Oracle9*i*

Application Developer's Guide - Object-Relational Features

Release 2 (9.2)

March 2002

Part No. A96594-01

Oracle9*i* Application Developer’s Guide - Object-Relational Features, Release 2 (9.2)

Part No. A96594-01

Primary Author: William Gietz

Contributing Author: C. Dupree

Contributors: G. Arora, C. Iyer, H. Yeh, A. Manikutty, A. Yoaz, Q. Yu

Contents

Send Us Your Comments

Oracle9*i* Application Developer's Guide - Object-Relational Features, Release 2 (9.2)

Part No. A96594-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this document. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most?

If you find any errors or have any other suggestions for improvement, please indicate the document title and part number, and the chapter, section, and page number (if available). You can send comments to us in the following ways:

- Electronic mail: infodev_us@oracle.com
- FAX: (650) 506-7227 Attn: Server Technologies Documentation Manager
- Postal service:
  Oracle Corporation
  Server Technologies Documentation
  500 Oracle Parkway, Mailstop 4op11
  Redwood Shores, CA 94065
  USA

If you would like a reply, please give your name, address, telephone number, and (optionally) electronic mail address.

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

Preface

Oracle9i Application Developer's Guide - Object-Relational Features describes how to use the object-relational features of the Oracle Server, Release 2.0.2. Information in this guide applies to versions of the Oracle Server that run on all platforms, and does not include system-specific information.

This preface contains these topics:

Audience

Oracle9i Application Developer's Guide - Object-Relational Features is intended for programmers developing new applications or converting existing applications to run in the Oracle environment. The object-relational features are often used in multimedia, Geographic Information Systems (GIS), and similar applications that deal with complex data. The object views feature can be valuable when writing new applications on top of an existing relational schema.

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

Organization

This document contains:

Chapter 1, "Introduction to Oracle Objects"

Introduces the key features and explains the advantages of the object-relational model.

Chapter 2, "Basic Components of Oracle Objects"

Explains the basic concepts and terminology that you need to work with Oracle Objects.

Chapter 3, "Object Support in Oracle Programming Environments"

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

Chapter 4, "Managing Oracle Objects"

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

Chapter 5, "Applying an Object Model to Relational Data"

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

Chapter 6, "Advanced Topics for Oracle Objects"

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

Chapter 7, "Frequently Asked Questions About Using Oracle Objects"

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

Chapter 8, "Design Considerations for Oracle Objects"

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

Chapter 9, "A Sample Application Using Object-Relational Features"

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

Related Documentation

For more information, see these Oracle resources:

- *PL/SQL User's Guide and Reference* to learn PL/SQL and to get a complete description of this high-level programming language, which is Oracle Corporation's procedural extension to SQL
- *Oracle9i Application Developer's Guide - Fundamentals* for general information about developing applications
- *Oracle9i JDBC Developer's Guide and Reference* and *Oracle9i Java Stored Procedures Developer's Guide* to use Oracle's object-relational features through Java
- *Oracle Call Interface Programmer's Guide* describes how to use the the Oracle Call Interface (OCI) to build third-generation language (3GL) applications that access the Oracle Server
- *Pro*C/C++ Precompiler Programmer's Guide* for information on Oracle's Pro* series of precompilers, which allow you to embed SQL and PL/SQL in 3GL application programs written in Ada, C, C++, COBOL, or FORTRAN
- Oracle Developer/2000 is a cooperative development environment that provides several tools including a form builder, reporting tools, and a debugging environment for PL/SQL. If you use Developer/2000, then refer to the appropriate Oracle Tools documentation.

- *Oracle9i SQL Reference* and *Oracle9i Database Administrator's Guide* for information on SQL
- *Oracle9i Database Concepts* for information on basic Oracle concepts

Many books in the documentation set use the sample schemas of the seed database, which is installed by default when you install Oracle. Refer to *Oracle9i Sample Schemas* for information on how these schemas were created and how you can use them yourself.

In North America, printed documentation is available for sale in the Oracle Store at

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

Customers in Europe, the Middle East, and Africa (EMEA) can purchase documentation from

`http://www.oraclebookshop.com/`

Other customers can contact their Oracle representative to purchase printed documentation.

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://otn.oracle.com/admin/account/membership.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://otn.oracle.com/docs/index.htm`

To access the database documentation search engine directly, please visit

`http://tahiti.oracle.com`

Conventions

This section describes the conventions used in the text and code examples of this documentation set. It describes:

- Conventions in Text
- Conventions in Code Examples

Conventions in Text

We use various conventions in text to help you more quickly identify special terms. The following table describes those conventions and provides examples of their use.

Convention	Meaning	Example
Bold	Bold typeface indicates terms that are defined in the text or terms that appear in a glossary, or both.	When you specify this clause, you create an **index-organized table**.
Italics	Italic typeface indicates book titles or emphasis.	*Oracle9i Database Concepts* Ensure that the recovery catalog and target database do *not* reside on the same disk.
`UPPERCASE monospace (fixed-width) font`	Uppercase monospace typeface indicates elements supplied by the system. Such elements include parameters, privileges, datatypes, RMAN keywords, SQL keywords, SQL*Plus or utility commands, packages and methods, as well as system-supplied column names, database objects and structures, usernames, and roles.	You can specify this clause only for a `NUMBER` column. You can back up the database by using the `BACKUP` command. Query the `TABLE_NAME` column in the `USER_TABLES` data dictionary view. Use the `DBMS_STATS.GENERATE_STATS` procedure.
`lowercase monospace (fixed-width) font`	Lowercase monospace typeface indicates executables, filenames, directory names, and sample user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as user-supplied database objects and structures, column names, packages and classes, usernames and roles, program units, and parameter values. **Note:** Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.	Enter `sqlplus` to open SQL*Plus. The password is specified in the `orapwd` file. Back up the datafiles and control files in the `/disk1/oracle/dbs` directory. The `department_id`, `department_name`, and `location_id` columns are in the `hr.departments` table. Set the `QUERY_REWRITE_ENABLED` initialization parameter to `true.` Connect as `oe` user. The `JRepUtil` class implements these methods.
`lowercase italic monospace (fixed-width) font`	Lowercase italic monospace font represents placeholders or variables.	You can specify the *`parallel_clause`*. Run `U`*`old_release`*`.SQL` where *`old_release`* refers to the release you installed prior to upgrading.

Conventions in Code Examples

Code examples illustrate SQL, PL/SQL, SQL*Plus, or other command-line statements. They are displayed in a monospace (fixed-width) font and separated from normal text as shown in this example:

```
SELECT username FROM dba_users WHERE username = 'MIGRATE';
```

The following table describes typographic conventions used in code examples and provides examples of their use.

Convention	Meaning	Example
`[ ]`	Brackets enclose one or more optional items. Do not enter the brackets.	`DECIMAL (`*`digits`*` [ , `*`precision`*` ])`
`{ }`	Braces enclose two or more items, one of which is required. Do not enter the braces.	`{ENABLE | DISABLE}`

`|`	A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar.	`{ENABLE | DISABLE}`
		`[COMPRESS | NOCOMPRESS]`

`...`	Horizontal ellipsis points indicate either:
	▪ That we have omitted parts of the code that are not directly related to the example
	▪ That you can repeat a portion of the code
		`CREATE TABLE ... AS `*`subquery`*`;`
		
		`SELECT `*`col1`*`, `*`col2`*`, ... , `*`coln`*` FROM employees;`

.
.
.
	Vertical ellipsis points indicate that we have omitted several lines of code not directly related to the example.	`SQL> SELECT NAME FROM V$DATAFILE;`
		`NAME`
		`------------------------------------`
		`/fsl/dbs/tbs_01.dbf`
		`/fs1/dbs/tbs_02.dbf`
		`.`
		`.`
		`.`
		`/fsl/dbs/tbs_09.dbf`
		`9 rows selected.`

Other notation	You must enter symbols other than brackets, braces, vertical bars, and ellipsis points as shown.	`acctbal NUMBER(11,2);`
		`acct    CONSTANT NUMBER(4) := 3;`

Convention	Meaning	Example
Italics	Italicized text indicates placeholders or variables for which you must supply particular values.	`CONNECT SYSTEM/`*system_password* `DB_NAME =` *database_name*

UPPERCASE	Uppercase typeface indicates elements supplied by the system. We show these terms in uppercase in order to distinguish them from terms you define. Unless terms appear in brackets, enter them in the order and with the spelling shown. However, because these terms are not case sensitive, you can enter them in lowercase.	`SELECT last_name, employee_id FROM employees;`
		`SELECT * FROM USER_TABLES;`
		`DROP TABLE hr.employees;`

lowercase	Lowercase typeface indicates programmatic elements that you supply. For example, lowercase indicates names of tables, columns, or files.
	Note: Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.
		`SELECT last_name, employee_id FROM employees;`
		`sqlplus hr/hr`
		`CREATE USER mjones IDENTIFIED BY ty3MU9;`

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Standards will continue to evolve over time, and Oracle Corporation 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 additional information, visit the Oracle Accessibility Program Web site at

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

Accessibility of Code Examples in Documentation JAWS, a Windows screen reader, 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, JAWS 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 Corporation does not own or control. Oracle Corporation neither evaluates nor makes any representations regarding the accessibility of these Web sites.

What’s New in Object-Relational Features

This section describes the new object-relational features of Oracle9*i* Release 2 (9.2) - BETA.

Oracle 9*i* Release 2 (9.2) - BETA New Object-Relational Features

- **Type synonyms**

  Synonyms can be defined for user-defined types so that a type can be used without having to qualify its name with the name of the schema in which the type was defined.

  See Also: "Synonyms for User-Defined Types" on page 4-9

- **User-defined constructors**

  User-defined constructor functions make possible custom initialization of newly created object instances. They also make it possible to evolve a type without having to update calls to constructors in existing code to accommodate a newly added attribute.

  See Also: "User-Defined Constructors" on page 6-21

Oracle9*i* Release 1 (9.0.1) New Object-Relational Features

- **SQL type inheritance**

  Specialized versions of user-defined types can be defined as subtypes in a SQL type hierarchy.

  See Also: Chapter 2, section "Type Inheritance"

- **Object view hierarchies**

  Hierarchies can be created of object views based on some or all of the types in a type hierarchy. Object view hierarchies simplify targeting a particular subtype (and perhaps its subtypes) in queries and other operations.

  See Also: Chapter 5, section "Object View Hierarchies"

- **Type evolution**

  User-defined SQL types can be changed, or evolved, instead of having to be re-created.

  See Also: Chapter 6, section "Type Evolution"

- **User-defined aggregate functions**

  Custom aggregate functions can be defined for working with complex data.

  > **See Also:** Chapter 6, section "User-Defined Aggregate Functions"

- **Generic and transient datatypes**

  External procedures can be given fields or parameters of a generic type that can contain values of any scalar or user-defined type, making it unnecessary to implement multiple versions of the same external procedure just to handle multiple datatypes.

  > **See Also:** Chapter 6, section "Transient and Generic Types"

- **Function-based indexes**

  Function-based indexes can be built on type method functions.

  > **See Also:** Chapter 8, section "Function-Based Indexes on the Return Values of Type Methods"

- **Multi-level collections**

  Collections (varrays and nested tables) can contain elements that are themselves collections or have attributes that are.

  > **See Also:** Chapter 2 and Chapter 5

- **C++ interface to Oracle**

  A C++ interface (OCCI) to Oracle, built on top of OCI, enables you to use the object-oriented features, native classes, and methods of the C++ programing language to access the Oracle database.

  > **See Also:** Chapter 3, section "Oracle C++ Call Interface (OCCI)"

- **Java object storage**

  You can now create SQL types mapped to existing Java classes to provide persistent storage for Java objects. Such SQL types are called SQL types of Language Java, or SQLJ types.

See Also: Chapter 3, section "Java: JDBC, Oracle SQLJ, JPublisher, and SQLJ Object Types"

1 Introduction to Oracle Objects

This chapter describes the advantages and key features of the Oracle9*i* object-relational model. The chapter contains these topics:

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

About Oracle Objects and Object Types

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

Oracle object technology is a layer of abstraction built on Oracle’s relational technology. New object types can be created from any built-in database types and any previously created object types, object references, and collection types. Metadata for user-defined types is stored in a schema that is available to SQL, PL/SQL, Java, and other published interfaces.

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

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

Advantages of Objects

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

Below are listed several other specific advantages that objects offer over a purely relational approach.

Objects Can Encapsulate Operations Along with Data

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

Objects Are Efficient

Using object types makes for greater efficiency:

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

Objects Can Represent Part-Whole Relationships

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

Objects Are Organic

Object types let you capture the "thingness" of an entity, that is, the fact that its parts make up a whole. For example, an address may need to contain a number, street, city, state, and zip code. If any of these elements is missing, the address is incomplete. Unlike an address object type, a relational table cannot express that the columns in the table collectively make up an organic whole.

Key Features of the Object-Relational Model

Oracle implements the object-type system as an extension of the relational model. The object-type interface continues to support standard relational database functionality such as queries (`SELECT…FROM…WHERE`), fast commits, backup and recovery, scalable connectivity, row-level locking, read consistency, partitioned tables, parallel queries, cluster database, export/import, loader, and so forth. But SQL and various programmatic interfaces to Oracle—including PL/SQL, Java, Oracle Call Interface, Pro*C/C++, OO4O—have been enhanced with new extensions to support objects. The result is an object-relational model, which offers the intuitiveness and economy of an object interface while preserving the high concurrency and throughput of a relational database.

Type Inheritance

Type inheritance adds to the usefulness of objects by enabling you to create type hierarchies by defining successive levels of increasingly specialized subtypes that derive from a common ancestor object type. Derived subtypes inherit the features of the parent object type but extend the parent type definition. For example, specialized types of customers—a corporate Customer type or a government Customer type—might be derived from a general Customer object type. The specialized types can add new attributes 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.

Type Evolution

Using an `ALTER TYPE` statement, you can modify—"evolve"—an existing user-defined type to make the following changes:

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

Dependencies of a type to be changed are checked using essentially the same validations applied for a `CREATE TYPE` statement. If a type or any of its dependent types fails the type validations, the ALTER TYPE statement rolls back; no new type version is created, and dependent schemna objects remain unchanged.

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

Object Views

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

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

SQL Object Extensions

To support the new object-related features, SQL extensions—including new DDL—have been added to create, alter, or drop object types; to store object types in tables; and to create, alter, or drop object views. There are DML and query extensions to support object types, references, and collections.

PL/SQL Object Extensions

PL/SQL is Oracle's database programming language that is tightly integrated with SQL. With the addition of user-defined types and other SQL types introduced in Oracle8*i*, PL/SQL has been enhanced to operate on user-defined types seamlessly. Thus, application developers can use PL/SQL to implement logic and operations on user-defined types that execute in the database server.

Java Support for Oracle Objects

Oracle's Java VM is tightly integrated with the RDBMS and supports access to Oracle Objects through object extensions to Java Database Connectivity (JDBC), which provides dynamic SQL, and SQLJ, which provides static SQL. Thus, application developers can use the Java to implement logic and operations on user-defined types that execute in the database server. With Oracle9*i*, you can now also create SQL types mapped to existing Java classes to provide persistent storage for Java objects.

External Procedures

Database functions, procedures, or member methods of an object type can be implemented in PL/SQL, Java, or C as external procedures. External procedures are best suited for tasks that are more quickly or easily done in a low-level language such as C, which is more efficient at machine-precision calculation. External procedures are always run in a safe mode outside the address space of the RDBMS server. "Generic" external procedures can be written that declare one or more parameters to be of a system-defined generic type. The generic type permits a procedure that uses it to work with data of any built-in or user-defined type.

Object Type Translator

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

Client-Side Cache

Oracle provides an object cache for efficient access to persistent objects stored in the database. Copies of objects can be brought into the object cache. Once the data has been cached in the client, the application can traverse through these at memory speed. Any changes made to objects in the cache can be committed to the database by using the object extensions to Oracle® Call Interface programmatic interfaces.

Oracle Call Interface Object Extensions

Oracle Call Interface provides a comprehensive application programming interface for application and tool developers seeking to use the object capabilities of Oracle. Oracle Call Interface provides a run-time environment with functions to connect to an Oracle server, and control transactions that access objects in the server. It allows application developers to access and manipulate objects and their attributes in the client-side object cache either *navigationally*, by traversing a graph of inter-connected objects, or *associatively* by specifying the nature of the data through declarative SQL DML. Oracle Call Interface also provides a number of functions for accessing metadata information at run-time about object types defined in the server. Such a set of functions facilitates dynamic access to the object metadata and the actual object data stored in the database.

Pro*C/C++ Object Extensions

The Oracle Pro*C™ precompiler provides an embedded SQL application programming interface and offers a higher level of abstraction than Oracle Call Interface. Like Oracle Call Interface, the Pro*C precompiler allows application developers to use the Oracle client-side object cache and the Object Type Translator Utility. Pro*C supports the use of "C" bind variables for Oracle9*i* object types. Furthermore, Pro*C provides new simplified syntax to allocate and free objects of SQL types and access them by either SQL DML, or through the navigational interface. Thus, it provides application developers many benefits, including compile-time type checking of (client-side) bind variables against the schema in the server, automatic mapping of object data in an Oracle9*i* server to program bind variables in the client, and simple ways to manage and manipulate database objects in the client process.

OO4O Object Extensions

Oracle9*i* Oracle Objects For OLE (OO4O) is a set of COM Automation interfaces / objects for connecting to Oracle9*i* database servers, executing queries and managing the results. Automation interfaces in OO4O provide easy and efficient access to Oracle9*i* 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.

2

Basic Components of Oracle Objects

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

This chapter contains these topics:

Object-Relational Elements

Object-relational functionality introduces a number of new concepts and resources. These are briefly described in the following sections.

Object Types

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

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

- A set of object types does not come ready-made with the database. Instead, you define the object types you want.
- Object types are not unitary: they have parts, called *attributes* and *methods*.

  Attributes hold the data about an object's features of interest. For example, a *soldier* object type might have the attributes `name`, `rank`, and `serial number`. An attribute has a declared datatype which can in turn be another object type. Taken together, the attributes of an object instance contain that object's data.

  Methods are procedures or functions provided to enable applications to perform useful operations on the attributes of the object type. Methods are an optional element of an object type. They define the behavior of objects of that type and determine what (if anything) that type of object can do.
- Object types are less generic than native datatypes. In fact, this is one of their major virtues: you can define object types to model the actual structure of the real-world entities—such as customers and purchase orders—that application programs deal with. This can make it easier and more intuitive to manage the data for these entities. In this respect object types are like Java and C++ classes.

You can think of an object type as a structural blueprint or template and an object as an actual thing built according to the template.

Object types are database schema objects, subject to the same kinds of administrative control as other schema objects (see Chapter 4, "Managing Oracle Objects").

You can use object types to model the actual structure of real-world objects. Object types enable you to capture the structural interrelationships of objects and their attributes instead of flattening this structure into a two-dimentional, purely relational schema of tables and columns. With object types you can store related pieces of data in a unit along with the behaviors defined for that data. Application code can then retrieve and manipulate these units as objects.

Type Inheritance

You can **specialize** an object type by creating **subtypes** that have some added, differentiating feature, such as an additional attribute or method. You create subtypes by deriving them from a parent object type, which is called a **supertype** of the derived subtypes.

Subtypes and supertypes are related by **inheritance**: as specialized versions of their parent, subtypes have all the parent's attributes and methods plus any specializations that are defined in the subtype itself. Subtypes and supertypes connected by inheritance make up a **type hierarchy**.

Objects

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

Methods

Methods are functions or procedures that you can declare in an object type definition to implement behavior that you want objects of that type to perform.

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

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

Object Tables

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

For example, the following statements create a `person` object type and define an object table for `person` objects:

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

CREATE TABLE person_table OF person;
```

You can view this table in two ways:

- As a single-column table in which each row is a`person` object, allowing you to perform object-oriented operations
- As a multi-column table in which each attribute of the object type `person`, namely `name` and `phone`, occupies a column, allowing you to perform relational operations

For example, you can execute the following instructions:

```
INSERT INTO person_table VALUES (
       "John Smith",
       "1-800-555-1212" );

SELECT VALUE(p) FROM person_table p
        WHERE p.name = "John Smith";
```

The first statement inserts a `person` object into `person_table`, treating `person_table` as a *multi-column* table. The second selects from `person_table` as a *single-column* table, using the `VALUE` function to return rows as object instances.

> **See Also:** "VALUE" on page 2-51 for information on the `VALUE` function

Row Objects and Column Objects

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

Object Views

An object view (see Chapter 5, "Applying an Object Model to Relational Data") is a way to access relational data using object-relational features. It lets you develop object-oriented applications without changing the underlying relational schema.

REF Datatype

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

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

Scoped REFs

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

The following example shows REF column `address_ref` scoped to an object table of `address_objtyp`.

```
CREATE TABLE people (
  id            NUMBER(4)
  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 ;
```

A REF can be scoped to an object table of the declared type (`address_objtyp` in the example) or of any subtype of the declared type. If scoped to an object table of a subtype, the REF column is effectively constrained to hold references only to instances of the subtype (and its subtypes, if any) in the table.

Subtypes are a feature of type inheritance.

See Also: "Type Inheritance" on page 2-33

Dangling REFs

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

Dereferencing REFs

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

Dereferencing a dangling REF returns a null object.

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

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

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

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

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

Obtaining REFs

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

```
DECLARE OrderRef REF to purchase_order;

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

The query must return exactly one row.

See Also: "Storage Size of REFs" on page 8-10

Collections

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

See Also: "Collections" on page 2-21.

Defining Object and Collection Types

You use the CREATE TYPE statement to define object types and collection types.

The following CREATE TYPE statements define the object types person, lineitem, lineitem_table, and purchase_order. lineitem_table is a collection type—a nested table type. The purchase_order object type has an attribute lineitems of this type. Each row in this nested table is an object of type lineitem.

The indented elements name, phone, item_name, and so on in the CREATE TYPE statements are attributes. Each has a datatype declared for it.

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

CREATE TYPE lineitem AS OBJECT (
  item_name   VARCHAR2(30),
  quantity    NUMBER,
  unit_price  NUMBER(12,2) );

CREATE TYPE lineitem_table AS TABLE OF lineitem;

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

  MEMBER FUNCTION
  get_value   RETURN NUMBER );
```

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

Defining an object type does not allocate any storage.

Once they are defined as types, `lineitem`, `person`, and `purchase_order` can be used in SQL statements in most of the same places you can use types like `NUMBER` or `VARCHAR2`.

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

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

The `CONTACTS` table is a relational table with an object type as the datatype of one of its columns. Objects that occupy columns of relational tables are called *column objects* (see "Row Objects and Column Objects" on page 2-4).

Object Types and References

This section describes object types and references, including:

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

Null Objects and Attributes

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

An object whose value is `NULL` is called *atomically null*. An atomically null object is different from one that simply happens to have null values for all its attributes. When all the attributes of an object are null, these attributes can still be changed, and the object's methods can be called. With an atomically null object, you can do neither of these things.

For example, consider the `CONTACTS` table defined as follows:

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

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

The statement

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

gives a different result from

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

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

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

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

Default Values for Objects and Collections

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

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

For example, consider the following statements:

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

CREATE TYPE people AS TABLE OF person;
```

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

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

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

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

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

Constraints for Object Tables

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

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

The following examples illustrate the possibilities.

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

```
CREATE TYPE location (
  building_no NUMBER,
  city        VARCHAR2(40) );

CREATE TYPE person (
  ssno        NUMBER,
  name        VARCHAR2(100),
  address     VARCHAR2(100),
  office      location );

CREATE TABLE person_extent OF person (
  ssno        PRIMARY KEY );
```

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

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

Indexes for Object Tables and Nested Tables

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

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

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

```
CREATE TABLE department (
  deptno      CHAR(5) PRIMARY KEY,
  dept_name   CHAR(20),
  dept_addr   address );

CREATE INDEX  i_dept_addr1
          ON  department (dept_addr.city);
```

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

Triggers for Object Tables

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

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

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

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

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

Rules for REF Columns and Attributes

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

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

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

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

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

Name Resolution

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

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

Oracle determines which table each column belongs to.

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

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

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

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

When Table Aliases are Required

Using unqualified names can lead to problems. If you add an ASSIGNMENT column to DEPTS and forget to change the query, Oracle automatically recompiles the query such that the inner SELECT uses the ASSIGNMENT column from the DEPTS table. This situation is called *inner capture*.

To avoid inner capture and similar problems resolving references, Oracle requires you to use a table alias to qualify any dot-notational reference to methods 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 an object type PERSON and two tables. ptab1 is an object table for objects of type PERSON, and ptab2 is a relational table that contains a column of an object type.

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

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

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

- In the first SELECT statement, ssno is the name of a column of ptab1. It references this top-level attribute directly, without using the dot notation, so no table alias is required.
- In the second SELECT statement, ssno is the name of an attribute of the PERSON object in the column named c1. This reference uses the dot notation and so requires a table alias, as shown in the fourth SELECT statement.
- The third SELECT uses the table name itself to qualify this the reference. This is incorrect; a table alias is required.

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

For example, the following expression tries to refer to the scott schema, projects table, assignment column, and duedate attribute of that column. But the expression is incorrect because projects is a table name, not an alias.

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

The same requirement applies to attribute references that use REFs.

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

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

Restriction on Using User-Defined Types with a Remote Database

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

- Connect to a remote database to query, insert, or update a user-defined type or an object REF on a remote table
- Use database links within PL/SQL code to declare a local variable of a remote user-defined type
- Convey a user-defined type argument or return value in a PL/SQL remote procedure call.

Methods

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

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

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

The parentheses are required. Unlike with PL/SQL functions and procedures, Oracle requires parentheses with all method calls, even ones that do not have arguments.

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

Two general kinds of methods can be declared in a type definition:

- Member
- Static

There is also a third kind of method, called a **constructor method**, that the system defines for every object type. You call a type's constructor method to construct or create an object instance of the type.

Member Methods

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

Member methods have a built-in parameter named `SELF` that denotes the object instance on which the method is currently being invoked. Member methods can reference the attributes and methods of `SELF` without a qualifier. This makes it simpler to write member methods. For example, the following code shows a method declaration that takes advantage of `SELF` to omit qualification of the attributes `num` and `den`:

```
CREATE TYPE Rational AS OBJECT (
  num INTEGER,
  den INTEGER,
  MEMBER PROCEDURE normalize,
  ...
);

CREATE TYPE BODY Rational AS
  MEMBER PROCEDURE normalize IS
    g INTEGER;
  BEGIN
    g := gcd(SELF.num, SELF.den);
    g := gcd(num, den);           -- equivalent to previous line
    num := num / g;
    den := den / g;
  END normalize;
  ...
END;
```

`SELF` does not need to be explicitly declared, although it can be. It is always the first parameter passed to the method. In member *functions*, if `SELF` is not declared, its parameter mode defaults to `IN`. In member *procedures*, if `SELF` is not declared, its parameter mode defaults to `IN OUT`.

You invoke a member method using the "dot" notation `object_variable.method()`. The notation specifies first the object on which to invoke the method and then the method to call. Any parameters occur inside the parentheses, which are required.

Methods for Comparing Objects

The values of a scalar datatype such as CHAR or REAL have a predefined order, which allows them to be compared. But an object type, such as a customer_typ, which can have multiple attributes of various datatypes, has no predefined axis of comparison. To be able to compare and order variables of an object type, you must specify a basis for comparing them.

Two special kinds of member methods can be defined for doing this: **map methods** and **order methods**.

Map Methods

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

From the standpoint of writing one, a map method is simply a parameterless member function that uses the MAP keyword and returns one of the datatypes just listed. What makes a map method special is that, if an object type defines one, the method is called automatically to evaluate such comparisons as obj_1 > obj_2 and comparisons implied by the DISTINCT, GROUP BY, and ORDER BY clauses. Where obj_1 and obj_2 are two object variables that can be compared using a map method map(), the comparison:

```
obj_1 > obj_2
```

is equivalent to:

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

And similarly for other relational operators besides ">".

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

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

```
CREATE TYPE BODY Rectangle_typ AS
  MAP MEMBER FUNCTION area RETURN NUMBER IS
  BEGIN
     RETURN len * wid;
  END area;
  ...
END;
```

An object type can declare at most one map method (or one order method). A subtype can declare a map method only if its root supertype declares one.

Order Methods

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

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

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

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

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

The following example shows an order method that compares customers by customer ID:

```
CREATE TYPE Customer_typ AS OBJECT (
  id   NUMBER,
  name VARCHAR2(20),
  addr VARCHAR2(30),
  ORDER MEMBER FUNCTION match (c Customer_typ) RETURN INTEGER
);
```

```
CREATE TYPE BODY Customer_typ AS
  ORDER MEMBER FUNCTION match (c Customer_typ) RETURN INTEGER IS
  BEGIN
    IF id < c.id THEN
      RETURN -1;               -- any negative number will do
    ELSIF id > c.id THEN
      RETURN 1;                -- any positive number will do
    ELSE
      RETURN 0;
    END IF;
  END;
END;
```

Guidelines

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

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

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

Comparison Methods in Type Hierarchies

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

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

So if the root type does not specify either a map or an order method, none of the subtypes can specify either a map or order method.

See Also: "Type Inheritance" on page 2-33

Static Methods

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

You invoke a static method by using the "dot" notation to qualify the method call with the name of the object type: `type_name.method()`.

Constructor Methods

Every object type has a **constructor method** implicitly defined for it by the system. A constructor method is a function that returns a new instance of the user-defined type and sets up the values of its attributes. You can also explicitly define your own constructors. The present section describes constructor methods in general and system-defined constructors in particular.

See Also: "User-Defined Constructors" on page 6-21 for information on user-defined constructors and their advantages

A constructor method is a function; it returns the new object as its value. The name of the constructor method is just the name of the object type. Its parameters have the names and types of the object type's attributes.

For example, suppose we have a type `Customer_typ`:

```
CREATE TYPE Customer_typ AS OBJECT (
  id    NUMBER,
  name  VARCHAR2(20),
  phone VARCHAR2(30),
);
```

The following example creates a new object instance of `Customer_typ`, specifies values for its attributes, and sets the object into a variable:

```
cust = Customer_typ(103, "Ravi", "1-800-555-1212")
```

The `INSERT` statement in the next example inserts a customer object that has an attribute of `Address_typ` object type. The constructor method `Address_typ` constructs an object of this type having the attribute values shown in the parentheses:

```
INSERT INTO Customer_objtab
  VALUES (
    1, 'Jean Nance',
    Address_typ('2 Avocet Drive', 'Redwood Shores', 'CA', '95054'),
    ...
    ) ;
```

Collections

Oracle supports two *collection* datatypes: varrays and nested tables.

- A varray is an *ordered* collection of elements: the position of each element has an index number, and you use this number to access particular elements. When you define a varray, you specify the maximum number of elements it can contain, although you can change this number later. Varrays are stored as opaque objects (that is, `RAW` or `BLOB`).
- A nested table can have any number of elements: no maximum is specified in the definition of the table; also, the order of the elements is not preserved. You select, insert, delete, and so on, in a nested table just as you do with ordinary tables. Elements of a nested table are actually stored in a separate storage table that contains a column that identifies the parent table row or object to which each element belongs.

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

If you need to run efficient queries on a collection, handle arbitrary numbers of elements, or do mass insert/update/delete operations, then use a nested table.

Varrays

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

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

For example, the following statement declares an array type:

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

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

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

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

A varray is normally stored in line, that is, in the same tablespace as the other data in its row. If it is sufficiently large, Oracle stores it as a `BLOB`.

A varray cannot contain LOBs. This means that a varray also cannot contain elements of a user-defined type that has a LOB attribute.

See Also: "Storage Considerations for Varrays" on page 8-15.

Nested Tables

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

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

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

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

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

When a column in a relational table is of nested table type, Oracle stores the nested table data for all rows of the relational table in the same storage table. Similarly, with an object table of a type that has a nested table attribute, Oracle stores nested table data for all object instances in a single storage table associated with the object table.

For example, the following statement defines an object table for the object type PURCHASE_ORDER:

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

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

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

> **See Also:** See *Oracle9i SQL Reference* for information about nested cursors, and see"Nested Tables" on page 8-16 for more information on using nested tables.

Multilevel Collection Types

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

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

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

The following example creates a multilevel collection type that is a nested table of nested tables. The example models a system of stars in which each star has a nested table collection of the planets revolving around it, and each planet has a nested table collection of its satellites.

```
CREATE TYPE satellite_t AS OBJECT (
  name        VARCHAR2(20),
  diameter    NUMBER);

CREATE TYPE nt_sat_t AS TABLE OF satellite_t;
CREATE TYPE planet_t AS OBJECT (
  name        VARCHAR2(20),
  mass        NUMBER,
  satellites  nt_sat_t);

CREATE TYPE nt_pl_t AS TABLE OF planet_t;
```

Nested Table Storage Tables

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

For example, the following code creates a table `stars` that contains a column `planets` whose type is a multilevel collection (a nested table of an object type that has a nested table attribute `satellites`). Separate nested table clauses are provided for the outer `planets` nested table and for the inner `satellites` one.

```
CREATE TABLE stars (
  name     VARCHAR2(20),
  age      NUMBER,
  planets  nt_pl_t)
NESTED TABLE planets STORE AS planets_tab
  (NESTED TABLE satellites STORE AS satellites_tab);
```

The preceding example can refer to the inner `satellite` nested table by name because this nested table is a named attribute of an object. However, if the inner nested table is not an attribute, it has no name. The keyword `COLUMN_VALUE` is provided for this case: you use it in place of a name for an inner nested table. For example:

```
CREATE TYPE inner_table AS TABLE OF NUMBER;

CREATE TYPE outer_table AS TABLE OF inner_table;

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

Physical attributes for the storage tables can be specified in the nested table clause. For example:

```
CREATE TABLE stars (
  name     VARCHAR2(20),
  age      NUMBER,
  planets  nt_pl_t)
NESTED TABLE planets STORE AS planets_tab
  ( PRIMARY KEY (NESTED_TABLE_ID, name)
    ORGANIZATION INDEX COMPRESS
    NESTED TABLE satellites STORE AS satellites_tab );
```

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

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

> **See Also:** "Nested Table Storage" on page 8-16 and "Object Tables with Embedded Objects" on page 9-23

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.

Varray Storage

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

- In a varray of varrays, the entire varray is stored inline (that is, in the row itself) unless it is too large (about 4000 bytes) or LOB storage is explicitly specified.
- In a varray of nested tables, the entire varray is stored in a LOB, with only the LOB locator stored in the row. There is no storage table associated with nested

table elements of a varray. The entire nested table collection is stored inside the varray.

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

```
CREATE TYPE va1 AS VARRAY(10) OF NUMBER;

CREATE TYPE nt3 AS TABLE OF va1;

CREATE TABLE tab2 (c1 NUMBER, c2 nt3)
NESTED TABLE c2 STORE AS c2_tab2_nt
  ( VARRAY column_value STORE AS LOB tab2_lob );
```

The following example shows explicit LOB storage specified for a varray of varray type:

```
CREATE TYPE t2 AS OBJECT (a NUMBER, b va1);

CREATE TYPE va2 AS VARRAY(2) OF t2;

CREATE TABLE tab5 (c1 NUMBER, c2 va2)
VARRAY c2 STORE AS tab5_lob;
```

Assignment and Comparison of Multilevel Collections

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

Items whose data types are collection types, including multilevel collection types, cannot be compared.

Creating a VARRAY or Nested Table

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

Calling a constructor method with an empty list creates an empty collection of that type. Note that an *empty* collection is an actual collection that happens to be empty; it is not the same as a *null* collection.

Constructors for Multilevel Collections

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

The following example calls the constructor for the multilevel collection type `nt_pl_t`. This type is a nested table of planets, each of which contains a nested table of satellites as an attribute. The constructor for the outer nested table calls the `planet_t` constructor for each planet to be created; each planet constructor calls the constructor for the satellites nested table type to create its nested table of satellites; and the satellites nested table type constructor calls the `satellite_t` constructor for each satellite instance to be created.

```
INSERT INTO stars
VALUES('Sun',23,
  nt_pl_t(
    planet_t(
      'Neptune',
      10,
      nt_sat_t(
        satellite_t('Proteus',67),
        satellite_t('Triton',82)
      )
    ),
    planet_t(
      'Jupiter',
      189,
      nt_sat_t(
        satellite_t('Callisto',97),
        satellite_t('Ganymede', 22)
      )
    )
  )
);
```

Querying Collections

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

Nesting Results of Collection Queries

In the following query, column `projects` is a nested table collection of `projects_list_nt` type. The `projects` collection column appears in the `SELECT` list like an ordinary, scalar column. Querying a collection column in the `SELECT` list like this *nests* the elements of the collection in the result row with which the collection is associated.

For example, the following query gets the name of each employee and the collection of projects for that employee. The collection of projects is nested:

```
SELECT e.empname, e.projects
  FROM employees e;

  EMPNAME   PROJECTS
  -------   --------
  'Bob'     PROJECTS_LIST_NT(14, 23, 144)
  'Daphne'  PROJECTS_LIST_NT(14, 35)
```

If project values or instances are a user-defined type—for example, `Proj_t`, with two attributes, `id` and `name`—a result row looks something like this:

```
EMPNAME   PROJECTS
-------   --------
'Bob'     PROJECTS_LIST_NT(PROJ_T(14, 'White Horse'), PROJ_T(23, 'Excalibur'), ...)
```

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 employees` would produce a nested result.

Unnesting Results of Collection Queries

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

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

Note: The TABLE expression takes the place of the THE *subquery* expression. THE *subquery* will eventually be deprecated.

Like the preceding example, the following query gets the name of each employee and the collection of projects for that employee, but the collection is unnested:

```
SELECT e.empname, p.*
  FROM employees e, TABLE(e.projects) p;

  EMPNAME   PROJECTS
  -------   --------
  'Bob'     14
  'Bob'     23
  'Bob'     144
  'Daphne'  14
  'Daphne'  35
```

As the preceding example shows, a TABLE expression can have its own table alias. In the example, a table alias for the TABLE expression appears in the SELECT list to select columns returned by the TABLE expression.

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

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

```
SELECT *
  FROM employees e, TABLE(e.projects);

PROJECTS
--------
14
23
144
14
35
```

Or:

```
SELECT p.*
  FROM employees e, TABLE(e.projects) p
  WHERE e.empid = 100;

PROJECTS
--------
14
23
144
```

The following example produces rows only for employees who have projects.

```
SELECT e.empname, p.*
  FROM employees e, TABLE(e.projects) p;
```

To get rows for employees with no projects, you can use outer-join syntax:

```
SELECT e.*, p.*
  FROM employees e, TABLE(e.projects)(+) p;
```

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

Unnesting Queries Containing Table Expression Subqueries

The preceding examples show a `TABLE` expression that contains the name of a collection. Alternatively, a `TABLE` expression can contain a subquery of a collection.

The following example returns the collection of projects for the employee whose id is `100`.

```
SELECT *
  FROM TABLE(SELECT e.projects
               FROM employees e
               WHERE e.empid = 100);

PROJECTS
--------
14
23
144
```

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

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

Here is an example showing a TABLE expression used in the FROM clause of a SELECT embedded in a CURSOR expression:

```
SELECT e.empid, CURSOR(SELECT * FROM TABLE(e.projects))
  FROM employees e;
```

Unnesting Queries with Multilevel Collections

Unnesting queries can be used with multilevel collections, too, for both varrays and nested tables. The following example shows an unnesting query on a multilevel nested table collection of nested tables. From a table stars in which each star has a nested table of planets and each planet has a nested table of satellites, the query returns the names of all satellites from the inner set of nested tables.

```
SELECT t.name
  FROM stars s, TABLE(s.planets) p, TABLE(p.satellites) t;
```

See Also: "Viewing Object Data in Relational Form with Unnesting Queries" on page 8-12

Performing DML Operations on Collections

Oracle supports the following DML operations on nested table columns:

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

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

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

The following DML statements demonstrate piecewise operations on nested table columns.

```
INSERT INTO TABLE(SELECT e.projects
                  FROM      employees e
                  WHERE     e.eno = 100)
   VALUES (1, 'Project Neptune');

UPDATE TABLE(SELECT e.projects
                 FROM       employees e
                 WHERE      e.eno = 100) p
   SET VALUE(p) = project_typ(1, 'Project Pluto')
   WHERE p.pno = 1;

DELETE FROM TABLE(SELECT e.projects
                  FROM       employee e
                  WHERE      e.eno = 100) p
   WHERE p.pno = 1;
```

Performing DML on Multilevel Collections

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

Collections as Atomic Data Items The section "Constructors for Multilevel Collections" shows an example of inserting an entire multilevel collection with an INSERT statement. Multilevel collections can also be updated atomically with an UPDATE statement. For example, suppose v_planets is a variable declared to be of the planets nested table type nt_pl_t. The following statement updates stars by setting the planets collection as a unit to the value of v_planets.

```
UPDATE stars  s
SET s.planets = :v_planets
WHERE s.name = 'Aurora Borealis';
```

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

The following example shows a piecewise insert operation on the `planets` nested table of nested tables: the example inserts a new planet, complete with its own nested table of `satellite_t`:

```
INSERT INTO TABLE( SELECT planets FROM stars WHERE name = 'Sun')
VALUES ('Saturn', 56,
  nt_sat_t(
    satellite_t('Rhea', 83)
  )
);
```

The next example performs a piecewise insert into an inner nested table to add a satellite for a planet. Like the preceding, this example uses a `TABLE` expression containing a subquery that selects the inner nested table to specify the target for the insert.

```
INSERT INTO TABLE( SELECT p.satellites
  FROM TABLE( SELECT s.planets
    FROM stars s
    WHERE s.name = 'Sun') p
  WHERE p.name = 'Uranus')
VALUES ('Miranda', 31);
```

Type Inheritance

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

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

Subtypes in a hierarchy are connected to their supertypes by **inheritance**. This means that subtypes automatically acquire the attributes and methods of their parent type. It also means that subtypes automatically acquire any changes made to these attributes or methods in the parent: any attributes or methods updated in a supertype are updated in subtypes as well.

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

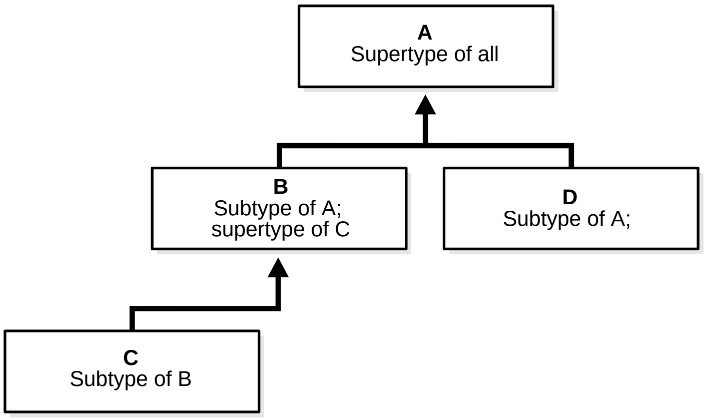
Polymorphism is the ability of a slot for a value in code to contain a value of *either* a certain declared type *or* any of a range of the declared type's subtypes. A method called on whatever value occupies the slot may execute differently depending on the value's type because the various types might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

Types and Subtypes

A subtype can be derived from a supertype either directly, or indirectly through intervening levels of other subtypes.

A subtype can directly derive only from a single supertype: it cannot derive jointly from more than one. A supertype can have multiple sibling subtypes, but a subtype can have at most one direct parent supertype. In other words, Oracle supports only single inheritance, not multiple inheritance.

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



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

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

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

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

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

  For example, a `shape` object type might define a method `calculate_area()`. Two subtypes of `shape`, `rectilinear_shape` and `circular_shape`, might each implement this method in a different way.

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

Remember, a child type is not a *different* type from its parent: it's a particular *kind* of that type. If the general definition of `customer` ever changes, the definition of `corp_customer` changes too.

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

FINAL and NOT FINAL Types and Methods

An object type's definition determines whether subtypes can be derived from that type. To permit subtypes, the object type must be defined as **not final**. This is done by including the `NOT FINAL` keyword in its type declaration. For example:

```
CREATE TYPE Person_typ AS OBJECT
( ssn NUMBER,
  name VARCHAR2(30),
  address VARCHAR2(100)) NOT FINAL;
```

The preceding statement declares `Person_typ` to be a not final type such that subtypes of `Person_typ` can be defined. By default, an object type is **final**—that is, subtypes cannot be derived from it.

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

```
ALTER TYPE Person_typ FINAL;
```

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

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

The following statement creates a not final type containing a final member function:

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

See Also: "Overriding Methods" on page 2-40

Creating Subtypes

You create a subtype using a `CREATE TYPE` statement that specifies the immediate parent of the subtype with an `UNDER` parameter:

```
CREATE TYPE Student_typ UNDER Person_typ
( deptid NUMBER,
   major VARCHAR2(30)) NOT FINAL;
```

The preceding statement creates `Student_typ` as a subtype of `Person_typ`. As a subtype of `Person_typ`, `Student_typ` inherits all the attributes declared in or inherited by `Person_typ` and any methods inherited by `Person_typ` or declared in `Person_typ`.

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

A type can have multiple child subtypes, and these can also have subtypes. The following statement creates another subtype `Employee_typ` under `Person_typ`.

```
CREATE TYPE Employee_typ UNDER Person_typ
( empid NUMBER,
  mgr VARCHAR2(30));
```

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. For example, the following statement defines a new subtype `PartTimeStudent_typ` under `Student_typ`. The new subtype inherits all the attributes and methods of `Student_typ` and adds another attribute.

```
CREATE TYPE PartTimeStudent_typ UNDER Student_typ
( numhours NUMBER);
```

NOT INSTANTIABLE Types and Methods

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

```
CREATE TYPE Address_typ AS OBJECT(...) NOT INSTANTIABLE NOT FINAL;
CREATE TYPE USAddress_typ UNDER Address_typ(...);
CREATE TYPE IntlAddress_typ UNDER Address_typ(...);
```

A method can also be declared to be not instantiable. Use this option when you want to declare a method in a type without implementing the method there. A type that contains a non-instantiable method must itself be declared not instantiable. For example:

```
CREATE TYPE T AS OBJECT (
  x NUMBER,
  NOT INSTANTIABLE MEMBER FUNCTION func1() RETURN NUMBER
) NOT INSTANTIABLE NOT FINAL;
```

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

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

A non-instantiable subtype can be defined under an instantiable supertype.

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

```
ALTER TYPE Example_typ INSTANTIABLE;
```

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

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

Inheriting, Overloading, and Overriding Methods

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

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

Giving a type multiple methods with the same name is called method **overloading**. Redefining an inherited method to customize its behavior for a subtype is called method **overriding**.

Overloading Methods

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

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

Subtype `MySubType_typ` in the following example creates an overload of `foo()`:

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

CREATE TYPE MySubType_typ UNDER MyType_typ (...,
  MEMBER PROCEDURE foo(x DATE),
  STATIC FUNCTION bar(...)...
  ...);
```

MySubType_typ contains two versions of foo( ): one inherited version, with a NUMBER parameter, and a new version with a DATE parameter.

Overriding Methods

Overriding redefines an inherited method to make it do something different in the subtype. For example, a subtype circular_shape derived from a shape supertype might override a method calculate_area() to customize it specifically for calculating the area of a circle.

When a subtype overrides a method, the new version is executed instead of the overridden one whenever an instance of the subtype invokes the method. If the subtype itself has subtypes, these inherit the override of the method instead of the original version.

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

In the type definition, precede a method declaration with the OVERRIDING keyword to signal that you are overriding the method. For example, in the following code, the subtype signals that it is overriding method Print():

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

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

As with new methods, you supply the declaration for an overridng method in a CREATE TYPE BODY statement.

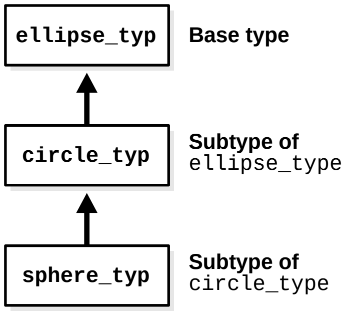
Restrictions on Overriding Methods

- You can override only methods that are not declared to be final in the supertype.

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

Dynamic Method Dispatch

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



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

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

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

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

Substituting Types in a Type Hierarchy

In a type hierarchy, the subtypes are variant *kinds* of the root, base type. For example, a `Student_typ` type and an `Employee_typ` are kinds of a `Person_typ`. The base type includes these other types.

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

The (polymorphic) ability to select all persons and get back not only objects whose declared type is `Person_typ` but also objects whose declared (sub)type 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 slot (a variable, column, and so forth) whose declared type is the supertype.

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

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

In principle, object attributes, collection elements and REFs are always substitutable: there is no syntax at the level of the type definition to constrain their substitutability to some subtype. You can, however, turn off or constrain substitutability at the storage level, for specific tables and columns.

See Also: "Turning Off Substitutability" and "Constraining Substitutability" on page 2-47

Attribute Substitutability

Object attributes, collection elements and REFs are substitutable. Where MyType is an object type:

- REF type attributes: An attribute defined as REF MyType can hold a REF to an instance of MyType or to an instance of any subtype of MyType.
- Object type attributes: An attribute defined to be of type MyType can hold an instance of MyType or of any subtype of MyType.
- Collection type elements: A collection of elements of type MyType can hold instances of MyType and instances of any subtype of MyType.

For instance, the author attribute is substitutable in the Book_typ defined in the following example:

```
CREATE TYPE Book_typ AS OBJECT
( title VARCHAR2(30),
  author Person_typ     /* substitutable */);
```

An instance of Book_typ can be created by specifying a title string and an author of Person_typ or of any subtype of Person_typ. The following example specifies an author of type Employee_typ:

```
Book_typ('My Oracle Experience',
       Employee_typ(12345, 'Joe', 'SF', 1111, NULL))
```

Attributes in general can be accessed using the dot notation. Attributes of a *subtype* of a row or column's declared type can be accessed with the TREAT function. For example, in an object view Books_v of Book_typ, you can use TREAT to get the employee id of authors of Employee_typ. (The author column is of Person_typ.)

```
SELECT TREAT(author AS Employee_typ).empid FROM Books_v;
```

See Also: "TREAT" on page 2-52

Column and Row Substitutability

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

For example, here again is the Person_typ type hierarchy introduced earlier:

```
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 PartTimeStudent_typ UNDER Student_typ
( numhours NUMBER);
```

An object table of `Person_typ` can contain rows of all three types. You insert an instance of a given type using the constructor for that type in the `VALUES` clause of the `INSERT` statement:

```
CREATE TABLE persons OF Person_typ;

INSERT INTO persons
  VALUES (Person_typ(1243, 'Bob', '121 Front St'));

INSERT INTO persons
  VALUES (Student_typ(3456, 'Joe', '34 View', 12, 'HISTORY'));

INSERT INTO persons
  VALUES (PartTimeStudent_typ(5678, 'Tim', 13, 'PHYSICS', 20));
```

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

```
CREATE TABLE books (title varchar2(100), author Person_typ);

INSERT INTO books
 VALUES('An Autobiography', Person_typ(1243, 'Bob'));

INSERT INTO books
 VALUES('Business Rules', Student_typ(3456, 'Joe', 12, 'HISTORY'));

INSERT INTO books
 VALUES('Mixing School and Work',
        PartTimeStudent_typ(5678, 'Tim', 13, 'PHYSICS', 20));
```

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.

Subtypes Having Supertype Attributes

A subtype can have an attribute that is a supertype. For example:

```
CREATE TYPE Student_typ UNDER Person_typ (..., advisor Person_typ);
```

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

You can, however, define substitutable columns of subtypes that have REF attributes that reference supertypes.

> **See Also:** "Turning Off Substitutability" on page 2-47

REF Columns and Attributes

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

Collection Elements

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

Creating Subtypes After Creating Substitutable Columns

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

The following example shows an attempt to create a subtype Student_typ. The attempt fails because Student_typ has a supertype attribute, and table persons has a substitutable column p of the supertype.

```
CREATE TYPE Person_typ AS OBJECT
( ssn NUMBER,
  name VARCHAR2(30),
  address VARCHAR2(100)) NOT FINAL;

CREATE TYPE Employee_typ UNDER Person_typ
( salary NUMBER) NOT FINAL;

CREATE TABLE persons (p person_typ);
-- Table persons can store Person_typ and Employee_typ

INSERT INTO persons
  VALUES (Person_typ(1243, 'Bob', '121 Front St'));

-- This statement fails because there exists a substitutable
-- column of the supertype.
CREATE TYPE Student_typ UNDER Person_typ
( advisor Person_typ);
```

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

```
CREATE TYPE Student_typ UNDER Person_typ
( deptid NUMBER,
  major VARCHAR2(30)) NOT FINAL;

-- Inserts an instance of the subtype in table persons
INSERT INTO persons
  VALUES (Student_typ(3456, 'Joe', '34 View', 12, 'HISTORY'));
```

Dropping Subtypes After Creating Substitutable Columns

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

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

```
-- This statement fails:
DROP TYPE Student_typ VALIDATE;
```

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

```
DELETE FROM persons WHERE p IS OF (Student_typ);

-- Now the DROP statement succeeds
DROP TYPE Student_typ VALIDATE;
```

Turning Off Substitutability

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

In the following example, the clause confines column `book` of a relational table to storing only `Person_typ` instances as authors and disallows any subtype instances:

```
CREATE TABLE catalog (book Book_typ, price NUMBER)
  COLUMN book NOT SUBSTITUTABLE AT ALL LEVELS;
```

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

```
CREATE TABLE Student_books OF Book_typ NOT SUBSTITUTABLE AT ALL LEVELS;
```

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

```
CREATE TABLE departments(name VARCHAR2(10), emps emp_set)
  NESTED TABLE (emps)
  NOT SUBSTITUTABLE AT ALL LEVELS STORE AS ...
```

Some things to note about turning off substitutability:

- There is no mechanism to turn off substitutability for `REF` columns.
- A column must be a top-level column for the clause `NOT SUBSTITUTABLE AT ALL LEVELS` to be applied to it: the clause cannot be applied to an object-type attribute.

Constraining Substitutability

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

For example, the following statement creates a table of `Book_typ` in which authors are constrained to just those persons who are students:

```
CREATE TABLE Student_books OF Book_typ
  COLUMN author IS OF (ONLY Student_typ);
```

Although the type `Book_typ` allows authors to be of type `Person_typ`, the column declaration imposes a constraint to store only instances of `Student_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.

Assignments Across Types

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

Objects and REFs to Objects

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

An assignment of a source of type `Source_typ` to a target of type `Target_typ` must be of one of the following two patterns:

- Case 1: `Source_typ` and `Target_typ` are the same type
- Case 2: `Source_typ` is a subtype of `Target_typ` ("widening")

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

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

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

```
TABLE T(perscol Person_typ, empcol Employee_typ, stucol Student_typ)
```

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

```
UPDATE T set perscol = empcol;
```

PL/SQL:

```
declare
  var1 Person_typ;
  var2 Employee_typ;
begin
  var1 := var2;
end;
```

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

To do a narrowing assignment, you must use the `TREAT` function to explicitly change the declared type of the source value to the more specialized target type, or one of its subtypes, in the hierarchy. The `TREAT` function checks at runtime to verify that the change can be made; then `TREAT` either makes the change or returns `NULL` if the source value—the person in question—is not of the target type or one of its subtypes.

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

```
UPDATE T set empcol = TREAT(perscol 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 empcol = perscol;
```

See Also: "TREAT" on page 2-52

Collection Assignments

In assignments of expressions of a collection type, the source and target must be of the same declared type. Neither widening nor narrowing is permitted. However, a subtype value can be assigned to a supertype collection.

For example, suppose we have the following collection types:

```
CREATE TYPE PersonSet AS TABLE OF Person_typ;
CREATE TYPE StudentSet AS TABLE OF Student_typ;
```

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

```
declare
  var1 PersonSet; var2 StudentSet;
  elem1 Person_typ; elem2 Student_typ;
begin
  var1 := var2;                 /* ILLEGAL - collections not of same type */
  var1 := PersonSet (elem1, elem2);      /* LEGAL : Element is of subtype */
```

Comparisons: Objects, REF Variables, and Collections

Comparing Object Instances

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

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

If a type does not define either a map method or an order method, object variables of that type can be compared only in SQL statements and only for equality or inequality. (Two objects of the same type count as equal only if the values of their corresponding attributes are equal.)

See Also: "Methods for Comparing Objects" on page 2-17

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.

Comparing Collections.

There is no mechanism for comparing collections.

Functions and Predicates Useful with Objects

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

- `VALUE`
- `REF`
- `DEREF`
- `TREAT`
- `IS OF TYPE`
- `SYS_TYPEID`

Examples are given throughout this book.

In PL/SQL the VALUE, REF and DEREF functions can appear only in a SQL statement.

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. For example, the following statement selects all persons whose name is John Smith:

```
SELECT VALUE(p) FROM person_table p
  WHERE p.name = "John Smith";
```

The VALUE function may return instances of the declared type of the row or any of its subtypes. For example, the following query returns all persons, including students and employees, from an object view Person_v of persons:

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

To retrieve *only* persons—that is, instances whose most specific type is person, use the ONLY keyword to confine the selection to the declared type of the view or subview that you are querying:

```
SELECT VALUE(p) FROM ONLY(Person_v) p;
```

The following example shows VALUE used to return object instance rows for updating:

```
UPDATE TABLE(SELECT e.projects
                FROM        employees e
                WHERE       e.eno = 100) p
   SET VALUE(p) = project_typ(1, 'Project Pluto')
   WHERE p.pno = 1;
```

REF

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

```
SELECT REF(p) FROM Person_v p;
```

The following example returns a REF to the person (or student or employee) whose id attribute is 0001:

```
SELECT REF(p)
  FROM Person_v p
  WHERE p.id = 0001 ;
```

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 objects from the object view Person_v, including persons who are students and persons who are employees.

```
SELECT DEREF(REF(p)) FROM Person_v p;
```

TREAT

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

The two main uses of TREAT are:

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

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

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

In the next example, TREAT returns all (and only) Student_typ instances from object view Person_v 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.

```
SELECT TREAT(VALUE(p) AS Student_typ)
FROM Person_v 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.

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_v p;
```

The example returns REFs to all Student_typ instances. It returns NULL REFs for all person instances that are not students.

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

```
SELECT name, TREAT(VALUE(p) AS Student_typ).major major
```

```
  FROM persons p;

NAME    MAJOR
----    ------
Bob     null
Joe     HISTORY
Tim     PHYSICS
```

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 persons p;
```

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

The following example shows `TREAT` used to access a subtype method:

```
SELECT name, TREAT(VALUE(p) AS Student_typ).major() major
  FROM persons p;
```

> **See Also:** "Assignments Across Types" on page 2-48 for information on using `TREAT` in assignments.

Currently, `TREAT` is supported only for SQL; it is not supported for PL/SQL.

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 persons table.

```
SELECT VALUE(p) FROM persons p
WHERE VALUE(p) IS OF (Student_typ);

VALUE(p)
--------
Student_typ('Joe', 3456, 12, 10000)
PartTimeStudent_typ('Tim', 5678, 13, 1000, 20)
```

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 type(s).

For example, the following query retrieves only books authored by students. It excludes books authored by any student subtype (such as PartTimeStudent_typ).

```
SELECT b.title title, b.author author FROM books b
WHERE b.author IS OF (ONLY Student_typ);

TITLE             AUTHOR
-----             ------
Business Rules    Student_typ('Joe', 3456, 12, 10000)
```

In the next example, the statement tests objects in object view Person_v, which contains persons, employees, and students, and returns REFs just to objects of the two specified person subtypes Employee_typ and Student_typ (and their subtypes, if any):

```
SELECT REF(p) FROM Person_v p
WHERE VALUE(p) IS OF (Employee_typ, Student_typ);
```

The following statement returns only students whose most specific or specialized type is Student_typ. If the view contains any objects of a subtype of Student_typ—for example, PartTimeStudent_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 (namely, Person_typ):

```
SELECT TREAT(VALUE(p) AS Student_t)
FROM Person_v p
WHERE VALUE(p) IS OF(ONLY Student_t);
```

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 PersRefCol is declared to be REF Person_typ, you can get just the rows for students as follows:

```
SELECT * FROM view
WHERE DEREF(PersRefCol) IS OF (Student_typ);
```

IS OF is currently supported only for SQL, not for PL/SQL.

SYS_TYPEID

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

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

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

The syntax of the function is:

```
SYS_TYPEID( object_type_value )
```

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

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

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

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

> **See Also:** "Hidden Columns for Substitutable Columns and Tables" on page 6-3 for more information about type discriminant columns

For an example of `SYS_TYPEID`, consider the substitutable object table `persons`, of `Person_typ`. `Person_typ` is the root type of a hierarchy that has `Student_typ` as a subtype and `PartTimeStudent_typ` as a subtype of `Student_typ`:

```
CREATE TABLE persons OF Person_typ;

INSERT INTO persons
  VALUES (Person_typ(1243, 'Bob', '121 Front St'));
```

```
INSERT INTO persons
  VALUES (Student_typ(3456, 'Joe', '34 View', 12, 'HISTORY'));

INSERT INTO persons
  VALUES (PartTimeStudent_typ(5678, 'Tim', 13, 'PHYSICS', 20));
```

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

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

NAME   TYPEID
----   ------
Bob    01
Joe    02
Tim    03
```

The following query returns the most specific types of authors stored in the books table. `author` is a substitutable column of `Person_typ`:

```
SELECT b.title, b.author.name, SYS_TYPEID(author) typeid FROM books b;

TITLE                      AUTHOR     TYPEID
----                       ------     ------
An Autobiography           Bob        01
Business Rules             Joe        02
Mixing School and Work     Tim        03
```

> **See Also:** "Hidden Columns for Substitutable Columns and Tables" in Chapter 6 for information about the type discriminant and other hidden columns

3

Object Support in Oracle Programming Environments

In Oracle9*i*, you can create object types with SQL data definition language (DDL) commands, and you can manipulate objects with SQL data manipulation language (DML) commands. Object support is built into Oracle's application programming environments:

- SQL
- PL/SQL
- Oracle Call Interface (OCI)
- Oracle C++ Call Interface (OCCI)
- Pro*C/C++
- Oracle Type Translator (OTT)
- Oracle Objects For OLE (OO4O)
- Java: JDBC, Oracle SQLJ, JPublisher, and SQLJ Object Types

SQL

Oracle SQL DDL provides the following support for object types:

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

Oracle SQL DML provides the following support for object types:

- Querying and updating objects and collections
- Manipulating REFs

> **See Also:** For a complete description of Oracle SQL syntax, see *Oracle9i SQL Reference*.

PL/SQL

Object types and subtypes can be used in PL/SQL procedures and functions in most places where built-in types can appear.

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

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

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

Oracle Call Interface (OCI)

LNOCI 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.

LNOCI provides functions to

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

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

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

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

Associative Access in OCI Programs

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

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

- Synchronize object and relational functionality through enhanced commit and rollback functions

See Also: "Associative Access in Pro*C/C++" on page 3-8

Navigational Access in OCI Programs

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

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

See Also: "Navigational Access in Pro*C/C++" on page 3-8

Object Cache

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

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

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

Building an OCI Program that Manipulates Objects

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

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

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

See Also: "LNOCI Tips and Techniques for Objects" on page 6-27

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.

For example:

```
CREATE OR REPLACE TYPE person AS OBJECT
(
    name VARCHAR2(30),
    CONSTRUCTOR FUNCTION person(name VARCHAR2) RETURN SELF AS RESULT
);

CREATE OR REPLACE TYPE BODY person IS
    CONSTRUCTOR FUNCTION 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.

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.

LNOCI 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 datatypes. 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 dataype 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

Pro*C/C++

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

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

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

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

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

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

Associative Access in Pro*C/C++

For background information on associative access, see "Associative Access in OCI Programs" on page 3-3.

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

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

Navigational Access in Pro*C/C++

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

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

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

Converting Between Oracle Types and C Types

The C representation for objects that is generated by the Oracle Type Translator (OTT) uses OCI types whose internal details are hidden, such as `LNOCIString` and `LNOCINumber` for scalar attributes. Collection types and object references are similarly represented using `LNOCITable`, `LNOCIArray`, and `LNOCIRef` 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++ Precompiler Programmer's Guide*.

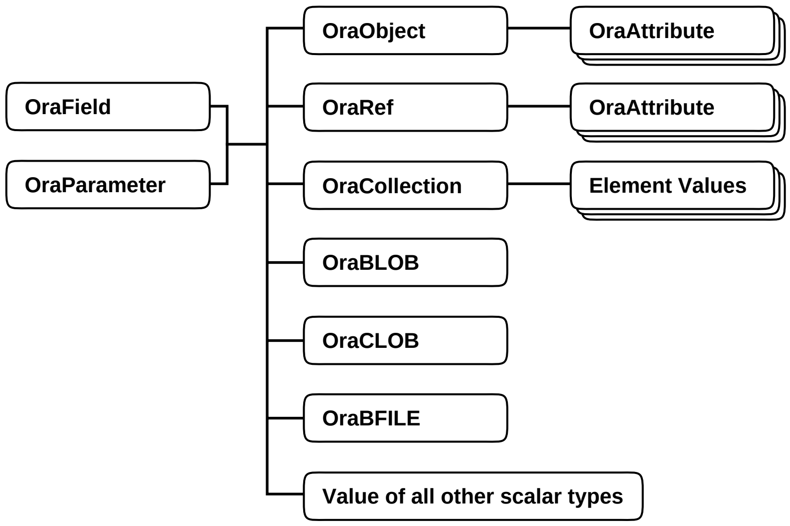
Oracle Objects For OLE (OO4O)

Oracle Objects for OLE (OO4O)—for Visual Basic, Excel, ActiveX, and Active Server Pages—provides full support for accessing and manipulating instances of REFs, value instances, variable-length arrays (VARRAYs), and nested tables in an Oracle database server.

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

Figure 3–1 illustrates the containment hierarchy for value instances of all types in OO4O.

Figure 3–1 Supported Oracle Datatypes



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

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

Representing Objects in Visual Basic (OraObject)

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

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

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

Representing REFs in Visual Basic (OraRef)

The OraRef interface represents an Oracle object reference (REF) as well as referenceable objects in client applications. The object attributes are accessed in the same manner as attributes of an object represented by the OraObject interface. OraRef is derived from an OraObject interface 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, described in "Pre-Fetching Related Objects (Complex Object Retrieval)" on page 6-32.

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
'access all elements of the EnameList VArray
for I=1 to I=EnameList.Size
   msgbox EnameList(I)
Next I
```

Java: JDBC, Oracle SQLJ, JPublisher, and SQLJ Object Types

Java has emerged as a powerful, modern object-oriented language that provides developers with a simple, efficient, portable, and safe application development platform. Oracle provides two ways to integrate Oracle object features with Java:

JDBC and Oracle SQLJ. These interfaces enable you both to access SQL data from Java and to provide persistent database storage for Java objects.

JDBC Access to Oracle Object Data

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

Oracle's JDBC:

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

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

- Let JDBC materialize the object as a `STRUCT`. In this case, JDBC creates the classes for the attributes and populates them for you.
- 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 `CustomDatum` interface.

See Also: For complete information about JDBC, see the *Oracle9i 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 *Oracle9i 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.CustomDatum`. The Oracle JPublisher utility can automatically generate such custom Java classes.

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

Using JPublisher to Create Java Classes for JDBC and SQLJ Programs

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

- Use JPublisher to automatically generate custom Java classes and use those classes without any change.
- Subclass the classes produced by JPublisher to create your own specialized Java classes.
- Manually code custom Java classes without using JPublisher if the classes meet the requirements stated in the *Oracle9i SQLJ Developer's Guide and Reference*.

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

What JPublisher Produces

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

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

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

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

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

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

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

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

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

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

See Also: The *Oracle9i 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.

With Oracle9i, you can now 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:

```
CREATE 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 address RETURN full_address EXTERNAL NAME 'get_address ()
      return long_address',
    STATIC create RETURN person_t EXTERNAL NAME 'create () return Person',
    STATIC create (name VARCHAR(100), addrs full_address, bDate DATE)
      RETURN person_t EXTERNAL NAME 'create (java.lang.String, Long_address,
      oracle.sql.date) return Person',
    ORDER 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`, `CustomDatum`, 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`, `CustomDatum`, 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.

Sample SQLJ Object Type Mapping

The following code defines two Java classes. Then follows code that shows corresponding `CREATE TYPE` statements of the sort that go in a deployment descriptor. The code defines a one-to-one mapping of Java classes to SQL types, with all Java fields mapped to attributes in the SQL types.

```
package Examples;

import java.sql.*;
//import oracle.jdbc2.*;

// Java Address class based on SQLJ part 2

public class Address implements SQLData {
  protected String street;
  protected String city;
  protected String state;
  protected int zipCode;
  protected String sql_type;
  public static int recommendedWidth = 250;

  public Address () {
    street = "Unknown";
    city = "somewhere";
    state = "nowhere";
    zipCode = 0;
  }

  public Address (String st, String cit, String stt, int zip) {
    street = st;
    state = stt;
    city = cit;
    zipCode = zip;
  }

  protected static String strip(String in) {

    int len;
    int i;

    if (in == null) return in;
    if (in.charAt (0) != ' ') return in;

    len = in.length();

    for (i = 0; i < len && in.charAt(i) == ' '; i++) {}

    if (i == len) return null;

    return in.substring (i, len);
  }

  public String getSQLTypeName() throws SQLException
  {
    return sql_type;
  }

  public void readSQL(SQLInput stream, String typeName)
    throws SQLException
  {
    sql_type = typeName;
```

```
    street = stream.readString();
    city = stream.readString();
    state = stream.readString();
    zipCode = stream.readInt();
  }

  public void writeSQL(SQLOutput stream)
    throws SQLException
  {
    stream.writeString(street);
    stream.writeString(city);
    stream.writeString(state);
    stream.writeInt(zipCode);
  }

  public static Address create () {
    return new Address() ;
  }

  public static Address create (String st, String cit, String stt, int zp) {
    return new Address(st, cit, stt, zp);
  }

  public String toString() {
    return "Street" + street + "City" + city + "State" + state + zipCode ;
  }

  public Address removeLeadingBlanks () {
    // The definition of the Misc class has been omitted in this example.
    // Misc is fully described in the SQLJ part 2 specification.

    street = strip (street);
    city = strip (city) ;
    state = strip (state);
    return this;
  }
}
// create LongAddress as subclass of Address

public class LongAddress extends Address {

  protected String street2;
  protected String country;
  protected String addrCode ;
```

```
public LongAddress () {

  super();
  street2 = " ";
  country = " ";
  addrCode = " ";
}

public LongAddress
(String st, String st2, String ct, String stt, String cntry, String cd){
  street = st;
  street2 = st2;
  state = stt;
  country = cntry;
  city = ct;
  zipCode = 0;
  addrCode = cd;
}

public void readSQL(SQLInput stream, String typeName)
  throws SQLException
{
  sql_type = typeName;

  street = stream.readString();
  city = stream.readString();
  state = stream.readString();
  zipCode = stream.readInt();
  street2 = stream.readString();
  country = stream.readString();
  addrCode = stream.readString();

}

public void writeSQL(SQLOutput stream)
  throws SQLException
{
  stream.writeString(street);
  stream.writeString(city);
  stream.writeString(state);
  stream.writeInt(zipCode);
  stream.writeString(street2);
  stream.writeString(country);
  stream.writeString(addrCode);
  }

  public static Address create () {
    return new LongAddress();
  }

  public static Address create (
    String st, String st2, String ct, String stt, String cntry, String cd){
    return new LongAddress (st, st2, ct, stt, cntry, cd);
  }

  public String toString () {
    if (zipCode != 0)
      return "Street " + street + "City " + city + "State " + state +
        "Zip" + zipCode + "USA";
    else
      return "Street " + street + street2 + "City " + city + "State " +
        state + "Country " + country + addrCode ;
  }

  public Address removeLeadingBlanks () {
    // Misc class is not defined please refer to the SQLJ specs
    street = strip (street);
    street2 = strip (street2);
    city = strip (city) ;
    state = strip (state);
    country = strip (country);
    addrCode = strip (addrCode);
    return this;
  }
}
```

The following code might go in a deployment descriptor to create SQLJ types to correspond to the Java classes defined in the preceding code.

```
CREATE TYPE address_t AS OBJECT
  EXTERNAL NAME 'Examples.Address' LANGUAGE JAVA
  USING SQLData(
    street_attr varchar(250) EXTERNAL NAME 'street',
    city_attr varchar(50) EXTERNAL NAME 'city',
    state varchar(50) EXTERNAL NAME 'state',
    zip_code_attr number EXTERNAL NAME 'zipCode',
    STATIC FUNCTION recom_width RETURN NUMBER
      EXTERNAL VARIABLE NAME 'recommendedWidth',
    STATIC FUNCTION create_address RETURN address_t
```

```
      EXTERNAL NAME 'create() return Examples.Address',
    STATIC FUNCTION construct RETURN address_t
      EXTERNAL NAME 'create() return Examples.Address',
    STATIC FUNCTION create_address (street VARCHAR, city VARCHAR,
        state VARCHAR, zip NUMBER) RETURN address_t
      EXTERNAL NAME 'create (java.lang.String, java.lang.String,
        java.lang.String, int) return Examples.Address',
    STATIC FUNCTION construct (street VARCHAR, city VARCHAR,
        state VARCHAR, zip NUMBER) RETURN address_t
      EXTERNAL NAME
        'create (java.lang.String, java.lang.String, java.lang.String, int)
        return Examples.Address',
    MEMBER FUNCTION to_string RETURN VARCHAR
      EXTERNAL NAME 'tojava.lang.String() return java.lang.String',
    MEMBER FUNCTION strip RETURN SELF AS RESULT
      EXTERNAL NAME 'removeLeadingBlanks () return Examples.Address'
  ) NOT FINAL;
/
CREATE OR REPLACE TYPE long_address_t
UNDER address_t
EXTERNAL NAME 'Examples.LongAddress' LANGUAGE JAVA
USING SQLData(
    street2_attr VARCHAR(250) EXTERNAL NAME 'street2',
    country_attr VARCHAR (200) EXTERNAL NAME 'country',
    address_code_attr VARCHAR (50) EXTERNAL NAME 'addrCode',
    STATIC FUNCTION create_address RETURN long_address_t
      EXTERNAL NAME 'create() return Examples.LongAddress',
    STATIC FUNCTION  construct (street VARCHAR, city VARCHAR,
        state VARCHAR, country VARCHAR, addrs_cd VARCHAR)
      RETURN long_address_t
      EXTERNAL NAME
        'create(java.lang.String, java.lang.String, java.lang.String,
        java.lang.String, java.lang.String) return Examples.LongAddress',
    STATIC FUNCTION construct RETURN long_address_t
      EXTERNAL NAME 'Examples.LongAddress() return Examples.LongAddress',
    STATIC FUNCTION create_longaddress (
      street VARCHAR, city VARCHAR, state VARCHAR, country VARCHAR,
      addrs_cd VARCHAR) return long_address_t
      EXTERNAL NAME
        'Examples.LongAddress (java.lang.String, java.lang.String,
        java.lang.String, java.lang.String, java.lang.String)
        return Examples.LongAddress',
    MEMBER FUNCTION get_country RETURN VARCHAR
      EXTERNAL NAME 'country_with_code () return java.lang.String'
  );
```

```
/
```

More About Mapping

- 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`, `CustomDatum`, 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—namely, 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

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 Also: "Type Evolution" on page 6-8

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 Also: "Constraining Substitutability" on page 2-47

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.

Here is an example of a type declaration that involves a user-defined constructor implemented in Java:

```
CREATE OR REPLACE TYPE Person1_typ AS OBJECT (
  name VARCHAR2(30),
  age NUMBER
  CONSTRUCTOR FUNCTION Person1_typ(name VARCHAR2, age NUMBER)
    RETURN Person1_typ AS RESULT
  AS LANGUAGE JAVA
    NAME 'pkg1.J_Person.J_Person(java.lang.String, int) return J_Person'
);
```

4 Managing Oracle Objects

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

- Privileges on Object Types and Their Methods
- Dependencies and Incomplete Types
- Synonyms for User-Defined Types
- Tools
- Utilities

Privileges on Object Types and Their Methods

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

System Privileges

Oracle defines the following system privileges for object types:

- `CREATE TYPE` enables you to create object types in your own schema
- `CREATE ANY TYPE` enables you to create object types in any schema
- `ALTER ANY TYPE` enables you to alter object types in any schema
- `DROP ANY TYPE` enables you to drop named types in any schema
- `EXECUTE ANY TYPE` enables you to use and reference named types in any schema
- `UNDER ANY TYPE` enables you to create subtypes under any non-final object types
- `UNDER ANY VIEW` enables you to create subviews under any object view

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

Schema Object Privileges

Two schema object privileges apply to object types:

- `EXECUTE` on an object type enables you to use the type to:
  - Define a table.
  - Define a column in a relational table.
  - Declare a variable or parameter of the named type.

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

  Method execution and the associated permissions are the same as for stored PL/SQL procedures.
- `UNDER` enables you to create a subtype/subview under the type/view on which the privilege is granted

  The `UNDER` privilege on a subtype or subview can be granted only if the grantor has the `UNDER` privilege on the direct supertype or superview `WITH GRANT OPTION`.

The phrase WITH HIERARCHY OPTION grants a specified object privilege on all subobjects of the object. This option is meaningful only with the SELECT object privilege granted on an object view in an object view hierarchy. In this case, the privilege applies to all subviews of the view on which the privilege is granted.

Using Types in New Types or Tables

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

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

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

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

Example

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

USER1 performs the following DDL in the USER1 schema:

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

USER2 performs the following DDL in the USER2 schema:

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

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

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

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

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

USER3 can successfully perform the following actions:

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

Privileges on Type Access and Object Access

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

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

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

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

Consider the following schema:

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

CREATE TABLE emp OF emp_type;
```

and the following two queries:

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

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

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

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

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

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

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

- Pinning an object in the object cache using its REF value causes Oracle to check SELECT privilege on the object table containing the object and EXECUTE privilege on the object type. (For more information about the OCI object cache, see "LNOCI Tips and Techniques for Objects" on page 6-27.)
- Modifying an existing object or flushing an object from the object cache, causes Oracle to check UPDATE privilege on the destination object table. Flushing a new object causes Oracle to check INSERT privilege on the destination object table.
- Deleting an object causes Oracle to check DELETE privilege on the destination table.
- Invoking a method causes Oracle to check EXECUTE privilege on the corresponding object type.

Oracle does not provide column level privileges for object tables.

Dependencies and Incomplete Types

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

Types that depend on each other in this way, either directly or through intermediate types, are called *mutually dependent*. In a diagram that uses arrows to show the dependency relationships among a set of types, connections among mutually dependent types form a loop.

To define such a circular dependency, you must use REFs for at least one segment of the circle.

For example, you can define the following types:

```
CREATE TYPE department;

CREATE TYPE employee AS OBJECT (
  name    VARCHAR2(30),
  dept    REF department,
  supv    REF employee );

CREATE TYPE emp_list AS TABLE OF employee;

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

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

Notice that the preceding code creates the type `department` twice. The first statement:

```
CREATE TYPE department;
```

is an optional, *incomplete* declaration of `department` that serves as a placeholder for the `REF` attribute of `employee` to point to. The declaration is incomplete in that it omits the `AS OBJECT` phrase and lists no attributes or methods. These are specified later in the full declaration that completes the type. In the meantime, `department` is created as an *incomplete object type*. This enables the compilation of `employee` to proceed without errors.

To complete an incomplete type, you execute a `CREATE TYPE` statement that specifies the attributes and methods of the type, as shown at the end of the example. Complete an incomplete type after all the types that it refers to are created.

If you do not create incomplete types as placeholders, types that refer to the missing types still compile, but the compilation proceeds with errors.

For example, if `department` did not exist at all, Oracle would create it as an incomplete type and compile `employee` with errors. Then `employee` would be recompiled the next time that some operation attempts to access it. This time, if all the types it depends on are created and its dependencies are satisfied, it will compile without errors.

Incomplete types also enable you to create types that contain REF attributes to a subtype that has not yet been created. To create such a supertype, first create an incomplete type of the subtype to be referenced. Create the complete subtype after you create the supertype.

A subtype is just a specialized version of its direct supertype and consequently has an explicit dependency on it. To ensure that subtypes are not left behind after a supertype is dropped, all subtypes must be dropped first: a supertype cannot be dropped until all its subtypes are dropped.

Completing Incomplete Types

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

You must complete an incomplete object type as an object type: you cannot complete an object type as a collection type (a nested table type or an array type). The only alternative to completing a type declaration is to drop the type.

You must also complete any incomplete types that Oracle creates for you because you did not explicitly create them yourself. The example in the preceding section explicitly creates `department` as an incomplete type. If `department` were not explicitly created as an incomplete type, Oracle would create it as one so that the `employee` type can compile (with errors). You must complete the declaration of `department` as an object type whether you or Oracle declared it as an incomplete type.

Manually Recompiling a Type

If a type was created with compilation errors, and you attempt some operation on it, such as creating tables or inserting rows, you may receive an error, *Recompile type <typename> before attempting this operation*. To manually recompile a type, execute an `ALTER TYPE` *`typename`* `RECOMPILE` statement. After you have successfully compiled the type, attempt the operation again.

Type Dependencies of Substitutable Tables and Columns

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

So, for example, a persons table of type `Person_typ` is dependent not only on `Person_typ` but also on the `Person_typ` subtypes `Student_typ` and `PartTimeStudent_typ`.

If you attempt to drop a subtype that has a dependent type, table, or column, the `DROP TYPE` statement returns an error and aborts. For example, trying to drop `PartTimeStudent_typ` will raise an error because of the dependent `persons` table.

If dependent tables or columns exist but contain no data of the type that you want to drop, you can use the `VALIDATE` keyword to drop the type. The `VALIDATE` keyword causes Oracle to check for actual stored instances of the specified type and to drop the type if none are found. Hidden columns associated with attributes unique to the type are removed as well.

For example, the first `DROP TYPE` statement in the following example fails because `PartTimeStudent_typ` has a dependent table (`persons`). But if `persons` contains no instances of `PartTimeStudent_typ` (and no other dependent table or column does, either), the `VALIDATE` keyword causes the second `DROP TYPE` statement to succeed:

```
DROP TYPE PartTimeStudent_typ; -- Error due to presence of Persons table
DROP TYPE PartTimeStudent_typ VALIDATE; -- Succeeds if there are no stored
                                        -- instances of PartTimeStudent_typ
```

Note: Oracle recommends that you always use the `VALIDATE` option while dropping subtypes.

The FORCE Option

The `DROP TYPE` statement also has a `FORCE` option that causes the type to be dropped even though it may have dependent types or tables. The `FORCE` option should be used only with great care, as any dependent types or tables that do exist are marked invalid and become inaccessible when the type is dropped. Data in a table that is marked invalid because a type it depends on has been dropped can never be accessed again. The only action that can be performed on such a table is to drop it.

See Also: "Type Evolution" in Chapter 6 for information about how to alter a type

Synonyms for User-Defined Types

Just as you can create synonyms for tables, views, and various other schema objects, you can also define synonyms for user-defined types.

Synonyms for types have the same advantages as synonyms for other kinds of schema objects: they provide a location-independent way to reference the underlying schema object. An application that uses public type synonyms can be deployed without alteration in any schema of a database without having to qualify a type name with the name of the schema in which the type was defined.

> **See Also:** *Oracle9i Database Administrator's Guide* for more information on synonyms in general

Creating a Type Synonym

You create a type synonym with a `CREATE SYNONYM` statement. The statement has the following syntax:

```
CREATE [OR REPLACE] [PUBLIC] SYNONYM [<schema>.]<name> FOR
[<schema>.]<type>[@<dblink>];
```

where `<type>` is a user-defined type.

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

```
CREATE TYPE typ1 AS OBJECT (x number);
CREATE SYNONYM syn1 FOR typ1;
```

Synonyms can be created for collection types, too. The following example creates a synonym for a nested table type:

```
CREATE TYPE typ2 AS TABLE OF NUMBER;
CREATE SYNONYM syn2 FOR typ2;
```

You create a public synonym by using the `PUBLIC` keyword:

```
CREATE TYPE shape AS OBJECT ( name VARCHAR2(10) );
CREATE PUBLIC SYNONYM pub_shape FOR shape;
```

The `REPLACE` option enables you to have the synonym point to a different underlying type. For example, the following statement causes `syn1` to point to type `typ2` instead of the type it formerly pointed to:

```
CREATE OR REPLACE SYNONYM syn1 FOR typ2;
```

Using a Type Synonym

You can use a type synonym anywhere that you can refer to a type. For instance, you can use a type synonym in a DDL statement to name the type of a table column or type attribute. In the following example, synonym `syn1` is used to specify the type of an attribute in type `typ3`:

```
CREATE TYPE typ1 AS OBJECT (x number);
CREATE SYNONYM syn1 FOR typ1;
CREATE TYPE typ3 AS OBJECT ( a syn1 );
```

The next example shows a type synonym `syn1` used to call the constructor of the user-defined type `typ1`, for which `syn1` is a synonym. The statement returns an object instance of `typ1`:

```
SELECT syn1(0) FROM dual;
```

In the following example, `syn2` is a synonym for a nested table type. The example shows the synonym used in place of the actual type name in a `CAST` expression:

```
SELECT CAST(MULTISET(SELECT sal FROM SCOTT.EMP) AS syn2) FROM dual;
```

Table 4–1 lists the kinds of statements in which type synonyms can be used.

Table 4–1

DML Statements	DDL Statements
SELECT	AUDIT
INSERT	NOAUDIT
UPDATE	GRANT
DELETE	REVOKE
EXPLAIN PLAN	COMMENT
LOCK TABLE	

Describing Schema Objects That Use Synonyms

If a type or table has been created using type synonyms, the `DESCRIBE` command will show the synonyms in place of the types they represent. Similarly, catalog views (such as `USER_TYPE_ATTRS`) that show type names will show the associated type synonym names in their place.

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

Dependents of Type Synonyms

A type that directly or indirectly references a synonym in its type declaration is a dependent of that synonym. Thus, in the following example, type `typ3` is a dependent type of synonym `syn1`.

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

Other kinds of schema objects that reference synonoyms in their DDL statements also become dependents of those synonyms. An object that depends on a type synonym depends on both the synonym and on the synonym's underlying type.

A synonym's dependency relationships affect your ability to drop or rename the synonym. Dependent schema objects are also affected by some operations on synonyms. The following sections describe these various ramifications.

Restriction on Replacing a Type Synonym

You can replace a synonym only if it has no dependent tables or valid user defined types.

Replacing a synonym is equivalent to dropping it and then re-creating a new synonym with the same name.

Dropping Type Synonyms

You drop a synonym with the `DROP SYNONYM` statement:

```
DROP SYNONYM pubsyn1;
```

You cannot drop a type synonym if it has table or valid user-defined types as dependents unless you use the `FORCE` option. The `FORCE` option causes any columns that directly or indirectly depend on the synonym to be marked unused, just as if the actual types of the columns were dropped. (A column *indirectly* depends on a synonym if, for instance, the synonym is used to specify the type of an attribute of the declared type of the column.)

Any dependent schema objects of a dropped synonym are invalidated. They can be revalidated by creating a local object of the same name as the dropped synonym or by creating a new public synonym with same name.

Dropping the underlying base type of a type synonym has the same effect on dependent objects as dropping the synonym.

Renaming Type Synonyms

You can rename a type synonym with the RENAME statement. The following example renames synonym employee:

```
RENAME employee TO emp;
```

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 user-defined types.

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 user-defined type in the local schema where you want to create the new schema object. Nor can you create a local schema object that has the same name as a private synonym in the same schema.

For instance, in the following example, table tab1 is a dependent table of public synonym pubsyn1 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 pubsyn1—in the same schema as the dependent table—will fail:

```
CREATE TABLE tab1 ( c1 pubsyn1 );    -- Uses public synonym pubsyn1
CREATE TABLE pubsyn1 ( c1 NUMBER );  -- Not allowed
```

Tools

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

JDeveloper

JDeveloper is a full-featured, integrated development environment for creating multitier Java applications. It enables you to develop, debug, and deploy Java client applications, dynamic HTML applications, web and application server components and database stored procedures based on industry-standard models.

JDeveloper provides powerful features in the following areas:

- Oracle Business Components for Java
- Web Application Development

- Java Client Application Development
- Java in the Database
- Component-Based Development with JavaBeans
- Simplified Database Access
- Visual Integrated Development Environment
- Java Language Support

JDeveloper runs on Windows NT. It provides a standard GUI based Java development environment that is well integrated with Oracle's Application Server and Database.

Business Components for Java (BC4J)

Supporting standard EJB and CORBA deployment architectures, Oracle Business Components for Java simplifies the development, delivery, and customization of Java business applications for the enterprise. Oracle Business Components for Java is an application component framework providing developers a set of reusable software building blocks that manage all the common facilities required to:

- Author and test business logic in components which integrate with relational databases
- Reuse business logic through multiple SQL-based views of data
- Access and update the views from servlets, JavaServer Pages (JSPs), and thin-Java Swing clients
- Customize application functionality in layers without requiring modification of the delivered application

JPublisher

JPublisher is a utility, written entirely in Java, 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: *Oracle9i JPublisher User's Guide*

Utilities

Import/Export of Object Types

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

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

Types

The definition statements for derived types are exported. On an Import, a subtype may be created before the supertype definition has been imported. In this case, the subtype will be created with compilation errors, which may be ignored. The type will be revalidated after its supertype is created.

Object View Hierarchies

View definitions for all views belonging to a view hierarchy are exported

SQL*Loader

The SQL*Loader utility moves data from external files into tables in an Oracle database. The files may contain data consisting of basic scalar datatypes, such as `INTEGER`, `CHAR`, or `DATE`, as well as complex user-defined datatypes 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 datafile(s).

SQL*Loader provides two approaches to loading data:

- **Conventional path loading**, which uses the `SQL INSERT` statement and a bind array buffer to load data into database tables
- **Direct path loading**, which uses the Direct Path Load API to write data blocks directly to the database on behalf of the SQL*Loader client.

  Direct path loading does not use a SQL interface and thus avoids the overhead of processing the associated SQL statements. Consequently, direct path loading tends to provide much better performance than conventional path loading.

Either approach can be used to load data of supported object and collection datatypes.

See Also: *Oracle9i Database Utilities* for instructions on how to use SQL*Loader

5

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 to Use Object Views

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

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

Object views can be used like relational views to present only the data that you want users to see. For example, you might create an object view that presents selected data from an employee table but omits sensitive data about salaries.

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

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

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

See Also:

- *Oracle9i SQL Reference* for a complete description of SQL syntax and usage.
- *PL/SQL User's Guide and Reference* for a complete discussion of PL/SQL capabilities
- *Oracle9i Java Stored Procedures Developer's Guide* for a complete discussion of Java.
- *Oracle Call Interface Programmer's Guide* for a complete discussion of those facilities.

Defining Object Views

The procedure for defining an object view is:

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

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

4. Write an `INSTEAD OF` trigger procedure (see "Updating Object Views" on page 5-13) for Oracle to execute whenever an application program tries to update data in the object view.

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

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

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

CREATE TYPE employee_t  (
    empno    NUMBER (5),
    ename    VARCHAR2 (20),
    salary   NUMBER (9, 2),
    job      VARCHAR2 (20) );

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

To access the data from the `empnum` column of the relational table, you would access the `empno` attribute of the object type.

Using Object Views in Applications

Data in the rows of an object view may come from more than one table, but the object still traverses the network in one operation. 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 *LNOCIObjectPin()* for pinning a REF and *LNOCIObjectFlush()* for flushing an object to the server. When you update or flush to the server an object in an object view, Oracle updates the object view.

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

Nesting Objects in Object Views

An object type can have other object types nested in it as attributes.

If the object type on which an object view is based has an attribute that itself is an object type, then you must provide column objects for this attribute as part of the process of creating the object view. If column objects of the attribute type already exist in a relational table, you can simply select them; otherwise, you must synthesize the object instances from underlying relational data just as you synthesize the principal object instances of the view. You "synthesize" or create these objects by calling the respective object type's constructor method to create the object instances, and you populate their attributes with data from relational columns that you specify in the constructor.

For example, consider the department table `dept`:

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

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

1. Create the type for the address object:

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

2. Create the type for the department object:

```
CREATE TYPE dept_t AS OBJECT
(
   deptno     NUMBER,
   deptname   VARHCAR2(20),
   address    address_t
);
```

3. Create the view containing the department number, name and address. The `address` objects are constructed from columns of the relational table.

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

Identifying Null Objects in Object Views

Because the constructor for an object never returns a null, none of the address objects in the preceding view can ever be null, even if the city, street, and so on columns in the relational table are all null. The relational table has no column that specifies whether the department address is null. If we define a convention so that a null deptstreet column indicates that the whole address is null, then we can capture the logic using the DECODE function, or some other function, to return either a null or the constructed object:

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

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

Using Nested Tables and Varrays in Object Views

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

Single-Level Collections in Object Views

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

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

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

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

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

CREATE TYPE employee_list_t AS TABLE OF employee_t;
```

2. Define a department type having a department number, name, address, and a nested table of employees:

```
CREATE TYPE dept_t AS OBJECT
(   deptno     NUMBER,
    deptname   VARHCAR2(20),
    address    address_t,
    emp_list   employee_list_t
);
```

3. Define the object view `dept_view`:

```
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
                           FROM emp e
                           WHERE e.deptno = d.deptno)
                        AS employee_list_t)
                   AS emp_list
   FROM   dept d;
```

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

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

Multilevel Collections in Object Views

Multilevel collections and single-level collections are created and used in object views in the same way. The only difference is that, for a multilevel collection, you must create an additional level of collections.

The following example 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.

These are the underlying relational tables:

```
CREATE TABLE depts
  ( deptno     NUMBER
  , deptname   VARHCAR2(20));

CREATE TABLE emp
  ( ename VARCHAR2(20),
  , salary     NUMBER
  , deptname   VARHCAR2(20));

CREATE TABLE projects
  ( projname   VARHCAR2(20)
  , mgr        VARHCAR2(20));
```

These are the object and collection types the view will use:

```
CREATE TYPE project_t AS OBJECT
  ( projname   VARHCAR2(20)
  , mgr        VARHCAR2(20));

CREATE TYPE nt_project_t AS TABLE OF project_t;
CREATE TYPE emp_t AS OBJECT
  ( ename      VARCHAR2(20)
  , salary     NUMBER
  , deptname   VARHCAR2(20)
  , projects   nt_project_t);

CREATE TYPE nt_emp_t AS TABLE OF emp_t;

CREATE TYPE dept_t AS OBJECT
  ( deptno     NUMBER
  , deptname   VARHCAR2(20)
  , emps       nt_emp_t);
```

The following statement creates the object view:

```
CREATE VIEW v_depts OF dept_t AS
  SELECT d.deptno, d.deptname,
    CAST(MULTISET(SELECT e.ename, e.salary, e.deptname,
        CAST(MULTISET(SELECT p.projname, p.mgr
          FROM projects
          WHERE p.mgr = e.ename)
        AS nt_project_t)
      FROM emp
      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. Since the view data is not stored persistently, you must specify a set of distinct values to be used as object identifiers. The notion of object identifiers allows the objects in object views to be referenced and pinned in the object cache.

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

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

Oracle lets you specify object identifiers based on the primary key. The set of unique keys that identify the row object is turned into an identifier for the object. These values must be unique within the rows selected out of the view, since duplicates would lead to problems during navigation through object references.

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

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

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

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

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

```
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 e.empno, e.empname, e.salary
                           FROM emp e
                           WHERE e.deptno = d.deptno)
                      AS employee_list_t)
   FROM   dept d;
```

See Also: *Object Identifiers on page 6-7*

Creating References to View Objects

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

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

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

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

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

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

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

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

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

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

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

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

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

Modelling Inverse Relationships with Object Views

Views with objects can be used to model inverse relationships.

One-to-One Relationships

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

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

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

Consider the department-employee case. In the underlying relational model, we have the foreign key in the employee table. Using collections in views, we can model the relationship between departments and employees. The department view can have a collection of employees, and the employee view can have a reference to the department (or inline the department values). This gives us both the forward relation (from employee to department) and the inverse relation (department to list of employees). The department view can also have a collection of references to employee objects instead of embedding the employee objects.

Updating Object Views

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

A view is not directly updatable if its view query contains joins, set operators, aggregate functions, or `GROUP BY` or `DISTINCT` clauses. Also, individual columns of a view are not directly updatable if they are based on pseudocolumns or expression in the view query.

If a view is not directly updatable, you can still update it indirectly using `INSTEAD OF` triggers. To do so, you define an `INSTEAD OF` trigger for each kind of DML statement you want to execute on the view. In the `INSTEAD OF` trigger, you code the operations that must take place on the underlying tables of the view to accomplish the desired change in the view. Then, when you issue a DML statement for which you have defined an `INSTEAD OF` trigger, Oracle transparently runs the associated trigger.

> **See Also:** "Using INSTEAD OF Triggers to Control Mutating and Validation" on page 5-14 for an example of an `INSTEAD OF` trigger

Something you want to be careful of: In an object view hierarchy, `UPDATE` and `DELETE` statements operate polymorphically just as `SELECT` statements do: the set of rows picked out by an `UPDATE` or `DELETE` statement on a view implicitly includes qualifying rows in any subviews of the specified view as well.

For example, the following statement, which deletes all persons from `Person_v`, also deletes all students from `Student_v` and all employees from the `Employee_v` view.

```
DELETE FROM Person_v;
```

To exclude subviews and restrict the affected rows just to those in the view actually specified, use the `ONLY` keyword. For example, the following statement updates only persons and not employees or students.

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

Updating Nested Table Columns in Views

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

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

Using INSTEAD OF Triggers to Control Mutating and Validation

`INSTEAD OF` triggers provide a way of updating complex views that otherwise could not be updated. They can also be used to enforce constraints, check privileges and validate the DML. Using these triggers, you can control mutation of the objects created though an object view that might be caused by inserting, updating and deleting.

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

We implement the trigger by means of the following code:

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

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

  IF emp_count < 9 THEN
       -- let us do the insert
```

```
      INSERT INTO emp VALUES (:NEW.eno,:NEW.ename,:NEW.salary,dept_var.dno);
  END IF;
END;
```

Applying the Object Model to Remote Tables

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

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

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

```
CREATE VIEW emp_washington_view (eno,ename,salary)
   AS SELECT e.empno, e.empname, e.salary
          FROM   emp@washington_link e;

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

We can now create the global view:

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

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

Defining Complex Relationships in Object Views

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

For example, in the case of the department and employee, the department object currently includes a list of employees. To conserve space, we may want to put references to the employee objects inside the department object, instead of materializing all the employees within the department object. We can construct ("pin") the references to employee objects, and later follow the references using the dot notation to extract employee information.

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

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

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

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

Method 2: Create First View Using FORCE Keyword

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

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

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

We will see the implementation for both the methods.

Tables and Types to Demonstrate Circular View References

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

The `emp` table stores the employee information:

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

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

```
CREATE TYPE dept_t;
/
```

The employee type includes a reference to the department:

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

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

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

The department table is a typical relational table:

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

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

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

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

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

Creating Object Views with Circular References

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

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

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

```
CREATE VIEW emp_view OF emp_t WITH OBJECT IDENTIFIER(eno)
```

```
   AS SELECT e.empno, e.empname, e.salary,
                      NULL
         FROM emp e;
```

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

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

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

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

This creates the views.

Method 2: Create First View Using FORCE Keyword

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

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

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

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

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

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

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

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

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

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

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

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

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

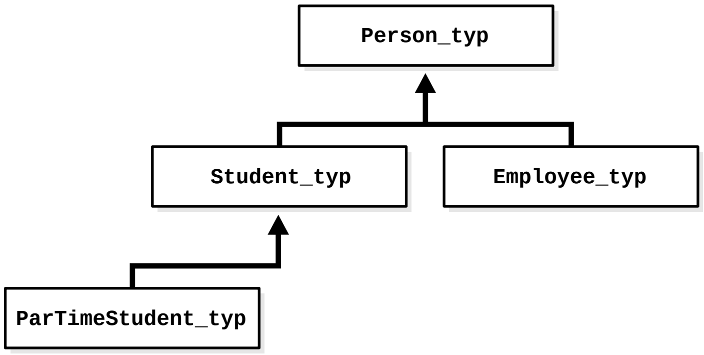
```
SELECT e.deptref.dno, DEREF(f.COLUMN_VALUE)
FROM emp_view e, TABLE(e.deptref.empreflist) f
WHERE e.deptref.dno = 100
AND f.COLUMN_VALUE.ename like ‘John%’;
```

Object View Hierarchies

An object view hierarchy is a set of object views each of which is based on a different type in a type hierarchy. Subviews in a view hierarchy are created *under* a superview, analogously to the way subtypes in a type hierarchy are created under a supertype.

Each object view in a view hierarchy is populated with objects of a single type, but queries on a given view implicitly address its subviews as well. Thus an object view hierarchy gives you a simple way to frame queries that can return a polymorphic set of objects of a given level of specialization or greater.

For example, suppose you have the following type hierarchy, with `Person_typ` as the root:



If you have created an object view hierarchy based on this type hierarchy, with an object view built on each type, you can query the object view that corresponds to the level of specialization you are interested in. For instance, you can query the view of `Student_typ` to get a result set that contains only students, including part-time students.

You can base the root view of an object view hierarchy on any type in a type hierarchy: you do not need to start the object view hierarchy at the root type. Nor do you need to extend an object view hierarchy to every leaf of a type hierarchy or cover every branch. However, you cannot skip intervening subtypes in the line of descent. Any subview must be based on a direct subtype of the type of its direct superview.

Just as a type can have multiple sibling subtypes, an object view can have multiple sibling subviews. But a subview based on a given type can participate in only one object view hierarchy: two different object view hierarchies cannot each have a subview based on the same subtype.

A subview inherits the object identifier (OID) from its superview. An OID cannot be explicitly specified in any subview.

A root view can explicitly specify an object identifier using the `WITH OBJECT ID` clause. If the OID is system-generated or the clause is not specified in the root view, then subviews can be created only if the root view is based on a table or view that also uses a system generated OID.

The query underlying a view determines whether the view is updatable. For a view to be updatable, its query must contain no joins, set operators, aggregate functions, `GROUP BY`, `DISTINCT`, pseudocolumns, or expressions. The same applies to subviews.

If a view is not updatable, you can define `INSTEAD OF` triggers to perform appropriate DML actions. Note that `INSTEAD OF` triggers are not inherited by subviews.

All views in a view hierarchy must be in the same schema.

Note: In Oracle9*i* 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.

```
CREATE VIEW Student_v OF Student_typ UNDER Person_v
  AS
  SELECT ssn, name, address, deptid, major
  FROM AllPersons
  WHERE typeid = 2;
```

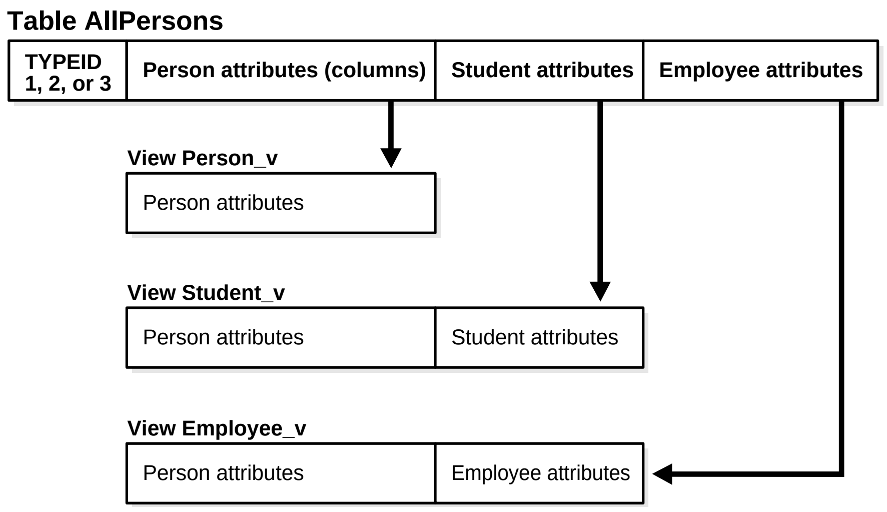
The same object view hierarchy can be based on different underlying storage models. In other words, a variety of layouts or designs of underlying tables can produce the same object view hierarchy. The design of the underlying storage model has implications for the performance and updatability of the object view hierarchy.

The following examples show three possible storage models. In the first, "flat" model, all views in the object view hierarchy are based on the same table. In the second, "horizontal" model, each view has a one-to-one correspondence with a different table. And in the third, "vertical" model, the views are constructed using joins.

The Flat Model

In the "flat" model, all the views in the hierarchy are based on the same table. In the following example, the single table `AllPersons` contains columns for all the attributes of `Person_typ`, `Student_typ`, or `Employee_typ`.

Figure 5–1 Flat Storage Model for Object View Hierarchy



```
CREATE TABLE AllPersons
( typeid NUMBER(1),
  ssn NUMBER,
  name VARCHAR2(30),
  address VARCHAR2(100),
  deptid NUMBER,
  major VARCHAR2(30),
  empid NUMBER,
  mgr VARCHAR2(30));
```

The `typeid` column identifies the type of each row. Possible values are:

```
1 = Person_typ
2 = Student_typ
3 = Employee_typ
```

The following statements create the views that make up the object view hierarchy:

```
CREATE VIEW Person_v OF Person_typ
  WITH OBJECT OID(ssn) AS
  SELECT ssn, name, address
  FROM AllPersons
  WHERE typeid = 1;
```

```
CREATE VIEW Student_v OF Student_typ UNDER Person_v
  AS
  SELECT ssn, name, address, deptid, major
  FROM AllPersons
  WHERE typeid = 2;

CREATE VIEW Employee_v OF Employee_typ UNDER Person_v
  AS
  SELECT ssn, name, address, empid, mgr
  FROM AllPersons
  WHERE typeid = 3;
```

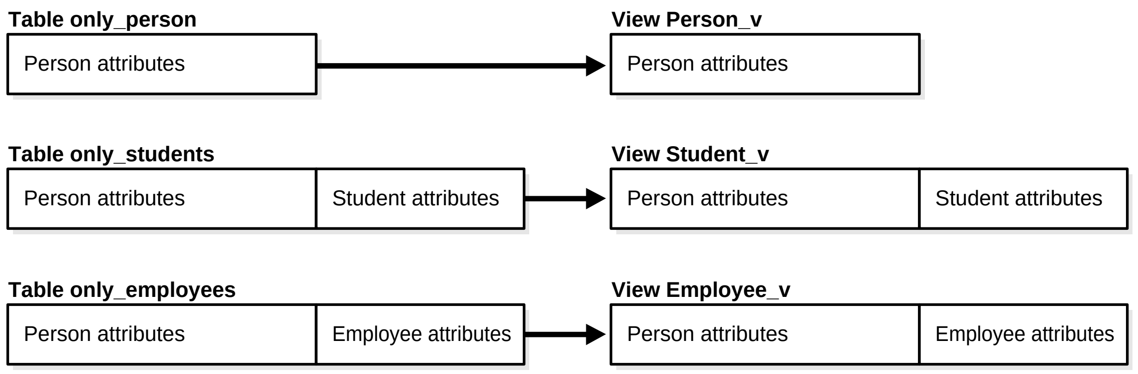
The flat model has the advantage of simplicity and poses no obstacles to supporting indexes and constraints. Its drawbacks are:

- A single table cannot contain more than 1000 columns, so the flat model imposes a 1000-column limit on the total number of columns that the object view hierarchy can contain.
- Each row of the table will have NULLs for all the attributes not belonging to its type. Such non-trailing NULLs can adversely affect performance.

The Horizontal Model

On the horizontal model, each view or subview is based on a different table. (In the example, the tables are relational, but they could just as well be object tables for which column substitutability is turned off.)

Figure 5–2 Horizontal Storage Model for Object View Hierarchy



```
CREATE TABLE only_persons
( ssn NUMBER,
  name VARCHAR2(30),
  address VARCHAR2(100));

CREATE TABLE only_students
( ssn NUMBER,
  name VARCHAR2(30),
  address VARCHAR2(100),
  deptid NUMBER,
  major VARCHAR2(30));

CREATE TABLE only_employees
( ssn NUMBER,
  name VARCHAR2(30),
  address VARCHAR2(100),
  empid NUMBER,
  mgr VARCHAR2(30));
```

These are the views:

```
CREATE VIEW Person_v OF Person_typ
  WITH OBJECT OID(ssn) AS
  SELECT *
  FROM only_persons

CREATE VIEW Student_v OF Student_typ UNDER Person_v
  AS
  SELECT *
  FROM only_students;

CREATE VIEW Employee_v OF Employee_typ UNDER Person_v
  AS
  SELECT *
  FROM only_employees;
```

The horizontal model is very efficient at processing queries of the form:

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

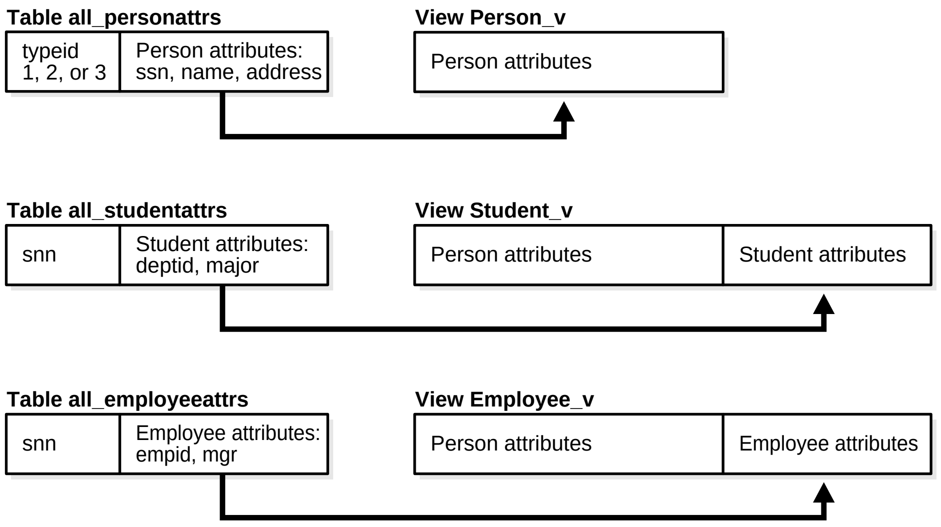
Such queries need access only a single physical table to get all the objects of the specific type. The drawbacks of this model are that queries of the sort `SELECT * FROM` *`view`* require performing a `UNION` over all the underlying tables and projecting the rows over just the columns in the specified view. (See "Querying a

View in a Hierarchy" on page 5-28.) Also, indexes on attributes (and unique constraints) must span multiple tables, and support for this does not currently exist.

The Vertical Model

In the vertical model, there is a physical table corresponding to each view in the hierarchy, but each physical table stores only those attributes that are unique to its corresponding subtype.

Figure 5–3 Vertical Storage Model for Object View Hierarchy



```
CREATE TABLE all_personattrs
( typeid NUMBER,
  ssn NUMBER,
  name VARCHAR2(30),
  address VARCHAR2(100));

CREATE TABLE all_studentattrs
( ssn NUMBER,
  deptid NUMBER,
  major VARCHAR2(30));

CREATE TABLE all_employeeattrs
( ssn NUMBER,
  empid NUMBER,
  mgr VARCHAR2(30));

CREATE VIEW Person_v OF Person_t
WITH OBJECT OID(ssn) AS
  SELECT ssn, name, address
  FROM all_personattrs
  WHERE typeid = 1;

CREATE VIEW Student_v OF Student_t 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 VIEW Employee_v OF Employee_t UNDER Person_v
  AS
  SELECT x.ssn, x.name, x.address, y.empid, y.mgr
  FROM all_personattrs x, all_studentattrs 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 contraints on them. However, to re-create an instance of a type, a join over OIDs must be performed for each level that the type is removed from the root in the hierarchy.

Querying a View in a Hierarchy

You can query any view or subview in an object view hierarchy; rows are returned for the declared type of the view that you query and for any of that type's subtypes. So, for instance, in an object view hierarchy based on the `Person_typ` type hierarchy, you can query the view of `Person_typ` to get a result set that contains all persons, including students and employees; or you can query the view of `Student_typ` to get a result set that contains only students, including part-time students.

In the `SELECT` list of a query, you can include either functions such as `REF()` and `VALUE()` that return an object instance, or you can specify object attributes of the view's declared type, such as the `name` and `ssn` attributes of `Person_typ`.

If you specify functions, to return object instances, the query returns a polymorphic result set: that is, it returns instances of both the view's declared type and any subtypes of that type.

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

```
SELECT REF(p), VALUE(p) FROM Person_v p;
```

If you specify individual attributes of the view's declared type in the SELECT list or do a SELECT * , again the query returns rows for the view's declared type and any subtypes of that type, but these rows are projected over columns for the attributes of the view's declared type, and only those columns are used. In other words, the subtypes are represented only with respect to the attributes they inherit from and share with the view's declared type.

So, for example, the following query returns rows for all persons and rows for employees and students of all types, but the result uses only the columns for the attributes of Person_typ—namely, name, ssn, and address. It does not show rows for attributes added in the subtypes, such as the deptid attribute of Student_typ.

```
SELECT * FROM Person_v;
```

To exclude subviews from the result, use the ONLY keyword. The ONLY keyword confines the selection to the declared type of the view that you are querying:

```
SELECT VALUE(p) FROM ONLY(Person_v) p;
```

Privileges for Operations on View Hierarchies

Generally, a query on a view with subviews requires only the SELECT privilege on the view being referenced and does not require any explicit privileges on subviews. For example, the following query requires only SELECT privileges on Person_v but not on any of its subviews.

```
SELECT * FROM Person_v;
```

However, a query that selects for any attributes added in subtypes but not used by the root type requires the SELECT privilege on all subviews as well. Such subtype attributes may hold sensitive information that should reasonably require additional privileges to access.

The following query, for example, requires SELECT privileges on Person_v and also on Student_v, Employee_v (and on any other subview of Person_v) because the query selects object instances and thus gets all the attributes of the subtypes.

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

To simplify the process of granting SELECT privileges on an entire view hierarchy, you can use the HIERARCHY option. Specifying the HIERARCHY option when granting a user SELECT privileges on a view implicitly grants SELECT privileges on all current and future subviews of the view as well. For example:

```
GRANT SELECT ON Person_v TO scott 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);
```

6

Advanced Topics for Oracle Objects

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

The chapter contains these topics:

Storage of Objects

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

Leaf-Level Attributes

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

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

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

How Row Objects are Split Across Columns

In an object table, Oracle stores the data for every leaf-level scalar or REF attribute in a separate column. Each VARRAY is also stored in a column, unless it is too large (see "Internal Layout of VARRAYs" on page 6-5). Oracle stores leaf-level attributes of nested table types in separate tables associated with the object table. You must declare these tables as part of the object table declaration (see "Internal Layout of Nested Tables" on page 6-4).

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

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

Hidden Columns for Tables with Column Objects

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

Hidden Columns for Substitutable Columns and Tables

A substitutable column or object table has a hidden column not only for each attribute of the column's object type but also for each attribute added in any subtype of the object type. These columns store the values of those attributes for any subtype instances inserted in the substitutable column.

For example, a substitutable column of `Person_typ` will have associated with it a hidden column for each of the attributes of `Person_typ`, namely: `ssn`, `name`, `address`. It will also have hidden columns for attributes of the subtypes of `Person_typ`: for example, the attributes `deptid` and `major` (for `Student_typ`) and `numhours` (for `PartTimeStudent_typ`).

When a subtype is created, hidden columns for attributes added in the subtype are automatically added to tables containing a substitutable column of any of the new subtype's ancestor types. These retrofit the tables to store data of the new type. If, for some reason, the columns cannot be added, creation of the subtype is rolled back.

When a subtype is dropped with the `VALIDATE` option to `DROP TYPE`, all such hidden columns for attributes unique to the subtype are automatically dropped as well if they do not contain data.

A substitutable column also has associated with it a hidden **type discriminant** column. This column contains an identifier, called a **typeid**, that identifies the most specific type of each object in the substitutable column. Typically, a typeid (`RAW`) is one byte, though it can be as big as four bytes for a large hierarchy.

You can find the typeid of a specified object instance using the function—`SYS_TYPEID`. For example, suppose that the substitutable object table `persons` contains three rows, as follows:

```
CREATE TABLE persons OF Person_typ;

INSERT INTO persons
  VALUES (Person_typ(1243, 'Bob', '121 Front St'));

INSERT INTO persons
  VALUES (Student_typ(3456, 'Joe', '34 View', 12, 'HISTORY'));

INSERT INTO persons
  VALUES (PartTimeStudent_typ(5678, 'Tim', 13, 'PHYSICS', 20));
```

The following query gets typeids of object instances stored in the table:

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

```
NAME   TYPEID
----   ------
Bob    01
Joe    02
Tim    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" in Chapter 2 for more information about SYS_TYPEID and typeids

REFs

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

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

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

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

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

Internal Layout of Nested Tables

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

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

A nested table type can contain objects or scalars:

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

For more information, see Nested Table Storage on page 8-16.

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 8-15 for details.

Creating Indexes on Typeids or Attributes

Indexing a Type Discriminant Column

Using the SYS_TYPEID function, you can build an index on the hidden type discriminant column that every substitutable column has. The type discriminant column contains typeids that identify the most specific type of every object instance stored in the substitutable column. This information is used by the system to evaluate queries that use the IS OF predicate to filter by type, but you can access the typeids for your own purposes using the SYS_TYPEID function.

Note: 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 author column of table books. Function SYS_TYPEID is used to reference the type discriminant column:

```
CREATE BITMAP INDEX typeid_i ON books (SYS_TYPEID(author));
```

Indexing Subtype Attributes of a Substitutable Column

You can build an index on attributes of any of the types that can be stored in a substitutable column. Attributes of subtypes can be referenced in the CREATE INDEX statement by using the TREAT function to filter out types other than the desired subtype (and its subtypes); you then use the dot notation to specify the desired attribute.

For example, the following statement creates an index on the major attribute of all student authors in the books table. The declared type of the author column is Person_typ, of which Student_typ is a subtype, so the column may contain instances of Person_typ, Student_typ, and subtypes of either one:

```
CREATE INDEX major_i ON books
  (TREAT(author AS Student_typ).major);
```

Student_typ is the type that first defined the major attribute: the Person_typ supertype does not have it. Consequently, all the values in the hidden column for the major attribute are values for Student_typ or PartTimeStudent_typ authors (a Student_typ subtype). This means that the hidden column's values are identical to the values returned by the TREAT expression, which returns major values for all students, including student subtypes: both the hidden column and the TREAT expression list majors for students and nulls for authors of other types. The system exploits this fact and creates index major_i as an ordinary btree index on the hidden column.

Values in a hidden column are identical to the values returned by a TREAT expression like the preceding one only if the type named as the target of the TREAT function (Student_typ) is the type that first defined the attribute. If the target of the TREAT function is a subtype that merely inherited the attribute, as in the following example, the TREAT expression will return non-null major values for the subtype (part-time students) but not for its supertype (other students).

```
CREATE INDEX major_func_i ON books
  (TREAT(author AS PartTimeStudent_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 btree index cannot be created on the underlying column.

In a case like this, Oracle treats the `TREAT` expression like any other function-based expression and tries to create the index as a function-based index on the result. However, creating a function-based index requires some privileges and session settings beyond those required to create a btree index.

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 `persons`:

```
CREATE INDEX major_func_i2 ON persons p
  (TREAT(VALUE(p) AS PartTimeStudent_typ).major);
```

Object Identifiers

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

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

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

Primary-key Based Object Identifiers

In an environment where a locally unique identifier can be assumed to be globally unique (in other words, where the table is not distributed or replicated), you can use the primary key value of a row object as its object identifier. Doing this saves the 16 bytes of storage for each object that a system-generated identifer requires.

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

Type Evolution

Changing a user-defined type is called **type evolution**. You can make the following changes to a user-defined type:

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

Changes to a type affect things that reference the type. For example, if you add a new attribute to a type, data in a column of that type must be presented so as to include the new attribute.

Schema objects that directly or indirectly reference a type and are affected by a change to it are called **dependents** of the type. A type can have these kinds of dependents:

- Table
- Type or subtype
- Program unit (PL/SQL block): procedure, function, package, trigger
- Indextype
- View (including object view)
- Function-based index
- Operator

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

All dependent program units, views, operators and indextypes are marked invalid when a type is modified. The next time one of these invalid schema objects is referenced, it is revalidated using the new type definition. If the object recompiles successfully, it becomes valid and can be used again. (Depending on the change to the type, function-based indexes may be dropped or disabled and need to be rebuilt.)

If a type has dependent tables, then, for each attribute added to a type, one or more internal columns are added to the table depending on the new attribute's type. New attributes are added with `NULL` values. For each dropped attribute, the columns associated with that attribute are dropped. For each modified attribute, the length, precision, or scale of its associated column is changed accordingly.

These changes mainly involve updating the tables' metadata (information about a table's structure, describing its columns and their types) and can be done quickly. However, the data in those tables must be updated to the format of the new type version as well. Updating this data can be time-consuming if there is a lot of it, so the ALTER TYPE command has options to let you choose whether to convert all dependent table data immediately or to leave it in the old format to be converted piecemeal as it is updated in the course of business.

The CASCADE option for ALTER TYPE propagates a type change to dependent types and tables (see "ALTER TYPE Options for Type Evolution" on page 6-16). CASCADE itself has options that let you choose whether to convert table data to the new type format as part of the propagation: the option INCLUDING TABLE DATA converts the data; the option NOT INCLUDING TABLE DATA does not convert it. By default, the CASCADE option converts the data. In any case, table data is always returned in the format of the latest type version. If the table data is stored in the format of an earlier type version, Oracle converts the data to the format of the latest version before returning it, even though the format in which the data is actually *stored* is not changed until the data is rewritten.

You can retrieve the definition of the latest type from the system view USER_SOURCE. You can view definitions of all versions of a type in the USER_TYPE_VERSIONS view.

The following example changes 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.

```
CREATE TYPE person_typ AS OBJECT (
  first_name   VARCHAR(30),
  last_name    VARCHAR(30),
  age          NUMBER(3));

CREATE TABLE person_tab of person_typ;

INSERT INTO person_tab VALUES
  (person_typ ('John', 'Doe', 50));

SELECT value(p) FROM person_tab p;

VALUE(P)(FIRST_NAME, LAST_NAME, AGE)
----------------------------------------------
```

```
PERSON_TYP('John', 'Doe', 50)

ALTER TYPE person_typ
  ADD ATTRIBUTE (dob DATE),
  DROP ATTRIBUTE age CASCADE NOT INCLUDING TABLE DATA;

-- The data of table person_tab 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_tab p;
VALUE(P)(FIRST_NAME, LAST_NAME, DOB)
----------------------------------------------
PERSON_TYP('John', 'Doe', NULL)
```

During SELECT statements, even though column data may be converted to the latest type version, the converted data is not written back to the column. If a certain user-defined type column in a table is retrieved often, you should consider converting that data to the latest type version to eliminate redundant data conversions. Converting is especially beneficial if the column contains a VARRAY attribute since a VARRAY typically takes more time to convert than an object or nested table column.

You can convert a column of data by issuing an UPDATE statement to set the column to itself. For example:

```
UPDATE dept_tab SET emp_array_col = emp_array_col;
```

You can convert all columns in a table by using ALTER TABLE UPGRADE DATA.

For example:

```
ALTER TYPE person_typ ADD ATTRIBUTE (photo BLOB)
  CASCADE NOT INCLUDING TABLE DATA;
ALTER TABLE dept_tab UPGRADE INCLUDING DATA;
```

Changes Involved When a Type Is Altered

Only structural changes to a type affect dependent data and require the data to be converted. Changes that are confined to a type's method definitions or behavior (in the type body, where the type's methods are implemented) do not.

These possible changes to a type are structural:

- Adding an attribute

- Dropping an attribute
- Modifying the length, precision, or scale of an attribute
- Changing the finality of a type (which determines whether subtypes can be derived from it) from `FINAL` to `NOT FINAL` or from `NOT FINAL` to `FINAL`.

These changes result in new versions of the altered type and all its dependent types and require the system to add, drop, or modify internal columns of dependent tables as part of the process of converting to the new version.

When you make any of these kinds of changes to a type that has dependent types or tables, the effects of propagating the change are not confined only to metadata but affect data storage arrangements and require the data to be converted.

Besides converting data, you may also need to make other changes. For example, if a new attribute is added to a type, and the type body invokes the type's constructor, then each constructor in the type body must be modified to specify a value for the new attribute. Similarly, if a new method is added, then the type body must be replaced to add the implementation of the new method. The type body can be modified by using the `CREATE OR REPLACE TYPE BODY` statement.

Steps to Change a Type

Here are the steps required to make a change to a type:

Assume we have the following schema:

```
CREATE TYPE Person_typ AS OBJECT
( name     CHAR(20),
  ssn      CHAR(12),
  address  VARCHAR2(100));

CREATE TYPE Person_nt IS TABLE OF Person_typ;
CREATE TYPE dept_typ AS OBJECT
( mgr    Person_typ,
  emps   Person_nt);

CREATE TABLE dept OF dept_typ;
```

1. Issue an `ALTER TYPE` statement to alter the type.

   The default behavior of an `ALTER TYPE` statement without any option specified is to check if there is any object dependent on the target type. The statement aborts if any dependent object exists. Optional keywords allow cascading the type change to dependent types and tables.

In the following code, conversion of table data is deferred by adding the phrase `NOT INCLUDING TABLE DATA`.

```
-- Add new attributes to Person_typ and propagate the change
-- to Person_nt and dept_typ
ALTER TYPE Person_typ ADD ATTRIBUTE (picture BLOB, dob DATE)
  CASCADE NOT INCLUDING TABLE DATA;
```

2. Use `CREATE OR REPLACE TYPE BODY` to update the corresponding type body to make it current with the new type definition.
3. Upgrade dependent tables to the latest type version and convert the tables' data.

   ```
   ALTER TABLE dept UPGRADE INCLUDING DATA;
   ```

4. Alter dependent PL/SQL program units as needed to take account of changes to the type.
5. Use OTT or JPUB (or another tool) to generate new header files for applications, depending on whether the application is written in C or Java.

   Adding a new attribute to a supertype also increases the number of attributes in all its subtypes because these inherit the new attribute. Inherited attributes always precede declared (locally defined) attributes, so adding a new attribute to a supertype causes the ordinal position of all declared attributes of any subtype to be incremented by one recursively. The mappings of the altered type must be updated to include the new attributes. OTT and JPUB do this. If you use some other tool, you must be sure that the type headers are properly synchronized with the type definition in the server; otherwise, unpredictable behavior may result.
6. Modify application code as needed and rebuild the application.

Validating a Type

When the system executes an `ALTER TYPE` statement, it first validates the requested type change syntactically and semantically to make sure it is legal. The system performs the same validations as for a `CREATE TYPE` statement plus some additional ones. For example, it checks to be sure an attribute being dropped is not used as a partitioning key. If the new spec of the target type or any of its dependent types fails the type validations, the `ALTER TYPE` statement aborts. No new type version is created, and all dependent objects remain unchanged.

If dependent tables exist, further checking is done to ensure that restrictions relating to the tables and any indexes are observed. Again, if the ALTER TYPE statement fails the check of table-related restrictions, then the type change is aborted, and no new version of the type is created.

When multiple attributes are added in a single ALTER TYPE statement, they are added in the order specified. Multiple type changes can be specified in the same ALTER TYPE statement, but no attribute name or method signature can be specified more than once in the statement. For example, adding and modifying the same attribute in a single statement is not allowed.

For example:

```
CREATE TYPE mytype AS OBJECT (attr1  NUMBER, attr2 NUMBER);
ALTER TYPE mytype ADD ATTRIBUTE (attr3 NUMBER),
  DROP ATTRIBUTE attr2,
  ADD ATTRIBUTE attr4 NUMBER CASCADE;
```

The resulting definition for mytype becomes:

```
  (attr1 NUMBER, attr3 NUMBER, attr4 NUMBER);
```

The following ALTER TYPE statement, which attempts to make multiple changes to the same attribute (`attr5`), is invalid:

```
-- invalid ALTER TYPE statement
ALTER TYPE mytype ADD ATTRIBUTE (attr5 NUMBER, attr6 CHAR(10)),
  DROP ATTRIBUTE attr5;
```

The following are other notes on validation constraints, table restrictions, and assorted information about the various kinds of changes that can be made to a type.

Dropping an attribute

- Dropping all attributes from a root type is not allowed. You must instead drop the type. Since a subtype inherits all the attributes from its supertype, dropping all the attributes from a subtype does not reduce its attribute count to zero; thus, dropping all attributes declared locally in a subtype is allowed.
- Only an attribute declared locally in the target type can be dropped. You cannot drop an inherited attribute from a subtype. Instead, drop the attribute from the type where it is locally declared.
- Dropping an attribute which is part of a table partitioning or sub-partitioning key in a table is not allowed.

- Dropping an attribute of a primary key OID of an object table or an index-organized table (IOT) is not allowed.
- When an attribute is dropped, the column corresponding to the dropped attribute is dropped.
- Indexes, statistics, constraints, and any referential integrity constraints referencing a dropped attribute are removed.

Modifying attribute type (to increase the length, precision or scale)

- Expanding the length of an attribute referenced in a function-based index, clustered key or domain index on a dependent table is not allowed.

Dropping a method

- You can drop a method only from the type in which the method is defined or overridden: You cannot drop an inherited method from a subtype, and you cannot drop an override from a supertype.
- If a method is *not* overridden, dropping it using the `CASCADE` option removes the method from the target type and all subtypes. However, if a method is overridden in a subtype, the `CASCADE` will fail and roll back. For the `CASCADE` to succeed, you must first drop each override from the subtype that defines it and only then drop the method from the supertype.

  You can consult the `USER_DEPENDENCIES` table to find all the schema objects, including types, that depend on a given type.
- You can use the `INVALIDATE` option to drop a method that has overrides, but the overrides in the subtypes must still be dropped manually. The subtypes will remain in an invalid state until they are explicitly altered to drop the overrides. (Until then, an attempt to recompile the subtypes for revalidation will produce the error, "Method does not override.")

  Unlike `CASCADE`, `INVALIDATE` bypasses all the type and table checks and simply invalidates all schema objects dependent on the type. The objects are revalidated the next time they are accessed. This option is faster than using `CASCADE`, but you must be certain that no problems will be encountered revalidating dependent types and tables. Table data cannot be accessed while a table is invalid; if a table cannot be validated, its data remains inaccessible.

See Also: "If a Type Change Validation Fails" on page 6-16

Modifying the FINAL or INSTANTIABLE property

- Altering a user-defined type from INSTANTIABLE to NOT INSTANTIABLE is allowed only if the type has no table dependents.
- Altering a user-defined type from NOT INSTANTIABLE to INSTANTIABLE is allowed anytime. This change does not affect tables.
- Altering a user-defined type from NOT FINAL to FINAL is allowed only if the target type has no subtypes.
- When you alter a user-defined type from FINAL to NOT FINAL or vice versa, you must use CASCADE to convert data in dependent columns and tables immediately. You may not use the CASCADE option NOT INCLUDING TABLE DATA to defer converting data.

  If you alter a type from NOT FINAL to FINAL, you must use CASCADE INCLUDING TABLE DATA. If you alter a type from FINAL to NOT FINAL, you may use either CASCADE INCLUDING TABLE DATA or CASCADE CONVERT TO SUBSTITUTABLE.

  When you alter a type from FINAL to NOT FINAL. the CASCADE option you should choose depends on whether you want to be able to insert new subtypes of the type you are altering in existing columns and tables.

  By default, altering a type from FINAL to NOT FINAL enables you to create *new* substitutable tables and columns of that type, but it does not automatically make existing columns (or object tables) of that type substitutable. In fact, just the opposite happens: existing columns and tables of the type are marked NOT SUBSTITUTABLE AT ALL LEVELS. If any embedded attribute of such a column is substitutable, an error is generated. New subtypes of the altered type cannot be inserted in such preexisting columns and tables.

  To alter a user-defined type to NOT FINAL in such a way as to make existing columns and tables of the type substitutable (assuming that they are not marked NOT SUBSTITUTABLE), use the CASCADE option CONVERT TO SUBSTITUTABLE. For example:

  ```
  ALTER TYPE t NOT FINAL CASCADE CONVERT TO SUBSTITUTABLE;
  ```

  This CASCADE option marks each existing column as SUBSTITUTABLE AT ALL LEVELS and causes a new, hidden column to be added for the TypeId of instances stored in the column. The column can then store subtype instances of the altered type.

If a Type Change Validation Fails

The INVALIDATE option of the ALTER TYPE statement lets you alter a type without propagating the type change to dependent objects. In this case, the system does not validate the dependent types and tables to ensure that all the ramifications of the type change are legal. Instead, all dependent schema objects are marked invalid. The objects, including types and tables, are revalidated when next referenced. If a type cannot be revalidated, it remains invalid, and any tables referencing it become inaccessible until the problem is corrected.

A table may fail validation because, for example, adding a new attribute to a type has caused the number of columns in the table to exceed the maximum allowable number of 1000, or because an attribute used as a partitioning or clustering key of a table was dropped from a type.

To force a revalidation of a type, users can issue the ALTER TYPE COMPILE statement. To force a revalidation of an invalid table, users can issue the ALTER TABLE UPGRADE statement and specify whether the data is to be converted to the latest type version. (Note that, in a table validation triggered by the system when a table is referenced, table data is always updated to the latest type version: you do not have the option to postpone conversion of the data.)

If a table is unable to convert to the latest type version, then INSERT, UPDATE and DELETE statements on the table are not allowed and its data becomes inaccessible. The following DDLs can be executed on the table, but all other statements which reference an invalid table are not allowed until the table is successfully validated:

- DROP TABLE
- TRUNCATE TABLE

All PL/SQL programs containing variables defined using %ROWTYPE of a table or %TYPE of a column or attribute from a table are compiled based on the latest type version. If the table fails the revalidation, then compiling any program units that reference that table will also fail.

ALTER TYPE Options for Type Evolution

The following is a synopsis of the options in the ALTER TYPE statement for altering the attribute or method definition of a type:

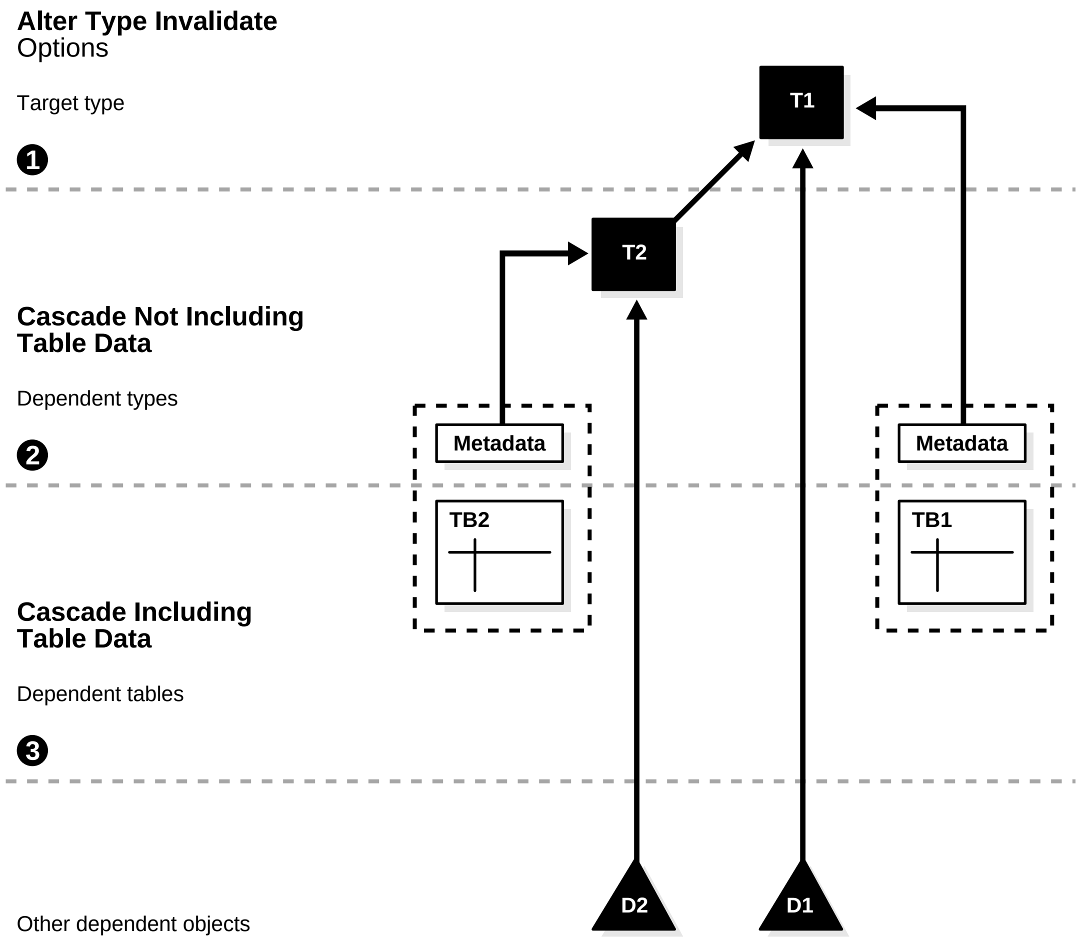
Option	Purpose
ADD *method_spec*	Adds specified method to a type

Option	Purpose
DROP *method_spec*	Drops the method with the specified spec from the target type
ADD ATTRIBUTE	Adds specified attribute to the target type
DROP ATTRIBUTE	Drops specified attribute from the target type
MODIFY ATTRIBUTE	Modifies the type of the specified attribute to increase its length, precision or scale
INVALIDATE	Invalidates all dependent objects. Using this option bypasses all the type and table checks, to save time. Use this option only if you are certain that problems will not be encountered validating dependent types and tables. Table data cannot be accessed again until it is validated; if it cannot be validated, it remains inaccessible.
CASCADE	Propagates the type change to dependent types and tables. The statement aborts if an error is found in dependent types or tables unless the FORCE option is specified. If CASCADE is specified with no other options, then the INCLUDING TABLE DATA option for CASCADE is implied, and Oracle converts all table data to the latest type version.
INCLUDING TABLE DATA	Converts data stored in all user-defined columns to the most recent version of the column's type

Option	Purpose
NOT INCLUDING TABLE DATA	Leaves column data as is, associated with the current type version. If an attribute is dropped from a type referenced by a table, then the corresponding column of the dropped attribute is not removed from the table. Only the metadata of the column is marked unused. If the dropped attribute is stored out-of-line (for example, VARRAY, LOB or nested table attribute) then the out-of-line data is not removed. (Unused columns can be removed afterward by using an ALTER TABLE DROP UNUSED COLUMNS statement.) This option is useful when you have many large tables and may run out of rollback segments if you convert them all in one transaction. This option enables you to convert the data of each dependent table later in a separate transaction (using an ALTER TABLE UPGRADE INCLUDING DATA statement). Specifying this option will speed up the table upgrade because the table's data is left in the format of the old type version. However, selecting data from this table will require converting the images stored in the column to the latest type version. This is likely to affect performance during subsequent SELECT statements.
FORCE	Forces the system to ignore errors from dependent tables and indexes. Errors are logged in a specified exception table so that they can be queried afterward. This option must be used with caution because dependent tables may become inaccessible if some table errors occur.
CONVERT TO SUBSTITUTABLE	For use when altering a type from FINAL to NOT FINAL: Converts data stored in all user-defined columns to the most recent version of the column's type and then marks these existing columns and object tables of the type SUBSTITUTABLE AT ALL LEVELS so that they can store any new subtypes of the type that are created. If the type is altered to NOT FINAL without specifying this option, existing columns and tables of the type are marked NOT SUBSTITUTABLE AT ALL LEVELS, and new subtypes of the type cannot be stored in them. You will be able to store such subtypes only in columns and tables created after the type was altered.

Figure 6–1 graphically summarizes the options for ALTER TYPE INVALIDATE and their effects. In the figure, T1 is a type, and T2 is a dependent type. Also see the notes beneath the figure.

Figure 6–1 ALTER TYPE Options



Notes on the figure:

1. **Invalidate**: All objects following line (1) are marked invalid

2. **Cascade Not Including Table Data**: All objects following line (2) are marked invalid. Metadata of all dependent tables are upgraded to the latest type version, but the table data are not converted.
3. **Cascade Including Table Data**: All objects following line (3) are marked invalid. All dependent tables are upgraded to the latest type version, including the table data.

ALTER TABLE Option for Type Evolution

You can use `ALTER TABLE` to convert table data to the latest version of referenced types. For example, the following statement converts the data in table `benefits` to the latest type version.

```
ALTER TABLE benefits UPGRADE INCLUDING DATA;
```

The `ALTER TABLE statement` contains the following options for converting table data to the latest type version:

Option	Purpose
`UPGRADE`	Converts the metadata of the target table to conform with the latest version of each referenced type. If the target table is already valid, then the table metadata remains unchanged. Specifying `INCLUDING DATA` converts the data in the table to the latest type version format. The default is `INCLUDING DATA`. You can determine which table contains data based on older type version by referring to the `USER_TAB_COLUMNS` view.
`INCLUDING DATA`	Converts data stored in all user-defined columns to the most recent version of the column's type. For each new attribute added to the column's type, a new attribute is added to the data and is initialized to NULL. For each attribute dropped from the referenced type, the corresponding attribute data is removed from each row in the table. All tablespaces containing the table's data must be in read write mode; otherwise, the statement will not succeed.

Option	Purpose
`NOT INCLUDING DATA`	Leaves column data as is and does not update its type version. If an attribute is dropped from a type referenced by the target table, then the corresponding column of the dropped attribute is not removed from the table. Only the metadata of the column is marked unused. If the dropped attribute is stored out-of-line (for example, a varray, LOB or nested table attribute) then the out-of-line data is not removed. To remove the data of those attributes, you can re-submit this statement with `INCLUDING DATA` option specified. Specifying this option will speed up the table upgrade because the table's data is left in the format of the old type version. However, data selected from this table will require converting to the latest type version, so performance may be affected during subsequent `SELECT` statements. This option is useful when there are not enough rollback segments to convert the entire table at once. In this case, you can upgrade the table's metadata first without converting the data, and then issue `UPDATE` statements to set each user-defined column to itself. The `UPDATE` statement will convert the data in the target column to the latest type version. Since this option only requires updating the table's metadata all tablespaces are not required to be on-line in read / write mode for the statement to succeed.
`COLUMN_STORAGE_ CLAUSE`	Specifies the storage for new VARRAY, nested table, or LOB attributes to be added to the table.

User-Defined Constructors

A constructor function is used to create an instance of a user-defined type.

The system implicitly defines a constructor function called the attribute value constructor for all object types with attributes. The attribute value constructor can be used to initialize the attributes of the object. Opaque types do not have attributes and therefore do not have attribute value constructors. However, opaque types can have constructor functions, and many system-defined opaque types, such as XMLTYPE, do have them.

See Also: "Transient and Generic Types" on page 6-37 for information about `SYS.ANYTYPE`, `SYS.ANYDATA`, and `SYS.ANYDATASET`

The attribute-value constructor is convenient to use because it already exists. But user-defined constructors have some important advantages over the attribute-value one with respect to type evolution.

> **See Also:** "Constructor Methods", on page 2-20, for general information about constructors

The Attribute-Value Constructor

With the attribute-value constructor, you must pass the constructor a value for each attribute of the type, to set the attributes of the new object instance to those values.

For example:

```
CREATE TYPE shape AS OBJECT (
    name VARCHAR2(30),
    area NUMBER
);
               -- 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 system-supplied constructor function saves you the trouble of defining your own constructors for a type. However, if you use an attribute-value constructor to create and initialize an instance, you must supply a value for *every* attribute declared in the type. Otherwise the constructor call will fail to compile.

This requirement of an attribute-value constructor can create a problem if you evolve the type later on—by adding an attribute, for example. When you change the attributes of a type, the type's attribute-value constructor changes, too. If you add an attribute, the updated attribute-value constructor expects a value for the new attribute as well as the old ones. As a result, all the attribute-value constructor calls in your existing code, where values for only the old number of attributes are supplied, will fail to compile.

> **See Also:** "Type Evolution" on page 6-8

Advantages of User-Defined Constructors

User-defined constructors avoid the problem with the attribute-value constructor because user-defined constructors do not need to explicitly set a value for every attribute of a type. A user-defined constructor can have any number of arguments, of any type, and these do not need to map directly to type attributes. In your definition of the constructor, you can initialize the attributes to any appropriate values. Any attributes for which you do not supply values are initialized by the system to NULL.

If you evolve a type—for example, by adding an attribute—calls to user-defined constructors for the type do not need to be changed. User-defined constructors, like ordinary methods, are not automatically modified when the type evolves, so the call signature of a user-defined constructor remains the same. You may, however, need to change the definition of the constructor if you do not want the new attribute to be initialized to NULL.

Defining and Implementing User-Defined Constructors

You define user-defined constructors in the type body, like an ordinary method function. You introduce the declaration and the definition with the phrase CONSTRUCTOR FUNCTION; you must also use the clause RETURN SELF AS RESULT.

A constructor for a type must have the same name as the type. The following code 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:

```
CREATE OR REPLACE TYPE shape AS OBJECT
(
    name VARCHAR2(30),
    area NUMBER,
    CONSTRUCTOR FUNCTION shape(name VARCHAR2) RETURN SELF AS RESULT,
    CONSTRUCTOR FUNCTION shape(name VARCHAR2, area NUMBER) RETURN
      SELF AS RESULT
) NOT FINAL;
```

```
CREATE OR REPLACE TYPE BODY shape IS
    CONSTRUCTOR FUNCTION shape(name VARCHAR2) RETURN SELF AS RESULT IS
    BEGIN
        SELF.name := name;
        SELF.area := 0;
        return;
    END;
    CONSTRUCTOR FUNCTION shape(name VARCHAR2, area NUMBER) RETURN shape
    SELF AS RESULT IS
    BEGIN
        SELF.name := name;
        SELF.area := area;
        return;
    END;
END;
```

A user-defined constructor has an implicit first parameter SELF. Specifying this parameter in the declaration of a user-defined constructor is optional. If you do specify it, its mode must be declared to be IN OUT.

The required clause RETURN SELF AS RESULT ensures that the most specific type of the instance being returned is the same as the most specific type of the SELF argument. For example, if the most specific type of the SELF argument on a call to the shape constructor is shape, then this clause ensures that the shape constructor returns an instance of shape (not an instance of a subtype of shape).

When a constructor function is called, the system initializes the attributes of the SELF argument to NULL. Names of attributes subsequently initialized in the function body may be qualified with SELF, as shown in the preceding example, to distinguish them from the names of the arguments of the constructor function, if these are the same. If the argument names are different, no such qualification is necessary. For example:

```
SELF.name := name;
```

or:

```
name := p1;
```

The function body must include an explicit return; as shown. The **return** keyword must not be followed by a return expression. The system automatically returns the newly constructed SELF instance.

A user-defined constructor may be implemented in PL/SQL, C, or Java.

Overloading and Overriding Constructors

Like other type methods, user-defined constructors can be overloaded.

User-defined constructors are not inherited, so they cannot be overridden. However, a user-defined constructor does hide, and thus supersede, an attribute-value constructor if the signature of the user-defined constructor exactly matches the signature of the attribute-value constructor. For the signatures to match, the names and types of the parameters (after the implicit SELF parameter) of the user-defined constructor must be the same as the names and types of the type's attributes. The mode of each of the user-defined constructor's parameters (after the implicit SELF parameter) must be IN.

If an attribute-value constructor is not hidden by a user-defined constructor having the same name and signature, the attribute-value constructor can still be called.

Note that, if you evolve a type—for example, by adding an attribute—the signature of the type's attribute-value constructor changes accordingly. This can cause a formerly hidden attribute-value constructor to become usable again.

Calling User-Defined Constructors

A user-defined constructor is called like any other function. You can use a user-defined constructor anywhere you can use an ordinary function.

The SELF argument is passed in implicitly and may not be passed in explicitly. In other words, usages like the following are not allowed:

```
NEW constructor(instance, argument_list)
```

A user-defined constructor cannot occur in the DEFAULT clause of a CREATE or ALTER TABLE statement, but an attribute-value constructor can. The arguments to the attribute-value constructor must not contain references to PL/SQL functions or to other columns, including the pseudocolumns LEVEL, PRIOR, and ROWNUM, or to date constants that are not fully specified. The same is true for check constraint expressions: an attribute-value constructor can be used as part of check constraint expressions while creating or altering a table, but a user-defined constructor cannot.

Parentheses are required in SQL even for constructor calls that have no arguments. For example:

```
SELECT NEW type_name() FROM dual;
```

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.

SQL Examples

```
CREATE OR REPLACE TYPE rectangle UNDER shape
(
    length NUMBER,
    breadth NUMBER,
    CONSTRUCTOR FUNCTION rectangle(
        name VARCHAR2, length NUMBER, breadth NUMBER
    ) RETURN SELF as RESULT
);

CREATE OR REPLACE TYPE BODY rectangle IS
    CONSTRUCTOR FUNCTION rectangle(
        name VARCHAR2, length NUMBER, breadth NUMBER
    ) RETURN  SELF AS RESULT IS
    BEGIN
        SELF.name := name;
        SELF.area := length*breadth;
        SELF.length := length;
        SELF.breadth := breadth;
        RETURN ;
    END;
END;

INSERT INTO rectangle_table
  SELECT NEW rectangle(s.name, s.length, s.breadth) FROM square_table s;

UPDATE rectangle_table SET rec = NEW rectangle('Quad', 12, 5) WHERE rec IS NULL;
```

PL/SQL Example

```
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.

For example:

```
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',
  zip_code number EXTERNAL NAME 'zipCode',
  CONSTRUCTOR FUNCTION address (full_address VARCHAR)
    EXTERNAL NAME 'address (java.lang.String)',
);
```

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.

LNOCI Tips and Techniques for Objects

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

Initializing an OCI Program in Object Mode

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

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

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

Creating a New Object

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

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

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

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

Updating an Object

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

Deleting an Object

You can delete an object by calling the *LNOCIObjectMarkDelete()* function or the *LNOCIObjectMarkDeleteByRef()* function.

Controlling Object Cache Size

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

- `LNOCI_ATTR_CACHE_MAX_SIZE` controls the maximum cache size
- `LNOCI_ATTR_CACHE_OPT_SIZE` controls the optimal cache size

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

Retrieving Objects into the Client Cache (Pinning)

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

The following are examples of *LNOCIObjectPin()* and *LNOCIObjectUnpin()* calls:

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

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

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

Specifying which Version of an Object to Retrieve

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

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

Specifying How Long to Keep the Object Pinned

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

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

Specifying Whether to Lock the Object on the Server

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

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

How to Choose the Locking Technique

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

If objects are updated frequently, you can use the *pessimistic locking scheme*. This scheme presumes that contention for update access is frequent. Objects are locked before the object in the cache is modified, ensuring that no other user can modify the object until the transaction owning the lock performs a commit or rollback. The object can be locked at the time of pin by choosing the appropriate locking options. An object that was not locked at the time of pin also can be locked by the function

LNOCIObjectLock(). The locking function LNOCIObjectLockNoWait() does not wait to acquire the lock if another user holds a lock on the object.

If objects are updated infrequently, you can use the *optimistic locking scheme*. This scheme presumes that contention for update access is rare. Objects are fetched and modified in the cache without acquiring a lock. A lock is acquired only when the object is flushed to the server. Optimistic locking allows for a higher degree of concurrent access than pessimistic locking. To use optimistic locking most effectively, the Oracle object cache detects if an object is changed by any other user since it was fetched into the cache. By turning on the *object change detection mode*, object modifications are made persistent only if the object has not been changed by any other user since it was fetched into the cache. This mode is activated by setting LNOCI_OBJECT_DETECTCHANGE attribute of the environment handle using the LNOCIAttrSet() function.

Flushing an Object from the Object Cache

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

Pre-Fetching Related Objects (Complex Object Retrieval)

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

The application specifies the set of related objects to be pre-fetched by providing the following information:

- A REF to the root object
- One or more pairs of object type and depth information to specify the content and boundary of objects to be pre-fetched. The type information indicates

which REF attributes should be de-referenced and which resulting object should be pre-fetched. The depth defines the boundary of objects pre-fetched. The depth level is the shortest number of references that need to be traversed from the root object to a related object.

For example, consider a purchase order system with the following properties:

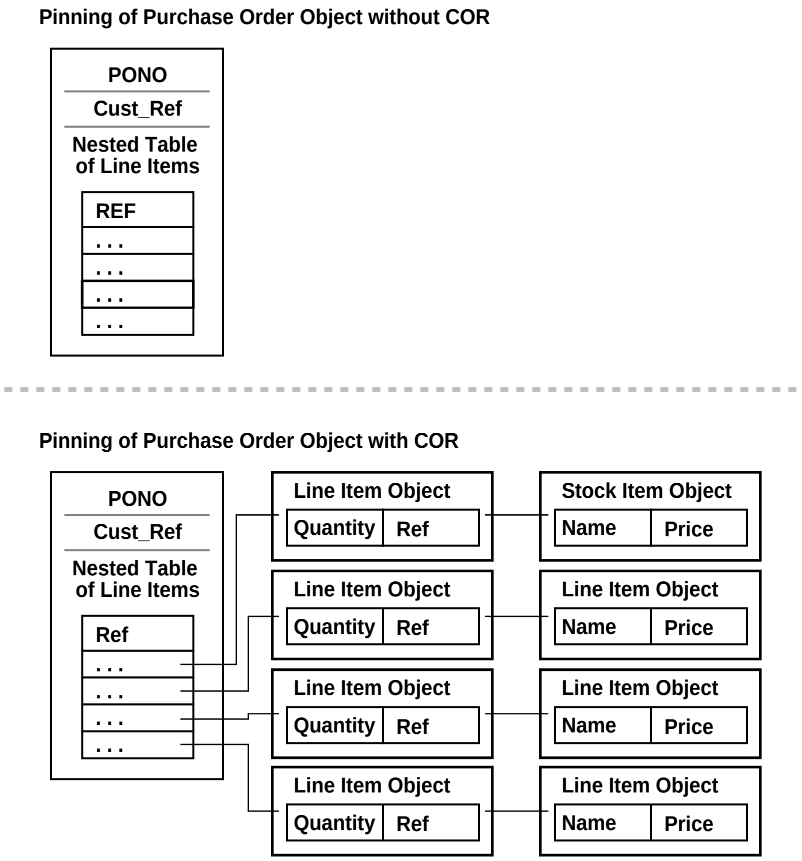
- Each purchase order object includes a purchase order number, a REF to a customer object, and a collection of REFs that point to line item objects.
- Each customer object includes information about the customer, such as the customer's name and address.
- Each line item object includes a reference to a stock item and the quantity of the order.
- Each stock item object includes the name of the item, its price, and other information about the item.

Suppose you want to calculate the total cost of a particular purchase order. To maximize efficiency, you want to fetch only the objects necessary for the calculation from the server to the client cache, and you want to fetch these objects with the least number of calls to the server possible.

If you do not use COR, your application must make several server calls to retrieve all of the necessary objects. However, if you use COR, you can specify the objects that you want to retrieve and exclude other objects that are not required. To calculate the total cost of a purchase order, you need the purchase order object, the related line item objects, and the related stock item objects, but you do not need the customer objects.

Therefore, as shown in Figure 6–2, COR enables you to retrieve the required information for the calculation in the most efficient way possible. When pinning the purchase order object without COR, only that object is retrieved. When pinning it with COR, the purchase order and the related line item objects and stock item objects are retrieved. However, the related customer object is not retrieved because it is not required for the calculation.

Figure 6–2 Difference Between Retrieving an Object Without COR and With COR


Pinning of Purchase Order Object without COR
PONO
Cust_Ref
Nested Table of Line Items
REF
. . .
Pinning of Purchase Order Object with COR
PONO
Cust_Ref
Nested Table of Line Items
Ref
. . .
Line Item Object
Quantity
Ref
Stock Item Object
Name
Price

Demonstration of OCI and Oracle Objects

For a demonstration of how to use OCI with Oracle objects, see the `cdemocor1.c` file in `$ORACLE_HOME/rdbms/demo`.

Using the OCI Object Cache with View Objects

We can pin and navigate objects synthesized from object views in the OCI Object Cache similar to the way we do this with object tables. We can also create new view objects, update them, delete them and flush them from the cache. The flush performs the appropriate DML on the view (such as insert for newly created objects and updates for any attribute changes). This fires any `INSTEAD-OF` triggers on the view and stores the object persistently.

There is a minor difference between the two approaches with regard to getting the reference to a newly created instance in the object cache.

In the case of object views with primary key based reference, the attributes that make up the identifier for the object need to be initialized before the `LNOCIObjectGetObjectRef` call can be called on the object to get the object reference. For example, to create a new object in the OCI Object cache for the purchase order object, we need to take the following steps:

```
.. /* Initialize all the settings including creating a connection, getting a
      environment handle and so forth. We do not check for error conditions to
make
      the example eaiser to read. */
LNOCIType *purchaseOrder_tdo = (OCIType *) 0; /* This is the type object for the
                                                purchase order */
dvoid * purchaseOrder_viewobj = (dvoid *) 0;   /* This is the view object */

/* The purchaseOrder struct is a structure that is defined to have the same
attributes as that of  PurchaseOrder_objtyp type. This can be created by the
user or generated automatically using  the OTT generator. */
purchaseOrder_struct *purchaseOrder_obj;

/* This is the null structure corresponding to the purchase order object's
attributes */
purchaseOrder_nullstruct *purchaseOrder_nullobj;

/* This is the variable containing the purchase order number that we need to
create */
int PONo = 1003;

/* This is the reference to the purchase order object */
LNOCIRef *purchaseOrder_ref = (OCIRef *)0;

/* Pin the object type first */
LNOCITypeByName( envhp, errhp, svchp,
                              (CONST text *) "",  (ub4) strlen( "") ,
                              (CONST text *) "PURCHASEORDER_OBJTYP" ,
                              (ub4) strlen("PURCHASEORDER_OBJTYP"),
                              (CONST char *) 0, (ub4)0,
                              OCI_DURATION_SESSION, OCI_TYPEGET_ALL,
&purchaseOrder_tdo);
```

```
/* Pin the table object - in this case it is the purchase order view */
LNOCIObjectPinObjectTable(envhp, errhp, svchp, (CONST text *) “”,
                               (ub4) strlen( “” ),
                               (CONST text *) “PURCHASEORDER_OBJVIEW”,
                               (ub4 ) strlen(“PURCHASEORDER_OBJVIEW”),
                               (CONST OCIRef *) NULL,
                                OCI_DURATION_SESSION,
                                &purchaseOrder_viewobj);

/* Now create a new object in the cache. This is a purchase order object */
LNOCIObjectNew(envhp, errhp, svchp, OCI_TYPECODE_OBJECT, purchaseOrder_tdo,
                     purchaseOrder_viewobj, OCI_DURATION_DEFAULT, FALSE,
                     (dvoid **) *purchaseOrder_obj);

/* Now we can initialize this object, and use it as a regular object. But before
getting the reference to this object we need to initialize the PONo attribute of
the object which makes up its object identifier in the view */

/* Initialize the null identifiers */
LNOCIObjectGetInd( envhp, errhp, purchaseOrder_obj, purchaseOrder_nullobj);

purchaseOrder_nullobj->purchaseOrder = OCI_IND_NOTNULL;
purchaseOrder_nullobj->PONo = OCI_IND_NOTNULL;

/* This sets the PONo attribute */
LNOCINumberFromInt( errhp, (CONST dvoid *) &PoNo, sizeof(PoNo), OCI_NUMBER_
SIGNED,
                                 &( purchaseOrder_obj->PONo));

/* Create an object reference */
LNOCIObjectNew( envhp, errhp, svchp, OCI_TYPECODE_REF, (OCIType *) 0,
                          (dvoid *) 0, (dvoid *) 0, OCI_DURATION_DEFAULT, TRUE,
                          (dvoid **) &purchaseOrder_ref);

/* Now get the reference to the newly created object */
LNOCIObjectGetObjectRef(envhp, errhp, (dvoid *) purchaseOrder_obj,
purchaseOrder_ref);

/* This reference may be used in the rest of the program ….. */
…
/* We can flush the changes to the disk and the newly instantiated purchase
order object in the object cache will become permanent. In the case of the
purchase order object, the insert will fire the INSTEAD-OF trigger defined over
the purchase order view to do the actual processing */
```

```
LNOCICacheFlush( envhp, errhp, svchp, (dvoid *) 0, 0, (OCIRef **) 0);
…
```

Transient and Generic Types

Oracle has three special SQL datatypes 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** (that is, unnamed) types, including anonymous collection types.

The three SQL types are implemented as **opaque types**. In other words, the internal structure of these types is not known to the database: their data can be queried only by implementing functions (typically 3GL routines) for the purpose. Oracle provides both an OCI and a PL/SQL API for implementing such functions.

The three generic SQL types are:

Type	Description
SYS.ANYTYPE	A type description type. A SYS.ANYTYPE can contain a type description of any SQL type, named or unnamed, including object types and collection types. An ANYTYPE can contain a type description of a persistent type, but an ANYTYPE itself is **transient**: in other words, the value in an ANYTYPE itself is not automatically stored in the database. To create a persistent type, use a CREATE TYPE statement from SQL.
SYS.ANYDATA	A **self-describing** data instance type. A SYS.ANYDATA contains an instance of a given type, with data, plus a description of the type. In this sense, a SYS.ANYDATA is self-describing. An ANYDATA can be persistently stored in the database.
SYS.ANYDATASET	A self-describing data set type. A SYS.ANYDATASET type contains a description of a given type plus a set of data instances of that type. An ANYDATASET can be persistently stored in the database.

Each of these three types can be used with any built-in type native to the database as well as with object types and collection types, both named and unnamed. The types provide a generic way to work dynamically with type descriptions, lone instances, and sets of instances of other types. Using the APIs, you can create a transient ANYTYPE description of any kind of type. Similarly, you can create or convert (cast) a data value of any SQL type to an ANYDATA and can convert an ANYDATA (back) to a SQL type. And similarly again with sets of values and ANYDATASET.

The generic types simplify working with stored procedures. You can use the generic types to encapsulate descriptions and data of standard types and pass the encapsulated information into parameters of the generic types. In the body of the procedure, you can detail how to handle the encapsulated data and type descriptions of whatever type.

You can also store encapsulated data of a variety of underlying types in one table column of type ANYDATA or ANYDATASET. For example, you can use ANYDATA with Advanced Queuing to model queues of heterogenous types of data. You can query the data of the underlying datatypes like any other data.

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:

- `LNOCIType`, corresponding to `SYS.ANYTYPE`
- `LNOCIAnyData`, corresponding to `SYS.ANYDATA`
- `LNOCIAnyDataSet`, corresponding to `SYS.ANYDATASET`

> **See Also:** *Oracle Call Interface Programmer's Guide* for the `LNOCIType`, `LNOCIAnyData`, and `LNOCIAnyDataSet` APIs and details on how to use them. See *Oracle9i Supplied PL/SQL Packages and Types Reference* for information about the interfaces to the ANYTYPE, ANYDATA, and ANYDATASET types and about the DBMS_ TYPES package, which defines constants for built-in and user-defined types, for use with ANYTYPE, ANYDATA, and ANYDATASET.

User-Defined Aggregate Functions

Oracle provides a number of pre-defined aggregate functions such as MAX, MIN, SUM for performing operations on a set of records. These pre-defined aggregate functions can be used only with scalar data. However, you can create your own custom implementations of these functions, or define entirely new aggregate functions, to use with complex data—for example, with multimedia data stored using object types, opaque types, and LOBs.

User-defined aggregate functions are used in SQL DML statements just like Oracle's own built-in aggregates. Once such functions are registered with the server, Oracle simply invokes the aggregation routines that you supplied instead of the native ones.

User-defined aggregates can be used with scalar data as well. For example, it may be worthwhile to implement special aggregate functions for working with complex statistical data associated with financial or scientific applications.

User-defined aggregates are a feature of the Extensibility Framework. You implement them using `ODCIAggregate` interface routines.

> **See Also:** *Oracle9i Data Cartridge Developer's Guide* for information on using the `ODCIAggregate` interface routines to implement user-defined aggregate functions

Partitioning Tables that Contain Oracle Objects

Partitioning addresses the key problem of supporting very large tables and indexes by allowing you to decompose them into smaller and more manageable pieces called partitions. Oracle extends partitioning capabilities by letting you partition tables that contain objects, `REF`s, varrays, and nested tables. Varrays stored in `LOB`s are equipartitioned in a way similar to `LOB`s.

The following example partitions the purchase order table along zip codes (`ToZip`), which is an attribute of the `ShipToAddr` embedded column object. For the purposes of this example, the `LineItemList` nested table was made a varray to illustrate storage for the partitioned varray.

> **Restriction:** Nested tables are allowed in tables that are partitioned; however, the storage table associated with the nested table is not partitioned.

Assuming that the `LineItemList` is defined as a varray:

```
CREATE TYPE LineItemList_vartyp as varray(10000) of LineItem_objtyp;

CREATE TYPE PurchaseOrder_typ AS OBJECT (
      PONo               NUMBER,
      Cust_ref           REF Customer_objtyp,
      OrderDate          DATE,
      ShipDate           DATE,
      OrderForm          BLOB,
      LineItemList        LineItemList_vartyp,
      ShipToAddr          Address_objtyp,

   MAP MEMBER FUNCTION
      ret_value RETURN NUMBER,

   MEMBER FUNCTION
      total_value RETURN NUMBER
  );

CREATE TABLE PurchaseOrders_tab of PurchaseOrder_typ
    LOB (OrderForm) store as (nocache logging)
    PARTITION BY RANGE (ShipToAddr.zip)
      (PARTITION PurOrderZone1_part
         VALUES LESS THAN ('59999')
         LOB (OrderForm) store as (
               storage (INITIAL 10 MINEXTENTS 10 MAXEXTENTS 100))
         VARRAY LineItemList store as LOB (
               storage (INITIAL 10 MINEXTENTS 10 MAXEXTENTS 100)),
    PARTITION PurOrderZone6_part
         VALUES LESS THAN ('79999')
         LOB (OrderForm) store as (
               storage (INITIAL 10 MINEXTENTS 10 MAXEXTENTS 100))
         VARRAY LineItemList store as LOB (
               storage (INITIAL 10 MINEXTENTS 10 MAXEXTENTS 100)),
    PARTITION PurOrderZoneO_part
       VALUES LESS THAN ('99999')
        LOB (OrderForm) store as (
               storage (INITIAL 10 MINEXTENTS 10 MAXEXTENTS 100))
        VARRAY LineItemList store as LOB (
               storage (INITIAL 10 MINEXTENTS 10 MAXEXTENTS 100)));
```

Parallel Query with Object Views

Parallel query is supported on the objects synthesized in views.

To execute queries involving joins and sorts (using the ORDER BY, GROUP BY, and SET operations) in parallel, a MAP function is needed. In the absence of a MAP function, the query automatically becomes serial.

Parallel queries on nested table columns are not supported. Even in the presence of parallel hints or parallel attributes for the view, the query will be serial if it involves the nested table column.

Parallel DML is not supported on views with `INSTEAD-OF` trigger. However, the individual statements within the trigger may be parallelized.

How Locators Improve the Performance of Nested Tables

Collection types do not map directly to a native type or structure in languages such as C++ and Java. An application using those languages must access the contents of a collection through Oracle interfaces, such as OCI.

Generally, when the client accesses a nested table explicitly or implicitly (by fetching the containing object), Oracle returns the entire collection value to the client process. For performance reasons, a client may wish to delay or avoid retrieving the entire contents of the collection. Oracle handles this case for you by using a locator instead of the real nested table value. When you really access the contents of the collection, they are automatically transferred to the client.

A nested table locator is like a handle to the collection value. It attempts to preserve the value or copy semantics of the nested table by containing the database snapshot as of its time of retrieval. The snapshot helps the database retrieve the correct instantiation of the nested table value at a later time when the collection elements are fetched using the locator. The locator is scoped to a session and cannot be used across sessions. Since database snapshots are used, it is possible to get a "snapshot too old" error if there is a high update rate on the nested table. Unlike a LOB locator, the nested table locator is truly a locator and cannot be used to modify the collection instance.

7

Frequently Asked Questions About Using Oracle Objects

Here are some questions and answers that new users often have about Oracle's object-relational features:

You can use this chapter as introductory information, or refer here if you still have questions after reading the rest of the book.

General Questions about Oracle Objects

Are the object-relational features a separate option?

Not anymore. As of Version 8.1, they are part of the base server product.

What are the design goals of Oracle9*i* Object-Relational & Extensibility technologies?

The design goals of Oracle9*i* Objects and Extensibility technologies are to:

- Provide users with the ability to model their business objects in the database by enhancing the type system to support user-defined types. These types are meant to closely model application objects and are treated as built-in types, such as number and character, by the database server.
- Provide an infrastructure to facilitate object-based access to data stored in an Oracle database and minimize the potential mismatch between the data model used in an application and the data model supported by a database.
- Provide built-in support for new data types needed in multimedia, financial and spatial applications.
- Provide a framework for database extensibility so that new multimedia and complex data types can be supported and managed natively in the database. This framework provides the infrastructure needed to allow extensions of the data server by third parties, using data cartridges.

This book talks about the object-relational technologies. For details about extensibility, see *Oracle9i Data Cartridge Developer's Guide*.

Object Types

What is structured data?

The SQL 92 standard defines the 19 atomic datatypes that are used in most database programming. We refer to these kinds of data as "simple structured".

Oracle Objects introduces the ideas of REFs and collections. We refer to these kinds of data as "complex structured".

LOBs provide another way to store information. We refer to them as "unstructured".

Where are the user-defined types, user-defined functions, and abstract data types?

The Oracle equivalent of a user-defined type or an abstract data type is an object type.

The Oracle equivalent of a user-defined function is an object type method.

We use these terms because their semantics are different from the common industry usage. For example, an Oracle object can be null, while an object of an abstract data type cannot.

What is an object type?

Oracle9*i* supports a form of user-defined data types called object types. Object types are abstractions of real-world entities. An object type is a schema object with the following components:

- A name, which identifies the object type uniquely within a schema
- Attributes, which model the structure and state of the real-world entity
- Methods, which implement the behavior of the real-world entity

Why are object types useful?

An object type is similar to the class mechanism supported by C++ and Java. Object reusability provides faster and more efficient database application development. Object support makes it easier to model complex, real-world business entities and logic. By supporting object types natively in the database, Oracle relieves application developers from having to write a mapping layer between client-side objects and database objects. Object abstraction and encapsulation also make applications easier to understand and maintain.

How is object data stored and managed in Oracle9*i*?

Objects are managed natively by the data server. Object types can be used as the type of a column (column objects) or as type of each row in an object table (row objects). When used as column objects, object types serve as classical relational domains. Each row object has a unique identity, called an object identifier (OID).

Objects are first-class citizens and are fully integrated with the database components. They can be indexed and partitioned. For example, queries involving objects can be parallelized and are optimized by the cost-based optimizer using statistics.

By building on the proven foundation of the Oracle data server, objects are managed with the same reliability, availability, and scalability as relational data.

Is inheritance supported in Oracle9*i*?

Oracle supports single inheritance of user-defined SQL types. You can derive one or more subtypes from a single supertype. Subtypes can themselves be further specialized, enabling you to construct type hierarchies having any number of levels. Keywords are provided to let you control whether a given type can be subtyped or instantiated.

Oracle also provides support for client-side inheritance through its C++ and Java mappings. For C++, use the Object Modelling Option of Oracle Designer to produce DDL and C++ code based on diagrams in the Universal Modelling Language (UML). For Java, use the "custom datum" feature of the Oracle JDBC driver.

Server-side method inheritance is provided in Java by the Oracle9*i* Java VM.

Object Methods

What language can I use to write my object methods?

Methods can be implemented in PL/SQL, Java, C or C++. C & C++ support is provided through the external procedure functionality in Oracle, whereas PL/SQL and Java methods run within the address space of the server. De-coupling of the specification of a method in SQL from its implementation provides a uniform way to invoke methods on object types, even though these object types can be implemented in various programming languages. Oracle provides a safe and secure environment for invoking PL/SQL methods, Java methods, and external C procedures from the server. Programming errors in user methods will not cause the server to fail or corrupt the database, thus ensuring the reliability and availability of the server in a mission critical environment.

How do I decide between using PL/SQL and Java for my object methods?

In Oracle, PL/SQL and Java can be used interchangeably as a server programming language. PL/SQL is a seamless extension of SQL and is the language of choice for doing SQL intensive operations in the database. Java is the emerging language of choice for writing portable applications that run in multiple tiers, including the database server.

When should I use external procedures?

External procedures are typically used for computationally intensive operations that are best written in a low-level language such as C. External procedures are also useful for invoking routines in some existing libraries that cannot be easily rewritten in Java or PL/SQL to run in the data server.

The IPC (inter-process communication) overhead of invoking an external procedure is an order of magnitude higher than that of invoking PL/SQL or Java procedure. However, the overhead of invoking an external procedure become insignificant if the computation done in the external procedure is complex and is in the order of tens of thousands of instructions.

What are definer and invoker rights?

The distinction between definer and invoker rights applies to more than just objects. You may find invoker rights especially useful for object-oriented programs because they typically contain reusable modules.

An object method can be executed with the privileges of its owner (definer rights) or with the privileges of the current user (invoker rights), based on the method definition. Invoker rights are useful for writing reusable objects because users of these objects do not have to grant access privileges to their tables to the implementor of the objects. Definer rights are useful when the as part of the object implementation, the object methods need to access some metadata maintained by the object implementor. Methods that access the metadata are executed using the definer rights so that the object implementor does not have to expose the proprietary metadata to the users.

Object References

What is an object reference?

An object reference (REF) uniquely identify a row object stored in an object table or an object constructed from an object view. Typically, a REF value is comprised of the object’s unique identifier, the unique identifier associated with the object table, and the ROWID of the row in the object table in which the object is stored. The optional ROWID is used as a hint to provide fast access to the object.

When should I use object references? How are they different from foreign keys?

Object references, like foreign keys, are useful in modeling complex relationships. Object references are more flexible than foreign keys for modeling complex relationships because:

- Object references are strongly typed and this provides better compile-time type checking
- One-to-many relationships can be modeled using a collection of object references
- Application can easily navigate and retrieve objects using object references without having to construct SQL statements
- REF navigation in SQL avoids the need to do complicated multitable joins
- Object references allow applications to retrieve objects connected by REFs in a single request to the server

Can I construct object references based on primary keys?

Yes, object references can be constructed based on foreign keys to reference objects in:

- Object views: When constructing objects from relational tables using an object view, the OIDs of the constructed objects are typically based on the primary keys on the underlying relational tables.
- Object tables with primary key-based OIDs: When defining an object table, Oracle provides the option of specifying the primary keys as the OIDs of the row objects instead of using the system generated OIDs.

What is a scoped REF and when should I use it?

In general, a column may contain references to objects of a particular declared type regardless of the object table(s) in which the objects are stored. However, a REF type column may be scoped (constrained) to only contain references to objects from a specified object table. One should use scoped REFs whenever possible because scoped REFs are smaller in size than regular REFs on disk because the system does not have to store the table identifier with the scoped REFs. Also, queries containing navigation of scoped REFs can be optimized into joins when appropriate.

Can I manipulate objects using object references in PL/SQL and Java?

Yes, both PL/SQL and Java support object references. In PL/SQL, an object can be retrieved and updated using the UTL_REF package given its object references. In Java, object references are mapped to reference classes with get and set methods to retrieve and update the objects.

Collections

What kinds of collections are supported by Oracle9*i*?

Oracle9*i* supports two types of collections: variable-length arrays (varrays) and nested tables. Attributes of object types and columns of tables can be of collection types, and so can collections themselves. By using varrays and nested tables, applications can model one-to-many and many-to-many relationships natively in their database schema.

Do Oracle Objects support collections within collections?

Yes. A varray can contain another varray or a nested table, and a nested table can contain another nested table or a varray. Similarly, you can have a collection of an object type that has an attribute of a collection type. Such multilevel collections can be nested to any number of levels.

How do I decide between using varrays and nested tables for modeling collections?

Varrays are useful when you need to maintain ordering of your collection elements. Varrays are very efficient when you always manipulate the entire collection as a unit, and that you don't require querying on individual elements of the collections. Varrays are stored inline with the containing row if it is small and automatically stored as a LOBs by the system when its size is beyond a certain threshold.

Nested tables are useful when there is no ordering among the collection elements and that efficient querying on individual elements are important. Elements of a collection type column are stored in a separate table, similar to parent-child tables in a relational schema.

What is a collection locator?

Collection locators allow applications to retrieve large collections without materializing the collections in memory. This allows for efficient transfer of large collections across interfaces. A collection will be transparently materialized when the application first accesses its elements. Also, applications can query and retrieve subsets of the collection using its locator.

The specification of retrieval of collection locators is done in CREATE and ALTER TABLE DDL. Since access to a collection is encapsulated, applications will use the same interface to retrieve a nested table specified to be returned as a locator as one specified to be returned as a value.

What is collection unnesting?

Collection unnesting allows applications to efficiently query over a set of collections in some specified rows, similar to query on the child rows in a relational schema for some specified parent rows. Collection unnesting allows applications the flexibility to view one-to-many relationships in the collection form or in the flat parent-child form.

Object Views

What are the differences between object views and object tables?

Like the similarity between relational views and tables, an object view has properties similar to an object table:

- It contains objects in rows. The columns of the view map to top-level attributes of the object type.
- Each object has an identifier associated with it. The identifier is specified by the view definer; in most cases, the primary key of the base table serves as the identifier.

Are object views updatable?

It is easy to update an object view where every attribute maps back to a real column in a table. For views that derive some attributes by more complex techniques, such as CAST-MULTISET, INSTEAD-OF triggers can be used to do the updates. When such a view is updated (or inserted into or deleted from), rather than attempting to implicitly modify any base tables, the system simply invokes the INSTEAD-OF trigger specified for the view. You can encapsulate whatever update semantics you want in the trigger body.

Object Cache

Why do we need the object cache?

The object cache gives applications the following benefits:

- Transparent mapping of database objects to host language objects in memory.
- Transparent, efficient memory management for persistent objects. Applications do not have to worry about allocation of memory for accessing database objects.
- Transactional semantics for client-side objects. Modified persistent objects in the object cache can be propagated (flushed) to the database in a single round-trip between the client and the server.
- Navigational object access. The object cache allows for navigational style object access. Using OCI's object functions, objects can be fetched into the object cache by pinning object REFs. Navigational object access may be more suitable when operating on a graph of objects that are inter-connected through object REFs.
- Complex object retrieval. That is, a single request to fetch an object from the server can be used to retrieve other objects, which are connected through REFs to the object being fetched, in a single round-trip between the client and the server.

Does the object cache support object locking?

The object cache supports both a pessimistic locking scheme and an optimistic locking scheme.

- In the pessimistic locking scheme, objects are locked up-front in the server prior to modifying the object in the cache. This ensures no other user can modify the object until the transaction owning the lock commits/rollbacks.
- In the optimistic locking scheme, an object is fetched and modified in the cache without acquiring a lock. The lock is acquired only when the object is flushed to the server. Optimistic locking allows for a higher degree of concurrent access than pessimistic locking. To use optimistic locking effectively, the object cache provides the ability for detecting if an object was changed by any other user since it was fetched into the cache. By turning on the "object change detection mode", object modifications will be made persistent if the nobody else has changed the object since it was fetched into the cache.

Large Objects (LOBs)

How can I manage large objects using Oracle?

Support for multimedia data types like text, images, audio, and video requires robust support for binary and character data. The data in these domains tends to be large and requires direct access to different pieces of the binary data. To address this need, Oracle provides significantly improved support for large-scale binary and character data. It introduces the Large Object type (LOB) which can be used to store large, domain-specific data from various domains, including images, audio files, text and spatial data.

Oracle supports three kinds of large data objects: binary, character-based, and file-based. In addition to providing the ability to create LOBs, Oracle server provides several other improvements in managing binary data. These improvements can be summarized as follows:

- Support for defining more than one LOB column in a table
- Random, piece-wise access to LOB data
- Support for transferring LOB data as a single stream
- Support for disabling logging and caching for LOB data
- Support for transparently moving LOBs from "in-line" row storage to "out-of-line" storage in another segment or even another tablespace

For more information about LOBs, see *Oracle9i Application Developer's Guide - Large Objects (LOBs)*.

User-Defined Operators

What is a user-defined operator?

Oracle allows developers of object-oriented applications to extend the list of built-in relational operators (for example, +, -, /, *, LIKE) with domain specific operators (for example, Contains, Within_Distance, Similar) called user-defined operators. A user-defined operator can be used anywhere built-in operators can be used, for example, in the select list or the where clause. Similar to built-in operators, a user-defined operator may support arguments of different types, and that it may be evaluated using an index.

See Also: For more information about user-defined operators, see:

- `CREATE OPERATOR` in the *Oracle9i SQL Reference*
- *Oracle9i Data Cartridge Developer's Guide*

Why are user-defined operators useful?

Similar to built-in operators, user-defined operators allow efficient content-based querying and sorting on object data. For example, to find a resume containing certain qualifications, one may specify the Contains operator as part of the SQL where clause. The optimizer may choose to use a Text index on the resume column to perform the query efficiently, similar to using a B-tree index to evaluate a relational operator.

8

Design Considerations for Oracle Objects

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

This chapter covers the following topics:

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

See Also:

- *Oracle9i Database Concepts* for conceptual information about Oracle objects
- *Oracle9i SQL Reference* for information about the SQL syntax for using Oracle objects.

Representing Objects as Columns or Rows

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

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

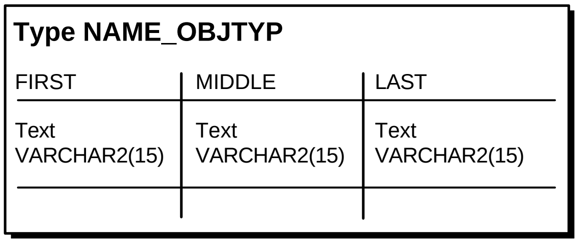
Column Object Storage

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

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

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

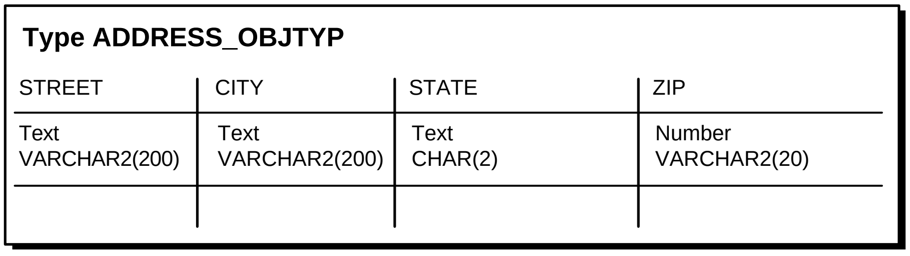
Figure 8–1 Object Relational Representation for the name_objtyp Type



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

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

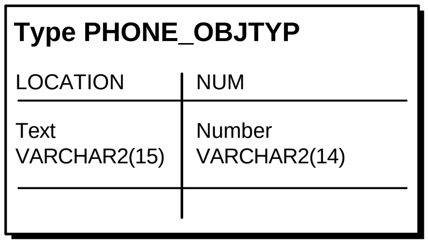
Figure 8–2 Object Relational Representation of the address_objtyp Type



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

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

Figure 8–3 Object Relational Representation of the phone_objtyp Type



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

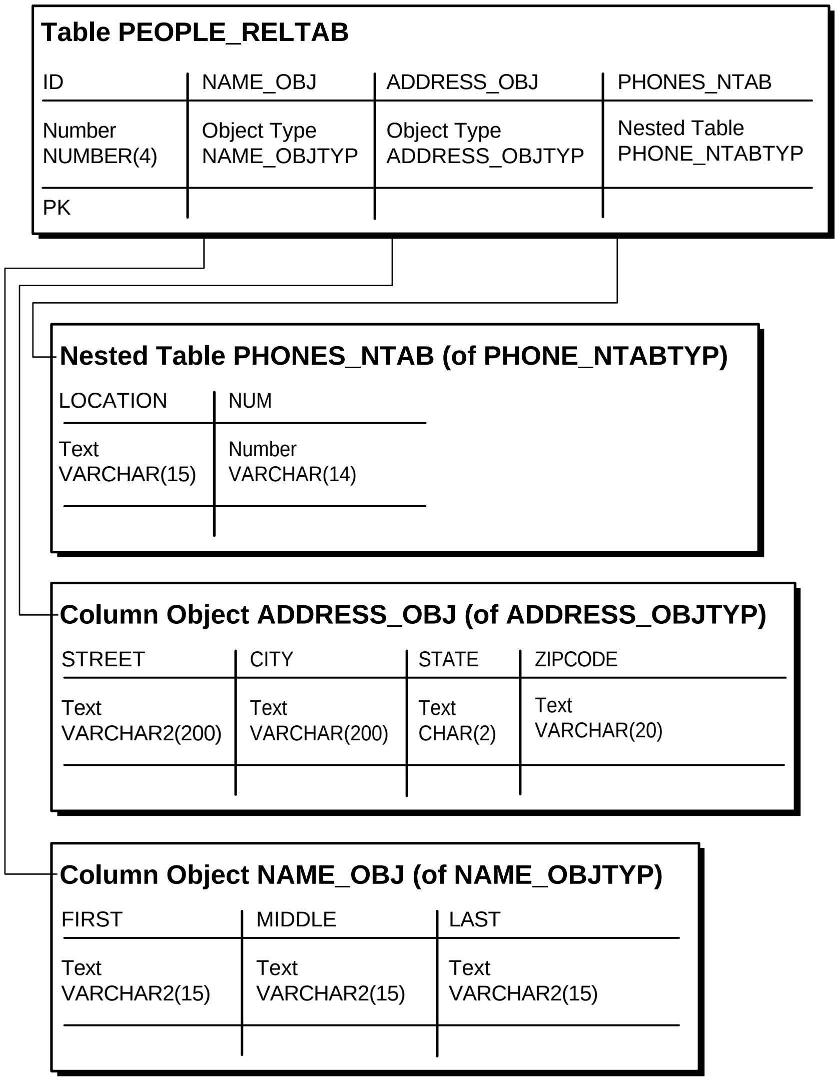
```
CREATE TYPE phone_ntabtyp AS TABLE OF phone_objtyp;
```

See Also: "Nested Tables" on page 8-16 for more information about nested tables.

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

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

Figure 8–4 Representation of the people_reltab Relational Table



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

Note: The `people_reltab` table and its columns and related types are used in examples throughout this chapter.

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

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

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

```
CEIL(n/8) + 6
```

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

```
CEIL(4/8) + 6 = 7
```

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

Therefore, the storage overhead and performance of manipulating a column object is similar to that of the equivalent set of scalar columns. The storage for collection attributes are described in the "Viewing Object Data in Relational Form with Unnesting Queries" section on page 8-12.

See Also: *Oracle9i SQL Reference* for more information about `CEIL`.

Row Object Storage in Object Tables

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

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

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

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

Performance of Object Comparisons

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

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

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

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

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

Storage Considerations for Object Identifiers (OIDs)

REFs use object identifiers (OIDs) to point to objects. You can use either system-generated OIDs or primary-key based OIDs. The differences between these types of OIDs are outlined in "Row Object Storage in Object Tables" on page 8-7. If you use system-generated OIDs for an object table, Oracle maintains an index on the column that stores these OIDs. The index requires storage space, and each row object has a system-generated OID, which requires an extra 16 bytes of storage for each row.

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

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

Storage Size of REFs

A REF contains the following three logical components:

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

Integrity Constraints for REF Columns

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

Performance and Storage Considerations for Scoped REFs

A *scoped REF* is constrained to contain only references to a specified object table. You can specify a scoped REF when you declare a column type, collection element, or object type attribute to be a REF.

In general, you should use scoped REFs instead of unscoped REFs because scoped REFs are stored more efficiently. Whereas an unscoped REF takes at least 36 bytes to store (more if it uses rowids), a scoped REF is stored as just the OID of its target object and can take less than 16 bytes, depending on whether the referenced OID is system-generated or primary-key based. A system-generated OID requires 16 bytes; a PK-based OID requires enough space to store the primary key value, which may be less than 16 bytes. (However, a REF to a PK-based OID, which must be dynamically constructed on being selected, may take more space *in memory* than a REF to a system-generated OID.)

Besides requiring less storage space, scoped REFs often enable the optimizer to optimize queries that dereference a scoped REF into more efficient joins. This optimization is not possible for unscoped REFs because the optimizer cannot determine the containing table(s) for unscoped REFs at query-optimization time.

Unlike referential integrity constraints, scoped REFs do not ensure that the referenced row object exists; they only ensure that the referenced object table exists.

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

Note: Referential integrity constraints are scoped implicitly.

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

Indexing Scoped REFs

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

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

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

Then, a `people_reltab2` table can be created that has the same definition as the `people_reltab` table discussed in "Column Object Storage" on page 8-3, except that a `REF` is used for the address:

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

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

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

The following query dereferences the `address_ref`:

```
SELECT id FROM people_reltab2 p
```

```
  WHERE p.address_ref.state = 'CA' ;
```

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

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

Speeding up Object Access using the WITH ROWID Option

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

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:

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

The next example shows an unnesting query on a multilevel nested table collection of nested tables. From a table of stars where each star has a nested table of planets, and each planet has a nested table of satellites, the query returns the names of all satellites.

```
CREATE TYPE satellite_t AS OBJECT (
  name        VARCHAR2(20),
  diameter    NUMBER);

CREATE TYPE nt_sat_t AS TABLE OF satellite_t;

CREATE TYPE planet_t AS OBJECT (
  name        VARCHAR2(20),
  mass        NUMBER,
  satellites  nt_sat_t);

CREATE TYPE nt_pl_t AS TABLE OF planet_t;
CREATE TABLE stars (
  name     VARCHAR2(20),
  age      NUMBER,
  planets  nt_pl_t)
NESTED TABLE planets STORE AS planets_tab
  (NESTED TABLE satellites STORE AS satellites_tab);

SELECT t.name FROM stars s, TABLE(s.planets) p, TABLE(p.satellites) t;
```

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

Outer-join syntax can also be used with queries of multilevel collections.

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:

```
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:

```
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:** *Oracle9i SQL Reference* for more information about using the `TABLE` syntax.

Storage Considerations for Varrays

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

Varrays are stored in columns either as raw values or `BLOB`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 `BLOB`s. Otherwise, the varray is stored in the column itself as a raw value. In addition, Oracle supports inline LOBs; therefore, elements that fit in the first 4000 bytes of a large varray (with some bytes reserved for the LOB locator) are stored in the column of the row itself.

Performance of Varrays 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 Also:** "Viewing Object Data in Relational Form with Unnesting Queries" on page 8-12 for more information.

Varray Updates

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

Nested Tables

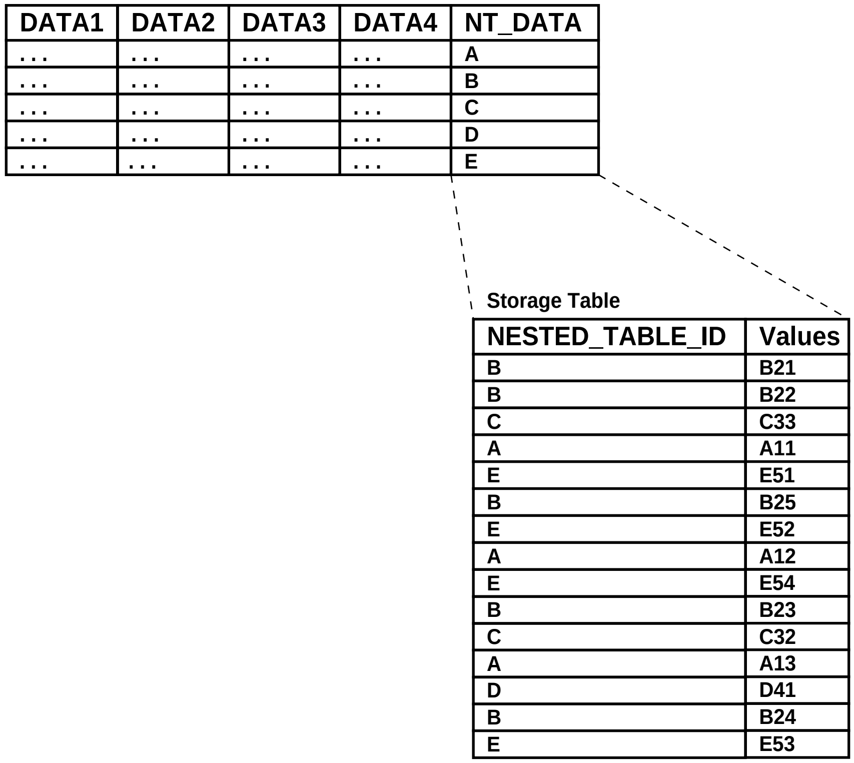
The following sections contain design considerations for using nested tables.

Nested Table Storage

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

Figure 8–5 shows how the storage table works. The storage table contains each value for each nested table in a nested table column. Each value occupies one row in the storage table. The storage table uses the NESTED_TABLE_ID to track the nested table for each value. So, in Figure 8–5, all of the values that belong to nested table A are identified, all of the values that belong to nested table B are identified, and so on.

Figure 8–5 Nested Table Storage



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

Storage Table

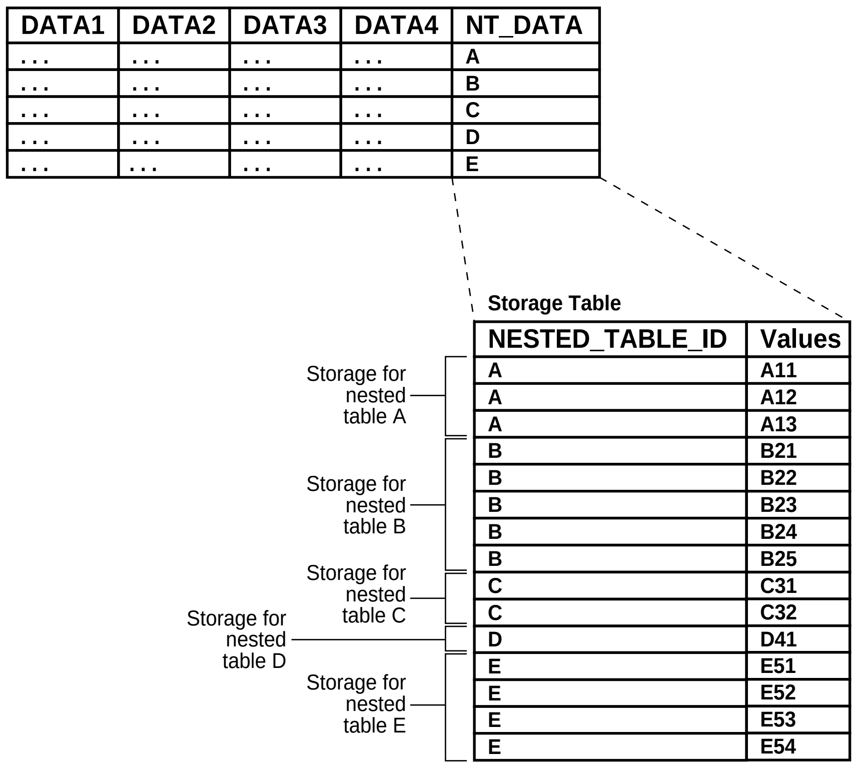
NESTED_TABLE_ID	Values
B	B21
B	B22
C	C33
A	A11
E	E51
B	B25
E	E52
A	A12
E	E54
B	B23
C	C32
A	A13
D	D41
B	B24
E	E53

Nested Table in an Index-Organized Table (IOT)

If a nested table has a primary key, you can organize the nested table as an index-organized table (IOT). If the `NESTED_TABLE_ID` column is a prefix of the primary key for a given parent row, Oracle physically clusters its 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 8–6 shows how the storage table works when the nested table is in an IOT. The storage table groups by `NESTED_TABLE_ID` the values for each nested table in a nested table column. In Figure 8–6, 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 8–6 Nested Table in IOT Storage



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

Storage Table

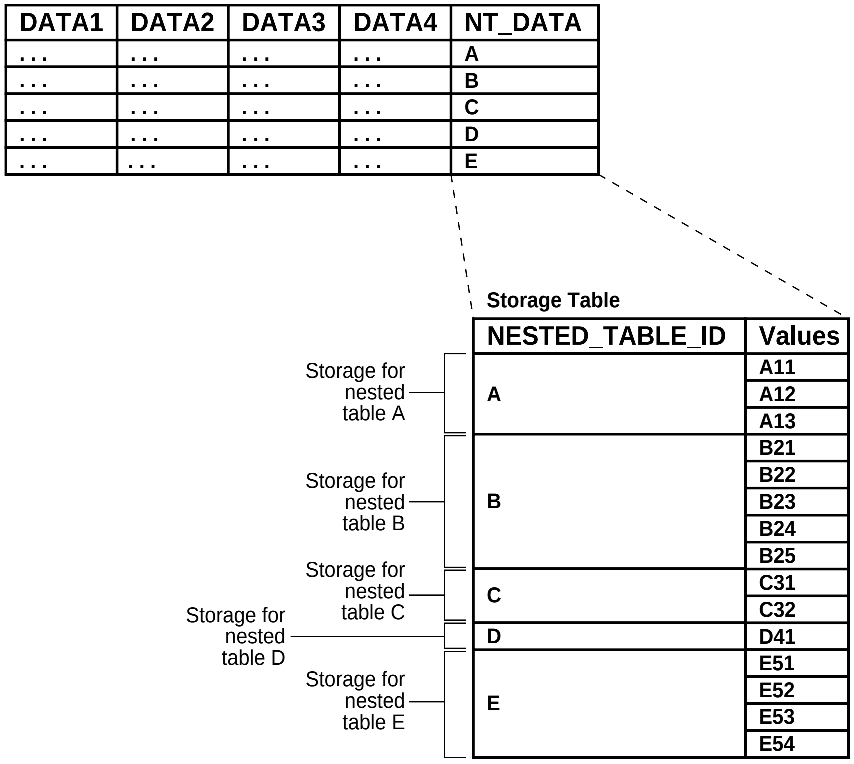
NESTED_TABLE_ID	Values
A	A11
A	A12
A	A13
B	B21
B	B22
B	B23
B	B24
B	B25
C	C31
C	C32
D	D41
E	E51
E	E52
E	E53
E	E54

Storage for nested table A
Storage for nested table B
Storage for nested table C
Storage for nested table D
Storage for nested table E

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

In other words, if you specify nested table compression using the `COMPRESS` clause, the amount of space required for the storage table is reduced because the `NESTED_TABLE_ID` is not repeated for each value in a group. Instead, the `NESTED_TABLE_ID` is stored only once for each group, as illustrated in Figure 8–7.

Figure 8–7 Nested Table in IOT Storage with Compression


DATA1
DATA2
DATA3
DATA4
NT_DATA
A
B
C
D
E
Storage Table
NESTED_TABLE_ID
Values
A11
A12
A13
B21
B22
B23
B24
B25
C31
C32
D41
E51
E52
E53
E54
Storage for nested table A
Storage for nested table B
Storage for nested table C
Storage for nested table D
Storage for nested table E

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

Nested Table Indexes

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

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

Nested Table Locators

For large child sets, the parent row and a locator to the child set can be returned so that the 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 *LNOCIColl** functions. The entire collection is retrieved implicitly on the first access.

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

- Use SQL to retrieve the rows corresponding to the nested table. This action is described in "The Object Table PurchaseOrder_objtab" on page 9-24.

In a multilevel collection, you can use a locator with a specified collection at any level of nesting. Following are described two ways in which to specify that a collection is to be retrieved as a locator.

At table creation time

When the collection type is being used as a column type and the `NESTED TABLE` storage clause is used, you can use the `RETURN LOCATOR` clause to specify that a particular collection is to be retrieved as a locator.

For instance, suppose that `third_level` is a collection type consisting of three levels of nested tables. In the following example, the `RETURN LOCATOR` clause specifies that the second, middle level of nested tables is always to be retrieved as a locator.

```
CREATE TABLE tab1 (
  a  NUMBER,
  b  third_level)
NESTED TABLE b STORE AS b_ntab
  (NESTED TABLE COLUMN_VALUE STORE AS cv1_ntab RETURN LOCATOR
    (NESTED TABLE COLUMN_VALUE STORE AS cv2_ntab ));
```

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 b from the table tab1 as a locator:

```
SELECT /*+ NESTED_TABLE_GET_REFS +*/ b
FROM tab1
WHERE a = 2;
```

Unlike with the RETURN 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 semi-join. However, this optimization only happens if the ALWAYS_SEMI_JOIN initialization parameter is set. If you want to perform semi-joins, the valid values for this parameter are MERGE and HASH; these parameter values indicate which join method to use.

Note: In the 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.

DML Operations on Nested Tables

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

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

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

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

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

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

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

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

You also can insert values into the nested table as you re-create it:

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

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

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

Multilevel Collections

Chapter 2 describes how to nest collection types to create a true multilevel collection—a nested table of nested tables, a nested table of varrays, a varray of nested tables, a varray of nested tables, or a varray or nested table of an object type that has an attribute of a collection type.

You can also nest collections indirectly using REFs. For example, you can create a nested table of an object type that has an attribute that *references* an object that has a nested table or varray attribute. If you do not actually need to access all elements of a multilevel collection, then nesting a collection with REFs may provide better performance because only the REFs need to be loaded, not the elements themselves.

True multilevel collections (specifically multilevel nested tables) perform better for queries that access individual elements of the collection. Using nested table locators can improve the performance of programmatic access if you do not need to access all elements.

For an example of a collection that uses REFs to nest another collection, suppose you want to create a new object type called person_objtyp using the object types described in "Column Object Storage" on page 8-3, which are name_objtyp, address_objtyp, and phone_ntabtyp. Remember that the phone_ntabtyp object type is a nested table because each person may have more than one phone number.

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

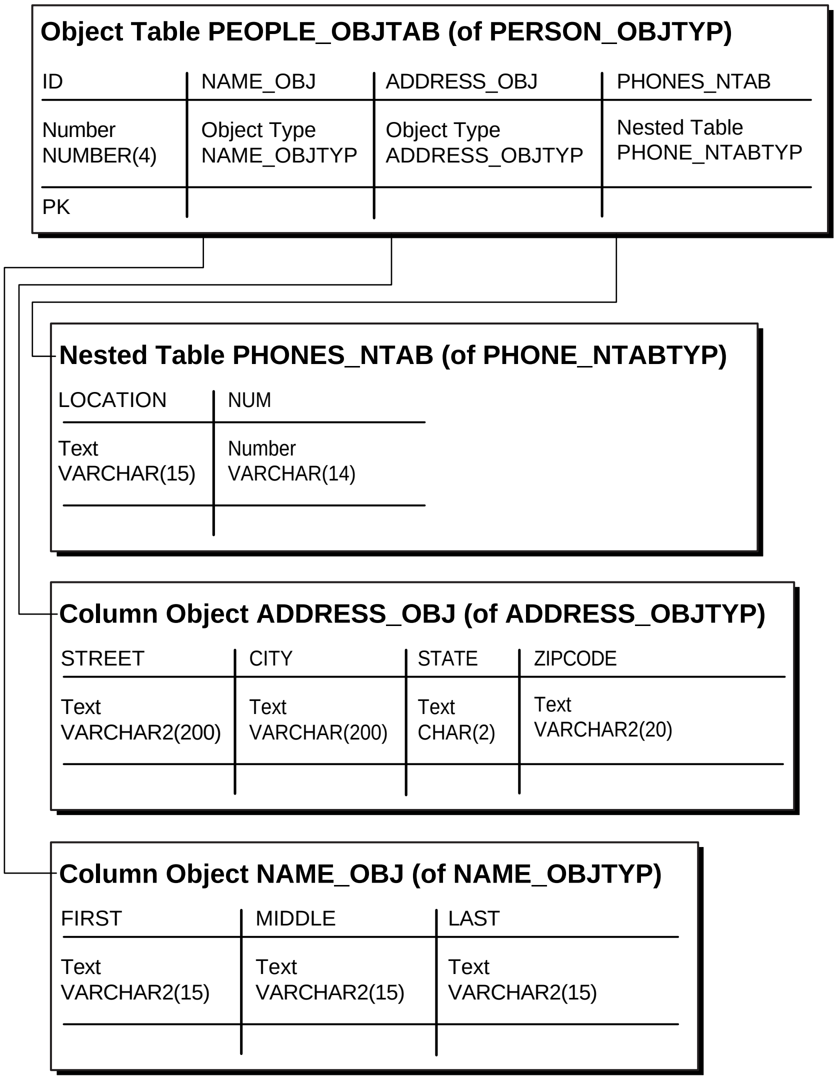
```
CREATE TYPE person_objtyp AS OBJECT (
   id            NUMBER(4),
   name_obj      name_objtyp,
   address_obj   address_objtyp,
   phones_ntab   phone_ntabtyp);
```

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

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

The people_objtab table has the same attributes as the people_reltab table discussed in "Column Object Storage" on page 8-3. The difference is that the `people_objtab` is an object table with row objects, while the `people_reltab` table is a relational table with three column objects.

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



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

- A project identification number for each project.

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

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

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

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

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

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

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

Figure 8–9 Object Relational Representation of the projects_objtab Object Table

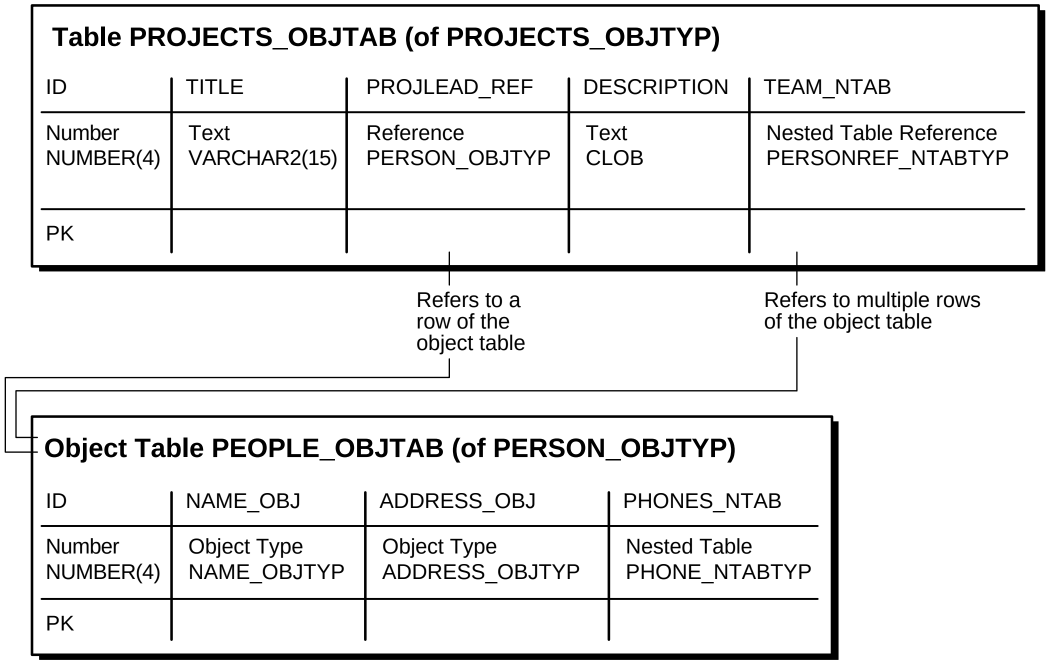


Table PROJECTS_OBJTAB (of PROJECTS_OBJTYP)
ID
TITLE
PROJLEAD_REF
DESCRIPTION
TEAM_NTAB
Number NUMBER(4)
Text VARCHAR2(15)
Reference PERSON_OBJTYP
Text CLOB
Nested Table Reference PERSONREF_NTABTYP
PK
Refers to a row of the object table
Refers to multiple rows of the object table
Object Table PEOPLE_OBJTAB (of PERSON_OBJTYP)
ID
NAME_OBJ
ADDRESS_OBJ
PHONES_NTAB
Number NUMBER(4)
Object Type NAME_OBJTYP
Object Type ADDRESS_OBJTYP
Nested Table PHONE_NTABTYP
PK

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

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

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

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

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

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

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

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

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

Choosing a Language for Method Functions

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

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

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

A method implemented in C executes in a separate process from the server using external 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.

Here is the object type specification and its methods:

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

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

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

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

END;
/
```

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

See Also:

- *Oracle9i Java Stored Procedures Developer's Guide* for more information.
- Chapter 3, "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.

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

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

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

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

Writing Reusable Code using Invoker Rights

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

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

For example, you can grant user SARA execute privileges on type atype created by SCOTT in "Static Methods" on page 8-30, and then create table atab based on the type:

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

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

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

This statement raises an error because the definer of the type (SCOTT) does not have the privileges required to perform the insert in the 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 SCOTT/TIGER ;
DROP TABLE atab ;

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

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

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

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

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

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

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's 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.

Examples :

```
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
```

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
  salary NUMBER,
  MEMBER FUNCTION bonus RETURN NUMBER DETERMINISTIC
);

CREATE OR REPLACE TYPE BODY emp_t IS
 MEMBER FUNCTION bonus RETURN NUMBER IS
 BEGIN
  RETURN self.salary * .1;
 END;
END;

CREATE TABLE emps OF emp_t ;
```

```
SELECT e
  FROM emps
  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`:

```
CREATE INDEX emps_bonus_idx ON emps x (x.bonus()) ;
```

> **See Also:** *Oracle9i Database Concepts* and *Oracle9i SQL Reference* for detailed information about function-based indexes.

Converting to the Current Object Format

Tables created in release 8.1 or higher store objects in a new format that uses less storage space and has better performance characteristics than the previous (relase 8.0) format. A more efficient transport protocol is used as well. If the `COMPATIBLE` parameter is set to 8.1.0 or higher, all objects in new tables and columns that you create are automatically stored in release 8.1 format, and all objects (new or old) are transported in the release 8.1 format. Tables created in release 8.0 will continue to store objects in the release 8.0 format unless explicitly converted.

You can convert objects created in a release 8.0 database to the format introduced in release 8.1. To do so, do the following steps:

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

> **See Also:** *Oracle9i Database Migration* for more information about compatibility and the `COMPATIBLE` initialization parameter.

Note: The release 8.0 format will be deprecated in a future release.

Replicating Object Tables and Columns

Object tables and object views can be replicated as materialized views. You can also replicate relational tables that contain columns of an object, collection, or REF type. Such materialized views are called **object-relational materialized views**.

All user-defined types required by an object-relational materialized view must exist at the materialized view site as well as at the master site. They must have the same object type IDs and versions at both sites.

Replicating Columns of Object, Collection, or REF Type

To be updatable, a materialized view based on a table that contains an object column must select the column *as an object* in the query that defines the view: if the query selects only certain attributes of the column's object type, then the materialized view is read-only.

The view-definition query can also select columns of collection or REF type. REFs can be either primary-key based or have a system-generated key, and they can be either scoped or unscoped. Scoped REF columns can be rescoped to a different table at the site of the materialized view—for example, to a local materialized view of the master table instead of the original, remote table.

Replicating Object Tables

A materialized view based on an object table is called an **object materialized view**. Such a materialized view is itself an object table. An object materialized view is created by adding the `OF <type>` keyword to the `CREATE MATERIALIZED VIEW` statement. For example:

```
CREATE MATERIALIZED VIEW customer OF cust_objtyp
AS SELECT * FROM Scott.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 Scott.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/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 Scott.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 Scott.Customer_objtab@dbs1;
```

Otherwise, the `FROM` clause must omit the `ONLY` keyword.

> **See Also:** *Oracle9i 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:

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

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

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

CREATE TABLE department_tab OF department_type (
   deptno PRIMARY KEY);

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

Type Evolution

The following sections contain design considerations relating to type evolution.

Pushing a Type Change Out to Clients

Once a type has evolved on the server side, all client applications using this type need to make the necessary changes to structures associated with the type. You can do this with OTT/JPUB. You also may need to make programmatic changes associated with the structural change. After making these changes, you must recompile your application and relink.

Types may be altered between releases of a third-party application. To inform client applications that they need to recompile to become compatible with the latest release of the third-party application, you can have the clients call a release-oriented compatibility initialization function. This function could take as input a string that tells it which release the client application is working with. If the release string mismatches with the latest version, an error is generated. The client application must then change the release string as part of the changes required to become compatible with the latest release.

For example:

```
FUNCTION compatibility_init(rel IN VARCHAR2, errmsg OUT VARCHAR2)
RETURN NUMBER;
```

where:

`rel` is a release string that is chosen by the product—for example, `'Release 8.2'`

`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 Also: "User-Defined Constructors" on page 6-21

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)`.

Performance Tuning

See *Oracle9i Database Performance Tuning Guide and Reference* for details on measuring and tuning the performance of your application. In particular, some of the key performance factors are the following:

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

Parallel Queries with Oracle Objects

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

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

- Parallel DML and parallel DDL are not supported with objects. DML and DDL are always performed in serial.

Tips and Techniques

The following sections provide assorted tips on various aspects of working with Oracle object types.

Deciding Whether to Evolve a Type or Create a Subtype Instead

As an application goes through its life cycle, the question often arises whether to change an existing user-defined 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 a user-defined 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 datatype, 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 5 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

What is the intended use of SQLJ Object Type?

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 Oracle9*i* JDBC driver. This enables the user to deploy the same class in the client, through JDBC, and in the server, through SQL method dispatch.

What is involved in 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, depdencies 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.

When would you use SQLJ Object Type?

The SQLJ object type is a special case of SQL object type in which all methods are implemented in a Java class(es). 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.

When would you use Custom Object Type?

The custom object type is the Java interface for accessing SQL object type. 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, JPub 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.

What are the differences between the SQLJ and Custom Object Types through JDBC?

The following table summarizes the differences between SQLJ object types 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`.
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 *Oracle9i SQL Reference* for a complete description of these rules.	There are no mapping rules.

Miscellaneous Tips

Column Substitutability and the Number of Attributes in a Hierarchy

If a column or table is of type `T`, Oracle adds a hidden column for each attribute of type `T` and, if the column or table is substitutable, for each attribute of every subtype of `T`, to store attribute data. A hidden `typeid` column is added as well, to keep track of the type of the object instance in a row.

The number of columns in a table is limited to 1,000. A type hierarchy with a number of total attributes approaching 1,000 puts you at risk of running up against this limit when using substitutable columns of a type in the hierarchy. To avoid problems as a result of this, consider one of the following options for dealing with a hierarchy that has a large number of total attributes:

- Use views
- Use REFs
- 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`.

PL/SQL and TREAT and IS OF

PL/SQL does not currently support the `TREAT` and `IS OF` operators (see Chapter 2), but SQL does. To use these operators, use SQL.

9
A Sample Application Using Object-Relational Features

This chapter contains an extended example that gives an overview of how to create and use user-defined datatypes (Oracle Objects).

The example develops 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 user-defined types for these entities and manages data for particular customers and purchase orders by instantiating instances of the respective user-defined types.

This chapter contains the following sections:

Introduction

User-defined types are schema objects in which users formalize the data structures and operations that appear in their applications.

The example in this chapter illustrates the most important aspects of defining, using, and evolving user-defined types. One important aspect of working with user-defined types is creating methods that perform operations on objects. In the example, definitions of object type methods use the PL/SQL language. Other aspects of using user-defined types, such as defining a type, use SQL.

PL/SQL and Java provide additional capabilities beyond those illustrated in this chapter, especially in the area of accessing and manipulating the elements of collections.

Client applications that use the Oracle Call Interface (OCI), Pro*C/C++, or Oracle Objects for OLE (OO4O) can take advantage of its extensive facilities for accessing objects and collections, and manipulating them on clients.

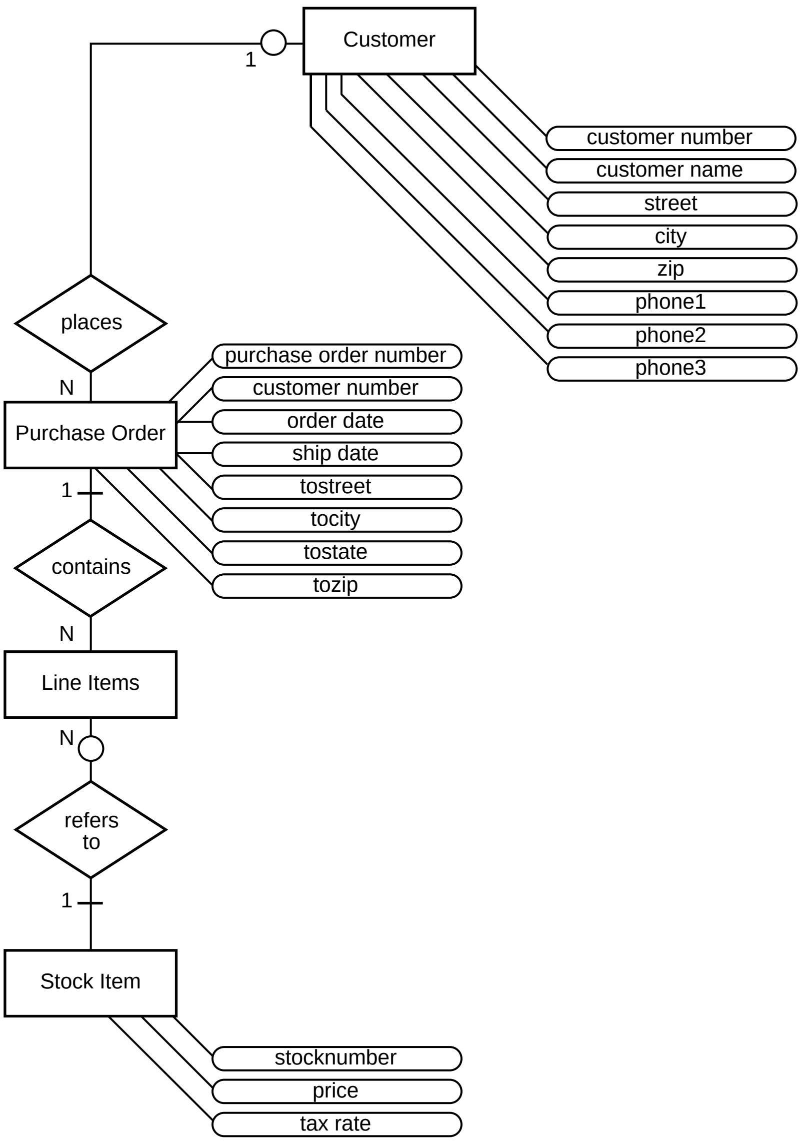
See Also:

- *Oracle9i SQL Reference* for a complete description of SQL syntax and usage for user-defined types.
- *PL/SQL User's Guide and Reference* for a complete discussion of PL/SQL capabilities
- *Oracle9i Java Stored Procedures Developer's Guide* for a complete discussion of Java.
- *Oracle Call Interface Programmer's Guide,*
- *Pro*C/C++ Precompiler Programmer's Guide*

Implementing the Schema on the Relational Model

This section implements the relational version of the purchase order schema depicted in Figure 9–1.

Figure 9–1 Entity-Relationship Diagram for Purchase Order Application



Entities and Relationships

The basic entities in this example are:

- Customers
- The stock of products for sale
- Purchase orders

As you can see from Figure 9–1, 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.

Creating Tables Under the Relational Model

The relational approach normalizes everything into tables. The table names are `Customer_reltab`, `PurchaseOrder_reltab`, and `Stock_reltab`.

Each part of an address becomes a column in the `Customer_reltab` table.

Structuring telephone numbers as columns sets an arbitrary limit on the number of telephone numbers a customer can have.

The relational approach separates line items from their purchase orders and puts each into its own table, named PurchaseOrder_reltab and LineItems_reltab. As depicted in Figure 9–1, 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 give them names that closely model the corresponding real-world objects.

The relational approach results in the following tables:

Customer_reltab

The Customer_reltab table has the following definition:

```
CREATE TABLE Customer_reltab (
  CustNo                NUMBER NOT NULL,
  CustName              VARCHAR2(200) NOT NULL,
  Street                VARCHAR2(200) NOT NULL,
  City                  VARCHAR2(200) NOT NULL,
  State                 CHAR(2) NOT NULL,
  Zip                   VARCHAR2(20) NOT NULL,
  Phone1                VARCHAR2(20),
  Phone2                VARCHAR2(20),
  Phone3                VARCHAR2(20),
  PRIMARY KEY (CustNo)
  ) ;
```

This table, Customer_reltab, stores all the information about customers, which means that it fully contains information that is intrinsic to the customer (defined with the NOT NULL constraint) and information that is not as essential. According to this definition of the table, the application requires that every customer have a shipping address.

Our Entity-Relationship (E-R) diagram showed a customer placing an order, but the table does not make allowance for any relationship between the customer and the purchase order. This relationship must be managed by the purchase order.

PurchaseOrder_reltab

The `PurchaseOrder_reltab` table has the following definition:

```
CREATE TABLE PurchaseOrder_reltab (
   PONo        NUMBER, /* purchase order no */
   Custno      NUMBER references Customer_reltab, /*  Foreign KEY referencing
                                                      customer */
   OrderDate   DATE, /*  date of order */
   ShipDate    DATE, /* date to be shipped */
   ToStreet    VARCHAR2(200), /* shipto address */
   ToCity      VARCHAR2(200),
   ToState     CHAR(2),
   ToZip       VARCHAR2(20),
   PRIMARY KEY(PONo)
   ) ;
```

`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 (next section) uses the purchase order number to relate a line item to its parent purchase order.

LineItems_reltab

The `LineItems_reltab` table has the following definition:

```
CREATE TABLE LineItems_reltab (
  LineItemNo          NUMBER,
  PONo                NUMBER REFERENCES PurchaseOrder_reltab,
  StockNo             NUMBER REFERENCES Stock_reltab,
  Quantity            NUMBER,
  Discount            NUMBER,
  PRIMARY KEY (PONo, LineItemNo)
  ) ;
```

Note: The `Stock_reltab` table, described in "Stock_reltab" on page 9-7, must be created before the `LineItems_reltab` table.

The table name is in the plural form `LineItems_reltab` to emphasize to someone reading the code that the table holds a collection of line items.

As shown in the E-R diagram, the list of line items has relationships with both the purchase order and the stock item. These relationships are managed by `LineItems_reltab` by means of two foreign key columns:

- `PONo`, which references the `PONo` column in `PurchaseOrder_reltab`
- `StockNo`, which references the `StockNo` column in `Stock_reltab`

Stock_reltab

The `Stock_reltab` table has the following definition:

```
CREATE TABLE Stock_reltab (
  StockNo      NUMBER PRIMARY KEY,
  Price        NUMBER,
  TaxRate      NUMBER
  ) ;
```

Inserting Values Under the Relational Model

In our application, statements like these insert data into the tables:

Establish Inventory

```
INSERT INTO Stock_reltab VALUES(1004, 6750.00, 2) ;
INSERT INTO Stock_reltab VALUES(1011, 4500.23, 2) ;
INSERT INTO Stock_reltab VALUES(1534, 2234.00, 2) ;
INSERT INTO Stock_reltab VALUES(1535, 3456.23, 2) ;
```

Register Customers

```
INSERT INTO Customer_reltab
  VALUES (1, 'Jean Nance', '2 Avocet Drive',
         'Redwood Shores', 'CA', '95054',
         '415-555-1212', NULL, NULL) ;

INSERT INTO Customer_reltab
  VALUES (2, 'John Nike', '323 College Drive',
         'Edison', 'NJ', '08820',
         ’609-555-1212’, ’201-555-1212’, NULL) ;
```

Place Orders

```
INSERT INTO PurchaseOrder_reltab
  VALUES (1001, 1, SYSDATE, ’10-MAY-1997’,
          NULL, NULL, NULL, NULL) ;

INSERT INTO PurchaseOrder_reltab
  VALUES (2001, 2, SYSDATE, ’20-MAY-1997’,
         ’55 Madison Ave’, ’Madison’, ’WI’, ’53715’) ;
```

Detail Line Items

```
INSERT INTO LineItems_reltab VALUES(01, 1001, 1534, 12,  0) ;
INSERT INTO LineItems_reltab VALUES(02, 1001, 1535, 10, 10) ;
INSERT INTO LineItems_reltab VALUES(01, 2001, 1004,  1,  0) ;
INSERT INTO LineItems_reltab VALUES(02, 2001, 1011,  2,  1) ;
```

Querying Data Under The Relational Model

The application can execute queries like these:

Get Customer and Line Item Data for a Specific Purchase Order

```
SELECT   C.CustNo, C.CustName, C.Street, C.City, C.State,
         C.Zip, C.phone1, C.phone2, C.phone3,
         P.PONo, P.OrderDate,
         L.StockNo, L.LineItemNo, L.Quantity, L.Discount
 FROM    Customer_reltab C,
         PurchaseOrder_reltab P,
         LineItems_reltab L
 WHERE   C.CustNo = P.CustNo
  AND    P.PONo = L.PONo
  AND    P.PONo = 1001 ;
```

Get the Total Value of Purchase Orders

```
SELECT     P.PONo, SUM(S.Price * L.Quantity)
 FROM      PurchaseOrder_reltab P,
           LineItems_reltab L,
           Stock_reltab S
 WHERE     P.PONo = L.PONo
  AND      L.StockNo = S.StockNo
 GROUP BY P.PONo ;
```

Get the Purchase Order and Line Item Data for those LineItems That Use a Stock Item Identified by a Specific Stock Number

```
SELECT    P.PONo, P.CustNo,
          L.StockNo, L.LineItemNo, L.Quantity, L.Discount
 FROM     PurchaseOrder_reltab P,
          LineItems_reltab     L
 WHERE    P.PONo = L.PONo
   AND    L.StockNo = 1004 ;
```

Updating Data Under The Relational Model

The application can execute statements like these to update the data:

Update the Quantity for Purchase Order 1001 and Stock Item 1534

```
UPDATE LineItems_reltab
   SET      Quantity = 20
   WHERE    PONo     = 1001
   AND      StockNo  = 1534 ;
```

Deleting Data Under The Relational Model

The application can execute statements like these to delete data:

Delete Purchase Order 1001

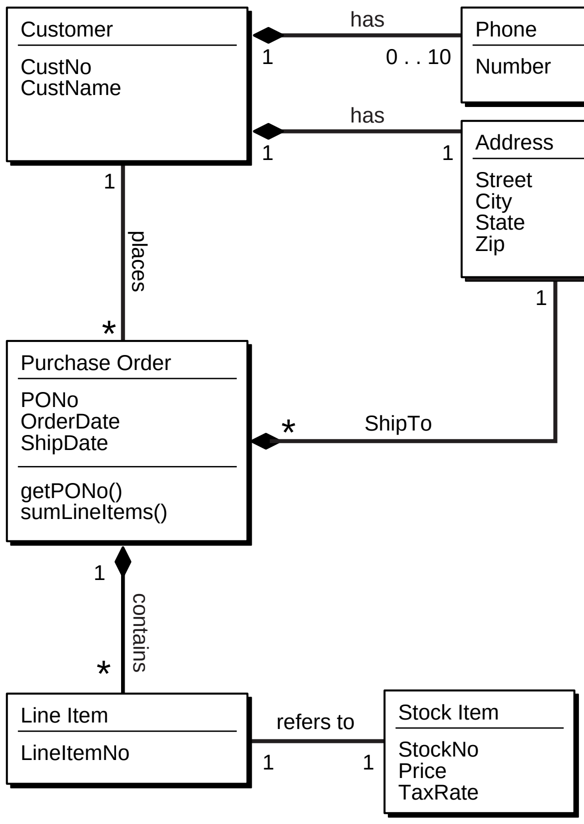
```
DELETE
   FROM   LineItems_reltab
   WHERE  PONo = 1001 ;

DELETE
   FROM   PurchaseOrder_reltab
   WHERE  PONo = 1001 ;
```

Implementing the Schema on the Object-Relational Model

The object-relational (O-R) approach begins with the same entity relationships as in "Entities and Relationships" on page 9-3. 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 9–2 Class Diagram for Purchase Order Application



Instead of breaking up addresses or multiple phone numbers into unrelated columns in relational tables, the O-R approach defines types to represent an entire address and an entire list of phone numbers. Similarly, the O-R 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 chapter implements an object-relational interface by building an object-relational schema from scratch. On 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 5 discusses object views.

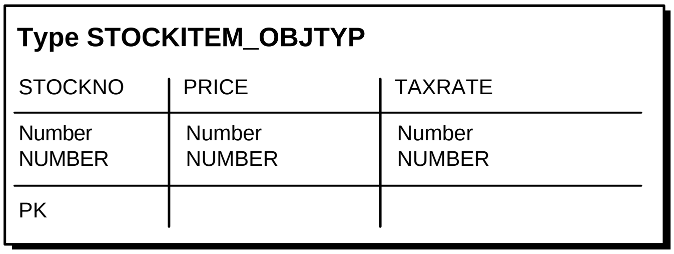
Defining Types

You create a user-defined type with a `CREATE TYPE` statement. For example, the following statement creates the type `StockItem_objtyp`:

```
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.

Figure 9–3 Object Relational Representation of the StockItem_objtyp



Type STOCKITEM_OBJTYP

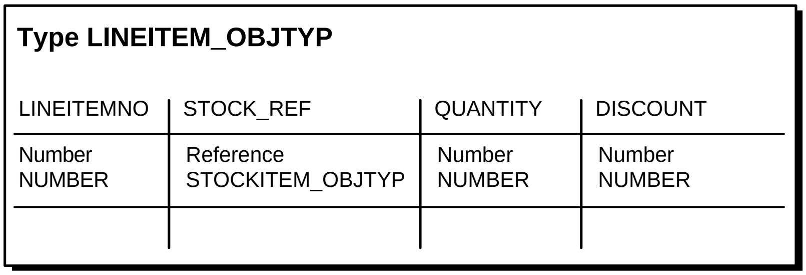
STOCKNO	PRICE	TAXRATE
Number NUMBER	Number NUMBER	Number NUMBER
PK		

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`:

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

Figure 9–4 Object Relational Representation of LineItem_objtyp Type



Type LINEITEM_OBJTYP

LINEITEMNO	STOCK_REF	QUANTITY	DISCOUNT
Number NUMBER	Reference STOCKITEM_OBJTYP	Number NUMBER	Number NUMBER

Instances of type `LineItem_objtyp` are objects that represent line items. They have three numeric attributes and one `REF` attribute. The `LineItem_objtyp` models the line item entity and includes an object reference to the corresponding stock object.

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 lacks that phrase `AS OBJECT` and also lacks the specification of attributes.

To replace an incomplete type with a complete definition, include the phrase OR REPLACE as shown in the following example:

```
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 let's create the remaining types we need for the schema. The following statement defines an array type for the list of phone numbers:

```
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.

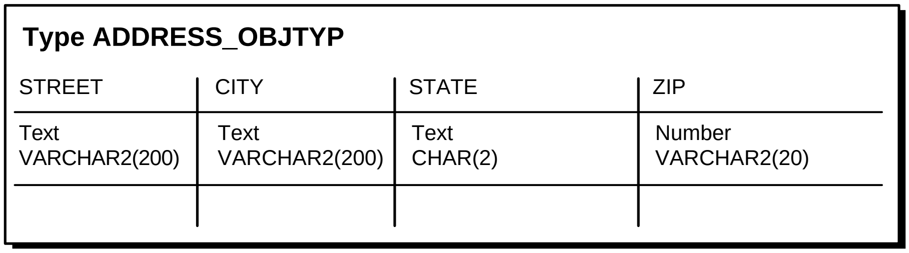
In general, if ordering and bounds are not important design considerations, then designers can use the following rule of thumb for deciding between varrays and nested tables: If you need to query the collection, then use nested tables; if you intend to retrieve the collection as a whole, then use varrays.

See Also: Chapter 8, "Design Considerations for Oracle Objects" for more information about the design considerations for varrays and nested tables.

The following statement defines the object type `Address_objtyp` to represent addresses:

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

Figure 9–5 Object Relational Representation of Address_objtyp Type



All of the attributes of an address are character strings, representing the usual parts of a simplified mailing address.

The following statement defines the object type `Customer_objtyp`, which uses other user-defined types as building blocks.

```
CREATE TYPE Customer_objtyp AS OBJECT (
  CustNo            NUMBER,
  CustName          VARCHAR2(200),
  Address_obj       Address_objtyp,
  PhoneList_var     PhoneList_vartyp,

  ORDER MEMBER FUNCTION
    compareCustOrders(x IN Customer_objtyp) RETURN INTEGER
) 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 chapter.

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 9-19.

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 for more information about map and order methods
- *PL/SQL User's Guide and Reference* for details about how to use pragma declarations

The following statement defines a type for a nested table of line items. Each purchase order will use an instance of this nested table type to contain the line items for that purchase order:

```
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.

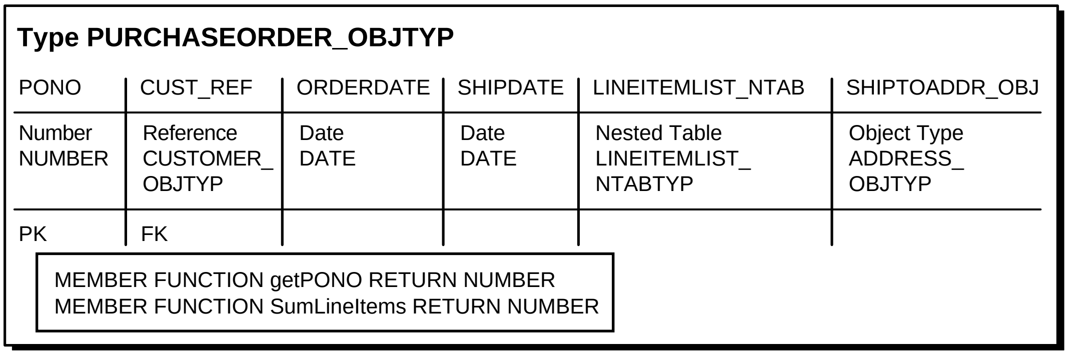
The following statement defines the object type `PurchaseOrder_objtyp`:

```
CREATE TYPE PurchaseOrder_objtyp AUTHID CURRENT_USER AS OBJECT (
  PONo                NUMBER,
  Cust_ref            REF Customer_objtyp,
  OrderDate           DATE,
  ShipDate            DATE,
  LineItemList_ntab   LineItemList_ntabtyp,
  ShipToAddr_obj      Address_objtyp,

  MAP MEMBER FUNCTION
    getPONo RETURN NUMBER,

  MEMBER FUNCTION
    sumLineItems RETURN NUMBER
  );
```

Figure 9–6 Object Relational Representation of the PuchaseOrder_objtyp



Type PURCHASEORDER_OBJTYP

PONO	CUST_REF	ORDERDATE	SHIPDATE	LINEITEMLIST_NTAB	SHIPTOADDR_OBJ
Number NUMBER	Reference CUSTOMER_ OBJTYP	Date DATE	Date DATE	Nested Table LINEITEMLIST_ NTABTYP	Object Type ADDRESS_ OBJTYP
PK	FK				

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`.

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 9-17.

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 supplys the PL/SQL programs that implement the type's methods:

```
CREATE OR REPLACE TYPE BODY PurchaseOrder_objtyp AS

MAP MEMBER FUNCTION getPONo RETURN NUMBER is
   BEGIN
      RETURN PONo;
   END;

MEMBER FUNCTION sumLineItems RETURN NUMBER is
      i             INTEGER;
      StockVal      StockItem_objtyp;
      Total         NUMBER := 0;

   BEGIN
      FOR i in 1..SELF.LineItemList_ntab.COUNT LOOP
         UTL_REF.SELECT_OBJECT(LineItemList_ntab(i).Stock_ref,StockVal);
         Total := Total + SELF.LineItemList_ntab(i).Quantity * StockVal.Price;
      END LOOP;
      RETURN Total;
   END;
END;
/
```

The getPONo Method

The `getPONo` method simply returns the value of the `PONo` attribute—namely, the purchase order number—of whatever instance of the type `PurchaseOrder_objtyp` that calls the method. Such "get" methods allow you to avoid reworking code that uses the object if its internal representation changes.

The sumLineItems Method

The `sumLineItems` method uses a number of object-relational features:

- As already noted, the basic function of the `sumLineItems` method is to return the sum of the values of the line items of its associated `PurchaseOrder_objtyp` object. The keyword `SELF`, which is implicitly created as a parameter to every function, lets you refer to that object.
- The keyword `COUNT` gives the count of the number of elements in a PL/SQL table or array. Here, in combination with `LOOP`, the application iterates through all the elements in the collection — in this case, the items of the purchase order. In this way `SELF.LineItemList_ntab.COUNT` counts the number of elements

in the nested table that match the LineItemList_ntab attribute of the PurchaseOrder_objtyp object, here represented by SELF.

- A method from package UTL_REF is used in the implementation. The UTL_REF methods are necessary because Oracle does not support implicit dereferencing of REFs within PL/SQL programs. The UTL_REF package provides methods that operate on object references. Here, the SELECT_OBJECT method is called to obtain the StockItem_objtyp object corresponding to the Stock_ref.
- The AUTHID CURRENT_USER syntax specifies that the PurchaseOrder_objtyp is defined using invoker-rights: the methods are executed under the rights of the current user, not under the rights of the user who defined the type.
- The PL/SQL variable StockVal is of type StockItem_objtyp. The UTL_REF.SELECT_OBJECT sets it to the object whose reference is the following:

  ```
  (LineItemList_ntab(i).Stock_ref)
  ```

  This object is the actual stock item referred to in the currently selected line item.
- Having retrieved the stock item in question, the next step is to compute its cost. The program refers to the stock item's cost as StockVal.Price, the Price attribute of the StockItem_objtyp object. But to compute the cost of the item, you also need to know the quantity of items ordered. In the application, the term LineItemList_ntab(i).Quantity represents the Quantity attribute of the currently selected LineItem_objtyp object.

The remainder of the method program is a loop that sums the values of the line items. The method returns the total.

The compareCustOrders Method

The following statement defines the compareCustOrders method in the type body of the Customer_objtyp object type:

```
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 user-defined 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 5, "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`:

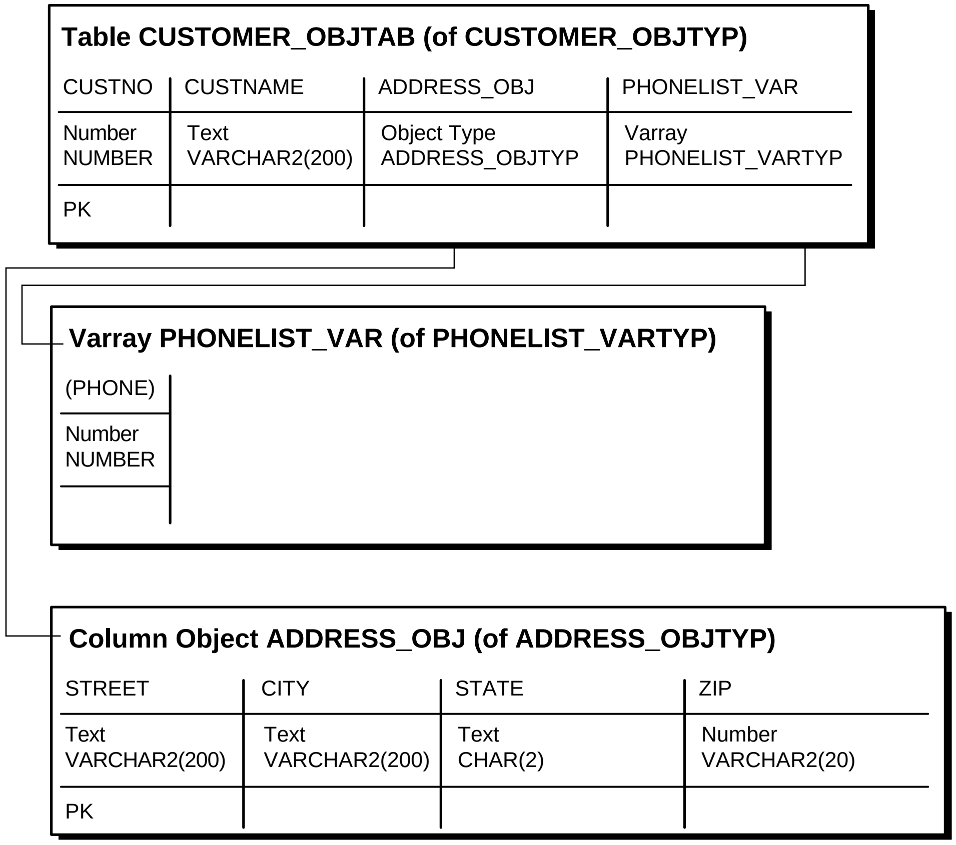
```
CREATE TABLE Customer_objtab OF Customer_objtyp (CustNo PRIMARY KEY)
   OBJECT IDENTIFIER IS PRIMARY KEY ;
```

Unlike with relational tables, when you create an object table, you specify a data type for it, namely, the type of objects it will contain.

The table has a column for each attribute of `Customer_objtyp`, namely:

```
CustNo          NUMBER
CustName        VARCHAR2(200)
Address_obj     Address_objtyp
PhoneList_var   PhoneList_vartyp
```

Figure 9–7 Object Relational Representation of Table Customer_objtab



Object Datatypes as a Template for Object Tables

Because there is a type `Customer_objtyp`, you could create numerous object tables of the same type. For example, you could create an object table `Customer_objtab2` also of type `Customer_objtyp`.

You can introduce variations when creating multiple tables. The statement that created `Customer_objtab` defined a primary key constraint on the `CustNo` column. This constraint applies only to this object table. Another object table of the same type might not have this constraint.

Object Identifiers and References

Customer_objtab contains customer objects, represented as row objects. Oracle allows row objects to be referenceable, meaning that other row objects or relational rows may reference a row object using its object identifier (OID). For example, a purchase order row object may reference a customer row object using its object reference. The object reference is a system-generated value represented by the type REF and is based on the row object's unique OID.

Oracle requires every row object to have a unique OID. You may specify the unique OID value to be system-generated or specify the row object's primary key to serve as its unique OID. You indicate this when you execute the CREATE TABLE statement by specifying OBJECT IDENTIFIER IS PRIMARY KEY or OBJECT IDENTIFIER IS SYSTEM GENERATED. The latter is the default. Using the primary key as the object identifier can be more efficient in cases where the primary key value is smaller than the default 16 byte system-generated identifier. For our example, the primary key is used as the row object identifier.

Object Tables with Embedded Objects

Note that the Address_obj column of Customer_objtab contains Address_objtyp objects. As this shows, an object type may have attributes that are themselves object types. Object instances of the declared type of an object table are called **row objects** because one object instance occupies an entire row of the table. But embedded objects such as those in the Address_obj column are referred to as **column objects**. These differ from row objects in that they do not take up an entire row. Consequently, they are not referenceable—they cannot be the target of a REF. Also, they can be NULL.

The attributes of Address_objtyp objects are of built-in types. They are scalar rather than complex (that is, they are not object types with attributes of their own), and so are called **leaf-level** attributes to reflect that they represent an end to branching. Columns for Address_objtyp objects and their attributes are created in the object table Customer_objtab. You can refer or *navidate* 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:

```
CREATE TYPE PhoneList_vartyp AS VARRAY(10) OF VARCHAR2(20);
```

Because each varray of type PhoneList_vartyp can contain no more than 200 characters (10 x 20), plus a small amount of overhead, Oracle stores the varray as a single data unit in the PhoneList_var column. Oracle stores varrays that 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 next statement creates an object table for StockItem_objtyp objects:

```
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 next statement defines an object table for PurchaseOrder_objtyp objects:

```
CREATE TABLE PurchaseOrder_objtab OF PurchaseOrder_objtyp (  /* Line 1 */
   PRIMARY KEY (PONo),                                        /* Line 2 */
   FOREIGN KEY (Cust_ref) REFERENCES Customer_objtab)         /* Line 3 */
   OBJECT 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
```

Figure 9–8 Object Relational Representation of Table PurchaseOrder_objtab

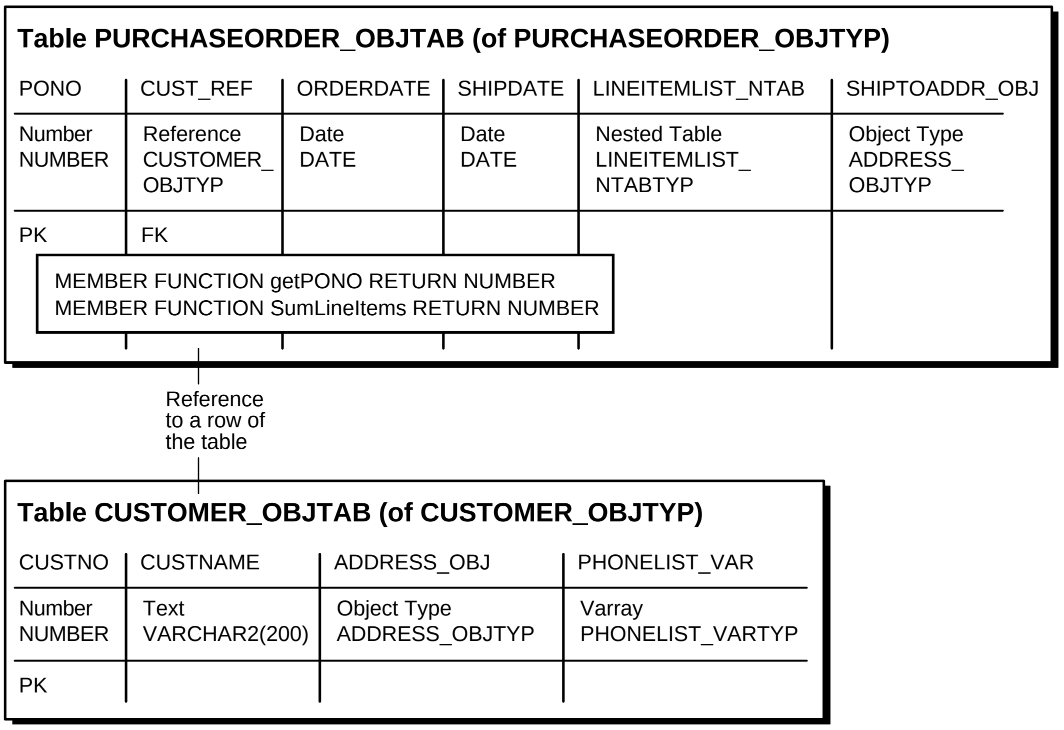


Table PURCHASEORDER_OBJTAB (of PURCHASEORDER_OBJTYP)
PONO
CUST_REF
ORDERDATE
SHIPDATE
LINEITEMLIST_NTAB
SHIPTOADDR_OBJ
Number NUMBER
Reference CUSTOMER_ OBJTYP
Date DATE
Date DATE
Nested Table LINEITEMLIST_ NTABTYP
Object Type ADDRESS_ OBJTYP
PK
FK
MEMBER FUNCTION getPONO RETURN NUMBER
MEMBER FUNCTION SumLineItems RETURN NUMBER
Reference to a row of the table
Table CUSTOMER_OBJTAB (of CUSTOMER_OBJTYP)
CUSTNO
CUSTNAME
ADDRESS_OBJ
PHONELIST_VAR
Number NUMBER
Text VARCHAR2(200)
Object Type ADDRESS_OBJTYP
Varray PHONELIST_VARTYP
PK

Line 2:

```
PRIMARY KEY (PONo),
```

This line specifies that the `PONo` attribute is the primary key for the table.

Line 3:

```
FOREIGN KEY (Cust_ref) REFERENCES Customer_objtab)
```

This line specifies a referential constraint on the `Cust_ref` column. This referential constraint is similar to those specified for relational tables. When there is no constraint, the `REF` column allows you to reference any row object. However, in this case, the `Cust_ref` `REF`s 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 is not directly queryable by the user but can be referenced in DDL statements for maintenance purposes. A hidden column in the storage table, called the `NESTED_TABLE_ID,` matches the rows with their corresponding parent row. All the elements in the nested table belonging to a particular parent have the same `NESTED_TABLE_ID` value. For example, all the elements of the nested table of a given row of `PurchaseOrder_objtab` have the same value of `NESTED_TABLE_ID`. The nested table elements that belong to a different row of `PurchaseOrder_objtab` have a different value of `NESTED_TABLE_ID`.

In the 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 allows you to specify the constraint and storage specification for the storage table. In this example, Line 7 indicates that the storage table is an index-organized table (`IOT`). In general, storing nested table rows in an IOT is beneficial because it provides clustering of rows belonging to the same parent. The specification of `COMPRESS` on the `IOT` saves storage space because, if you do not specify `COMPRESS`, the `NESTED_`

TABLE_ID part of the IOT's key is repeated for every row of a parent row object. If, however, you specify COMPRESS, the NESTED_TABLE_ID is stored only once for each parent row object.

The SCOPE FOR constraint on a REF is not allowed in a CREATE TABLE statement. Therefore, to specify that Stock_ref can reference only the object table Stock_objtab, issue the following ALTER TABLE statement on the PoLine_ntab storage table:

```
ALTER TABLE PoLine_ntab
   ADD (SCOPE FOR (Stock_ref) IS stock_objtab) ;
```

Note that this statement is executed on the storage table, not the parent table.

See Also: "Nested Table Storage" on page 8-16 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 LNOCICollIsLocator 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 9-17. That implementation assumed that the `LineItemList_ntab` nested table would be returned as a `VALUE`. In order to handle large nested tables more efficiently, and to take advantage of the fact that the nested table in the `PurchaseOrder_objtab` is returned as a locator, the `sumLineItems` method must be rewritten as follows:

```
CREATE OR REPLACE TYPE BODY PurchaseOrder_objtyp AS

   MAP MEMBER FUNCTION getPONo RETURN NUMBER is
      BEGIN
         RETURN PONo;
      END;

   MEMBER FUNCTION sumLineItems RETURN NUMBER IS
      i          INTEGER;
      StockVal   StockItem_objtyp;
      Total      NUMBER := 0;

   BEGIN
      IF (UTL_COLL.IS_LOCATOR(LineItemList_ntab)) -- check for locator
         THEN
            SELECT SUM(L.Quantity * L.Stock_ref.Price) INTO Total
            FROM   TABLE(CAST(LineItemList_ntab AS LineItemList_ntabtyp)) L;
      ELSE
         FOR i in 1..SELF.LineItemList_ntab.COUNT LOOP
            UTL_REF.SELECT_OBJECT(LineItemList_ntab(i).Stock_ref,StockVal);
            Total := Total + SELF.LineItemList_ntab(i).Quantity *
                                                               StockVal.Price;
         END LOOP;
      END IF;
   RETURN Total;
   END;
END;
/
```

The rewritten `sumLineItems` method checks whether the nested table attribute, `LineItemList_ntab`, is returned as a locator using the `UTL_COLL.IS_LOCATOR` function. 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.

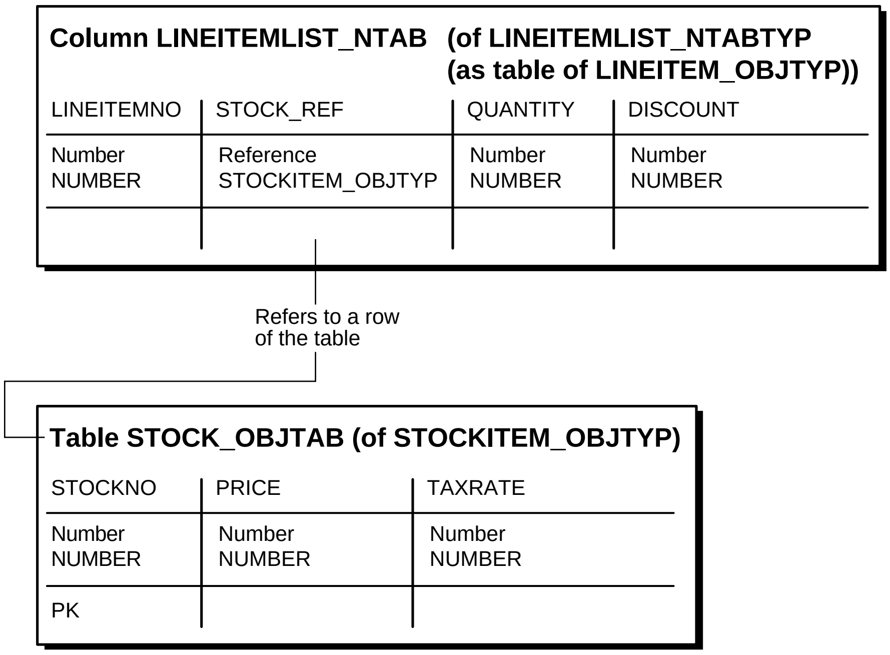
The querying of the nested table locator results in more efficient processing of the large line item list of a purchase order. The previous code that iterates over the LineItemList_ntab is kept to deal with the case where the nested table is returned as a VALUE.

After the table is created, the following ALTER TABLE statement is issued:

```
ALTER TABLE PoLine_ntab
   ADD (SCOPE FOR (Stock_ref) IS stock_objtab);
```

This statement specifies that the Stock_ref column of the nested table is scoped to Stock_objtab. This indicates that the values stored in this column must be references to row objects in Stock_objtab. The SCOPE constraint is different from the referential constraint 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 9–9 Object Relational Representation of Nested Table LineItemList_ntab


Column LINEITEMLIST_NTAB (of LINEITEMLIST_NTABTYP (as table of LINEITEM_OBJTYP))
LINEITEMNO
STOCK_REF
QUANTITY
DISCOUNT
Number NUMBER
Reference STOCKITEM_OBJTYP
Number NUMBER
Number NUMBER
Refers to a row of the table
Table STOCK_OBJTAB (of STOCKITEM_OBJTYP)
STOCKNO
PRICE
TAXRATE
Number NUMBER
Number NUMBER
Number NUMBER
PK

Oracle does not support a referential constraint specification for storage tables. In this situation, specifying the SCOPE clause for a REF column is useful. In general, specifying scope or referential constraints for REF columns has several benefits:

- It saves storage space because it allows Oracle to store just the row object's unique identifier as the REF value in the column.
- It enables an index to be created on the storage table's REF column.
- It allows Oracle to rewrite queries containing dereferences of these REFs as joins involving the referenced table.

At this point, all of the tables for the purchase order application are in place. The next section shows how to operate on these tables.

Figure 9–10 Object Relational Representation of Table PurchaseOrder_objtab

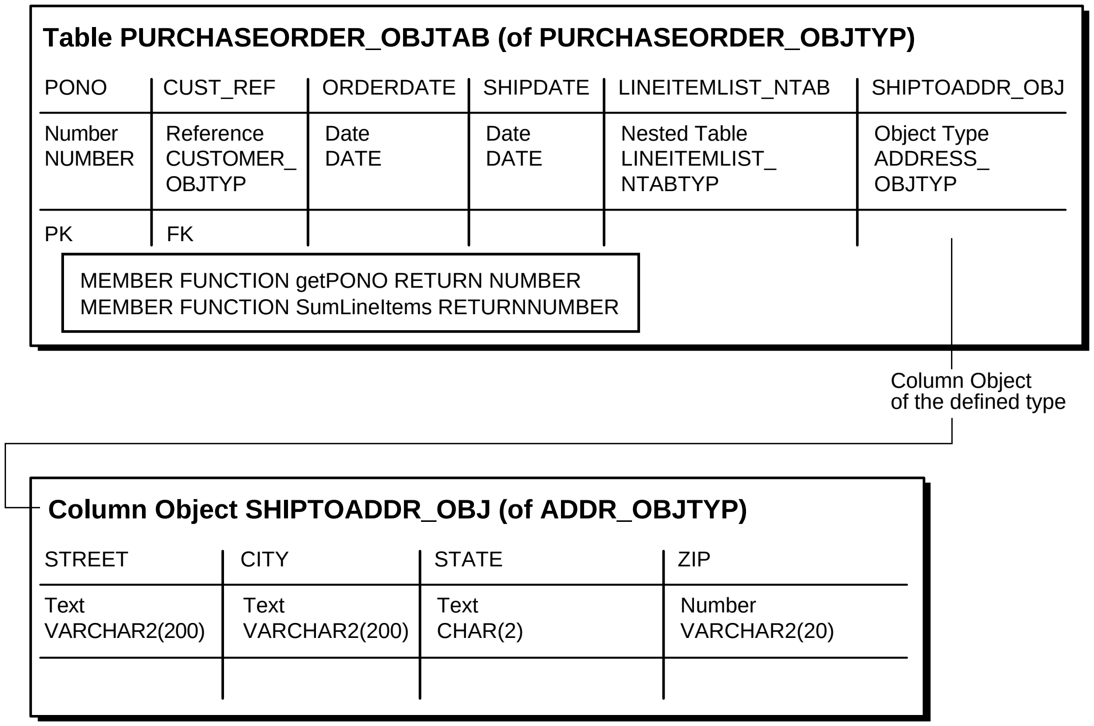


Table PURCHASEORDER_OBJTAB (of PURCHASEORDER_OBJTYP)

PONO	CUST_REF	ORDERDATE	SHIPDATE	LINEITEMLIST_NTAB	SHIPTOADDR_OBJ
Number NUMBER	Reference CUSTOMER_ OBJTYP	Date DATE	Date DATE	Nested Table LINEITEMLIST_ NTABTYP	Object Type ADDRESS_ OBJTYP
PK	FK				

MEMBER FUNCTION getPONO RETURN NUMBER
MEMBER FUNCTION SumLineItems RETURNNUMBER

Column Object SHIPTOADDR_OBJ (of ADDR_OBJTYP)

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

Inserting Values

Here is how to insert the same data into the object tables that we inserted earlier into relational tables. Notice how some of the values incorporate calls to the constructors for object types, to create instances of the types.

Stock_objtab

```
INSERT INTO Stock_objtab VALUES(1004, 6750.00, 2) ;
INSERT INTO Stock_objtab VALUES(1011, 4500.23, 2) ;
INSERT INTO Stock_objtab VALUES(1534, 2234.00, 2) ;
INSERT INTO Stock_objtab VALUES(1535, 3456.23, 2) ;
```

Customer_objtab

```
INSERT INTO Customer_objtab
  VALUES (
    1, 'Jean Nance',
    Address_objtyp('2 Avocet Drive', 'Redwood Shores', 'CA', '95054'),
    PhoneList_vartyp('415-555-1212')
    ) ;

INSERT INTO Customer_objtab
  VALUES (
    2, 'John Nike',
    Address_objtyp('323 College Drive', 'Edison', 'NJ', '08820'),
    PhoneList_vartyp('609-555-1212','201-555-1212')
    ) ;
```

PurchaseOrder_objtab

```
INSERT INTO PurchaseOrder_objtab
  SELECT  1001, REF(C),
          SYSDATE, '10-MAY-1999',
          LineItemList_ntabtyp(),
          NULL
   FROM   Customer_objtab C
   WHERE  C.CustNo = 1 ;
```

The preceding statement constructs a `PurchaseOrder_objtyp` object with the following attributes:

```
  PONo                1001
  Cust_ref            REF to customer number 1
  OrderDate           SYSDATE
  ShipDate            10-MAY-1999
  LineItemList_ntab   an empty LineItem_ntabtyp
  ShipToAddr_obj      NULL
```

The statement uses a query to construct a `REF` to the row object in the `Customer_objtab` object table that has a `CustNo` value of `1`.

The following statement uses a TABLE expression to identify the nested table as the target for the insertion, namely the nested table in the LineItemList_ntab column of the row object in the PurchaseOrder_objtab table that has a PONo value of 1001.

Note: The "flattened subquery" or "THE (subquery)" expression supported in Oracle release 8.0 to identify a nested table is now deprecated in favor of TABLE expression syntax like that shown in the following example.

```
INSERT INTO TABLE (
  SELECT  P.LineItemList_ntab
   FROM   PurchaseOrder_objtab P
   WHERE  P.PONo = 1001
  )
  SELECT  01, REF(S), 12, 0
   FROM   Stock_objtab S
   WHERE  S.StockNo = 1534 ;
```

The preceding statement inserts a line item into the nested table identified by the TABLE expression. The inserted line item contains a REF to the row object with a StockNo value of 1534 in the object table Stock_objtab.

The following statements follow the same pattern as the previous ones:

```
INSERT INTO PurchaseOrder_objtab
  SELECT  2001, REF(C),
          SYSDATE, '20-MAY-1997',
          LineItemList_ntabtyp(),
          Address_objtyp('55 Madison Ave','Madison','WI','53715')
   FROM   Customer_objtab C
   WHERE  C.CustNo = 2 ;

INSERT INTO TABLE (
  SELECT  P.LineItemList_ntab
   FROM   PurchaseOrder_objtab P
   WHERE  P.PONo = 1001
  )
  SELECT  02, REF(S), 10, 10
   FROM   Stock_objtab S
   WHERE  S.StockNo = 1535 ;
```

```
INSERT INTO TABLE (
  SELECT  P.LineItemList_ntab
   FROM   PurchaseOrder_objtab P
   WHERE  P.PONo = 2001
  )
  SELECT  10, REF(S), 1, 0
   FROM   Stock_objtab S
   WHERE  S.StockNo = 1004 ;

INSERT INTO TABLE (
  SELECT  P.LineItemList_ntab
   FROM   PurchaseOrder_objtab P
   WHERE  P.PONo = 2001
  )
  VALUES(11, (SELECT REF(S)
    FROM  Stock_objtab S
    WHERE S.StockNo = 1011), 2, 1) ;
```

Querying

The following query statement implicitly invokes a comparison method. It shows how Oracle orders objects of type `PurchaseOrder_objtyp` using that type's comparison method:

```
SELECT  p.PONo
 FROM   PurchaseOrder_objtab p
 ORDER BY VALUE(p) ;
```

Oracle invokes the map method `getPONo` for each `PurchaseOrder_objtyp` object in the selection. Because that method returns the object's `PONo` attribute, the selection produces a list of purchase order numbers in ascending numerical order.

The following queries correspond to the queries executed under the relational model.

Customer and Line Item Data for Purchase Order 1001

```
SELECT  DEREF(p.Cust_ref), p.ShipToAddr_obj, p.PONo,
        p.OrderDate, LineItemList_ntab
 FROM   PurchaseOrder_objtab p
 WHERE  p.PONo = 1001 ;
```

Total Value of Each Purchase Order

```
SELECT   p.PONo, p.sumLineItems()
 FROM    PurchaseOrder_objtab p ;
```

Purchase Order and Line Item Data Involving Stock Item 1004

```
SELECT   po.PONo, po.Cust_ref.CustNo,
         CURSOR (
           SELECT  *
            FROM   TABLE (po.LineItemList_ntab) L
            WHERE  L.Stock_ref.StockNo = 1004
           )
 FROM    PurchaseOrder_objtab po ;
```

The 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 ;
```

Average Discount across all Line Items of all Purchase Orders

This request requires querying the rows of all `LineItemList_ntab` nested tables of all `PurchaseOrder_objtab` rows. Again, unnesting is required:

```
SELECT   AVG(L.DISCOUNT)
 FROM    PurchaseOrder_objtab po, TABLE (po.LineItemList_ntab) L ;
```

Deleting

The following example has the same effect as the two deletions needed in the relational case (see "Deleting Data Under The Relational Model" on page 9-9). Here 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.

Delete Purchase Order 1001

```
DELETE
 FROM   PurchaseOrder_objtab
 WHERE  PONo = 1001 ;
```

Evolving User-Defined Types

Even a completed, fully built application tends to be a work in progress. Sometimes requirements change, forcing us to change an underlying object model or schema to adapt it to new circumstances, and sometimes we simply see 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.

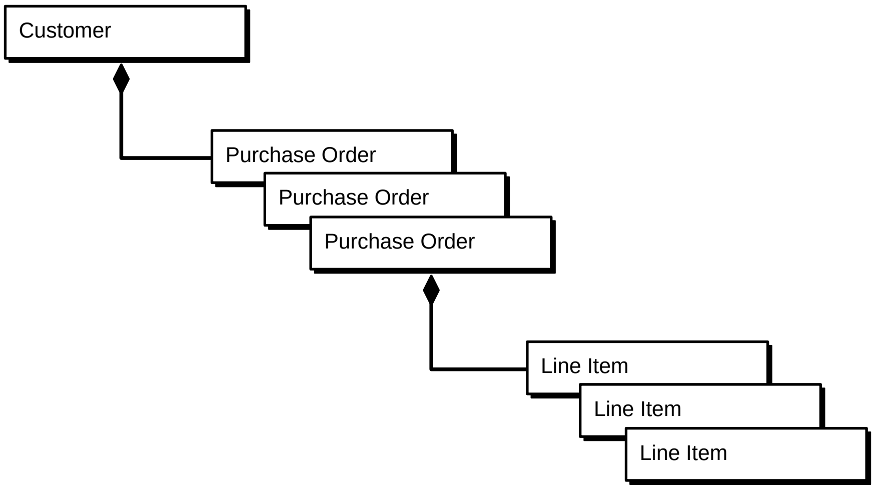
One way to do this is to change the `Customer_objtyp` so that information about a customer's purchase orders is included right in the object instance that represents that customer. In other words, we want to add an attribute for purchase order information to `Customer_objtyp`. To hold information about 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*, a user-defined 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, we need to do several steps:

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 REFs. More on this in the next section.

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:

```
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:

```
ALTER TYPE Customer_objtyp
  ADD ATTRIBUTE (PurchaseOrderList_ntab PurchaseOrderList_ntabtyp)
  INVALIDATE;
```

Everything about this ALTER TYPE statement is straightforward except the INVALIDATE option at the end. This option has to do with updating types and tables that refer to the Customer_objtyp.

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 is 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.

The reason that we used the riskier INVALIDATE option in our ALTER TYPE statement was not to save time but to prevent the system from automatically creating and naming storage tables for the new nested tables of purchase orders and line items that must be added to the Customer_objtab table. This object table is a dependent table of the Customer_objtyp type. CASCADE would cause the storage tables to be automatically created and would give them system-generated names.

We do not want to allow this because we need to be able to alter these tables to add scope for a REF column in each one. To do this we must set up the storage tables ourselves (with an ALTER TABLE statement on the table Customer_objtab) so that we have the opportunity to name them. Then, using the names we have given them, we can alter the storage tables with a couple more ALTER TABLE statements 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," as these are also called) into an unscoped column, you will get an error like this: "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.

Here is the ALTER TABLE statement that upgrades table Customer_objtab to take account of the change to the table's declared type and to set up storage tables for the new nested tables:

Note: The two ALTER TYPE statements preceding the ALTER TABLE are a workaround for a bug that exists in Release 9.2.0 at this writing (but may be fixed by the time you read this). The statements force the referenced dependent types of Customer_objtyp to recompile. They should recompile automatically.

```
ALTER TYPE PurchaseOrder_objtyp COMPILE;

ALTER TYPE PurchaseOrderList_ntabtyp COMPILE;
ALTER TABLE Customer_objtab UPGRADE
NESTED TABLE PurchaseOrderList_ntab STORE AS PO_List_nt
  (NESTED TABLE LineItemList_ntab STORE AS Items_List_nt);
```

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:

```
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);
```

Now there is just one more thing to do before we can insert purchase orders for customers in `Customer_objtab`: we must instantiate an actual nested table of `PurchaseOrderList_ntabtyp` 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 like this: "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:

```
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`:

```
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` in this case—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 (on the somewhat dubious assumption that the shipping address is always the same as the customer address…).

```
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.)

```
SQL> INSERT INTO TABLE (
  2    SELECT c.PurchaseOrderList_ntab
  3      FROM Customer_objtab c
  4      WHERE c.CustName = 'John Nike'
  5     )
  6    VALUES (1020, SYSDATE, SYSDATE + 1,
  7      LineItemList_ntabtyp(
  8        LineItem_objtyp(1, MAKE_REF(Stock_objtab, 1004), 1, 0),
  9        LineItem_objtyp(2, MAKE_REF(Stock_objtab, 1011), 3, 5),
 10        LineItem_objtyp(3, MAKE_REF(Stock_objtab, 1535), 2, 10)
 11       )
 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:

```
VALUES (
  PurchaseOrder_objtyp(1020, 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:

```
SELECT c.PurchaseOrderList_ntab
  FROM Customer_objtab c
  WHERE CustName = 'John Nike';
```

The query produces a result like this:

```
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
B8C8340000001426010001000100290000000000090600812A00078401FE0000000B03C20B050000
0000000000000000000000000000000000, 1, 0), LINEITEM_OBJTYP(11, 00004A038A00468ED
552CE6A5803ACE034080020B8C8340000001426010001000100290000000000090600812A0007840
1FE0000000B03C20B0C00000000000000000000000000000000000000, 2, 1))), PURCHASEORDE
R_OBJTYP(1020, '25-SEP-01', '26-SEP-01', LINEITEMLIST_NTABTYP(LINEITEM_OBJTYP(1,
 00004A038A00468ED552CE6A5803ACE034080020B8C83400000014260100010001002900000000
0090600812A00078401FE0000000B03C20B0500000000000000000000000000000000000000, 1,
0), LINEITEM_OBJTYP(2, 00004A038A00468ED552CE6A5803ACE034080020B8C83400000014260
100010001002900000000000090600812A00078401FE0000000B03C20B0C000000000000000000000
00000000000000000, 3, 5), LINEITEM_OBJTYP(3, 00004A038A00468ED552CE6A5803ACE0340
80020B8C8340000001426010001000100290000000000090600812A00078401FE0000000B03C2102
400000000000000000000000000000000000000, 2, 10))))
```

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 following query selects the PO number, order date, and shipdate for all purchase orders belonging to John Nike:

```
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:

```
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.

```
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 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 Also: "Type Evolution" in Chapter 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.

```
INSERT INTO Customer_objtab
  VALUES (
    3, 'William Kidd',
    Address_objtyp('43 Harbor Drive', 'Redwood Shores', 'CA', '95054'),
    PhoneList_vartyp('415-555-1212'),
    PurchaseOrderList_ntabtyp()
  );
```

The `VALUES` clause contains data for each `Customer_objtyp` attribute but omits the `Customer_objtyp` constructor. The constructor is optional here because the declared type of the table is `Customer_objtyp`. For the nested table attribute, the constructor `PurchaseOrderList_ntabtyp()` creates an empty nested table, but no data is specified for any purchase orders.

Here is a statement that inserts a new corporate customer in the same table. Note the use of the constructor `Corp_Customer_objtyp()` and the extra data value `531` for the account manager ID:

```
INSERT INTO Customer_objtab
  VALUES (
    Corp_Customer_objtyp(        -- Subtype requires a constructor
      4, 'Edward Teach',
      Address_objtyp('65 Marina Blvd', 'San Francisco', 'CA', '94777'),
      PhoneList_vartyp('415-555-1212', '416-555-1212'),
      PurchaseOrderList_ntabtyp(), 531
    )
  );
```

The following statements insert a purchase order for each of the two new customers. Unlike the statements that insert the new customers, the two statements that insert purchase orders are structurally the same (except for the number of line items in the purchase orders):

```
          -- Insert PO 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)
     )
   );

          -- Insert PO 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 particuar 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.

Selecting all customers

The following query uses the `VALUE()` function to select instances of every kind of customer in the table:

```
SELECT VALUE(c)
  FROM Customer_objtab c;
```

Selecting All Corporate Customers (and Their Subtypes)

The following query 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:

```
SELECT VALUE(c)
  FROM Customer_objtab c
  WHERE VALUE(c) IS OF (Corp_Customer_objtyp);
```

Selecting All Corporate Customers (No Subtypes)

This query is identical 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`:

```
SELECT p.PONo
  FROM Customer_objtab c, TABLE(c.PurchaseOrderList_ntab) p
  WHERE VALUE(c) IS OF (ONLY Corp_Customer_objtyp);
```

Selecting PONo Just for Corporate Customers

The following query 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:

```
SELECT p.PONo
  FROM Customer_objtab c, TABLE(c.PurchaseOrderList_ntab) p
  WHERE VALUE(c) IS OF (Corp_Customer_objtyp);
```

Selecting a Subtype Attribute

The following query 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 this 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:

```
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 here 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 following one will return the error "invalid column name." 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.

```
    -- Returns error, "invalid column name" for Account_mgr_id

SELECT CustName, Account_mgr_id
  FROM Customer_objtab
  WHERE VALUE(c) IS OF (ONLY Corp_Customer_objtyp);
```

Discovering the (Sub)Type of Each Instance

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 following query 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:

```
SELECT c.CustName, u.TYPE_NAME
  FROM Customer_objtab c, USER_TYPES u
  WHERE SYS_TYPEID(VALUE(c)) = u.TYPEID;
```

```
CUSTNAME
--------------------------------------------------------------------------------
TYPE_NAME
------------------------------
Jean Nance
CUSTOMER_OBJTYP

John Nike
CUSTOMER_OBJTYP

William Kidd
CUSTOMER_OBJTYP

Edward Teach
CORP_CUSTOMER_OBJTYP
```

See Also: "Functions and Predicates Useful with Objects" in Chapter 2 for more on `SYS_TYPEID()`, `VALUE()`, and `TREAT()`

Manipulating Objects with Oracle Objects for OLE

On Windows systems, you can use Oracle Objects for OLE (OO4O) to write object-oriented database programs in Visual Basic or other environments that support the COM protocol, such as Excel.

The following examples all begin with a similar header section that connects to the database. Then each shows how to perform a different operation on object data.

Selecting Data

Here is an event handler for a button that performs a `SELECT` operation.

- We get a set of rows from the database, each row containing some relational columns and some columns that are objects.
- Using the name of the `CUSTREF` column, we retrieve its value, which is an object.
- Then we can use the dot notation to access the attributes of the object. We define the variable as a generic object type, `OraObject`. After it is instantiated with a real object, it takes on the properties of the corresponding object type.

```
Private Sub obj_select_Click()
 Dim OO4OSession As OraSession
 Dim InvDB As OraDatabase
 Dim PurchaseOrder As OraDynaset
 Dim CustomerInfo As OraRef
 Dim LineItemsList As OraCollection
 Dim LineItem As OraObject
 Dim ShipToAddr As OraObject
 Dim StockInfo As OraRef
 Dim CustomerAddr As OraObject

 'Create the OraSession Object.
 Set OO4OSession = CreateObject("OracleInProcServer.XOraSession")

 'Create the OraDatabase Object by opening a connection to Oracle.
 Set InvDB = OO4OSession.OpenDatabase("exampledb", "scott/tiger", 0&)

 'Select from purchase_tab
 Set PurchaseOrder = InvDB.CreateDynaset("select * from purchase_tab", 0&)

 'Get the custref attribute from PurchaseOrder
 Set CustomerInfo = PurchaseOrder.Fields("custref").Value

 ' Accessing attributes CustomerInfo object

 'Display custno,custname,phonelist attibutes of CustomerInfo
 MsgBox CustomerInfo.custno
 MsgBox CustomerInfo.custname

 'Get address and phonelist attibutes of CustomerInfo
 Set CustomerAddr = CustomerInfo.Address

 'Display all the atributes of CustomerAddr
 MsgBox CustomerAddr.Street
 MsgBox CustomerAddr.State
 MsgBox CustomerAddr.Zip

 '  Accessing elements of LineItemsList Object

 'Get line_item_list attribute from PurchaseOrder
 Set LineItemsList = PurchaseOrder.Fields("line_item_list").Value

 'Get LineItem object element from LineItemList collection
  Set LineItem = LineItemsList(1)

 'Display lineitemno,quantity,discount attibutes
 MsgBox LineItem.lineitemno
```

```
 MsgBox LineItem.quantity
 MsgBox LineItem.discount

 'Access stockref attribute of LineItem
 Set StockInfo = LineItem.Stockref

 'Display stockno,cost,tax_code of StockInfo
 MsgBox StockInfo.stockno
 MsgBox StockInfo.cost
 MsgBox StockInfo.tax_code

End Sub
```

Inserting Data

Here is a program that retrieves a set of rows from the database, then adds a new row.

- We create some objects of the appropriate object types.
- We populate the objects with sample values.
- We create a new row for the purchase order table, and fill in the values for its columns. The columns that are not objects can be set directly. The columns that are objects must be set using the `VALUE` field.

```
Dim OO4OSession As OraSession
Dim InvDB As OraDatabase
Dim PurchaseOrder As OraDynaset
Dim CustomerInfo As OraRef
Dim LineItemsList As OraCollection
Dim LineItem As OraObject
Dim ShipToAddr As OraObject
Dim StockInfo As OraRef
Dim CustomerAddr As OraObject

 'Create the OraSession Object.
 Set OO4OSession = CreateObject("OracleInProcServer.XOraSession")

 'Create the OraDatabase Object by opening a connection to Oracle.
 Set InvDB = OO4OSession.OpenDatabase("exampledb", "scott/tiger", 0&)

 'Select from purchase_tab
 Set PurchaseOrder = InvDB.CreateDynaset("select * from purchase_tab", 0&)
 '  Step 1  - Creating CustomerInfo ref object

 'select a ref from customer_tab for custono 2
 Set CustomerDyn = InvDB.CreateDynaset("select REF(C) from customer_tab c
where c.custno  = 2", 0&)

 'get the CustomerInfo ref object
 Set CustomerInfo = CustomerDyn.Fields(0).Value

 ' Step 2  - Creating LineItemsList object

 ' Create a new line_items_list object
 Set LineItemsList = InvDB.CreateOraObject("line_item_list_t")

 ' Create a new line_items object
 Set LineItem = InvDB.CreateOraObject("line_item_t")

 'set attributes of LineItem object
 LineItem.lineitemno = 2
 LineItem.quantity = 15
 LineItem.discount = 30
 LineItem.Stockref = Null

 'set the LineItem to first element of LineItemList
 LineItemsList(1) = LineItem

 ' Step 3 - Creating ShipToAddr object

 ' create a shiptoaddr object
 Set ShipToAddr = InvDB.CreateOraObject("address_t")

 'set the attributes of ShipToAddr Object
 ShipToAddr.city = "Belmont"
 ShipToAddr.Street = "Continentals way"
 ShipToAddr.Zip = "94002"
 ShipToAddr.State = "CA"

 ' Start the AddNew operation on PurchaseOrder dynaset

 PurchaseOrder.AddNew

  PurchaseOrder.Fields("pono").Value = 1002
  PurchaseOrder.Fields("orderdate").Value = "5/15/99"
  PurchaseOrder.Fields("shipdate").Value = "6/15/99"
  'set the custref field to CustomerInfo object created in step1
  PurchaseOrder.Fields("custref").Value = CustomerInfo

  'set the line_item_list  field to LineItemslist object created in step2
  PurchaseOrder.Fields("line_item_list").Value = LineItemsList

  'set the shiptoaddr  field to ShipToAddr  object created in step3
  PurchaseOrder.Fields("shiptoaddr").Value = ShipToAddr

 ' Call the update method on Purchaseorder Dynaset which inserts a new row
 ' in purchase_tab table

 PurchaseOrder.Update
```

Updating Data

Here is a program that retrieves some rows from the database, then updates a specific one.

- We select the purchase order using a query that returns a single row.
- We get individual data items to manipulate from other tables and from the original purchase order.
- We lock the purchase order row for updating, and put in the new values.

```
Dim OO4OSession As OraSession
Dim InvDB As OraDatabase
Dim PurchaseOrder As OraDynaset
Dim CustomerInfo As OraRef
Dim LineItemsList As OraCollection
Dim LineItem As OraObject
Dim ShipToAddr As OraObject
Dim StockInfo As OraRef
Dim CustomerAddr As OraObject

'Create the OraSession Object.
Set OO4OSession = CreateObject("OracleInProcServer.XOraSession")

'Create the OraDatabase Object by opening a connection to Oracle.
Set InvDB = OO4OSession.OpenDatabase("exampledb", "scott/tiger", 0&)

'Select from purchase_tab for pono 1002
Set PurchaseOrder = InvDB.CreateDynaset("select * from purchase_tab where
pono = 1002", 0&)
```

```
'Create a StockInfo from stock_tab for stockno 1535
Set StockDyn = InvDB.CreateDynaset("select REF(s) from stock_tab s where
s.stockno = 1535", 0&)
Set StockInfo = StockDyn.Fields(0).Value

'Get line_item_list attribute from PurchaseOrder
Set LineItemsList = PurchaseOrder.Fields("line_item_list").Value

'Get LineItem object element from LineItemList collection
Set LineItem = LineItemsList(1)

'Start the edit operation on PurchaseOrder dynaset
PurchaseOrder.Edit

' Set the StockInfo object created in Step1 to stockref attribute
' of LineItem
LineItem.Stockref = StockInfo
PurchaseOrder.Update
```

Calling a Method Function

Here is a program that retrieves a purchase order, and calls its member function `TOTAL_VALUE` to sum the cost of the line items that are part of the purchase order.

- We select one row from the purchase order table. Notice we select the `VALUE` so that the result comes back as an object.
- We get a pointer to the purchase order object (the zero'th column of the result row). Later this pointer is passed to a PL/SQL stored procedure, to simulate the `SELF` pointer in Java or C++ methods.
- We build a list of parameters corresponding to the implicit `SELF` parameter and the return value of the method function. For each, we specify the bind variable, its value, its mode, and its type.
- We call the stored procedure corresponding to the method function, storing the result in the `TOTALVALUE` bind variable.
- To use the result, we retrieve the return value from the parameter list.

```
Dim OO4OSession As OraSession
Dim InvDB As OraDatabase
Dim PurchaseOrderObj As OraDynaset
```

```
'Create the OraSession Object.
Set OO4OSession = CreateObject("OracleInProcServer.XOraSession")

'Create the OraDatabase Object by opening a connection to Oracle.
Set InvDB = OO4OSession.OpenDatabase("exampledb", "scott/tiger", 0&)

'Select from purchase_tab
Set PurchaseOrderDyn = InvDB.CreateDynaset("select VALUE(p)  from
purchase_tab p where p.pono = 1001", 0&)

'Get the PurchaseOrderObj
Set PurchaseOrderObj = PurchaseOrderDyn.Fields(0).Value

'Create a OraParameter object for purchase_order_t object and set it to
PurchaseOrder
InvDB.Parameters.Add "PURCHASEORDER", PurchaseOrderObj, ORAPARM_BOTH,
ORATYPE_OBJECT, "PURCHASE_ORDER_T"

'Create a parameter for total_value return
InvDB.Parameters.Add "TOTALVALUE", "", ORAPARM_OUTPUT

'Execute a member method
InvDB.ExecuteSQL ("BEGIN :TOTALVALUE :=
PURCHASE_ORDER_T.TOTAL_VALUE(:PURCHASEORDER); END;")

'Display the totalvalue
MsgBox InvDB.Parameters("TOTALVALUE").Value
```

Index