Oracle® Database

Object-Relational Developer's Guide

11*g* Release 2 (11.2)

E11822-01

August 2009

ORACLE®

Oracle Database Object-Relational Developer's Guide 11*g* Release 2 (11.2)

E11822-01

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

Contents

3 Using PL/SQL With Object Types

4 Object Support in Oracle Programming Environments

5 Support for Collection Data Types

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

Preface

Oracle Database Object-Relational Developer's Guide explains 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

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++, 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 databases. You should be familiar with the information in *Oracle Database SQL Quick Reference*, *Oracle Database PL/SQL Language Reference*, and *Oracle Database 2 Day Developer's Guide*, and with object-oriented programming techniques.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible to all users, including users that are disabled. 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.

Deaf/Hard of Hearing Access to Oracle Support Services

To reach Oracle Support Services, use a telecommunications relay service (TRS) to call Oracle Support at 1.800.223.1711. An Oracle Support Services engineer will handle technical issues and provide customer support according to the Oracle service request process. Information about TRS is available at `http://www.fcc.gov/cgb/consumerfacts/trs.html`, and a list of phone numbers is available at `http://www.fcc.gov/cgb/dro/trsphonebk.html`.

Related Documents

For more information, see these Oracle resources:

- *Oracle Database Concepts* for information about basic Oracle concepts
- *Oracle Database SQL Language Reference* and *Oracle Database Administrator's Guide* for information about SQL
- *Oracle Database PL/SQL Language Reference* for information about PL/SQL, the procedural language extension to Oracle SQL
- *Oracle Database Advanced Application Developer's Guide* for general information about developing applications
- *Oracle Database JDBC Developer's Guide and Reference* and *Oracle Database Java Developer's Guide* for information about Oracle object-relational features through Java
- *Oracle Call Interface Programmer's Guide* and *Oracle C++ Call Interface Programmer's Guide* for information about using the Oracle Call Interface (OCI) and Oracle C++ Call Interface (OCCI) to build third-generation language (3GL) applications that interact with one or more the Oracle databases
- *Pro*C/C++ Programmer's Guide* for information about 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 XML DB Developer's Guide* and *Oracle XML Developer's Kit Programmer's Guide* for information about developing applications with XML
- *Oracle Database SecureFiles and Large Objects Developer's Guide* for information about Large Objects (LOBs)
- *Oracle Data Provider for .NET Developer's Guide*
- *Oracle Developer Tools for Visual Studio Help*

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 users upgrading to the current release.

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

- Oracle Database 11g Release 2 (11.2) New Features in 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 2 (11.2) New Features in Object-Relational Features

FORCE Option Added to CREATE OR REPLACE TYPE Statement

`CREATE OR REPLACE TYPE` statements now have a `FORCE` option, which enables you to replace the type even if it has type dependents. This does not work without the `FORCE` option.

> **See Also:** "Using CREATE OR REPLACE TYPE with Type and Table Dependencies" on page 7-7

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 2-18.

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 "Collection Data Types" on page 5-1.

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 5-12.

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

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.

New object types can be created from any built-in database types and any previously created object types, object references, and collection types. Object types can work with complex data, such as images, audio, and video. Oracle Database stores metadata for user-defined types in a schema that is available to SQL, PL/SQL, Java, and other languages.

Object types and related object-oriented features, such as varrays 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 can work with the data in terms of the real-world entities that make the data meaningful. Instead of thinking in terms of columns and tables when you query the database, you can simply select entities that you have created, such as customers and purchase orders.

You can begin to use object-oriented features while continuing to work with most of your data relationally, or you use to an object-oriented approach entirely.

Object types are also known as user-defined types or ADTs. *Oracle Database PL/SQL Language Reference* generally refers to them as ADTs.

Advantages of Objects

In general, the object-type model is similar to the class mechanism found in C++ and Java. Like classes, the reusability of objects makes it possible to develop database applications faster and more efficiently. By natively supporting object types in the database, Oracle Database enables application developers to directly access the data structures used by their applications.

Objects offer other advantages over a purely relational approach, such as:

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 performed on that data. Thus, a purchase order object might include a method to calculate 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 allows 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 do not need to re-create similar structures and methods in every application. This also ensures that developers are using consistent standards.
- 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. When you reference a column of a SQL object type, you retrieve the whole object.

Objects Can Represent Part-Whole Relationships

Object types allow you to represent part-whole relationships. For example: in a relational table for stock items, a piston and an engine may have the same status. Using objects can reduce the need to represent pistons as parts of engines with complicated schemas of multiple tables with primary key-foreign key 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 Database implements the object-type model as an extension of the relational model, while continuing to support standard relational database functionality, such as queries, fast commits, backup and recovery, scalable connectivity, row-level locking, read consistency, and more.

SQL and various programmatic interfaces and languages, including PL/SQL, Java, Oracle Call Interface, Pro*C/C++, OO4O, and C# have been enhanced with extensions to support Oracle 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.

This section contains these topics:

Database Features of Oracle Objects

This section describes features and concepts of the object-relational model that are related to the database.

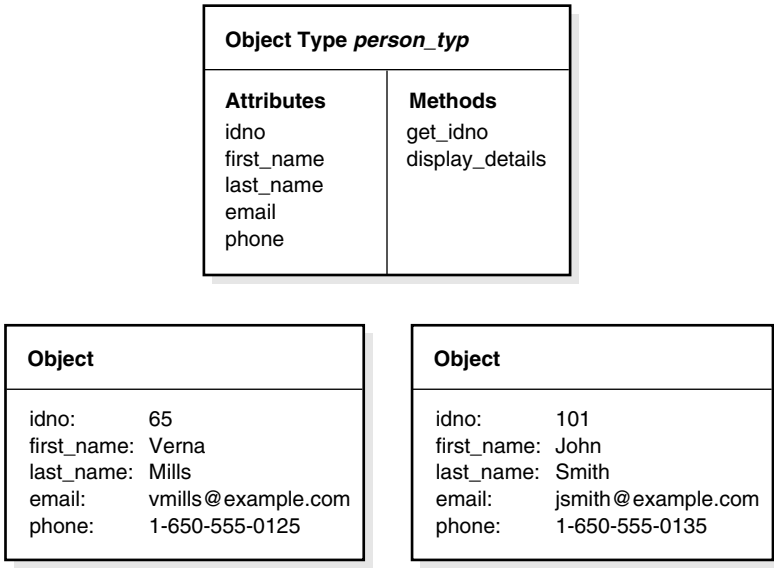
This section contains these topics:

About Object Types

An object type is a kind of data type. You can use it in the same ways that you use standard 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. The value is a variable or an instance of that type. An object instance is also called an object.

Figure 1–1 shows an object type, `person_typ`, and two instances of the object type.

Figure 1–1 An Object Type and Object Instances



Object types serve as blueprints or templates that define both structure and behavior. Object types are database schema objects, subject to the same kinds of administrative control as other schema objects. Application code can retrieve and manipulate these objects. See Chapter 7, "Managing Oracle Objects".

You use the `CREATE TYPE` SQL 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 PL/SQL Language Reference*.

> **Note: Running Examples:** Many examples in this guide can be run using the HR sample schema. Comments at the beginning of most examples indicate if any previous example code is required.
>
> Refer to *Oracle Database Sample Schemas* for information on how these schemas were created and how you can use them yourself.

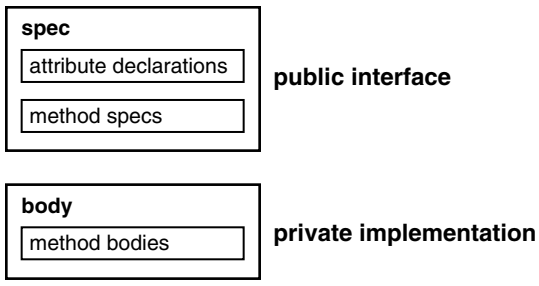
Example 1–1 Creating the person_typ Object Type

```
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 differ from the standard data types that are native to a relational database:

- Oracle Database does not supply predefined object types. You define the object types you want by combining built-in types with user-defined ones as shown in Example 1–1.
- Object types are composed of attributes and methods as illustrated in Figure 1–2.
  - Attributes hold the data about an object. Attributes have declared data types which can, in turn, be other object types.
  - Methods are procedures or functions that applications can use to perform operations on the attributes of the object type. Methods are optional. They define the behavior of objects of that type.

Figure 1–2 Object Attributes and Methods



About Object Instances

A variable of an object type is an instance of the type, or 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.

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 use types such as `NUMBER` or `VARCHAR2`. Storage is allocated once you create an instance of the object type.

Example 1–2 shows how to create object instances of the `person_typ` created in Example 1–1, and define a relational table to keep track of these instances as contacts.

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

```
-- requires existing person_typ fr. Ex 1-1
CREATE TABLE contacts (
  contact         person_typ,
  contact_date    DATE );

INSERT INTO contacts VALUES (
  person_typ (65, 'Verna', 'Mills', 'vmills@example.com', '1-650-555-0125'),
 '24 Jun 2003' );
```

The `contacts` table is a relational table with an object type as the data type of its contact column. Objects that occupy columns of relational tables are called column objects. See "How Objects are Stored in Tables" on page 1-6.

About Object Methods

Object 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.

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

- Member Methods

  Using member methods, you can provide access to the data of an object, and otherwise define operations that an application performs on the data. To perform an operation, the application calls the appropriate method on the appropriate object.

- Static Methods

  Static methods compare object instances and perform operations that do not use the data of any particular object, but, instead, are global to an object type.

- Constructor Methods

A default constructor method is implicitly defined for every object type, unless it is overwritten with a user-defined constructor. A constructor method is called on a type to construct or create an object instance of the type.

Example 1–3 show the `get_idno()` method, created in Example 1–1, to display the Id number of persons in the `contacts` table:

Example 1–3 Using the get_idno Object Method

```
-- requires Ex 1-1 and Ex 1-2
SELECT c.contact.get_idno() FROM contacts c;
```

See Also: "Object Methods" on page 2-8 for detailed information

How Objects are Stored in Tables

Objects can be stored in two types of tables:

- Object tables: store only objects

  In an object table, each row represents an object, which is referred to as a **row object**. See "Creating and Using Object Tables" on page 1-6

- Relational tables: store objects with other table data

  Objects that are stored as columns of a relational table, or are attributes of other objects, are called **column object**s. Example 1–2 shows the `contacts` table which stores an instance of the `person_typ` object.

Objects that have meaning outside of the relational database 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 as a row object in an object table instead of in a column of a relational table.

See Also: "Storing Objects as Columns or Rows" on page 9-1

Creating and Using Object Tables

Example 1–4 shows a `CREATE TABLE` statement that creates an object table for `person_typ` objects.

Example 1–4 Creating the person_obj_table Object Table

```
-- requires Ex. 1-1
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`, `first_name`, `last_name`, and so on, 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

```
-- requires Ex. 1-1 and 1-4
INSERT INTO person_obj_table VALUES (
       person_typ(101, 'John', 'Smith', 'jsmith@example.com', '1-650-555-0135') );
```

```
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 `INSERT INTO` 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 `SELECT` 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 2-37 for information on the `VALUE` function.

The PL/SQL block selects a specific person and executes a member function of `person_typ` to display details about the specified person. For more information about using PL/SQL with objects, see Chapter 3, "Using PL/SQL With Object Types".

Using Object Identifiers to Identify Row Objects

Object identifiers (OIDs) uniquely identify row objects in object tables. You cannot directly access object identifiers, but you can make references (`REF`s) to the object identifiers and directly access the `REF`s, as discussed in "Using References to Row Objects" on page 1-7.

There are two types of **object identifier**s.

- System-Generated Object Identifiers (default)

  Oracle automatically creates system-generated object identifiers for row objects in object tables unless you choose the primary-key based option.

- Primary-Key Based Object Identifiers

  You have the option to create primary-key based OIDs when you create the table using the `CREATE TABLE` statement.

> **Note:** Column objects are identified by the primary key of the row, and, therefore, do not need a specific object identifier.

See Also:

- "Specifying Object Identifiers for Object Views" on page 6-8
- "Storage Considerations for Object Identifiers (OIDs)" on page 9-4
- "Object Identifiers and References" on page A-16

Using References to Row Objects

A `REF` is a logical pointer or reference to a row object that you can construct from an object identifier (OID). You can use the `REF` to obtain, examine, or update the object. 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.

REFs are Oracle Database built-in data types. REFs and collections of REFs model associations among objects, particularly many-to-one relationships, thus reducing the need for foreign keys. REFs provide an easy mechanism for navigating between objects.

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';
```

This example first creates the emp_person_typ John Smith, with NULL value for a manager. Then it adds the emp_person_typ Bob Jones as John Smith's supervisee.

The following query and its output show the effect:

```
COLUMN name FORMAT A10
COLUMN manager FORMAT A50
select * from emp_person_obj_table e;


NAME       MANAGER
---------- --------------------------------------------------
John Smith
Bob Jones  0000220208424E801067C2EABBE040578CE70A0707424E8010
           67C1EABBE040578CE70A0707
```

Example 1–10 on page 1-9 shows how to dereference the object, so that Manager appears as a name rather than an object identifier.

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

Using Scoped REFs You can constrain a column type, collection element, or object type attribute to reference a specified object table by using the SQL constraint subclause SCOPE IS when you declare the 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

```
-- requires Ex. 1-1, 1-4, and 1-5
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 a REF is scoped to an object table of a subtype, the REF column is effectively constrained to hold only references to instances of the subtype (and its subtypes, if any) in the table. See "Inheritance in SQL Object Types" on page 2-13.

Checking for Dangling REFs It is possible for the object identified by a REF to become unavailable if the object has been deleted or some necessary privilege has been deleted. This is a dangling REF. You can use the Oracle Database SQL predicate IS DANGLING to test 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 that the REF refers to is called dereferencing the REF. Oracle Database provides the DEREF operator to do this.

Example 1–8 Using DEREF to Dereference a REF

```
-- requires Ex. 1-6
SELECT DEREF(e.manager) FROM emp_person_obj_table e;

DEREF(E.MANAGER)(NAME, MANAGER)
--------------------------------------------------------------------
EMP_PERSON_TYP('John Smith', NULL)
```

Example 1–9 shows that dereferencing a dangling REF returns a null object.

Example 1–9 Dereferencing a Dangling Ref

```
--requires Ex. 1-1, 1-4, 1-5, and 1-7
-- DELETE command needed to cause dangling ref
DELETE from person_obj_table WHERE idno = 101;
/
SELECT DEREF(c.contact_ref), c.contact_date FROM contacts_ref c;
```

Oracle Database also provides implicit dereferencing of REFs. For example, to access the manager's name for an employee, you can use a SELECT statement.

Example 1–10 follows the pointer from the person's name and retrieves the manager's name e.manager.name.

Example 1–10 Implicitly Deferencing a REF

```
-- requires Ex. 1-6
SELECT e.name, e.manager.name FROM emp_person_obj_table e
  WHERE e.name = 'Bob Jones';
```

Following the REF in this manner is allowed in SQL, but PL/SQL requires the DEREF keyword as in Example 1–8 on page 1-9.

Obtaining a REF to a Row Object You can obtain a `REF` to a row object by selecting the object from its object table and applying the `REF` operator.

Example 1–11 shows how to obtain a `REF` to the person with an `idno` equal to `101`.

Example 1–11 Obtaining a REF to a Row Object

```
-- requires Ex. 1-1, 1-4, and 1-5
DECLARE
  person_ref REF person_typ;
  person person_typ;
BEGIN

  SELECT REF(p) INTO person_ref
    FROM person_obj_table p
    WHERE p.idno = 101;

   select deref(person_ref) into person from dual;
   person.display_details();

END;
/
```

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

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. They can only be compared for equality.

Using Oracle Collections

For modeling multi-valued attributes and many-to-many relationships, Oracle Database supports two collection data types: varrays and nested tables. You can use collection types anywhere other data types are used. You can have object attributes of a collection type in addition to columns of a collection type. For example, a purchase order object type might contain a nested table attribute that holds the collection of line items for the purchase order.

To define a collection type, use the `CREATE TYPE . . . AS TABLE OF` statement.

Example 1–12 shows `CREATE TYPE` statements that define a collection and an object type.

Example 1–12 Creating the people_typ Collection Data Type

```
-- requires Ex. 1-1
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);
/
```

Note the following about this example:

- The collection type, `people_typ`, is specifically a nested table type.

- The `dept_persons_typ` object type has an attribute `dept.emps` of `people_typ`. Each row in the `dept.emps` nested table is an object of type `person_typ` which was defined in Example 1–1 on page 1-4.

See Also: "Collection Data Types" on page 5-1

Using Object Views to Access Relational Data

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.

You can access objects that belong to an object view in the same way that you access row objects in an object table. Oracle Database also supports **materialized view** objects of user-defined types from data stored in relational schemas and tables.

Object views let you exploit the **polymorphism** that a type hierarchy makes possible. A polymorphic expression takes 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".

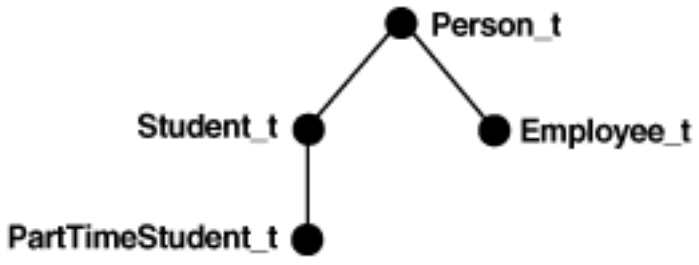
Using Type Inheritance

Type inheritance enables you to create type hierarchies. A type hierarchy is a set of successive levels of increasingly specialized subtypes that derive from a common ancestor object type, which is called a supertype. Derived subtypes inherit the features of the parent object type and can 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 Also: "Inheritance in SQL Object Types" on page 2-13

Figure 1–3 illustrates two subtypes, `Student_t` and `Employee_t`, created under `Person_t`, and the `PartTimeStudent_t`, a subtype under `Student_t`.

Figure 1–3 A Type Hierarchy



Using Type Evolution to Change an Existing Object Type

Type evolution enables you to modify, or evolve, an existing object type, even those already used in tables. Type evolution works through the `ALTER TYPE` statement, enabling you to propagate changes through all instances of the object type.

The `ALTER TYPE` statement checks for dependencies of the type to be altered, using essentially the same validations as 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 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.

See Also:

- "Type Evolution" on page 8-6
- "Considerations Related to Type Evolution" on page 9-25

Language Binding Features of Oracle Objects

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 object-related features, Oracle Database provides SQL extensions, including DDL, to create, alter, or drop object types; to store object types in tables; and to create, alter, or drop object views. There are DML and query extensions to support object types, references, and collections. See "SQL and Object Types" on page 4-1.

PL/SQL Object Extensions

PL/SQL can 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 Chapter 3, "Using PL/SQL With Object Types" on page 3-1.

Java Support for Oracle Objects

Oracle Java VM is tightly integrated with Oracle Database and supports access to Oracle Objects through object extensions to Java Database Connectivity (JDBC). This 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. You can map SQL types to existing Java classes to provide persistent storage for Java objects. See "Java Object Storage" on page 4-12.

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

External Procedures

You can implement database functions, procedures, or member methods of an object type in PL/SQL, Java, C, or .NET 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. External procedures are always run in a safe mode outside the address space of the database. Generic external procedures can be written that declare one or more parameters to be of a system-defined generic type. Thus, an external procedure can use the system-defined generic type 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. You can use these generated header files in host-language applications for transparent access to database objects.

Client-Side Cache

Oracle Database 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. Oracle Call Interface provides a run-time environment with functions to connect to an Oracle Database, and control transactions that access objects in the database. 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 provides a number of functions to access metadata about object types defined in the database at run-time. See "Oracle Call Interface (OCI)" on page 4-2 and "Oracle C++ Call Interface (OCCI)" on page 4-7.

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. Pro*C/C++ also provides simplified syntax to allocate and free objects of SQL types and access them using SQL DML or the navigational interface. See "Oracle Call Interface (OCI)" on page 4-2.

OO4O Object Extensions

Oracle Objects For OLE (OO4O) is a set of COM Automation interfaces and objects for connecting to Oracle database servers, executing queries and managing the results. Automation interfaces in OO4O provide easy and efficient access to Oracle Database 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 4-8.

.NET Object Extensions

Oracle Developer Tools for Visual Studio (ODT) and Oracle Data Provider for .NET (ODP.NET) support .NET custom objects that map to Oracle object-relational data types, collections, and REFs. ODT is a set of tools incorporated into a Visual Studio integrated development environment, which allow managing these data types inside the Oracle database. Through the ODT Custom Class Wizard, Oracle objects can be automatically mapped to .NET custom types to ease data sharing between Oracle

databases and .NET applications. Data access to these .NET custom types occur through ODP.NET.

See Also:

- *Oracle Database Extensions for .NET Developer's Guide*
- *Oracle Data Provider for .NET Developer's Guide*
- *Oracle Developer Tools for Visual Studio Help*

2

Basic Components of Oracle Objects

This chapter provides basic information about working with Oracle SQL 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.

> **Note:** **Running Examples:** In order to run examples in chapter 2, you may need to drop any objects you created for Chapter 1.

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 create Oracle SQL object types with the `CREATE TYPE` statement. A typical example of object type creation is shown in Example 2–1 on page 2-2.

See Also:

- *Oracle Database PL/SQL Language Reference* for information on the `CREATE TYPE` SQL statement
- *Oracle Database PL/SQL Language Reference* for information on the `CREATE TYPE BODY` SQL statement

Null Objects and Attributes

An object whose value is NULL is called atomically null. An **atomically null object** is different from an object that has null values for all its attributes.

In an object with null values, a table column, object attribute, collection, or collection element might be 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. When all the attributes are null, you can still change these attributes and call the object's subprograms or methods. With an atomically null object, you can do neither of these things.

Example 2–1 creates the contacts table and defines the person_typ object type and two instances of this type.

Example 2–1 Inserting NULLs for Objects in a Table

```
CREATE OR REPLACE 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 OR REPLACE 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' );
```

Two instances of person_typ are inserted into the table and give two different results. In both cases, Oracle Database allocates space in the contacts table for a new row and sets its DATE column to the value given. But in the first case, Oracle Database allocates space for an object in the contact column and sets each of the object's attributes to NULL. In the second case, Oracle Database 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, one that has no elements.

See Also: "How PL/SQL Treats Uninitialized Objects" on page 3-3

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 the qualifier char to the length specification.

Like CHAR and VARCHAR2, NCHAR and NVARCHAR2 may also be used as attribute types in objects and collections. NCHAR and NVARCHAR2 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 OR REPLACE 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 NLS_LENGTH_SEMANTICS initialization parameter setting (CHAR or BYTE) indicates the default unit of measure.

> **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 attribute**s of a column object, with the exception of REFs that are not scoped.

The following examples illustrate defining constraints.

Example 2–3 places an implicit PRIMARY KEY constraint on the office_id column of the object table office_tab.

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

```
-- requires Ex. 2-1
CREATE OR REPLACE TYPE location_typ AS OBJECT (
  building_no  NUMBER,
  city         VARCHAR2(40) );
/

CREATE OR REPLACE 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 object type location_typ defined in Example 2–3 is the type of the dept_loc column in the department_mgrs table in Example 2–4.

Example 2–4 defines constraints on scalar attributes of the `location_typ` objects in the table.

Example 2–4 Creating the department_mgrs Table with Multiple Constraints

```
-- requires Ex. 2-1 and 2-3
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', '1-1-650-555-0125'),
           location_typ(300, 'Palo Alto'));
```

See Also: "Constraints on Objects" on page 9-25

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 5–5 on page 5-5.

You can define indexes on **leaf-level scalar attribute**s of column objects, as shown in Example 2–5. You can only define indexes on `REF` attributes or columns if the `REF` is scoped. This example indexes `city`, which is a leaf-level scalar attribute of the column object `dept_addr`.

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

```
-- requires Ex. 2-1, 2-3,
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 Database expects a column name in an index definition, you can also specify a scalar attribute of a column object.

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

```
-- requires Ex. 2-1 and 2-3
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 office_tab VALUES
    ('BE32', location_typ(300, 'Palo Alto' ),person_typ(280, 'John Chan',
       '415-555-0101'));

UPDATE office_tab set office_loc =location_typ(600, 'Redwood Shores')
  where office_id = 'BE32';

select * from office_tab;

select * from movement;
```

> **See Also:** "Using INSTEAD OF Triggers to Control Mutating and Validation" on page 6-11

Rules for REF Columns and Attributes

In Oracle Database, 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 Database 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 SCOPE constraint can be applied 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.

See Also:

- "Using References to Row Objects" on page 1-7
- "Substitution of REF Columns and Attributes" on page 2-26

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 Database determines which table each column belongs to.

Using 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

```
-- requires Ex. 2-1, 2-3, 2-5, and 2-6
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. For example, if you add an assignment column to depts and forget to change the query, Oracle Database 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 Database 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 the object person_typ.

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

Note: These statements are not related to other examples in this chapter.

```
#1 SELECT idno FROM person_obj_table;          --Correct

#2 SELECT contact.idno FROM contacts;          --Illegal
#3 SELECT contacts.contact.idno FROM contacts; --Illegal
#4 SELECT p.contact.idno FROM contacts p;      --Correct
```

- In #1, `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 #2, `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 #4.
- #3 uses the table name itself to qualify 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 incorrectly refers to the `HR` schema, `department_loc` table, `dept_addr` column, and `city` attribute of that column. 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 `REF`s.

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

Objects or 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. Oracle Database restricts use of a database link as follows:

- You cannot 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*.
- You cannot use database links within PL/SQL code to declare a local variable of a remote user-defined type.

- You cannot convey a user-defined type argument or return value in a PL/SQL remote procedure call.

Object Methods

Object methods, also known as subprograms, 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.

Subprograms 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.

Note: SQL requires parentheses for all subprogram calls, even those that do not have arguments. This is not true for PL/SQL.

This section focuses on declaring methods. See "Calling Object Constructors and Methods" on page 3-4 for further discussion of invoking methods in PL/SQL.

This section describes these methods:

- Member Methods
- Static Methods
- Constructor Methods
- External Implemented Methods

Member Methods

Member methods provide an application with 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. Non-comparison member methods are declared as either `MEMBER FUNCTION` or `MEMBER PROCEDURE`. Comparison methods use `MAP MEMBER FUNCTION` or `ORDER MEMBER FUNCTION` as described in "Member Methods for Comparing Objects" on page 2-9.

As an example of a member method, you might declare a function `get_sum()` that sums the total cost of a purchase order's line items. The following line of code calls this function for purchase order `po` and returns the amount into `sum_line_items`.

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

Dot notation specifies the current object and the method it calls. Parentheses are required even if there are no parameters.

This section contains these topics:

- SELF Parameters in Member Methods
- Member Methods for Comparing Objects

SELF Parameters in Member Methods

Member methods have a built-in parameter named `SELF` that denotes the object instance currently invoking the method.

`SELF` can be explicitly declared, but that is not necessary. It is simpler to write member methods that reference the attributes and methods of `SELF` implicitly without the `SELF` qualifier. In Example 2–8, the code and comments demonstrate method

invocations that use an implicit SELF parameter rather than qualify the attributes hgt, len, and wth.

Example 2–8 Creating a Member Method

```
-- Ex. 2-8 Creating a Member Method
CREATE OR REPLACE 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 OR REPLACE 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 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;
/
```

SELF 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.

See also "Using SELF IN OUT NOCOPY with Member Procedures" on page 9-21.

Member Methods for Comparing Objects

To compare and order variables of an object type, you must specify a basis for comparing them. 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. You have the option to define an map method or an order method for comparing objects, but not both.

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

Map Methods Map methods return values that can be used for comparing and sorting. Return values can be any Oracle built-in data types (except LOBs and BFILEs) and ANSI SQL types such as CHARACTER or REAL. See the specific sections in *Oracle Database SQL Quick Reference*.

Generally, map methods perform calculations on the attributes of the object to produce the return value.

Map methods are 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()
```

Comparisons are similar for other relational operators.

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

CREATE OR REPLACE TYPE BODY rectangle_typ AS
  MAP MEMBER FUNCTION area RETURN NUMBER IS
  BEGIN
     RETURN len * wid;
  END area;
END;
/
```

Example 2–10 Invoking a Map Method

```
DECLARE
  po rectangle_typ;

BEGIN
  po :=NEW rectangle_typ(10,5);

DBMS_OUTPUT.PUT_LINE('AREA:' || po.area()); -- prints AREA:50
  END;
```

```
/
```

A subtype can declare a map method only if its root supertype declares one.

See "Equal and Not Equal Comparisons" on page 5-19 for the use of map methods when comparing collections that contain object types.

Order Methods Order methods make direct one-to-one object comparisons. Unlike map methods, they cannot determine the order of a number of objects. They simply tell you that the current object is less than, equal to, or greater than the object that it is being compared to, based on the criterion used.

An order method is a function for an object (`SELF`), with one declared parameter that is an object of the same type. The method must return either a negative number, zero, or a positive number. This value signifies that the object (the implicit undeclared `SELF` parameter) is less than, equal to, or greater than the declared parameter 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.

Example 2–11 shows an order method that compares locations by building number:

Example 2–11 Creating and Invoking an Order Method

```
DROP TYPE location_typ FORCE;
-- above necessary if you have previously created object
CREATE OR REPLACE TYPE location_typ AS OBJECT (
  building_no  NUMBER,
  city         VARCHAR2(40),
  ORDER MEMBER FUNCTION match (l location_typ) RETURN INTEGER );
/
CREATE OR REPLACE 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;
/

-- invoking match method
DECLARE
loc location_typ;
secloc location_typ;
a number;

BEGIN
 loc :=NEW location_typ(300, 'San Francisco');
 secloc :=NEW location_typ(200, 'Redwood Shores');
 a := loc.match(secloc);

DBMS_OUTPUT.PUT_LINE('order (1 is greater, -1 is lesser):' ||a); -- prints order:1
 END;
```

/

Note: Only a type that is not derived from another type can declare an order method; a subtype cannot define one.

Guidelines for Comparison Methods

You can declare a map method or an order method but not both. For either method type, you can compare objects using SQL statements and PL/SQL procedural statements. However, if you do not declare one of these methods, you can only compare objects 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, which 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, if the root type (supertype) does not specify a map or an order method, neither can the subtypes.

- Map Method in a Type Hierarchy

  If the root type specifies a map method, any of its subtypes can override it. If the root type does not specify a map method, no subtype can specify one either.

- Order Method in a Type Hierarchy

  Only the root type can define an order method. If the root type does not define one, its subtypes cannot add one.

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.

Static methods are declared using `STATIC FUNCTION` or `STATIC PROCEDURE`.

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

```
type_name.method()
```

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

Constructor Methods

A constructor method is a function that returns a new instance of the user-defined type and sets up the values of its attributes. Constructor methods are either system-defined or user-defined.

To invoke a constructor, the keyword `NEW` can be used, but is not required.

See Also: See Example 2–10 and "Calling Object Constructors and Methods" on page 3-4

System-Defined Constructors

By default, the system implicitly defines a constructor function for all object types that have attributes. This constructor is sometimes known as the attribute value constructor.

For the `person_typ` object type defined in Example 2–1 on page 2-2 the name of the constructor method is the name of the object type, as shown in the following invocation:

```
person_typ (1, 'John Smith', '1-650-555-0135'),
```

User-Defined Constructors

You can also define constructor functions of your own to create and initialize user-defined types. The default system-defined constructors (or 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 8-17. See "Constructor Methods for Collections" on page 5-2 for information on user-defined constructors for collections.

Literal Invocation of a Constructor Method

A **literal invocation** of a constructor method is a call to the constructor method in which arguments are either literals (as opposed to bind variables), 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', '1-650-555-0135') );
```

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:** Chapter 4, "Object Support in Oracle Programming Environments"

Inheritance in SQL Object Types

SQL object inheritance is based on a family tree of object types that forms a type hierarchy. The type hierarchy consists of a parent object type, called a supertype, and one or more levels of child object types, called subtypes, which are derived from the parent.

Inheritance is the mechanism that connects subtypes in a hierarchy to their supertypes. Subtypes automatically inherit the attributes and methods of their parent type. Also, the inheritance link remains alive. 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.

> **Note:** Oracle only supports single inheritance. Therefore, a subtype can derive directly from only one supertype, not more than one.

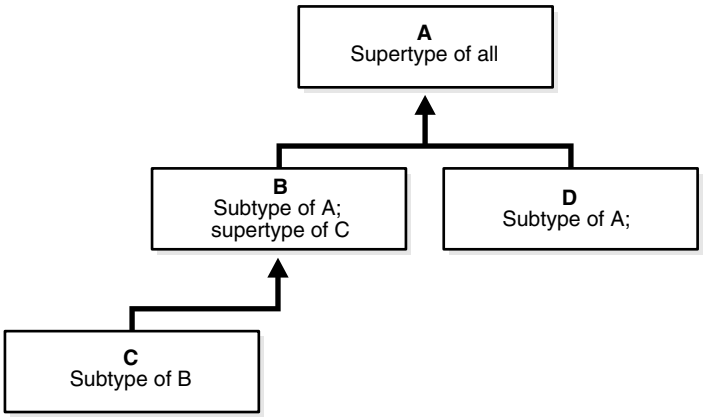
With object types in a type hierarchy, you can model an entity such as a customer, and also define different **specializing**d subtypes of customers under the original type. You can then perform operations on a hierarchy and have each type implement and execute the operation in a special way.

The topics described in this section are:

Supertypes and Subtypes

A subtype can be derived from a supertype either directly or indirectly through intervening levels of other subtypes. A supertype can have multiple sibling subtypes, but a subtype can have at most one direct parent supertype (single inheritance).

Figure 2–1 Supertypes and Subtypes in Type Hierarchy



To derive a subtype from a supertype, define a specialized variant of the supertype that adds new attributes and methods to the set inherited from the parent or redefine (override) the inherited methods. 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 distinguishes a subtype from its parent supertype is some change made to the attributes or methods that the subtype received from its parent.

Unless a subtype redefines 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. 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(). If the definition of person_typ changes, so do the definitions of any subtypes.

Subtypes are created using the keyword UNDER as follows:

```
CREATE TYPE student_typ UNDER person_typ
```

See Also: Example 2–15 on page 2-17 for a complete example

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 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 that a subtype inherits so that the subtype's version executes different code from the parent's.

  For example, a ellipse object might define a method calculate(). Two subtypes of ellipse_typ, circle_typ and sphere_typ, might each implement this method in a different way.

See Also: "Overloading and Overriding Methods" on page 2-21

The inheritance relationship between a supertype and its subtypes is the source of much of the power of objects and much of their complexity. Being able to change a method in a supertype and have the change take effect in all the subtypes downstream just by recompiling is very powerful. But this same capability means that you have to consider whether or not you want to allow a type to be specialized or a method to be redefined. Similarly, for a table or column to be able to contain any type in a hierarchy is also powerful, but you must decide whether or not to allow this in a particular case. Also, you may need to constrain DML statements and queries so that they pick out just the range of types that you want from the type hierarchy.

Defining FINAL and NOT FINAL Types and Methods

For an object type to be inheritable, the object type definition must specify that it is inheritable. Then subtypes can be derived from it. For a method, the definition must indicate whether or not it can be overridden The keywords FINAL or NOT FINAL are used for both types and methods.

In order for an object type to be inheritable, thus allowing subtypes to be derived from it, the object definition must specify this. For a method, the definition must indicate whether or not it can be overridden The keywords FINAL or NOT FINAL are used for both types and methods.

- For a type FINAL, (default) means that no subtypes can be derived from it. NOT FINAL means subtypes can be derived.

- For a method, FINAL means that subtypes cannot override it by providing their own implementation. NOT FINAL (default) means that you can override the method of the supertype.

Definitions of object types and method includes the NOT FINAL or FINAL keywords in the type and method declarations, as shown in Example 2–12 and Example 2–13.

Example 2–12 Creating the person_typ Object Type as NOT FINAL

```
DROP TYPE person_typ FORCE;
-- above necessary if you have previously created object

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

Example 2–12 declares person_typ to be a not final type and therefore subtypes of person_typ can be defined.

Example 2–13 creates a not final object type that contains a final member function.

Example 2–13 Creating an Object Type as NOT FINAL with a FINAL Member Function

```
DROP TYPE person_typ FORCE;
-- above necessary if you have previously created object

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

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

Creating Subtypes

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

Creating a Parent or Supertype Object

Example 2–14 provides a parent or supertype person_typ object to demonstrate subtype definitions in Example 2–15, Example 2–18, and Example 2–19.

Note the show() in Example 2–14. In the subtype examples that follow, the show() function of the parent type is overridden to specifications for each subtype using the OVERRIDING keyword.

Example 2–14 Creating the Parent or Supertype person_typ Object

```
DROP TYPE person_typ FORCE;
```

```
-- if created
CREATE OR REPLACE 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 OR REPLACE 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.

Example 2–15 defines the `student_typ` object as a subtype of `person_typ`, which inherits all the attributes declared in or inherited by `person_typ` and any methods inherited by or declared in `person_typ`.

Example 2–15 Creating a student_typ Subtype Using the UNDER Clause

```
-- requires Ex. 2-14
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;
/
```

The statement that defines `student_typ` specializes `person_typ` by adding two new attributes, `dept_id` and `major` and overrides the `show` method. 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 subtype member method. Example 2–15 demonstrates this 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.

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 evaluates to.

In Example 2–16, 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 2–16 Using Generalized Invocation

```
-- Requires Ex. 2-14 and 2-15
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 2–17 Using Generalized Expression

```
-- Requires Ex. 2-14 and 2-15
DECLARE
 myvar2 student_typ := student_typ(101, 'Sam', '6505556666', 100, 'Math');
 name2 VARCHAR2(100);
BEGIN
 name2 := person_typ.show((myvar2 AS person_typ)); -- Generalized expression
END;
/
```

Double parentheses are used in this example because `((myvar2 AS person_typ))` is both an expression that must be resolved and the parameter of the `show` function.

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 subtypes can also have subtypes. Example 2–18 creates another subtype `employee_typ` under `person_typ` in addition to the already existing subtype, `student_typ`, created in Example 2–15.

Example 2–18 Creating an employee_typ Subtype Using the UNDER Clause

```
-- requires Ex. 2-14

DROP TYPE employee_typ FORCE;
-- if previously created
CREATE OR REPLACE TYPE employee_typ UNDER person_typ (
    emp_id NUMBER,
    mgr VARCHAR2(30),
    OVERRIDING MEMBER FUNCTION show RETURN VARCHAR2);
/

CREATE OR REPLACE 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 2–19 defines a new subtype `part_time_student_typ` under `student_typ` created in Example 2–15. The new subtype inherits all the attributes and methods of `student_typ` and adds another attribute, `number_hours`.

Example 2–19 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 supertype and subtype instances and populate the table as shown with the `person_obj_table` in Example 2–20.

Example 2–20 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', '650-555-0130'));

INSERT INTO person_obj_table
  VALUES (student_typ(51, 'Joe Lane', '1-650-555-0140', 12, 'HISTORY'));

INSERT INTO person_obj_table
  VALUES (employee_typ(55, 'Jane Smith', '1-650-555-0144',
                       100, 'Jennifer Nelson'));

INSERT INTO person_obj_table
  VALUES (part_time_student_typ(52, 'Kim Patel', '1-650-555-0135', 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;
```

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.

Declaring Types and Methods NOT INSTANTIABLE

Types and methods can be declared `NOT INSTANTIABLE` when they are created.

- `NOT INSTANTIABLE` Types

  If a type is not instantiable, you cannot instantiate instances of that type. There are no constructors (default or user-defined) for it. You might use this with types intended to serve solely as supertypes from which specialized subtypes are instantiated.

- `NOT INSTANTIABLE` Methods

  A non-instantiable method serves as a placeholder. It is declared but not implemented in the type. You might define a non-instantiable method when you expect every subtype to override the method in a different way. In this case, there is no point in defining the method in the supertype.

A type that contains a non-instantiable method must itself be declared not instantiable, as shown in Example 2–21.

Example 2–21 Creating an Object Type that is NOT INSTANTIABLE

```
DROP TYPE person_typ FORCE;
-- if previously created
CREATE OR REPLACE 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;
/
```

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 2–22 Altering an Object Type to INSTANTIABLE

```
CREATE OR REPLACE 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.

Overloading and Overriding Methods

A subtype can redefine methods it inherits, and it can also add new methods, including methods with the same name.

See the examples in "Creating Subtypes" on page 2-16 and Example 8–10 on page 8-17.

Overloading Methods

Adding new methods that have the same names as inherited methods to the subtype is called overloading. When they exist in the same user-defined type, methods that have the same name, but different signatures are called overloads. A method signature consists of the method's name and the number, types, and the order of the method's formal parameters, including the implicit `self` parameter.

Overloading is useful when you want to provide a variety of ways of doing something. For example, an ellipse object might overload a `calculate()` method with another `calculate()` method to enable calculation of a different shape.

The compiler uses the method signatures to determine which method to call when a type has several overloaded methods.

In the following pseudocode, subtype `circle_typ` creates an overload of `calculate()`:

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

```
CREATE TYPE circle_typ UNDER ellipse_typ (...,
  MEMBER PROCEDURE calculate(x NUMBER),
...);
```

The `circle_typ` contains two versions of `calculate()`. One is the inherited version with two `NUMBER` parameters and the other is the newly created method with one `NUMBER` parameter.

Overriding and Hiding Methods

Redefining an inherited method to customize its behavior in a subtype is called overriding, in the case of member methods, or hiding, in the case of static methods.

Unlike overloading, you do not create a new method, just redefine an existing one, using the keyword `OVERRIDING`.

Overriding and hiding redefine an inherited method to make it do something different in the subtype. For example, a subtype `circle_typ` derived from a `ellipse_typ` supertype might override a member method `calculate()` to customize it specifically for calculating the area of a circle. For examples of overriding methods, see "Creating Subtypes" on page 2-16.

Overriding and hiding are similar in that, in either case, the version of the method redefined in the subtype eclipses the original version of the same name and signature so that the new version is executed rather than the original one whenever a subtype instance invokes the method. If the subtype itself has subtypes, these inherit the redefined method instead of the original version.

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 is identified at compile time, and dynamic dispatch is not necessary. See "Dynamic Method Dispatch" on page 2-23.

To override or hide a method, you must preserve its signature. 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.

You signal the override with the `OVERRIDING` keyword in the `CREATE TYPE BODY` statement. This is not required when a subtype hides a static method.

In the following pseudocode, the subtype signals that it is overriding method `calculate()`:

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

CREATE TYPE circle_typ UNDER ellipse_typ (...,
  OVERRIDING MEMBER PROCEDURE calculate(),
...);
```

For a diagram of this hierarchy, see Figure 2–2 on page 2-23.

Restrictions on Overriding Methods

The following are restrictions on overriding methods:

- Only methods that are not declared to be final in the supertype can be overridden.

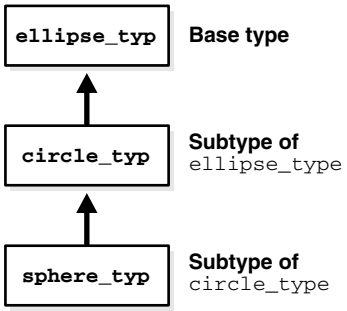
- Order methods may appear only in the root type of a type hierarchy: they may not be redefined (overridden) in subtypes.
- A static method in a subtype may not redefine a member method in the supertype.
- A member method in a subtype may not redefine a static method in the supertype.
- If a method being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

Dynamic Method Dispatch

Dynamic method dispatch refers to the way that method calls are dispatched to the nearest implementation at run time, working up the type hierarchy from the current or specified type. This feature is only available when overriding member methods and does not apply to static methods.

With method overriding, a type hierarchy can define multiple implementations of the same method. In the following hierarchy of types `ellipse_typ`, `circle_typ`, and `sphere_typ`, each type might define a `calculate()` method differently.

Figure 2–2 Hierarchy of Types



When one of these methods is invoked, the type of the object instance that invokes it determines 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.

The method call works up the type hierarchy: never down. 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.

> **See Also:** *Oracle Database PL/SQL Language Reference* for information on how subprograms calls are resolved

Substituting Types in a Type Hierarchy

When you work with types in a type hierarchy, sometimes you need to work at the most general level, for example, to select or update all persons. But at other times, you need to select or update only a specific subtype such as a student, 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 seems expected, given that a subtype is, after all, just a specialized kind of one 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.

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 and object-type rows in object tables are substitutable, and so are views: a column or row of a specific type can contain instances of that type and any of its subtypes.

For example, consider the person_typ type hierarchy such as the one introduced in Example 2–14. You can create an object table of person_typ that contains rows of all types. To do this, you insert an instance of a given type into an object table using the constructor for that type in the VALUES clause of the INSERT statement as shown in Example 2–20.

Similarly, Example 2–23 shows that a substitutable column of type person_typ can contain instances of all three types, in a relational table or view. The example recreates person, student, and part-time student objects from that type hierarchy and inserts them into the person_typ column contact.

Example 2–23 Inserting Values into Substitutable Columns of a Table

```
DROP TYPE person_typ FORCE;
-- if previously created

DROP TYPE student_typ FORCE; -- if previously created

DROP TYPE part_time_student_typ FORCE; -- if previously created
DROP TABLE contacts; if previously created
CREATE OR REPLACE TYPE person_typ AS OBJECT (
  idno           NUMBER,
  name           VARCHAR2(30),
  phone          VARCHAR2(20))
  NOT FINAL;
/
CREATE TYPE student_typ UNDER person_typ (
    dept_id NUMBER,
    major VARCHAR2(30))
    NOT FINAL;
/
CREATE TYPE part_time_student_typ UNDER student_typ (
  number_hours NUMBER);
/
CREATE TABLE contacts (
```

```
  contact         person_typ,
  contact_date    DATE );

INSERT INTO contacts
  VALUES (person_typ (12, 'Bob Jones', '650-555-0130'), '24 Jun 2003' );

INSERT INTO contacts
  VALUES (student_typ(51, 'Joe Lane', '1-650-555-0178', 12, 'HISTORY'),
         '24 Jun 2003' );

INSERT INTO contacts
  VALUES (part_time_student_typ(52, 'Kim Patel', '1-650-555-0190', 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.

In general, you can access attributes using dot notation. To access attributes of a subtype of a row or column's declared type, you can use the `TREAT` function. For example:

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

See "TREAT" on page 2-36.

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 object identifier (OID) of a substitutable row in an object table as shown in Example 2–24.

> **See Also:** *Oracle Database SQL Language Reference* for further information on these pseudocolumns

Example 2–24 Using OBJECT_VALUE and OBJECT_ID

```
DROP TABLE person_obj_table; -- required if previously created
CREATE TABLE person_obj_table OF person_typ;

INSERT INTO person_obj_table
  VALUES (person_typ(20, 'Bob Jones', '650-555-0130'));

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

Subtypes with Attributes of a Supertype

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

Example 2–25 Creating a Subtype with a Supertype Attribute

```
-- requires Ex 2-22
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 can have a collection attribute whose element type is one of its 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 2–26 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 2–26 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.

Substitution of 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.

Substitution of 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.

Storing Newly Created Subtypes in Substitutable Columns

If you create a subtype, any table that already has substitutable columns of the supertype can 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 do not violate table limits or constraints.

The following example creates a person_typ and then shows several attempts to create a subtype student_typ under person_typ.

Example 2–27 Creating a Subtype After Creating Substitutable Columns

```
DROP TYPE person_typ FORCE;
DROP TABLE person_obj_table;
DROP TYPE student_typ;
-- perform above drops if objects/tables created
CREATE OR REPLACE TYPE person_typ AS OBJECT (
  idno           NUMBER,
  name           VARCHAR2(30),
  phone          VARCHAR2(20))
  NOT FINAL;
/
```

```
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 next attempt succeeds. This version of the `student_typ` subtype is substitutable. Oracle Database 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', '1-650-555-0178', 12, 'HISTORY'));
```

Dropping Subtypes After Creating Substitutable Columns

When you drop a subtype with the `VALIDATE` option, it checks that no instances of the subtype are stored in any substitutable column of the supertype. If there are no such instances, the `DROP` operation completes.

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: an instance still exists ;
```

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;
```

> **See Also:** *Oracle Database PL/SQL Language Reference* for further information on `DROP` and `VALIDATE`

Turning Off Substitutability in a New Table

When you create 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 the column `office` of a relational table to storing only `office_typ` instances and disallows any subtype instances:

Example 2–28 Turning off Substitutability When Creating a Table

```
DROP TYPE location_typ FORCE; -- required if previously created
DROP TYPE office_typ FORCE; -- required if previously created
CREATE OR REPLACE TYPE location_typ AS OBJECT (
  building_no  NUMBER,
  city         VARCHAR2(40) );
/

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

```
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, such as:

```
DROP TABLE office_tab; -- if previously created
CREATE TABLE office_tab OF office_typ
  NOT SUBSTITUTABLE AT ALL LEVELS;
```

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

```
DROP TABLE office_tab; -- if previously created
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 such as the following:

```
DROP TABLE people_tab;
-- required if previously created
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.

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

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.

The following statement creates a table of `office_typ` in which occupants are constrained to just those persons who are employees:

Example 2–29 Constraining Substitutability When Creating a Table

```
DROP TABLE office_tab;
-- if previously created
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, not for an object table as a whole.

The following statement makes the column office substitutable:

Example 2–30 Modifying Substitutability in a Table

```
-- Requires Ex. 2-28
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 you do not use the FORCE keyword to make a column 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 Object 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 only modify substitutability for a column if substitutability was not explicitly set at the table level, 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:

```
DROP TABLE office_tab;
-- if previously created
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:

```
DROP TABLE office_tab;
-- if previously created
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.

Typical Object to Object Assignment

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

Assigning 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.

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, which first requires you to create a `person_typ` and an `employee_typ`:

Example 2–31 PL/SQL Assignment

```
DROP TYPE person_typ FORCE;
-- if previously created
```

```
CREATE TYPE person_typ AS OBJECT (
 idno           NUMBER,
 name           VARCHAR2(30),
 phone          VARCHAR2(20))
 NOT FINAL;
/
DROP TYPE employee_typ FORCE; -- if previously created
CREATE TYPE employee_typ UNDER person_typ (
   emp_id NUMBER,
   mgr VARCHAR2(30));
/
-- PL/SQL assignment example
DECLARE
  var1 person_typ;
  var2 employee_typ;
BEGIN
  var2 := employee_typ(55, 'Jane Smith', '1-650-555-0144', 100, 'Jennifer
Nelson');
  var1 := var2;
END;
/
```

Narrowing Assignment

A narrowing assignment 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. Thus, in the end, narrowing assignments only work in cases such as Case 1, described in "Typical Object to Object Assignment" on page 2-30.

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 if that person is also an employee. If the person 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 "Using TREAT for Narrowing Assignments" on page 2-36.

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, after creating a new `student_typ`, suppose we have the following collection types:

Example 2–32 Create Collection person_set

```
-- Requires 2-21
DROP student_typ;
-- if previously created
CREATE TYPE student_typ UNDER person_typ (
    dept_id NUMBER,
    major VARCHAR2(30))
  NOT FINAL;
/
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 `person_set` 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;
/
```

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 2–33 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, *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 2–34 Using the DEREF Function

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

See "Dereferencing REFs" on page 1-9. 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 2–35 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', '1-650-555-0144', 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 (or table alias) 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 2–36 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 (a hidden type) of the most specific type of the object instance passed as an argument.

The most specific type of an object instance is the type that the instance belongs to, that is, the farthest removed instance 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 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.

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 2–20 on page 2-19.

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 2–37 Using the SYS_TYPEID Function

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

See "Hidden Columns for Substitutable Columns and Object 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 Database to return the results of a table function iteratively. A table function returns a nested table or varray 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 Language 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 declared type of the expression. In other words, the function attempts to treat a supertype instance as a subtype instance, for example, to treat a person as a 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 is not 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: that is, 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 contain subtype attribute data, but you cannot list them with a DESCRIBE statement. TREAT enables you to access these columns.

Using TREAT for Narrowing Assignments

The TREAT function is used for narrowing assignments, that is, assignments that set a supertype value into a subtype. For a comparison to widening assignments, see "Assignments Across Types" on page 2-30.

In Example 2–38, 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 2–38 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.

Using the TREAT Function to Access Subtype Attributes or Methods

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 2–39 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', '1-650-555-0144', 100, 'Jennifer Nelson');
  DBMS_OUTPUT.PUT_LINE(TREAT(var AS employee_typ).mgr);
END;
/
```

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.

Example 2–40 first create a part_time_student_typ, and then shows a SELECT query returning all persons, including students and employees, from table person_obj_table of person_typ.

Example 2–40 Using the VALUE Function

```
-- Requires Ex. 2-31 and 2-32
CREATE TYPE part_time_student_typ UNDER student_typ (
  number_hours NUMBER);
/
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', '1-650-555-0130')
   WHERE p.idno = 12;
```

See also Example 5–22, "Using VALUE to Update a Nested Table" on page 5-16. For more information about the SQL `VALUE` function, see *Oracle Database SQL Language Reference*.

3

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
- Manipulating Objects in PL/SQL
- Using Overloading in PL/SQL with Inheritance
- Using Dynamic SQL With Objects

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.

   See Also: "About Object Types" on page 1-3

   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 user-defined type or ADT that you just defined.

Objects or ADTs follow the usual scope and instantiation rules.

Defining Object Types

Example 3–1 provides two object types, and a table of object types. Subsequent examples show how to declare variables of those object types in PL/SQL and perform other operations with these objects.

Example 3–1 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.

Example 3–2 declares object `emp` of type `employee_typ`. Then, the constructor for object type `employee_typ` initializes the object.

Example 3–2 Declaring Objects in a PL/SQL Block

```
-- Requires Ex. 3-1
DECLARE
  emp employee_typ; -- emp is atomically null
BEGIN
-- call the constructor for employee_typ
  emp := employee_typ(315, 'Francis', 'Logan', 'FLOGAN',
        '415.555.0100', '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 can have data types of user-defined types. Therefore, you can pass objects to stored subprograms and from one subprogram to another. In the next example, the object type `employee_typ` specifies the data type of a formal parameter:

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

In the following example, object type `employee_typ` specifies the return type of a function:

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

How PL/SQL Treats Uninitialized Objects

User-defined types, 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`.

See Example 2–1 on page 2-2 for an illustration of null objects and objects with null attributes.

Manipulating Objects in PL/SQL

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

This section includes the following topics:

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.

Example 3–3 uses dot notation and generates the same output as Example 3–2.

Example 3–3 Accessing Object Attributes

```
-- Requires Ex. 3-1
DECLARE
  emp employee_typ;
BEGIN
  emp := employee_typ(315, 'Francis', 'Logan', 'FLOGAN',
        '415.555.0100', '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 3–3 on page 3-3 and Example 3–4.

Example 3–4 Inserting Rows in an Object Table

```
-- Requires Ex. 3-1
DECLARE
  emp employee_typ;
BEGIN
  INSERT INTO employee_tab VALUES (employee_typ(310, 'Evers', 'Boston', 'EBOSTON',
   '617.555.0100', '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',
    '650.555.0150', '30-SEP-04', 'AC_MGR', 12500, 0, 101, 110,
    address_typ('123 Broadway', 'Redwood City', 'CA', '94065')) );
END;
/
SELECT VALUE(e) from employee_tab e;
```

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 3–5, 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 3–5 Accessing Object Methods

```
-- Requires Ex. 3-1 and Ex. 3-4
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. Also, if you chain two function calls, the first function must return an object that can be passed to the second function.

If a PL/SQL function is used in place of an ADT constructor during a DML operation, the function may execute multiple times as part of the DML execution. For the function to execute only once per occurrence, it must be a deterministic 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 declarations in the type hierarchy. If the subtype overrides the method that it inherits from its supertype, the call uses the subtype implementation.

Otherwise, the call uses the supertype implementation. This capability is known as dynamic method dispatch.

See Also: "Dynamic Method Dispatch" on page 2-23

Updating and Deleting Objects

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

Example 3–6 Updating and Deleting Rows in an Object Table

```
-- Requires Ex. 3-1 and 3-4
DECLARE
  emp employee_typ;
BEGIN
  INSERT INTO employee_tab VALUES (employee_typ(370, 'Robert', 'Myers', 'RMYERS',
   '415.555.0150', '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;
/
SELECT VALUE(e) from employee_tab e;
```

Manipulating Objects Through Ref Modifiers

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

Example 3–7 Updating Rows in an Object Table With a REF Modifier

```
-- Requires Ex. 3-1, 3-4, and 3-6
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 PL/SQL. For example, the assignment in Example 3–8 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 3–8 Trying to Use DEREF in a SELECT INTO Statement, Incorrect

```
-- Requires Ex. 3-1, 3-4, and 3-6
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 := DEREF(emp_ref);  --  cannot use DEREF in procedural statements
  emp_name := emp.first_name || ' ' || emp.last_name;
  DBMS_OUTPUT.PUT_LINE(emp_name);
END;
/
```

This assignment raises an error as described below:

```
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
```

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

Using Overloading in PL/SQL with Inheritance

Overloading allows you to substitute a subtype value for a formal parameter that is a supertype. This capability is known as substitutability. This section is about this aspect of overloading.

Rules of Substitution

If more than one instance of an overloaded procedure matches the procedure call, the following substitution rules determine which procedure, if any, is called:

- If the signatures of the overloaded procedures only differ in 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.
- If instances of two overloaded methods match, and some argument types are closer in one overloaded procedure while others are closer in the second procedure, a semantic error occurs.
- If some parameters differ in their position within the object type hierarchy, and other parameters are of different data types so that an implicit conversion would be necessary, then a semantic error occurs.

Example 3–9 creates a type hierarchy that has three levels starting with `super_t`. There is a package with two overloaded instances of a function that are the same except for the position of the argument type in the type hierarchy. The invocation declares a variable of type `final_t`, and then calls the overloaded function.

The instance of the function that executes is the one that accepts a `sub_t` parameter, because `sub_t` is closer to `final_t` than `super_t` in the hierarchy. This follows the rules of substitution.

Note that because determining which instance to call happens at compile time, the fact that the argument passed in was also a `final_t` is ignored. If the declaration was `v super_t := final_t(1,2,3)`, the overloaded function with the argument `super_t` would be called.

Example 3–9 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 3–10, determining which instance to call happens at run time because the functions are overriding member functions of the type hierarchy. This is dynamic method dispatch, described in "Dynamic Method Dispatch" on page 2-23.

Though `v` is an instance of `super_t`, because the value of `final_t` is assigned to `v`, the `sub_t` instance of the function is called, following the rules of substitution.

Example 3–10 Resolving PL/SQL Functions With Inheritance Dynamically

```
-- Perform the following drop commands if you created these objects in Ex. 3-9
-- DROP PACKAGE p;
-- DROP TYPE final_t;
-- DROP TYPE _sub_t;
-- DROP TYPE super_t FORCE;
CREATE OR REPLACE TYPE super_t AS OBJECT
  (n NUMBER, MEMBER FUNCTION func RETURN NUMBER) NOT final;
/
CREATE OR REPLACE TYPE BODY super_t AS
 MEMBER FUNCTION func RETURN NUMBER IS BEGIN RETURN 1; END; END;
/
CREATE TYPE sub_t UNDER super_t
  (n2 NUMBER,
   OVERRIDING MEMBER FUNCTION func RETURN NUMBER) NOT final;
/
CREATE OR REPLACE 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('answer:'|| v.func); -- prints 2
END;
```

```
/
```

Using Dynamic SQL With Objects

Dynamic SQL is a feature of PL/SQL that enables you to enter SQL information at run time, such as: a table name, the full text of a SQL statement, or variable information.

See Also: *Oracle Database PL/SQL Language Reference*

Example 3–11 illustrates the use of objects and collections with dynamic SQL. First, the example defines the object type `person_typ` and the `VARRAY` type `hobbies_var`, then it defines the package, `teams`, that uses these types. You need `AUTHID CURRENT_USER` to execute dynamic package methods; otherwise, these methods raise an insufficient privileges error when you run Example 3–12.

Example 3–11 A Package that Uses Dynamic SQL for Object Types and Collections

```
CREATE OR REPLACE 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 3–12 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;
/
```

4

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. From there, you can use many Oracle application programming environments and tools that have built-in support for Oracle Objects.

This chapter discusses the following topics:

SQL and Object Types

Oracle SQL data definition language (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*

SQL Developer

SQL Developer provides a visual development environment for database developers and DBAs to create and manipulate database schema objects including Oracle Objects, and to run reports, monitor performance and perform many other database-related tasks using a rich graphical user interface.

> **See Also:** *Oracle Database Oracle SQL Developer User's Guide*

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 Language 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 (`REF`s), and collections as input variables in SQL statements
- Return object instances, `REF`s, and collections as output of SQL statement fetches
- Describe the properties of SQL statements that return object instances, `REF`s, 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 4-5.

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
- Dereference 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 4-6.

Object Cache

To support high-performance navigational access of objects, OCI runtime provides an object cache for caching objects in memory. The object cache supports references (`REF`s) 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 `REF`s to objects. When an application dereferences 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 dereferences 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 4–1 shows how to define a user-defined constructor in C.

Example 4–1 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 4-2), 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 4-3.

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

- Support for transient copies of objects allocated in the object cache
- Support for transient copies of objects referenced as input host variables in embedded SQL `INSERT`, `UPDATE`, and `DELETE` statements, or in the `WHERE` clause of a `SELECT` statement
- Support for transient copies of objects referenced as output host variables in embedded SQL `SELECT` and `FETCH` statements

- Support for ANSI dynamic SQL statements that reference object types through the `DESCRIBE` statement, to get the object's type and schema information

Navigational Access in Pro*C/C++

For background information on navigational access, see "Navigational Access in OCI Programs" on page 4-3.

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

- Support for dereferencing, 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)

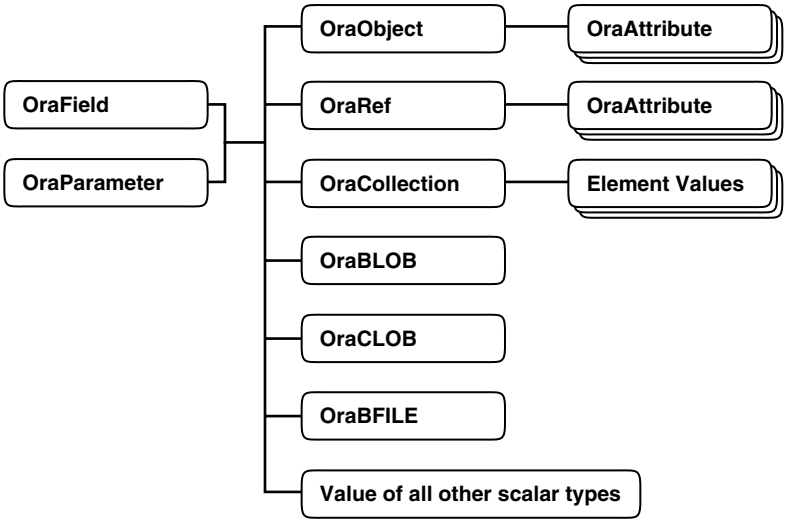
Oracle Objects for OLE (OO4O) provides full support for accessing and manipulating instances of `REF`s, value instances, variable-length arrays (`VARRAY`s), and nested tables in an Oracle database server.

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 "Server Objects" chapter of the Oracle Objects for OLE *Oracle Objects for OLE Developer's Guide* for detailed information and examples on using OO4O with Oracle objects

Figure 4–1 illustrates the containment hierarchy for value instances of all types in OO4O.

Figure 4–1 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 Tools for Accessing Oracle Objects

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 various ways to integrate Oracle object features with Java. These interfaces enable you both to access SQL data from Java and to provide persistent database storage for Java objects.

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 (REFs), 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).

JPublisher

Oracle 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*

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 *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 4–2 Mapping SQL Types to Java Classes

```
-- Mapping SQL Types to Java Classes example, not sample schema
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 8-6.

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 4–3 is an example of a type declaration that involves a user-defined constructor implemented in Java.

Example 4–3 Defining a User-Defined Constructor in Java

```
-- Defining a User-Defined Constructor in Java example, not sample schema
```

```
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')
/
DROP TYPE person1_typ FORCE;
DROP TYPE person_t FORCE;
DROP TYPE full_address FORCE;
DROP TYPE person FORCE;
DROP LIBRARY person_lib;
SPOOL OFF
COMMIT;
```

JDeveloper

Oracle JDeveloper is a full-featured, cross-platform, integrated development environment for creating multitier Java applications that is well integrated with Oracle Application Server and Database.

Oracle JDeveloper enables you to develop, debug, and deploy Java client applications, dynamic HTML applications, web and application server components, JavaBean components, and database stored procedures based on industry-standard models.

JDeveloper is also the integrated development environment for ADF and TopLink.

Application Development Framework (ADF)

ADF is a framework for building scalable enterprise Java EE applications. Developers can use ADF to build applications where the application data is persisted to Oracle Object tables as well as other schema objects.

TopLink

TopLink is a framework for mapping Java objects to a variety of persistence technologies, including databases, and provides facilities to build applications leveraging Oracle Objects.

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

Utilities Providing Support for Objects

This section describes several Oracle utilities that provide support for Oracle objects.

This section contains these topics:

Import/Export of Object Types

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.

> **See Also:** *Oracle Database Utilities* for instructions on how to use the Import and Export utilities

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 is created with compilation errors, which may be ignored. The type is 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 generally provides 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

5

Support for Collection Data Types

This chapter explains how to create and manage these collection types: varrays and nested tables.

> **See Also:** *Oracle Database PL/SQL Language Reference* for a complete introduction to collections

This chapter contains these topics:

- Collection Data Types
- Multilevel Collection Types
- Operations on Collection Data Types
- Partitioning Tables That Contain Oracle Objects

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 and is unordered.

If you need to store only a fixed number of items, or loop through the elements in order, or often need 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.

This section includes the following topics:

- Creating a Collection Type
- Creating a Collection Type
- Creating an Instance of a VARRAY or Nested Table
- Constructor Methods for Collections
- Varrays
- Nested Tables
- Increasing the Size and Precision of VARRAY and Nested Table Elements
- Increasing VARRAY Limit Size

- Creating a Varray Containing LOB References

Creating a Collection Type

This section shows the creation of a nested table type. Creation of a `VARRAY` type is demonstrated in "Varrays" on page 5-3.

Example 5–1 demonstrates creating a `person_typ` object and a `people_typ` as a nested table type of `person_typ` objects, which are both used in subsequent examples in this chapter.

Example 5–1 CREATE TYPE person_typ for Subsequent Examples

```
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 TYPE people_typ AS TABLE OF person_typ; -- nested table type
/
```

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 constructor method of the type. The name of a 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, for example.

```
( person_typ(1, 'John Smith', '1-650-555-0135'),
```

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

You can use a constructor method in a SQL statement to insert values into a nested table.

Example 5–2 first creates a table that contains an instance of the nested table type `people_typ`, named `people_column`, and then shows how to use the constructor

method in a SQL statement to insert values into people_typ. This example uses a literal invocation of the constructor method.

Example 5–2 Using the Constructor Method to Insert Values into a Nested Table

```
-- Requires Ex. 5-1
CREATE TABLE people_tab (
    group_no NUMBER,
    people_column people_typ )  -- an instance of nested table
    NESTED TABLE people_column STORE AS people_column_nt; -- storage table for NT

INSERT INTO people_tab VALUES (
            100,
            people_typ( person_typ(1, 'John Smith', '1-650-555-0135'),
                        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 5–3 shows how to use literal invocations of constructor methods to specify defaults for the person_typ object and the people_typ nested table:

Example 5–3 Creating the department_persons Table Using the DEFAULT Clause

```
-- requires Ex. 5-1
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() ) -- instance of nested table type
  NESTED TABLE dept_emps STORE AS dept_emps_tab;

INSERT INTO department_persons VALUES
   ( 101, 'Physical Sciences', person_typ(65,'Vrinda Mills', '1-650-555-0125'),
           people_typ( person_typ(1, 'John Smith', '1-650-555-0135'),
                       person_typ(2, 'Diane Smith', NULL) ) );
INSERT INTO department_persons VALUES
  ( 104, 'Life Sciences', person_typ(70,'James Hall', '1-415-555-0101'),
    people_typ() ); -- an empty people_typ table
```

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

The department_persons table can be queried in two ways as shown in Example 5–16 and Example 5–17.

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 position of the element 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.

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);
/
```

Example 5–4 creates a VARRAY type 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 by invoking the constructors for the varray phone_varray_typ and the object phone_typ.

Example 5–4 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', '5550123'),
                     phone_typ ('01', '650', '5550148'),
                     phone_typ ('01', '650', '5550192')));
```

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 inline, 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 LOB type for procedural purposes, such as in PL/SQL or view queries. However, database storage for varrays of these types is not supported. Thus you cannot create an object table or an object type column of a varray type of XMLType or 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.

A nested table can be viewed as a single column. 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.

To declare nested table types, use the `CREATE TYPE ... AS TABLE OF` statement. For example:

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

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.

Storing Elements of Nested Tables

Elements of a nested table are actually stored in a separate storage table.

Oracle stores nested table data in a single storage table associated with the object table for both nested table types that are columns in a relational table or attributes in an object table. The storage table contains a column that identifies the parent table row or object that each element of the nested table belongs to. See Figure 9–2, "Nested Table Storage" on page 9-11.

The `NESTED TABLE..STORE AS` clause specifies storage names for nested tables. Storage names are used to create an index on a nested table.

Example 5–5 demonstrates creating and populating a nested table, and specifying the nested table storage using the `person_typ` object and the `people_typ` nested table as defined in Example 5–1.

Example 5–5 Creating and Populating Simple Nested Tables

```
-- Requires 5-1
CREATE TABLE students (
   graduation DATE,
   math_majors people_typ, -- nested tables (empty)
   chem_majors people_typ,
   physics_majors people_typ)
  NESTED TABLE math_majors STORE AS math_majors_nt  -- storage tables
  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', '650-555-0130'),
                    person_typ(31, 'Sarah Chen', '415-555-0120'),
                    person_typ(45, 'Chris Woods', '415-555-0124')),
      chem_majors =
        people_typ (person_typ(51, 'Joe Lane', '650-555-0140'),
                    person_typ(31, 'Sarah Chen', '415-555-0120'),
                    person_typ(52, 'Kim Patel', '650-555-0135')),
   physics_majors =
        people_typ (person_typ(12, 'Bob Jones', '650-555-0130'),
```

```
                    person_typ(45, 'Chris Woods', '415-555-0124'))
WHERE graduation = '01-JUN-03';

SELECT m.idno math_id, c.idno chem_id, p.idno physics_id  FROM students s,
 TABLE(s.math_majors) m, TABLE(s.chem_majors) c, TABLE(s.physics_majors) p;
```

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 5-12.

Specifying a Tablespace When Storing a Nested Table

A nested table can be stored in a different tablespace than its parent table. In Example 5–6, the nested table is stored in the system tablespace:

Example 5–6 Specifying a Different Tablespace for Storing a Nested Table

```
-- Requires Ex. 5-1, must remove code in Ex. 5-2 if created
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 immediately preceding parent table.

You can issue an ALTER TABLE.. MOVE statement to move a table to a different tablespace. If you do this on a table with nested table columns, only the parent table moves; no action is taken on the storage tables of the nested table. To move a storage table for a nested table to a different tablespace, issue ALTER TABLE.. MOVE on the storage table. For example:

```
ALTER TABLE people_tab MOVE TABLESPACE system;  -- moving table
ALTER TABLE people_column_nt MOVE TABLESPACE example; -- moving storage table
```

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

Increasing the Size and Precision of VARRAY and Nested Table Elements

When the element type of a VARRAY type or nested table type is a variable character, or a RAW or 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 or nested table type.

You make these changes using an ALTER TYPE..MODIFY statement, which has these options:

- INVALIDATE: Invalidates all dependent objects
- CASCADE: Propagates the change to its type and table dependents

> **See Also:** "ALTER TYPE Statement for Type Evolution" on page 8-14 for further description of INVALIDATE and CASCADE

Example 5–7 increases the sizes of a VARRAY and a nested table element type.

Example 5–7 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 you to increase the number of elements of a VARRAY type. If the number is increased, a new type version is generated for the VARRAY type, and becomes part of the type change history.

The ALTER TYPE ... MODIFY LIMIT statement has these options:

- INVALIDATE: Invalidates all dependent objects
- CASCADE: Propagates the change to its type and table dependents

Example 5–8 Increasing the VARRAY Limit Size

```
-- if you have already creating following types, drop them.
DROP TYPE email_list_tab FORCE;
DROP TYPE email_list_arr FORCE;
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

To create a varray of LOB references, first define a VARRAY type of type REF email_list_typ. Note: email_list_typ was defined in Example 5–8. Next, create a table dept_email_list and define a column email_addrs of the array type in it.

Example 5–9 Creating a VARRAY Containing LOB References

```
-- Requires Ex. 5-8
CREATE TYPE ref_email_varray_typ AS VARRAY(5) OF REF email_list_typ;
/

CREATE TABLE dept_email_list (
  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 single-level collection types, multilevel collection types:

- Can be used as columns in a relational table or with object attributes in an object table.
- Require that both the source and the target be of the same declared data type for assignment.

This section contains the following topics:

Nested Table Storage Tables for Multilevel Collection Types

A nested table type column or object table attribute requires a storage table to store rows for all its nested tables as described in "Storing Elements of Nested Tables" on page 5-5. With a multilevel nested table collection of nested tables, you must specify nested-table storage clauses for both the inner set and the outer set of nested tables.

> **See Also:** "Unnesting Queries with Multilevel Collections" on page 5-14

Example 5–10 creates the multilevel collection type `nt_country_typ`, 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 requires the `regions`, `countries`, and `locations` tables of the Oracle `HR` sample schema.

> **See Also:** *Oracle Database Sample Schemas* for information on using sample schemas

In Example 5–10, the SQL statements create the table `region_tab`, which contains the column `countries`, whose type is a multilevel collection, `nt_country_typ`. This multilevel collection is a nested table of an object type that has the nested table attribute `locations`. Separate nested table clauses are provided for the outer `countries` nested table and for the inner `locations` nested table.

Example 5–10 Multilevel Nested Table Storage

```
-- Requires the HR sample schema
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;  -- nested table type
/

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

CREATE TYPE nt_country_typ AS TABLE OF country_typ;  -- multilevel collection type
/

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

In Example 5–10 you can refer to the inner nested table `locations` by name because this nested table is a named attribute of an object. However, *if the inner nested table is not an attribute of an object, it has no name*. The keyword `COLUMN_VALUE` is provided for this case.

Example 5–11 shows keyword `COLUMN_VALUE` used in place of a name for an inner nested table.

Example 5–11 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,  -- inner nested table, unnamed
  col2 outer_table)
NESTED TABLE col2 STORE AS col2_ntab
  (NESTED TABLE COLUMN_VALUE STORE AS cv_ntab);
```

Example 5–12 shows how to specify physical attributes for the storage tables in the nested table clause.

Example 5–12 Specifying Physical Attributes for Nested Table Storage

```
-- Requires Ex. 5-10
-- drop the following if you have previously created it
DROP TABLE region_tab FORCE;

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:

- A system-supplied ID column that is referenceable by NESTED_TABLE_ID, which keys its rows back to rows in its parent table.
- A system-supplied ID column that is hidden and referenced by the NESTED_TABLE_ID column in its nested table children.

In Example 5–12, 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 causes Oracle database to physically cluster all the nested table rows that belong to the same parent row, for more efficient access.

If you do not specify a primary key with a NESTED_TABLE_ID column, then the database automatically creates a b-tree index on the NESTED_TABLE_ID column for better performance.

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.

Varray Storage for Multilevel Collections

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.

You can explicitly specify LOB storage for varrays. The following example does this for a nested table of varray elements.

Example 5–13 shows explicit LOB storage specified for a varray of varray type.

Example 5–13 Specifying LOB Storage for a VARRAY of VARRAY Type

```
-- Requires Ex. 5-8, drop following if created

DROP TYPE email_varray_typ FORCE;
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*.

Example 5–14 Specifying LOB Storage for a Nested Table of VARRAYs

```
-- drop the following types if you have created them
DROP TYPE email_list_typ FORCE;
DROP TABLE dept_email_list FORCE;
DROP TYPE email_list_arr FORCE;

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 5–14 shows the `COLUMN_VALUE` keyword used with varrays. See Example 5–11 for discussion of this keyword and its use with nested tables.

Constructors for Multilevel Collections

Multilevel collection types are created by calling the constructor of the respective type, just like single-level collections and other object types. The 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. Constructor parameters have the names and types of the attributes of the object type.

Example 5–15 shows the constructor call for the multilevel collection type `nt_country_typ`. The `nt_country_typ` constructor calls the `country_typ` constructor, which calls the `nt_location_typ`, which calls the `location_typ` constructor.

> **Note:** `nt_country_typ` is a multilevel collection because it is a nested table that contains another nested table as an attribute.

Example 5–15 Using Constructors for Multilevel Collections

```
-- Requires 5-10 and HR sample schema
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.

This section contains the following topics:

- Querying Collections
- Performing DML Operations on Collections
- Using BULK COLLECT to Return Entire Result Sets
- Conditions that Compare Nested Tables
- Multiset Operations for Nested Tables

Querying Collections

There are two general ways to query a table that contains a collection type as a column or attribute.

- Nest the collections in the result rows that contain them.
- Distribute or unnest collections so 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 5–3 on page 5-3. 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 this way nests the elements of the collection in the result row that the collection is associated with.

Example 5–16 shows the query retrieving the nested collection of employees.

Example 5–16 Nesting Results of Collection Queries

```
-- Requires Ex. 5-1 and Ex. 5-3
SELECT d.dept_emps
  FROM department_persons d;

DEPT_EMPS(IDNO, NAME, PHONE)
------------------------------------------------------------
PEOPLE_TYP(PERSON_TYP(1, 'John Smith', '1-650-555-0135'),
PERSON_TYP(2, 'Diane Smith', '1-650-555-0135'))
```

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` produces a nested result.

Unnesting Results of Collection Queries

Not all tools or applications can 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 using a TABLE expression with the collection. TABLE expressions enable 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.

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

> **See Also:** *Oracle Database SQL Language Reference* for further information on the TABLE expression and unnesting collections

The query in Example 5–17, like that of Example 5–16, retrieves the collection of employees, but here the collection is unnested.

Example 5–17 Unnesting Results of Collection Queries

```
-- Requires Ex. 5-1 and 5-3
SELECT e.*
  FROM department_persons d, TABLE(d.dept_emps) e;

 IDNO        NAME                              PHONE
---------- ------------------------------ ---------------
         1 John Smith                     1-650-555-0135
         2 Diane Smith                    1-650-555-0135
```

Example 5–17 shows that a TABLE expression can have its own table alias. 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. To reference a table column, a TABLE expression can use the table alias of any table appearing to the left of it in a FROM clause. This 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 with or without 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 5-13 show a TABLE expression that contains the name of a collection. Alternatively, a TABLE expression can contain a subquery of a collection.

Example 5–18 returns the collection of employees whose department number is 101.

Example 5–18 Using a Table Expression Containing a Subquery of a Collection

```
-- Requires Ex. 5-1 and 5-3
SELECT *
  FROM TABLE(SELECT d.dept_emps
               FROM department_persons d
               WHERE d.dept_no = 101);
```

Subqueries in a TABLE expression have these restrictions:

- 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 5–19 shows a TABLE expression used in the FROM clause of a SELECT embedded in a CURSOR expression.

Example 5–19 Using a Table Expression in a CURSOR Expression

```
-- Requires Ex. 5-1 and 5-3
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 also used with multilevel collections, both varrays and nested tables. Example 5–20 shows an unnesting query on a multilevel nested table collection of nested tables. From the table region_tab where 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 5–20 Unnesting Queries with Multilevel Collections Using the TABLE Function

```
-- Requires Ex. 5-10 and 5-15
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;
```

The output should be as follows:

```
REGION_NAME              COUNTRY_NAME                              LOCATION_ID
------------------------ ---------------------------------------- -----------
Europe                   Italy                                            1000
Europe                   Italy                                            1100
Europe                   Switzerland                                      2900
```

```
Europe                    Switzerland                                3000
Europe                    United Kingdom                             2400
Europe                    United Kingdom                             2500
Europe                    United Kingdom                             2600

7 rows selected.

LOCATION_ID CITY
----------- ------------------------------
       1000 Roma
       1100 Venice
       2900 Geneva
       3000 Bern
       2400 London
       2500 Oxford
       2600 Stretford

7 rows selected.
```

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 collections:

- Inserts and updates that provide a new value for the entire collection
- Individual or piecewise updates of nested tables and multilevel nested tables, including inserting, deleting, and updating elements

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

This section contains these topics:

- Piecewise Operations on Nested Tables
- Piecewise Operations on Multilevel Nested Tables
- Atomical Changes on VARRAYs and Nested Tables
- Collections as Atomic Data Items

Piecewise Operations on Nested Tables

For piecewise operations on nested table columns, use the `TABLE` expression.

The `TABLE` expression uses a subquery to extract the nested table, so that the `INSERT`, `UPDATE`, or `DELETE` statement applies to the nested table rather than the top-level table.

`CAST` operators are also helpful. With them, you can do set operations on nested tables using SQL notation, without actually storing the nested tables in the database.

See Also:

- *Oracle Database SQL Language Reference*
- "CAST" on page 2-32

The DML statements in Example 5–21 demonstrate piecewise operations on nested table columns.

Example 5–21 Piecewise Operations on Collections

```
-- Requires Ex. 5-1 and 5-3
INSERT INTO TABLE(SELECT d.dept_emps
                  FROM department_persons d
                  WHERE d.dept_no = 101)
   VALUES (5, 'Kevin Taylor', '1-408-555-0199');

UPDATE TABLE(SELECT d.dept_emps
                  FROM department_persons d
                  WHERE d.dept_no = 101) e
   SET VALUE(e) = person_typ(5, 'Kevin Taylor', '1-408-555-0199')
   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 5–22 shows `VALUE` used to return object instance rows for updating:

Example 5–22 Using VALUE to Update a Nested Table

```
-- Requires Ex. 5-1, 5-3
UPDATE TABLE(SELECT d.dept_emps FROM department_persons d
               WHERE  d.dept_no = 101) p
   SET VALUE(p) = person_typ(2, 'Diane Smith', '1-650-555-0148')
   WHERE p.idno = 2;
```

Piecewise Operations on Multilevel Nested Tables

Piecewise DML is possible only on multilevel nested tables, not on multilevel varrays. You can perform DML operation atomically on both VARRAYs and nested tables multilevel collections as described in "Collections as Atomic Data Items" on page 5-17.

Example 5–23 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 5–23 Piecewise INSERT on a Multilevel Collection

```
-- Requires Ex. 5-10 and 5-15
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 5–24 performs a piecewise insert into an inner nested table to add a location for a country. Like the preceding example, this example uses a `TABLE` expression containing a subquery that selects the inner nested table to specify the target for the insert.

Example 5–24 Piecewise INSERT into an Inner Nested Table

```
-- Requires Ex. 5-10 and 5-15
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;
```

Atomical Changes on VARRAYs and Nested Tables

This section discusses atomical changes to nested tables and VARRAYs.

Note: While nested tables can also be changed in a piecewise fashions, varrays cannot.

Example 5–25 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 collections, update a row to replace its collection, and select collections into PL/SQL variables.

However, 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. You can also do this with nested tables, but nested tables have the option of doing piecewise updates and deletes.

Example 5–25 Using INSERT, UPDATE, DELETE, and SELECT Statements With Varrays

```
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;
/
```

Collections as Atomic Data Items

The section "Constructors for Multilevel Collections" on page 5-11 shows how to insert an entire multilevel collection with an INSERT statement. Multilevel collections (both VARRAY and nested tables) 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 5–26 updates region_tab by setting the countries collection as a unit to the value of v_country.

Example 5–26 Using UPDATE to Insert an Entire Multilevel Collection

```
-- Requires Ex. 5-10 and 5-15
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;
/
-- Invocation:
SELECT r.region_name, c.country_name, l.location_id
  FROM region_tab r, TABLE(r.countries) c, TABLE(c.locations) l
     WHERE r.region_id = 2;
```

Using BULK COLLECT to Return Entire Result Sets

The PL/SQL `BULK COLLECT` clause is an alternative to using DML statements, which can be time consuming to process. You can return an entire result set in one operation.

In Example 5–27, `BULK COLLECT` is used with a multilevel collection that includes an object type.

Example 5–27 Using BULK COLLECT with Collections

```
-- unrelated to other examples in this chapter
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;
/
```

Conditions that Compare Nested Tables

The conditions listed in this section allow comparisons of nested tables, including multilevel nested tables. There is no mechanism for comparing varrays. The SQL examples in this section use the nested tables created in Example 5–5 on page 5-5, and contain the objects created in Example 5–1 on page 5-2.

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. Equality is determined in the existing order of the elements, because nested tables are unordered.

Example 5–28 Using an Equality Comparison with Nested Tables

```
-- Requires Ex. 5-1 and 5-5
SELECT p.name
  FROM students, TABLE(physics_majors) p
WHERE math_majors = physics_majors;
```

In Example 5–28, the nested tables contain `person_typ` objects, which have an associated map method. See Example 5–1 on page 5-2. Since the two nested tables in the `WHERE` clause are not equal, no rows are selected.

IN Comparison

The `IN` condition checks whether or not 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 5–29 Using an IN Comparison with Nested Tables

```
-- Requires Ex. 5-1 and 5-5
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 or not a nested table is a subset of another nested table, returning the result as a Boolean value. The `OF` keyword is optional and does not change the functionality of `SUBMULTISET`.

This condition is implemented only for nested tables.

See Also: "Multiset Operations for Nested Tables" on page 5-20

Example 5–30 Testing the SUBMULTISET OF Condition on a Nested Table

```
-- Requires Ex. 5-1 and 5-5
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 or not 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.

In Example 5–31, the `person_typ` is an element of the same type as the elements of the nested table `math_majors`.

Example 5–31 Using MEMBER OF on a Nested Table

```
-- Requires Ex. 5-1 and 5-5
SELECT graduation
  FROM students
WHERE person_typ(12, 'Bob Jones', '1-650-555-0130') MEMBER OF math_majors;
```

Example 5–32 presents an alternative approach to the `MEMBER OF` condition, which performs more efficiently for large collections.

Example 5–32 Alternative to Using MEMBER OF on a Nested Table

```
-- Requires Ex. 5-1 and 5-5
SELECT graduation
  FROM students
WHERE person_typ(12, 'Bob Jones', '1-650-555-0130') in (select value(p)
    from TABLE( math_majors) p);
```

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 5–33 Using IS NOT on a Nested Table

```
-- Requires Ex. 5-1 and 5-5
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 or not a given nested table is composed of unique elements, returning a Boolean value.

Example 5–34 Using IS A SET on a Nested Table

```
-- Requires Ex. 5-1 and 5-5
SELECT p.idno, p.name
  FROM students, TABLE(physics_majors) p
WHERE physics_majors IS A SET;
```

Multiset Operations for Nested Tables

This section describes **multiset operators** for nested tables. Multiset operations are not available for varrays.

The SQL examples in this section use the nested tables created in Example 5–5 on page 5-5 and the objects created in Example 5–1 on page 5-2.

See Also:

- "Functions and Operators Useful with Objects" on page 2-32 for a description of additional operations
- *Oracle Database SQL Language Reference*.for more information about using operators with nested tables

CARDINALITY

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

Example 5–35 Determining the CARDINALITY of a Nested Table

```
-- Requires Ex. 5-1 and 5-5
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 creates a multiset from a set of elements. The function takes a column of the element type as input and creates 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 the second. The input nested tables and the output nested table will all be of the same nested table type.

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 are part of the result. If a particular element occurs *m* times in ntab1 and *n* times in ntab2, the result shows (*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 and is also present in ntab2 is eliminated, irrespective of the number of occurrences.

Example 5–36 Using the MULTISET EXCEPT Operation on Nested Tables

```
-- Requires Ex. 5-1 and 5-5
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 INTERSECT

The MULTISET INTERSECT operator returns a nested table whose values are common to 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 contains the element MIN(*m*, *n*) times. With the DISTINCT option, the

duplicates from the result are eliminated, including duplicates of NULL values if they exist.

Example 5–37 Using the MULTISET INTERSECT Operation on Nested Tables

```
-- Requires Ex. 5-1 and 5-5
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 in ntab1 and ntab2 are 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 contains 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 5–38 Using the MULTISET UNION Operation on Nested Tables

```
-- Requires Ex. 5-1 and 5-5
SELECT math_majors MULTISET UNION DISTINCT physics_majors
  FROM students
WHERE graduation = '01-JUN-03';

PEOPLE_TYP(PERSON_TYP(12, 'Bob Jones', '1-650-555-0130'),
       PERSON_TYP(31, 'Sarah Chen', '1-415-555-0120'),
       PERSON_TYP(45, 'Chris Woods', '1-408-555-0128'))

-- Requires Ex. 5-1 and 5-5
SELECT math_majors MULTISET UNION ALL physics_majors
  FROM students
WHERE graduation = '01-JUN-03';

PEOPLE_TYP(PERSON_TYP(12, 'Bob Jones', '1-650-555-0130'),
    PERSON_TYP(31, 'Sarah Chen', '1-415-555-0120'),
    PERSON_TYP(45, 'Chris Woods', '1-408-555-0128'),
    PERSON_TYP(12, 'Bob Jones', '1-650-555-0130'),
    PERSON_TYP(45, 'Chris Woods', '1-408-555-0128'))
```

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 can be any expression which evaluates to a multiset. The limit on the cardinality of the multiset argument is 32.

Example 5–39 Using the POWERMULTISET Operation on Multiset

```
-- Requires Ex. 5-1 and 5-5
SELECT * FROM TABLE(POWERMULTISET( people_typ (
           person_typ(12, 'Bob Jones', '1-650-555-0130'),
           person_typ(31, 'Sarah Chen', '1-415-555-0120'),
           person_typ(45, 'Chris Woods', '1-415-555-0124'))));
```

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 is 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 can be any expression which evaluates to a nested table. The length parameter must be a positive integer, otherwise an error is returned. The limit on the cardinality of the nested table argument is 32.

Example 5–40 Using the POWERMULTISET_BY_CARDINALITY Function

```
-- Requires Ex. 5-1 and 5-5
SELECT * FROM TABLE(POWERMULTISET_BY_CARDINALITY( people_typ (
           person_typ(12, 'Bob Jones', '1-650-555-0130'),
           person_typ(31, 'Sarah Chen', '1-415-555-0120'),
           person_typ(45, 'Chris Woods', '1-415-555-0124')),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 5–41 Using the SET Function on a Nested Table

```
-- Requires Ex. 5-1 and 5-5
SELECT SET(physics_majors)
  FROM students
WHERE graduation = '01-JUN-03';
```

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

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*.

With Oracle Database 11*g* release 1 (11.1), nested tables are equipartitioned, meaning that the partitioning of storage tables associated with nested tables corresponds to that of the top level base tables. The keyword LOCAL is the default and indicates this behavior. Previous releases stored unpartitioned nested tables with partitioned base tables. To obtain this behavior, specify the GLOBAL keyword. To partition your existing non-partitioned nested tables, see the discussion of online redefinition in *Oracle Database VLDB and Partitioning Guide*.

Generally, maintenance operations are carried out on the top level (or parent table) and cascade to the associated nested tables. However, you must perform the following operations directly on the nested table partition:

- Modifying default attributes, that is, changing default physical attributes of a partition.
- Modifying a partition, that is, changing the physical attributes of a table partition.
- Moving a partition, that is, moving contents of a table partition into another segment or another tablespace.
- Renaming a partition, that is, assigning a new name to a table partition.

See Also: For further information on equipartitioning

- *Oracle Database VLDB and Partitioning Guide*
- *Oracle XML DB Developer's Guide*

Example 5–42 partitions the purchase order table along zip codes (ToZip), which is an attribute of the ShipToAddr embedded column object. The LineItemList_nt nested table illustrates storage for the partitioned nested table.

Example 5–42 Partitioning a Nested Table That Contains Objects

```
CREATE TYPE StockItem_objtyp AS OBJECT (
      StockNo NUMBER,
      Price NUMBER,
      TaxRate NUMBER);
/

CREATE TYPE LineItem_objtyp AS OBJECT (
      LineItemNo NUMBER,
      Stock_ref REF StockItem_objtyp,
      Quantity NUMBER,
      Discount NUMBER);
/

CREATE TYPE Address_objtyp AS OBJECT (
      Street VARCHAR2(200),
      City VARCHAR2(200),
      State CHAR(2),
      Zip VARCHAR2(20))
/

CREATE TYPE LineItemList_nt as table of LineItem_objtyp;
/

CREATE TYPE PurchaseOrder_ntyp AS OBJECT (
      PONo                NUMBER,
```

```
      OrderDate           DATE,
      ShipDate            DATE,
      OrderForm           BLOB,
      LineItemList        LineItemList_nt,
      ShipToAddr          Address_objtyp,

   MAP MEMBER FUNCTION
      ret_value RETURN NUMBER,
   MEMBER FUNCTION
      total_value RETURN NUMBER);
/

CREATE TABLE PurchaseOrders_ntab of PurchaseOrder_ntyp
LOB (OrderForm) store as (nocache logging)
NESTED TABLE LineItemList STORE AS LineItemList_ntab
PARTITION BY RANGE (ShipToAddr.zip)
   (PARTITION PurOrderZone1_part VALUES LESS THAN ('59999')
       LOB (OrderForm) store as (
       storage (INITIAL 10 MINEXTENTS 10 MAXEXTENTS 100))
       NESTED TABLE LineItemList store as LineitemZone1_part(
       storage (INITIAL 10 MINEXTENTS 10 MAXEXTENTS 100)),
   PARTITION PurOrderZone2_part  VALUES LESS THAN ('79999')
       LOB (OrderForm) store as (
       storage (INITIAL 10 MINEXTENTS 10 MAXEXTENTS 100))
      NESTED TABLE LineItemList store as LineitemZone2_part(
       storage (INITIAL 10 MINEXTENTS 10 MAXEXTENTS 100)),
   PARTITION PurOrderZone3_part  VALUES LESS THAN ('99999')
       LOB (OrderForm) store as (
       storage (INITIAL 10 MINEXTENTS 10 MAXEXTENTS 100))
       NESTED TABLE LineItemList store as LineitemZone3_part(
   storage (INITIAL 10 MINEXTENTS 10 MAXEXTENTS 100)) )
/
```

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.

You can run object-oriented applications without converting existing tables to a different physical structure. To do this, you can use object views to prototype or transition 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.

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 makes up a row of an object view traverses 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.

- You can query the data in new ways by synthesizing objects from relational data. You can view data from multiple tables by using object dereferencing instead of writing complex joins with multiple tables.
- You can pin the object data from object views and use the data 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, enabling you to continue developing the model. If you need to alter an object type, you can simply replace the invalidated views with a new definition.
- You do not place any restrictions on the characteristics of the underlying storage mechanisms by using objects in views. 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 Language 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 the relational table. 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, enabling you to create pointers (`REF`s) to the objects in the view. You can often use an existing primary key.

See Also: "Using Object Identifiers to Identify Row Objects" on page 1-7

To update an object view where the attributes of the object type do not correspond exactly to columns in existing tables, you may need to do the following:

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 6-10.

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

Example 6–1 contains SQL statements to 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';

insert into emp_table values(1,'John',1000.00,'Architect');
insert into emp_table values(2,'Robert',900.00,'Developer');
insert into emp_table values(3,'James',2000.00,'Director');

select * from emp_view1;


    EMPNO ENAME                    SALARY JOB
---------- -------------------- ---------- --------------------
         2 Robert                      900 Developer
```

To access the data from the `empnum` column of the relational table, 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 view 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 an object to the database in an object view, the database 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 constructor methods of the object type to create the object instances, and you can populate their attributes with data from relational columns specified 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;

insert into dept values(1,'Sales','500 Oracle pkwy','Redwood S','CA','94065');
insert into dept values(2,'ST','400 Oracle Pkwy','Redwood S','CA','94065');
insert into dept values(3,'Apps','300 Oracle pkwy','Redwood S','CA','94065');

select * from dept_view;
```

```
    DEPTNO DEPTNAME
---------- --------------------
ADDRESS(STREET, CITY, STATE, ZIP)
--------------------------------------------------------------------------------
         1 Sales
ADDRESS_T('500 Oracle pkwy', 'Redwood S', 'CA', '94065')

         2 ST
ADDRESS_T('400 Oracle Pkwy', 'Redwood S', 'CA', '94065')

         3 Apps
ADDRESS_T('300 Oracle pkwy', 'Redwood S', 'CA', '94065')
```

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 or not the department address is null.

By using the `DECODE` function, or some other function, to return either a null or the constructed object, the null `deptstreet` column can be used to indicate that the whole address is null.

Example 6–3 Identifying Null Objects in an Object View

```
-- Requires Ex. 6-2
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;
```

This 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, 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 `VARRAY`s, 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.

This section contains the following topics:

- Single-Level Collections in Object Views
- Multilevel Collections in Object Views

Single-Level Collections in Object Views

Using Example 6–1 and Example 6–2 as starting points, each employee in an `emp` relational table has the structure in Example 6–4. Using this relational table, you 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 with 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

```
-- Requires Ex. 6-1 and Ex. 6-2
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;  -- nested table
/
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;

insert into dept values(100,'ST','400 Oracle Pkwy','Redwood S','CA',94065);
insert into dept values(200,'Sales','500 Oracle Pkwy','Redwood S','CA',94065);
insert into emp values(1,'John',900,'Developer1',100);

insert into emp values(2,'Robert',1000,'Developer2',100);
insert into emp values(3,'Mary', 1000,'Apps1',200);
insert into emp values(4,'Maria',1500,'Developer3',200);
select * from dept_view where deptno = 100;

    DEPTNO DEPTNAME
---------- --------------------
ADDRESS(STREET, CITY, STATE, ZIP)
--------------------------------------------------------------------------------
EMP_LIST(EMPNO, ENAME, SALARY, JOB)
--------------------------------------------------------------------------------
       100 ST
ADDRESS_T('400 Oracle Pkwy', 'Redwood S', 'CA', '94065')
EMPLOYEE_LIST_T(EMPLOYEE_T(1, 'John', 900, 'Developer1'), EMPLOYEE_T(2, 'Robert'
, 1000, 'Developer2'))


select emp_list from dept_view where deptno = 100;

EMP_LIST(EMPNO, ENAME, SALARY, JOB)
--------------------------------------------------------------------------------
EMPLOYEE_LIST_T(EMPLOYEE_T(1, 'John', 900, 'Developer1'), EMPLOYEE_T(2, 'Robert'
, 1000, 'Developer2'))
```

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 provide 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. Object identifiers allow you to reference the objects in object views and pin them 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. To do this, 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.

You can specify object identifiers based on the primary key. This turns the set of unique keys that identify the row object 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.

> **See Also:** "Using Object Identifiers to Identify Row Objects" on page 1-7 for a description of primary-key based and system-generated object identifiers

- Object views created with the WITH OBJECT IDENTIFIER Clause

  An object view created with the WITH OBJECT IDENTIFIER clause has an object identifier derived from the primary key.

  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 you know that deptno is unique within the view objects, you can specify it as the object identifier.

- Object views created with the WITH OBJECT IDENTIFIER DEFAULT Clause

  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.

  > **See Also:** See "Storage Considerations for Object Identifiers (OIDs)" on page 9-4.

Creating References to View Objects

In this connected group of examples, Example 6–2 and Example 6–4, 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. The primary key value creates the object identifier in the dept_view, allowing the foreign key value in the emp_view to create a reference to the primary key value in dept_view.

To synthesize a primary key object reference, use the MAKE_REF operator. This takes the view or table name that the reference points to, and a list of foreign key values, to create the object identifier portion of the reference that matches a specific object in the referenced view.

Example 6–6 creates an emp_view view which has the employee's number, name, salary and a reference to the employee's department, by first creating the employee type emp_t and then the view based on that type.

Example 6–6 Creating a Reference to Objects in a View

```
-- Requires Ex. 6-2 and Ex. 6-4
-- if you have previously created emp_t, you must drop it
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 departments are located in the city of Redwood S:

```
SELECT e.empno, e.salary, e.deptref.deptno
  FROM emp_view e
 WHERE e.deptref.address.city = 'Redwood S';

   EMPNO     SALARY DEPTREF.DEPTNO
---------- ---------- --------------
         2       1000            100
         1        900            100
         4       1500            200
         3       1000            200
```

Note that you can also create emp_view using the REF modifier instead of MAKE_REF as follows to get the reference to the dept_view objects:

Example 6–7 Query References to Objects with REF

```
-- Requires Ex. 6-2, Ex. 6-4, and Ex. 6-6
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 Example 6–7, the `dept_view` joins the `emp` table on the `deptno` key.

The advantage of using the `MAKE_REF` operator, as in Example 6–6, instead of the `REF` modifier is that with the former, you can create circular references. For example, you can create an employee view that has a reference to the employee's department, while the department view has 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 is 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.

> **See Also:** "Object Cache" on page 4-3

As with synthesized objects, you 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

You can use views with objects to model inverse relationships.

One-to-One Relationships

One-to-one relationships can be modeled with inverse object references. For example, suppose that each employee has a particular desktop computer, 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, you can model the objects so there is an object reference from the employee to the computer object and also 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, the foreign key is in the employee table. The relationship between departments and employees can be modeled using collections in views. 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 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.

Views are not always directly updatable.

A view is not directly updatable if the 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 expressions 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, 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 Also:** "Using INSTEAD OF Triggers to Control Mutating and Validation" on page 6-11 for an example of an INSTEAD OF trigger

---

Note: 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.

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 to just those in the view specified, use the ONLY keyword. For example, the following statement updates only persons and not employees or students.

```
UPDATE ONLY(Person_v) SET address = ...
```

> **See Also:** "Object View Hierarchies" on page 6-17 for a discussion of object view hierarchy and examples defining Student_v and Employee_v views

Updating Nested Table Columns in Views

You can modify a nested table 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 to update complex views that otherwise could not be updated. They can also be used to enforce constraints, check privileges, and validate DML statements. Using these triggers, you can control mutation that might be caused by inserting, updating, and deleting in the objects created though an object view.

For instance, to enforce the condition that the number of employees in a department cannot exceed 10, you can write an INSTEAD OF trigger for the employee view. The

trigger is not needed to execute the DML statement because the view can be updated, but you need it to enforce the constraint.

Example 6–8 shows how to implement the trigger by means of SQL statements.

Example 6–8 Creating INSTEAD OF Triggers on a View

```
-- Requires Ex. 6-2, Ex. 6-4, and Ex. 6-6
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;
/
```

See Also: "Triggers for Object Tables" on page 2-5

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 a list of all the departments. To get a total view of the entire organization, you can create views over the individual remote tables and then a overall view of the organization.

To this requires the following:

- Update the entry in `listener.ora`, such as: `(ADDRESS=(PROTOCOL=tcp)(HOST=stadv07.us.example.com)(PORT=1640))`
- Add entries to `tnsnames.ora`, such as: `chicago=(DESCRIPTION=(ADDRESS=(PROTOCOL=ipc)(KEY=linux)) (CONNECT_DATA=(SERVICE_NAME=linux.regress.rdbms.dev.us.example.com)))`
- Provide `CREATE DATABASE LINK` code as shown in Example 6–9

Example 6–9 begins by creating an object view for each employee table and then creates the global view.

Example 6–9 Creating an Object View to Access Remote Tables

```
-- Requires Ex. 6-2, Ex. 6-4, and Ex. 6-6
-- Example requires DB links, such as these, modify for your use and uncomment
-- CREATE DATABASE LINK chicago CONNECT TO hr IDENTIFIED BY hr USING 'inst1';
-- CREATE DATABASE LINK washington CONNECT TO hr IDENTIFIED BY hr USING 'inst1';
```

```
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 a list of all employees for each department. The `UNION ALL` clause is used because employees cannot work 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, you may want to put references to the employee objects inside the department object, instead of materializing all the employees within the department object. You 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:

> **Note:** Both ways to create circular references require the setup described in "Tables and Types to Demonstrate Circular View References" on page 6-14.

- First View After 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.

See the example in "Method 1: Re-create First View After Creating Second View" on page 15

- First View Using the FORCE Keyword

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

See the example in "Method 2: Create First View Using FORCE Keyword" on page 16

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.

Perform the setup described next before attempting to use either method of creating circular view references.

Tables and Types to Demonstrate Circular View References

First, you need 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, you will create object views later.

The emp table stores the employee information:

Example 6–10 Creating emp table to demonstrate circular references

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

-- first create a dummy, that is, incomplete, department type, so emp_t type
-- created later will succeed

CREATE TYPE dept_t;
/

-- Create the employee type with a reference to the department, dept_t:
CREATE TYPE emp_t AS OBJECT
( eno NUMBER,
  ename VARCHAR2(20),
  salary  NUMBER,
  deptref REF dept_t );
/

-- Represent the list of references to employees as a nested table:
CREATE TYPE employee_list_ref_t AS TABLE OF REF emp_t;
/

-- Create the department table as a relational table
CREATE TABLE dept
```

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

-- Create 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));
/

-- Fill in the definition for dept_t, the incomplete type you previously created:
CREATE OR REPLACE TYPE dept_t AS OBJECT
( dno           NUMBER,
  dname         VARCHAR2(20),
  deptaddr      address_t,
  empreflist    employee_list_ref_t);
/
```

As Example 6–10 indicates, you must create the `emp` table, then create a dummy department type, `dept_t` which will enable the `emp_t` type to succeed once you create it. After that, create `emp_t` with a reference to `dept_t`. Create a list of references to employees as a nested table, `employee_list_ref_t` and create the department table, `dept`. Then create an object type, `address_t` that has columns mapping to the relational tables, and finally fill in the definition for the incomplete `dept_t`.

The following is example data you could use:

```
insert into emp values(1,'John','900',100);
insert into emp values(2,'james','1000',100);
insert into emp values(3,'jack',2000,200);
```

Creating Object Views with Circular References

Now that you have established the underlying relational table definitions, in the previous section, create the object views on top of them.

Method 1: Re-create First View After Creating Second View

First create the employee view with a null in the `deptref` column. Later, you can turn that column into a reference.

Next, create the department view, which includes references to the employee objects. This creates a list of references to employee objects, instead of including the entire employee object.

Next, re-create the employee view with the reference to the department view.

Example 6–11 Creating an Object View with a Circular Reference, Method 1

```
-- Requires Ex. 6-10
CREATE VIEW emp_view OF emp_t WITH OBJECT IDENTIFIER(eno)
   AS SELECT e.empno, e.empname, e.salary, NULL
         FROM emp e;

-- create department view, including 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;

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

First, 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.

Next, create a department view that includes references to the employee objects. You do not have to use the `FORCE` keyword here, because `emp_view` already exists. This allows you to query the department view, getting the employee object by dereferencing the employee reference from the nested table `empreflist`.

> **Note:** If you previously ran Example 6–11, remove the views you created before running Example 6–12.

Example 6–12 Creating view with FORCE Method 2

```
-- Requires Ex. 6-10
-- create employee view
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;

-- create a department view that includes references to the employee objects
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;

-- Querying with DEREF method
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.

You can also access the employee number only, for all those employees whose name begins with John.

Example 6–13 Querying with COLUMN_VALUE

```
-- Requires Ex. 6-10 and 6-12
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:

Example 6–14 Querying with COLUMN_VALUE, Unnesting References

```
-- Requires Ex. 6-10 and 6-12
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, you can rewrite the preceding query to use the emp_view instead of the dept_view to show how to navigate from one view to the other:

Example 6–15 Querying with COLUMN_VALUE, Querying emp_view

```
-- Requires Ex. 6-10 and 6-12
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%';
```

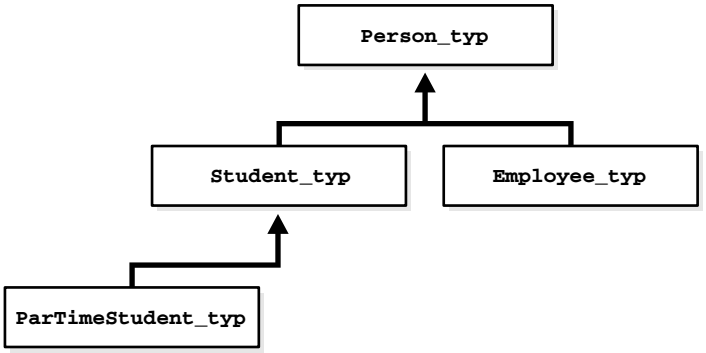
Object View Hierarchies

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 Type 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. However, 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 or not the view is updatable. For a view to be updatable, its query must contain no joins, set operators, aggregate functions, `GROUP BY` clause, `DISTINCT` clause, 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 6–17.

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 affects the performance and updatability of the object view hierarchy.

This section describes 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.

To execute any of these storage models, first create types shown in Example 6–16.

Example 6–16 Creating Types for Storage Model Examples

```
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));
/
```

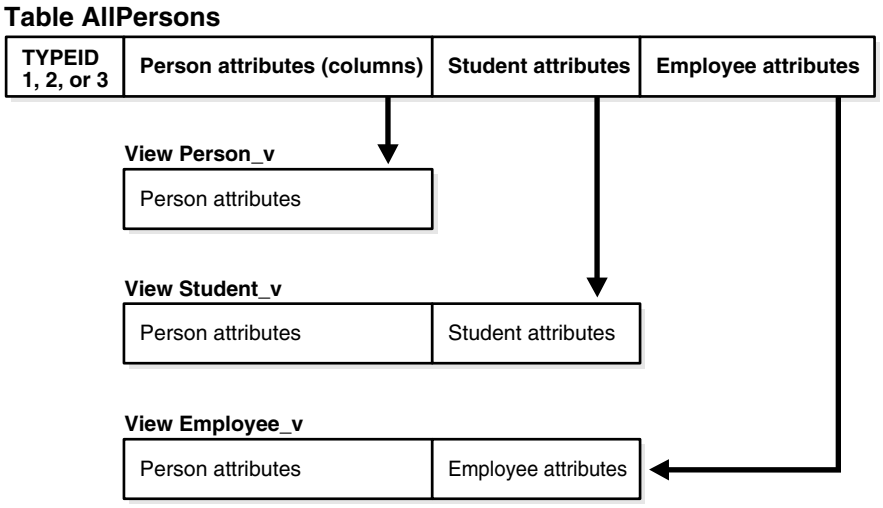
This section contains the following topics:

- The Flat Model
- The Horizontal Model
- The Vertical Model

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`, and `employee_typ`.

Figure 6–2 Flat Storage Model for Object View Hierarchy



The `typeid` column identifies the type of each row. These possible values are the types created in Example 6–16, 1 = `person_typ`, 2 = `student_typ, and` 3 = `employee_typ`:

Example 6–17 creates the table `AllPersons` and then the views that make up the object view hierarchy:

Example 6–17 Creating an Object View Hierarchy

```
-- Requires Ex. 6-16
CREATE TABLE AllPersons
( typeid NUMBER(1),
  ssn NUMBER,
  name VARCHAR2(30),
  address VARCHAR2(100),
  deptid NUMBER,
  major VARCHAR2(30),
  empid NUMBER,
  mgr VARCHAR2(30));

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;
```

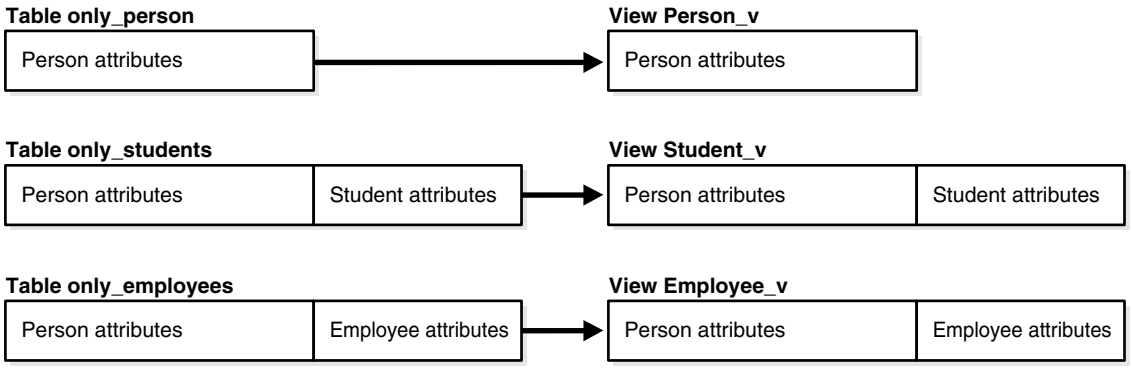
The flat model has the advantage of simplicity and poses no obstacles to supporting indexes and constraints. Its drawbacks are:

- 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



Example 6–18 creates tables and then views based on these tables.

Example 6–18 -- Creating Table Horizontal Model

```
-- Requires Ex. 6-16 and Ex. 6-17
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));

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:

Example 6–19 -- Querying views horizontal model

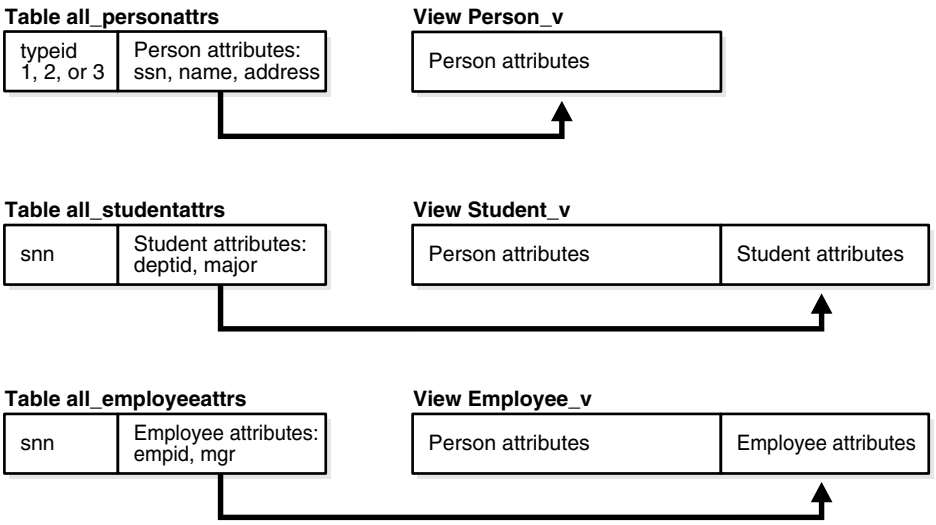
```
-- Requires Ex. 6-16 and Ex. 6-17
-- add the following data
insert into only_persons values(1234,'John','abc');
insert into only_students values(1111,'James','abc',100,'CS');
insert into only_employees values(2222,'jack','abc',400,'Juliet');

SELECT VALUE(p) FROM Person_v p
  WHERE VALUE(p) IS OF (ONLY student_typ);

OUTPUT:
VALUE(P)(SSN, NAME, ADDRESS)
--------------------------------------------------------------------------------
STUDENT_TYP(1111, 'James', 'abc', 100, 'CS')
```

Such queries only need to access a single physical table to get all the objects of the specific type. The drawbacks of this model are that queries such as `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 6-23.) 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 the physical tables store only those attributes that are unique to their corresponding subtypes.

Figure 6–4 Vertical Storage Model for Object View Hierarchy



Example 6–20 creates tables and then corresponding views.

Example 6–20 Creating table, views vertical model

```
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 object identifiers (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 the subtypes of that type. 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 declared type of the view, 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 declared type of the view and any subtypes of that type.

For example, the following query returns instances of persons, employees, and students of all types, as well as REFs to those instances.

Example 6–21 Query with REF and Value

```
-- Requires Ex. 6-20
insert into all_personattrs values(1,1111,'John','abc');
insert into all_personattrs values(2,2222,'Jack','def');
insert into all_personattrs values(3,3333,'James','ghi');
insert into all_studentattrs values(2222,100,'CS');
insert into all_employeeattrs values(3333,444,'Julia');
SELECT REF(p), VALUE(p) FROM Person_v p;

OUTPUT:
REF(P)
--------------------------------------------------------------------------------
VALUE(P)(SSN, NAME, ADDRESS)
--------------------------------------------------------------------------------
00004A038A00465A6E6E779EC1F25FE040578CE70A447E00000014260100010001002900000000000
090600812A00078401FE0000000B03C20C0C000000000000000000000000000000000000000
PERSON_TYP(1111, 'John', 'abc')

00004A038A00465A6E6E779EC1F25FE040578CE70A447E00000014260100010001002900000000000
090600812A00078401FE0000000B03C222220000000000000000000000000000000000000000
EMPLOYEE_TYP(3333, 'James', 'ghi', 444, 'Julia')

00004A038A00465A6E6E779EC1F25FE040578CE70A447E00000014260100010001002900000000000

REF(P)
--------------------------------------------------------------------------------
VALUE(P)(SSN, NAME, ADDRESS)
--------------------------------------------------------------------------------
090600812A00078401FE0000000B03C21717000000000000000000000000000000000000000
STUDENT_TYP(2222, 'Jack', 'def', 100, 'CS')
```

If you specify individual attributes of the declared type of the view in the SELECT list or do a SELECT *, again the query returns rows for the declared type of the view and any subtypes of that type, but these rows are projected over columns for the attributes of the declared type of the view, 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 declared type of the view.

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 user 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
- Type Dependencies
- Synonyms for Object Types
- Performance Tuning

Privileges on Object Types and Their Methods

Privileges for object types exist at the system level and the schema object level.

This section contains the following topics:

- System Privileges for Object Types
- Schema Object Privileges
- Using Types in New Types or Tables
- Example: Privileges on Object Types
- Access Privileges on Objects, Types, and Tables

System Privileges for Object Types

Oracle database 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 following roles are helpful:

- 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 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 methods of a type, 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.

  Only a grantor with the UNDER privilege WITH GRANT OPTION on the direct supertype or superview can grant the UNDER privilege on a subtype or subview.

The phrase WITH HIERARCHY OPTION grants a specified object privilege on all subtypes 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 either the EXECUTE ANY TYPE system privilege or the EXECUTE object privilege for any type used to define a new type or table. You must have been granted these privileges explicitly, and not through a role.

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 been granted these privileges explicitly, not through a role.

Example: Privileges on Object Types

Example 7–1 creates three users or schemas, USER1, USER2, and USER3, and grants them the CREATE SESSION and RESOURCE roles. Some of the subsequent examples in this chapter use these schemas.

This example requires you to create and use several passwords. If you plan to run the example, make these changes to your SQL code first.

> **Note:** For simplicity, this example does not perform the password management techniques that a deployed system normally uses. In a production environment, follow the Oracle Database password management guidelines, and disable any sample accounts. See *Oracle Database Security Guide* for password management guidelines and other security recommendations.

Example 7–1 Creating User Schemas

```
-- Requires passwords
CONNECT SYSTEM
-- Enter password
CREATE USER user1 PROFILE default
   IDENTIFIED BY password DEFAULT TABLESPACE example ACCOUNT UNLOCK;
GRANT CREATE SESSION TO user1;
GRANT RESOURCE TO user1;
GRANT CREATE SYNONYM TO user1;
GRANT CREATE PUBLIC SYNONYM TO user1;
GRANT DROP PUBLIC SYNONYM TO user1;
CREATE USER user2 PROFILE default
   IDENTIFIED BY password DEFAULT TABLESPACE example ACCOUNT UNLOCK;
GRANT CREATE SESSION TO user2;
GRANT RESOURCE TO user2;
CREATE USER user3  PROFILE default
   IDENTIFIED BY password DEFAULT TABLESPACE example ACCOUNT UNLOCK;
GRANT CREATE SESSION TO user3;
GRANT RESOURCE TO user3;
```

Example 7–2 requires the input of a password, USER1 performs the CREATE and GRANT Data Definition Language (DDL) statements in the USER1 schema:

Example 7–2 Granting Privileges on Object Types

```
-- Requires Ex. 7-1 and the input of a password
CONNECT user1
-- Enter password

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;
```

In Example 7–3, USER2 performs the CREATE DDL statement in the USER2 schema:

Example 7–3 Performing DDL Statements in USER2 Schema

```
-- Requires Ex. 7-1, 7-2 and password input
CONNECT user2
-- Enter password
CREATE TABLE tab1 OF user1.type1;
CREATE TYPE type3 AS OBJECT ( attr3 user1.type2 );
/
CREATE TABLE tab2 (col1 user1.type2 );
```

In Example 7–4, the first two statements succeed because USER2 was granted the EXECUTE privilege with the GRANT option on USER1's TYPE2 in the last line of Example 7–2 and Example 7–3 created type3 as an object using attr3 user1.type2.

However, the last grant Example 7–4 fails because USER2 has not been granted the EXECUTE privilege with the GRANT option on USER1.TYPE1.

Example 7–4 Performing Grants to USER3

```
-- Requires Ex. 7-1, 7-2, and 7-3
GRANT EXECUTE ON type3 TO user3;
```

```
GRANT SELECT ON tab2 TO user3;

-- Privileges on Object Types
GRANT SELECT ON tab1 TO user3 -- incorrect statement;
```

In Example 7–5, USER3 has the necessary privileges to perform the following actions:

Example 7–5 Creating Tables and Types

```
-- Requires Ex. 7-1, 7-2, 7-3, and 7-4
CONNECT user3
-- Enter password
CREATE TYPE type4 AS OBJECT (attr4 user2.type3);
/
CREATE TABLE tab3 OF type4;
```

Access Privileges on Objects, Types, and Tables

Object types only make use of the EXECUTE privilege, but 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 7–6:

Example 7–6 SELECT Privileges on Type Access

```
-- Requires Ex. 7-1, 7-2, 7-3, 7-4, and 7-5
CREATE TYPE emp_type AS OBJECT (
  eno     NUMBER,
  ename   VARCHAR2(36));
/
CREATE TABLE emp OF emp_type; // an object table
GRANT SELECT on emp TO user1;
SELECT VALUE(e) FROM emp e;
SELECT eno, ename FROM emp;
```

For both queries, Oracle database checks the user's SELECT privilege for the object table emp. 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, the database checks the user's EXECUTE privilege.

The second query, however, does not involve named types, so the database does not check type privileges.

Additionally, USER3 can perform queries such as these:

```
SELECT t.col1.attr2 from user2.tab2 t;
SELECT t.attr4.attr3.attr2 FROM tab3 t;
```

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

Oracle database 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 the database to check the SELECT privilege on the object table containing the object and the 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 the database to check the UPDATE privilege on the destination object table. Flushing a new object causes the database to check the INSERT privilege on the destination object table.
- Deleting an object causes the database to check the DELETE privilege on the destination table.
- Invoking a method causes the database to check the EXECUTE privilege on the corresponding object type.

Oracle database does not provide column level privileges for object tables.

Type Dependencies

This section discusses type dependencies in two broad categories:

- Situations where types depend upon each other for their definitions, where one type might be part of the definition of another type.
- Situations where creating or dropping types is complicated by dependencies that the type has such, as tables or types.

This section covers the following topics:

Creating Incomplete Types

Types that depend on each other for their definitions, either directly or through intermediate types, are called mutually dependent. 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.

If you visualize a diagram with arrows showing the relationships among a set of mutually dependent types, the connections 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 7–7.

Example 7–7 Creating Dependent Object Types

```
-- Requires Ex. 7-1 and password
CONNECT user1
-- Enter password
ALTER SESSION SET PLSQL_WARNINGS = 'enable:all';

CREATE TYPE department; // a placeholder
/

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 database compiles it without errors.

Notice that the code in Example 7–7 creates the type `department` twice. The first statement 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.

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 database would create it as an incomplete type and compile `employee` with errors. Then `employee` would be recompiled the next time that some operation accesses it. This time, if all the types it depends on have been created and its dependencies are satisfied, it compiles 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.

Completing Incomplete Types

When you have created all the types referenced by an incomplete type, complete the declaration of the incomplete type, because there is no longer any need for it to remain incomplete. Completing the type recompiles it and enables the system to release various locks.

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 Example 7–7.

Also, you must complete any incomplete types that the database creates for you. If, as discussed in the preceding section, you did not explicitly create department as an incomplete type, then the database did. In this case, you still need to complete it.

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 is to drop the 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 the type 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.

Using CREATE OR REPLACE TYPE with Type and Table Dependencies

The CREATE OR REPLACE TYPE statement throws an error if the type being replaced has table or type dependencies. This applies to objects, varrays, and nested table types. This also applies to type dependencies involving either inheritance or type composition (embedding one type into another). The latter might be a situation where one type is attribute of another.

Using the FORCE option with a CREATE OR REPLACE TYPE statement enables you to replace a type if it has type dependencies, but not table dependencies. Table dependencies still cause errors.

Example 7–8 shows a CREATE OR REPLACE statement (second statement) that fails due to a type dependency.

Example 7–8 CREATE OR REPLACE Type and Table Failure

```
SQL> CREATE type t1 AS OBJECT (a number) not final;
  2  /
Type created.

SQL> CREATE TYPE t2 UNDER t1 (b varchar(10));
  2  /
Type created.

SQL> CREATE OR REPLACE TYPE t1 AS OBJECT (c varchar(20));
  2  /
CREATE OR REPLACE TYPE t1 AS OBJECT (c varchar(20));
*
ERROR at line 1:
ORA-02303: cannot drop or replace a type with type or table dependents
```

Example 7–9 shows code in which a CREATE OR REPLACE FORCE statement succeeds in replacing a type that has a type dependency and then creates a table using the parent type. However, the final CREATE OR REPLACE FORCE statement fails because the type now has a table dependency and even with the FORCE option, a type with a table dependency cannot be replaced.

Example 7–9 CREATE OR REPLACE with FORCE

```
SQL> CREATE OR REPLACE TYPE t1 FORCE AS OBJECT (c varchar(20));
  2  /
Type created.

SQL> CREATE TABLE tb1 (c1 t1);
Table created.

SQL> CREATE OR REPLACE TYPE t1 FORCE AS OBJECT (d number);
  2  /
CREATE OR REPLACE TYPE t1 FORCE AS OBJECT (d number);
*
ERROR at line 1:
ORA-22866: cannot replace a type with table dependents
```

> **See Also:** *Oracle Database PL/SQL Language Reference* for details of the `CREATE OR REPLACE TYPE` SQL statement

Type Dependencies of Substitutable Tables and Columns

A substitutable table or column of a specific type is dependent not only on that type but on all subtypes of the type as well. This is because a hidden column is added to the table for each attribute added in a subtype of the type. The hidden columns are added even if the substitutable table or column contains no data of that subtype.

> **See Also:** "Substituting Types in a Type Hierarchy" on page 2-23 for further explanation of substitutability

In Example 7–10, 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. Consequently, trying to drop a `part_time_student_typ` raises an error because of the dependent `persons` table.

If dependent tables or columns exist but contain no data of the type being dropped, you can use the `VALIDATE` keyword to drop the type. The `VALIDATE` keyword causes Oracle database to check for actual stored instances of the specified type and to drop the type if none are found. This also removes hidden columns associated with attributes unique to the type.

In Example 7–10, the first `DROP TYPE` statement fails because `part_time_student_typ` has a dependent table (`persons`). But if `persons` contains no instances of `part_time_student_typ` (nor does any other dependent table or column), the `VALIDATE` keyword causes the second `DROP TYPE` statement to succeed.

Example 7–10 DROP TYPE with and without VALIDATE

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

```
    major VARCHAR2(30))
NOT FINAL;
/
CREATE TYPE part_time_student_typ UNDER student_typ (number_hours NUMBER);
/
CREATE TABLE persons OF person_typ;
-- 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 DROP TYPE 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. Use the `FORCE` option with great care, because 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 for this reason 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 8-6 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 unaltered, in any schema of a database, without requiring a qualified type name with the schema name.

See Also: *Oracle Database Administrator's Guide* for more information on synonyms in general

This section contains the following topics:

- Creating a Type Synonym
- Using a Type Synonym

Creating a Type Synonym

You create a type synonym with a `CREATE SYNONYM` statement. The user must have been granted `CREATE SYNONYM` and `CREATE PUBLIC SYNONYM` privileges.

For example, these statements create a type `typ1` and then create a synonym for it:

Example 7–11 CREATE TYPE / SYNONYM for user1

```
-- Example requires Ex.7-1 which created user1 and granted it the CREATE SYNONYM
-- and CREATE PUBLIC SYNONYM privileges
-- connect as user1 if not already connected.
```

```
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;
```

With the `REPLACE` option you can make 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.

Example 7–12 uses synonym `syn1` to specify the type of an attribute in type `typ3`:

Example 7–12 Using a Type Synonym in a Create Statement

```
-- Requires Ex 7-1 and connection as user1
-- drop syn1 and typ1 if created for Ex. 7-12
CREATE TYPE typ1 AS OBJECT (x number);
/
CREATE SYNONYM syn1 FOR typ1;

CREATE TYPE typ3 AS OBJECT ( a syn1 );
/
```

In the next statement, the type synonym `syn1` calls 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, `syn2` is a synonym for a nested table type. The synonym replaces the actual type name in a `CAST` expression.

```
SELECT CAST(MULTISET(SELECT eno FROM USER3.EMP) AS syn2) FROM dual;
```

This code returns the following output:

```
SQL> -- Type synonym used to call a constructor / nested table
SELECT syn1(0) FROM dual;
SELECT CAST(MULTISET(SELECT eno FROM USER3.EMP) AS syn2) FROM
   dual;
```

```
SQL>
SYN1(0)(X)
----------------------------------------------------------------
TYP1(0)

SQL>
CAST(MULTISET(SELECTENOFROMUSER3.EMP)ASSYN2)
----------------------------------------------------------------
TYP2()
```

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 shows the synonyms in place of the types they represent. Similarly, catalog views, which show type names, such as `USER_TYPE_ATTRS`, show the type synonym names in their place.

You can query the catalog view `USER_SYNONYMS` to find out the underlying type of a type synonym.

> **See Also:** Chapter 2 of *Oracle Database Reference* for a complete list of the data dictionary catalog views

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 line from Example 7–12, 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 the underlying type of the synonym.

The dependency relationships of a synonym 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 7–13.

Example 7–13 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 or a new public synonym with the same name as the dropped synonym.

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 that will hold 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

8

Advanced Topics for Oracle Objects

The previous chapters in this book discuss 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 database automatically maps the complex structure of object types into simple table structure for storage.

This section discusses these related topics:

Leaf-Level Attributes

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

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

The following topics relate to the discussion of object tables and relational tables in "How Objects are Stored in Tables" on page 1-6.

How Row Objects Are Split Across Columns

In an **object table**, Oracle database stores the data for every leaf-level scalar or `REF` attribute in a separate column.

> **Note:** Each `VARRAY` is also stored in a column, unless it is too large. Oracle database 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 8-5 and "Internal Layout of Nested Tables" on page 8-4.

When you retrieve or change attributes of **row object**s in an object table, the database performs the corresponding operations on the columns of the table. Accessing the value of the **row object** itself invokes the default constructor for the type, using the columns of the object table as arguments and produces a copy of the object.

The database stores the system-generated object identifier in a hidden column. The database uses the object identifier to construct `REF`s to the object.

Hidden Columns for Tables with Column Objects

When a table (relational table) is defined with a column of an object type, the database adds hidden columns to the table for the leaf-level attributes of the object type. 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 Object Tables

A substitutable column or object table has a hidden column not only for each attribute of the object type of the column 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.

Besides the type-discriminant column and the null-image column, the following are associated with a substitutable column of `person_typ`, created by Example 8–1

- A hidden column for each of the attributes of `person_typ`: `idno`, `name`, and `phone`
- Hidden columns for attributes of the subtypes of `person_typ`

Thus, the following might be associated with a substitutable column of `person_typ`: the attributes `dept_id` and `major` (for `student_typ`) and `number_hours` (for `part_time_student_typ`).

When you create a subtype, the database automatically adds hidden columns for new attributes in the subtype to tables containing a substitutable column of any of the ancestor types of the new subtype. 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 you drop a subtype using DROP TYPE with the VALIDATE option, the database automatically drops hidden columns for attributes unique to the subtype that do not contain data. Errors are raised if these columns contain data.

Example 8–1 creates types needed for subsequent examples in this chapter

Example 8–1 Creating Types and Inserting in Tables

```
-- drop any of these objects created for Ex.7-10
CREATE TYPE person_typ AS OBJECT (
  idno           NUMBER,
  name           VARCHAR2(30),
  phone          VARCHAR2(20),
  MAP MEMBER FUNCTION get_idno RETURN NUMBER )
  NOT FINAL;
/
CREATE TYPE BODY person_typ AS
  MAP MEMBER FUNCTION get_idno RETURN NUMBER IS
  BEGIN
    RETURN idno;
  END;
END;
/
CREATE TYPE student_typ UNDER person_typ (
    dept_id NUMBER,
    major VARCHAR2(30))
NOT FINAL;
/
CREATE TYPE part_time_student_typ UNDER student_typ (
  number_hours NUMBER);
/
CREATE TYPE employee_typ UNDER person_typ (
    emp_id NUMBER,
    mgr VARCHAR2(30));
/
CREATE TABLE person_obj_table OF person_typ;  // an object table
INSERT INTO person_obj_table
  VALUES (person_typ(12, 'Bob Jones', '650-555-0130'));
INSERT INTO person_obj_table
  VALUES (student_typ(51, 'Joe Lane', '1-650-555-0140', 12, 'HISTORY'));
INSERT INTO person_obj_table
  VALUES (part_time_student_typ(52, 'Kim Patel', '1-650-555-0135', 14,
          'PHYSICS', 20));
```

Substitutable columns are associated with hidden type-discriminant columns. The hidden columns contains an identifier, called a typeid, that identifies the most specific type of each object in the substitutable columns. 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.

Example 8–2 retrieves typeids of object instances stored in the substitutable object table created in Example 8–1:

Example 8–2 Querying for Typeids of Objects Stored in the Table

```
-- Requires Ex. 8-1
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 Also:** "SYS_TYPEID" on page 2-35 for more information about SYS_TYPEID, typeids, and type-discriminant columns.

Storage of REFs

When the database 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 requirements associated with the column, as follows:

- If the column is declared as a REF WITH ROWID, the database 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 decreases due to the omission of the object table metadata and the ROWID. A scoped REF is 16 bytes long.

If the object identifier is primary-key based, the database 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, it is referred to 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. 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 database 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.

See "Nested Table Storage" on page 9-10.

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. However, you cannot construct REFs to objects in a nested table because a nested table row has no object identifier column.

- If the elements are scalars, the storage table contains a single column called COLUMN_VALUE that contains the scalar values.

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.

This section contains the following topics:

- Indexing a Type-Discriminant Column
- Indexing Subtype Attributes of a Substitutable Column

Indexing a Type-Discriminant Column

Using the SYS_TYPEID function, you can build an index on the hidden type-discriminant column of substitutable columns. The type-discriminant column contains typeids that identify the most specific type of every object instance stored in the substitutable column. The system uses this information to evaluate queries that filter by type using the IS OF predicate, 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. The function SYS_TYPEID references the type-discriminant column:

Example 8–3 Create bitmap index on type-discriminant column

```
-- Requires Ex. 8-1
CREATE TABLE contacts (
  contact         person_typ,
  contact_date    DATE );
INSERT INTO contacts VALUES (
  person_typ (65,'Vrinda Mills', '1-650-555-0125'),'24 Jun 2003' );
INSERT INTO contacts VALUES (
  person_typ (12, 'Bob Jones', '650-555-0130'),'24 Jun 2003' );
INSERT INTO contacts VALUES (
  student_typ(51, 'Joe Lane', '1-650-555-0140', 12, 'HISTORY'),'24 Jun 2003' );
INSERT INTO contacts VALUES ( part_time_student_typ(52, 'Kim Patel',
'1-650-555-0135', 14, 'PHYSICS', 20),'24 Jun 2003' );
CREATE BITMAP INDEX typeid_idx ON contacts (SYS_TYPEID(contact));
```

Indexing Subtype Attributes of a Substitutable Column

You can build an index on attributes for any types that can be stored in a substitutable column. You can reference attributes of subtypes in the CREATE INDEX statement by

filtering out types other than the desired subtype (and its subtypes) using the TREAT function; you then use 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:

Example 8–4 Create index on attribute of all students

```
-- Requires Ex.8-1- and 8-3
CREATE INDEX major1_idx ON contacts
  (TREAT(contact AS student_typ).major);
```

The student_typ type 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 persons of type student_typ or parttimestudent_typ (a student_typ subtype). This means that the values of the hidden column are identical to the values returned by the TREAT expression, major values for all students, including student subtypes: both the hidden column and the TREAT expression list majors for students and nulls for non-students. 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 only identical to the values returned by the TREAT expression just described if the type named as the target of the TREAT function (student_typ) is the type that first defined the major attribute. If the target of the TREAT function is a subtype that merely inherited the attribute, as in the following example, the TREAT expression returns 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. Therefore, the database 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

Type evolution is the process of changing a object type. 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 the FINAL and INSTANTIABLE properties of a type

- 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.

This section includes the following topics:

Type Evolution and Dependent Schema Objects

Dependent schema objects of a type are objects that directly or indirectly reference the type and are affected by a change to it.

A type can have these kinds of dependent schema objects: tables; types or subtypes; program units (PL/SQL blocks) such as procedures, functions, packages, and triggers; indextypes; views (including object views); function-based indexes; and operators.

How a dependent schema object is affected by a change to a type depends on the object and on the nature of the change.

- Dependent program units, views, operators, and indextypes are marked invalid when the 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.
- Dependent function-based indexes may be dropped or disabled, depending on the type change, and must be rebuilt.
- Dependent tables have one or more internal columns added for each attribute added to the type, depending on the attribute 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 metadata of the tables and can be performed quickly. However, the data in those tables must be updated to the format of the new type version as well, as discussed in "Options for Updating Data" on page 8-7.

Options for Updating Data

Depending on the amount of data, updating can be time-consuming, 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 8-14. CASCADE itself has the following options that let you choose whether or not to convert table data to the new type format as part of the propagation:

- INCLUDING TABLE DATA: converts the data (default)
- NOT INCLUDING TABLE DATA : does not convert data

By default, the CASCADE option converts the data. In either 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, the database 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 Language Reference for details about type specification and body compilation

Effects of Structural Changes to Types

Structural changes to a type affect dependent data and require the data to be converted. This is not true for changes that are confined to method definitions or behavior (implementation) of the type.

These possible changes to a type are structural:

- Add or drop an attribute
- Modify the length, precision, or scale of an attribute
- Change the finality of a type from FINAL to NOT FINAL or the reverse

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 also affect data storage arrangements and require data conversion.

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 constructor of the type, 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.

Altering a Type by Adding and Dropping Attributes

Example 8–5 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 8–5 Altering an Object Type by Adding and Dropping an Attribute

```
-- Drop person_typ and person_obj_table if they exist
```

```
CREATE TYPE person_typ AS OBJECT (
  idno           NUMBER,
  name           VARCHAR2(30),
  phone          VARCHAR2(20));
/
CREATE TABLE person_obj_table OF person_typ;

INSERT INTO person_obj_table
  VALUES (person_typ(12, 'Bob Jones', '650-555-0130'));

SELECT value(p) FROM person_obj_table p;

VALUE(P)(IDNO, NAME, PHONE)
-------------------------------------------
PERSON_TYP(12, 'Bob Jones', '650-555-0130')
```

You can add the email attribute and drop the phone attribute as follows:

```
ALTER TYPE person_typ
  ADD ATTRIBUTE (email VARCHAR2(80)),
  DROP ATTRIBUTE phone CASCADE NOT INCLUDING TABLE DATA;
```

Then disconnect and reconnect to accommodate the type change:

```
connect oe/oe;
connect hr/hr;
ALTER SESSION SET PLSQL_WARNINGS = 'enable:all';
-- 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 you retrieve a particular user-defined type column in a table often, consider converting that data to the latest type version to eliminate redundant data conversions. Converting is especially beneficial if the column contains VARRAY attributes which typically take more time to convert than objects or nested table columns.

You can convert a column of data by issuing an UPDATE statement to set the column to itself, as indicated in the following code snippet, which is unrelated to previous code.

```
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;
ALTER TABLE person_obj_table UPGRADE INCLUDING DATA;
```

The ALTER TABLE line converts only the table listed. The CASCADE option prevents conversion of other tables or dependents.

Altering a Type by Adding a Nested Table Attribute

This section describes the steps required to make a complex change to a type: the addition of a nested table attribute to an object type that is included in a nested table.

Example 8–6 provides the initial schema which is altered by Example 8–7.

Example 8–6 Initial Schema

```
-- Drop existing person_typ, department_type, people_typ objects or tables
CREATE TYPE person_typ AS OBJECT (
  idno           NUMBER,
  name           VARCHAR2(30),
  phone          VARCHAR2(20));
/
-- creating a nested table type
CREATE TYPE people_typ AS TABLE OF person_typ;
/
CREATE TYPE department_typ AS OBJECT (
  manager    person_typ,
  employee   people_typ);  // a nested table
/
CREATE TABLE department OF department_typ
  NESTED TABLE employee STORE AS employee_store_nt;
```

Example 8–7 starts by creating a new object `tasks_typ` and a nested table type to hold it, `tasks_nttab`.

The following steps, both in Example 8–7, and in other programs, are necessary to add the nested table `tasks` as an attribute to the object type `person_typ`, which is already included in the nested table `people_typ`.

1. Issue an `ALTER TYPE..INVALIDATE` statement to alter the type `person_typ`. This statement bypasses all type and table checks to save time and invalidates dependent objects. You cannot access table data until it is validated.

   The `ALTER TYPE` statement includes `ADD ATTRIBUTE` to add the nested table `tasks`.

   The `UPGRADE.. STORE AS` clause upgrades the affected nested table, and specifies name of the new storage table.

Example 8–7 Altering an Object Type by Adding a Nested Table Attribute

```
-- Requires Ex. 8-6
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` for `person_typ` 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. This validates the table and allow for data access again.

   ```
   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 JPublisher 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. Oracle Type Translator (OTT) and JPublisher do this. If you use another 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. 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 ensures that restrictions relating to the tables and indexes are observed. For example, it ensures that an attribute being dropped is not used as a partitioning key. 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 a single `ALTER TYPE` statement adds multiple attributes, it is done 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.

The following sections contain other notes on type changes including:

- Dropping an Attribute
- Modifying the Length, Precision, or Scale of an Attribute Type
- Dropping a Method
- Modifying the INSTANTIABLE Property

Dropping an Attribute

- Dropping all attributes from a root type is not allowed. Instead, you must 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; therefore, 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.
- When an attribute is dropped, any indexes, statistics, constraints, and referential integrity constraints that reference it are removed.

Modifying the Length, Precision, or Scale of an Attribute Type

- You are not allowed to expand the length of an attribute referenced in a function-based index, clustered key or domain index on a dependent table.
- You are not allowed to decrease the length, precision, or scale of an attribute.

Dropping a Method

- You can only drop a method 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 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 occur when 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 8-13.

Modifying the 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.

Modifying the FINAL Property

- Altering an object type from NOT FINAL to FINAL is only allowed 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.
  - From NOT FINAL to FINAL, you must use CASCADE INCLUDING TABLE DATA.
  - 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, select the CASCADE option based on whether or not you want to insert new subtypes of the altered types into 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 these columns is substitutable, an error is generated. New subtypes of the altered type cannot be inserted into these preexisting columns and tables.

  To alter an object type to NOT FINAL in a way that makes existing columns and tables of the type substitutable (assuming that they are not marked NOT SUBSTITUTABLE), use the CASCADE option CONVERT TO SUBSTITUTABLE.

  Example 8–8 shows the use of CASCADE with the option CONVERT TO SUBSTITUTABLE:

Example 8–8 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, that is, does not ensure that all the ramifications of the type change are legal. Instead, the system marks all dependent schema objects invalid. These objects, including types and tables, are revalidated the

next time they are 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 for reasons such as: the addition of a new attribute to a type increased the number of columns in the table beyond the maximum of 1000, or 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 or not the data is to be converted to the latest type version.

> **Note:** In a system-triggered table validation, the 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 cannot be converted to the latest type version, then `INSERT`, `UPDATE` and `DELETE` statements on the table are not allowed, and the table 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 also fails.

ALTER TYPE Statement for Type Evolution

Table 8–1 lists some of the important options in the `ALTER TYPE` and `ALTER TYPE...CASCADE` statements 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. Use this option to bypass all the type and table checks, and save time. Use this option only if you are certain that problems will not be encountered revalidating 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 without other options, then the `INCLUDING TABLE DATA` option for `CASCADE` is implied, and the database converts all table data to the latest type version.

Table 8–1 (Cont.) ALTER TYPE Options for Type Evolution

Option	Description
INCLUDING TABLE DATA (Option of CASCADE)	Converts data stored in all user-defined columns to the most recent version of the column type. For each new attribute added to the column 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 the table. All tablespaces containing the table data must be in read-write mode; otherwise, the statement will not succeed.
NOT INCLUDING TABLE DATA (Option of CASCADE)	Leaves column data as is, does not change type version. If an attribute is dropped from a type referenced by a table, the corresponding column of the dropped attribute is not removed from the table. However, 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), 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 speeds up the table upgrade because the table data remains in the format of the old type version. However, selecting data from this table requires 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 metadata, it does not require that all tablespaces be on-line in read/write mode for the statement to succeed.
FORCE (Option of CASCADE)	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. Use this option with caution because dependent tables can become inaccessible if some table errors occur.
CONVERT TO SUBSTITUTABLE (Option of CASCADE)	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 type and then marks these existing columns and object tables of the type SUBSTITUTABLE AT ALL LEVELS so that they can store any newly created subtypes of the type. 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 can only store these subtypes in columns and tables created after the type was altered.

See Also: *Oracle Database SQL Language Reference* for further information about ALTER TYPE options

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, see "Altering a Type by Adding a Nested Table Attribute" on page 8-10. See Table 8–1 on page 8-14 for a discussion of the INCLUDING DATA option.

See Also: *Oracle Database SQL Language Reference* for information about `ALTER TABLE` options

System-Defined and User-Defined Constructors

This section discusses various aspects of using system-defined constructors, also known as attribute-value constructors, and user-defined constructors.

This section includes these topics:

The Attribute-Value Constructor

The system-defined constructor, also known as the 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 8–9.

Example 8–9 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 saves you the trouble of defining your own constructors for a type. However, you must supply a value for every attribute declared in the type or the constructor call fails to compile.

This requirement can create a problem if you evolve the type later on, especially because the attribute-value constructor is implicit and not visible in the code, unlike a user-defined constructor. When you change the attributes of a type, the attribute-value constructor of the type changes, too. If you add an attribute, the updated attribute-value constructor expects a value for the new attribute; otherwise, any attribute-value constructor calls in your existing code fail.

See "Type Evolution" on page 8-6.

Advantages of User-Defined Constructors

User-defined constructors do not need to explicitly set a value for every attribute of a type, unlike attribute-value constructors. A user-defined constructor can have any number of arguments, of any type, and these do not need to map directly to type attributes. When you define a constructor, you can initialize the attributes to any appropriate values. For any attributes which you do not supply values, the system initialized 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 are not automatically modified when the type evolves, so their signatures remain 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. You introduce the declaration and the definition with the phrase CONSTRUCTOR FUNCTION and end with the clause RETURN SELF AS RESULT.

A constructor for a type must have the same name as the type. Example 8–10 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 8–10 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, you must declare its mode 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`, such as `SELF.name` in Example 8–10, to distinguish them from the names of the arguments of the constructor function, if these are the same. If the argument names are different, this qualification is not necessary.

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

You can overload user-defined constructors, like other type methods.

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 attributes of the type. The mode of the parameters (after the implicit `SELF` parameter) of the user-defined constructor must be `IN`.

If an attribute-value constructor is not hidden by a user-defined constructor that has 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 attribute-value constructor of the type changes accordingly. This can cause a formerly hidden attribute-value constructor to become usable again.

Calling User-Defined Constructors

You call a user-defined constructor like any other function and you can use it 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 8–11 creates a subtype under the type created in Example 8–10 and shows examples for calling the user-defined constructors.

Example 8–11 Calling User-Defined Constructors

```
-- Requires Ex. 8-10
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 8–12 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 database 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 database 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.

Table 8–2 (Cont.) Generic SQL Types

Type	Description
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. The following cannot be stored in an ANYDATA column: ▪ Another opaque type such as ANYDATA or XMLTYPE ▪ LOB types (BLOB/CLOB/NCLOB) ▪ VARRAY types with maximum size greater than 4K ▪ ADTs that contain any of the above types
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 8–13 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 8–13 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`: corresponds to `SYS.ANYTYPE`
- `OCIAnyData`: corresponds to `SYS.ANYDATA`
- `OCIAnyDataSet`: corresponds 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 database 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 the Oracle database built-in aggregates. Once such functions are registered with the server, the database 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

How Locators Improve the Performance of Nested Tables

You can use nested table locators to improve performance when retrieving data.

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 database interfaces, such as OCI.

Generally, when the client accesses a nested table explicitly or implicitly (by fetching the containing object), the database 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 database handles this case for you by using a locator instead of the actual 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.

> **See Also:** "Nested Table Locators" on page 9-13 for more specific information

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.

This chapter covers the following topics:

General Storage Considerations for Objects

This section discusses general storage considerations for various object types.

This section contains the following topics:

Storing Objects as Columns or Rows

You can store objects in relational tables as column objects or in object tables as row objects. Those objects that have meaning outside of the relational database they reside in, should be made referenceable as row objects in an object table. Otherwise, they should be stored as column objects in a relational table.

See "How Objects are Stored in Tables" on page 1-6 for an introduction to table storage.

This section describes the following topics:

- Column Object Storage in Relational Tables
- Row Object Storage in Object Tables

Column Object Storage in Relational Tables

The storage of a **column object** is similar to the storage of an equivalent set of scalar columns that collectively make up the object. The difference is the additional overhead of maintaining the atomic null values of any noncollection columns objects and their **embedded object attribute**s. These values, called null indicators (or sometimes, null images), specify for every column object, whether or not the column object is null and whether or not each of its embedded object attributes is null.

Note that 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 numbers and an object `employee_objtyp` that contains the name and address objects. 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, enter the SQL statements in Example 9–1 to create the four object types and a table for phone number objects.

Example 9–1 Creating Object Types for Columns in a Relational 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        VARCHAR2(2),
  zipcode      VARCHAR2(20));
NOT FINAL;
/
CREATE TYPE phone_objtyp AS OBJECT (
  location     VARCHAR2(15),
  num          VARCHAR2(14));
/

CREATE TYPE employee_objtyp AS OBJECT (
  name name_objtyp;
  address address_objtyp;
/

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

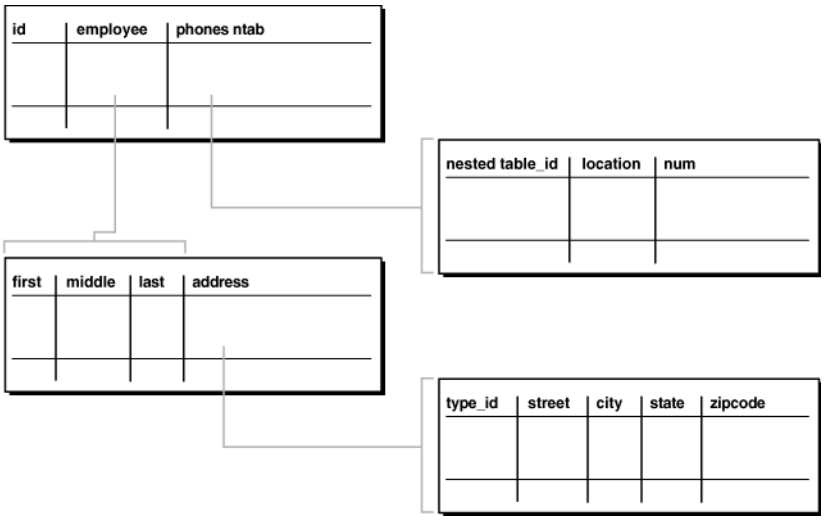
See Also: "Design Considerations for Nested Tables" on page 9-10 for more information about nested tables

Next, create a table to hold the information about the people in the organization with the SQL statement in Example 9–2. This statement also creates an id for people in the organization.

Example 9–2 Creating a Table with Column Objects

```
CREATE TABLE people_reltab (
  id            NUMBER(4)   CONSTRAINT pk_people_reltab PRIMARY KEY,
  employee      employee_objtyp
  phones_ntab   phone_ntabtyp)
  NESTED TABLE  phones_ntab STORE AS phone_store_ntab;
```

Figure 9–1 Representation of the people_reltab Relational Table



The `people_reltab` table has two column objects: `employee` and `phones_ntab`. The `phones_ntab` column object is a nested table, a collection type of column object.

The storage for each object in the `people_reltab` table is that of the attributes of the object plus overhead for the null indicator.

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. There is one null indicator column for each noncollection column object, `name_obj` and `address_obj`. The null indicator column length is one byte, as one bit represents the object itself, which translates to `CEIL(1/8)` or `1`.

Since the null indicator is one byte in size, the overhead of null information for each row of the `people_reltab` table is two bytes, one for each **object column**.

Every noncollection object has a null indicator column, whether or not the object is `FINAL`. The columns in these examples are `FINAL`.

> **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)

This section discusses the two types of object identifiers for row objects in object tables and how they are stored and referenced.

An object identifier (OID) allows the corresponding row object to be referred to and from other objects or from relational tables. A built-in data type called a `REF` represents such references. `REF`s use object identifiers (OIDs) to point to row objects.

You can use either system-generated OIDs or primary-key based OIDs.

> **See Also:** "Using References to Row Objects" on page 1-7

System-Generated Object Identifiers (OIDs)

System-generated OIDs are the default for row objects in an object table.

Oracle assigns to 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 environment, the system-generated unique identifier lets Oracle identify objects unambiguously.

The object identifier column is a hidden column that Oracle uses to construct references to the row objects. Oracle provides no access to the internal structure of object identifiers. This structure can change at any time. Applications are only concerned with using object references for fetching and navigating objects.

Primary-Key Based Object Identifiers (OIDs)

Oracle allows the option of specifying the primary key value of a row object as its object identifier, if there is a primary key column.

Instead of using the system-generated OIDs, you use a `CREATE TABLE` statement with this clause, `OBJECT IDENTIFIER IS PRIMARY KEY`. This specifies that the system use the primary key column(s) as the OIDs of the objects in the table. That way, 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.

> **See Also:** Oracle Database SQL Language Reference for further information on `OBJECT IDENTIFER` syntax

You can enforce referential integrity on columns that store references to these row objects in a way similar to foreign keys in relational tables.

Note: 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.

System-Generated Versus Primary-Key Based OIDs

Primary-key based identifiers 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. If you use system-generated OIDs for an object table, Oracle maintains an index on the column that stores these OIDs. A system-generated OID requires extra storage space for this index and an extra 16 bytes of storage for each row object.

However, if each primary key value requires more than 16 bytes of storage and you have a large number of REFs, using the primary key might require more space than system-generated OIDs because each REF is the size of the primary key.

Performance of Object Comparisons

You can compare objects by invoking either a map or order method. A map method converts objects into scalar values while preserving the ordering of the objects. Using a map method is preferable because it allows the system to efficiently order objects.

> **Note:** For any one object type, you can implement either a map or an order method, but not both. Neither are required.

The way objects are mapped has significant performance implications when sorting the objects using ORDER BY or GROUP BY processes. 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 (the map method). If the comparison semantics are extremely complex, or if the objects cannot be mapped to 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.

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 LOB or ANYDATA 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

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 primary key based (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 upon selection, 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 9-3, 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

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

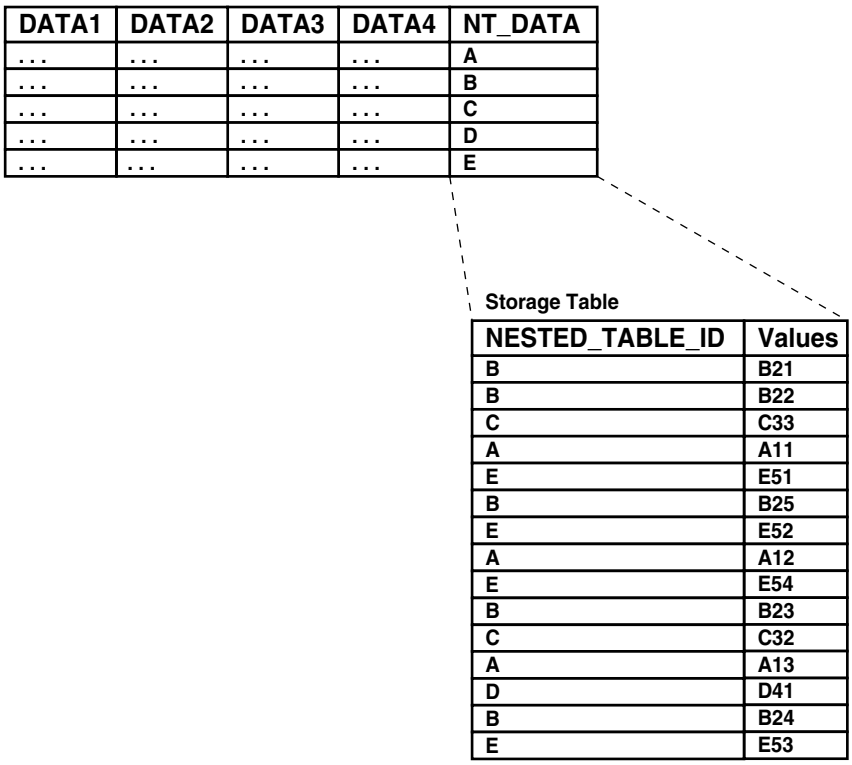
The following sections contain design considerations for using nested tables.

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

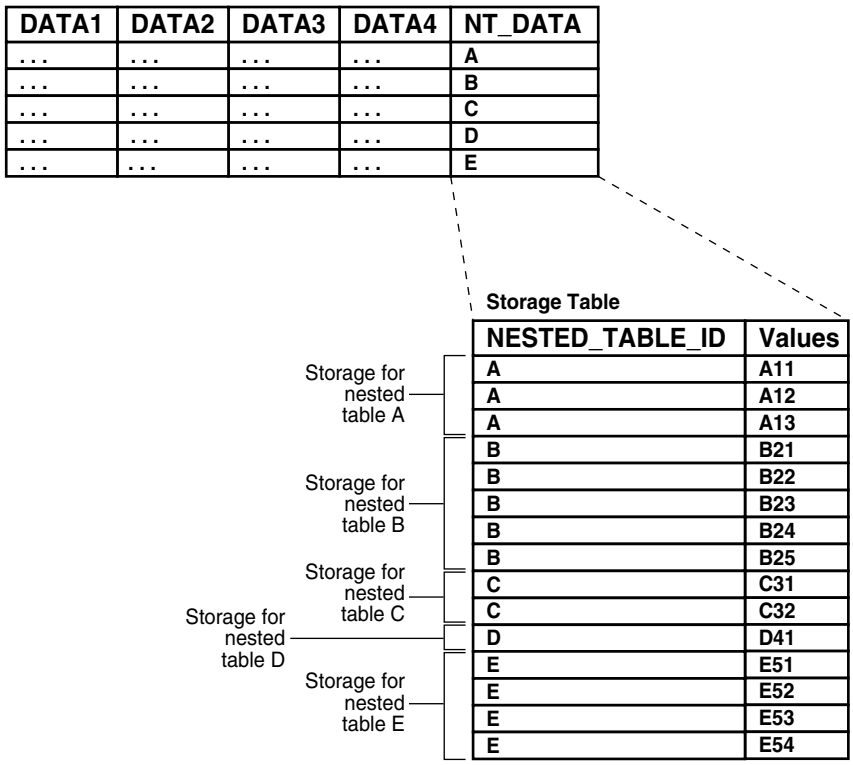


Nested Table in an Index-Organized Table (IOT)

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 the values for each nested table in a nested table column by `NESTED_TABLE_ID`. 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.

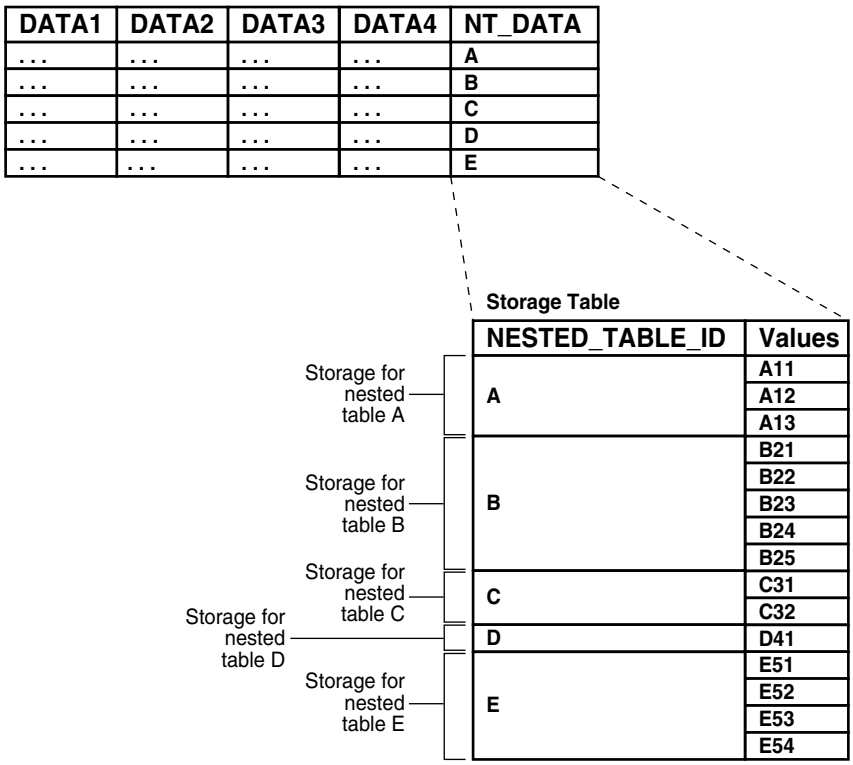
Figure 9–3 Nested Table in IOT Storage



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



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.

The following topics specify ways that a collection can be retrieved as a locator:

- At Table Creation Time
- As a HINT During Retrieval

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 5, "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 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:

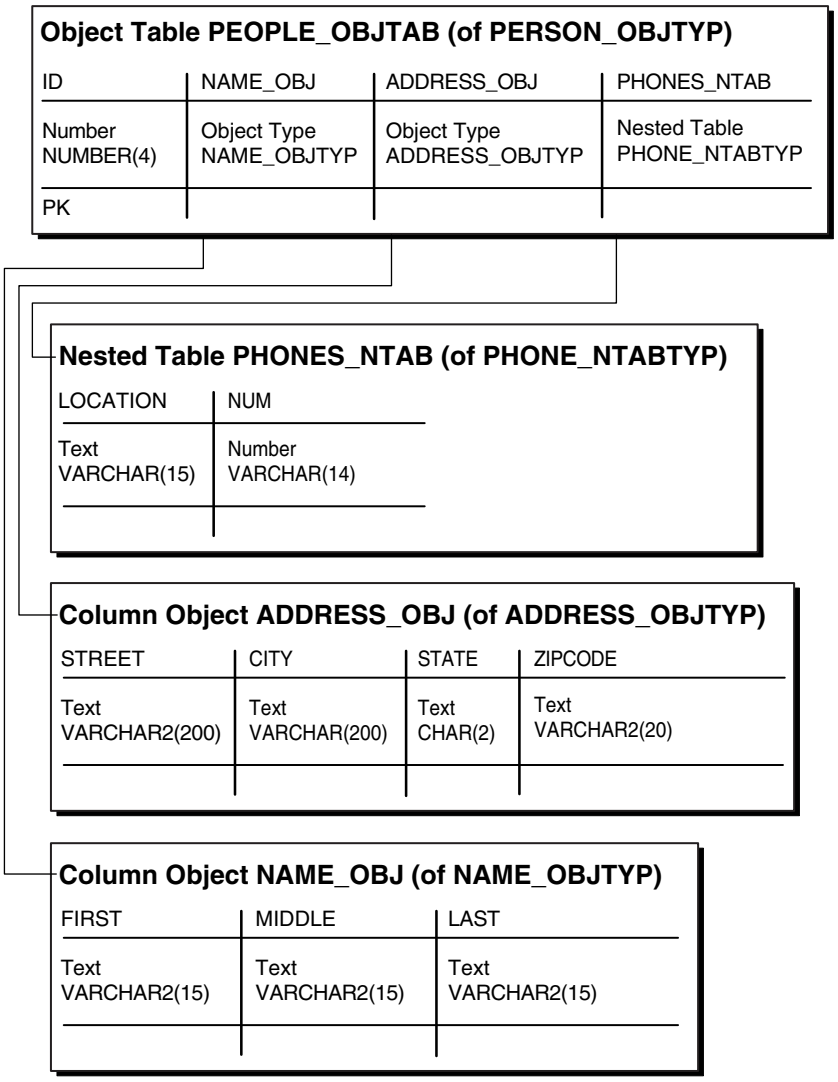
Example 9–8 Creating an Object Table with a Multilevel Collection

```
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. The difference is that the people_objtab is an object table with row objects, while the people_reltab table is a relational table with column objects.

> **See Also:** "Column Object Storage in Relational Tables" on page 9-2

Figure 9–5 Object-Relational Representation of the people_objtab Object Table



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 in 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:

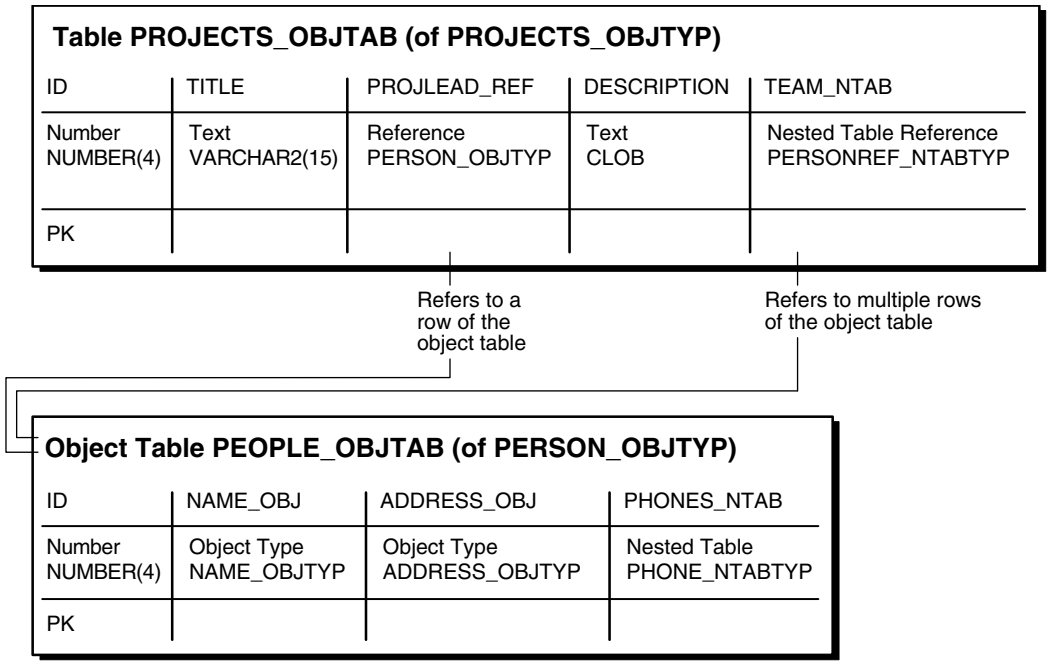
```
CREATE TYPE personref_ntabtyp AS TABLE OF REF person_objtyp;
/
```

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



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 `REF`s 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', '650.555.0141'),
      phone_objtyp('work', '510.555.0122'))) ;

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', '415.555.0143'),
      phone_objtyp('work', '650.555.0192'))) ;

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', '510.555.0101'),
      phone_objtyp('work', '650.555.0178'),
      phone_objtyp('cell', '650.555.0143'))) ;
```

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 4, "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 2-8.

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 Language 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;
Enter password: password
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;
Enter password: password
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;
Enter password: password
CREATE TABLE atab OF HR.atype;

BEGIN
  HR.atype.newa(1, 'atab', 'OE');
END;
/
DROP TABLE atab;
CONNECT hr;
Enter password: password
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.

This section contains the following topics:

- Replicating Columns of Object, Collection, or REF Type
- Replicating Object Tables

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.

This section contains the following topics:

- Pushing a Type Change Out to Clients
- Changing Default Constructors
- Altering the FINAL Property of a Type

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 8-17.

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.

This section contains the following topics:

Deciding Whether to Evolve a Type or Create a Subtype

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 Design Tips

This section discusses miscellaneous tips for designing with Oracle objects.

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++, Oracle Objects for OLE (OO4O), or Oracle Data Providers for .NET (ODP.NET) 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 Language 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*
- *Oracle Data Provider for .NET Developer'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:

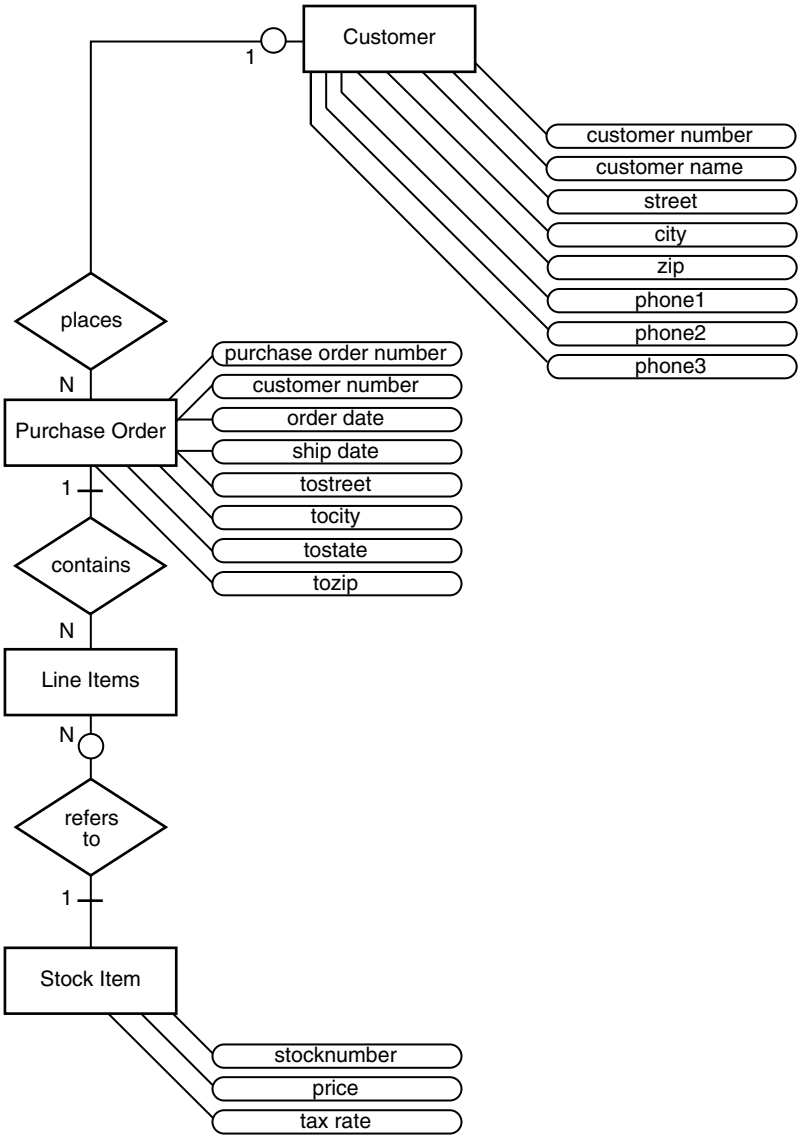
- Customers
- The stock of products for sale
- Purchase orders

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 **normalize** 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',
         '415-555-0102', NULL, NULL);

INSERT INTO Customer_reltab
  VALUES (2, 'John Nike', '323 College Drive',
         'Edison', 'NJ', '08820',
         '609-555-0190', '201-555-0140', 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.

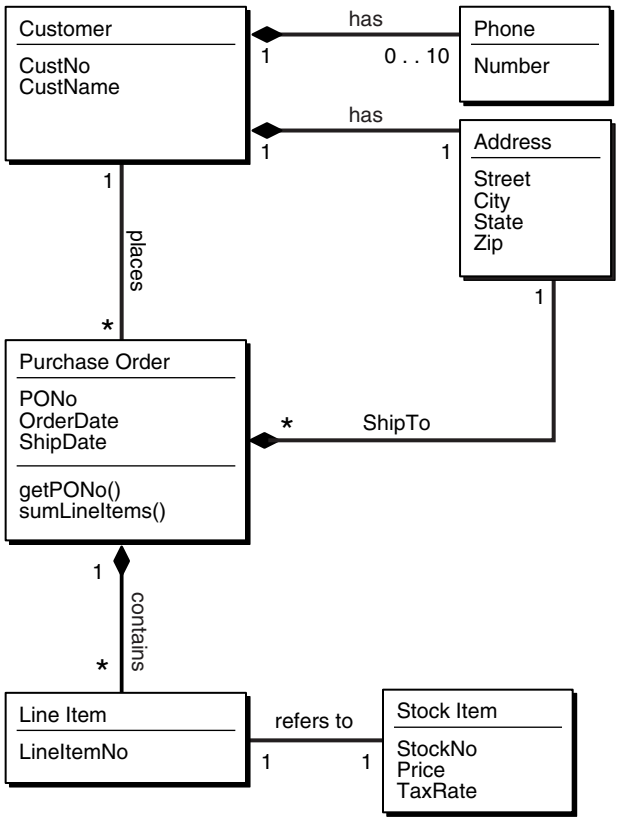
Example A–13 Delete Purchase Order 1001 under the Relational Model

```
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" on page 2-1 for more information about map and order methods
- *Oracle Database PL/SQL Language 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 `REF`s 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;
```

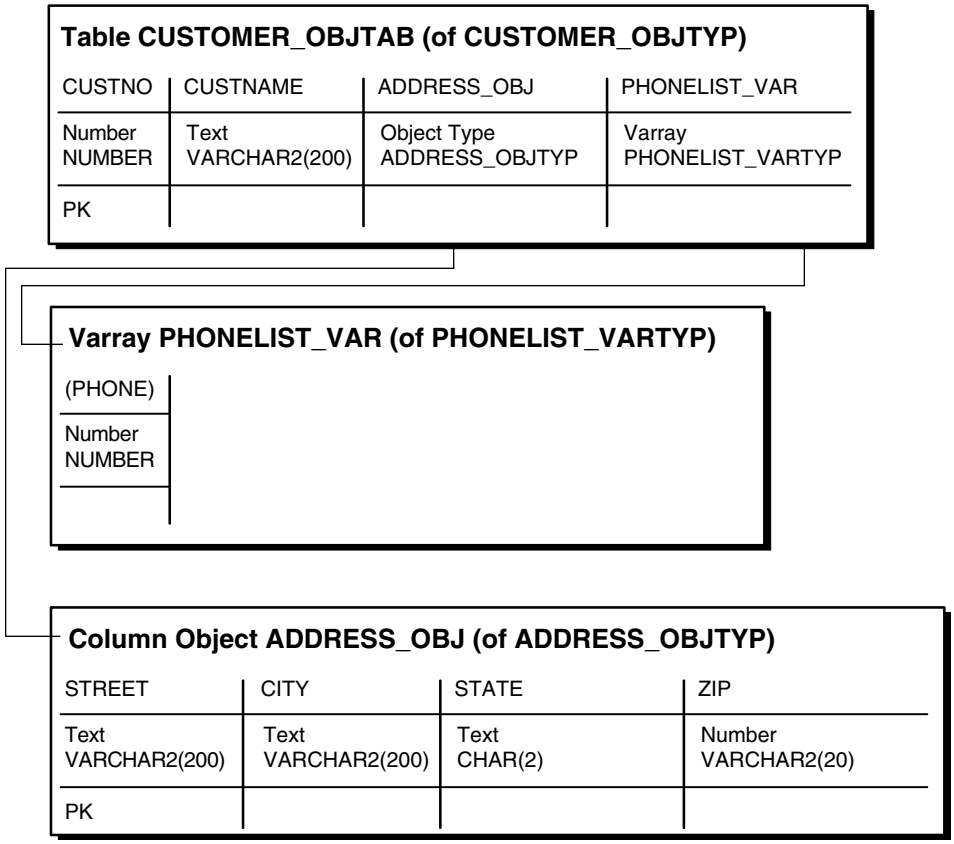
Unlike relational tables, when you create an object table, you specify a data type for it, namely, the type of objects it will contain.

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



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 unique OID of the row object.

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:

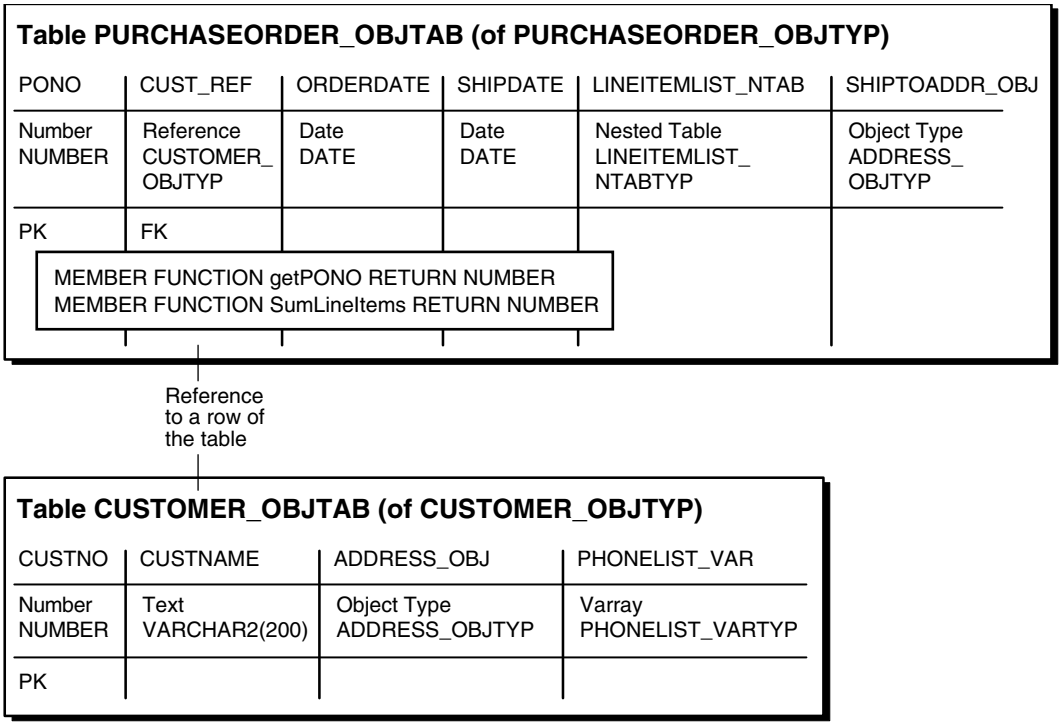
`CREATE TABLE PurchaseOrder_objtab OF PurchaseOrder_objtyp (`

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



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.

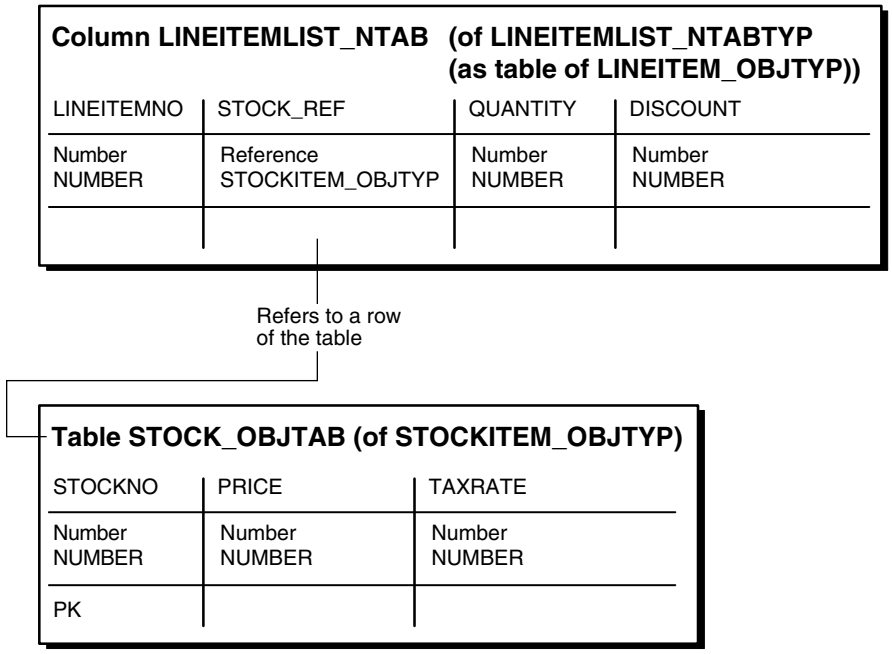
After the table is created, the ALTER TABLE statement is issued to add the SCOPE FOR constraint on a REF. The SCOPE FOR constraint on a REF is not allowed in a CREATE TABLE statement. To specify that Stock_ref can reference only the object table Stock_objtab, issue the following ALTER TABLE statement 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

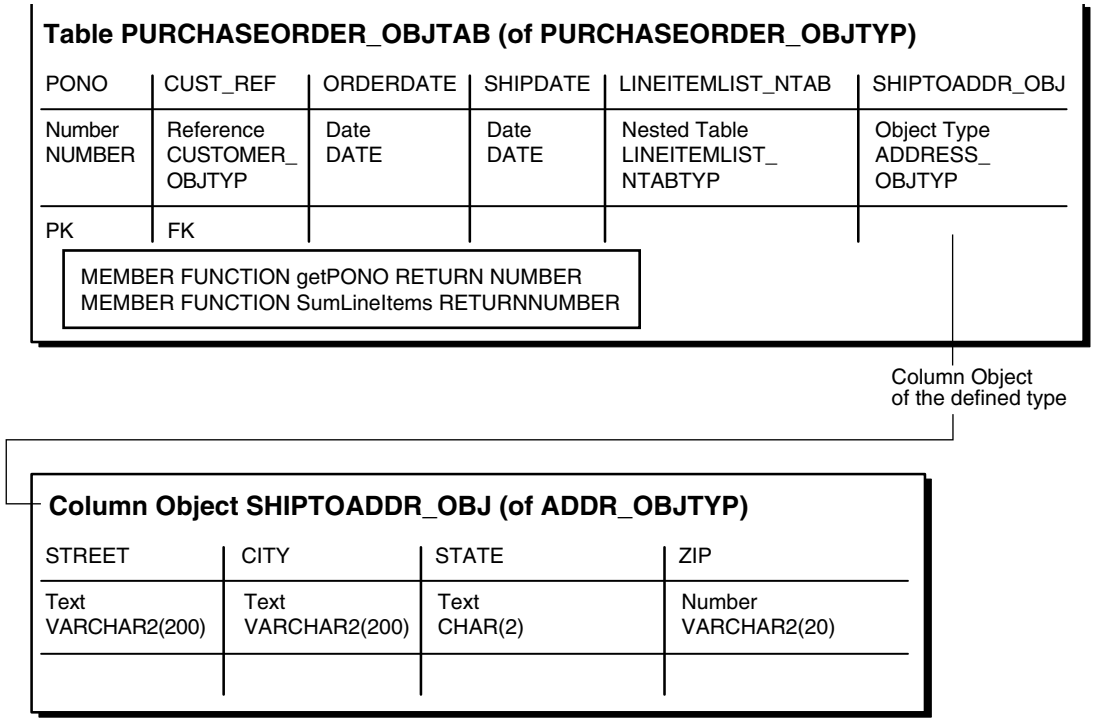


Oracle does not support a referential constraint specification for storage tables. In this situation, specifying the `SCOPE` clause for a `REF` column is useful. In general, specifying scope or referential constraints for `REF` columns has several benefits:

- 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 `REF`s 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



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('415-555-0102')
    ) ;

INSERT INTO Customer_objtab
  VALUES (
    2, 'John Nike',
    Address_objtyp('323 College Drive', 'Edison', 'NJ', '08820'),
    PhoneList_vartyp('609-555-0190','201-555-0140')
    ) ;
```

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:** If 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.

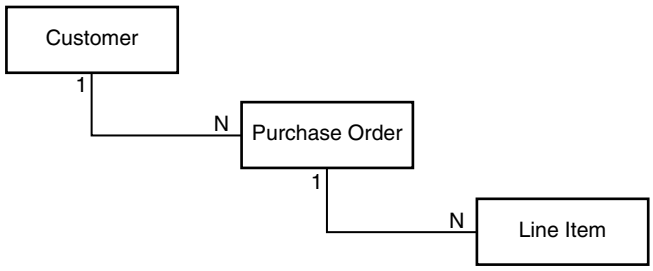
One way to do this is to change the `Customer_objtyp` so that information about a customer's purchase orders is included right in the object instance that represents that customer. In other words, we want to add an attribute for purchase order information to `Customer_objtyp`. To hold information about 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
B8C834000000142601000100010029000000000090600812A00078401FE0000000B03C20B050000
...
```

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 8-6.

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('650-555-0188'),
    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('415-555-0198', '415-555-0199'),
      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 2-32.

Glossary

atomically null object

An object whose value is `NULL` is called atomically null. An atomically null object is different from an object that has null values for all its attributes.

Binary Large Object (BLOB)

A large object data type whose value consists of raw binary data.

character large object (CLOB)

The large object (LOB) data type whose value is composed of character data corresponding to the database character set.

column object

An object that is stored as a column of a relational database table (as opposed to an object table). A column object can also be an attribute of another object. A column object is also known as a stored inline object or an embedded object.

embedded object attribute

An attribute of a **column object**.

dynamic method dispatch

A method call that is dispatched at run-time to the nearest method implementation when there are multiple implementations of the same method using overriding.

index-organized table IOT

A table organized by its index.

leaf-level scalar object

An object that is not a collection and is not composed of other types.

leaf-level scalar attribute

An attribute of a **leaf-level scalar object**.

literal invocation

An invocation where all arguments are literals or invocations of literal methods. Arguments cannot be variables.

materialized view

A view that contains both the query and its results.

multilevel collection type

Ccollection types whose elements are collection types, either directly or indirectly.

multiset operators

An operator that combines elements of two nested tables into a single nested table.

nested table

An unordered set of data elements of the same data type.

normalize

The process of removing redundancy in data by separating the data into multiple tables.

object column

A column of user-defined types or abstract data types (ADT)s .

object identifier

Identifier for a row object which can be either system-generated (default) or based on a primary key using the `CREATE TABLE` statement.

object instance

An instance of an object type. Also referred to as an object.

object table

A table in which each row represents an object. See **row object**.

object type

The type of the object instance. It is similar to a record that has methods. Object types are user-defined.

outer table

A table that contains a nested table.

pinning

Fetching.

PL/SQL

The Oracle procedural language extension to SQL.

polymorphism

Allows handling data types and methods generically using the same interface. In polymorphic overriding, subtypes redefine a method they have inherited. In polymorphic overloading, there may be several versions of the same method, with different parameters.

primary key

The column or set of columns specified in the PRIMARY KEY constraint of a table.

REF

An Oracle built-in data type that encapsulates references to row objects of a specified object type.

row object

An object that is stored in a complete row in an **object table**.

specializing

Adding new attributes or methods to a subtype that the parent supertype does not have, or changing the implementation of a method or methods.

stored procedure

A PL/SQL block that is stored in the database and can be executed from an application.

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.

type evolution

The modification of a subtype of a type, typically using an `ALTER` statement.

VARRAY

An ordered set of data elements, that are of the same data type or a subtype of the declared data type.

Index

A

B

C

D

E

F

G

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X