantiable. The ability to create object views of non-instantiable types enables you to base an object view hierarchy on a type hierarchy that contains a non-instantiable type.

Creating an Object View Hierarchy

You build an object view hierarchy by creating subviews under a root view. You do this by using the UNDER keyword in the CREATE VIEW statement, as show in Example 6–9.

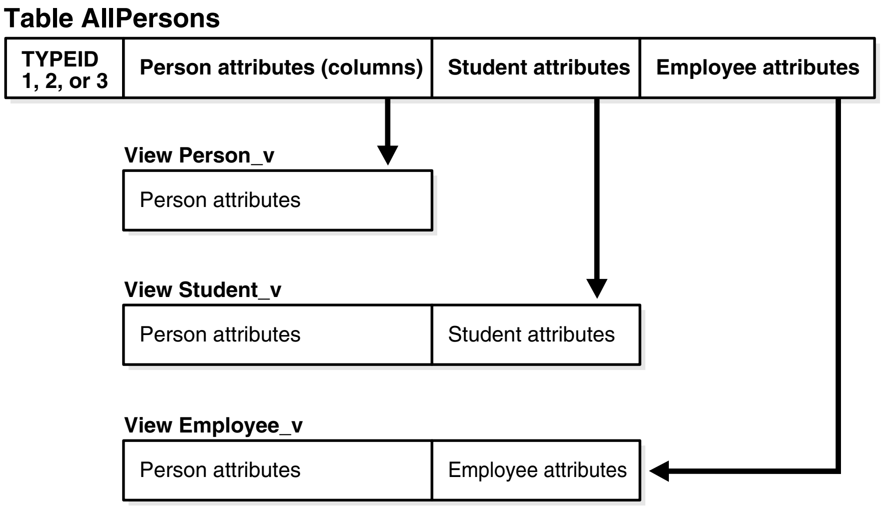
The same object view hierarchy can be based on different underlying storage models. In other words, a variety of layouts or designs of underlying tables can produce the same object view hierarchy. The design of the underlying storage model has implications for the performance and updatability of the object view hierarchy.

The following examples show three possible storage models. In the first, a flat model, all views in the object view hierarchy are based on the same table. In the second, a horizontal model, each view has a one-to-one correspondence with a different table. And in the third, a vertical model, the views are constructed using joins.

The Flat Model

In the flat model, all the views in the hierarchy are based on the same table. In the following example, the single table AllPersons contains columns for all the attributes of person_typ, student_typ, or employee_typ.

Figure 6–2 Flat Storage Model for Object View Hierarchy



```
CREATE TABLE AllPersons
( typeid NUMBER(1),
  ssn NUMBER,
  name VARCHAR2(30),
  address VARCHAR2(100),
  deptid NUMBER,
  major VARCHAR2(30),
  empid NUMBER,
  mgr VARCHAR2(30));
```

The typeid column identifies the type of each row. Possible values are:

```
-- 1 = person_typ
-- 2 = student_typ
-- 3 = employee_typ

CREATE TYPE person_typ AS OBJECT
( ssn NUMBER,
  name VARCHAR2(30),
  address VARCHAR2(100)) NOT FINAL;
/

CREATE TYPE student_typ UNDER person_typ
( deptid NUMBER,
   major VARCHAR2(30)) NOT FINAL;
/

CREATE TYPE employee_typ UNDER person_typ
( empid NUMBER,
  mgr VARCHAR2(30));
/
```

The following statements create the views that make up the object view hierarchy:

Example 6–9 Creating an Object View Hierarchy

```
CREATE VIEW Person_v OF person_typ
  WITH OBJECT OID(ssn) AS
  SELECT ssn, name, address
  FROM AllPersons
  WHERE typeid = 1;

CREATE VIEW Student_v OF student_typ UNDER Person_v
  AS
  SELECT ssn, name, address, deptid, major
  FROM AllPersons
  WHERE typeid = 2;

CREATE VIEW Employee_v OF employee_typ UNDER Person_v
  AS
  SELECT ssn, name, address, empid, mgr
  FROM AllPersons
  WHERE typeid = 3;
```

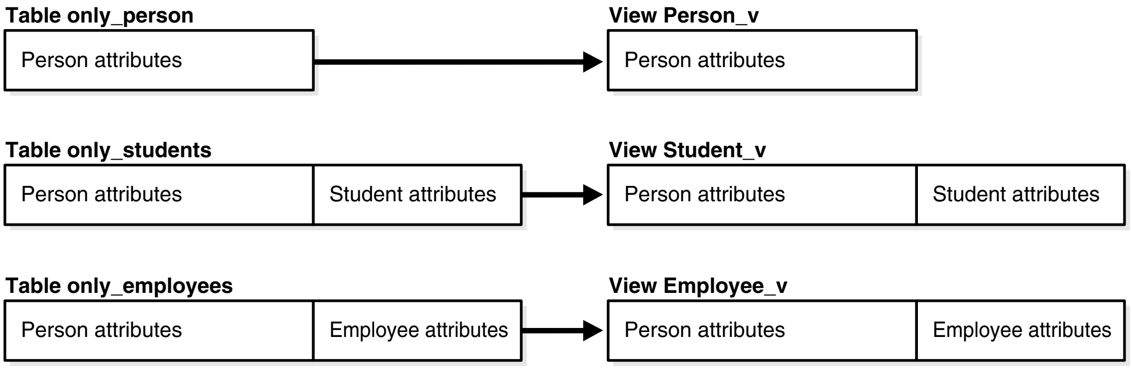
The flat model has the advantage of simplicity and poses no obstacles to supporting indexes and constraints. Its drawbacks are:

- A single table cannot contain more than 1000 columns, so the flat model imposes a 1000-column limit on the total number of columns that the object view hierarchy can contain.
- Each row of the table will have NULLs for all the attributes not belonging to its type. Such non-trailing NULLs can adversely affect performance.

The Horizontal Model

On the horizontal model, each view or subview is based on a different table. In the example, the tables are relational, but they could just as well be object tables for which column substitutability is turned off.

Figure 6–3 Horizontal Storage Model for Object View Hierarchy



```
CREATE TABLE only_persons
( ssn NUMBER,
  name VARCHAR2(30),
  address VARCHAR2(100));

CREATE TABLE only_students
( ssn NUMBER,
  name VARCHAR2(30),
  address VARCHAR2(100),
  deptid NUMBER,
  major VARCHAR2(30));

CREATE TABLE only_employees
( ssn NUMBER,
  name VARCHAR2(30),
  address VARCHAR2(100),
  empid NUMBER,
  mgr VARCHAR2(30));
```

These are the views:

```
CREATE OR REPLACE VIEW Person_v OF person_typ
  WITH OBJECT OID(ssn) AS
  SELECT *
  FROM only_persons

CREATE OR REPLACE VIEW Student_v OF student_typ UNDER Person_v
  AS
  SELECT *
  FROM only_students;

CREATE OR REPlACE VIEW Employee_v OF employee_typ UNDER Person_v
  AS
  SELECT *
  FROM only_employees;
```

The horizontal model is very efficient at processing queries of the form:

```
SELECT VALUE(p) FROM Person_v p
  WHERE VALUE(p) IS OF (ONLY student_typ);
```

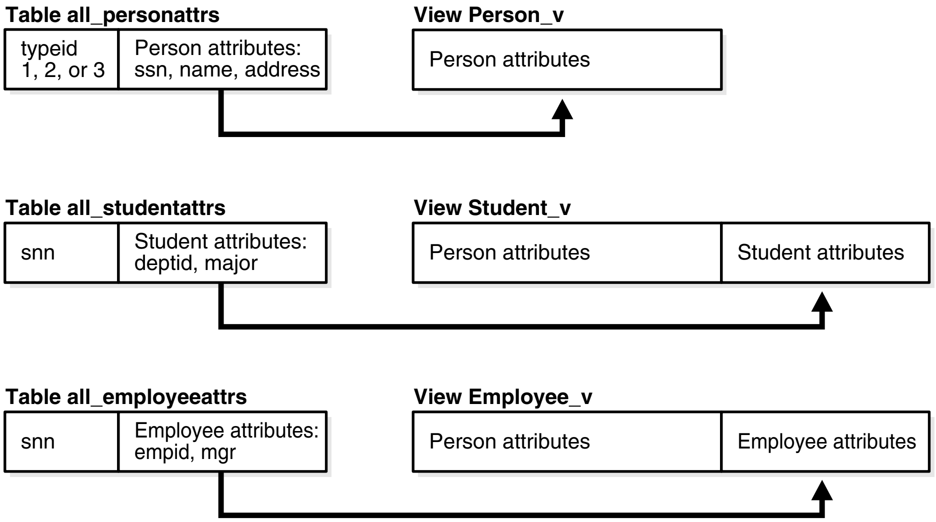
Such queries need access only a single physical table to get all the objects of the specific type. The drawbacks of this model are that queries of the sort `SELECT * FROM` *view* require performing a `UNION` over all the underlying tables and projecting the rows over just the columns in the specified view. (See "Querying a View in a

Hierarchy" on page 6-20.) Also, indexes on attributes (and unique constraints) must span multiple tables, and support for this does not currently exist.

The Vertical Model

In the vertical model, there is a physical table corresponding to each view in the hierarchy, but each physical table stores only those attributes that are unique to its corresponding subtype.

Figure 6–4 Vertical Storage Model for Object View Hierarchy



```
CREATE TABLE all_personattrs
( typeid NUMBER,
  ssn NUMBER,
  name VARCHAR2(30),
  address VARCHAR2(100));

CREATE TABLE all_studentattrs
( ssn NUMBER,
  deptid NUMBER,
  major VARCHAR2(30));

CREATE TABLE all_employeeattrs
( ssn NUMBER,
  empid NUMBER,
  mgr VARCHAR2(30));

CREATE OR REPLACE VIEW Person_v OF person_typ
WITH OBJECT OID(ssn) AS
  SELECT ssn, name, address
  FROM all_personattrs
  WHERE typeid = 1;

CREATE OR REPLACE VIEW Student_v OF student_typ UNDER Person_v
  AS
  SELECT x.ssn, x.name, x.address, y.deptid, y.major
  FROM all_personattrs x, all_studentattrs y
  WHERE x.typeid = 2 AND x.ssn = y.ssn;

CREATE OR REPLACE VIEW Employee_v OF employee_typ UNDER Person_v
  AS
  SELECT x.ssn, x.name, x.address, y.empid, y.mgr
  FROM all_personattrs x, all_employeeattrs y
  WHERE x.typeid = 3 AND x.ssn = y.ssn;
```

The vertical model can efficiently process queries of the kind `SELECT * FROM` *`root_view`*, and it is possible to index individual attributes and impose unique constraints on them. However, to re-create an instance of a type, a join over OIDs must be performed for each level that the type is removed from the root in the hierarchy.

Querying a View in a Hierarchy

You can query any view or subview in an object view hierarchy; rows are returned for the declared type of the view that you query and for any of that type's subtypes. So, for instance, in an object view hierarchy based on the `person_typ` type hierarchy, you can query the view of `person_typ` to get a result set that contains all persons, including students and employees; or you can query the view of `student_typ` to get a result set that contains only students, including part-time students.

In the `SELECT` list of a query, you can include either functions such as `REF()` and `VALUE()` that return an object instance, or you can specify object attributes of the view's declared type, such as the `name` and `ssn` attributes of `person_typ`.

If you specify functions, to return object instances, the query returns a polymorphic result set: that is, it returns instances of both the view's declared type and any subtypes of that type.

For example, the following query returns instances of persons, employees, and students of all types, as well as `REF`s to those instances.

```
SELECT REF(p), VALUE(p) FROM Person_v p;
```

If you specify individual attributes of the view's declared type in the `SELECT` list or do a `SELECT *`, again the query returns rows for the view's declared type and any subtypes of that type, but these rows are projected over columns for the attributes of the view's declared type, and only those columns are used. In other words, the subtypes are represented only with respect to the attributes they inherit from and share with the view's declared type.

So, for example, the following query returns rows for all persons and rows for employees and students of all types, but the result uses only the columns for the attributes of `person_typ`—namely, `name`, `ssn`, and `address`. It does not show rows for attributes added in the subtypes, such as the `deptid` attribute of `student_typ`.

```
SELECT * FROM Person_v;
```

To exclude subviews from the result, use the `ONLY` keyword. The `ONLY` keyword confines the selection to the declared type of the view that you are querying:

```
SELECT VALUE(p) FROM ONLY(Person_v) p;
```

Privileges for Operations on View Hierarchies

Generally, a query on a view with subviews requires only the `SELECT` privilege on the view being referenced and does not require any explicit privileges on subviews. For example, the following query requires only `SELECT` privileges on `Person_v` but not on any of its subviews.

```
SELECT * FROM Person_v;
```

However, a query that selects for any attributes added in subtypes but not used by the root type requires the SELECT privilege on all subviews as well. Such subtype attributes may hold sensitive information that should reasonably require additional privileges to access.

The following query, for example, requires SELECT privileges on Person_v and also on Student_v, Employee_v (and on any other subview of Person_v) because the query selects object instances and thus gets all the attributes of the subtypes.

```
SELECT VALUE(p) FROM Person_v p;
```

To simplify the process of granting SELECT privileges on an entire view hierarchy, you can use the HIERARCHY option. Specifying the HIERARCHY option when granting a user SELECT privileges on a view implicitly grants SELECT privileges on all current and future subviews of the view as well. For example:

```
GRANT SELECT ON Person_v TO oe WITH HIERARCHY OPTION;
```

A query that excludes rows belonging to subviews also requires SELECT privileges on all subviews. The reason is that information about which rows belong exclusively to the most specific type of an instance may be sensitive, so the system requires SELECT privileges on subviews for queries (such as the following one) that exclude all rows from subviews.

```
SELECT * FROM ONLY(Person_v);
```

7

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:

- Privileges on Object Types and Their Methods
- Dependencies and Incomplete Types
- Synonyms for Object Types
- Performance Tuning
- Tools Providing Support for Objects
- Utilities Providing Support for Objects

Privileges on Object Types and Their Methods

Privileges for object types exist at the system level and the schema object level.

System Privileges for Object Types

Oracle defines the following system privileges for object types:

- `CREATE TYPE` enables you to create object types in your own schema
- `CREATE ANY TYPE` enables you to create object types in any schema
- `ALTER ANY TYPE` enables you to alter object types in any schema
- `DROP ANY TYPE` enables you to drop named types in any schema
- `EXECUTE ANY TYPE` enables you to use and reference named types in any schema
- `UNDER ANY TYPE` enables you to create subtypes under any non-final object types
- `UNDER ANY VIEW` enables you to create subviews under any object view

The `RESOURCE` role includes the `CREATE TYPE` system privilege. The DBA role includes all of these privileges.

Schema Object Privileges

Two schema object privileges apply to object types:

- `EXECUTE` on an object type enables 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.

- UNDER enables you to create a subtype or subview under the type or view on which the privilege is granted

  The UNDER privilege on a subtype or subview can be granted only if the grantor has the UNDER privilege on the direct supertype or superview WITH GRANT OPTION.

The phrase WITH HIERARCHY OPTION grants a specified object privilege on all subobjects of the object. This option is meaningful only with the SELECT object privilege granted on an object view in an object view hierarchy. In this case, the privilege applies to all subviews of the view on which the privilege is granted.

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: Privileges on Object Types

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

USER1 performs the following DDL in the USER1 schema:

```
CONNECT user1/user1
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:

```
CONNECT user2/user2
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 -- incorrect statement;
```

USER3 can successfully perform the following actions:

```
CONNECT user3/user3
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 and queries in Example 7–1:

Example 7–1 SELECT Privileges on Type Access

```
CREATE TYPE emp_type as object (
  eno     NUMBER,
  ename   VARCHAR2(36));
/
CREATE TABLE emp OF emp_type;
GRANT SELECT on emp TO user1;
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 described in "Example: Privileges on Object Types" on page 7-2, USER3 can perform the following queries:

```
SELECT t.col1.attr2 from user2.tab2 t;

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.

  See Also: *Oracle Call Interface Programmer's Guide* for tips and techniques for using OCI program effectively with objects

- 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 through intermediate types, are called mutually dependent. In a diagram that uses arrows to show the dependency relationships among a set of types, connections among mutually dependent types form a loop.

To define such a circular dependency, you must use REFs for at least one segment of the circle.

For example, you can define the types show in Example 7–2.

Example 7–2 Creating Dependent Object 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.

Notice that the code in Example 7–2 creates the type department twice. The first statement:

```
CREATE TYPE department;
```

is an optional, incomplete declaration of department that serves as a placeholder for the REF attribute of employee to point to. The declaration is incomplete in that it omits the AS OBJECT phrase and lists no attributes or methods. These are specified later in the full declaration that completes the type. In the meantime, department is created as an incomplete object type. This enables the compilation of employee to proceed without errors.

To complete an incomplete type, you execute a CREATE TYPE statement that specifies the attributes and methods of the type, as shown at the end of the example. Complete an incomplete type after all the types that it refers to are created.

If you do not create incomplete types as placeholders, types that refer to the missing types still compile, but the compilation proceeds with errors.

For example, if department did not exist at all, Oracle would create it as an incomplete type and compile employee with errors. Then employee would be recompiled the next time that some operation attempts to access it. This time, if all the types it depends on are created and its dependencies are satisfied, it will compile without errors.

Incomplete types also enable you to create types that contain REF attributes to a subtype that has not yet been created. To create such a supertype, first create an incomplete type of the subtype to be referenced. Create the complete subtype after you create the supertype.

A subtype is just a specialized version of its direct supertype and consequently has an explicit dependency on it. To ensure that subtypes are not left behind after a supertype is dropped, all subtypes must be dropped first: a supertype cannot be dropped until all its subtypes are dropped.

Completing Incomplete Types

When all the types that an incomplete type refers to have been created, there is no longer any need for the incomplete type to remain incomplete, and you should complete the declaration of the type. Completing the type recompiles it and enables the system to release various locks.

You must complete an incomplete object type as an object type: you cannot complete an object type as a collection type (a nested table type or an array type). The only alternative to completing a type declaration is to drop the type.

You must also complete any incomplete types that Oracle creates for you because you did not explicitly create them yourself. The example in the preceding section explicitly creates department as an incomplete type. If department were not explicitly created as an incomplete type, Oracle would create it as one so that the employee type can compile (with errors). You must complete the declaration of department as an object type whether you or Oracle declared it as an incomplete type.

Manually Recompiling a Type

If a type was created with compilation errors, and you attempt an operation on it, such as creating tables or inserting rows, you may receive an error. You need to recompile type *typename* before attempting the operation. To manually recompile a type, execute an ALTER TYPE *typename* COMPILE statement. After you have successfully compiled the type, attempt the operation again.

Type Dependencies of Substitutable Tables and Columns

A substitutable table or column of type T is dependent not only on T but on all subtypes of T as well. This is because a hidden column is added to the table for each attribute added in a subtype of T. The hidden columns are added even if the substitutable table or column contains no data of that subtype.

So, for example, a persons table of type person_typ is dependent not only on person_typ but also on the person_typ subtypes student_typ and part_time_student_typ.

If you attempt to drop a subtype that has a dependent type, table, or column, the DROP TYPE statement returns an error and aborts. For example, trying to drop part_time_student_typ will raise an error because of the dependent persons table.

If dependent tables or columns exist but contain no data of the type that you want to drop, you can use the VALIDATE keyword to drop the type. The VALIDATE keyword causes Oracle to check for actual stored instances of the specified type and to drop the type if none are found. Hidden columns associated with attributes unique to the type are removed as well.

For example, the first DROP TYPE statement in the following example fails because part_time_student_typ has a dependent table (persons). But if persons contains no instances of part_time_student_typ (and no other dependent table or column does, either), the VALIDATE keyword causes the second DROP TYPE statement to succeed:

```
-- Following generates an error due to presence of Persons table
DROP TYPE part_time_student_typ -- incorrect statement;
-- Following succeeds if there are no stored instances of part_time_student_typ
DROP TYPE part_time_student_typ VALIDATE;
```

Note: Oracle recommends that you always use the VALIDATE option while dropping subtypes.

The FORCE Option

The DROP TYPE statement also has a FORCE option that causes the type to be dropped even though it may have dependent types or tables. The FORCE option should be used only with great care, as any dependent types or tables that do exist are marked invalid and become inaccessible when the type is dropped. Data in a table that is marked invalid because a type it depends on has been dropped can never be accessed again. The only action that can be performed on such a table is to drop it.

See "Type Evolution" on page 8-5 for information about how to alter a type.

Synonyms for Object Types

Just as you can create synonyms for tables, views, and various other schema objects, you can also define synonyms for object types.

Synonyms for types have the same advantages as synonyms for other kinds of schema objects: they provide a location-independent way to reference the underlying schema object. An application that uses public type synonyms can be deployed without alteration in any schema of a database without having to qualify a type name with the name of the schema in which the type was defined.

> **See Also:** *Oracle Database Administrator's Guide* for more information on synonyms in general

Creating a Type Synonym

You create a type synonym with a `CREATE SYNONYM` statement. For example, these statements create a type `typ1` and then create a synonym for it:

```
-- For the synonym examples, CREATE SYNONYM and CREATE PUBLIC SYNONYM
-- are granted to USER1
CREATE TYPE typ1 AS OBJECT (x number);
/
CREATE SYNONYM syn1 FOR typ1;
```

Synonyms can be created for collection types, too. The following example creates a synonym for a nested table type:

```
CREATE TYPE typ2 AS TABLE OF NUMBER;
/
CREATE SYNONYM syn2 FOR typ2;
```

You create a public synonym by using the `PUBLIC` keyword:

```
CREATE TYPE shape AS OBJECT ( name VARCHAR2(10) );
/
CREATE PUBLIC SYNONYM pub_shape FOR shape;
```

The `REPLACE` option enables you to have the synonym point to a different underlying type. For example, the following statement causes `syn1` to point to type `typ2` instead of the type it formerly pointed to:

```
CREATE OR REPLACE SYNONYM syn1 FOR typ2;
```

Using a Type Synonym

You can use a type synonym anywhere that you can refer to a type. For instance, you can use a type synonym in a DDL statement to name the type of a table column or type attribute. In Example 7–3, synonym `syn1` is used to specify the type of an attribute in type `typ3`:

Example 7–3 Using a Type Synonym in a Create Statement

```
CREATE TYPE typ1 AS OBJECT (x number);
/
CREATE SYNONYM syn1 FOR typ1;

CREATE TYPE typ3 AS OBJECT ( a syn1 );
/
```

The next example shows a type synonym syn1 used to call the constructor of the object type typ1, for which syn1 is a synonym. The statement returns an object instance of typ1:

```
SELECT syn1(0) FROM dual;
```

In the following example, syn2 is a synonym for a nested table type. The example shows the synonym used in place of the actual type name in a CAST expression:

```
SELECT CAST(MULTISET(SELECT eno FROM USER3.EMP) AS syn2) FROM dual;
```

Type synonyms can be used in the following kinds of statements:

- DML statements: SELECT, INSERT, UPDATE, DELETE, FLASHBACK TABLE, EXPLAIN PLAN, and LOCK TABLE
- DDL statements: AUDIT, NOAUDIT, GRANT, REVOKE, and COMMENT

Describing Schema Objects That Use Synonyms

If a type or table has been created using type synonyms, the DESCRIBE command will show the synonyms in place of the types they represent. Similarly, catalog views, such as USER_TYPE_ATTRS, that show type names will show the associated type synonym names in their place.

You can query the catalog view USER_SYNONYMS to find out the underlying type of a type synonym.

Dependents of Type Synonyms

A type that directly or indirectly references a synonym in its type declaration is a dependent of that synonym. Thus, in the following example, type typ3 is a dependent type of synonym syn1.

```
CREATE TYPE typ3 AS OBJECT ( a syn1 );
/
```

Other kinds of schema objects that reference synonyms in their DDL statements also become dependents of those synonyms. An object that depends on a type synonym depends on both the synonym and on the synonym's underlying type.

A synonym's dependency relationships affect your ability to drop or rename the synonym. Dependent schema objects are also affected by some operations on synonyms. The following sections describe these various ramifications.

Restriction on Replacing a Type Synonym

You can replace a synonym only if it has no dependent tables or valid user defined types. Replacing a synonym is equivalent to dropping it and then re-creating a new synonym with the same name.

Dropping Type Synonyms

You drop a synonym with the DROP SYNONYM statement as shown in Example 7–4.

Example 7–4 Dropping Type Synonyms

```
CREATE SYNONYM syn4 FOR typ1;

DROP SYNONYM syn4;
```

You cannot drop a type synonym if it has table or valid object types as dependents unless you use the FORCE option. The FORCE option causes any columns that directly or indirectly depend on the synonym to be marked unused, just as if the actual types of the columns were dropped. (A column indirectly depends on a synonym if, for instance, the synonym is used to specify the type of an attribute of the declared type of the column.)

Any dependent schema objects of a dropped synonym are invalidated. They can be revalidated by creating a local object of the same name as the dropped synonym or by creating a new public synonym with same name.

Dropping the underlying base type of a type synonym has the same effect on dependent objects as dropping the synonym.

Renaming Type Synonyms

You can rename a type synonym with the RENAME statement. Renaming a synonym is equivalent to dropping it and then re-creating it with a new name. You cannot rename a type synonym if it has dependent tables or valid object types. The following example fails because synonym syn1 has a dependent object type:

```
RENAME syn1 TO syn3 -- invalid statement;
```

Public Type Synonyms and Local Schema Objects

You cannot create a local schema object that has the same name as a public synonym if the public synonym has a dependent table or valid object type in the local schema where you want to create the new schema object. Nor can you create a local schema object that has the same name as a private synonym in the same schema.

For instance, in the following example, table shape_tab is a dependent table of public synonym pub_shape because the table has a column that uses the synonym in its type definition. Consequently, the attempt to create a table that has the same name as public synonym pub_shape, in the same schema as the dependent table, fails:

```
-- Following uses public synonym pub_shape
CREATE TABLE shape_tab ( c1 pub_shape );
-- Following is not allowed
CREATE TABLE pub_shape ( c1 NUMBER ) -- invalid statement;
```

Performance Tuning

When tuning objects, the following items need to be addressed:

- How objects and object views consume CPU and memory resources during runtime
- How to monitor memory and CPU resources during runtime
- How to manage large numbers of objects

Some of the key performance factors are the following:

- DBMS_STATS package to collect statistics
- tkprof to profile execution of SQL commands
- EXPLAIN PLAN to generate the query plans

> **See Also:** *Oracle Database Performance Tuning Guide* for details on measuring and tuning the performance of your application

Tools Providing Support for Objects

This section describes several Oracle tools that provide support for Oracle objects.

JDeveloper

JDeveloper is a full-featured, integrated development environment for creating multitier Java applications. It enables you to develop, debug, and deploy Java client applications, dynamic HTML applications, web and application server components and database stored procedures based on industry-standard models.

JDeveloper provides features in the following areas:

- Oracle Business Components for Java
- Web Application Development
- Java Client Application Development
- Java in the Database
- Component-Based Development with JavaBeans
- Simplified Database Access
- Visual Integrated Development Environment
- Java Language Support

JDeveloper is a cross-platform IDE. It provides a standard GUI based Java development environment that is well integrated with Oracle Application Server and Database.

Business Components for Java (BC4J)

Supporting standard EJB and CORBA deployment architectures, Oracle Business Components for Java simplifies the development, delivery, and customization of Java business applications for the enterprise. Oracle Business Components for Java is an application component framework providing developers a set of reusable software building blocks that manage all the common facilities required to:

- Author and test business logic in components which integrate with relational databases
- Reuse business logic through multiple SQL-based views of data
- Access and update the views from servlets, JavaServer Pages (JSPs), and thin-Java Swing clients
- Customize application functionality in layers without requiring modification of the delivered application

JPublisher

JPublisher is a utility that generates Java classes to represent the following user-defined database entities in your Java program:

- Database object types
- Database reference (`REF`) types
- Database collection types (varrays or nested tables)
- PL/SQL packages

JPublisher enables you to specify and customize the mapping of database object types, reference types, and collection types (varrays or nested tables) to Java classes, in a strongly typed paradigm.

> **See Also:** *Oracle Database JPublisher User's Guide*

Utilities Providing Support for Objects

This section describes several Oracle utilities that provide support for Oracle objects.

Import/Export 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.

Types

The definition statements for derived types are exported. On an Import, a subtype may be created before the supertype definition has been imported. In this case, the subtype will be created with compilation errors, which may be ignored. The type will be revalidated after its supertype is created.

Object View Hierarchies

View definitions for all views belonging to a view hierarchy are exported

SQL*Loader

The SQL*Loader utility moves data from external files into tables in an Oracle database. The files may contain data consisting of basic scalar data types, such as `INTEGER`, `CHAR`, or `DATE`, as well as complex user-defined data types such as row and column objects (including objects that have object, collection, or `REF` attributes), collections, and LOBs. Currently, SQL*Loader supports single-level collections only: you cannot yet use SQL*Loader to load multilevel collections, that is, collections whose elements are, or contain, other collections.

SQL*Loader uses control files, which contain SQL*Loader data definition language (DDL) statements, to describe the format, content, and location of the datafiles.

SQL*Loader provides two approaches to loading data:

- **Conventional path loading**, which uses the `SQL INSERT` statement and a bind array buffer to load data into database tables
- **Direct path loading**, which uses the Direct Path Load API to write data blocks directly to the database on behalf of the SQL*Loader client.

  Direct path loading does not use a SQL interface and thus avoids the overhead of processing the associated SQL statements. Consequently, direct path loading tends to provide much better performance than conventional path loading.

Either approach can be used to load data of supported object and collection data types.

> **See Also:** *Oracle Database Utilities* for instructions on how to use SQL*Loader

8

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.

The chapter contains these topics:

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. Oracle stores leaf-level attributes of nested 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 VARRAYs" on page 8-3 and "Internal Layout of Nested Tables" on page 8-3.

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 object-type column also has a corresponding hidden column to store the NULL information for the column objects (that is, the atomic nulls of the top-level and the nested objects).

Hidden Columns for Substitutable Columns and Tables

A substitutable column or object table has a hidden column not only for each attribute of the column's object type but also for each attribute added in any subtype of the object type. These columns store the values of those attributes for any subtype instances inserted in the substitutable column.

For example, a substitutable column of person_typ will have associated with it a hidden column for each of the attributes of person_typ: idno, name, and phone. It will also have hidden columns for attributes of the subtypes of person_typ. For example, the attributes dept_id and major (for student_typ) and number_hours (for part_time_student_typ).

When a subtype is created, hidden columns for attributes added in the subtype are automatically added to tables containing a substitutable column of any of the new subtype's ancestor types. These retrofit the tables to store data of the new type. If, for some reason, the columns cannot be added, creation of the subtype is rolled back.

When a subtype is dropped with the VALIDATE option to DROP TYPE, all such hidden columns for attributes unique to the subtype are automatically dropped as well if they do not contain data.

A substitutable column also has associated with it a hidden type discriminant column. This column contains an identifier, called a typeid, that identifies the most specific type of each object in the substitutable column. Typically, a typeid (RAW) is one byte, though it can be as big as four bytes for a large hierarchy.

You can find the typeid of a specified object instance using the function SYS_TYPEID. For example, suppose that the substitutable object table person_obj_table contains three rows, as shown in Example 2–19 on page 2-18.

The query in Example 8–1 retrieves typeids of object instances stored in the table:

Example 8–1 Retrieving Typeids in a Table

```
SELECT name, SYS_TYPEID(VALUE(p)) typeid
  FROM person_obj_table p;

NAME                           TYPEID
------------------------------ ---------------------------
Bob Jones                      01
Joe Lane                       02
Kim Patel                      03
```

The catalog views USER_TYPES, DBA_TYPES and ALL_TYPES contain a TYPEID column (not hidden) that gives the typeid value for each type. You can join on this column to get the type names corresponding to the typeids in a type discriminant column. See "SYS_TYPEID" on page 2-34 for more information about SYS_TYPEID and typeids.

REFs

When Oracle constructs a REF to a row object, the constructed REF is made up of the object identifier (OID), 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 a 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.

See "Nested Table Storage" on page 9-10.

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 9-9 for details.

Creating Indexes on Typeids or Attributes

This section discusses the use of indexes on typeids and attributes.

Indexing a Type Discriminant Column

Using the SYS_TYPEID function, you can build an index on the hidden type discriminant column that every substitutable column has. The type discriminant column contains typeids that identify the most specific type of every object instance stored in the substitutable column. This information is used by the system to evaluate queries that use the IS OF predicate to filter by type, but you can access the typeids for your own purposes using the SYS_TYPEID function.

Generally, a type discriminant column contains only a small number of distinct typeids: at most, there can be only as many as there are types in the related type hierarchy. The low cardinality of this column makes it a good candidate for a bitmap index.

For example, the following statement creates a bitmap index on the type discriminant column underlying the substitutable contact column of table contacts. Function SYS_TYPEID is used to reference the type discriminant column:

```
CREATE BITMAP INDEX typeid_idx ON contacts (SYS_TYPEID(contact));
```

Indexing Subtype Attributes of a Substitutable Column

You can build an index on attributes of any of the types that can be stored in a substitutable column. Attributes of subtypes can be referenced in the CREATE INDEX statement by using the TREAT function to filter out types other than the desired subtype (and its subtypes); you then use the dot notation to specify the desired attribute.

For example, the following statement creates an index on the major attribute of all students in the contacts table. The declared type of the contact column is person_typ, of which student_typ is a subtype, so the column may contain instances of person_typ, student_typ, and subtypes of either one:

```
CREATE INDEX major1_idx ON contacts
  (TREAT(contact AS student_typ).major);
```

student_typ is the type that first defined the major attribute: the person_typ supertype does not have it. Consequently, all the values in the hidden column for the major attribute are values for student_typ or parttimestudent_typ authors (a student_typ subtype). This means that the hidden column's values are identical to the values returned by the TREAT expression, which returns major values for all students, including student subtypes: both the hidden column and the TREAT expression list majors for students and nulls for authors of other types. The system exploits this fact and creates index major1_idx as an ordinary B-tree index on the hidden column.

Values in a hidden column are identical to the values returned by a TREAT expression like the preceding one only if the type named as the target of the TREAT function (student_typ) is the type that first defined the attribute. If the target of the TREAT function is a subtype that merely inherited the attribute, as in the following example, the TREAT expression will return non-null major values for the subtype (part-time students) but not for its supertype (other students).

```
CREATE INDEX major2_idx ON contacts
  (TREAT(contact AS part_time_student_typ).major);
```

Here the values stored in the hidden column for major may be different from the results of the TREAT expression. Consequently, an ordinary B-tree index cannot be created on the underlying column. In a case like this, Oracle treats the TREAT expression like any other function-based expression and tries to create the index as a function-based index on the result.

The following example, like the previous one, creates a function-based index on the major attribute of part-time students, but in this case the hidden column for major is associated with a substitutable object table person_obj_table:

```
CREATE INDEX major3_idx ON person_obj_table p
  (TREAT(VALUE(p) AS part_time_student_typ).major);
```

Type Evolution

Changing a object type is called type evolution. You can make the following changes to an object type:

- Add and drop attributes
- Add and drop methods
- Modify a numeric attribute to increase its length, precision, or scale
- Modify a varying length character attribute to increase its length
- Change a type's FINAL and INSTANTIABLE properties
- Modify limit and size of VARRAYs
- Modify length, precision, and scale of collection elements

Changes to a type affect things that reference the type. For example, if you add a new attribute to a type, data in a column of that type must be presented so as to include the new attribute.

Schema objects that directly or indirectly reference a type and are affected by a change to it are called dependents of the type. A type can have these kinds of dependents:

- Table
- Type or subtype
- Program unit (PL/SQL block): procedure, function, package, trigger
- Indextype
- View (including object view)
- Function-based index
- Operator

How a dependent schema object is affected by a change to a type depends on the dependent object and on the nature of the change to the type.

All dependent program units, views, operators and indextypes are marked invalid when a type is modified. The next time one of these invalid schema objects is referenced, it is revalidated using the new type definition. If the object recompiles successfully, it becomes valid and can be used again. Depending on the change to the type, function-based indexes may be dropped or disabled and need to be rebuilt.

If a type has dependent tables, then, for each attribute added to a type, one or more internal columns are added to the table depending on the new attribute's type. New attributes are added with NULL values. For each dropped attribute, the columns associated with that attribute are dropped. For each modified attribute, the length, precision, or scale of its associated column is changed accordingly.

These changes mainly involve updating the tables' metadata (information about a table's structure, describing its columns and their types) and can be done quickly. However, the data in those tables must be updated to the format of the new type version as well. Updating this data can be time-consuming if there is a lot of it, so the ALTER TYPE command has options to let you choose whether to convert all dependent table data immediately or to leave it in the old format to be converted piecemeal as it is updated in the course of business.

The CASCADE option for ALTER TYPE propagates a type change to dependent types and tables. See "ALTER TYPE Statement for Type Evolution" on page 8-12. CASCADE itself has options that let you choose whether to convert table data to the new type format as part of the propagation: the option INCLUDING TABLE DATA converts the data; the option NOT INCLUDING TABLE DATA does not convert it. By default, the CASCADE option converts the data. In any case, table data is always returned in the format of the latest type version. If the table data is stored in the format of an earlier type version, Oracle converts the data to the format of the latest version before returning it, even though the format in which the data is actually stored is not changed until the data is rewritten.

You can retrieve the definition of the latest type from the system view USER_SOURCE. You can view definitions of all versions of a type in the USER_TYPE_VERSIONS view.

See Also:

Oracle Database PL/SQL User's Guide and Reference for details about type specification and body compilation

Changes Involved When a Type Is Altered

Only structural changes to a type affect dependent data and require the data to be converted. Changes that are confined to a type's method definitions or behavior (in the type body, where the type's methods are implemented) do not.

These possible changes to a type are structural:

- Adding an attribute
- Dropping an attribute
- Modifying the length, precision, or scale of an attribute
- Changing the finality of a type (which determines whether subtypes can be derived from it) from FINAL to NOT FINAL or from NOT FINAL to FINAL.

These changes result in new versions of the altered type and all its dependent types and require the system to add, drop, or modify internal columns of dependent tables as part of the process of converting to the new version.

When you make any of these kinds of changes to a type that has dependent types or tables, the effects of propagating the change are not confined only to metadata but affect data storage arrangements and require the data to be converted.

Besides converting data, you may also need to make other changes. For example, if a new attribute is added to a type, and the type body invokes the type's constructor, then each constructor in the type body must be modified to specify a value for the new attribute. Similarly, if a new method is added, then the type body must be replaced to add the implementation of the new method. The type body can be modified by using the CREATE OR REPLACE TYPE BODY statement.

Example 8–2 illustrates how to make a simple change to person_typ by adding one attribute and dropping another. The CASCADE keyword propagates the type change to dependent types and tables, but the phrase NOT INCLUDING TABLE DATA prevents conversion of the related data.

Example 8–2 Altering an Object Type by Adding and Dropping an Attribute

```
CREATE TABLE person_obj_table OF person_typ;

INSERT INTO person_obj_table
  VALUES (person_typ(12, 'Bob Jones', '111-555-1212'));

SELECT value(p) FROM person_obj_table p;

VALUE(P)(IDNO, NAME, PHONE)
--------------------------------------------
PERSON_TYP(12, 'Bob Jones', '111-555-1212')

-- add the email attribute and drop the phone attribute
ALTER TYPE person_typ
  ADD ATTRIBUTE (email VARCHAR2(80)),
  DROP ATTRIBUTE phone CASCADE NOT INCLUDING TABLE DATA;

-- Disconnect and reconnect to accommodate the type change
-- The data of table person_obj_table has not been converted yet, but
-- when the data is retrieved, Oracle returns the data based on
-- the latest type version. The new attribute is initialized to NULL.
SELECT value(p) FROM person_obj_table p;

VALUE(P)(IDNO, NAME, EMAIL)
---------------------------------
PERSON_TYP(12, 'Bob Jones', NULL)
```

During SELECT statements, even though column data may be converted to the latest type version, the converted data is not written back to the column. If a certain user-defined type column in a table is retrieved often, you should consider converting that data to the latest type version to eliminate redundant data conversions. Converting is especially beneficial if the column contains a VARRAY attribute because a VARRAY typically takes more time to convert than an object or nested table column.

You can convert a column of data by issuing an UPDATE statement to set the column to itself. For example:

```
UPDATE dept_tab SET emp_array_col = emp_array_col;
```

You can convert all columns in a table by using ALTER TABLE with the UPGRADE INCLUDING DATA. For example:

```
ALTER TYPE person_typ ADD ATTRIBUTE (photo BLOB)
  CASCADE NOT INCLUDING TABLE DATA;
```

Altering a Type by Adding a Nested Table Attribute

This section describes the steps required to make a complex change to a type. This change involves the addition of a nested table attribute to an object type that is already included in a nested table. When upgrading the affected nested table, the name of the new storage table is specified.

Assume we have the following schema based on the person_typ object type:

```
CREATE TYPE people_typ AS TABLE OF person_typ;
/
CREATE TYPE department_typ AS OBJECT (
  manager    person_typ,
  employee   people_typ);
/
CREATE TABLE department OF department_typ
  NESTED TABLE employee STORE AS employee_store_nt;
```

1. Issue an ALTER TYPE statement to alter the type.

   The default behavior of an ALTER TYPE statement without any option specified is to check if there is any object dependent on the target type. The statement aborts if any dependent object exists. Optional keywords allow cascading the type change to dependent types and tables.

   With the ALTER TYPE statement in Example 8–3, all type and table checks are bypassed to save time and dependent objects are invalidated. Table data cannot be accessed until is validated.

Example 8–3 Altering an Object Type by Adding a Nested Table Attribute

```
-- Create and add a new nested table attribute to person_typ
CREATE TYPE tasks_typ AS OBJECT (
  priority       VARCHAR2(2),
  description    VARCHAR2(30));
/

CREATE TYPE tasks_nttab AS TABLE OF tasks_typ;
/

ALTER TYPE person_typ ADD ATTRIBUTE tasks tasks_nttab
  INVALIDATE;

-- Propagate the change to employee_store_nt
-- Specify a storage name for the new nested table
ALTER TABLE employee_store_nt
  UPGRADE NESTED TABLE tasks STORE AS tasks_nt;
```

2. Use CREATE OR REPLACE TYPE BODY to update the corresponding type body to make it current with the new type definition if necessary.
3. Upgrade the dependent tables to the latest type version and convert the data in the tables.

Example 8–4 Upgrading Dependent Tables

```
ALTER TABLE department UPGRADE INCLUDING DATA;
```

4. Alter dependent PL/SQL program units as needed to take account of changes to the type.
5. Use OTT or JPUB (or another tool) to generate new header files for applications, depending on whether the application is written in C or Java.

   Adding a new attribute to a supertype also increases the number of attributes in all its subtypes because these inherit the new attribute. Inherited attributes always precede declared (locally defined) attributes, so adding a new attribute to a supertype causes the ordinal position of all declared attributes of any subtype to be incremented by one recursively. The mappings of the altered type must be updated to include the new attributes. OTT and JPUB do this. If you use some other tool, you must be sure that the type headers are properly synchronized with the type definition in the server; otherwise, unpredictable behavior may result.

6. Modify application code as needed and rebuild the application.

Validating a Type That Has Been Altered

When the system executes an ALTER TYPE statement, it first validates the requested type change syntactically and semantically to make sure it is legal. The system performs the same validations as for a CREATE TYPE statement plus some additional ones. For example, it checks to be sure an attribute being dropped is not used as a partitioning key. If the new spec of the target type or any of its dependent types fails the type validations, the ALTER TYPE statement aborts. No new type version is created, and all dependent objects remain unchanged.

If dependent tables exist, further checking is done to ensure that restrictions relating to the tables and any indexes are observed. Again, if the ALTER TYPE statement fails the check of table-related restrictions, then the type change is aborted, and no new version of the type is created.

When multiple attributes are added in a single ALTER TYPE statement, they are added in the order specified. Multiple type changes can be specified in the same ALTER TYPE statement, but no attribute name or method signature can be specified more than once in the statement. For example, adding and modifying the same attribute in a single statement is not allowed.

For example:

```
CREATE TYPE mytype AS OBJECT (attr1 NUMBER, attr2 NUMBER);
/
ALTER TYPE mytype ADD ATTRIBUTE (attr3 NUMBER),
  DROP ATTRIBUTE attr2,
  ADD ATTRIBUTE attr4 NUMBER CASCADE;
```

The resulting definition for mytype becomes:

```
(attr1 NUMBER, attr3 NUMBER, attr4 NUMBER);
```

The following ALTER TYPE statement, which attempts to make multiple changes to the same attribute (attr5), is invalid:

```
-- invalid ALTER TYPE statement
ALTER TYPE mytype ADD ATTRIBUTE (attr5 NUMBER, attr6 CHAR(10)),
  DROP ATTRIBUTE attr5;
```

The following are other notes on validation constraints, table restrictions, and assorted information about the various kinds of changes that can be made to a type.

Dropping an Attribute

- Dropping all attributes from a root type is not allowed. You must instead drop the type. Because a subtype inherits all the attributes from its supertype, dropping all the attributes from a subtype does not reduce its attribute count to zero; thus, dropping all attributes declared locally in a subtype is allowed.
- Only an attribute declared locally in the target type can be dropped. You cannot drop an inherited attribute from a subtype. Instead, drop the attribute from the type where it is locally declared.
- Dropping an attribute which is part of a table partitioning or sub-partitioning key in a table is not allowed.

- Dropping an attribute of a primary key OID of an object table or an index-organized table (IOT) is not allowed.
- When an attribute is dropped, the column corresponding to the dropped attribute is dropped.
- Indexes, statistics, constraints, and any referential integrity constraints referencing a dropped attribute are removed.

Modifying Attribute Type to Increase the Length, Precision, or Scale

- Expanding the length of an attribute referenced in a function-based index, clustered key or domain index on a dependent table is not allowed.

Dropping a Method

- You can drop a method only from the type in which the method is defined (or redefined): You cannot drop an inherited method from a subtype, and you cannot drop an redefined method from a supertype.
- If a method is not redefined, dropping it using the `CASCADE` option removes the method from the target type and all subtypes. However, if a method is redefined in a subtype, the `CASCADE` will fail and roll back. For the `CASCADE` to succeed, you must first drop each redefined method from the subtype that defines it and only then drop the method from the supertype.

  You can consult the `USER_DEPENDENCIES` table to find all the schema objects, including types, that depend on a given type. You can also run the `DBMS_UTILITY.GET_DEPENDENCY` utility to find the dependencies of a type.
- You can use the `INVALIDATE` option to drop a method that has been redefined, but the redefined versions in the subtypes must still be dropped manually. The subtypes will remain in an invalid state until they are explicitly altered to drop the redefined versions. Until then, an attempt to recompile the subtypes for revalidation will produce the error `Method does not override`.

  Unlike `CASCADE`, `INVALIDATE` bypasses all the type and table checks and simply invalidates all schema objects dependent on the type. The objects are revalidated the next time they are accessed. This option is faster than using `CASCADE`, but you must be certain that no problems will be encountered revalidating dependent types and tables. Table data cannot be accessed while a table is invalid; if a table cannot be validated, its data remains inaccessible.

  See "If a Type Change Validation Fails" on page 8-11.

Modifying the FINAL or INSTANTIABLE Property

- Altering an object type from `INSTANTIABLE` to `NOT INSTANTIABLE` is allowed only if the type has no table dependents.
- Altering an object type from `NOT INSTANTIABLE` to `INSTANTIABLE` is allowed anytime. This change does not affect tables.
- Altering an object type from `NOT FINAL` to `FINAL` is allowed only if the target type has no subtypes.
- When you alter an object type from `FINAL` to `NOT FINAL` or vice versa, you must use `CASCADE` to convert data in dependent columns and tables immediately. You may not use the `CASCADE` option `NOT INCLUDING TABLE DATA` to defer converting data.

  If you alter a type from `NOT FINAL` to `FINAL`, you must use `CASCADE INCLUDING TABLE DATA`. If you alter a type from `FINAL` to `NOT FINAL`, you may use either CASCADE INCLUDING TABLE DATA or CASCADE CONVERT TO SUBSTITUTABLE.

When you alter a type from FINAL to NOT FINAL. the CASCADE option you should choose depends on whether you want to be able to insert new subtypes of the type you are altering in existing columns and tables.

By default, altering a type from FINAL to NOT FINAL enables you to create new substitutable tables and columns of that type, but it does not automatically make existing columns (or object tables) of that type substitutable. In fact, just the opposite happens: existing columns and tables of the type are marked NOT SUBSTITUTABLE AT ALL LEVELS. If any embedded attribute of such a column is substitutable, an error is generated. New subtypes of the altered type cannot be inserted in such preexisting columns and tables.

To alter an object type to NOT FINAL in such a way as to make existing columns and tables of the type substitutable (assuming that they are not marked NOT SUBSTITUTABLE), use the CASCADE option CONVERT TO SUBSTITUTABLE. For example:

Example 8–5 Converting a Type from FINAL to NOT FINAL

```
CREATE TYPE shape AS OBJECT (
    name VARCHAR2(30),
    area NUMBER)
    FINAL;
/
ALTER TYPE shape NOT FINAL CASCADE CONVERT TO SUBSTITUTABLE;
```

This CASCADE option marks each existing column as SUBSTITUTABLE AT ALL LEVELS and causes a new, hidden column to be added for the TypeId of instances stored in the column. The column can then store subtype instances of the altered type.

If a Type Change Validation Fails

The INVALIDATE option of the ALTER TYPE statement lets you alter a type without propagating the type change to dependent objects. In this case, the system does not validate the dependent types and tables to ensure that all the ramifications of the type change are legal. Instead, all dependent schema objects are marked invalid. The objects, including types and tables, are revalidated when next referenced. If a type cannot be revalidated, it remains invalid, and any tables referencing it become inaccessible until the problem is corrected.

A table may fail validation because, for example, adding a new attribute to a type has caused the number of columns in the table to exceed the maximum allowable number of 1000, or because an attribute used as a partitioning or clustering key of a table was dropped from a type.

To force a revalidation of a type, users can issue the ALTER TYPE COMPILE statement. To force a revalidation of an invalid table, users can issue the ALTER TABLE UPGRADE statement and specify whether the data is to be converted to the latest type version. (Note that, in a table validation triggered by the system when a table is referenced, table data is always updated to the latest type version: you do not have the option to postpone conversion of the data.)

If a table is unable to convert to the latest type version, then INSERT, UPDATE and DELETE statements on the table are not allowed and its data becomes inaccessible. The following DDLs can be executed on the table, but all other statements which reference an invalid table are not allowed until the table is successfully validated:

- DROP TABLE
- TRUNCATE TABLE

All PL/SQL programs containing variables defined using %ROWTYPE of a table or %TYPE of a column or attribute from a table are compiled based on the latest type version. If the table fails the revalidation, then compiling any program units that reference that table will also fail.

ALTER TYPE Statement for Type Evolution

Table 8–1 lists some of the important options in the ALTER TYPE statement for altering the attribute or method definition of a type.

Table 8–1 ALTER TYPE Options for Type Evolution

Option	Description
INVALIDATE	Invalidates all dependent objects. Using this option bypasses all the type and table checks, to save time. Use this option only if you are certain that problems will not be encountered validating dependent types and tables. Table data cannot be accessed again until it is validated; if it cannot be validated, it remains inaccessible.
CASCADE	Propagates the type change to dependent types and tables. The statement aborts if an error is found in dependent types or tables unless the FORCE option is specified. If CASCADE is specified with no other options, then the INCLUDING TABLE DATA option for CASCADE is implied, and Oracle converts all table data to the latest type version.
INCLUDING TABLE DATA	Converts data stored in all user-defined columns to the most recent version of the column's type. For each new attribute added to the column's type, a new attribute is added to the data and is initialized to NULL. For each attribute dropped from the referenced type, the corresponding attribute data is removed from each row in the table. All tablespaces containing the table's data must be in read write mode; otherwise, the statement will not succeed.

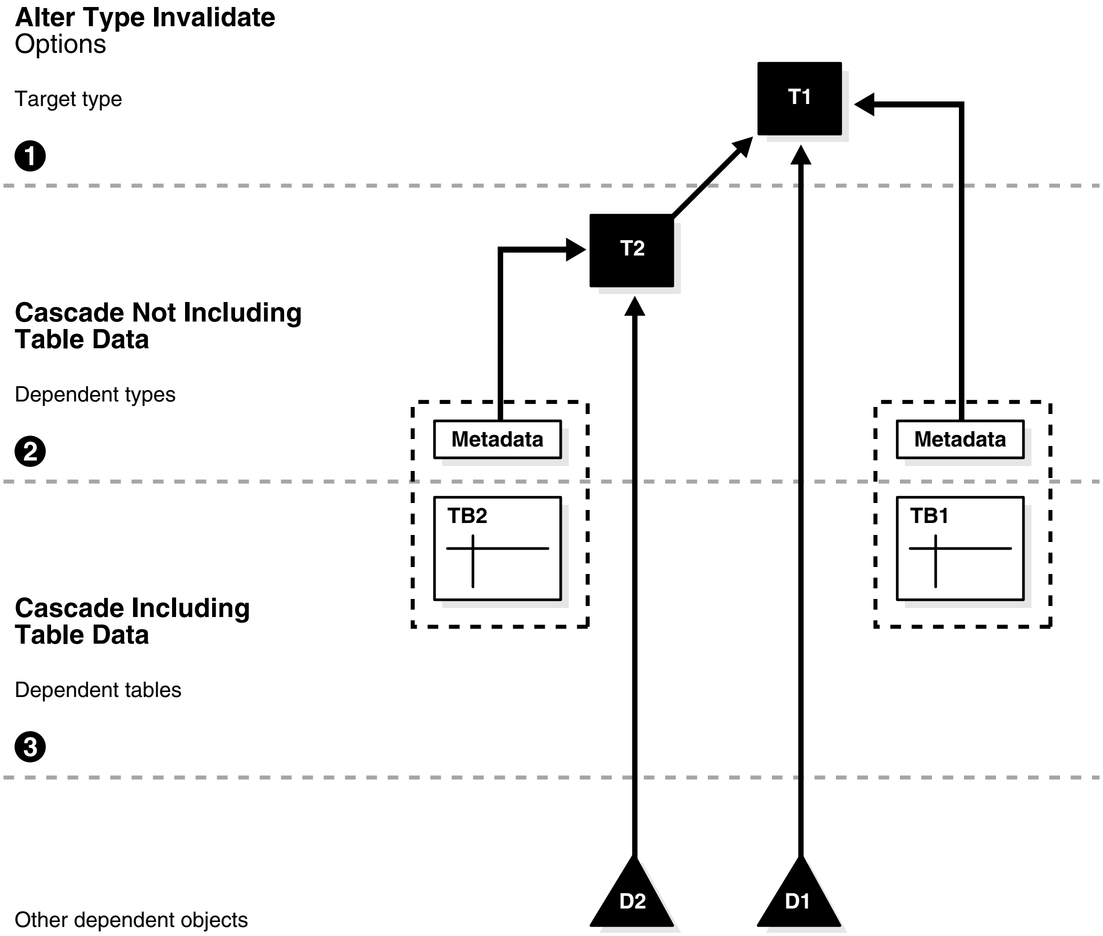
Table 8–1 (Cont.) ALTER TYPE Options for Type Evolution

Option	Description
NOT INCLUDING TABLE DATA	Leaves column data as is, associated with the current type version. If an attribute is dropped from a type referenced by a table, then the corresponding column of the dropped attribute is not removed from the table. Only the metadata of the column is marked unused. If the dropped attribute is stored out-of-line (for example, VARRAY, LOB or nested table attribute) then the out-of-line data is not removed. (Unused columns can be removed afterward by using an ALTER TABLE DROP UNUSED COLUMNS statement.) This option is useful when you have many large tables and may run out of rollback segments if you convert them all in one transaction. This option enables you to convert the data of each dependent table later in a separate transaction (using an ALTER TABLE UPGRADE INCLUDING DATA statement). Specifying this option will speed up the table upgrade because the table's data is left in the format of the old type version. However, selecting data from this table will require converting the images stored in the column to the latest type version. This is likely to affect performance during subsequent SELECT statements. Because this option only requires updating the table's metadata all tablespaces are not required to be on-line in read/write mode for the statement to succeed.
FORCE	Forces the system to ignore errors from dependent tables and indexes. Errors are logged in a specified exception table so that they can be queried afterward. This option must be used with caution because dependent tables may become inaccessible if some table errors occur.
CONVERT TO SUBSTITUTABLE	For use when altering a type from FINAL to NOT FINAL: Converts data stored in all user-defined columns to the most recent version of the column's type and then marks these existing columns and object tables of the type SUBSTITUTABLE AT ALL LEVELS so that they can store any new subtypes of the type that are created. If the type is altered to NOT FINAL without specifying this option, existing columns and tables of the type are marked NOT SUBSTITUTABLE AT ALL LEVELS, and new subtypes of the type cannot be stored in them. You will be able to store such subtypes only in columns and tables created after the type was altered.

> **See Also:** *Oracle Database SQL Language Reference* for information about ALTER TYPE options

Figure 8–1 graphically summarizes the options for ALTER TYPE INVALIDATE and their effects. In the figure, T1 is a type and T2 is a dependent type. See the notes following the figure.

Figure 8–1 ALTER TYPE Options



Notes on the figure:

1. **Invalidate**: All objects following line (1) are marked invalid
2. **Cascade Not Including Table Data**: All objects following line (2) are marked invalid. Metadata of all dependent tables are upgraded to the latest type version, but the table data are not converted.
3. **Cascade Including Table Data**: All objects following line (3) are marked invalid. All dependent tables are upgraded to the latest type version, including the table data.

ALTER TABLE Statement for Type Evolution

You can use ALTER TABLE to convert table data to the latest version of referenced types. For an example of converting table data to latest type version, see "Altering a Type by Adding a Nested Table Attribute" on page 8-7. See Table 8–1 on page 8-12 for a discussion of the INCLUDING DATA option.

> **See Also:** *Oracle Database SQL Language Reference* for information about ALTER TABLE options

The Attribute-Value Constructor

The system-defined attribute value constructor requires you to pass the constructor a value for each attribute of the type. The constructor then sets the attributes of the new object instance to those values, as shown in Example 8–6.

Example 8–6 Setting the Attribute Value with the Constructor

```
CREATE TYPE shape AS OBJECT (
    name VARCHAR2(30),
    area NUMBER);
/
CREATE TABLE building_blocks of shape;

-- Attribute value constructor: Sets instance attributes to the specified values
INSERT INTO building_blocks
  VALUES (
    NEW shape('my_shape', 4));
```

The keyword NEW preceding a call to a constructor is optional but recommended.

Constructors and Type Evolution

The attribute value constructor function saves you the trouble of defining your own constructors for a type. However, with an attribute-value constructor, you must supply a value for every attribute declared in the type. Otherwise the constructor call will fail to compile.

This requirement of an attribute-value constructor can create a problem if you evolve the type later on—by adding an attribute, for example. When you change the attributes of a type, the type's attribute-value constructor changes, too. If you add an attribute, the updated attribute-value constructor expects a value for the new attribute as well as the old ones. As a result, all the attribute-value constructor calls in your existing code, where values for only the old number of attributes are supplied, will fail to compile.

See "Type Evolution" on page 8-5.

Advantages of User-Defined Constructors

User-defined constructors avoid the problem with the attribute-value constructor because user-defined constructors do not need to explicitly set a value for every attribute of a type. A user-defined constructor can have any number of arguments, of any type, and these do not need to map directly to type attributes. In your definition of the constructor, you can initialize the attributes to any appropriate values. Any attributes for which you do not supply values are initialized by the system to NULL.

If you evolve a type—for example, by adding an attribute—calls to user-defined constructors for the type do not need to be changed. User-defined constructors, like ordinary methods, are not automatically modified when the type evolves, so the call signature of a user-defined constructor remains the same. You may, however, need to change the definition of the constructor if you do not want the new attribute to be initialized to NULL.

Defining and Implementing User-Defined Constructors

You define user-defined constructors in the type body, like an ordinary method function. You introduce the declaration and the definition with the phrase CONSTRUCTOR FUNCTION; you must also use the clause RETURN SELF AS RESULT.

A constructor for a type must have the same name as the type. Example 8–7 defines two constructor functions for the shape type. As the example shows, you can overload user-defined constructors by defining multiple versions with different signatures.

Example 8–7 Defining and Implementing User-Defined Constructors

```
CREATE TYPE shape AS OBJECT (
    name VARCHAR2(30),
    area NUMBER,
    CONSTRUCTOR FUNCTION shape(SELF IN OUT NOCOPY shape, name VARCHAR2)
                               RETURN SELF AS RESULT,
    CONSTRUCTOR FUNCTION shape(SELF IN OUT NOCOPY shape, name VARCHAR2,
                               area NUMBER) RETURN SELF AS RESULT
) NOT FINAL;
/

CREATE TYPE BODY shape AS
    CONSTRUCTOR FUNCTION shape(SELF IN OUT NOCOPY shape, name VARCHAR2)
                               RETURN SELF AS RESULT IS
    BEGIN
        SELF.name := name;
        SELF.area := 0;
        RETURN;
    END;
    CONSTRUCTOR FUNCTION shape(SELF IN OUT NOCOPY shape, name VARCHAR2,
                                area NUMBER) RETURN SELF AS RESULT IS
    BEGIN
        SELF.name := name;
        SELF.area := area;
        RETURN;
    END;
END;
/
```

A user-defined constructor has an implicit first parameter SELF. Specifying this parameter in the declaration of a user-defined constructor is optional. If you do specify it, its mode must be declared to be IN OUT.

The required clause RETURN SELF AS RESULT ensures that the most specific type of the instance being returned is the same as the most specific type of the SELF argument. In the case of constructors, this is the type for which the constructor is defined.

For example, if the most specific type of the SELF argument on a call to the shape constructor is shape, then this clause ensures that the shape constructor returns an instance of shape (not an instance of a subtype of shape).

When a constructor function is called, the system initializes the attributes of the SELF argument to NULL. Names of attributes subsequently initialized in the function body may be qualified with SELF, as shown in the preceding example, to distinguish them from the names of the arguments of the constructor function, if these are the same. If the argument names are different, no such qualification is necessary. For example:

```
SELF.name := name;
```

or:

```
name := p1;
```

The function body must include an explicit `return;` as shown. The **return** keyword must not be followed by a `return` expression. The system automatically returns the newly constructed `SELF` instance.

A user-defined constructor may be implemented in PL/SQL, C, or Java.

Overloading and Hiding Constructors

Like other type methods, user-defined constructors can be overloaded.

User-defined constructors are not inherited, so a user-defined constructor defined in a supertype cannot be hidden in a subtype. However, a user-defined constructor does hide, and thus supersede, the attribute-value constructor for its type if the signature of the user-defined constructor exactly matches the signature of the attribute-value constructor. For the signatures to match, the names and types of the parameters (after the implicit `SELF` parameter) of the user-defined constructor must be the same as the names and types of the type's attributes. The mode of each of the user-defined constructor's parameters (after the implicit `SELF` parameter) must be `IN`.

If an attribute-value constructor is not hidden by a user-defined constructor having the same name and signature, the attribute-value constructor can still be called.

Note that, if you evolve a type—for example, by adding an attribute—the signature of the type's attribute-value constructor changes accordingly. This can cause a formerly hidden attribute-value constructor to become usable again.

Calling User-Defined Constructors

A user-defined constructor is called like any other function. You can use a user-defined constructor anywhere you can use an ordinary function.

The `SELF` argument is passed in implicitly and may not be passed in explicitly. In other words, usages like the following are not allowed:

```
NEW constructor(instance, argument_list)
```

A user-defined constructor cannot occur in the `DEFAULT` clause of a `CREATE` or `ALTER TABLE` statement, but an attribute-value constructor can. The arguments to the attribute-value constructor must not contain references to PL/SQL functions or to other columns, including the pseudocolumns `LEVEL`, `PRIOR`, and `ROWNUM`, or to date constants that are not fully specified. The same is true for check constraint expressions: an attribute-value constructor can be used as part of check constraint expressions while creating or altering a table, but a user-defined constructor cannot.

Parentheses are required in SQL even for constructor calls that have no arguments. In PL/SQL, parentheses are optional when invoking a zero-argument constructor. They do, however, make it more obvious that the constructor call is a function call. The following PL/SQL example omits parentheses in the constructor call to create a new shape:

```
shape s := NEW my_schema.shape;
```

The `NEW` keyword and the schema name are optional.

Example 8–8 creates a subtype under the type created in Example 8–7 and shows examples for calling the user-defined constructors.

Example 8–8 Calling User-Defined Constructors

```
CREATE TYPE rectangle UNDER shape (
    len NUMBER,
    wth NUMBER,
    CONSTRUCTOR FUNCTION rectangle(SELF IN OUT NOCOPY rectangle,
        name VARCHAR2, len NUMBER, wth NUMBER) RETURN SELF as RESULT,
    CONSTRUCTOR FUNCTION rectangle(SELF IN OUT NOCOPY rectangle,
        name VARCHAR2, side NUMBER) RETURN SELF as RESULT);
/
SHOW ERRORS
CREATE TYPE BODY rectangle IS
    CONSTRUCTOR FUNCTION rectangle(SELF IN OUT NOCOPY rectangle,
        name VARCHAR2, len NUMBER, wth NUMBER) RETURN  SELF AS RESULT IS
    BEGIN
        SELF.name := name;
        SELF.area := len*wth;
        SELF.len := len;
        SELF.wth := wth;
        RETURN ;
    END;
    CONSTRUCTOR FUNCTION rectangle(SELF IN OUT NOCOPY rectangle,
        name VARCHAR2, side NUMBER) RETURN  SELF AS RESULT IS
    BEGIN
        SELF.name := name;
        SELF.area := side * side;
        SELF.len := side;
        SELF.wth := side;
        RETURN ;
    END;
END;
/

CREATE TABLE shape_table OF shape;
INSERT INTO shape_table VALUES(shape('shape1'));
INSERT INTO shape_table VALUES(shape('shape2', 20));
INSERT INTO shape_table VALUES(rectangle('rectangle', 2, 5));
INSERT INTO shape_table VALUES(rectangle('quadrangle', 12, 3));
INSERT INTO shape_table VALUES(rectangle('square', 12));
```

The following query selects the rows in the `shape_table`:

```
SELECT VALUE(s) FROM shape_table s;

VALUE(S)(NAME, AREA)
---------------------------------------------
SHAPE('shape1', 0)
SHAPE('shape2', 20)
RECTANGLE('rectangle', 10, 2, 5)
RECTANGLE('quadrangle', 36, 12, 3)
RECTANGLE('square', 144, 12, 12)
```

The following PL/SQL code calls the constructor:

```
s shape := NEW shape('void');
```

Constructors for SQLJ Object Types

A SQLJ object type is a SQL object type mapped to a Java class. A SQLJ object type has an attribute-value constructor. It can also have user-defined constructors that are mapped to constructors in the referenced Java class.

Example 8–9 Creating a SQLJ Object

```
CREATE TYPE address AS OBJECT
   EXTERNAL NAME 'university.address' LANGUAGE JAVA
   USING SQLData(
     street   VARCHAR2(100) EXTERNAL NAME 'street',
     city     VARCHAR2(50)  EXTERNAL NAME 'city',
     state    VARCHAR2(50)  EXTERNAL NAME 'state',
     zipcode  NUMBER        EXTERNAL NAME 'zipcode',
    CONSTRUCTOR FUNCTION address (SELF IN OUT NOCOPY address, street VARCHAR2,
                                  city VARCHAR2, state VARCHAR2, zipcode NUMBER)
      RETURN SELF AS RESULT AS LANGUAGE JAVA
      NAME  'university.address (java.lang.String, java.lang.String,
                    java.lang.String, int) return address');
/
```

A SQLJ type of a serialized representation can have only a user-defined constructor. The internal representation of an object of SQLJ type is opaque to SQL, so an attribute-value constructor is not possible for a SQLJ type.

Transient and Generic Types

Oracle has three special SQL data types that enable you to dynamically encapsulate and access type descriptions, data instances, and sets of data instances of any other SQL type, including object and collection types. You can also use these three special types to create anonymous types, including anonymous collection types.

The three SQL types are implemented as opaque types. In other words, the internal structure of these types is not known to the database; their data can be queried only by implementing functions (typically 3GL routines) for the purpose. Oracle provides both an OCI and a PL/SQL API for implementing such functions.

The three generic SQL types are described in Table 8–2.

Table 8–2 Generic SQL Types

Type	Description
SYS.ANYTYPE	A type description type. A SYS.ANYTYPE can contain a type description of any SQL type, named or unnamed, including object types and collection types. An ANYTYPE can contain a type description of a persistent type, but an ANYTYPE itself is transient: in other words, the value in an ANYTYPE itself is not automatically stored in the database. To create a persistent type, use a CREATE TYPE statement from SQL.
SYS.ANYDATA	A self-describing data instance type. A SYS.ANYDATA contains an instance of a given type, with data, plus a description of the type. In this sense, a SYS.ANYDATA is self-describing. An ANYDATA can be persistently stored in the database.
SYS.ANYDATASET	A self-describing data set type. A SYS.ANYDATASET type contains a description of a given type plus a set of data instances of that type. An ANYDATASET can be persistently stored in the database.

Each of these three types can be used with any built-in type native to the database as well as with object types and collection types, both named and unnamed. The types provide a generic way to work dynamically with type descriptions, lone instances, and sets of instances of other types. Using the APIs, you can create a transient ANYTYPE description of any kind of type. Similarly, you can create or convert (cast) a data value of any SQL type to an ANYDATA and can convert an ANYDATA (back) to a SQL type. And similarly again with sets of values and ANYDATASET.

The generic types simplify working with stored procedures. You can use the generic types to encapsulate descriptions and data of standard types and pass the encapsulated information into parameters of the generic types. In the body of the procedure, you can detail how to handle the encapsulated data and type descriptions of whatever type.

You can also store encapsulated data of a variety of underlying types in one table column of type ANYDATA or ANYDATASET. For example, you can use ANYDATA with Advanced Queuing to model queues of heterogeneous types of data. You can query the data of the underlying data types like any other data.

Example 8–10 defines and executes a PL/SQL procedure that uses methods built into SYS.ANYDATA to access information about data stored in a SYS.ANYDATA table column.

Example 8–10 Using SYS.ANYDATA

```
CREATE OR REPLACE TYPE dogowner AS OBJECT (
    ownerno NUMBER, ownername VARCHAR2(10) );
/
CREATE OR REPLACE TYPE dog AS OBJECT (
    breed VARCHAR2(10), dogname VARCHAR2(10) );
/
CREATE TABLE mytab ( id NUMBER, data SYS.ANYDATA );
INSERT INTO mytab VALUES ( 1, SYS.ANYDATA.ConvertNumber (5) );
INSERT INTO mytab VALUES ( 2, SYS.ANYDATA.ConvertObject (
    dogowner ( 5555, 'John') ) );
commit;

CREATE OR REPLACE procedure P IS
  CURSOR cur IS SELECT id, data FROM mytab;

  v_id mytab.id%TYPE;
  v_data mytab.data%TYPE;
  v_type SYS.ANYTYPE;
  v_typecode PLS_INTEGER;
  v_typename VARCHAR2(60);
  v_dummy PLS_INTEGER;
  v_n NUMBER;
  v_dogowner dogowner;
  non_null_anytype_for_NUMBER exception;
  unknown_typename exception;

BEGIN
  OPEN cur;
    LOOP
      FETCH cur INTO v_id, v_data;
      EXIT WHEN cur%NOTFOUND;
      v_typecode := v_data.GetType ( v_type /* OUT */ );
      CASE v_typecode
        WHEN Dbms_Types.Typecode_NUMBER THEN
          IF v_type IS NOT NULL
            THEN RAISE non_null_anytype_for_NUMBER; END IF;
          v_dummy := v_data.GetNUMBER ( v_n /* OUT */ );
          Dbms_Output.Put_Line (
            To_Char(v_id) || ': NUMBER = ' || To_Char(v_n) );
    WHEN Dbms_Types.Typecode_Object THEN
     v_typename := v_data.GetTypeName();
     IF v_typename NOT IN ( 'HR.DOGOWNER' )
      THEN RAISE unknown_typename; END IF;
     v_dummy := v_data.GetObject ( v_dogowner /* OUT */ );
     Dbms_Output.Put_Line (
      To_Char(v_id) || ': user-defined type = ' || v_typename ||
      '(' || v_dogowner.ownerno || ', ' || v_dogowner.ownername || ' )' );
   END CASE;
  END LOOP;
  CLOSE cur;

EXCEPTION
 WHEN non_null_anytype_for_NUMBER THEN
   RAISE_Application_Error ( -20000,
    'Paradox: the return AnyType instance FROM GetType ' ||
    'should be NULL for all but user-defined types' );
 WHEN unknown_typename THEN
   RAISE_Application_Error ( -20000,
    'Unknown user-defined type ' || v_typename ||
    ' - program written to handle only HR.DOGOWNER' );
END;
/

SELECT t.data.gettypename() FROM mytab t;
SET SERVEROUTPUT ON;
EXEC P;
```

The query and the procedure P in the preceding code sample produce output like the following:

```
T.DATA.GETTYPENAME()
--------------------------------------------------------------
SYS.NUMBER
HR.DOGOWNER

1: NUMBER = 5
2: user-defined type = HR.DOGOWNER(5555, John )
```

Corresponding to the three generic SQL types are three OCI types that model them. Each has a set of functions for creating and accessing the respective type:

- `OCIType`, corresponding to `SYS.ANYTYPE`
- `OCIAnyData`, corresponding to `SYS.ANYDATA`
- `OCIAnyDataSet`, corresponding to `SYS.ANYDATASET`

See Also:

- *Oracle Call Interface Programmer's Guide* for the `OCIType`, `OCIAnyData`, and `OCIAnyDataSet` APIs and details on how to use them.
- *Oracle Database PL/SQL Packages and Types Reference* for information about the interfaces to the `ANYTYPE`, `ANYDATA`, and `ANYDATASET` types and about the `DBMS_TYPES` package, which defines constants for built-in and user-defined types, for use with `ANYTYPE`, `ANYDATA`, and `ANYDATASET`.

User-Defined Aggregate Functions

Oracle provides a number of pre-defined aggregate functions such as MAX, MIN, SUM for performing operations on a set of records. These pre-defined aggregate functions can be used only with scalar data. However, you can create your own custom implementations of these functions, or define entirely new aggregate functions, to use with complex data—for example, with multimedia data stored using object types, opaque types, and LOBs.

User-defined aggregate functions are used in SQL DML statements just like Oracle's own built-in aggregates. Once such functions are registered with the server, Oracle simply invokes the aggregation routines that you supplied instead of the native ones.

User-defined aggregates can be used with scalar data as well. For example, it may be worthwhile to implement special aggregate functions for working with complex statistical data associated with financial or scientific applications.

User-defined aggregates are a feature of the Extensibility Framework. You implement them using ODCIAggregate interface routines.

> **See Also:** *Oracle Database Data Cartridge Developer's Guide* for information on using the ODCIAggregate interface routines to implement user-defined aggregate functions

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. Oracle extends 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. See also *Oracle Database SecureFiles and Large Objects Developer's Guide*.

Example 8–11 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.

Example 8–11 Partitioning a Table That Contains Objects

```
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)));
```

How Locators Improve the Performance of Nested Tables

Collection types do 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. Because 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.

9

Design Considerations for Oracle Objects

This chapter explains the implementation and performance characteristics of the Oracle 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. You should be familiar with the basic concepts behind Oracle objects before you read this chapter.

This chapter covers the following topics:

General Storage Considerations for Objects

This section discusses general storage considerations for various object types.

Storing 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. Because each person may have more than one phone number, you need to create a nested table type based on the phone number object type

First, to create the name_objtyp object type, enter the SQL statements in Example 9–1.

Example 9–1 Creating Object Types for Columns in a Table

```
CREATE TYPE name_objtyp AS OBJECT (
  first       VARCHAR2(15),
  middle      VARCHAR2(15),
  last        VARCHAR2(15));
/
CREATE TYPE address_objtyp AS OBJECT (
  street       VARCHAR2(200),
  city         VARCHAR2(200),
  state        CHAR(2),
  zipcode      VARCHAR2(20));
/
CREATE TYPE phone_objtyp AS OBJECT (
  location     VARCHAR2(15),
  num          VARCHAR2(14));
/
CREATE TYPE phone_ntabtyp AS TABLE OF phone_objtyp;
/
```

See "Design Considerations for Nested Tables" on page 9-10 for more information about nested tables.

After 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 SQL statement in Example 9–2.

Example 9–2 Creating a Table with Column Objects

```
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 9–1 Representation of the people_reltab Relational Table

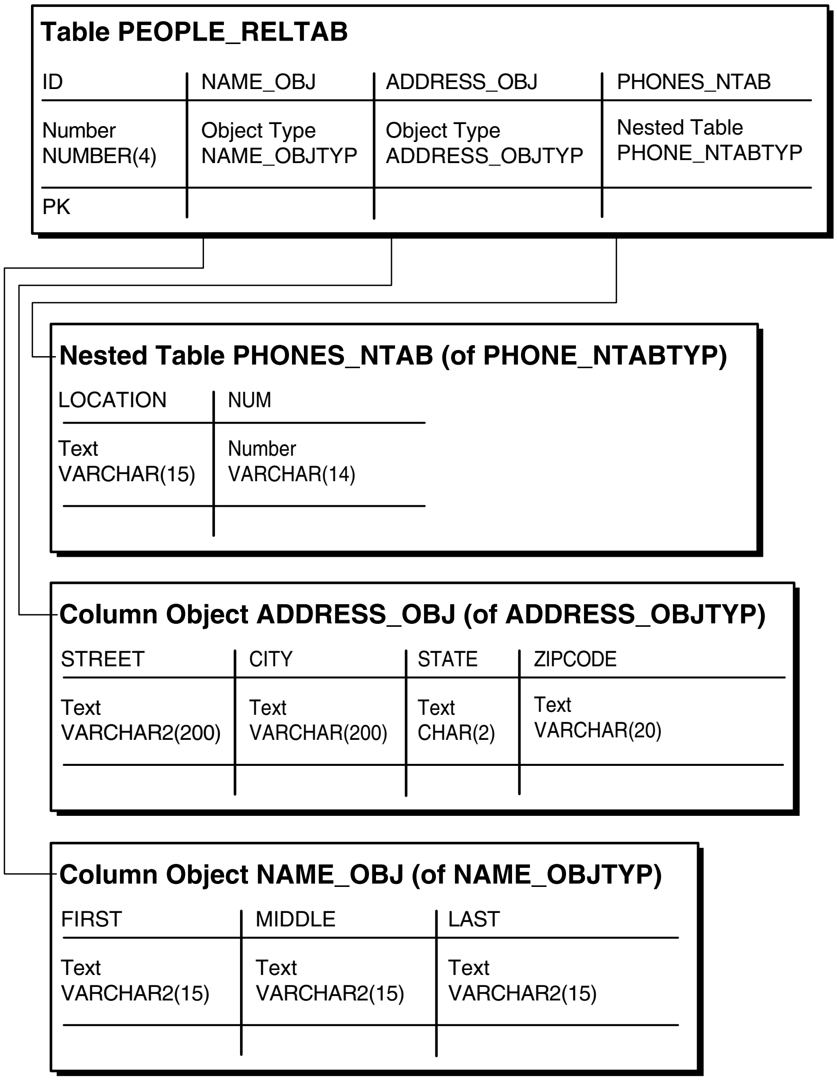


Table PEOPLE_RELTAB

ID	NAME_OBJ	ADDRESS_OBJ	PHONES_NTAB
Number NUMBER(4)	Object Type NAME_OBJTYP	Object Type ADDRESS_OBJTYP	Nested Table PHONE_NTABTYP
PK			

Nested Table PHONES_NTAB (of PHONE_NTABTYP)

LOCATION	NUM
Text VARCHAR(15)	Number VARCHAR(14)

Column Object ADDRESS_OBJ (of ADDRESS_OBJTYP)

STREET	CITY	STATE	ZIPCODE
Text VARCHAR2(200)	Text VARCHAR(200)	Text CHAR(2)	Text VARCHAR(20)

Column Object NAME_OBJ (of NAME_OBJTYP)

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

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

The `people_reltab` table and its columns and related types 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 indicators 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 `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 `CEIL(3/8)` or `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 lower than 8.1.0, such as 8.0.0, the storage is determined by the following calculation:

```
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:

```
CEIL(4/8) + 6 = 7
```

`CEIL(4/8)` is `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 9-8.

> **See Also:** *Oracle Database SQL Language 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.

Storage Considerations for Object Identifiers (OIDs)

By default, every row object in an object table has an associated logical object identifier (OID) that uniquely identifies it in an object table. Oracle assigns each row object a unique system-generated OID, 16 bytes in length that is automatically indexed for efficient OID-based lookups. The OID column is the equivalent of having an extra 16-byte primary key column. In a distributed and replicated environment, the system-generated unique identifier lets Oracle identify objects unambiguously.

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.

An OID allows the corresponding row object to be referred to from other objects or from relational tables. A built-in data type called a `REF` represents such references. A `REF` encapsulates a reference to a row object of a specified object type.

`REF`s 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 9-4. 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 for each row.

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.

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.

Primary-key based identifiers also make it faster and easier to load 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.

However, if each primary key value requires more than 16 bytes of storage and you have a large number of `REF`s, 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.

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.

Design Considerations for REFs

This section discusses considerations when working with REFs.

- Storage Size of REFs
- Integrity Constraints for REF Columns
- Performance and Storage Considerations for Scoped REFs
- Speeding up Object Access Using the WITH ROWID Option

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 instead of unscoped REFs because scoped REFs are stored more efficiently. Whereas an unscoped REF takes at least 36 bytes to store (more if it uses rowids), a scoped REF is stored as just the OID of its target object and can take less than 16 bytes, depending on whether the referenced OID is system-generated or primary-key based. A system-generated OID requires 16 bytes; a PK-based OID requires enough space to store the primary key value, which may be less than 16 bytes. However, a REF to a PK-based OID, which must be dynamically constructed on being selected, may take more space in memory than a REF to a system-generated OID.

Besides requiring less storage space, scoped REFs often enable the optimizer to optimize queries that dereference a scoped REF into more 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.

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 in the "Speeding up Object Access Using the WITH ROWID Option" on page 9-7, 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. This allows you to 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 ;
```

A people_reltab2 table can be created that has the same definition as the people_reltab table shown in Example 9–2 on page 9-2, except that a REF is used for the address. Next, an index can be created on the address_ref column.

Example 9–3 Creating an Index on Scoped REF Columns

```
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   phone_ntabtyp)
  NESTED TABLE  phones_ntab STORE AS phone_store_ntab2 ;

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 preceding 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' ;
```

The Oracle query 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 10 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.

Design Considerations for Collections

This section discusses considerations when working with collections.

- Viewing Object Data in Relational Form with Unnesting Queries
- Storage Considerations for Varrays
- Performance of Varrays Versus Nested Tables
- Design Considerations for Nested Tables
- Design Considerations for Multilevel Collections

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 single-level and multilevel collections of either nested tables or varrays. This section contains examples of unnesting queries.

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

Example 9–4 Unnesting a Nested Table with the TABLE Function

```
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 retrieve even parent rows that have no child rows (no phone numbers, in this case), use the outer join syntax, with the +. For example:

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

If the SELECT list of a query does not refer to any columns from the parent table other than the nested table column, the query is optimized to execute only against the nested table's storage table.

The unnesting query syntax is the same for varrays as for nested tables. For instance, suppose the phones_ntab nested table is instead a varray named phones_var. The following example shows how to use the TABLE syntax to query the varray:

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

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 like the following:

Example 9–5 Creating the home_phones Function

```
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;
         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:

Example 9–6 Using the TABLE Function to Unnest a Query

```
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:** *Oracle Database SQL Language Reference* and *Oracle Database Data Cartridge Developer's Guide* for more information about the `TABLE` function

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. Because the storage of varrays incurs some overhead, such as null information, 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 `LOB`s. 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 `LOB`s. Otherwise, the varray is stored in the column itself as a raw value. In addition, Oracle supports inline LOBs which means that 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. See also *Oracle Database SecureFiles and Large Objects Developer's Guide*.

Propagating VARRAY Size Change

When changing the size of a `VARRAY` type, a new type version is generated for the dependent types. It is important to be aware of this when a `VARRAY` column is not explicitly stored as a LOB and its maximum size is originally smaller than 4000 bytes. If the size is larger than or equal to 4000 bytes after the increase, the `VARRAY` column has to be stored as a LOB. This requires an extra operation to upgrade the metadata of the `VARRAY` column in order to set up the necessary LOB metadata information including the LOB segment and LOB index.

The `CASCADE` option in the `ALTER TYPE` statement propagates the `VARRAY` size change to its dependent types and tables. A new version is generated for each valid dependent type and dependent tables metadata are updated accordingly based on the different case scenarios described previously. If the `VARRAY` column is in a cluster table, an `ALTER TYPE` statement with the `CASCADE` option fails because a cluster table does not support a LOB.

The `CASCADE` option in the `ALTER TYPE` statement also provides the `[NOT] INCLUDING TABLE DATA` option. The `NOT INCLUDING TABLE DATA` option only updates the metadata of the table, but does not convert the data image. In order to convert the `VARRAY` image to the latest version format, you can either specify `INCLUDING TABLE DATA` explicitly in `ALTER TYPE CASCADE` statement or issue `ALTER TABLE UPGRADE` statement.

Performance of Varrays Versus 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 "Viewing Object Data in Relational Form with Unnesting Queries" on page 9-8 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.

Design Considerations for Nested Tables

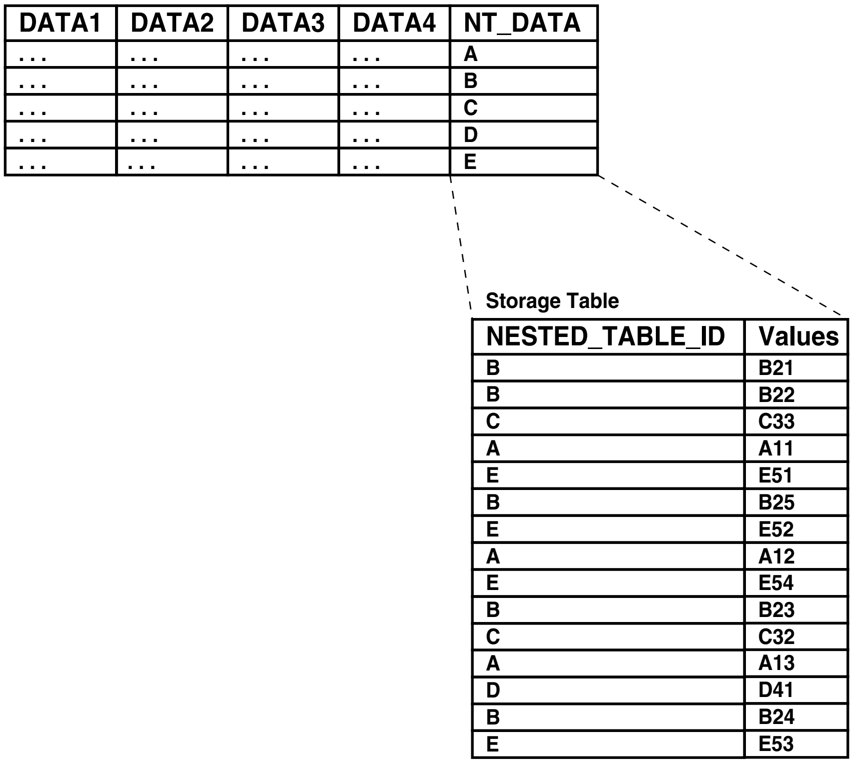
The following sections contain design considerations for using nested tables.

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 9–2 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 9–2, all of the values that belong to nested table A are identified, all of the values that belong to nested table B are identified, and so on.

Figure 9–2 Nested Table Storage



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

Storage Table

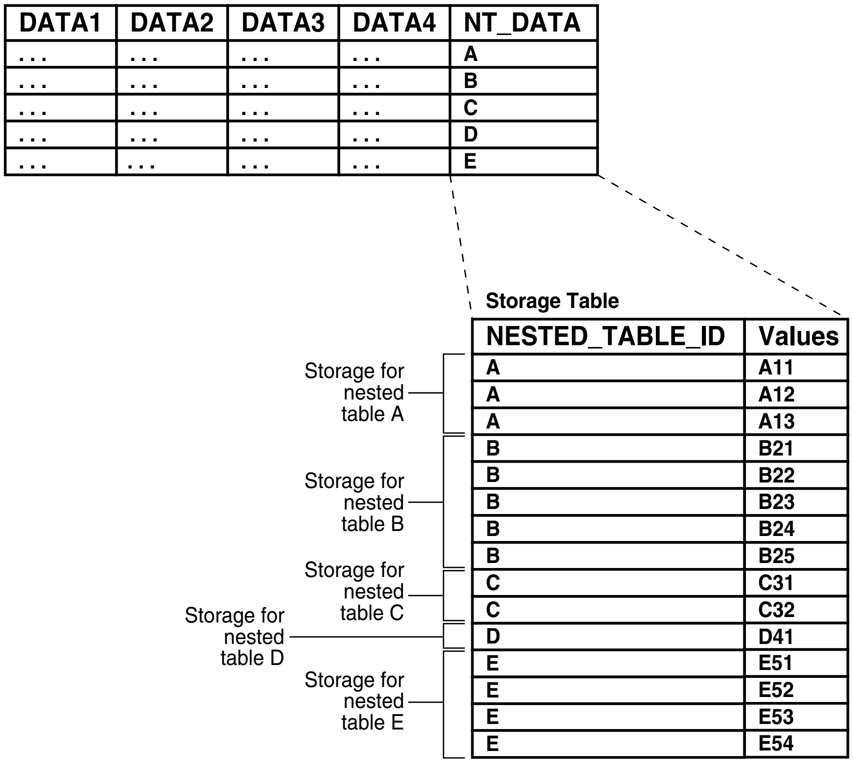
NESTED_TABLE_ID	Values
B	B21
B	B22
C	C33
A	A11
E	E51
B	B25
E	E52
A	A12
E	E54
B	B23
C	C32
A	A13
D	D41
B	B24
E	E53

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 child rows together. So, when a parent row is accessed, all its child rows can be efficiently retrieved. When only parent rows are accessed, efficiency is maintained because the child rows are not inter-mixed with the parent rows.

Figure 9–3 shows how the storage table works when the nested table is in an IOT. The storage table groups by NESTED_TABLE_ID the values for each nested table in a nested table column. In Figure 9–3, for each nested table in the NT_DATA column of the parent table, the data is grouped in the storage table: all of the values in nested table A are grouped together, all of the values in nested table B are grouped together, and so on.

Figure 9–3 Nested Table in IOT Storage



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

Storage Table

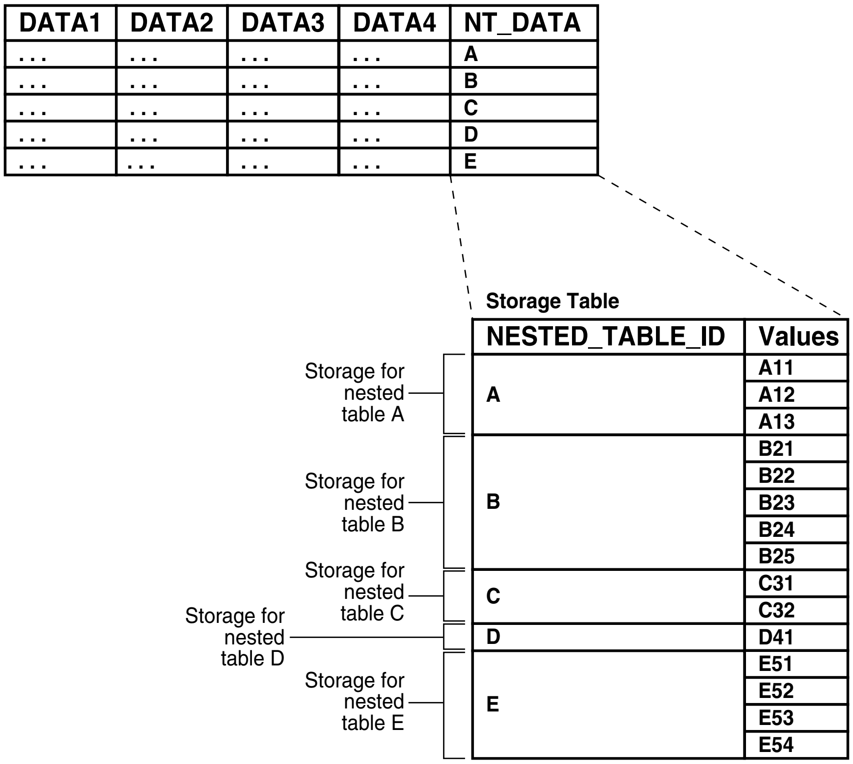
NESTED_TABLE_ID	Values
A	A11
A	A12
A	A13
B	B21
B	B22
B	B23
B	B24
B	B25
C	C31
C	C32
D	D41
E	E51
E	E52
E	E53
E	E54

Storage for nested table A
Storage for nested table B
Storage for nested table C
Storage for nested table D
Storage for nested table E

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 for each group, as illustrated in Figure 9–4.

Figure 9–4 Nested Table in IOT Storage with Compression



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

Storage Table

NESTED_TABLE_ID	Values
A	A11
	A12
	A13
B	B21
	B22
	B23
	B24
	B25
C	C31
	C32
D	D41
E	E51
	E52
	E53
	E54

Storage for nested table A
Storage for nested table B
Storage for nested table C
Storage for nested table D
Storage for nested table E

In general, Oracle 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 child rows can be accessed on demand; the child sets also can be filtered. Using nested table locators enables you to avoid unnecessarily transporting child rows for every parent.

You can perform either one of the following actions to access the child 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.

In a multilevel collection, you can use a locator with a specified collection at any level of nesting. Following are described two ways in which to specify that a collection is to be retrieved as a locator.

At Table Creation Time

When the collection type is being used as a column type and the NESTED TABLE storage clause is used, you can use the RETURN AS LOCATOR clause to specify that a particular collection is to be retrieved as a locator.

For instance, suppose that inner_table is a collection type consisting of three levels of nested tables. In the following example, the RETURN AS LOCATOR clause specifies that the third level of nested tables is always to be retrieved as a locator.

Example 9–7 Using the RETURN AS LOCATOR Clause

```
CREATE TYPE inner_table AS TABLE OF NUMBER;
/
CREATE TYPE middle_table AS TABLE OF inner_table;
/
CREATE TYPE outer_table AS TABLE OF middle_table;
/
CREATE TABLE tab1 (
  col1 NUMBER,
  col2 outer_table)
 NESTED TABLE col2 STORE AS col2_ntab
  (NESTED TABLE COLUMN_VALUE STORE AS cval1_ntab
    (NESTED TABLE COLUMN_VALUE STORE AS cval2_ntab RETURN AS LOCATOR) );
```

As a HINT During Retrieval

A query can retrieve a collection as a locator by means of the hint NESTED_TABLE_GET_REFS. Here is an example of retrieving the column col2 from the table tab1 as a locator:

```
SELECT /*+ NESTED_TABLE_GET_REFS +*/ col2
  FROM tab1
 WHERE col1 = 2;
```

Unlike with the RETURN AS LOCATOR clause, however, you cannot specify a particular inner collection to return as a locator when using the hint.

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 semijoin. However, this optimization only happens if the ALWAYS_SEMI_JOIN initialization parameter is set. If you want to perform semijoins, the valid values for this parameter are MERGE and HASH; these parameter values indicate which join method to use.

Note: In the preceding example, 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.

Design Considerations for Multilevel Collections

Chapter 3, "Support for Collection Data Types" describes how to nest collection types to create a true multilevel collection, such as a nested table of nested tables, a nested table of varrays, a varray of nested tables, or a varray or nested table of an object type that has an attribute of a collection type.

You can also nest collections indirectly using REFs. For example, you can create a nested table of an object type that has an attribute that references an object that has a nested table or varray attribute. If you do not actually need to access all elements of a multilevel collection, then nesting a collection with REFs may provide better performance because only the REFs need to be loaded, not the elements themselves.

True multilevel collections (specifically multilevel nested tables) perform better for queries that access individual elements of the collection. Using nested table locators can improve the performance of programmatic access if you do not need to access all elements.

For an example of a collection that uses REFs to nest another collection, suppose you want to create a new object type called person_objtyp using the object types shown in Example 9–1 on page 9-2, 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.

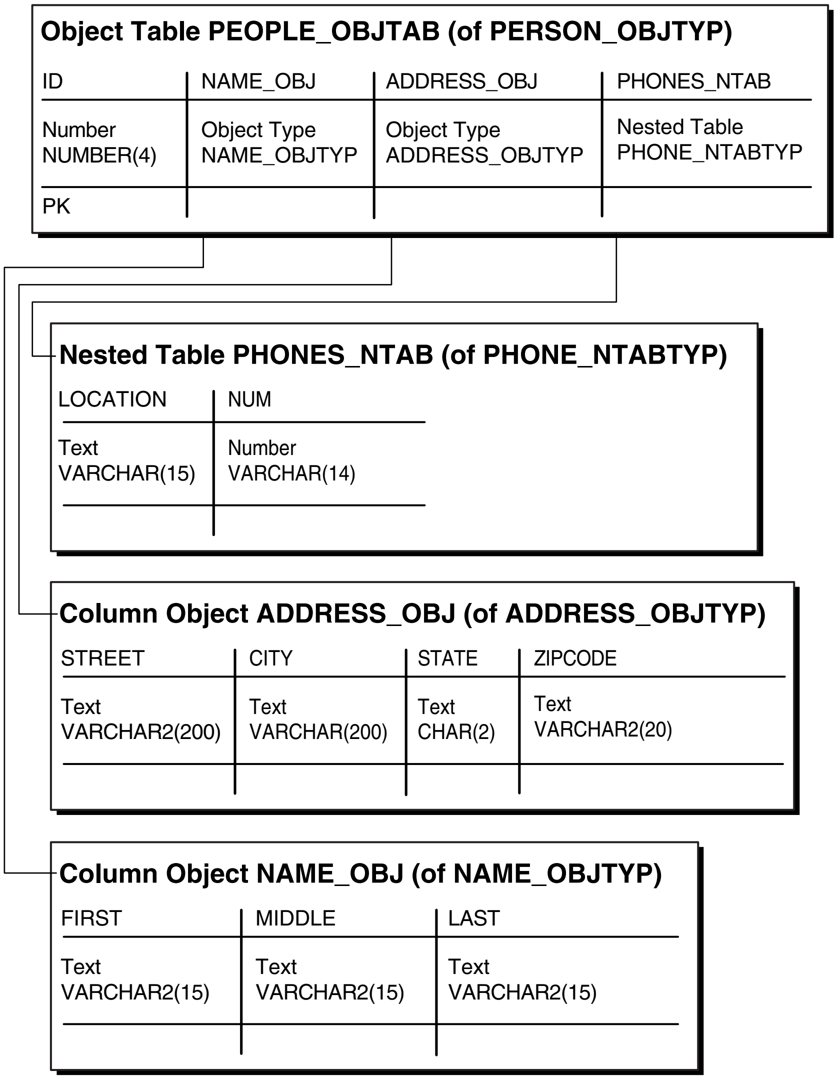
To create the person_objtyp object type and an object table called people_objtab of person_objtyp object type, issue the following SQL statement:

Example 9–8 Creating an Object Table with a Multilevel Collection

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

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 9-2. 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 9–5 Object-Relational Representation of the people_objtab Object Table



Object Table PEOPLE_OBJTAB (of PERSON_OBJTYP)

ID	NAME_OBJ	ADDRESS_OBJ	PHONES_NTAB
Number NUMBER(4)	Object Type NAME_OBJTYP	Object Type ADDRESS_OBJTYP	Nested Table PHONE_NTABTYP
PK			

Nested Table PHONES_NTAB (of PHONE_NTABTYP)

LOCATION	NUM
Text VARCHAR(15)	Number VARCHAR(14)

Column Object ADDRESS_OBJ (of ADDRESS_OBJTYP)

STREET	CITY	STATE	ZIPCODE
Text VARCHAR2(200)	Text VARCHAR(200)	Text CHAR(2)	Text VARCHAR2(20)

Column Object NAME_OBJ (of NAME_OBJTYP)

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

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 `REF`s to the `people_objtab` for the project leads, and you can use a nested table collection of `REF`s 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`, then create the object table `projects_objtab` based on the `projects_objtyp` as shown in Example 9–9.

Example 9–9 Creating an Object Table Using REFs

```
CREATE TYPE projects_objtyp AS OBJECT (
   id             NUMBER(4),
   title          VARCHAR2(15),
   projlead_ref   REF person_objtyp,
   description    CLOB,
   team_ntab      personref_ntabtyp);
/
CREATE TABLE projects_objtab OF projects_objtyp (id PRIMARY KEY)
   NESTED TABLE team_ntab STORE AS team_store_ntab ;
```

Figure 9–6 Object-Relational Representation of the projects_objtab Object Table

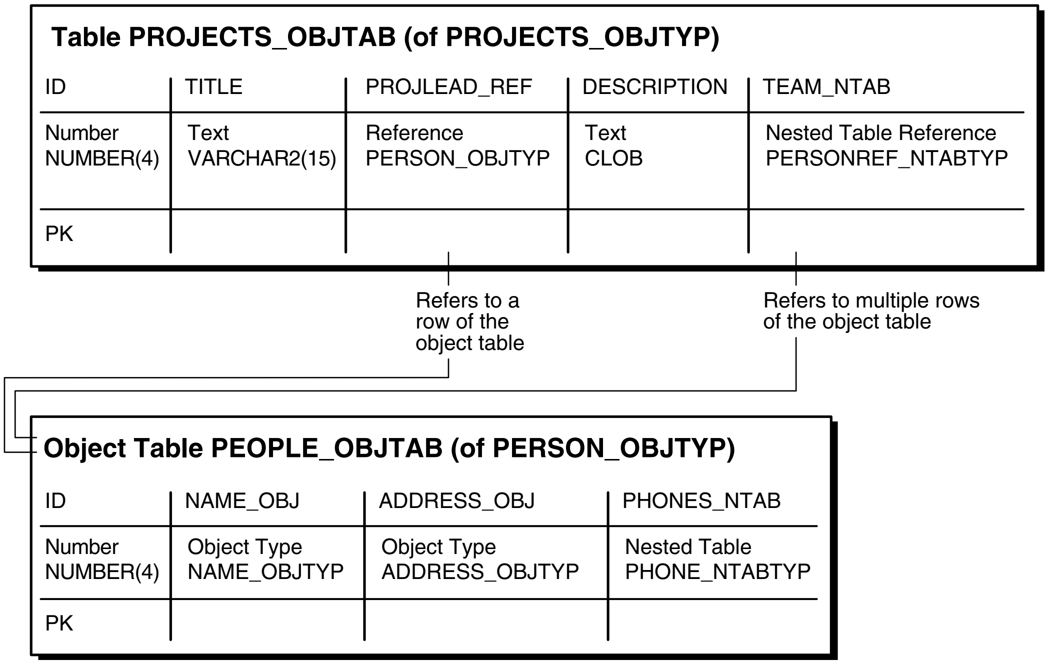


Table PROJECTS_OBJTAB (of PROJECTS_OBJTYP)

ID	TITLE	PROJLEAD_REF	DESCRIPTION	TEAM_NTAB
Number NUMBER(4)	Text VARCHAR2(15)	Reference PERSON_OBJTYP	Text CLOB	Nested Table Reference PERSONREF_NTABTYP
PK				

Refers to a row of the object table
Refers to multiple rows of the object table

Object Table PEOPLE_OBJTAB (of PERSON_OBJTYP)

ID	NAME_OBJ	ADDRESS_OBJ	PHONES_NTAB
Number NUMBER(4)	Object Type NAME_OBJTYP	Object Type ADDRESS_OBJTYP	Nested Table PHONE_NTABTYP
PK			

After 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 as shown in Example 9–10.

Example 9–10 Inserting Values into the people_objtab Object Table

```
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 Example 9–11.

Example 9–11 Inserting Values into the projects_objtab Object Table

```
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 `REF`s, but you also can store `REF`s in varrays. That is, you can have a varray of `REF`s.

Design Considerations for Methods

This section discusses considerations when working with methods.

- Choosing a Language for Method Functions
- Static Methods
- Using SELF IN OUT NOCOPY with Member Procedures
- Function-Based Indexes on the Return Values of Type Methods

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 procedures. In contrast, a method implemented in Java or PL/SQL executes in the same process as the server.

Example: Implementing a Method

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.

The object type specification and its methods are shown in Example 9–12.

Example 9–12 Creating an Object Type with Methods Implemented in Different Languages

```
CREATE LIBRARY myCfuncs TRUSTED AS STATIC
/

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
   IS
    imgname  VARCHAR2(100);
    sqlstmt VARCHAR2(200);
   BEGIN
      sqlstmt := 'SELECT name INTO imgname FROM imgtab WHERE imgid = id';
      EXECUTE IMMEDIATE sqlstmt;
      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(oracle.sql.NUMBER) return oracle.sql.BLOB';

END;
/
```

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

See Also:

- *Oracle Database Java Developer's Guide* for more information
- Chapter 5, "Object Support in Oracle Programming 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.

Example 9–13 shows 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:.

Example 9–13 Creating an Object Type with a STATIC Method

```
CREATE TYPE atype AS OBJECT(
   a1 NUMBER,
   STATIC PROCEDURE newa (
      p1        NUMBER,
      tabname   VARCHAR2,
      schname   VARCHAR2));
/
CREATE 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', 'HR');
END;
/
```

Using SELF IN OUT NOCOPY with Member Procedures

In member procedures, if SELF is not declared, its parameter mode defaults to IN OUT. However, the default behavior does not include the NOCOPY compiler hint. See "Member Methods" on page 2-7.

Because the value of the IN OUT actual parameter is copied into the corresponding formal parameter, the copying slows down execution when the parameters hold large data structures such as instances of large object types.

For performance reasons, you may want to include SELF IN OUT NOCOPY when passing a large object type as a parameter. For example:

```
MEMBER PROCEDURE my_proc (SELF IN OUT NOCOPY my_LOB)
```

See Also:

- *Oracle Database PL/SQL User's Guide and Reference* for information on performance issues and restrictions on the use of NOCOPY
- *Oracle Database SQL Language Reference* for information about using NOCOPY in the CREATE PROCEDURE statement

Function-Based Indexes on the Return Values of Type Methods

A function-based index is an index based on the return values of an expression or function. The function may be a method function of an object type.

A function-based index built on a method function precomputes the return value of the function for each object instance in the column or table being indexed and stores those values in the index. There they can be referenced without having to evaluate the function again.

Function-based indexes are useful for improving the performance of queries that have a function in the WHERE clause. For example, the following code contains a query of an object table emps:

```
CREATE TYPE emp_t AS OBJECT(
  name   VARCHAR2(36),
  salary NUMBER,
  MEMBER FUNCTION bonus RETURN NUMBER DETERMINISTIC);
/
CREATE TYPE BODY emp_t IS
 MEMBER FUNCTION bonus RETURN NUMBER DETERMINISTIC IS
 BEGIN
  RETURN self.salary * .1;
 END;
END;
/

CREATE TABLE emps OF emp_t ;

SELECT e.name
  FROM emps e
  WHERE e.bonus() > 2000;
```

To evaluate this query, Oracle must evaluate bonus() for each row object in the table. If there is a function-based index on the return values of bonus(), then this work has already been done, and Oracle can simply look up the results in the index. This enables Oracle to return a result from the query more quickly.

Return values of a function can be usefully indexed only if those values are constant, that is, only if the function always returns the same value for each object instance. For this reason, to use a user-written function in a function-based index, the function must have been declared with the DETERMINISTIC keyword, as in the preceding example. This keyword promises that the function always returns the same value for each object instance's set of input argument values.

The following example creates a function-based index on the method bonus() in the table emps:

Example 9–14 Creating a Function-Based Index on a Method

```
CREATE INDEX emps_bonus_idx ON emps x (x.bonus()) ;
```

> **See Also:** *Oracle Database Concepts* and *Oracle Database SQL Language Reference* for detailed information about function-based indexes

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 OE execute privileges on type atype created by HR in "Static Methods" on page 9-20, and then create table atab based on the type:

```
GRANT EXECUTE ON atype TO oe;
CONNECT oe/oe;
CREATE TABLE atab OF HR.atype ;
```

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

```
BEGIN -- follwing call raises an error, insufficient privileges
  HR.atype.newa(1, 'atab', 'OE');
END;
/
```

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

```
DROP TABLE atab;
CONNECT hr/hr;
DROP TABLE atab;
DROP TYPE atype FORCE;
COMMIT;

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

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

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

```
GRANT EXECUTE ON atype TO oe;
CONNECT oe/oe;
CREATE TABLE atab OF HR.atype;

BEGIN
  HR.atype.newa(1, 'atab', 'OE');
END;
/
DROP TABLE atab;
CONNECT hr/hr;
DROP TYPE atype FORCE;
```

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

In a type hierarchy, a subtype has the same rights model as its immediate supertype. That is, it implicitly inherits the rights model of the supertype and cannot explicitly specify one. Furthermore, if the supertype was declared with definer rights, the subtype must reside in the same schema as the supertype. These rules allow invoker-rights type hierarchies to span schemas. However, type hierarchies that use a definer-rights model must reside within a single schema. For example:

```
CREATE TYPE deftype1 AS OBJECT (...); --Definer-rights type
CREATE TYPE subtype1 UNDER deftype1 (...); --subtype in same schema as supertype
CREATE TYPE schema2.subtype2 UNDER deftype1 (...); --ERROR
CREATE TYPE invtype1 AUTHID CURRENT_USER AS OBJECT (...); --Invoker-rights type
CREATE TYPE schema2.subtype2 UNDER invtype1 (...); --LEGAL
```

Using Roles with Invoker's Rights Subprograms

The use of roles in a subprogram depends on whether it executes with definer's rights or invoker's rights. Within a definer's rights subprogram, all roles are disabled. Roles are not used for privilege checking, and you cannot set roles.

Within an invoker's rights subprogram, roles are enabled (unless the subprogram was called directly or indirectly by a definer's rights subprogram). Roles are used for privilege checking, and you can use native dynamic SQL to set roles for the session. However, you cannot use roles to grant privileges on template objects because roles apply at run time, not at compile time.

Replicating Object Tables and Columns

Object tables and object views can be replicated as materialized views. You can also replicate relational tables that contain columns of an object, collection, or REF type. Such materialized views are called object-relational materialized views.

All user-defined types required by an object-relational materialized view must exist at the materialized view site as well as at the master site. They must have the same object type IDs and versions at both sites.

Replicating Columns of Object, Collection, or REF Type

To be updatable, a materialized view based on a table that contains an object column must select the column as an object in the query that defines the view: if the query selects only certain attributes of the column's object type, then the materialized view is read-only.

The view-definition query can also select columns of collection or REF type. REFs can be either primary-key based or have a system-generated key, and they can be either scoped or unscoped. Scoped REF columns can be rescoped to a different table at the site of the materialized view—for example, to a local materialized view of the master table instead of the original, remote table.

Replicating Object Tables

A materialized view based on an object table is called an object materialized view. Such a materialized view is itself an object table. An object materialized view is created by adding the OF *type* keyword to the CREATE MATERIALIZED VIEW statement. For example:

```
CREATE MATERIALIZED VIEW customer OF cust_objtyp AS
  SELECT * FROM HR.Customer_objtab@dbs1;
```

As with an ordinary object table, each row of an object materialized view is an object instance, so the view-definition query that creates the materialized view must select entire objects from the master table: the query cannot select only a subset of the object type's attributes. For example, the following materialized view is not allowed:

```
CREATE MATERIALIZED VIEW customer OF cust_objtyp AS
  SELECT CustNo FROM HR.Customer_objtab@dbs1;
```

You can create an object-relational materialized view from an object table by omitting the OF *type* keyword, but such a view is read-only: you cannot create an updatable object-relational materialized view from an object table.

For example, the following CREATE MATERIALIZED VIEW statement creates a read-only object-relational materialized view of an object table. Even though the view-definition query selects all columns and attributes of the object type, it does not select them as attributes of an object, so the view created is object-relational and read-only:

```
CREATE MATERIALIZED VIEW customer AS
  SELECT * FROM HR.Customer_objtab@dbs1;
```

For both object-relational and object materialized views that are based on an object table, if the type of the master object table is not FINAL, the FROM clause in the materialized view definition query must include the ONLY keyword. For example:

```
CREATE MATERIALIZED VIEW customer OF cust_objtyp AS
```

```
SELECT CustNo FROM ONLY HR.Customer_objtab@dbs1;
```

Otherwise, the `FROM` clause must omit the `ONLY` keyword.

> **See Also:** *Oracle Database Advanced Replication* for more information on replicating object tables and columns

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:

Example 9–15 Specifying Constraints on an Object Type When Creating a Table

```
CREATE TYPE customer_typ AS OBJECT(
   cust_id INTEGER);
/
CREATE TYPE department_typ AS OBJECT(
   deptno INTEGER);
/
CREATE TABLE customer_tab OF customer_typ (
   cust_id default 1 NOT NULL);

CREATE TABLE department_tab OF department_typ (
   deptno PRIMARY KEY);

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

Considerations Related to Type Evolution

The following sections contain design considerations relating to type evolution.

Pushing a Type Change Out to Clients

Once a type has evolved on the server side, all client applications using this type need to make the necessary changes to structures associated with the type. You can do this with OTT/JPUB. You also may need to make programmatic changes associated with the structural change. After making these changes, you must recompile your application and relink.

Types may be altered between releases of a third-party application. To inform client applications that they need to recompile to become compatible with the latest release of the third-party application, you can have the clients call a release-oriented compatibility initialization function. This function could take as input a string that tells it which release the client application is working with. If the release string mismatches with the latest version, an error is generated. The client application must then change the release string as part of the changes required to become compatible with the latest release.

For example:

```
FUNCTION compatibility_init(
  rel IN VARCHAR2, errmsg OUT VARCHAR2)
RETURN NUMBER;
```

where:

- `rel` is a release string that is chosen by the product, such as, `'Release 10.1'`
- `errmsg` is any error message that may need to be returned
- The function returns `0` on success and a nonzero value on error

Changing Default Constructors

When a type is altered, its default, system-defined constructors need to be changed in order (for example) to include newly added attributes in the parameter list. If you are using default constructors, you need to modify their invocations in your program in order for the calls to compile.

You can avoid having to modify constructor calls if you define your own constructor functions instead of using the system-defined default ones. See "Advantages of User-Defined Constructors" on page 8-15.

Altering the FINAL Property of a Type

When you alter a type `T1` from `FINAL` to `NOT FINAL`, any attribute of type `T1` in the client program changes from being an inlined structure to a pointer to `T1`. This means that you need to change the program to use dereferencing when this attribute is accessed.

Conversely, when you alter a type from `NOT FINAL` to `FINAL`, the attributes of that type change from being pointers to inlined structures.

For example, say that you have the types `T1(a int)` and `T2(b T1),` where `T1`'s property is `FINAL`. The C/JAVA structure corresponding to `T2` is `T2(T1 b).` But if you change `T1`'s property to `NOT FINAL`, then `T2`'s structure becomes `T2(T1 *b).`

Parallel Queries with Oracle Objects

Oracle lets you perform parallel queries with objects and objects synthesized in views, 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.
- Parallel DML is not supported on views with `INSTEAD-OF` trigger. However, the individual statements within the trigger may be parallelized.

Design Consideration Tips and Techniques

The following sections provide assorted tips on various aspects of working with Oracle object types.

Deciding Whether to Evolve a Type or Create a Subtype Instead

As an application goes through its life cycle, the question often arises whether to change an existing object type or to create a specialized subtype to meet new requirements. The answer depends on the nature of the new requirements and their context in the overall application semantics. Here are two examples:

Changing a Widely Used Base Type

Suppose that we have an object type address with attributes Street, State, and ZIP:

```
CREATE TYPE address AS OBJECT (
  Street  VARCHAR2(80),
  State   VARCHAR2(20),
  ZIP     VARCHAR2(10));
/
```

We later find that we need to extend the address type by adding a Country attribute to support addresses internationally. Is it better to create a subtype of address or to evolve the address type itself?

With a general base type that has been widely used throughout an application, it is better to implement the change using type evolution.

Adding Specialization

Suppose that an existing type hierarchy of Graphic types (for example, curve, circle, square, text) needs to accommodate an additional variation, namely, Bezier curve. To support a new specialization of this sort that does not reflect a shortcoming of the base type, we should use inheritance and create a new subtype BezierCurve under the Curve type.

To sum up, the semantics of the required change dictates whether we should use type evolution or inheritance. For a change that is more general and affects the base type, use type evolution. For a more specialized change, implement the change using inheritance.

How ANYDATA Differs from User-Defined Types

ANYDATA is an Oracle-supplied type that can hold instances of any Oracle data type, whether built-in or user-defined. ANYDATA is a self-describing type and supports a reflection-like API that you can use to determine the shape of an instance.

While both inheritance, through the substitutability feature, and ANYDATA provide the polymorphic ability to store any of a set of possible instances in a placeholder, the two models give the capability two very different forms.

In the inheritance model, the polymorphic set of possible instances must form part of a single type hierarchy. A variable can potentially hold instances only of its defined type or of its subtypes. You can access attributes of the supertype and call methods defined in the supertype (and potentially overridden by the subtype). You can also test the specific type of an instance using the IS OF and the TREAT operators.

ANYDATA variables, however, can store heterogeneous instances. You cannot access attributes or call methods of the actual instance stored in an ANYDATA variable (unless you extract out the instance). You use the ANYDATA methods to discover and extract the type of the instance. ANYDATA is a very useful mechanism for parameter passing when the function/procedure does not care about the specific type of the parameter(s).

Inheritance provides better modeling, strong typing, specialization, and so on. Use ANYDATA when you simply want to be able to hold one of any number of possible instances that do not necessarily have anything in common.

Polymorphic Views: An Alternative to an Object View Hierarchy

Chapter 6, "Applying an Object Model to Relational Data" describes how to build up a view hierarchy from a set of object views each of which contains objects of a single type. Such a view hierarchy enables queries on a view within the hierarchy to see a polymorphic set of objects contained by the queried view or its subviews.

As an alternative way to support such polymorphic queries, you can define an object view based on a query that returns a polymorphic set of objects. This approach is especially useful when you want to define a view over a set of tables or views that already exists.

For example, an object view of Person_t can be defined over a query that returns Person_t instances, including Employee_t instances. The following statement creates a view based on queries that select persons from a persons table and employees from an employees table.

```
CREATE VIEW Persons_view OF Person_t AS
  SELECT Person_t(...) FROM persons
  UNION ALL
  SELECT TREAT(Employee_t(...) AS Person_t) FROM employees;
```

An INSTEAD OF trigger defined for this view can use the VALUE function to access the current object and to take appropriate action based on the object's most specific type.

Polymorphic views and object view hierarchies have these important differences:

- **Addressability**: In a view hierarchy, each subview can be referenced independently in queries and DML statements. Thus, every set of objects of a particular type has a logical name. However, a polymorphic view is a single view, so you must use predicates to obtain the set of objects of a particular type.
- **Evolution**: If a new subtype is added, a subview can be added to a view hierarchy without changing existing view definitions. With a polymorphic view, the single view definition must be modified by adding another UNION branch.
- **DML Statements**: In a view hierarchy, each subview can be either inherently updatable or can have its own INSTEAD OF trigger. With a polymorphic view, only one INSTEAD OF trigger can be defined for a given operation on the view.

The SQLJ Object Type

This section discusses the SQLJ object type.

The Intended Use of SQLJ Object Types

According to the *Information Technology - SQLJ - Part 2* document (SQLJ Standard), a SQLJ object type is a database object type designed for Java. A SQLJ object type maps to a Java class. Once the mapping is registered through the extended SQL CREATE TYPE command (a DDL statement), the Java application can insert or select the Java objects directly into or from the database through an Oracle JDBC driver. This enables the user to deploy the same class in the client, through JDBC, and in the server, through SQL method dispatch.

Actions Performed When Creating a SQLJ Object Type

The extended SQL `CREATE TYPE` command:

- Populates the database catalog with the external names for attributes, functions, and the Java class. Also, dependencies between the Java class and its corresponding SQLJ object type are maintained.
- Validates the existence of the Java class and validates that it implements the interface corresponding to the value of the `USING` clause.
- Validates the existence of the Java fields (as specified in the `EXTERNAL NAME` clause) and whether these fields are compatible with corresponding SQL attributes.
- Generates an internal class to support constructors, external variable names, and external functions that return `self` as a result.

Uses of SQLJ Object Types

The SQLJ object type is a special case of SQL object type in which all methods are implemented in a Java class. The mapping between a Java class and its corresponding SQL type is managed by the SQLJ object type specification. That is, the SQLJ Object type specification cannot have a corresponding type body specification.

Also, the inheritance rules among SQLJ object types specify the legal mapping between a Java class hierarchy and its corresponding SQLJ object type hierarchy. These rules ensure that the SQLJ Type hierarchy contains a valid mapping. That is, the supertype or subtype of a SQLJ object type has to be another SQLJ object type.

Uses of Custom Object Types

The custom object type is the Java interface for accessing SQL object types. A SQL object type may include methods that are implemented in languages such as PLSQL, Java, and C. Methods implemented in Java in a given SQL object type can belong to different unrelated classes. That is, the SQL object type does not map to a specific Java class.

In order for the client to access these objects, JPublisher can be used to generate the corresponding Java class. Furthermore, the user has to augment the generated classes with the code of the corresponding methods. Alternatively, the user can create the class corresponding to the SQL object type.

At runtime, the JDBC user has to register the correspondence between a SQL Type name and its corresponding Java class in a map.

Differences Between SQLJ and Custom Object Types Through JDBC

The following table summarizes the differences between SQLJ object types and custom object types.

Table 9–1 Differences Between SQLJ and Custom Object Types

Feature	SQLJ Object Type Behavior	Custom Object Type Behavior
Typecodes	Use the `OracleTypes.JAVA_STRUCT` typecode to register a SQLJ object type as a SQL `OUT` parameter. The `OracleTypes.JAVA_STRUCT` typecode is also used in the `_SQL_TYPECODE` field of a class implementing the `ORAData` or `SQLData` interface.	Use the `OracleTypes.STRUCT` typecode to register a custom object type as a SQL `OUT` parameter. The `OracleTypes.STRUCT` typecode is also used in the `_SQL_TYPECODE` field of a class implementing the `ORAData` or `SQLData` interface.
Creation	Create a Java class implementing the `SQLData` or `ORAData` and `ORADataFactory` interfaces first and then load the Java class into the database. Next, you issue the extended SQL `CREATE TYPE` command for SQLJ object type.	Issue the extended SQL `CREATE TYPE` command for a custom object type and then create the `SQLData` or `ORAData` Java wrapper class using JPublisher or do this manually.
Method Support	Supports external names, constructor calls, and calls for member functions with side effects.	There is no default class for implementing type methods as Java methods. Some methods may also be implemented in SQL.
Type Mapping	Type mapping is automatically done by the extended SQL `CREATE TYPE` command. However, the SQLJ object type must have a defining Java class on the client.	Register the correspondence between SQL and Java in a type map. Otherwise, the type is materialized as `oracle.sql.STRUCT`.
Inheritance	There are rules for mapping SQL hierarchy to a Java class hierarchy. See the *Oracle Database SQL Language Reference* for a complete description of these rules.	There are no mapping rules.

Miscellaneous Tips

This section discusses miscellaneous tips.

Column Substitutability and the Number of Attributes in a Hierarchy

If a column or table is of type `T`, Oracle adds a hidden column for each attribute of type `T` and, if the column or table is substitutable, for each attribute of every subtype of `T`, to store attribute data. A hidden `typeid` column is added as well, to keep track of the type of the object instance in a row.

The number of columns in a table is limited to 1,000. A type hierarchy with a number of total attributes approaching 1,000 puts you at risk of running up against this limit when using substitutable columns of a type in the hierarchy. To avoid problems as a result of this, consider one of the following options for dealing with a hierarchy that has a large number of total attributes:

- Use views
- Use `REF`s
- Break up the hierarchy

Circular Dependencies Among Types

Avoid creating circular dependencies among types. In other words, do not create situations in which a method of type `T` returns a type `T1`, which has a method that returns a type `T`.

A

Sample Application Using Object-Relational Features

This appendix describes a sample application that provides an overview of how to create and use user-defined data types (Oracle Objects). An application is first developed with the relational model and then with the object-relational model.

This appendix contains the following sections:

Introduction to the Sample Application

User-defined types are schema objects in which users formalize the data structures and operations that appear in their applications.

The examples in this appendix illustrate the most important aspects of defining, using, and evolving object types. One important aspect of working with object 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 object types, such as defining a type, use SQL.

The examples develop different versions of a database schema for an application that manages customer purchase orders. First a purely relational version is shown, and then an equivalent, object-relational version. Both versions provide for the same basic kinds of entities—customers, purchase orders, line items, and so on. But the object-relational version creates object types for these entities and manages data for particular customers and purchase orders by instantiating instances of the respective object types.

PL/SQL and Java provide additional capabilities beyond those illustrated in this appendix, 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 their extensive facilities for accessing objects and collections, and manipulating them on clients.

See Also:

- *Oracle Database SQL Language Reference* for a complete description of SQL syntax and usage for user-defined types
- *Oracle Database PL/SQL User's Guide and Reference* for a complete discussion of PL/SQL capabilities
- *Oracle Database Java Developer's Guide* for a complete discussion of Java
- *Oracle Call Interface Programmer's Guide*
- *Pro*C/C++ Programmer's Guide*

Implementing the Schema on the Relational Model

This section implements the relational version of the purchase order schema depicted in Figure A–1.

Entities and Relationships

The basic entities in this example are:

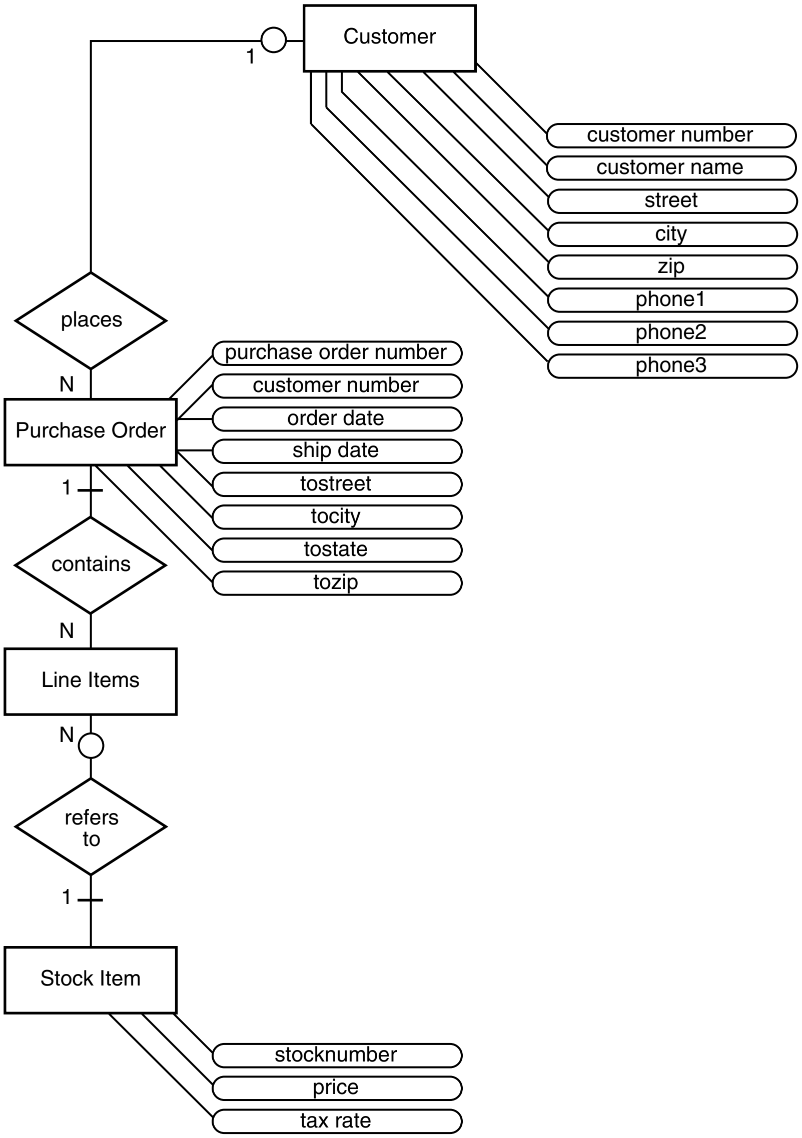
- Customers
- The stock of products for sale
- Purchase orders

As shown in Figure A–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 his 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 he places 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.

Figure A–1 Entity-Relationship Diagram for Purchase Order Application



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 A–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 advantages of object-relational constructs is that you can use names that closely model the corresponding real-world objects.

The relational approach results in the tables described in the following sections.

Customer_reltab

The `Customer_reltab` table has the following definition:

Example A–1 Creating the Customer_reltab Table

```
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 relationship must be managed by the purchase order.

PurchaseOrder_reltab

The `PurchaseOrder_reltab` table has the following definition:

Example A–2 Creating the PurchaseOrder_reltab Table

```
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));
```

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 Customer_reltab. The PurchaseOrder_reltab table contains no information about related line items. The line items table, described in the next section, uses the purchase order number to relate a line item to its parent purchase order.

Stock_reltab

The Stock_reltab table has the following definition:

Example A–3 Creating the Stock_reltab Table

```
CREATE TABLE Stock_reltab (
  StockNo      NUMBER PRIMARY KEY,
  Price        NUMBER,
  TaxRate      NUMBER);
```

LineItems_reltab

The LineItems_reltab table has the following definition:

Example A–4 Creating the LineItems_reltab Table

```
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 and PurchaseOrder_reltab tables 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

Inserting Values Under the Relational Model

In our application, statements like these insert data into the tables:

Example A–5 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);
```

Example A–6 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);
```

Example A–7 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');
```

Example A–8 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:

Example A–9 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;
```

Example A–10 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;
```

Example A–11 Get the Purchase Order and Line Item Data for Stock Item 1004

```
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:

Example A–12 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 similar to Example A–13 to delete data.

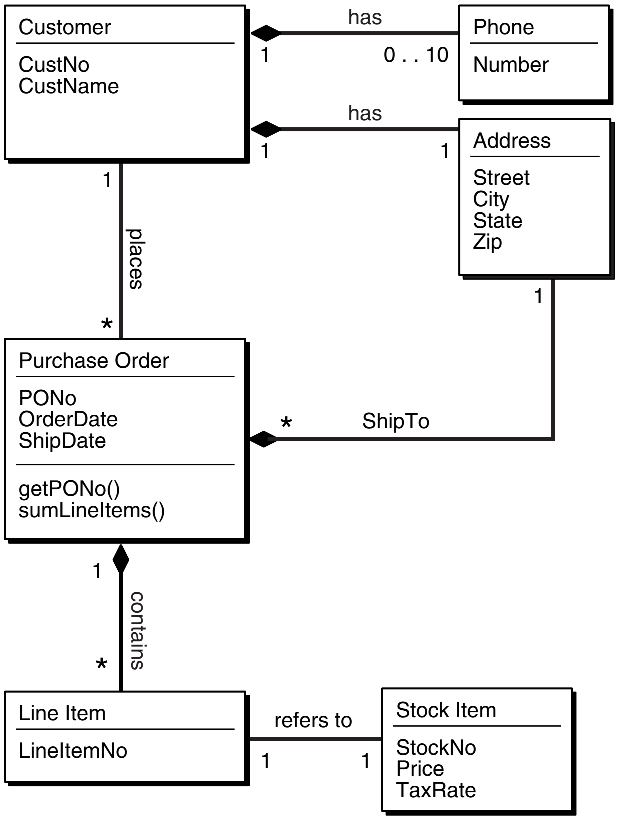
Example A–13 Delete Purchase Order 1001 under the Relational Model

```
DELETE
   FROM   LineItems_reltab
   WHERE  PONo = 1001;

DELETE
   FROM   PurchaseOrder_reltab
   WHERE  PONo = 1001;
```

Implementing the Schema on the Object-Relational Model

The object-relational approach begins with the same entity relationships as in "Entities and Relationships" on page A-2. Viewing these from the object-oriented perspective, as in the following class diagram, allows us to translate more of the real-world structure into the database schema.

Figure A–2 Class Diagram for Purchase Order Application



Instead of breaking up addresses or multiple phone numbers into unrelated columns in relational tables, the object-relational approach defines types to represent an entire address and an entire list of phone numbers. Similarly, the object-relational approach uses nested tables to keep line items with their purchase orders instead of storing them separately.

The main entities—customers, stock, and purchase orders—become object types. Object references are used to express some of the relationships among them. Collection types—varrays and nested tables—are used to model multi-valued attributes.

Note: This appendix implements an object-relational interface by building an object-relational schema from scratch. With this approach, we create object tables for data storage. Alternatively, instead of object tables, you can use object views to implement an object-relational interface to existing data stored in relational tables. Chapter 6 discusses object views.

Defining Types

You create an object type with a `CREATE TYPE` statement. For example, the following statement creates the type `StockItem_objtyp`:

Example A–14 Creating the StockItem_objtyp Object

```
CREATE TYPE StockItem_objtyp AS OBJECT (
  StockNo    NUMBER,
  Price      NUMBER,
  TaxRate    NUMBER
  );
/
```

Instances of type `StockItem_objtyp` are objects representing the stock items that customers order. They have three numeric attributes. `StockNo` is the primary key.

The order in which you define types can make a difference. Ideally, you want to wait to define types that refer to other types until you have defined the other types they refer to.

For example, the type `LineItem_objtyp` refers to, and thus presupposes, `StockItem_objtyp` by containing an attribute that is a `REF` to objects of `StockItem_objtyp`. You can see this in the statement that creates the type `LineItem_objtyp`.

Example A–15 Creating the LineItem_objtyp Object

```
CREATE TYPE LineItem_objtyp AS OBJECT (
  LineItemNo   NUMBER,
  Stock_ref    REF StockItem_objtyp,
  Quantity     NUMBER,
  Discount     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.

Sometimes the web of references among types makes it difficult or impossible to avoid creating a type before all the types that it presupposes are created. To deal with this sort of situation, you can create what is called an incomplete type to use as a placeholder for other types that you want to create to refer to. Then, when you have created the other types, you can come back and replace the incomplete type with a complete one.

For example, if we had needed to create `LineItem_objtyp` before we created `StockItem_objtyp`, we could have used a statement like the following to create `LineItem_objtyp` as an incomplete type:

`CREATE TYPE LineItem_objtyp;`

The form of the `CREATE TYPE` statement used to create an incomplete type does not have the phrase `AS OBJECT` or the specification of attributes.

To replace an incomplete type with a complete definition, include the phrase `OR REPLACE` as shown in the following example:

Example A–16 Replacing the LineItem_objtyp Object

```
CREATE OR REPLACE TYPE LineItem_objtyp AS OBJECT (
  LineItemNo   NUMBER,
  Stock_ref    REF StockItem_objtyp,
  Quantity     NUMBER,
  Discount     NUMBER
  );
/
```

It is never wrong to include the words `OR REPLACE`, even if you have no incomplete type to replace.

Now create the remaining types we need for the schema. The following statement defines an array type for the list of phone numbers:

Example A–17 Creating the PhoneList_vartyp Type

```
CREATE TYPE PhoneList_vartyp AS VARRAY(10) OF VARCHAR2(20);
/
```

Any data unit, or instance, of type `PhoneList_vartyp` is a varray of up to 10 telephone numbers, each represented by a data item of type `VARCHAR2`.

Either a varray or a nested table could be used to contain a list of phone numbers. 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.
- You can query a nested table but not a varray. But there is no reason to query the phone number list, so using a nested table 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 9, "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:

Example A–18 Creating the Address_objtyp Object

```
CREATE TYPE Address_objtyp AS OBJECT (
  Street         VARCHAR2(200),
  City           VARCHAR2(200),
  State          CHAR(2),
  Zip            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 object types as building blocks.

Example A–19 Creating the Customer_objtyp Object

```
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
) NOT FINAL;
/
```

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.

The clause NOT FINAL enables us to create subtypes of the customer type later if we wish. By default, types are created as FINAL, which means that the type cannot be further specialized by deriving subtypes from it. We define a subtype of Customer_objtyp for a more specialized kind of customer later in this appendix.

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 implicitly invokes the compareCustOrders method to do so.

Note: The PL/SQL to implement the comparison method appears in "The compareCustOrders Method" on page A-14.

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 for each 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 2, "Basic Components of Oracle Objects" for more information about map and order methods
- *Oracle Database PL/SQL User's Guide and Reference* for details about how to use pragma declarations

The following statement defines a type for a nested table of line items. Each purchase order will use an instance of this nested table type to contain the line items for that purchase order:

Example A–20 Creating the LineItemList_ntabtyp Type

```
CREATE TYPE LineItemList_ntabtyp AS TABLE OF LineItem_objtyp;
/
```

An instance of this type is a nested table object (in other words, 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 can be done using SQL if the line items are stored in a nested table but not if they are stored in a varray.
- If an application needs to index on line item data, this can be done with nested tables but not with varrays.

- The order in which line items are stored is probably not important, and a query can order them by line item number 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 defines the object type PurchaseOrder_objtyp:

Example A–21 Creating the PurchaseOrder_objtyp Object

```
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
  );
/
```

Instances of type PurchaseOrder_objtyp 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. PONo is the primary key and Cust_ref is a foreign key.

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. Those appear in "Method Definitions" on page A-12.

Method Definitions

If a type has no methods, its definition consists just of a CREATE TYPE statement. However, for a type that has methods, you must also define a type body to complete the definition of the type. You do this with a CREATE TYPE BODY statement. As with CREATE TYPE, you can include the words OR REPLACE. You must include this phrase if you are replacing an existing type body with a new one, to change the methods.

The following statement defines the body of the type PurchaseOrder_objtyp. The statement supplies the PL/SQL programs that implement the type's methods:

Example A–22 Creating the PurchaseOrder_objtyp Type Body

```
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 simply returns the value of the PONo attribute—namely, the purchase order number—of whatever instance of the type PurchaseOrder_objtyp that calls the method. 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 values of the line items. The method returns the total.

The compareCustOrders Method

The following statement defines the compareCustOrders method in the type body of the Customer_objtyp object type:

Example A–23 Creating the Customer_objtyp Type Body

```
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 both orders are associated with the same customer.

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 the explicit argument) to an ORDER method is NULL, Oracle does not call the ORDER method and simply treats the result as NULL.

We have now defined all of the object types for the object-relational version of the purchase order schema. We have not yet created any instances of these types to contain actual purchase order data, nor have we created any tables in which to store such data. We show how to do this in the next section.

Creating Object Tables

Creating an object type is not the same as creating a table. Creating a type merely defines a logical structure; it does not create storage. To use an object-relational interface to your data, you must create object types whether you intend to store your data in object tables or leave it in relational tables and access it through object views. Object views and object tables alike presuppose object types: an object table or object view is always a table or view of a certain object type. In this respect, it is like a relational column, which always has a specified data type.

> **See Also:** Chapter 6, "Applying an Object Model to Relational Data" for a discussion of object views

Like a relational column, an object table can contain rows of just one kind of thing, namely, object instances of the same declared type as the table. (And, if the table is substitutable, it can contain instances of subtypes of its declared type as well.)

Each row in an object table is a single object instance. So, in one sense, an object table has, or consists of, only a single column of the declared object type. But this is not as different as it may seem from the case with relational tables. Each row in a relational table theoretically represents a single entity as well—for example, a customer, in a relational `Customers` table. The columns of a relational table store data for attributes of this entity.

Similarly, in an object table, attributes of the object type map to columns that can be inserted into and selected from. The major difference is that, in an object table, data is stored—and can be retrieved—in the structure defined by the table's type, making it possible for you to retrieve an entire, multilevel structure of data with a very simple query.

The Object Table Customer_objtab

The following statement defines an object table `Customer_objtab` to hold objects of type `Customer_objtyp`:

Example A–24 Creating the Customer_objtab Table

```
CREATE TABLE Customer_objtab OF Customer_objtyp (CustNo PRIMARY KEY)
   OBJECT IDENTIFIER IS PRIMARY KEY;
```

Unlike with relational tables, when you create an object table, you specify a data type for it, namely, the type of objects it will contain.

The table has a column for each attribute of `Customer_objtyp`, namely:

```
CustNo          NUMBER  /* Primary key */
CustName        VARCHAR2(200)
Address_obj     Address_objtyp
PhoneList_var   PhoneList_vartyp
```

See Example A–18, "Creating the Address_objtyp Object" on page A-10 and Example A–17, "Creating the PhoneList_vartyp Type" on page A-10 for the definitions of those types.

Figure A–3 Object Relational Representation of Table Customer_objtab

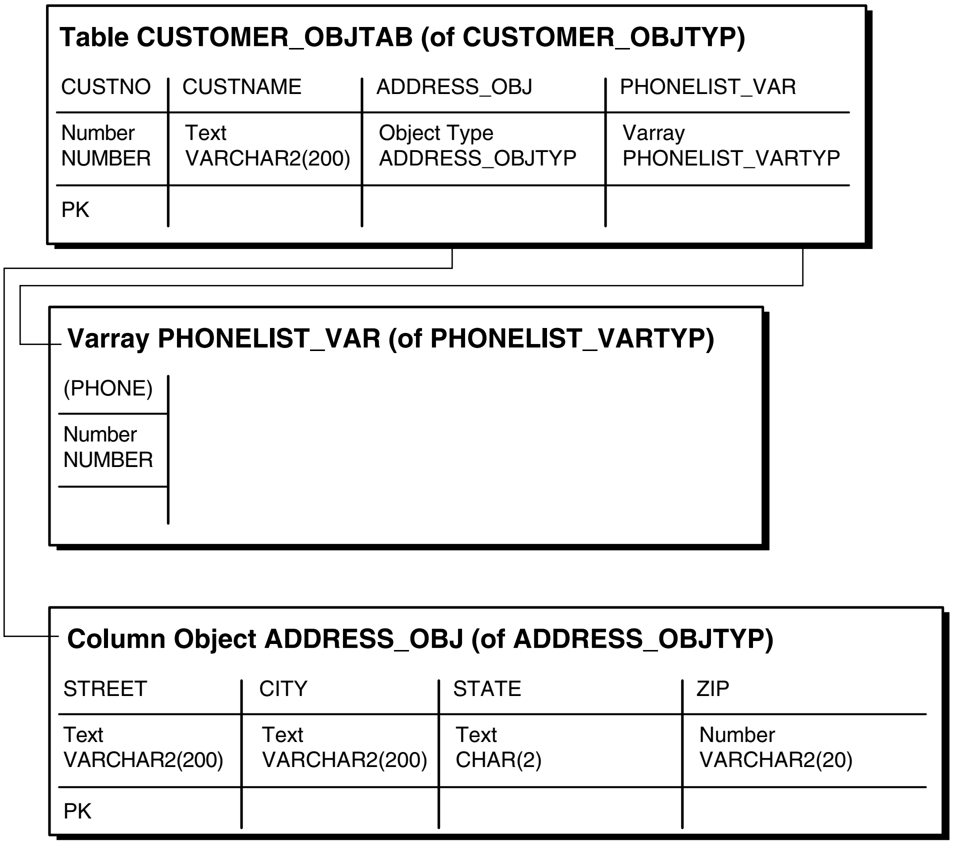


Table CUSTOMER_OBJTAB (of CUSTOMER_OBJTYP)

CUSTNO	CUSTNAME	ADDRESS_OBJ	PHONELIST_VAR
Number NUMBER	Text VARCHAR2(200)	Object Type ADDRESS_OBJTYP	Varray PHONELIST_VARTYP
PK			

Varray PHONELIST_VAR (of PHONELIST_VARTYP)

(PHONE)
Number NUMBER

Column Object ADDRESS_OBJ (of ADDRESS_OBJTYP)

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

Object Data Types 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`.

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 a system-generated value represented by the type `REF` and is based on 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 IDENTIFIER IS PRIMARY KEY` or `OBJECT IDENTIFIER IS SYSTEM GENERATED`. The latter is the default. Using the primary key as the object identifier can be more efficient in cases where the primary key value is smaller than the default 16 byte system-generated identifier. For our example, the primary key is used as the row object identifier.

Object Tables with Embedded Objects

Note that the Address_obj column of Customer_objtab contains Address_objtyp objects. As this shows, an object type may have attributes that are themselves object types. Object instances of the declared type of an object table are called row objects because one object instance occupies an entire row of the table. But embedded objects such as those in the Address_obj column are referred to as column objects. These differ from row objects in that they do not take up an entire row. Consequently, they are not referenceable—they cannot be the target of a REF. Also, they can be NULL.

The attributes of Address_objtyp objects are of built-in types. They are scalar rather than complex (that is, they are not object types with attributes of their own), and so are called leaf-level attributes to reflect that they represent an end to branching. Columns for Address_objtyp objects and their attributes are created in the object table Customer_objtab. You can refer or navigate to these columns using the dot notation. For example, if you want to build an index on the Zip column, you can refer to it as Address.Zip.

The PhoneList_var 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. See Example A–17 on page A-10.

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 do not 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 following statement creates an object table for StockItem_objtyp objects:

Example A–25 Creating the Stock_objtab Table

```
CREATE TABLE Stock_objtab OF StockItem_objtyp (StockNo PRIMARY KEY)
   OBJECT IDENTIFIER IS PRIMARY KEY;
```

Each row of the table is a StockItem_objtyp object having three numeric attributes:

```
StockNo     NUMBER
Price       NUMBER
TaxRate     NUMBER
```

Oracle creates a column for each attribute. 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 following statement defines an object table for PurchaseOrder_objtyp objects:

Example A–26 Creating the PurchaseOrder_objtab Table

```
CREATE TABLE PurchaseOrder_objtab OF PurchaseOrder_objtyp (  /* Line 1 */
   PRIMARY KEY (PONo),                                       /* Line 2 */
   FOREIGN KEY (Cust_ref) REFERENCES Customer_objtab)        /* Line 3 */
   OBJECT IDENTIFIER IS 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
```

See Example A–19, "Creating the Customer_objtyp Object" on page A-10 and Example A–20, "Creating the LineItemList_ntabtyp Type" on page A-11 for the definitions of those types.

Figure A–4 Object Relational Representation of Table PurchaseOrder_objtab

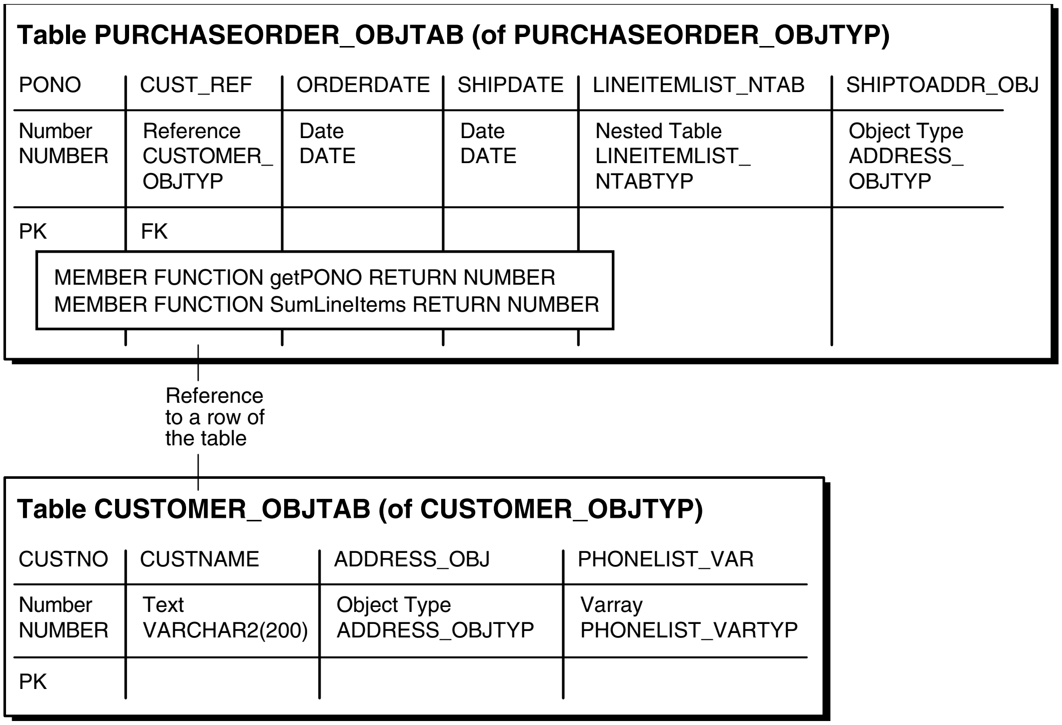


Table PURCHASEORDER_OBJTAB (of PURCHASEORDER_OBJTYP)

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				

MEMBER FUNCTION getPONO RETURN NUMBER
MEMBER FUNCTION SumLineItems RETURN NUMBER

Table CUSTOMER_OBJTAB (of CUSTOMER_OBJTYP)

CUSTNO	CUSTNAME	ADDRESS_OBJ	PHONELIST_VAR
Number NUMBER	Text VARCHAR2(200)	Object Type ADDRESS_OBJTYP	Varray PHONELIST_VARTYP
PK			

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 permits 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 IDENTIFIER IS 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 LineItemList_ntab STORE AS PoLine_ntab (
     (PRIMARY KEY(NESTED_TABLE_ID, LineItemNo))
     ORGANIZATION INDEX COMPRESS)
   RETURN AS LOCATOR
```

These lines pertain to the storage specification and properties of the nested table column, LineItemList_ntab. The rows of a nested table are stored in a separate storage table. This storage table cannot be directly queried 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 preceding CREATE TABLE example, Line 5 indicates that the rows of LineItemList_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 permits 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 parent row object.

> **See Also:** "Nested Table Storage" on page 9-10 for information about the benefits of organizing a nested table as an IOT, specifying nested table compression, and for more information about nested table storage in general

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 specifies the key for the IOT; second, it enforces uniqueness of the column LineItemNo of the 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 causes the entire nested table to be returned instead of just a locator to it. If a nested table collection contains many elements, it is inefficient 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 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 A-12. 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:

Example A–27 Replacing the PurchaseOrder_objtyp Type Body

```
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. If 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 tell the SQL compilation engine the actual type of the collection attribute (or parameter or variable) so that it can compile the query.

The querying of the nested table locator results in 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.

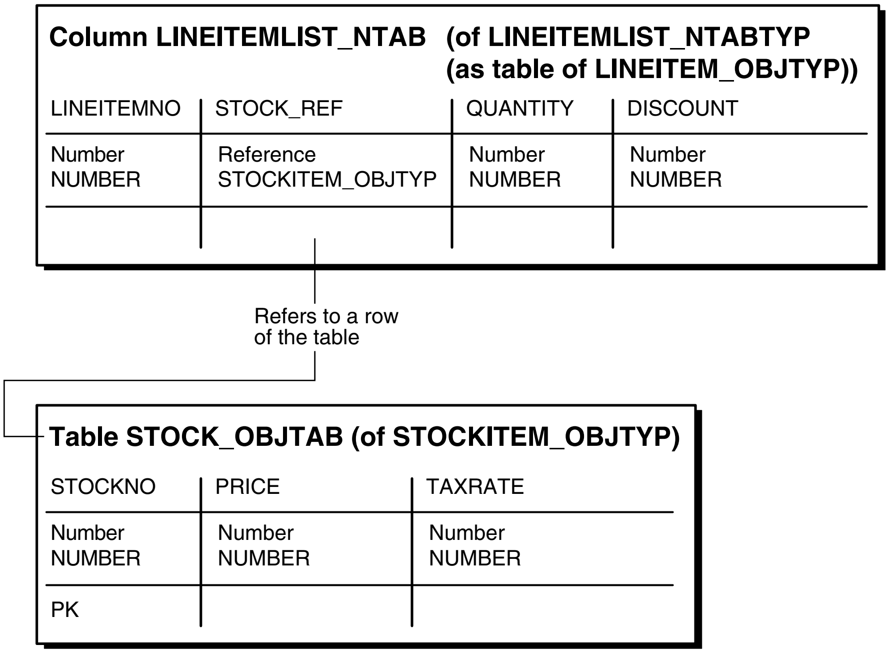
After the table is created, the ALTER TABLE statement is issued to add the SCOPE FOR constraint on a REF. The SCOPE FOR constraint on a REF is not allowed in a CREATE TABLE statement. To specify that Stock_ref can reference only the object table Stock_objtab, issue the following ALTER TABLE statement on the PoLine_ntab storage table:

Example A–28 Adding the SCOPE FOR Constraint

```
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 in that the SCOPE constraint has no dependency 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 A–5 Object Relational Representation of Nested Table LineItemList_ntab



Column LINEITEMLIST_NTAB (of LINEITEMLIST_NTABTYP (as table of LINEITEM_OBJTYP))

LINEITEMNO	STOCK_REF	QUANTITY	DISCOUNT
Number NUMBER	Reference STOCKITEM_OBJTYP	Number NUMBER	Number NUMBER

Table STOCK_OBJTAB (of STOCKITEM_OBJTYP)

STOCKNO	PRICE	TAXRATE
Number NUMBER	Number NUMBER	Number NUMBER
PK		

Oracle does not support a 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 several 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 A–6 Object Relational Representation of Table PurchaseOrder_objtab

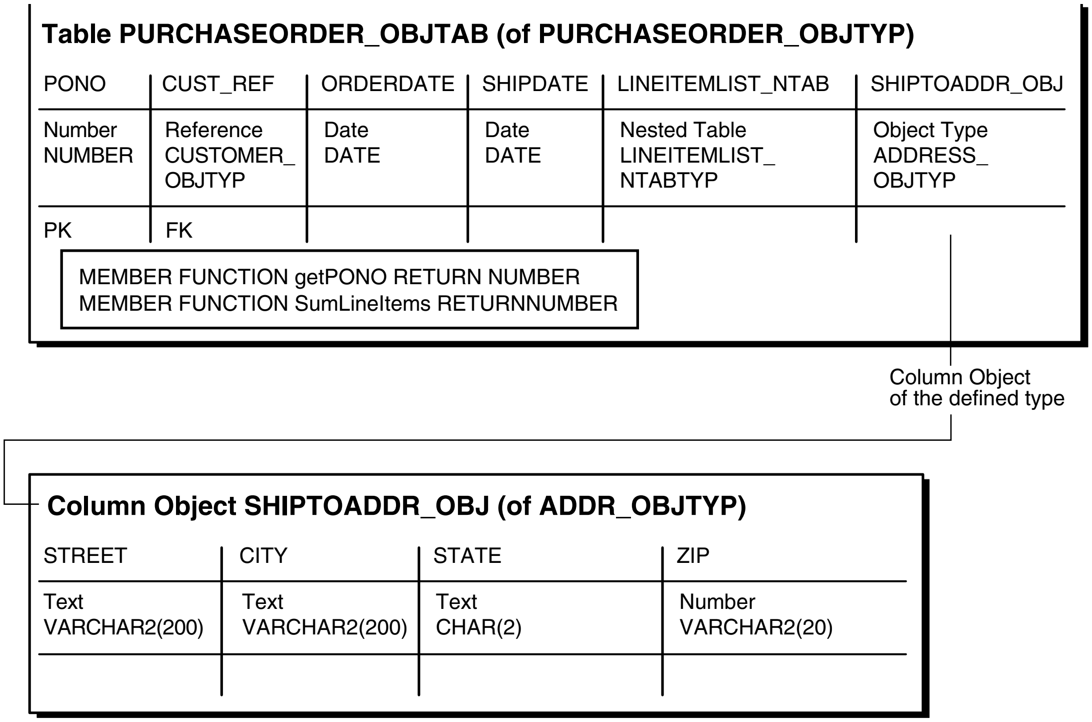


Table PURCHASEORDER_OBJTAB (of PURCHASEORDER_OBJTYP)

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				

MEMBER FUNCTION getPONO RETURN NUMBER
MEMBER FUNCTION SumLineItems RETURNNUMBER

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 that we inserted earlier into relational tables. Notice how some of the values incorporate calls to the constructors for object types, to create instances of the types.

Example A–29 Inserting Values in 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) ;
```

Example A–30 Inserting Values in 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')
    ) ;
```

Example A–31 Inserting Values in 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.

Example A–32 Inserting Values in LineItemList_ntab

```
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:

Example A–33 Inserting Values in PurchaseOrder_objtab and LineItemList_ntab

```
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:

Example A–34 Query Purchase Orders

```
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.

Example A–35 Query 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 ;
```

Example A–36 Query Total Value of Each Purchase Order

```
SELECT   p.PONo, p.sumLineItems()
 FROM    PurchaseOrder_objtab p ;
```

Example A–37 Query Purchase Order and Line Item Data for 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 preceding 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 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 preceding 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 preceding unnesting example, only the rows of the `PurchaseOrder_objtab` object table that have 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 ;
```

In Example A–38, the request requires querying the rows of all `LineItemList_ntab` nested tables of all `PurchaseOrder_objtab` rows. Again, unnesting is required:

Example A–38 Query Average Discount across all Line Items of all Purchase Orders

```
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 shown in Example A–13 on page A-4. In Example A–39, Oracle deletes the entire purchase order object, including its line items, in a single SQL operation. In the relational case, line items for the purchase order must be deleted from the line items table, and the purchase order must be separately deleted from the purchase orders table.

> **Note:** I f you are performing the SQL statements in this sample, do not execute the DELETE statement in Example A–39 because the purchase order is needed in the following examples.

Example A–39 Delete Purchase Order 1001 in an Object-Relational Model

```
DELETE
 FROM   PurchaseOrder_objtab
 WHERE  PONo = 1001 ;
```

Evolving Object Types

Even a completed, fully built application tends to be a work in progress. Sometimes requirements change, forcing a change to an underlying object model or schema to adapt it to new circumstances, and sometimes there are ways to improve an object model so that it does a better job of what it was originally intended to do.

Suppose that, after living with our object-relational application for a while, we discover some ways that we could improve the design. In particular, suppose that we discover that users almost always want to see a history of purchases when they bring up the record for a customer. To do this with the present object model requires a join on the two tables `Customer_objtab` and `PurchaseOrder_objtab` that hold information about customers and purchase orders. We decide that a better design would be to provide access to data about related purchase orders directly from the customers table.

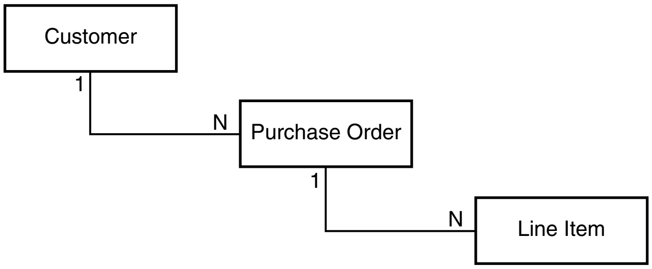
One way to do this is to change the `Customer_objtyp` so that information about a customer's purchase orders is included right in the object instance that represents that customer. In other words, we want to add an attribute for purchase order information to `Customer_objtyp`. To hold information about multiple purchase orders, the attribute must be a collection type—a nested table.

Adding an attribute is one of several ways that you can alter, or evolve, an object type. When you evolve a type, Oracle applies your changes to the type itself and to all its dependent schema objects, including subtypes of the type, other object types that have the altered type as an attribute, and tables and columns of the altered type.

To change `Customer_objtyp` to add an attribute for a nested table of purchase orders, several steps are needed:

1. Create a new type for a nested table of purchase orders
2. Alter `Customer_objtyp` to add a new attribute of the new type
3. In the `Customer_objtab` object table, name and scope the storage tables for the newly added nested tables
   - Upgrading the `Customer_objtab` object table for the new attribute actually adds two levels of nested tables, one inside the other, because a purchase order itself contains a nested table of line items.
   - Both the purchase orders nested table and the line items nested table need to be scoped so that they can contain primary key-based `REF`s. More on this in the next section.

Figure A–7 Nested Tables in the Customer Object Type



When we are done with the preceding steps, information about customers and purchase orders will be more logically related in our model, and we will be able to query the customers table for all information about customers, purchase orders, and line items. We will also be able to insert a new purchase order for a new customer with a single INSERT statement on the customers table.

Adding an Attribute to the Customer Type

Before we can add a nested table of purchase orders as an attribute of Customer_objtyp, we need to define a type for this sort of nested table. The following statement does this:

Example A–40 Create PurchaseOrderList_ntabtyp

```
CREATE TYPE PurchaseOrderList_ntabtyp AS TABLE OF PurchaseOrder_objtyp;
/
```

Now we can use an ALTER TYPE statement to add an attribute of this type to Customer_objtyp:

Example A–41 Alter Customer_objtyp

```
ALTER TYPE Customer_objtyp
  ADD ATTRIBUTE (PurchaseOrderList_ntab PurchaseOrderList_ntabtyp)
  CASCADE;
```

If a type being altered has dependent types or tables, an ALTER TYPE statement on the type needs to specify either CASCADE or INVALIDATE to say how to apply the change to the dependents.

- CASCADE performs validation checks on the dependents before applying a type change. These checks confirm that the change does not entail doing something illegal, such as dropping an attribute that is being used as a partitioning key of a table. If a dependent fails validation, the type change aborts. On the other hand, if all dependents validate successfully, the system goes ahead with whatever changes to metadata and data are required to propagate the change to the type. These can include automatically adding and dropping columns, creating storage tables for nested tables, and so forth.
- The INVALIDATE option skips the preliminary validation checks and directly applies the type change to dependents. These are then validated the next time that they are accessed. Altering a type this way saves the time required to do the validations, but if a dependent table cannot be validated later when someone tries to access it, its data cannot be accessed until the table is made to pass the validation.

We need to add scope for a REF column in each of the new nested tables of purchase orders and line items that are added to the Customer_objtab table. For convenience, first we rename the new tables from system-generated names to recognizable names. Then, using the names we have given them, we can alter the storage tables to add scope for their REF columns.

The reason we must do all this is that, in order for a column to store REFs to objects in a table that bases its object identifiers on the primary key, the column must be scoped to that table or have a referential constraint placed on it. Scoping a column to a particular table declares that all REFs in the column are REFs to objects in that table. This declaration is necessary because a primary key-based object identifier is guaranteed unique only in the context of the particular table: it may not be unique across all tables. If you try to insert a primary key-based REF, or user-defined REF, into an unscoped column, you will get an error similar to:

```
cannot INSERT object view REF or user-defined REF
```

Line items contain a REF to objects in table Stock_objtab, whose object identifier uses the table's primary key. This is why we had to add scope for the REF column in the storage table for the line items nested table in table PurchaseOrder_objtab after we created that table. Now we have to do it again for the new nested table of line items in table Customer_objtab.

We have to do the same again for the new nested table of purchase orders we are adding in table Customer_objtab: a purchase order references a customer in the table Customer_objtab, and object identifiers in this table are primary-key based as well.

Using the following statement, we determine the names of the system-generated tables so they can be renamed:

```
SELECT table_name, parent_table_name, parent_table_column FROM user_nested_tables;
```

The output is similar to the following:

```
TABLE_NAME                     PARENT_TABLE_NAME              PARENT_TABLE_COLUMN
------------------------------ ------------------------------ ----------------------
SYSNTQOFArJyBTHu6iOMMKU4wHw== CUSTOMER_OBJTAB                PURCHASEORDERLIST_NTAB
POLINE_NTAB                    PURCHASEORDER_OBJTAB           LINEITEMLIST_NTAB
SYSNTZqu6IQItR++UAtgz1rMB8A== SYSNTQOFArJyBTHu6iOMMKU4wHw== LINEITEMLIST_NTAB
```

For convenience, rename the system-generated nested tables to appropriate names. For example, using the system-generated names in the previous sample output:

```
ALTER TABLE "SYSNTQOFArJyBTHu6iOMMKU4wHw==" RENAME TO PO_List_nt;
ALTER TABLE "SYSNTZqu6IQItR++UAtgz1rMB8A==" RENAME TO Items_List_nt;
```

The process of renaming the system-generated nested tables can also be done automatically with the following PL/SQL procedure:

```
DECLARE
  nested_table_1 VARCHAR2(30);
  nested_table_2 VARCHAR2(30);
  cust_obj_table VARCHAR2(30) := 'CUSTOMER_OBJTAB';
BEGIN
 EXECUTE IMMEDIATE ' SELECT table_name FROM user_nested_tables
    WHERE parent_table_name = :1 ' INTO nested_table_1 USING cust_obj_table;
 EXECUTE IMMEDIATE ' SELECT table_name FROM user_nested_tables
    WHERE parent_table_name = :1 ' INTO nested_table_2 USING nested_table_1;
 EXECUTE IMMEDIATE 'ALTER table "'|| nested_table_1 ||'" RENAME TO PO_List_nt';
 EXECUTE IMMEDIATE 'ALTER table "'|| nested_table_2 ||'" RENAME TO Items_List_nt';
END;
/
```

The new storage tables are named PO_List_nt and Items_List_nt. The following statements scope the REF columns in these tables to specific tables:

Example A–42 Add SCOPE for REF to Nested Tables

```
ALTER TABLE PO_List_nt ADD (SCOPE FOR (Cust_Ref) IS Customer_objtab);
ALTER TABLE Items_List_nt ADD (SCOPE FOR (Stock_ref) IS Stock_objtab);
```

There is just one more thing to do before inserting purchase orders for customers in Customer_objtab. An actual nested table of PurchaseOrderList_ntabtyp must be instantiated for each customer in the table.

When a column is added to a table for a new attribute, column values for existing rows are initialized to NULL. This means that each existing customer's nested table of purchase orders is atomically NULL—there is no actual nested table there, not even an empty one. Until we instantiate a nested table for each customer, attempts to insert purchase orders will get an error similar to:

reference to NULL table value

The following statement prepares the column to hold purchase orders by updating each row to contain an actual nested table instance:

Example A–43 Update Customer_objtab

```
UPDATE Customer_objtab c
  SET c.PurchaseOrderList_ntab = PurchaseOrderList_ntabtyp();
```

In the preceding statement, PurchaseOrderList_ntabtyp() is a call to the nested table type's constructor method. This call, with no purchase orders specified, creates an empty nested table.

Working with Multilevel Collections

At this point, we have evolved the type Customer_objtyp to add a nested table of purchase orders, and we have set up the table Customer_objtab so that it is ready to store purchase orders in the nested table. Now we are ready to insert purchase orders into Customer_objtab.

There are two purchase orders already in table PurchaseOrder_objtab. The following two statements copy these into Customer_objtab:

Example A–44 Insert Purchase Orders into Customer_objtab

```
INSERT INTO TABLE (
  SELECT   c.PurchaseOrderList_ntab
    FROM   Customer_objtab c
    WHERE  c.CustNo = 1
  )
  SELECT VALUE(p)
    FROM PurchaseOrder_objtab p
    WHERE p.Cust_Ref.CustNo = 1;

INSERT INTO TABLE (
  SELECT   c.PurchaseOrderList_ntab
    FROM   Customer_objtab c
    WHERE  c.CustNo = 2
  )
  SELECT VALUE(p)
    FROM PurchaseOrder_objtab p
    WHERE p.Cust_Ref.CustNo = 2;
```

Inserting into Nested Tables

Each of the preceding INSERT statements has two main parts: a TABLE expression that specifies the target table of the insert operation, and a SELECT that gets the data to be inserted. The WHERE clause in each part picks out the customer object to receive the purchase orders (in the TABLE expression) and the customer whose purchase orders are to be selected (in the subquery that gets the purchase orders).

The WHERE clause in the subquery uses dot notation to navigate to the CustNo attribute: p.Cust_Ref.CustNo. Note that a table alias p is required whenever you use dot notation. To omit it and say instead Cust_Ref.CustNo would produce an error.

Another thing to note about the dot notation in this WHERE clause is that we are able to navigate to the CustNo attribute of a customer right through the Cust_Ref REF attribute of a purchase order. SQL (though not PL/SQL) implicitly dereferences a REF used with the dot notation in this way.

The TABLE expression in the first part of the INSERT statement tells the system to treat the collection returned by the expression as a table. The expression is used here to select the nested table of purchase orders for a particular customer as the target of the insert.

In the second part of the INSERT statement, the VALUE() function returns selected rows as objects. In this case, each row is a purchase order object, complete with its own collection of line items. Purchase order rows are selected from one table of type PurchaseOrder_objtyp for insertion into another table of that type.

The preceding INSERT statements use the customer-reference attribute of PurchaseOrder_objtyp to identify the customer to whom each of the existing purchase orders belongs. However, now that all the old purchase orders are copied from the purchase orders table into the upgraded Customer_objtab, this customer-reference attribute of a purchase order is obsolete. Now purchase orders are stored right in the customer object itself.

The following ALTER TYPE statement evolves PurchaseOrder_objtyp to drop the customer-reference attribute. The statement also drops the ShipToAddr_obj attribute as redundant, assuming that the shipping address is always the same as the customer address.

Example A–45 Alter PurchaseOrder_objtyp

```
ALTER TYPE PurchaseOrder_objtyp
    DROP ATTRIBUTE Cust_ref,
    DROP ATTRIBUTE ShipToAddr_obj
    CASCADE;
```

This time we were able to use the CASCADE option to let the system perform validations and make all necessary changes to dependent types and tables.

Inserting a New Purchase Order with Line Items

The previous INSERT example showed how to use the VALUE() function to select and insert into the nested table of purchase orders an existing purchase order object complete with its own nested table of line items. The following example shows how to insert a new purchase order that has not already been instantiated as a purchase order object. In this case, the purchase order's nested table of line items must be instantiated, as well as each line item object with its data. Line numbers are shown on the left for reference.

Example A–46 Insert into LineItemList_ntabtyp with VALUE()

```
INSERT INTO TABLE (                                              /* Line 1  */
  SELECT c.PurchaseOrderList_ntab                                /* Line 2  */
    FROM Customer_objtab c                                       /* Line 3  */
    WHERE c.CustName = 'John Nike'                               /* Line 4  */
   )                                                             /* Line 5  */
  VALUES (1020, SYSDATE, SYSDATE + 1,                            /* Line 6  */
    LineItemList_ntabtyp(                                        /* Line 7  */
      LineItem_objtyp(1, MAKE_REF(Stock_objtab, 1004), 1, 0),    /* Line 8  */
      LineItem_objtyp(2, MAKE_REF(Stock_objtab, 1011), 3, 5),    /* Line 9  */
      LineItem_objtyp(3, MAKE_REF(Stock_objtab, 1535), 2, 10)    /* Line 10 */
      )                                                          /* Line 11 */
);                                                          /* Line 12 */
```

Lines 1-5 use a `TABLE` expression to select the nested table to insert into—namely, the nested table of purchase orders for customer John Nike.

The `VALUES` clause (lines 6-12) contains a value for each attribute of the new purchase order, namely:

```
PONo
OrderDate
ShipDate
LineItemList_ntab
```

Line 6 of the `INSERT` statement specifies values for the three purchase order attributes `PONo`, `OrderDate`, and `ShipDate`.

Only attribute values are given; no purchase order constructor is specified. You do not need to explicitly specify a purchase order constructor to instantiate a purchase order instance in the nested table because the nested table is declared to be a nested table of purchase orders. If you omit a purchase order constructor, the system instantiates a purchase order automatically. You can, however, specify the constructor if you want to, in which case the `VALUES` clause will look like this:

```
INSERT INTO TABLE (
  SELECT c.PurchaseOrderList_ntab
    FROM Customer_objtab c
    WHERE c.CustName = 'John Nike'
   )
VALUES (
  PurchaseOrder_objtyp(1025, SYSDATE, SYSDATE + 1,
    LineItemList_ntabtyp(
      LineItem_objtyp(1, MAKE_REF(Stock_objtab, 1004), 1, 0),
      LineItem_objtyp(2, MAKE_REF(Stock_objtab, 1011), 3, 5),
      LineItem_objtyp(3, MAKE_REF(Stock_objtab, 1535), 2, 10)
     )
  )
)
```

Lines 7-11 instantiate and supply data for a nested table of line items. The constructor method `LineItemList_ntabtyp(...)` creates an instance of such a nested table that contains three line items.

The line item constructor `LineItem_objtyp()` creates an object instance for each line item. Values for line item attributes are supplied as arguments to the constructor.

The `MAKE_REF` function creates a `REF` for the `Stock_ref` attribute of a line item. The arguments to `MAKE_REF` are the name of the stock table and the primary key value of the stock item there that we want to reference. We can use `MAKE_REF` here because object identifiers in the stock table are based on the primary key: if they were not, we would have to use the `REF` function in a subquery to get a `REF` to a row in the stock table.

Querying Multilevel Nested Tables

You can query a top-level nested table column by naming it in the `SELECT` list like any other top-level (as opposed to embedded) column or attribute, but the result is not very readable. For instance, the following query selects the nested table of purchase orders for John Nike:

Example A–47 Query Customer_objtab for Customer John Nike

```
SELECT c.PurchaseOrderList_ntab
   FROM Customer_objtab c
   WHERE CustName = 'John Nike';
```

The query produces a result similar to the following:

```
PURCHASEORDERLIST_NTAB(PONO, ORDERDATE, SHIPDATE, LINEITEMLIST_NTAB(LINEITEMNO,
--------------------------------------------------------------------------------
PURCHASEORDERLIST_NTABTYP(PURCHASEORDER_OBJTYP(2001, '25-SEP-01', '20-MAY-97', L
INEITEMLIST_NTABTYP(LINEITEM_OBJTYP(10, 00004A038A00468ED552CE6A5803ACE034080020
B8C83400000014260100010001002900000000000090600812A00078401FE0000000B03C20B050000
...
```

For humans, at least, you probably want to display the instance data in an unnested form and not to show the REFs at all. TABLE expressions—this time in the FROM clause of a query—can help you do this.

For example, the query in Example A–48 selects the PO number, order date, and shipdate for all purchase orders belonging to John Nike:

Example A–48 Query Customer_objtab Using TABLE Expression

```
SELECT p.PONo, p.OrderDate, p.Shipdate
    FROM Customer_objtab c, TABLE(c.PurchaseOrderList_ntab) p
    WHERE c.CustName = 'John Nike';

PONO    ORDERDATE SHIPDATE
------- --------- ---------
2001    25-SEP-01 26-SEP-01
1020    25-SEP-01 26-SEP-01
```

A TABLE expression takes a collection as an argument and can be used like a SQL table in SQL statements. In the preceding query, listing the nested table of purchase orders in a TABLE expression in the FROM clause enables us to select columns of the nested table just as if they were columns of an ordinary table. The columns are identified as belonging to the nested table by the table alias they use: p. As the example shows, a TABLE expression in the FROM clause can have its own table alias.

Inside the TABLE expression, the nested table is identified as a column of customer table Customer_objtab by the customer table's own table alias c. Note that the table Customer_objtab appears in the FROM clause before the TABLE expression that refers to it. This ability of a TABLE expressions to make use of a table alias that occurs to the left of it in the FROM clause is called left correlation. It enables you to daisy-chain tables and TABLE expressions—including TABLE expressions that make use of the table alias of another TABLE expression. In fact, this is how you are able to select columns of nested tables that are embedded in other nested tables.

Here, for example, is a query that selects information about all line items for PO number 1020:

Example A–49 Query Customer_objtab for Purchase Order 1020

```
SELECT p.PONo, i.LineItemNo, i.Stock_ref.StockNo, i.Quantity, i.Discount
  FROM Customer_objtab c, TABLE(c.PurchaseOrderList_ntab) p,
    TABLE(p.LineItemList_ntab) i
  WHERE p.PONo = 1020;

PONO   LINEITEMNO STOCK_REF.STOCKNO    QUANTITY    DISCOUNT
```

```
----- ---------- ---------------- ---------- ----------
1020           1             1004          1          0
1020           2             1011          3          5
1020           3             1535          2         10
```

The query uses two TABLE expressions, the second referring to the first. Line item information is selected from the inner nested table that belongs to purchase order number 1020 in the outer nested table.

Notice that no column from the customer table occurs in either the SELECT list or the WHERE clause. The customer table is listed in the FROM clause solely to provide a starting point from which to access the nested tables.

Here is a variation on the preceding query. This version shows that you can use the * wildcard to specify all columns of a TABLE expression collection:

```
SELECT p.PONo, i.*
  FROM Customer_objtab c, TABLE(c.PurchaseOrderList_ntab) p,
    TABLE(p.LineItemList_ntab) i
  WHERE p.PONo = 1020;
```

Type Inheritance and Substitutable Columns

Suppose that we deal with a lot of our larger, regular customers through an account manager. We would like to add a field for the ID of the account manager to the customer record for these customers.

Earlier, when we wanted to add an attribute for a nested table of purchase orders, we evolved the customer type itself. We could do that again to add an attribute for account manager ID, or we could create a subtype of the customer type and add the attribute only in the subtype. Which should we do?

To make this kind of decision, you need to consider whether the proposed new attribute can be meaningfully and usefully applied to all instances of the base type—to all customers, in other words—or only to an identifiable subclass of the base type.

All customers have purchase orders, so it was appropriate to alter the type itself to add an attribute for them. But not all customers have an account manager; in fact, it happens that only our corporate customers do. So, instead of evolving the customer type to add an attribute that will not be meaningful for customers in general, it makes more sense to create a new subtype for the special *kind* of customer that we have identified and to add the new attribute there.

Creating a Subtype

You can create a subtype under a base type only if the base type allows subtypes. Whether a type can be subtyped depends on the type's FINAL property. By default, new types are created as FINAL. This means that they are the last of the series and cannot have subtypes created under them. To create a type that can be subtyped, you must specify NOT FINAL in the CREATE TYPE statement as we did when we created the customer type.

You define a subtype by using a CREATE TYPE statement with the UNDER keyword. The following statement creates a new subtype Corp_Customer_objtyp under Customer_objtyp. The type is created as NOT FINAL so that it can have subtypes if we want to add them later.

Example A–50 Create Corp_Customer_objtyp

```
CREATE TYPE Corp_Customer_objtyp UNDER Customer_objtyp
```

```
            (account_mgr_id     NUMBER(6) ) NOT FINAL;
/
```

When you use a CREATE TYPE statement to create a new subtype, you list only the new attributes and methods that you are adding. The subtype inherits all existing attributes and methods from its base type, so these do not need to be specified. The new attributes and methods are added after the inherited ones. For example, the complete list of attributes for the new Corp_Customer_objtyp subtype looks like this:

```
CustNo
CustName
Address_obj
Phonelist_var
PurchaseOrderList_ntab
Account_mgr_id
```

By default, you can store instances of a subtype in any column or object table that is of any base type of the subtype. This ability to store subtype instances in a base type slot is called substitutability. Columns and tables are substitutable unless they have been explicitly declared to be NOT SUBSTITUTABLE. The system automatically adds new columns for subtype attributes and another, hidden column for the type ID of the instance stored in each row.

Actually, it is possible to create a subtype of a FINAL type, but first you must use an ALTER TYPE statement to evolve the type from a FINAL type to a NOT FINAL one. If you want existing columns and tables of the altered type to be able to store instances of new subtypes, specify the CASCADE option CONVERT TO SUBSTITUTABLE in the ALTER TYPE statement. See "Type Evolution" on page 8-5.

Inserting Subtypes

If a column or object table is substitutable, you can insert into it not only instances of the declared type of the column or table but also instances of any subtype of the declared type. In the case of table Customer_objtab, this means that the table can be used to store information about all kinds of customers, both ordinary and corporate. However, there is one important difference in the way information is inserted for a subtype: you must explicitly specify the subtype's constructor. Use of the constructor is optional only for instances of the declared type of the column or table.

For example, the following statement inserts a new ordinary customer, William Kidd.

Example A–51   Insert Data for Ordinary Customer

```
INSERT INTO Customer_objtab
  VALUES (
    3, 'William Kidd',
    Address_objtyp('43 Harbor Drive', 'Redwood Shores', 'CA', '95054'),
    PhoneList_vartyp('415-555-1212'),
    PurchaseOrderList_ntabtyp()
  );
```

The VALUES clause contains data for each Customer_objtyp attribute but omits the Customer_objtyp constructor. The constructor is optional here because the declared type of the table is Customer_objtyp. For the nested table attribute, the constructor PurchaseOrderList_ntabtyp() creates an empty nested table, but no data is specified for any purchase orders.

Here is a statement that inserts a new corporate customer in the same table. Note the use of the constructor `Corp_Customer_objtyp()` and the extra data value `531` for the account manager ID:

Example A–52 Insert Data for Corporate Customer

```
INSERT INTO Customer_objtab
  VALUES (
    Corp_Customer_objtyp(   -- Subtype requires a constructor
      4, 'Edward Teach',
      Address_objtyp('65 Marina Blvd', 'San Francisco', 'CA', '94777'),
      PhoneList_vartyp('415-555-1212', '416-555-1212'),
      PurchaseOrderList_ntabtyp(), 531
    )
  );
```

The following statements insert a purchase order for each of the two new customers. Unlike the statements that insert the new customers, the two statements that insert purchase orders are structurally the same except for the number of line items in the purchase orders:

Example A–53 Insert Purchase Order for Ordinary Customer

```
INSERT INTO TABLE (
  SELECT c.PurchaseOrderList_ntab
    FROM Customer_objtab c
    WHERE c.CustName = 'William Kidd'
   )
  VALUES (1021, SYSDATE, SYSDATE + 1,
    LineItemList_ntabtyp(
      LineItem_objtyp(1, MAKE_REF(Stock_objtab, 1535), 2, 10),
      LineItem_objtyp(2, MAKE_REF(Stock_objtab, 1534), 1, 0)
     )
   );
```

Example A–54 Insert Purchase Order for Corporate Customer

```
INSERT INTO TABLE (
  SELECT c.PurchaseOrderList_ntab
    FROM Customer_objtab c
    WHERE c.CustName = 'Edward Teach'
   )
  VALUES (1022, SYSDATE, SYSDATE + 1,
    LineItemList_ntabtyp(
      LineItem_objtyp(1, MAKE_REF(Stock_objtab, 1011), 1, 0),
      LineItem_objtyp(2, MAKE_REF(Stock_objtab, 1004), 3, 0),
      LineItem_objtyp(3, MAKE_REF(Stock_objtab, 1534), 2, 0)
     )
   );
```

Querying Substitutable Columns

A substitutable column or table can contain data of several data types. This enables you, for example, to retrieve information about all kinds of customers with a single query of the customers table. But you can also retrieve information just about a particular kind of customer, or about a particular attribute of a particular kind of customer.

The following examples show some useful techniques for getting the information you want from a substitutable table or column.

The query in Example A–55 uses a WHERE clause that contains an IS OF predicate to filter out customers that are not some kind of corporate customer. In other words, the query returns all kinds of corporate customers but does not return instances of any other kind of customer:

Example A–55 Selecting All Corporate Customers and Their Subtypes

```
SELECT c.*
  FROM Customer_objtab c
  WHERE VALUE(c) IS OF (Corp_Customer_objtyp);
```

The query in Example A–56 is similar to the preceding one except that it adds the ONLY keyword in the IS OF predicate to filter out any subtypes of Corp_Customer_objtyp. Rows are returned only for instances whose most specific type is Corp_Customer_objtyp.

Example A–56 Selecting All Corporate Customers with No Subtypes

```
SELECT p.PONo
  FROM Customer_objtab c, TABLE(c.PurchaseOrderList_ntab) p
  WHERE VALUE(c) IS OF (ONLY Corp_Customer_objtyp);
```

The query in Example A–57 uses a TABLE expression to get purchase order numbers (from the nested table of purchase orders). Every kind of customer has this attribute, but the WHERE clause confines the search just to corporate customers:

Example A–57 Selecting PONo Just for Corporate Customers

```
SELECT p.PONo
  FROM Customer_objtab c, TABLE(c.PurchaseOrderList_ntab) p
  WHERE VALUE(c) IS OF (Corp_Customer_objtyp);
```

The query in Example A–58 returns data for account manager ID. This is an attribute possessed only by the corporate customer subtype: the declared type of the table lacks it. In the query the TREAT() function is used to cause the system to try to regard or treat each customer as a corporate customer in order to access the subtype attribute Account_mgr_id:

Example A–58 Selecting a Subtype Attribute Using the TREAT Function

```
SELECT CustName, TREAT(VALUE(c) AS Corp_Customer_objtyp).Account_mgr_id
  FROM Customer_objtab c
  WHERE VALUE(c) IS OF (ONLY Corp_Customer_objtyp);
```

TREAT() is necessary in Example A–58 because Account_mgr_id is not an attribute of the table's declared type Customer_objtyp. If you simply list the attribute in the SELECT list as if it were, a query like the one in Example A–59 will return the error invalid column name error. This is so even with a WHERE clause that excludes all but instances of Corp_Customer_objtyp. The WHERE clause is not enough here because it merely excludes rows from the result.

Example A–59 Selecting a Subtype Attribute Without the TREAT Function

```
-- Following statement returns error, invalid column name for Account_mgr_id
SELECT CustName, Account_mgr_id
  FROM Customer_objtab c
  WHERE VALUE(c) IS OF (ONLY Corp_Customer_objtyp);
```

Every substitutable column or object table has an associated hidden type-ID column that identifies the type of the instance in each row. You can look up the type ID of a type in the `USER_TYPES` catalog view.

The function `SYS_TYPEID()` returns the type ID of a particular instance. The query in Example A–60 uses `SYS_TYPEID()` and a join on the `USER_TYPES` catalog view to return the type name of each customer instance in the table `Customer_objtab`:

Example A–60 Discovering the Type of Each Instance

```
SELECT c.CustName, u.TYPE_NAME
  FROM Customer_objtab c, USER_TYPES u
  WHERE SYS_TYPEID(VALUE(c)) = u.TYPEID;
```

```
--------------------------------- ---------------------
Jean Nance                        CUSTOMER_OBJTYP
John Nike                         CUSTOMER_OBJTYP
William Kidd                      CUSTOMER_OBJTYP
Edward Teach                      CORP_CUSTOMER_OBJTYP
```

For more information on `SYS_TYPEID()`, `VALUE()`, and `TREAT()`, see "Functions and Operators Useful with Objects" on page 2-31.

Index