**CHAPTER 12  
Database Objects**

In this chapter, you will do the following:

Image Learn about objects in the database

Image Learn how to create object types containing attributes and methods

Image Use object types to define column objects and object tables

Image Create and manipulate objects in SQL and PL/SQL

Image Learn how a type may inherit from another type and create hierarchies of types

Image Define your own constructors to set the attributes of an object

Image See how to override a method in one type with a method from another type

**INTRODUCING OBJECTS**

Object-oriented programming languages such as Java, C++, and C# allow you to define classes, and these classes act as templates from which you can create objects. Classes define attributes and methods; attributes are used to store an object’s state, and methods are used to model an object’s behaviors.

With the release of Oracle Database 8, objects became available within the database, and object features have been improved upon in subsequent product releases. The availability of objects in the database was a major breakthrough because they enable you to define your own classes, known as *object types*, in the database. Like classes in Java and C#, database object types can contain attributes and methods. Object types are also sometimes known as user-defined types.

A simple example of an object type would be a type that represents a product. This object type could contain attributes for the product’s name, description, price, and, in the case of a product that is perishable, the number of days the product can sit on the shelf before it must be thrown away. This product object type could also contain a method that returns the sell-by date of the product, based on the shelf life of the product and the current date. Another example of an object type is one that represents a person; this object type could store attributes for the person’s first name, last name, date of birth, and address; the person’s address could itself be represented by an object type, and it could store things like the street, city, state, and zip code. In this chapter you’ll see examples of object types that represent a product, person, and address. You’ll also see how to create tables from those object types, populate those tables with actual objects, and manipulate those objects in SQL and PL/SQL.

I’ve provided an SQL\*Plus script named object\_schema.sql in the SQL directory, which creates a user named object\_user with a password of object\_password. This script also creates the types and tables, performs the various INSERT statements, and creates the PL/SQL code shown in the first part of this chapter. You must run this script while logged in as a user with the required privileges to create a new user with the CONNECT, RESOURCE, and CREATE PUBLIC SYNONYM privileges; I log in as the system user on my database to run the scripts. After the script completes, you will be logged in as object\_user.

**CREATING OBJECT TYPES**

You create an object type using the CREATE TYPE statement. The following example uses the CREATE TYPEstatement to create an object type named t\_address. This object type is used to represent an address and contains four attributes named street, city, state, and zip:

CREATE TYPE t\_address AS OBJECT (

street VARCHAR2(15),

city VARCHAR2(15),

state CHAR(2),

zip VARCHAR2(5)

);

/

The example shows that each attribute is defined using a database type. For example, street is defined as VARCHAR2(15). As you’ll see shortly, the type of an attribute can itself be an object type.

The next example creates an object type named t\_person; notice that t\_person has an attribute named address, which is of type t\_address:

CREATE TYPE t\_person AS OBJECT (

id INTEGER,

first\_name VARCHAR2(10),

last\_name VARCHAR2(10),

dob DATE,

phone VARCHAR2(12),

address t\_address

);

/

The following example creates an object type named t\_product that will be used to represent products; notice that this type declares a function named get\_sell\_by\_date() using the MEMBER FUNCTION clause:

CREATE TYPE t\_product AS OBJECT (

id INTEGER,

name VARCHAR2(15),

description VARCHAR2(22),

price NUMBER(5, 2),

days\_valid INTEGER,

-- get\_sell\_by\_date() returns the date by which the

-- product must be sold

MEMBER FUNCTION get\_sell\_by\_date RETURN DATE

);

/

Because t\_product contains a method declaration, a *body* for t\_product must also be created. The body contains the actual code for the method, and the body is created using the CREATE TYPE BODY statement. The following example creates the body for t\_product; notice the body contains the code for the get\_sell\_by\_date() function.

CREATE TYPE BODY t\_product AS

-- get\_sell\_by\_date() returns the date by which the

-- product must be sold

MEMBER FUNCTION get\_sell\_by\_date RETURN DATE IS

v\_sell\_by\_date DATE;

BEGIN

-- calculate the sell by date by adding the days\_valid attribute

-- to the current date (SYSDATE)

SELECT days\_valid + SYSDATE

INTO v\_sell\_by\_date

FROM dual;

-- return the sell by date

RETURN v\_sell\_by\_date;

END;

END;

/

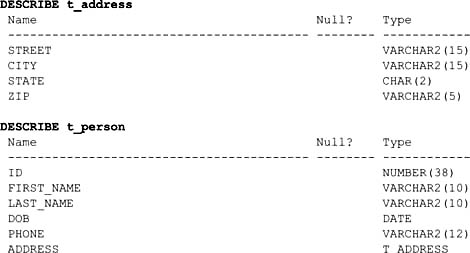
As you can see, get\_sell\_by\_date() calculates and returns the date by which the product must be sold; it does this by adding the days\_valid attribute to the current date returned by the built-in database SYSDATE()function.

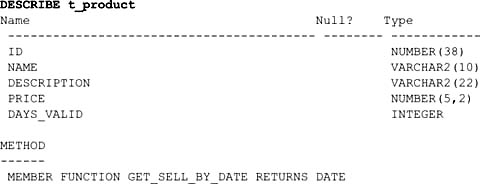
You can also create a public synonym for a type, which enables all users to see the type and use it to define columns in their own tables. The following example creates a public synonym named t\_pub\_product for t\_product:

CREATE PUBLIC SYNONYM t\_pub\_product FOR t\_product;

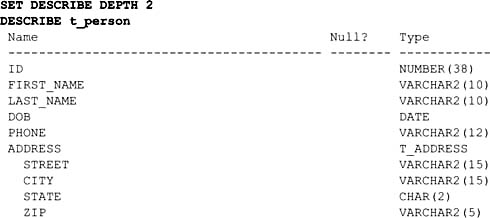
**USING DESCRIBE TO GET INFORMATION ON OBJECT TYPES**

You can use the DESCRIBE command to get information on an object type. The following examples show the t\_address, t\_person, and t\_product types:





You can set the depth to which DESCRIBE will show information for embedded types using SET DESCRIBE DEPTH. The following example sets the depth to 2 and then describes t\_person again; notice that the attributes of address are displayed, which is an embedded object of type t\_address:



**USING OBJECT TYPES IN DATABASE TABLES**

Now that you’ve seen how to create object types, let’s look at how you use these types in database tables. You can use an object type to define an individual column in a table, and the objects subsequently stored in that column are known as *column objects*. You can also use an object type to define an entire row in a table; the table is then known as an *object table*. Finally, you can use an *object reference* to access an individual row in an object table; an object reference is similar to a pointer in C++. You’ll see examples of column objects, object tables, and object references in this section.

**Column Objects**

The following example creates a table named products that contains a column named product of type t\_product; the table also contains a column named quantity\_in\_stock, which is used to store the number of those products currently in stock:

CREATE TABLE products (

product t\_product,

quantity\_in\_stock INTEGER

);

When adding a row to this table, you must use a *constructor* to supply the attribute values for the new t\_product object; as a reminder, the t\_product type was created using the following statement:

CREATE TYPE t\_product AS OBJECT (

id INTEGER,

name VARCHAR2(10),

description VARCHAR2(22),

price NUMBER(5, 2),

days\_valid INTEGER,

-- declare the get\_sell\_by\_date() member function,

-- get\_sell\_by\_date() returns the date by which the

-- product must be sold

MEMBER FUNCTION get\_sell\_by\_date RETURN DATE

);

/

A constructor is a built-in method for the object type, and it has the same name as the object type; the constructor accepts parameters that are used to set the attributes of the new object. The constructor for the t\_product type is named t\_product and accepts five parameters, one to set each of the attributes; for example, t\_product(1, pasta, 20 oz bag of pasta, 3.95, 10) creates a new t\_product object and sets its id to 1, name to pasta, description to 20 oz bag of pasta, price to 3.95, and days\_valid to 10.

The following INSERT statements add two rows to the products table; notice the use of the t\_productconstructor to supply the attribute values for the product column objects:

INSERT INTO products (

product,

quantity\_in\_stock

) VALUES (

t\_product(1, 'pasta', '20 oz bag of pasta', 3.95, 10),

50

);

INSERT INTO products (

product,

quantity\_in\_stock

) VALUES (

t\_product(2, ’sardines', '12 oz box of sardines', 2.99, 5),

25

);

The following query retrieves these rows from the products table; notice that the product column objects' attributes are displayed within a constructor for t\_product:

**SELECT \***

**FROM products;**

PRODUCT(ID, NAME, DESCRIPTION, PRICE, DAYS\_VALID)

----------------------------------------------------------

QUANTITY\_IN\_STOCK

-----------------

T\_PRODUCT(1, 'pasta', '20 oz bag of pasta', 3.95, 10)

50

T\_PRODUCT(2, ’sardines', '12 oz box of sardines', 2.99, 5)

25

You can also retrieve an individual column object from a table; to do this, you must supply a table alias through which you select the object. The following query retrieves product #1 from the products table; notice the use of the table alias p for the products table, through which the product object’s id attribute is specified in the WHERE clause:

**SELECT p.product**

**FROM products p**

**WHERE p.product.id = 1;**

PRODUCT(ID, NAME, DESCRIPTION, PRICE, DAYS\_VALID)

-----------------------------------------------------

T\_PRODUCT(1, 'pasta', '20 oz bag of pasta', 3.95, 10)

The next query explicitly includes the product object’s id, name, price, and days\_valid attributes in the SELECT statement, plus the quantity\_in\_stock:

**SELECT p.product.id, p.product.name**,

**p.product.price, p.product.days\_valid, p.quantity\_in\_stock**

**FROM products p**

**WHERE p.product.id = 1;**

PRODUCT.ID PRODUCT.NA PRODUCT.PRICE PRODUCT.DAYS\_VALID QUANTITY\_IN\_STOCK

---------- ---------- ------------- ------------------ -----------------

1 pasta 3.95 10 50

The t\_product object type contains a function named get\_sell\_by\_date(), which calculates and returns the date by which the product must be sold. The function does this by adding the days\_valid attribute to the current date, which is obtained from the database using the SYSDATE() function. You can call the get\_sell\_by\_date() function using a table alias, as shown in the following query that uses the table alias pfor the products table:

**SELECT p.product.get\_sell\_by\_date()**

**FROM products p;**

P.PRODUCT

---------

19-JUN-07

13-JUN-07

Of course, if you run this query your dates will be different, because they are calculated using SYSDATE(), which returns the current date and time.

The following UPDATE statement modifies the description of product #1; notice that the table alias p is used again:

**UPDATE products p**

**SET p.product.description = '30 oz bag of pasta'**

**WHERE p.product.id = 1;**

1 row updated.

The following DELETE statement removes product #2:

**DELETE FROM products p**

**WHERE p.product.id = 2;**

1 row deleted.

**ROLLBACK;**

Image

**NOTE**  
*If you run these* UPDATE *and* DELETE *statements, make sure you execute the* ROLLBACK *so that your example data matches that shown in the rest of this chapter*.

**Object Tables**

You can use an object type to define an entire table, and such a table is known as an object table. The following example creates an object table named object\_products, which stores objects of type t\_product; notice the use of the OF keyword to identify the table as an object table of type t\_product:

CREATE TABLE object\_products OF t\_product;

When inserting a row into an object table, you can choose whether to use a constructor to supply attribute values or to supply the values in the same way that you would supply column values in a relational table. The following INSERT statement adds a row to the object\_products table using the constructor for t\_product:

INSERT INTO object\_products VALUES (

t\_product(1, 'pasta', '20 oz bag of pasta', 3.95, 10)

);

The next INSERT statement omits the constructor for t\_product; notice that the attribute values for t\_product are supplied in the same way that columns would be in a relational table:

INSERT INTO object\_products (

id, name, description, price, days\_valid

) VALUES (

2, ’sardines', '12 oz box of sardines', 2.99, 5

);

The following query retrieves these rows from the object\_products table:

**SELECT \***

**FROM object\_products;**

ID NAME DESCRIPTION PRICE DAYS\_VALID

---------- ---------- ---------------------- ---------- ----------

1 pasta 20 oz bag of pasta 3.95 10

2 sardines 12 oz box of sardines 2.99 5

You can also specify individual object attributes in a query; for example, by doing this:

**SELECT id, name, price**

**FROM object\_products op**

**WHERE id = 1;**

ID NAME PRICE

---------- ---------- ----------

1 pasta 3.95

or this:

**SELECT op.id, op.name, op.price**

**FROM object\_products op**

**WHERE op.id = 1;**

ID NAME PRICE

---------- ---------- ----------

1 pasta 3.95

You can use the built-in Oracle database VALUE() function to select a row from an object table. VALUE()treats the row as an actual object and returns the attributes for the object within a constructor for the object type. VALUE() accepts a parameter containing a table alias, as shown in the following query:

**SELECT VALUE(op)**

**FROM object\_products op;**

VALUE(OP)(ID, NAME, DESCRIPTION, PRICE, DAYS\_VALID)

----------------------------------------------------------

T\_PRODUCT(1, 'pasta', '20 oz bag of pasta', 3.95, 10)

T\_PRODUCT(2, ’sardines', '12 oz box of sardines', 2.99, 5)

You can also add an object attribute after VALUE():

**SELECT VALUE(op).id, VALUE(op).name, VALUE(op).price**

**FROM object\_products op;**

VALUE(OP).ID VALUE(OP). VALUE(OP).PRICE

------------ ---------- ---------------

1 pasta 3.95

2 sardines 2.99

The following UPDATE statement modifies the description of product #1:

**UPDATE object\_products**

**SET description = '25 oz bag of pasta'**

**WHERE id = 1;**

1 row updated.

The following DELETE statement removes product #2:

**DELETE FROM object\_products**

**WHERE id = 2;**

1 row deleted.

**ROLLBACK;**

Let’s take a look at a more complex object table. The following CREATE TABLE statement creates an object table named object\_customers, which stores objects of type t\_person:

CREATE TABLE object\_customers OF t\_person;

The t\_person type contains an embedded t\_address object; t\_person was created using the following statement:

CREATE TYPE t\_person AS OBJECT (

id INTEGER,

first\_name VARCHAR2(10),

last\_name VARCHAR2(10),

dob DATE,

phone VARCHAR2(12),

address t\_address

);

/

The following INSERT statements add two rows into object\_customers. The first INSERT uses constructors for t\_person and t\_address, while the second INSERT omits the t\_person constructor:

INSERT INTO object\_customers VALUES (

t\_person(1, 'John', 'Brown', '01-FEB-1955', '800-555-1211',

t\_address('2 State Street', 'Beantown', 'MA', '12345')

)

);

INSERT INTO object\_customers (

id, first\_name, last\_name, dob, phone,

address

) VALUES (

2, 'Cynthia', 'Green', '05-FEB-1968', '800-555-1212',

t\_address('3 Free Street', 'Middle Town', 'CA', '12345')

);

The following query retrieves these rows from the object\_customers table; notice that the attributes for the embedded address column object are displayed within the t\_address constructor:

**SELECT \***

**FROM object\_customers;**

ID FIRST\_NAME LAST\_NAME DOB PHONE

---------- ---------- ---------- --------- ------------

ADDRESS(STREET, CITY, STATE, ZIP)

--------------------------------------------------------

1 John Brown 01-FEB-55 800-555-1211

T\_ADDRESS('2 State Street', 'Beantown', 'MA', '12345')

2 Cynthia Green 05-FEB-68 800-555-1212

T\_ADDRESS('3 Free Street', 'Middle Town', 'CA', '12345')

The next query retrieves customer #1 from object\_customers; notice the use of the table alias oc through which the id attribute is specified in the WHERE clause:

**SELECT \***

**FROM object\_customers oc**

**WHERE oc.id = 1;**

ID FIRST\_NAME LAST\_NAME DOB PHONE

---------- ---------- ---------- --------- ------------

ADDRESS(STREET, CITY, STATE, ZIP)

------------------------------------------------------

1 John Brown 01-FEB-55 800-555-1211

T\_ADDRESS('2 State Street', 'Beantown', 'MA', '12345')

In the following query, a customer is retrieved based on the state attribute of the address column object:

**SELECT \***

**FROM object\_customers oc**

**WHERE oc.address.state = 'MA';**

ID FIRST\_NAME LAST\_NAME DOB PHONE

---------- ---------- ---------- --------- ------------

ADDRESS(STREET, CITY, STATE, ZIP)

------------------------------------------------------

1 John Brown 01-FEB-55 800-555-1211

T\_ADDRESS('2 State Street', 'Beantown', 'MA', '12345')

In the next query, the id, first\_name, and last\_name attributes of customer #1 are explicitly included in the SELECT statement, along with the attributes of the embedded address column object:

**SELECT oc.id, oc.first\_name, oc.last\_name**,

**oc.address.street, oc.address.city, oc.address.state, oc.address.zip**

**FROM object\_customers oc**

**WHERE oc.id = 1;**

ID FIRST\_NAME LAST\_NAME ADDRESS.STREET ADDRESS.CITY AD ADDRE

---------- ---------- ---------- --------------- --------------- -- -----

1 John Brown 2 State Street Beantown MA 12345

**Object Identifiers and Object References**

Each object in an object table has a unique *object identifier* (OID), and you can retrieve the OID for an object using the REF() function. For example, the following query retrieves the OID for customer #1 in the object\_customers table:

**SELECT REF(oc)**

**FROM object\_customers oc**

**WHERE oc.id = 1;**

REF(OC)

---------------------------------------------------------------------

0000280209D66AB93F991647649D78D08B267EE44858C7B9989D9D40689FB4DA92820

AFFE2010003280000

The long string of numbers and letters are the OID, which identifies the location of the object in the database. You can store an OID in an object reference and later access the object it refers to. An object reference, which is similar to a pointer in C++, points to an object stored in an object table using the OID. You may use object references to model relationships between object tables, and, as you’ll see later, you can use object references in PL/SQL to access objects.

You use the REF type to define an object reference; the following statement creates a table named purchasesthat contains two object reference columns named customer\_ref and product\_ref:

CREATE TABLE purchases (

id INTEGER PRIMARY KEY,

customer\_ref REF t\_person SCOPE IS object\_customers,

product\_ref REF t\_product SCOPE IS object\_products

);

The SCOPE IS clause restricts an object reference to point to objects in a specific table. For example, the customer\_ref column is restricted to point to objects in the object\_customers table only; similarly, the product\_ref column is restricted to point to objects in the object\_products table only.

As I mentioned earlier, each object in an object table has a unique object identifier (OID) that you can store in an object reference; you can retrieve an OID using the REF() function and store it in an object reference. For example, the following INSERT statement adds a row to the purchases table; notice that the REF() function is used in the queries to get the object identifiers for customer #1 and product #1 from the object\_customers and object\_products tables:

INSERT INTO purchases (

id,

customer\_ref,

product\_ref

) VALUES (

1,

(SELECT REF(oc) FROM object\_customers oc WHERE oc.id = 1),

(SELECT REF(op) FROM object\_products op WHERE op.id = 1)

);

This example records that customer #1 purchased product #1.

The following query selects the row from the purchases table; notice that the customer\_ref and product\_ref columns contain references to the objects in the object\_customers and object\_products tables:

**SELECT \***

**FROM purchases;**

ID

----------

CUSTOMER\_REF

---------------------------------------------------------------------

PRODUCT\_REF

---------------------------------------------------------------------

1

0000220208D66AB93F991647649D78D08B267EE44858C7B9989D9D40689FB4DA92820

AFFE2

0000220208662E2AB4256711D6A1B50010A4E7AE8A662E2AB2256711D6A1B50010A4E

7AE8A

You can retrieve the actual objects stored in an object reference using the DEREF() function, which accepts an object reference as a parameter and returns the actual object. For example, the following query uses DEREF() to retrieve customer #1 and product #1 through the customer\_ref and product\_ref columns of the purchases table:

**SELECT DEREF(customer\_ref), DEREF(product\_ref)**

**FROM purchases;**

DEREF(CUSTOMER\_REF)(ID, FIRST\_NAME, LAST\_NAME, DOB, PHONE,

ADDRESS(STREET, CITY,

----------------------------------------------------------

DEREF(PRODUCT\_REF)(ID, NAME, DESCRIPTION, PRICE, DAYS\_VALID)

------------------------------------------------------------

T\_PERSON(1, 'John', 'Brown', '01-FEB-55', '800-555-1211',

T\_ADDRESS('2 State Street', 'Beantown', 'MA', '12345'))

T\_PRODUCT(1, 'pasta', '20 oz bag of pasta', 3.95, 10)

The next query retrieves the customer’s first\_name and address.street attributes, plus the product’s name attribute:

**SELECT DEREF(customer\_ref).first\_name**,

**DEREF(customer\_ref).address.street, DEREF(product\_ref).name**

**FROM purchases;**

DEREF(CUST DEREF(CUSTOMER\_ DEREF(PROD

---------- --------------- ----------

John 2 State Street pasta

The following UPDATE statement modifies the product\_ref column to point to product #2:

**UPDATE purchases SET product\_ref = (**

**SELECT REF(op) FROM object\_products op WHERE op.id = 2**

**) WHERE id = 1;**

1 row updated.

The following query verifies this change:

**SELECT DEREF(customer\_ref), DEREF(product\_ref)**

**FROM purchases;**

DEREF(CUSTOMER\_REF)(ID, FIRST\_NAME, LAST\_NAME, DOB, PHONE,

ADDRESS(STREET, CITY,

----------------------------------------------------------

DEREF(PRODUCT\_REF)(ID, NAME, DESCRIPTION, PRICE, DAYS\_VALID)

------------------------------------------------------------

T\_PERSON(1, 'John', 'Brown', '01-FEB-55', '800-555-1211',

T\_ADDRESS('2 State Street', 'Beantown', 'MA', '12345'))

T\_PRODUCT(2, ’sardines', '12 oz box of sardines', 2.99, 5)

**Comparing Object Values**

You can compare the value of two objects in a WHERE clause of a query using the equality operator (=). For example, the following query retrieves customer #1 from the object\_customers table:

**SELECT oc.id, oc.first\_name, oc.last\_name, oc.dob**

**FROM object\_customers oc**

**WHERE VALUE(oc) =**

**t\_person(1, 'John', 'Brown', '01-FEB-1955', '800-555-1211'**,

**t\_address('2 State Street', 'Beantown', 'MA', '12345')**

**);**

ID FIRST\_NAME LAST\_NAME DOB

---------- ---------- ---------- ---------

1 John Brown 01-FEB-55

The next query retrieves product #1 from the object\_products table:

**SELECT op.id, op.name, op.price, op.days\_valid**

**FROM object\_products op**

**WHERE VALUE(op) = t\_product(1, 'pasta', '20 oz bag of pasta', 3.95, 10);**

ID NAME PRICE DAYS\_VALID

---------- ---------- ---------- ----------

1 pasta 3.95 10

You can also use the <> and IN operators in the WHERE clause:

**SELECT oc.id, oc.first\_name, oc.last\_name, oc.dob**

**FROM object\_customers oc**

**WHERE VALUE(oc) <>**

**t\_person(1, 'John', 'Brown', '01-FEB-1955', '800-555-1211'**,

**t\_address('2 State Street', 'Beantown', 'MA', '12345')**

**);**

ID FIRST\_NAME LAST\_NAME DOB

---------- ---------- ---------- ---------

2 Cynthia Green 05-FEB-68

**SELECT op.id, op.name, op.price, op.days\_valid**

**FROM object\_products op**

**WHERE VALUE(op) IN t\_product(1, 'pasta', '20 oz bag of pasta', 3.95, 10);**

ID NAME PRICE DAYS\_VALID

---------- ---------- ---------- ----------

1 pasta 3.95 10

If you want to use an operator like <, >, <=, >=, LIKE, or BETWEEN, you need to provide a map function for the type. A map function must return a single value of one of the built-in types that the database can then use to compare two objects. The value returned by the map function will be different for every object type, and you need to figure out what the best attribute, or concatenation of attributes, represents an object’s value. For example, with the t\_product type, I’d return the price attribute; with the t\_person type, I’d return a concatenation of the last\_name and first\_name attributes.

The following statements create a type named t\_person2 that contains a map function named get\_string(); notice that get\_string() returns a VARCHAR2 string containing a concatenation of the last\_name and first\_name attributes:

CREATE TYPE t\_person2 AS OBJECT (

id INTEGER,

first\_name VARCHAR2(10),

last\_name VARCHAR2(10),

dob DATE,

phone VARCHAR2(12),

address t\_address,

-- declare the get\_string() map function,

-- which returns a VARCHAR2 string

MAP MEMBER FUNCTION get\_string RETURN VARCHAR2

);

/

CREATE TYPE BODY t\_person2 AS

-- define the get\_string() map function

MAP MEMBER FUNCTION get\_string RETURN VARCHAR2 IS

BEGIN

-- return a concatenated string containing the

-- last\_name and first\_name attributes

RETURN last\_name || ' ' || first\_name;

END get\_string;

END;

/

As you’ll see shortly, the database will automatically call get\_string() when comparing t\_person2objects.

The following statements create a table named object\_customers2 and add rows to it:

CREATE TABLE object\_customers2 OF t\_person2;

INSERT INTO object\_customers2 VALUES (

t\_person2(1, 'John', 'Brown', '01-FEB-1955', '800-555-1211',

t\_address('2 State Street', 'Beantown', 'MA', '12345')

)

);

INSERT INTO object\_customers2 VALUES (

t\_person2(2, 'Cynthia', 'Green', '05-FEB-1968', '800-555-1212',

t\_address('3 Free Street', 'Middle Town', 'CA', '12345')

)

);

The following query uses > in the WHERE clause:

**SELECT oc2.id, oc2.first\_name, oc2.last\_name, oc2.dob**

**FROM object\_customers2 oc2**

**WHERE VALUE(oc2) >**

**t\_person2(1, 'John', 'Brown', '01-FEB-1955', '800-555-1211'**,

**t\_address('2 State Street', 'Beantown', 'MA', '12345')**

);

ID FIRST\_NAME LAST\_NAME DOB

---------- ---------- ---------- ---------

2 Cynthia Green 05-FEB-68

When the query is executed, the database automatically calls get\_string() to compare the objects in the object\_customers2 table to the object after the > in the WHERE clause. The get\_string() function returns a concatenation of the last\_name and first\_name attributes of the objects, and because Green Cynthia is greater than Brown John, she is returned by the query.

**USING OBJECTS IN PL/SQL**

You can create and manipulate objects in PL/SQL. In this section, you’ll see the use of a package named product\_package, which is created when you run the object\_schema.sql script; product\_package contains the following methods:

Image A function named get\_products() that returns a REF CURSOR that points to the objects in the object\_products table

Image A procedure named display\_product() that displays the attributes of a single object in the object\_products table

Image A procedure named insert\_product() that adds an object to the object\_products table

Image A procedure named update\_product\_price() that updates the price attribute of an object in the object\_products table

Image A function named get\_product() that returns a single object from the object\_products table

Image A procedure named update\_product() that updates all the attributes of an object in the object\_products table

Image A function named get\_product\_ref() that returns a reference to a single object from the object\_products table

Image A procedure named delete\_product() that deletes a single object from the object\_products table

The object\_schema.sql script contains the following package specification:

CREATE PACKAGE product\_package AS

TYPE t\_ref\_cursor IS REF CURSOR;

FUNCTION get\_products RETURN t\_ref\_cursor;

PROCEDURE display\_product(

p\_id IN object\_products.id%TYPE

);

PROCEDURE insert\_product(

p\_id IN object\_products.id%TYPE,

p\_name IN object\_products.name%TYPE,

p\_description IN object\_products.description%TYPE,

p\_price IN object\_products.price%TYPE,

p\_days\_valid IN object\_products.days\_valid%TYPE

);

PROCEDURE update\_product\_price(

p\_id IN object\_products.id%TYPE,

p\_factor IN NUMBER

);

FUNCTION get\_product(

p\_id IN object\_products.id%TYPE

) RETURN t\_product;

PROCEDURE update\_product(

p\_product t\_product

);

FUNCTION get\_product\_ref(

p\_id IN object\_products.id%TYPE

) RETURN REF t\_product;

PROCEDURE delete\_product(

p\_id IN object\_products.id%TYPE

);

END product\_package;

/

You’ll see the methods in the body of product\_package in the following sections.

**The get\_products() Function**

The get\_products() function returns a REF CURSOR that points to the objects in the object\_productstable; get\_products() is defined as follows in the body of product\_package:

FUNCTION get\_products

RETURN t\_ref\_cursor IS

-- declare a t\_ref\_cursor object

v\_products\_ref\_cursor t\_ref\_cursor;

BEGIN

-- get the REF CURSOR

OPEN v\_products\_ref\_cursor FOR

SELECT VALUE(op)

FROM object\_products op

ORDER BY op.id;

-- return the REF CURSOR

RETURN v\_products\_ref\_cursor;

END get\_products;

The following query calls product\_package.get\_products() to retrieve the products from object\_products:

**SELECT product\_package.get\_products**

**FROM dual;**

GET\_PRODUCTS

--------------------

CURSOR STATEMENT : 1

CURSOR STATEMENT : 1

VALUE(OP)(ID, NAME, DESCRIPTION, PRICE, DAYS\_VALID)

----------------------------------------------------------

T\_PRODUCT(1, 'pasta', '20 oz bag of pasta', 3.95, 10)

T\_PRODUCT(2, ’sardines', '12 oz box of sardines', 2.99, 5)

**The display\_product() Procedure**

The display\_product() procedure displays the attributes of a single object in the object\_productstable; display\_product() is defined as follows in the body of product\_package:

PROCEDURE display\_product(

p\_id IN object\_products.id%TYPE

) AS

-- declare a t\_product object named v\_product

v\_product t\_product;

BEGIN

-- attempt to get the product and store it in v\_product

SELECT VALUE(op)

INTO v\_product

FROM object\_products op

WHERE id = p\_id;

-- display the attributes of v\_product

DBMS\_OUTPUT.PUT\_LINE('v\_product.id=' ||

v\_product.id);

DBMS\_OUTPUT.PUT\_LINE('v\_product.name=' ||

v\_product.name);

DBMS\_OUTPUT.PUT\_LINE('v\_product.description=' ||

v\_product.description);

DBMS\_OUTPUT.PUT\_LINE('v\_product.price=' ||

v\_product.price);

DBMS\_OUTPUT.PUT\_LINE('v\_product.days\_valid=' ||

v\_product.days\_valid);

-- call v\_product.get\_sell\_by\_date() and display the date

DBMS\_OUTPUT.PUT\_LINE(’Sell by date=' ||

v\_product.get\_sell\_by\_date());

END display\_product;

The following example calls product\_package.display\_product(1) to retrieve product #1 from the object\_products table:

**SET SERVEROUTPUT ON**

**CALL product\_package.display\_product(1);**

v\_product.id=1

v\_product.name=pasta

v\_product.description=20 oz bag of pasta

v\_product.price=3.95

v\_product.days\_valid=10

Sell by date=25-JUN-07

**The insert\_product() Procedure**

The insert\_product() procedure adds an object to the object\_products table; insert\_product() is defined as follows in the body of product\_package:

PROCEDURE insert\_product(

p\_id IN object\_products.id%TYPE,

p\_name IN object\_products.name%TYPE,

p\_description IN object\_products.description%TYPE,

p\_price IN object\_products.price%TYPE,

p\_days\_valid IN object\_products.days\_valid%TYPE

) AS

-- create a t\_product object named v\_product

v\_product t\_product :=

t\_product(

p\_id, p\_name, p\_description, p\_price, p\_days\_valid

);

BEGIN

-- add v\_product to the object\_products table

INSERT INTO object\_products VALUES (v\_product);

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

END insert\_product;

The following example calls product\_package.insert\_product() to add a new object to the object\_products table:

**CALL product\_package.insert\_product(3, ’salsa'**,

**'15 oz jar of salsa', 1.50, 20);**

**The update\_product\_price() Procedure**

The update\_product\_price() procedure updates the price attribute of an object in the object\_products table; update\_product\_price() is defined as follows in the body of product\_package:

PROCEDURE update\_product\_price(

p\_id IN object\_products.id%TYPE,

p\_factor IN NUMBER

) AS

-- declare a t\_product object named v\_product

v\_product t\_product;

BEGIN

-- attempt to select the product for update and

-- store the product in v\_product

SELECT VALUE(op)

INTO v\_product

FROM object\_products op

WHERE id = p\_id

FOR UPDATE;

-- display the current price of v\_product

DBMS\_OUTPUT.PUT\_LINE('v\_product.price=' ||

v\_product.price);

-- multiply v\_product.price by p\_factor

v\_product.price := v\_product.price \* p\_factor;

DBMS\_OUTPUT.PUT\_LINE('New v\_product.price=' ||

v\_product.price);

-- update the product in the object\_products table

UPDATE object\_products op

SET op = v\_product

WHERE id = p\_id;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

END update\_product\_price;

The following example calls product\_package.update\_product\_price() to update the price of product #3 in the object\_products table:

**CALL product\_package.update\_product\_price(3, 2.4);**

v\_product.price=1.5

New v\_product.price=3.6

**The get\_product() Function**

The get\_product() function returns a single object from the object\_products table; get\_product() is defined as follows in the body of product\_package:

FUNCTION get\_product(

p\_id IN object\_products.id%TYPE

)

RETURN t\_product IS

-- declare a t\_product object named v\_product

v\_product t\_product;

BEGIN

-- get the product and store it in v\_product

SELECT VALUE(op)

INTO v\_product

FROM object\_products op

WHERE op.id = p\_id;

-- return v\_product

RETURN v\_product;

END get\_product;

The following query calls product\_package.get\_product() to get product #3 from the object\_products table:

**SELECT product\_package.get\_product(3)**

**FROM dual;**

PRODUCT\_PACKAGE.GET\_PRODUCT(3)(ID, NAME, DESCRIPTION

----------------------------------------------------

T\_PRODUCT(3, ’salsa', '15 oz jar of salsa', 3.6, 20)

**The update\_product() Procedure**

The update\_product() procedure updates all the attributes of an object in the object\_products table; update\_product() is defined as follows in the body of product\_package:

PROCEDURE update\_product(

p\_product IN t\_product

) AS

BEGIN

-- update the product in the object\_products table

UPDATE object\_products op

SET op = p\_product

WHERE id = p\_product.id;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

END update\_product;

The following example calls product\_package.update\_product() to update product #3 in the object\_products table:

**CALL product\_package.update\_product(t\_product(3, ’salsa'**,

**'25 oz jar of salsa', 2.70, 15));**

**The get\_product\_ref() Function**

The get\_product\_ref() function returns a reference to a single object from the object\_productstable; get\_product\_ref() is defined as follows in the body of product\_package:

FUNCTION get\_product\_ref(

p\_id IN object\_products.id%TYPE

)

RETURN REF t\_product IS

-- declare a reference to a t\_product

v\_product\_ref REF t\_product;

BEGIN

-- get the REF for the product and

-- store it in v\_product\_ref

SELECT REF(op)

INTO v\_product\_ref

FROM object\_products op

WHERE op.id = p\_id;

-- return v\_product\_ref

RETURN v\_product\_ref;

END get\_product\_ref;

The following query calls product\_package.get\_product\_ref() to get the reference to product #3 from the object\_products table:

**SELECT product\_package.get\_product\_ref(3)**

**FROM dual;**

PRODUCT\_PACKAGE.GET\_PRODUCT\_REF(3)

------------------------------------------------------------------------------

000028020956DBE8BEFDEF4D5BA8C806A7B31B49DF916CDB2CAC1B46E9808BA181F9F2760F0100

033D0002

The next example calls product\_package.get\_product\_ref() again, this time using DEREF() to get to the actual product:

**SELECT DEREF(product\_package.get\_product\_ref(3))**

**FROM dual;**

DEREF(PRODUCT\_PACKAGE.GET\_PRODUCT\_REF(3))(ID, NAME,

----------------------------------------------------

T\_PRODUCT(3, ’salsa', '25 oz jar of salsa', 2.7, 15)

**The delete\_product() Procedure**

The delete\_product() procedure deletes a single object from the object\_products table; delete\_product() is defined as follows in the body of product\_package:

PROCEDURE delete\_product(

p\_id IN object\_products.id%TYPE

) AS

BEGIN

-- delete the product

DELETE FROM object\_products op

WHERE op.id = p\_id;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

END delete\_product;

The following example calls product\_package.delete\_product () to delete product #3 from the object\_products table:

**CALL product\_package.delete\_product(3);**

Now that you’ve seen all the methods in product\_package, it’s time for you to see two procedures named product\_lifecycle() and product\_lifecycle2() that call the various methods in the package. Both procedures are created when you run the object\_schema.sql script.

**The product\_lifecycle() Procedure**

The product\_lifecycle() procedure is defined as follows:

CREATE PROCEDURE product\_lifecycle AS

-- declare object

v\_product t\_product;

BEGIN

-- insert a new product

product\_package.insert\_product(4, 'beef',

'25 lb pack of beef', 32, 10);

-- display the product

product\_package.display\_product(4);

-- get the new product and store it in v\_product

SELECT product\_package.get\_product(4)

INTO v\_product

FROM dual;

-- change some attributes of v\_product

v\_product.description := '20 lb pack of beef';

v\_product.price := 36;

v\_product.days\_valid := 8;

-- update the product

product\_package.update\_product(v\_product);

-- display the product

product\_package.display\_product(4);

-- delete the product

product\_package.delete\_product(4);

END product\_lifecycle;

/

The following example calls product\_lifecycle():

**CALL product\_lifecycle();**

v\_product.id=4

v\_product.name=beef

v\_product.description=25 lb pack of beef

v\_product.price=32

v\_product.days\_valid=10

Sell by date=27-JUN-07

v\_product.id=4

v\_product.name=beef

v\_product.description=20 lb pack of beef

v\_product.price=36

v\_product.days\_valid=8

Sell by date=25-JUN-07

**The product\_lifecycle2() Procedure**

The product\_lifecycle2() procedure uses an object reference to access a product; product\_lifecycle2() is defined as follows:

CREATE PROCEDURE product\_lifecycle2 AS

-- declare object

v\_product t\_product;

-- declare object reference

v\_product\_ref REF t\_product;

BEGIN

-- insert a new product

product\_package.insert\_product(4, 'beef',

'25 lb pack of beef', 32, 10);

-- display the product

product\_package.display\_product(4);

-- get the new product reference and store it in v\_product\_ref

SELECT product\_package.get\_product\_ref(4)

INTO v\_product\_ref

FROM dual;

-- dereference v\_product\_ref using the following query

SELECT DEREF(v\_product\_ref)

INTO v\_product

FROM dual;

-- change some attributes of v\_product

v\_product.description := '20 lb pack of beef';

v\_product.price := 36;

v\_product.days\_valid := 8;

-- update the product

product\_package.update\_product(v\_product);

-- display the product

product\_package.display\_product(4);

-- delete the product

product\_package.delete\_product(4);

END product\_lifecycle2;

/

One point to note in this procedure is that, in order to dereference v\_product\_ref, you have to use the following query:

SELECT DEREF(v\_product\_ref)

INTO v\_product

FROM dual;

The reason you have to use this query is that you cannot use DEREF() directly in PL/SQL code. For example, the following statement won’t compile in PL/SQL:

v\_product := DEREF(v\_product\_ref);

The following example calls product\_lifecycle2():

**CALL product\_lifecycle2();**

v\_product.id=4

v\_product.name=beef

v\_product.description=25 lb pack of beef

v\_product.price=32

v\_product.days\_valid=10

Sell by date=27-JUN-07

v\_product.id=4

v\_product.name=beef

v\_product.description=20 lb pack of beef

v\_product.price=36

v\_product.days\_valid=8

Sell by date=25-JUN-07

**TYPE INHERITANCE**

Oracle Database 9*i* introduced object type *inheritance*, which allows you to define hierarchies of object types. For example, you might want to define a business person object type and have that type inherit the existing attributes from t\_person. The business person type could extend t\_person with attributes to store the person’s job title and the name of the company they work for. For t\_person to be inherited from, the t\_person definition must include the NOT FINAL clause:

CREATE TYPE t\_person AS OBJECT (

id INTEGER,

first\_name VARCHAR2(10),

last\_name VARCHAR2(10),

dob DATE,

phone VARCHAR2(12),

address t\_address,

MEMBER FUNCTION display\_details RETURN VARCHAR2

) NOT FINAL;

/

The NOT FINAL clause indicates that t\_person can be inherited from when defining another type. (The default when defining types is FINAL, meaning that the object type cannot be inherited from.)

The following statement creates the body for t\_person; notice that the display\_details() function returns a VARCHAR2 containing the id and name of the person:

CREATE TYPE BODY t\_person AS

MEMBER FUNCTION display\_details RETURN VARCHAR2 IS

BEGIN

RETURN 'id=' || id || ', name=' || first\_name || ' ' || last\_name;

END;

END;

/

Image

**NOTE**  
*I’ve provided an SQL\*Plus script named* object\_schema2.sql, *which creates all the items shown in this and the following sections. You can run the script if you are using Oracle Database 9* i *or above. After the script completes, you will be logged in as* object\_user2.

To have a new type inherit attributes and methods from an existing type, you use the UNDER keyword when defining your new type. Our business person type, which I’ll name t\_business\_person, uses the UNDERkeyword to inherit the attributes from t\_person:

CREATE TYPE t\_business\_person UNDER t\_person (

title VARCHAR2(20),

company VARCHAR2(20)

);

/

In this example, t\_person is known as the *supertype*, and t\_business\_person is known as the *subtype*. You can then use t\_business\_person when defining column objects or object tables. For example, the following statement creates an object table named object\_business\_customers:

CREATE TABLE object\_business\_customers OF t\_business\_person;

The following INSERT statement adds an object to object\_business\_customers; notice that the two additional title and company attributes are supplied at the end of the t\_business\_person constructor:

INSERT INTO object\_business\_customers VALUES (

t\_business\_person(1, 'John', 'Brown', '01-FEB-1955', '800-555-1211',

t\_address('2 State Street', 'Beantown', 'MA', '12345'),

'Manager', 'XYZ Corp'

)

);

The following query retrieves this object:

**SELECT \***

**FROM object\_business\_customers**

**WHERE id = 1;**

ID FIRST\_NAME LAST\_NAME DOB PHONE

---------- ---------- ---------- --------- ------------

ADDRESS(STREET, CITY, STATE, ZIP)

------------------------------------------------------

TITLE COMPANY

-------------------- --------------------

1 John Brown 01-FEB-55 800-555-1211

T\_ADDRESS('2 State Street', 'Beantown', 'MA', '12345')

Manager XYZ Corp

The following query calls the display\_details() function for this object:

**SELECT o.display\_details()**

**FROM object\_business\_customers o**

**WHERE id = 1;**

0.DISPLAY\_DETAILS()

---------------------

id=1, name=John Brown

When you call a method, the database searches for that method in the subtype first; if the method isn’t found, the supertype is searched. If you have a hierarchy of types, the database will search for the method up the hierarchy; if the method cannot be found, the database will report an error.

**USING A SUBTYPE OBJECT IN PLACE OF A SUPERTYPE OBJECT**

In this section you’ll see how you can use a subtype object in place of a supertype object; doing this gives you great flexibility when storing and manipulating related types. In the examples, you’ll see how you use a t\_business\_person object (a subtype object) in place of a t\_person object (a supertype object).

**SQL Examples**

The following statement creates a table named object\_customers of type t\_person:

CREATE TABLE object\_customers OF t\_person;

The following INSERT statement adds a t\_person object to this table (the name is Jason Bond):

INSERT INTO object\_customers VALUES (

t\_person(1, 'Jason', 'Bond', '03-APR-1965', '800-555-1212',

t\_address('21 New Street', 'Anytown', 'CA', '12345')

)

);

There’s nothing unusual about the previous statement: The INSERT simply adds a t\_person object to the object\_customers table. Now, because the object\_customers table stores objects of type t\_person, and t\_person is a supertype of t\_business\_person, you can store a t\_business\_person object in object\_customers; the following INSERT shows this, adding a customer named Steve Edwards:

INSERT INTO object\_customers VALUES (

t\_business\_person(2, ’Steve', 'Edwards', '03-MAR-1955', '800-555-1212',

t\_address('1 Market Street', 'Anytown', 'VA', '12345'),

'Manager', 'XYZ Corp'

)

);

The object\_customers table now contains two objects: the t\_person object added earlier (Jason Bond) and the new t\_business\_person object (Steve Edwards). The following query retrieves these two objects; notice that the title and company attributes for Steve Edwards are missing from the output:

**SELECT \***

**FROM object\_customers o;**

ID FIRST\_NAME LAST\_NAME DOB PHONE

---------- ---------- ---------- --------- ------------

ADDRESS(STREET, CITY, STATE, ZIP)

------------------------------------------------------

1 Jason Bond 03-APR-65 800-555-1212

T\_ADDRESS('21 New Street', 'Anytown', 'CA', '12345')

2 Steve Edwards 03-MAR-55 800-555-1212

T\_ADDRESS('1 Market Street', 'Anytown', 'VA', '12345')

You can get the full set of attributes for Steve Edwards by using VALUE() in the query, as shown in the following example; notice the different types of the objects for Jason Bond (a t\_person object) and Steve Edwards (a t\_business\_person object) and that the title and company attributes for Steve Edwards now appear in the output:

**SELECT VALUE(o)**

**FROM object\_customers o;**

VALUE(O)(ID, FIRST\_NAME, LAST\_NAME, DOB, PHONE,

ADDRESS(STREET, CITY, STATE, ZIP

---------------------------------------------------------------------

T\_PERSON(1, 'Jason', 'Bond', '03-APR-65', '800-555-1212',

T\_ADDRESS('21 New Street', 'Anytown', 'CA', '12345'))

T\_BUSINESS\_PERSON(2, ’Steve', 'Edwards', '03-MAR-55', '800-555-1212',

T\_ADDRESS('1 Market Street', 'Anytown', 'VA', '12345'),

'Manager', 'XYZ Corp')

**PL/SQL Examples**

You can also manipulate subtype and supertype objects in PL/SQL. For example, the following procedure named subtypes\_and\_supertypes() manipulates t\_business\_person and t\_person objects:

CREATE PROCEDURE subtypes\_and\_supertypes AS

-- create objects

v\_business\_person t\_business\_person :=

t\_business\_person(

1, 'John', 'Brown',

'01-FEB-1955', '800-555-1211',

t\_address('2 State Street', 'Beantown', 'MA', '12345'),

'Manager', 'XYZ Corp'

);

v\_person t\_person :=

t\_person(1, 'John', 'Brown', '01-FEB-1955', '800-555-1211',

t\_address('2 State Street', 'Beantown', 'MA', '12345'));

v\_business\_person2 t\_business\_person;

v\_person2 t\_person;

BEGIN

-- assign v\_business\_person to v\_person2

v\_person2 := v\_business\_person;

DBMS\_OUTPUT.PUT\_LINE('v\_person2.id = ' || v\_person2.id);

DBMS\_OUTPUT.PUT\_LINE('v\_person2.first\_name = ' ||

v\_person2.first\_name);

DBMS\_OUTPUT.PUT\_LINE('v\_person2.last\_name = ' ||

v\_person2.last\_name);

-- the following lines will not compile because v\_person2

-- is of type t\_person, and t\_person does not know about the

-- additional title and company attributes

-- DBMS\_OUTPUT.PUT\_LINE('v\_person2.title = ' ||

-- v\_person2.title);

-- DBMS\_OUTPUT.PUT\_LINE('v\_person2.company = ' ||

-- v\_person2.company);

-- the following line will not compile because you cannot

-- directly assign a t\_person object to a t\_business\_person

-- object

-- v\_business\_person2 := v\_person;

END subtypes\_and\_supertypes;

/

The following example shows the result of calling subtypes\_and\_supertypes():

**SET SERVEROUTPUT ON**

**CALL subtypes\_and\_supertypes();**

v\_person2.id = 1

v\_person2.first\_name = John

v\_person2.last\_name = Brown

**NOT SUBSTITUTABLE Objects**

If you want to prevent the use of a subtype object in place of a supertype object, you can mark an object table or object column as "not substitutable"; for example, the following statement creates a table named object\_customers2:

CREATE TABLE object\_customers\_not\_subs OF t\_person

NOT SUBSTITUTABLE AT ALL LEVELS;

The NOT SUBSTITUTABLE AT ALL LEVELS clause indicates that no objects of a type other than t\_person can be inserted into the table. If an attempt is made to add an object of type t\_business\_person to this table, an error is returned:

SQL> **INSERT INTO object\_customers\_not\_subs VALUES (**

2 **t\_business\_person(1, ’Steve', 'Edwards', '03-MAR-1955', '800-555-1212'**,

3 **t\_address('1 Market Street', 'Anytown', 'VA', '12345')**,

4 **'Manager', 'XYZ Corp'**

5 **)**

6 **);**

t\_business\_person(1, ’Steve', 'Edwards', '03-MAR-1955', '800-555-1212',

\*

ERROR at line 2:

ORA-00932: inconsistent datatypes: expected OBJECT\_USER2.T\_PERSON got

OBJECT\_USER2.T\_BUSINESS\_PERSON

You can also mark an object column as not substitutable; for example, the following statement creates a table with an object column named product that can store only objects of type t\_product:

CREATE TABLE products (

product t\_product,

quantity\_in\_stock INTEGER

)

COLUMN product NOT SUBSTITUTABLE AT ALL LEVELS;

Any attempts to add an object not of type t\_product to the product column will result in an error.

**OTHER USEFUL OBJECT FUNCTIONS**

In the earlier sections of this chapter you saw the use of the REF(), DEREF(), and VALUE() functions. In this section, you’ll see the following additional functions that may be used with objects:

Image IS OF() checks if an object is of a particular type or subtype.

Image TREAT() does a run-time check to see if an object’s type may be treated as a supertype.

Image SYS\_TYPEID() returns the ID of an object’s type.

**IS OF()**

You use IS OF() to check whether an object is of a particular type or subtype. For example, the following query uses IS OF() to check whether the objects in the object\_business\_customers table are of type t\_business\_person—because they are, a row is returned by the query:

**SELECT VALUE(o)**

**FROM object\_business\_customers o**

**WHERE VALUE(o) IS OF (t\_business\_person);**

VALUE(O)(ID, FIRST\_NAME, LAST\_NAME, DOB, PHONE,

ADDRESS(STREET, CITY, STATE, ZIP

------------------------------------------------------------------

T\_BUSINESS\_PERSON(1, 'John', 'Brown', '01-FEB-55', '800-555-1211',

T\_ADDRESS('2 State Street', 'Beantown', 'MA', '12345'),

'Manager', 'XYZ Corp')

You can also use IS OF() to check whether an object is of a subtype of the specified type. For example, the objects in the object\_business\_customers table are of type t\_business\_person, which is a subtype of t\_person; therefore, the following query returns the same result as that shown in the previous example:

**SELECT VALUE(o)**

**FROM object\_business\_customers o**

**WHERE VALUE(o) IS OF (t\_person);**

VALUE(O)(ID, FIRST\_NAME, LAST\_NAME, DOB, PHONE,

ADDRESS(STREET, CITY, STATE, ZIP

------------------------------------------------------------------

T\_BUSINESS\_PERSON(1, 'John', 'Brown', '01-FEB-55', '800-555-1211',

T\_ADDRESS('2 State Street', 'Beantown', 'MA', '12345'),

'Manager', 'XYZ Corp')

You can include more than one type in IS OF(); for example:

**SELECT VALUE(o)**

**FROM object\_business\_customers o**

**WHERE VALUE(o) IS OF (t\_business\_person, t\_person);**

VALUE(O)(ID, FIRST\_NAME, LAST\_NAME, DOB, PHONE,

ADDRESS(STREET, CITY, STATE, ZIP

------------------------------------------------------------------

T\_BUSINESS\_PERSON(1, 'John', 'Brown', '01-FEB-55', '800-555-1211',

T\_ADDRESS('2 State Street', 'Beantown', 'MA', '12345'),

'Manager', 'XYZ Corp')

In the earlier section entitled "Using a Subtype Object in Place of a Supertype Object," you saw the addition of a t\_person object (Jason Bond) and t\_business\_person object (Steve Edwards) to the object\_customers table. As a reminder, the following query shows these objects:

**SELECT VALUE(o)**

**FROM object\_customers o;**

VALUE(O)(ID, FIRST\_NAME, LAST\_NAME, DOB, PHONE,

ADDRESS(STREET, CITY, STATE, ZIP

---------------------------------------------------------------------

T\_PERSON(1, 'Jason', 'Bond', '03-APR-65', '800-555-1212',

T\_ADDRESS('21 New Street', 'Anytown', 'CA', '12345'))

T\_BUSINESS\_PERSON(2, ’Steve', 'Edwards', '03-MAR-55', '800-555-1212',

T\_ADDRESS('1 Market Street', 'Anytown', 'VA', '12345'),

'Manager', 'XYZ Corp')

Because t\_business\_person type is a subtype of t\_person, IS OF (t\_person) returns true when a t\_business\_person object or a t\_person object is checked; this is illustrated in the following query that retrieves both Jason Bond and Steve Edwards using IS OF (t\_person):

**SELECT VALUE(o)**

**FROM object\_customers o**

**WHERE VALUE(o) IS OF (t\_person);**

VALUE(O)(ID, FIRST\_NAME, LAST\_NAME, DOB, PHONE,

ADDRESS(STREET, CITY, STATE, ZIP

---------------------------------------------------------------------

T\_PERSON(1, 'Jason', 'Bond', '03-APR-65', '800-555-1212',

T\_ADDRESS('21 New Street', 'Anytown', 'CA', '12345'))

T\_BUSINESS\_PERSON(2, ’Steve', 'Edwards', '03-MAR-55', '800-555-1212',

T\_ADDRESS('1 Market Street', 'Anytown', 'VA', '12345'),

'Manager', 'XYZ Corp')

You can also use the ONLY keyword in conjunction with IS OF () to check for objects of a specific type only: IS OF () returns false for objects of another type in the hierarchy. For example, IS OF (ONLY t\_person) returns true for objects of type t\_person only and returns false for objects of type t\_business\_person. In this way, you can use IS OF (ONLY t\_person) to restrict the object returned by a query against the object\_customers table to Jason Bond, as shown in the following example:

**SELECT VALUE(o)**

**FROM object\_customers o**

**WHERE VALUE(o) IS OF (ONLY t\_person);**

VALUE(O)(ID, FIRST\_NAME, LAST\_NAME, DOB, PHONE,

ADDRESS(STREET, CITY, STATE, ZIP

---------------------------------------------------------

T\_PERSON(1, 'Jason', 'Bond', '03-APR-65', '800-555-1212',

T\_ADDRESS('21 New Street', 'Anytown', 'CA', '12345'))

Similarly, IS OF (ONLY t\_business\_person) returns true for objects of type t\_business\_person only, and returns false for objects of type t\_person. For example, the following query retrieves the t\_business\_person object only and therefore Steve Edwards is returned:

**SELECT VALUE(o)**

**FROM object\_customers o**

**WHERE VALUE(o) IS OF (ONLY t\_business\_person);**

VALUE(O)(ID, FIRST\_NAME, LAST\_NAME, DOB, PHONE,

ADDRESS(STREET, CITY, STATE, ZIP

---------------------------------------------------------------------

T\_BUSINESS\_PERSON(2, ’Steve', 'Edwards', '03-MAR-55', '800-555-1212',

T\_ADDRESS('1 Market Street', 'Anytown', 'VA', '12345'),

'Manager', 'XYZ Corp')

You can include multiple types after ONLY. For example, IS OF (ONLY t\_person, t\_business\_person) returns true for t\_person and t\_business\_person objects only; the following query shows this by returning, as expected, both Jason Bond and Steve Edwards:

**SELECT VALUE(o)**

**FROM object\_customers o**

**WHERE VALUE(o) IS OF (ONLY t\_person, t\_business\_person);**

VALUE(O)(ID, FIRST\_NAME, LAST\_NAME, DOB, PHONE,

ADDRESS(STREET, CITY, STATE, ZIP

---------------------------------------------------------------------

T\_PERSON(1, 'Jason', 'Bond', '03-APR-65', '800-555-1212',

T\_ADDRESS('21 New Street', 'Anytown', 'CA', '12345'))

T\_BUSINESS\_PERSON(2, ’Steve', 'Edwards', '03-MAR-55', '800-555-1212',

T\_ADDRESS('1 Market Street', 'Anytown', 'VA', '12345'),

'Manager', 'XYZ Corp')

You can also use IS OF() in PL/SQL. For example, the following procedure named check\_types() creates t\_business\_person and t\_person objects, and it uses IS OF() to check their types:

CREATE PROCEDURE check\_types AS

-- create objects

v\_business\_person t\_business\_person :=

t\_business\_person(

1, 'John', 'Brown',

'01-FEB-1955', '800-555-1211',

t\_address('2 State Street', 'Beantown', 'MA', '12345'),

'Manager', 'XYZ Corp'

);

v\_person t\_person :=

t\_person(1, 'John', 'Brown', '01-FEB-1955', '800-555-1211',

t\_address('2 State Street', 'Beantown', 'MA', '12345'));

BEGIN

-- check the types of the objects

IF v\_business\_person IS OF (t\_business\_person) THEN

DBMS\_OUTPUT.PUT\_LINE('v\_business\_person is of type ' ||

't\_business\_person');

END IF;

IF v\_person IS OF (t\_person) THEN

DBMS\_OUTPUT.PUT\_LINE('v\_person is of type t\_person');

END IF;

IF v\_business\_person IS OF (t\_person) THEN

DBMS\_OUTPUT.PUT\_LINE('v\_business\_person is of type t\_person');

END IF;

IF v\_business\_person IS OF (t\_business\_person, t\_person) THEN

DBMS\_OUTPUT.PUT\_LINE('v\_business\_person is of ' ||

'type t\_business\_person or t\_person');

END IF;

IF v\_business\_person IS OF (ONLY t\_business\_person) THEN

DBMS\_OUTPUT.PUT\_LINE('v\_business\_person is of only ' ||

'type t\_business\_person');

END IF;

IF v\_business\_person IS OF (ONLY t\_person) THEN

DBMS\_OUTPUT.PUT\_LINE('v\_business\_person is of only ' ||

'type t\_person');

ELSE

DBMS\_OUTPUT.PUT\_LINE('v\_business\_person is not of only ' ||

'type t\_person');

END IF;

END check\_types;

/

The following example shows the result of calling check\_types():

**SET SERVEROUTPUT ON**

**CALL check\_types();**

v\_business\_person is of type t\_business\_person

v\_person is of type t\_person

v\_business\_person is of type t\_person

v\_business\_person is of type t\_business\_person or t\_person

v\_business\_person is of only type t\_business\_person

v\_business\_person is not of only type t\_person

**TREAT()**

You use TREAT() to do a run-time check to see whether an object of a subtype may be treated as an object of a supertype; if this is so, TREAT() returns an object, and if not so, TREAT() returns null. For example, because t\_business\_person is a subtype of t\_person, a t\_business\_person object can be treated as a t\_person object; you saw this earlier in the section entitled "Using a Subtype Object in Place of a Supertype Object," where a t\_business\_person object (Steve Edwards) was inserted into the object\_customers table, which normally holds t\_person objects. The following query uses TREAT() to check that Steve Edwards can be treated as a t\_person object:

**SELECT NVL2(TREAT(VALUE(o) AS t\_person), 'yes', 'no')**

**FROM object\_customers o**

**WHERE first\_name = ’Steve' AND last\_name = 'Edwards';**

NVL

---

yes

NVL2() returns yes because TREAT(VALUE(o) AS t\_person) returns an object (that is, not a null value). This means that Steve Edwards can be treated as a t\_person object.

The next query checks whether Jason Bond (a t\_person object) can be treated as a t\_business\_person object—he cannot, and, therefore, TREAT() returns null, and NVL2() returns no:

**SELECT NVL2(TREAT(VALUE(o) AS t\_business\_person), 'yes', 'no')**

**FROM object\_customers o**

**WHERE first\_name = 'Jason' AND last\_name = 'Bond';**

NVL

---

no

Because TREAT() returns null for the whole object, all the individual attributes for the object are also null. For example, the following query attempts to access the first\_name attribute through Jason Bond—null is returned (as expected):

**SELECT**

**NVL2(TREAT(VALUE(o) AS t\_business\_person).first\_name, 'not null', 'null')**

**FROM object\_customers o**

**WHERE first\_name = 'Jason' AND last\_name = 'Bond';**

NVL2

----

null

The next query uses TREAT() to check whether Jason Bond can be treated as a t\_person object—he *is* a t\_person object and therefore yes is returned:

**SELECT NVL2(TREAT(VALUE(o) AS t\_person).first\_name, 'yes', 'no')**

**FROM object\_customers o**

**WHERE first\_name = 'Jason' AND last\_name = 'Bond';**

NVL

---

yes

You can also retrieve an object through the use of TREAT(); for example, the following query retrieves Steve Edwards:

**SELECT TREAT(VALUE(o) AS t\_business\_person)**

**FROM object\_customers o**

**WHERE first\_name = ’Steve' AND last\_name = 'Edwards';**

TREAT(VALUE(O)AST\_BUSINESS\_PERSON)(ID, FIRST\_NAME, LAST\_NAME, DOB, PHONE,

ADDRESS

-------------------------------------------------------------------------

T\_BUSINESS\_PERSON(2, ’Steve', 'Edwards', '03-MAR-55', '800-555-1212',

T\_ADDRESS('1 Market Street', 'Anytown', 'VA', '12345'),

'Manager', 'XYZ Corp')

If you try this query with Jason Bond, null is returned, as expected; therefore, nothing appears in the output of the following query:

**SELECT TREAT(VALUE(o) AS t\_business\_person)**

**FROM object\_customers o**

**WHERE first\_name = 'Jason' AND last\_name = 'Bond';**

TREAT(VALUE(O)AST\_BUSINESS\_PERSON)(ID, FIRST\_NAME, LAST\_NAME, DOB, PHONE,

ADDRESS

-------------------------------------------------------------------------

Let’s take look at using TREAT() with the object\_business\_customers table, which contains the t\_business\_person object John Brown:

**SELECT VALUE(o)**

**FROM object\_business\_customers o;**

VALUE(O)(ID, FIRST\_NAME, LAST\_NAME, DOB, PHONE,

ADDRESS(STREET, CITY, STATE, ZIP

------------------------------------------------------------------

T\_BUSINESS\_PERSON(1, 'John', 'Brown', '01-FEB-55', '800-555-1211',

T\_ADDRESS('2 State Street', 'Beantown', 'MA', '12345'),

'Manager', 'XYZ Corp')

The following query uses TREAT() to check whether John Brown can be treated as a t\_person object—he can, because t\_business\_person is a subtype of t\_person; therefore, yes is returned by the query:

**SELECT NVL2(TREAT(VALUE(o) AS t\_person), 'yes', 'no')**

**FROM object\_business\_customers o**

**WHERE first\_name = 'John' AND last\_name = 'Brown';**

NVL

---

yes

The following example shows the object returned by TREAT() when querying the object\_business\_customers table; notice that you still get the title and company attributes for John Brown:

**SELECT TREAT(VALUE(o) AS t\_person)**

**FROM object\_business\_customers o;**

TREAT(VALUE(O)AST\_PERSON)(ID, FIRST\_NAME, LAST\_NAME, DOB, PHONE,

ADDRESS(STREET,

------------------------------------------------------------------

T\_BUSINESS\_PERSON(1, 'John', 'Brown', '01-FEB-55', '800-555-1211',

T\_ADDRESS('2 State Street', 'Beantown', 'MA', '12345'),

'Manager', 'XYZ Corp')

You can also use TREAT() in PL/SQL. For example, the following procedure named treat\_example()illustrates the use of TREAT() (you should study the comments in the code to understand how TREAT() works in PL/SQL):

CREATE PROCEDURE treat\_example AS

-- create objects

v\_business\_person t\_business\_person :=

t\_business\_person(

1, 'John', 'Brown',

'01-FEB-1955', '800-555-1211',

t\_address('2 State Street', 'Beantown', 'MA', '12345'),

'Manager', 'XYZ Corp'

);

v\_person t\_person :=

t\_person(1, 'John', 'Brown', '01-FEB-1955', '800-555-1211',

t\_address('2 State Street', 'Beantown', 'MA', '12345'));

v\_business\_person2 t\_business\_person;

v\_person2 t\_person;

BEGIN

-- assign v\_business\_person to v\_person2

v\_person2 := v\_business\_person;

DBMS\_OUTPUT.PUT\_LINE('v\_person2.id = ' || v\_person2.id);

DBMS\_OUTPUT.PUT\_LINE('v\_person2.first\_name = ' ||

v\_person2.first\_name);

DBMS\_OUTPUT.PUT\_LINE('v\_person2.last\_name = ' ||

v\_person2.last\_name);

-- the following lines will not compile because v\_person2

-- is of type t\_person, and t\_person does not know about the

-- additional title and company attributes

-- DBMS\_OUTPUT.PUT\_LINE('v\_person2.title = ' ||

-- v\_person2.title);

-- DBMS\_OUTPUT.PUT\_LINE('v\_person2.company = ' ||

-- v\_person2.company);

-- use TREAT when assigning v\_business\_person to v\_person2

DBMS\_OUTPUT.PUT\_LINE('Using TREAT');

v\_person2 := TREAT(v\_business\_person AS t\_person);

DBMS\_OUTPUT.PUT\_LINE('v\_person2.id = ' || v\_person2.id);

DBMS\_OUTPUT.PUT\_LINE('v\_person2.first\_name = ' ||

v\_person2.first\_name);

DBMS\_OUTPUT.PUT\_LINE('v\_person2.last\_name = ' ||

v\_person2.last\_name);

-- the following lines will still not compile because v\_person2

-- is of type t\_person, and t\_person does not know about the

-- additional title and company attributes

-- DBMS\_OUTPUT.PUT\_LINE('v\_person2.title = ' ||

-- v\_person2.title);

-- DBMS\_OUTPUT.PUT\_LINE('v\_person2.company = ' ||

-- v\_person2.company);

-- the following lines do compile because TREAT is used

DBMS\_OUTPUT.PUT\_LINE('v\_person2.title = ' ||

TREAT(v\_person2 AS t\_business\_person).title);

DBMS\_OUTPUT.PUT\_LINE('v\_person2.company = ' ||

TREAT(v\_person2 AS t\_business\_person).company);

-- the following line will not compile because you cannot

-- directly assign a t\_person object to a t\_business\_person

-- object

-- v\_business\_person2 := v\_person;

-- the following line throws a runtime error because you cannot

-- assign a supertype object (v\_person) to a subtype object

-- (v\_business\_person2)

-- v\_business\_person2 := TREAT(v\_person AS t\_business\_person);

END treat\_example;

/

The following example shows the result of calling treat\_example():

**SET SERVEROUTPUT ON**

**CALL treat\_example();**

v\_person2.id = 1

v\_person2.first\_name = John

v\_person2.last\_name = Brown

Using TREAT

v\_person2.id = 1

v\_person2.first\_name = John

v\_person2.last\_name = Brown

v\_person2.title = Manager

v\_person2.company = XYZ Corp

**SYS\_TYPEID()**

You use SYS\_TYPEID() to get the ID of an object’s type. For example, the following query uses SYS\_TYPEID() to get the ID of the object type in the object\_business\_customers table:

**SELECT first\_name, last\_name, SYS\_TYPEID(VALUE(o))**

**FROM object\_business\_customers o;**

FIRST\_NAME LAST\_NAME SY

---------- ---------- --

John Brown 02

You can get details on the types defined by the user through the user\_types view. The following query retrieves the details of the type with a typeid of '02' (the ID returned by SYS\_TYPEID() earlier) and the type\_name of T\_BUSINESS\_PERSON:

**SELECT typecode, attributes, methods, supertype\_name**

**FROM user\_types**

**WHERE typeid = '02'**

**AND type\_name = 'T\_BUSINESS\_PERSON';**

TYPECODE ATTRIBUTES METHODS SUPERTYPE\_NAME

-------- ---------- ---------- --------------

OBJECT 8 1 T\_PERSON

From the output of this query you can see that the supertype of t\_business\_person is t\_person. Also, t\_business\_person has eight attributes and one method.

**NOT INSTANTIABLE OBJECT TYPES**

You can mark an object type as NOT INSTANTIABLE, which prevents objects of that type from being created. You might want to mark an object type as NOT INSTANTIABLE when you use the type as an abstract supertype only and never create any objects of that type. For example, you could create a t\_vehicle abstract type and use it as a supertype for a t\_car subtype and a t\_motorcycle subtype; you would then create actual t\_carand t\_motorcycle objects, but never t\_vehicle objects.

The following statement creates a type named t\_vehicle, which is marked as NOT INSTANTIABLE:

CREATE TYPE t\_vehicle AS OBJECT (

id INTEGER,

make VARCHAR2(15),

model VARCHAR2(15)

) NOT FINAL NOT INSTANTIABLE;

/

Image

**NOTE**  
*The* t\_vehicle *type is also marked as* NOT FINAL, *because a* NOT INSTANTIABLE *type cannot be* FINAL. *If it* were FINAL, *you wouldn’t be able to use it as a supertype, which is the whole point of creating it in the first place*.

The next example creates a subtype named t\_car under the t\_vehicle supertype; notice that t\_car has an additional attribute named convertible, which will be used to record whether the car has a convertible roof (Y for yes, N for no):

CREATE TYPE t\_car UNDER t\_vehicle (

convertible CHAR(1)

);

/

The following example creates a subtype named t\_motorcycle under the t\_vehicle supertype; notice that t\_motorcycle has an additional attribute named sidecar, which will be used to record whether the motorcycle has a sidecar (Y for yes, N for no):

CREATE TYPE t\_motorcycle UNDER t\_vehicle (

sidecar CHAR(1)

);

/

The next example creates tables named vehicles, cars, and motorcycles, which are object tables of the types t\_vehicle, t\_car, and t\_motorcycle, respectively:

CREATE TABLE vehicles OF t\_vehicle;

CREATE TABLE cars OF t\_car;

CREATE TABLE motorcycles OF t\_motorcycle;

Because t\_vehicle is NOT INSTANTIABLE, you cannot add an object to the vehicles table. If you attempt to do so, the database returns an error:

SQL> **INSERT INTO vehicles VALUES (**

2 **t\_vehicle(1, 'Toyota', 'MR2', '01-FEB-1955')**

3 **);**

t\_vehicle(1, 'Toyota', 'MR2', '01-FEB-1955')

\*

ERROR at line 2:

ORA-22826: cannot construct an instance of a non instantiable type

The following examples add objects to the cars and motorcycles tables:

INSERT INTO cars VALUES (

t\_car(1, 'Toyota', 'MR2', 'Y')

);

INSERT INTO motorcycles VALUES (

t\_motorcycle(1, 'Harley-Davidson', 'V-Rod', 'N')

);

The following queries retrieve the objects from the cars and motorcycles tables:

**SELECT \***

**FROM cars;**

ID MAKE MODEL C

---------- --------------- --------------- -

1 Toyota MR2 Y

**SELECT \***

**FROM motorcycles;**

ID MAKE MODEL S

---------- --------------- --------------- -

1 Harley-Davidson V-Rod N

**USER-DEFINED CONSTRUCTORS**

As in other object-oriented languages like Java and C#, you can define your own constructors in PL/SQL to initialize a new object. You can define your own constructor to do such things as programmatically setting the attributes of a new object to default values.

The following example creates a type named t\_person2 that declares two constructor methods with differing numbers of parameters:

CREATE TYPE t\_person2 AS OBJECT (

id INTEGER,

first\_name VARCHAR2(10),

last\_name VARCHAR2(10),

dob DATE,

phone VARCHAR2(12),

CONSTRUCTOR FUNCTION t\_person2(

p\_id INTEGER,

p\_first\_name VARCHAR2,

p\_last\_name VARCHAR2

) RETURN SELF AS RESULT,

CONSTRUCTOR FUNCTION t\_person2(

p\_id INTEGER,

p\_first\_name VARCHAR2,

p\_last\_name VARCHAR2,

p\_dob DATE

) RETURN SELF AS RESULT

);

/

Notice the following about the constructor declarations:

Image The CONSTRUCTOR FUNCTION keywords are used to identify the constructors.

Image The RETURN SELF AS RESULT keywords indicate the current object being processed is returned by each constructor; SELF represents the current object being processed. What this means is that the constructor returns the new object it creates.

Image The first constructor accepts three parameters (p\_id, p\_first\_name, and p\_last\_name), and the second constructor accepts four parameters (p\_id, p\_first\_name, p\_last\_name, and p\_dob).

The constructor declarations don’t contain the actual code definitions for the constructors; the definitions are contained in the type body, which is created by the following statement:

CREATE TYPE BODY t\_person2 AS

CONSTRUCTOR FUNCTION t\_person2(

p\_id INTEGER,

p\_first\_name VARCHAR2,

p\_last\_name VARCHAR2

) RETURN SELF AS RESULT IS

BEGIN

SELF.id := p\_id;

SELF.first\_name := p\_first\_name;

SELF.last\_name := p\_last\_name;

SELF.dob := SYSDATE;

SELF.phone := '555-1212';

RETURN;

END;

CONSTRUCTOR FUNCTION t\_person2(

p\_id INTEGER,

p\_first\_name VARCHAR2,

p\_last\_name VARCHAR2,

p\_dob DATE

) RETURN SELF AS RESULT IS

BEGIN

SELF.id := p\_id;

SELF.first\_name := p\_first\_name;

SELF.last\_name := p\_last\_name;

SELF.dob := p\_dob;

SELF.phone := '555-1213';

RETURN;

END;

END;

/

Notice the following:

Image The constructors use SELF to reference the new object being created. For example, SELF.id := p\_id sets the id attribute of the new object to the value of the p\_id parameter passed into the constructor.

Image The first constructor sets the id, first\_name, and last\_name attributes to the p\_id, p\_first\_name, and p\_last\_name parameter values passed into the constructor; the dob attribute is set to the current datetime returned by SYSDATE(), and the phone attribute is set to 555-1212.

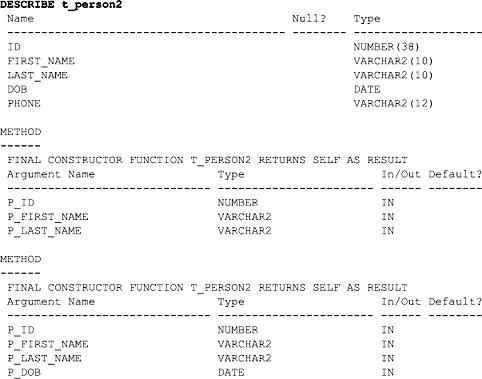
Image The second constructor sets the id, first\_name, last\_name, and dob attributes to the p\_id, p\_first\_name, p\_last\_name, and p\_dob parameter values passed into the constructor; the remaining phone attribute is set to 555-1213.

Although not shown, the database automatically provides a default constructor that accepts five parameters and sets each attribute to the appropriate parameter value passed into the constructor. You’ll see an example of this shortly.

Image

**NOTE**  
*The constructors show an example of* method overloading, *whereby methods of the same name but different parameters are defined in the same type. A method may be overloaded by providing different*numbers *of parameters*, types *of parameters, or* ordering *of parameters*.

The following example describes t\_person2; notice the constructor definitions in the output:



The following statement creates a table of type t\_person2:

CREATE TABLE object\_customers2 OF t\_person2;

The following INSERT statement adds an object to the table; notice that three parameters are passed to the t\_person2 constructor:

INSERT INTO object\_customers2 VALUES (

t\_person2(1, 'Jeff', 'Jones')

);

Because three parameters are passed to t\_person2, this INSERT statement exercises the first constructor. This constructor sets the id, first\_name, and last\_name attributes of the new object to 1, Jeff, and Jones; the remaining dob and phone attributes are set to the result returned by SYSDATE() and the literal 555-1212. The following query retrieves the new object:

**SELECT \***

**FROM object\_customers2**

**WHERE id = 1;**

ID FIRST\_NAME LAST\_NAME DOB PHONE

---------- ---------- ---------- --------- --------

1 Jeff Jones 17-JUN-07 555-1212

The next INSERT statement adds another object to the table; notice that four parameters are passed to the t\_person2 constructor:

INSERT INTO object\_customers2 VALUES (

t\_person2(2, 'Gregory', ’Smith', '03-APR-1965')

);

Because four parameters are passed to t\_person2, this INSERT statement exercises the second constructor. This constructor sets the id, first\_name, last\_name, and dob attributes of the object to 2, Gregory, Smith, and 03-APR-1965, respectively; the remaining phone attribute is set to 555-1213. The following query retrieves the new object:

**SELECT \***

**FROM object\_customers2**

**WHERE id = 2;**

ID FIRST\_NAME LAST\_NAME DOB PHONE

---------- ---------- ---------- --------- --------

2 Gregory Smith 03-APR-65 555-1213

The next INSERT statement adds another object to the table; notice that five parameters are passed to the t\_person2 constructor:

INSERT INTO object\_customers2 VALUES (

t\_person2(3, 'Jeremy', 'Hill', '05-JUN-1975', '555-1214')

);

Because five parameters are passed to t\_person2, this INSERT statement exercises the default constructor. This constructor sets the id, first\_name, last\_name, dob, and phone attributes to 3, Jeremy, Hill, 05-JUN-1975, and 555-1214, respectively. The following query retrieves the new object:

**SELECT \***

**FROM object\_customers2**

**WHERE id = 3;**

ID FIRST\_NAME LAST\_NAME DOB PHONE

---------- ---------- ---------- --------- --------

3 Jeremy Hill 05-JUN-75 555-1214

**OVERRIDING METHODS**

When you create a subtype under a supertype, you can override a method in the supertype with a method in the subtype. This gives you a very flexible way of defining methods in a hierarchy of types.

The following statements create a supertype named t\_person3; notice that the display\_details()function returns a VARCHAR2 containing the attribute values of the object:

CREATE TYPE t\_person3 AS OBJECT (

id INTEGER,

first\_name VARCHAR2(10),

last\_name VARCHAR2(10),

MEMBER FUNCTION display\_details RETURN VARCHAR2

) NOT FINAL;

/

CREATE TYPE BODY t\_person3 AS

MEMBER FUNCTION display\_details RETURN VARCHAR2 IS

BEGIN

RETURN 'id=' || id ||

', name=' || first\_name || ' ' || last\_name;

END;

END;

/

The next set of statements creates a subtype named t\_business\_person3 under t\_person3; notice that the display\_details() function is overridden using the OVERRIDING keyword and that the function returns a VARCHAR2 containing the original and extended attribute values of the object:

CREATE TYPE t\_business\_person3 UNDER t\_person3 (

title VARCHAR2(20),

company VARCHAR2(20),

OVERRIDING MEMBER FUNCTION display\_details RETURN VARCHAR2

);

/

CREATE TYPE BODY t\_business\_person3 AS

OVERRIDING MEMBER FUNCTION display\_details RETURN VARCHAR2 IS

BEGIN

RETURN 'id=' || id ||

', name=' || first\_name || ' ' || last\_name ||

', title=' || title || ', company=' || company;

END;

END;

/

The use of the OVERRIDING keyword indicates that display\_details() in t\_business\_person3overrides display\_details() in t\_person3; therefore, when display\_details() in t\_business\_person3 is called, it calls display\_details() in t\_business\_person3, not display\_details() in t\_person3.

Image

**NOTE**  
*In the next section of this chapter, you’ll see how you can directly call a method in a supertype from a subtype. This saves you from having to recreate code in the subtype that is already in the supertype. You do this direct calling by using a new feature called* generalized invocation *in Oracle Database 11*g.

The following statements create a table named object\_business\_customers3 and add an object to this table:

CREATE TABLE object\_business\_customers3 OF t\_business\_person3;

INSERT INTO object\_business\_customers3 VALUES (

t\_business\_person3(1, 'John', 'Brown', 'Manager', 'XYZ Corp')

);

The following example calls display\_details() using object\_business\_customers3:

**SELECT o.display\_details()**

**FROM object\_business\_customers3 o**

**WHERE id = 1;**

O.DISPLAY\_DETAILS()

------------------------------------------------------

id=1, name=John Brown, title=Manager, company=XYZ Corp

Because the display\_details() function as defined in t\_business\_person3 is called, the VARCHAR2 returned by the function contains the id, first\_name, and last\_name attributes, along with the title and company attributes.

**GENERALIZED INVOCATION**

As you saw in the previous section, you can override a method in the supertype with a method in the subtype. *Generalized invocation* is a new feature in Oracle Database 11*g* and allows you to call a method in a supertype from a subtype. As you’ll see, generalized invocation saves you from having to recreate code in the subtype that is already in the supertype.

Image

**NOTE**  
*I’ve provided an SQL\*Plus script named* object\_schema3.sql, *which creates all the items shown in the rest of this chapter. You can run the* object\_schema3.sql *script only if you are using Oracle Database 11*g. *After the script completes, you will be logged in as* object\_user3.

The following statements create a supertype named t\_person; notice that the display\_details()function returns a VARCHAR2 containing the attribute values:

CREATE TYPE t\_person AS OBJECT (

id INTEGER,

first\_name VARCHAR2(10),

last\_name VARCHAR2(10),

MEMBER FUNCTION display\_details RETURN VARCHAR2

) NOT FINAL;

/

CREATE TYPE BODY t\_person AS

MEMBER FUNCTION display\_details RETURN VARCHAR2 IS

BEGIN

RETURN 'id=' || id ||

', name=' || first\_name || ' ' || last\_name;

END;

END;

/

The next set of statements creates a subtype named t\_business\_person under t\_person; notice that the display\_details() function is overridden using the OVERRIDING keyword:

CREATE TYPE t\_business\_person UNDER t\_person (

title VARCHAR2(20),

company VARCHAR2(20),

OVERRIDING MEMBER FUNCTION display\_details RETURN VARCHAR2

);

/

CREATE TYPE BODY t\_business\_person AS

OVERRIDING MEMBER FUNCTION display\_details RETURN VARCHAR2 IS

BEGIN

-- use generalized invocation to call display\_details() in t\_person

RETURN (SELF AS t\_person).display\_details ||

', title=' || title || ', company=' || company;

END;

END;

/

As you can see, display\_details() in t\_business\_person overrides display\_details() in t\_person. The following line in display\_details() uses generalized invocation to call a method in a supertype from a subtype:

RETURN (SELF AS t\_person).display\_details ||

', title=' || title || ', company=' || company;

What (SELF AS t\_person).display\_details does is to treat an object of the current type (which is t\_business\_person) as an object of type t\_person and then to call display\_details() in t\_person. So, when display\_details() in t\_business\_person is called, it first calls display\_details() in t\_person (which displays the id, first\_name, and last\_name attribute values), then displays the title and company attribute values. This meant I didn’t have to re-create the code already in t\_person.display\_details() in t\_business\_person.display\_details(), thereby saving some work. If you have more complex methods in your types, this feature can save a lot of work and make your code easier to maintain.

The following statements create a table named object\_business\_customers and add an object to this table:

CREATE TABLE object\_business\_customers OF t\_business\_person;

INSERT INTO object\_business\_customers VALUES (

t\_business\_person(1, 'John', 'Brown', 'Manager', 'XYZ Corp')

);

The following query calls display\_details() using object\_business\_customers:

**SELECT o.display\_details()**

**FROM object\_business\_customers o;**

O.DISPLAY\_DETAILS()

--------------------------------------------------------------------

id=1, name=John Brown, dob=01-FEB-55, title=Manager, company=XYZ Corp

As you can see, the id, name, and date of birth (dob) are displayed (which come from display\_details()in t\_person), followed by the title and company (which come from display\_details() in t\_business\_person).

**SUMMARY**

In this chapter, you learned the following:

Image The Oracle database allows you to create object types. An object type is like a class in Java, C++, and C#. An object type may contain attributes and methods; you create an object type using the CREATE TYPE statement.

Image You can use an object type to define a column object or an object table.

Image You can use an object reference to access an individual row in an object table. An object reference is similar to a pointer in C++.

Image You can create and manipulate objects in SQL and PL/SQL.

Image With the release of Oracle Database 9*i*, you can use object type inheritance. This allows you to define hierarchies of database types.

Image You can use a subtype object in place of a supertype object, which gives you great flexibility when storing and manipulating related types. If you want to prevent the use of a subtype object in place of supertype object, you can mark an object table or object column as NOT SUBSTITUTABLE.

Image You can use a number of useful functions with objects, such as REF(), DEREF(), VALUE(), IS OF(), SYS\_TYPEID(), and TREAT().

Image You can mark an object type as NOT INSTANTIABLE, which prevents objects of that type from being created. You’ll want to mark an object type as NOT INSTANTIABLE when you use that type as an abstract supertype and never actually create objects of that type.

Image You can define your own constructors to do things like programmatically setting a default for attributes of an object.

Image You can override a method in a supertype with a method in a subtype, giving you a very flexible way of defining methods in a hierarchy of types.

Image You can use the new Oracle Database 11*g* generalized invocation feature to call methods in supertype from a subtype. Doing this can save you a lot of work and make your code easier to maintain.

**CHAPTER 13  
Collections**

In this chapter, you will do the following:

Image Learn about collections

Image Learn how to create collection types

Image Use collection types to define columns in tables

Image Create and manipulate collection data in SQL and PL/SQL

Image Learn how a collection may itself contain embedded collections (a "multilevel" collection)

Image Examine the enhancements to collections that were introduced in Oracle Database 10*g*

**INTRODUCING COLLECTIONS**

Oracle Database 8 introduced two new database types, known as *collections*, that allow you to store sets of elements. Oracle Database 9*i* extended these features to include multilevel collections, which allow you to embed a collection within another collection. Oracle Database 10*g* further extended collections to include associative arrays and much more.

There are three types of collections:

Image **Varrays** A varray is similar to an array in Java, C++, and C#. A varray stores an ordered set of elements, and each element has an index that records its position in the array. Elements in a varray can be modified only as a whole, not individually; this means that even if you only want to modify one element, you must supply all the elements for the varray. A varray has a maximum size that you set when creating it, but you can change the size later.

Image **Nested tables** A nested table is a table that is embedded within another table. You can insert, update, and delete individual elements in a nested table; this makes them more flexible than a varray, whose elements can be modified only as a whole. A nested table doesn’t have a maximum size, and you can store an arbitrary number of elements in a nested table.

Image **Associative arrays (formerly known as index-by tables)** An associative array is similar to a hash table in Java. Introduced in Oracle Database 10*g*, an associative array is a set of key and value pairs. You can get the value from the array using the key (which may be a string) or an integer that specifies the position of the value in the array. An associative array can be used only in PL/SQL and cannot be stored in the database.

You might be asking yourself why you would want to use collections in the first place. After all, using two tables with a foreign key already allows you to model relationships between data. The answer is that collections follow the object-oriented style of modern programming; in addition, the data stored in the collection may be accessed more rapidly by the database than if you were to use two relational tables to store the same data.

I’ve provided an SQL\*Plus script named collection\_schema.sql in the SQL directory. The script creates a user named collection\_user with a password of collection\_password, and creates the collection types, tables, and PL/SQL code used in the first part of this chapter. You must run this script while logged in as a user with the required privileges to create a new user with the CONNECT and RESOURCE privileges; I log in as the system user on my database to run the scripts. After the script completes, you will be logged in as collection\_user.

**CREATING COLLECTION TYPES**

In this section, you’ll see how to create a varray type and a nested table type.

**Creating a Varray Type**

A varray stores an ordered set of elements, all of the same type, and the type can be a built-in database type or a user-defined object type. Each element has an index that corresponds to its position in the array, and you can modify elements in the varray only as a whole.

You create a varray type using the CREATE TYPE statement, in which you specify the maximum size and the type of elements stored in the varray. The following example creates a type named t\_varray\_address that can store up to three VARCHAR2 strings:

CREATE TYPE t\_varray\_address AS VARRAY(3) OF VARCHAR2(50);

/

Each VARCHAR2 will be used to represent a different address for a customer of our example store.

In Oracle Database 10*g* and higher, you can change the maximum number of elements of a varray using the ALTER TYPE statement. For example, the following statement alters the maximum number of elements to ten:

ALTER TYPE t\_varray\_address MODIFY LIMIT 10 CASCADE;

The CASCADE option propagates the change to any dependent objects in the database.

**Creating a Nested Table Type**

A nested table stores an unordered set of any number of elements. You can insert, update, and delete individual elements in a nested table. A nested table doesn’t have a maximum size, and you can store an arbitrary number of elements in a nested table.

In this section, you’ll see a nested table type that stores t\_address object types. You saw the use of t\_address in the previous chapter; it is used to represent an address and is defined as follows:

CREATE TYPE t\_address AS OBJECT (

street VARCHAR2(15),

city VARCHAR2(15),

state CHAR(2),

zip VARCHAR2(5)

);

/

You create a nested table type using the CREATE TYPE statement, and the following example creates a type named t\_nested\_table\_address that stores t\_address objects:

CREATE TYPE t\_nested\_table\_address AS TABLE OF t\_address;

/

Notice that you don’t specify the maximum size of a nested table. That’s because a nested table can store any number of elements.

**USING A COLLECTION TYPE TO DEFINE A COLUMN IN A TABLE**

Once you’ve created a collection type, you can use it to define a column in a table. You’ll see how to use the varray type and nested table type created in the previous section to define a column in a table.

**Using a Varray Type to Define a Column in a Table**

The following statement creates a table named customers\_with\_varray, which uses t\_varray\_address to define a column named addresses:

CREATE TABLE customers\_with\_varray (

id INTEGER PRIMARY KEY,

first\_name VARCHAR2(10),

last\_name VARCHAR2(10),

addresses t\_varray\_address

);

The elements in a varray are stored directly inside the table when the size of the varray is 4KB or less; otherwise, the varray is stored outside of the table. When a varray is stored with the table, accessing its elements is faster than accessing elements in a nested table.

**Using a Nested Table Type to Define a Column in a Table**

The following statement creates a table named customers\_with\_nested\_table, which uses t\_nested\_table\_address to define a column named addresses:

CREATE TABLE customers\_with\_nested\_table (

id INTEGER PRIMARY KEY,

first\_name VARCHAR2(10),

last\_name VARCHAR2(10),

addresses t\_nested\_table\_address

)

NESTED TABLE

addresses

STORE AS

nested\_addresses;

The NESTED TABLE clause identifies the name of the nested table column (addresses in the example), and the STORE AS clause specifies the name of the nested table (nested\_addresses in the example) where the actual elements are stored. You cannot access the nested table independently of the table in which it is embedded.

**GETTING INFORMATION ON COLLECTIONS**

As you’ll see in this section, you can use the DESCRIBE command and a couple of user views to get information on your collections.

**Getting Information on a Varray**

The following example describes t\_varray\_address:

**DESCRIBE t\_varray\_address**

t\_varray\_address VARRAY(3) OF VARCHAR2(50)

The next example describes the customers\_with\_varray table, whose addresses column is of the t\_varray\_address type:

**DESCRIBE customers\_with\_varray**

Name Null? Type

------------------------------- -------- ------------------

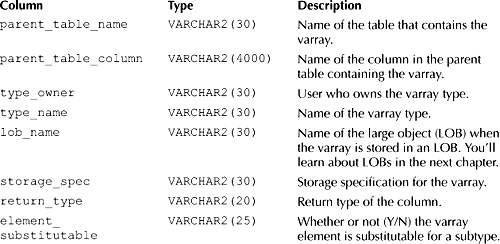
ID NOT NULL NUMBER(38)

FIRST\_NAME VARCHAR2(10)

LAST\_NAME VARCHAR2(10)

ADDRESSES T\_VARRAY\_ADDRESS

You can also get information on your varrays from the user\_varrays view. [Table 13-1](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch14.html#table_13-1) describes the columns in user\_varrays.



**TABLE 13-1** *Columns in the user\_varrays View*

Image

**NOTE**  
*You can get information on all the varrays you have access to using the* all\_varrays *view*.

The following example retrieves the details for t\_varray\_address from user\_varrays:

**SELECT parent\_table\_name, parent\_table\_column, type\_name**

**FROM user\_varrays**

**WHERE type\_name = 'T\_VARRAY\_ADDRESS';**

PARENT\_TABLE\_NAME

---------------------

PARENT\_TABLE\_COLUMN

---------------------

TYPE\_NAME

---------------------

CUSTOMERS\_WITH\_VARRAY

ADDRESSES

T\_VARRAY\_ADDRESS

**Getting Information on a Nested Table**

You can also use DESCRIBE with a nested table, as shown in the following example that describes t\_nested\_table\_address:

**DESCRIBE t\_nested\_table\_address**

t\_nested\_table\_address TABLE OF T\_ADDRESS

Name Null? Type

----------------------------------------- -------- ------------

STREET VARCHAR2(15)

CITY VARCHAR2(15)

STATE CHAR(2)

ZIP VARCHAR2(5)

The next example describes the customers\_with\_nested\_table table, whose addresses column is of type t\_nested\_table\_address:

**DESCRIBE customers\_with\_nested\_table**

Name Null? Type

---------------------------------- -------- ----------------------

ID NOT NULL NUMBER(38)

FIRST\_NAME VARCHAR2(10)

LAST\_NAME VARCHAR2(10)

ADDRESSES T\_NESTED\_TABLE\_ADDRESS

If you set the depth to 2 and describe customers\_with\_nested\_table, you can see the attributes that make up t\_nested\_table\_address:

**SET DESCRIBE DEPTH 2**

**DESCRIBE customers\_with\_nested\_table**

Name Null? Type

----------------------------------- -------- -------------

ID NOT NULL NUMBER(38)

FIRST\_NAME VARCHAR2(10)

LAST\_NAME VARCHAR2(10)

ADDRESSES T\_NESTED\_TABLE\_ADDRESS

STREET VARCHAR2(15)

CITY VARCHAR2(15)

STATE CHAR(2)

ZIP VARCHAR2(5)

You can also get information on your nested tables from the user\_nested\_tables view. [Table 13-2](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch14.html#table_13-2)describes the columns in user\_nested\_tables.

Image

**NOTE**  
*You can get information on all the nested tables you have access to using the* all\_nested\_tables*view*.

The following example retrieves the details for the nested\_addresses table from user\_nested\_tables:

**SELECT table\_name, table\_type\_name, parent\_table\_name, parent\_table\_column**

**FROM user\_nested\_tables**

**WHERE table\_name = 'NESTED\_ADDRESSES';**

TABLE\_NAME TABLE\_TYPE\_NAME

------------------------------ ----------------------

PARENT\_TABLE\_NAME

------------------------------

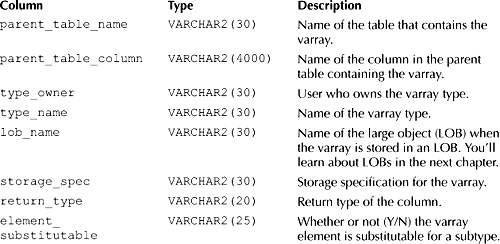
PARENT\_TABLE\_COLUMN

-----------------------------------------------------

NESTED\_ADDRESSES T\_NESTED\_TABLE\_ADDRESS

CUSTOMERS\_WITH\_NESTED\_TABLE

ADDRESSES



**TABLE 13-2** *Columns in the user\_nested\_tables View*

**POPULATING A COLLECTION WITH ELEMENTS**

In this section, you’ll see how to populate a varray and a nested table with elements using INSERT statements. You don’t have to run the INSERT statements shown in this section: they are executed when you run the collection\_schema.sql script.

**Populating a Varray with Elements**

The following INSERT statements add rows to the customers\_with\_varray table; notice the use of the t\_varray\_address constructor to specify the strings for the elements of the varray:

INSERT INTO customers\_with\_varray VALUES (

1, ’Steve', 'Brown',

t\_varray\_address(

'2 State Street, Beantown, MA, 12345',

'4 Hill Street, Lost Town, CA, 54321'

)

);

INSERT INTO customers\_with\_varray VALUES (

2, 'John', ’Smith',

t\_varray\_address(

'1 High Street, Newtown, CA, 12347',

'3 New Street, Anytown, MI, 54323',

'7 Market Street, Main Town, MA, 54323'

)

);

As you can see, the first row has two addresses and the second has three. Any number of addresses up to the maximum limit for the varray can be stored.

**Populating a Nested Table with Elements**

The following INSERT statements add rows to customers\_with\_nested\_table; notice the use of the t\_nested\_table\_address and t\_address constructors to specify the elements of the nested table:

INSERT INTO customers\_with\_nested\_table VALUES (

1, ’Steve', 'Brown',

t\_nested\_table\_address(

t\_address('2 State Street', 'Beantown', 'MA', '12345'),

t\_address('4 Hill Street', 'Lost Town', 'CA', '54321')

)

);

INSERT INTO customers\_with\_nested\_table VALUES (

2, 'John', ’Smith',

t\_nested\_table\_address(

t\_address('1 High Street', 'Newtown', 'CA', '12347'),

t\_address('3 New Street', 'Anytown', 'MI', '54323'),

t\_address('7 Market Street', 'Main Town', 'MA', '54323')

)

);

As you can see, the first row has two addresses and the second has three. Any number of addresses can be stored in a nested table.

**RETRIEVING ELEMENTS FROM COLLECTIONS**

In this section, you’ll see how to retrieve elements from a varray and a nested table using queries. The output from the queries has been formatted slightly to make the results more readable.

**Retrieving Elements from a Varray**

The following query retrieves customer #1 from the customers\_with\_varray table; one row is returned, and it contains the two addresses stored in the varray:

**SELECT \***

**FROM customers\_with\_varray**

**WHERE id = 1;**

ID FIRST\_NAME LAST\_NAME

---------- ---------- ----------

ADDRESSES

-------------------------------------------------------

1 Steve Brown

T\_VARRAY\_ADDRESS('2 State Street, Beantown, MA, 12345',

'4 Hill Street, Lost Town, CA, 54321')

The next query specifies the actual column names:

**SELECT id, first\_name, last\_name, addresses**

**FROM customers\_with\_varray**

**WHERE id = 1;**

ID FIRST\_NAME LAST\_NAME

---------- ---------- ----------

ADDRESSES

-------------------------------------------------------

1 Steve Brown

T\_VARRAY\_ADDRESS('2 State Street, Beantown, MA, 12345',

'4 Hill Street, Lost Town, CA, 54321')

These examples all return the addresses in the varray as a single row. Later, in the section "Using TABLE() to Treat a Collection as a Series of Rows," you’ll see how you can treat the data stored in a collection as a series of rows.

**Retrieving Elements from a Nested Table**

The following query retrieves customer #1 from customers\_with\_nested\_table; one row is returned, and it contains the two addresses stored in the nested table:

**SELECT \***

**FROM customers\_with\_nested\_table**

**WHERE id = 1;**

ID FIRST\_NAME LAST\_NAME

---------- ---------- ----------

ADDRESSES(STREET, CITY, STATE, ZIP)

--------------------------------------------------------

1 Steve Brown

T\_NESTED\_TABLE\_ADDRESS(

T\_ADDRESS('2 State Street', 'Beantown', 'MA', '12345'),

T\_ADDRESS('4 Hill Street', 'Lost Town', 'CA', '54321'))

The next query specifies the actual column names:

**SELECT id, first\_name, last\_name, addresses**

**FROM customers\_with\_nested\_table**

**WHERE id = 1;**

ID FIRST\_NAME LAST\_NAME

---------- ---------- ----------

ADDRESSES(STREET, CITY, STATE, ZIP)

--------------------------------------------------------

1 Steve Brown

T\_NESTED\_TABLE\_ADDRESS(

T\_ADDRESS('2 State Street', 'Beantown', 'MA', '12345'),

T\_ADDRESS('4 Hill Street', 'Lost Town', 'CA', '54321'))

The next query gets just the addresses nested table; as in the previous examples, one row is returned, and it contains the two addresses stored in the nested table:

**SELECT addresses**

**FROM customers\_with\_nested\_table**

**WHERE id = 1;**

ADDRESSES(STREET, CITY, STATE, ZIP)

--------------------------------------------------------

T\_NESTED\_TABLE\_ADDRESS(

T\_ADDRESS('2 State Street', 'Beantown', 'MA', '12345'),

T\_ADDRESS('4 Hill Street', 'Lost Town', 'CA', '54321'))

**USING TABLE() TO TREAT A COLLECTION AS A SERIES OF ROWS**

The previous queries you’ve seen in this chapter return the contents of a collection as a single row. Sometimes, you may wish to treat the data stored in a collection as a series of rows; for example, you might be working with a legacy application that can only use rows. To treat a collection as a series of rows, you use the TABLE()function. In this section, you’ll see how to use TABLE() with a varray and a nested table.

**Using TABLE() with a Varray**

The following query uses TABLE() to retrieve customer #1’s two addresses from the customers\_with\_varray table; two separate rows are returned:

**SELECT a.\***

**FROM customers\_with\_varray c, TABLE(c.addresses) a**

**WHERE id = 1;**

COLUMN\_VALUE

-----------------------------------

2 State Street, Beantown, MA, 12345

4 Hill Street, Lost Town, CA, 54321

Notice how the Oracle database software automatically adds the column name of COLUMN\_VALUE to the rows returned by the query. COLUMN\_VALUE is a pseudo column alias, and it is automatically added when a collection contains data of one of the built-in data types, like VARCHAR2, CHAR, NUMBER, or DATE. Because the example varray contains VARCHAR2 data, the COLUMN\_VALUE alias is added. If the varray had contained data of a user-defined object type, then TABLE() would return objects of that type and COLUMN\_VALUE would not appear; you’ll see an example of this in the next section.

You can also embed an entire SELECT statement inside TABLE(). For example, the following query rewrites the previous example, placing a SELECT inside TABLE():

**SELECT \***

**FROM TABLE(**

**-- get the addresses for customer #1**

**SELECT addresses**

**FROM customers\_with\_varray**

**WHERE id = 1**

**);**

COLUMN\_VALUE

-----------------------------------

2 State Street, Beantown, MA, 12345

4 Hill Street, Lost Town, CA, 54321

The following query shows another example that uses TABLE() to get the addresses:

**SELECT c.id, c.first\_name, c.last\_name, a.\***

**FROM customers\_with\_varray c, TABLE(c.addresses) a**

**WHERE id = 1;**

ID FIRST\_NAME LAST\_NAME

---------- ---------- ----------

COLUMN\_VALUE

-----------------------------------

1 Steve Brown

2 State Street, Beantown, MA, 12345

1 Steve Brown

4 Hill Street, Lost Town, CA, 54321

**Using TABLE() with a Nested Table**

The following query uses TABLE() to retrieve customer #1’s two addresses from customers\_with\_nested\_table; notice that two separate rows are returned:

**SELECT a.\***

**FROM customers\_with\_nested\_table c, TABLE(c.addresses) a**

**WHERE id = 1;**

STREET CITY ST ZIP

--------------- --------------- -- -----

2 State Street Beantown MA 12345

4 Hill Street Lost Town CA 54321

The next query gets the street and state attributes of the addresses:

**SELECT a.street, a.state**

**FROM customers\_with\_nested\_table c, TABLE(c.addresses) a**

**WHERE id = 1;**

STREET ST

--------------- --

2 State Street MA

4 Hill Street CA

The following query shows another example that uses TABLE() to get the addresses:

**SELECT c.id, c.first\_name, c.last\_name, a.\***

**FROM customers\_with\_nested\_table c, TABLE(c.addresses) a**

**WHERE c.id = 1;**

ID FIRST\_NAME LAST\_NAME STREET CITY ST ZIP

------ ---------- ---------- --------------- -------------- -- -----

1 Steve Brown 2 State Street Beantown MA 12345

1 Steve Brown 4 Hill Street Lost Town CA 54321

You’ll see an important use of TABLE() later in the section "Modifying Elements of a Nested Table."

**MODIFYING ELEMENTS OF COLLECTIONS**

In this section, you’ll see how to modify the elements in a varray and a nested table. You should feel free to run the UPDATE, INSERT, and DELETE statements shown in this section.

**Modifying Elements of a Varray**

The elements in a varray can be modified only as a whole, which means that even if you only want to modify one element, you must supply all the elements for the varray. The following UPDATE statement modifies the addresses of customer #2 in the customers\_with\_varray table:

**UPDATE customers\_with\_varray**

**SET addresses = t\_varray\_address(**

**'6 Any Street, Lost Town, GA, 33347'**,

**'3 New Street, Anytown, MI, 54323'**,

**'7 Market Street, Main Town, MA, 54323'**

**)**

**WHERE id = 2;**

1 row updated.

**Modifying Elements of a Nested Table**

Unlike in a varray, elements in a nested table can be modified individually. You can insert, update, and delete individual elements in a nested table; you’ll see how to do all three of these modifications in this section.

The following INSERT statement adds an address to customer #2 in customer\_with\_nested\_table; notice that TABLE() is used to get the addresses as a series of rows:

**INSERT INTO TABLE(**

**-- get the addresses for customer #2**

**SELECT addresses**

**FROM customers\_with\_nested\_table**

**WHERE id = 2**

**) VALUES (**

**t\_address('5 Main Street', 'Uptown', 'NY', '55512')**

**);**

1 row created.

The following UPDATE statement changes the '1 High Street' address of customer #2 to '9 Any Street'; notice the use of the alias addr in the VALUE clauses when specifying the addresses:

**UPDATE TABLE(**

**-- get the addresses for customer #2**

**SELECT addresses**

**FROM customers\_with\_nested\_table**

**WHERE id = 2**

**) addr**

**SET VALUE(addr) =**

**t\_address('9 Any Street', 'Lost Town', 'VA', '74321')**

**WHERE VALUE(addr) =**

**t\_address('1 High Street', 'Newtown', 'CA', '12347');**

1 row updated.

The following DELETE statement removes the '3 New Street…' address from customer #2:

**DELETE FROM TABLE(**

**-- get the addresses for customer #2**

**SELECT addresses**

**FROM customers\_with\_nested\_table**

**WHERE id = 2**

**) addr**

**WHERE VALUE(addr) =**

**t\_address('3 New Street', 'Anytown', 'MI', '54323');**

1 row deleted.

**USING A MAP METHOD TO COMPARE THE CONTENTS OF NESTED TABLES**

You can compare the contents of one nested table with the contents of another. Two nested tables are equal only if

Image They are of the same type.

Image They have the same number of rows.

Image All their elements contain the same values.

If the elements of the nested table are of a built-in database type, like NUMBER, VARCHAR2, and so on, then the database will automatically compare the contents of the nested tables for you. If, however, the elements are of a user-defined object type, then you will need to provide a map function that contains code to compare the objects (map functions were shown in the section "Comparing Object Values" of the previous chapter).

The following statements create a type named t\_address2 that contains a map function named get\_string(); notice that get\_string() returns a VARCHAR2 containing the values for the zip, state, city, and street attributes:

CREATE TYPE t\_address2 AS OBJECT (

street VARCHAR2(15),

city VARCHAR2(15),

state CHAR(2),

zip VARCHAR2(5),

-- declare the get\_string() map function,

-- which returns a VARCHAR2 string

MAP MEMBER FUNCTION get\_string RETURN VARCHAR2

);

/

CREATE TYPE BODY t\_address2 AS

-- define the get\_string() map function

MAP MEMBER FUNCTION get\_string RETURN VARCHAR2 IS

BEGIN

-- return a concatenated string containing the

-- zip, state, city, and street attributes

RETURN zip || ' ' || state || ' ' || city || ' ' || street;

END get\_string;

END;

/

As you’ll see shortly, the database will automatically call get\_string() when comparing t\_address2objects.

The following statements create a nested table type and a table, and add a row to the table:

CREATE TYPE t\_nested\_table\_address2 AS TABLE OF t\_address2;

/

CREATE TABLE customers\_with\_nested\_table2 (

id INTEGER PRIMARY KEY,

first\_name VARCHAR2(10),

last\_name VARCHAR2(10),

addresses t\_nested\_table\_address2

)

NESTED TABLE

addresses

STORE AS

nested\_addresses2;

INSERT INTO customers\_with\_nested\_table2 VALUES (

1, ’Steve', 'Brown',

t\_nested\_table\_address2(

t\_address2('2 State Street', 'Beantown', 'MA', '12345'),

t\_address2('4 Hill Street', 'Lost Town', 'CA', '54321')

)

);

The following query includes a nested table in the WHERE clause; notice that the addresses after the = in the WHERE clause are the same as those in the previous INSERT statement:

**SELECT cn.id, cn.first\_name, cn.last\_name**

**FROM customers\_with\_nested\_table2 cn**

**WHERE cn.addresses =**

**t\_nested\_table\_address2(**

**t\_address2('2 State Street', 'Beantown', 'MA', '12345')**,

**t\_address2('4 Hill Street', 'Lost Town', 'CA', '54321')**

**);**

ID FIRST\_NAME LAST\_NAME

---------- ---------- ----------

1 Steve Brown

When the query is executed, the database automatically calls get\_string() to compare the t\_address2objects in cn.addresses to the t\_address2 objects after the = in the WHERE clause. The get\_string() function returns a VARCHAR2 string containing the zip, state, city, and streetattributes of the objects, and when the strings are equal for every object, the nested tables are also equal.

The next query returns no rows because the single address after the = in the WHERE clause matches only one of the addresses in cn.addresses (remember: two nested tables are equal only if they are of the same type, *have the same number of rows*, and their elements contain the same values):

**SELECT cn.id, cn.first\_name, cn.last\_name**

**FROM customers\_with\_nested\_table2 cn**

**WHERE cn.addresses =**

**t\_nested\_table\_address2(**

**t\_address2('4 Hill Street', 'Lost Town', 'CA', '54321')**

**);**

no rows selected

In Oracle Database 10*g* and higher, you can use the SUBMULTISET operator to check whether the contents of one nested table are a subset of another nested table. The following query rewrites the previous example and returns a row:

**SELECT cn.id, cn.first\_name, cn.last\_name**

**FROM customers\_with\_nested\_table2 cn**

**WHERE**

**t\_nested\_table\_address2(**

**t\_address2('4 Hill Street', 'Lost Town', 'CA', '54321')**

**)**

**SUBMULTISET OF cn.addresses;**

ID FIRST\_NAME LAST\_NAME

---------- ---------- ----------

1 Steve Brown

Because the address in the first part of the WHERE clause is a subset of the addresses in cn.addresses, a match is found and a row is returned.

The following query shows another example; this time the addresses in cn.addresses are a subset of the addresses after OF in the WHERE clause:

**SELECT cn.id, cn.first\_name, cn.last\_name**

**FROM customers\_with\_nested\_table2 cn**

**WHERE**

**cn.addresses SUBMULTISET OF**

**t\_nested\_table\_address2(**

**t\_address2('2 State Street', 'Beantown', 'MA', '12345')**,

**t\_address2('4 Hill Street', 'Lost Town', 'CA', '54321')**,

**t\_address2('6 State Street', 'Beantown', 'MA', '12345')**

**);**

ID FIRST\_NAME LAST\_NAME

---------- ---------- ----------

1 Steve Brown

You’ll learn more about the SUBMULTISET operator later in this chapter in the section "SUBMULTISET Operator." Also, in the section "Equal and Not-Equal Operators," you’ll see how to use the ANSI operators implemented in Oracle Database 10*g* to compare nested tables.

Image

**NOTE**  
*There is no direct mechanism for comparing the contents of varrays*.

**USING CAST() TO CONVERT COLLECTIONS FROM ONE TYPE TO ANOTHER**

You may use CAST() to convert a collection of one type to another collection type. In this section, you’ll see how to use CAST() to convert a varray to a nested table and vice versa.

**Using CAST() to Convert a Varray to a Nested Table**

The following statements create and populate a table named customers\_with\_varray2 that contains an addresses column of type t\_varray\_address2:

CREATE TYPE t\_varray\_address2 AS VARRAY(3) OF t\_address;

/

CREATE TABLE customers\_with\_varray2 (

id INTEGER PRIMARY KEY,

first\_name VARCHAR2(10),

last\_name VARCHAR2(10),

addresses t\_varray\_address2

);

INSERT INTO customers\_with\_varray2 VALUES (

1, 'Jason', 'Bond',

t\_varray\_address2(

t\_address('9 Newton Drive', ’Sometown', 'WY', '22123'),

t\_address('6 Spring Street', 'New City', 'CA', '77712')

)

);

The following query uses CAST() to return the varray addresses for customer #1 as a nested table; notice that the addresses appear in a constructor for the T\_NESTED\_TABLE\_ADDRESS type, indicating the conversion of the elements to this type:

**SELECT CAST(cv.addresses AS t\_nested\_table\_address)**

**FROM customers\_with\_varray2 cv**

**WHERE cv.id = 1;**

CAST(CV.ADDRESSESAST\_NESTED\_TABLE\_ADDRESS)(STREET, CITY, STATE, ZIP)

--------------------------------------------------------------------

T\_NESTED\_TABLE\_ADDRESS(

T\_ADDRESS('9 Newton Drive', ’Sometown', 'WY', '22123'),

T\_ADDRESS('6 Spring Street', 'New City', 'CA', '77712'))

**Using CAST() to Convert a Nested Table to a Varray**

The following query uses CAST() to return the addresses for customer #1 in customers\_with\_nested\_table as a varray; notice that the addresses appear in a constructor for T\_VARRAY\_ADDRESS2:

**SELECT CAST(cn.addresses AS t\_varray\_address2)**

**FROM customers\_with\_nested\_table cn**

**WHERE cn.id = 1;**

CAST(CN.ADDRESSESAST\_VARRAY\_ADDRESS2)(STREET, CITY, STATE, ZIP)

---------------------------------------------------------------

T\_VARRAY\_ADDRESS2(

T\_ADDRESS('2 State Street', 'Beantown', 'MA', '12345'),

T\_ADDRESS('4 Hill Street', 'Lost Town', 'CA', '54321'))

**USING COLLECTIONS IN PL/SQL**

You can use collections in PL/SQL. In this section, you’ll see how to perform the following tasks in PL/SQL:

Image Manipulate a varray

Image Manipulate a nested table

Image Use the PL/SQL collection methods to access and manipulate collections

All the packages you’ll see in this section are created when you run the collection\_schema.sql script. If you performed any of the INSERT, UPDATE, or DELETE statements shown in the earlier sections of this chapter, go ahead and rerun the collection\_schema.sql script so that your output matches mine in this section.

**Manipulating a Varray**

In this section, you’ll see a package named varray\_package; this package contains the following items:

Image A REF CURSOR type named t\_ref\_cursor

Image A function named get\_customers(), which returns a t\_ref\_cursor object that points to the rows in the customers\_with\_varray table

Image A procedure named insert\_customer(), which adds a row to the customers\_with\_varray table

The collection\_schema.sql script contains the following package specification and body for varray\_package:

CREATE PACKAGE varray\_package AS

TYPE t\_ref\_cursor IS REF CURSOR;

FUNCTION get\_customers RETURN t\_ref\_cursor;

PROCEDURE insert\_customer(

p\_id IN customers\_with\_varray.id%TYPE,

p\_first\_name IN customers\_with\_varray.first\_name%TYPE,

p\_last\_name IN customers\_with\_varray.last\_name%TYPE,

p\_addresses IN customers\_with\_varray.addresses%TYPE

);

END varray\_package;

/

CREATE PACKAGE BODY varray\_package AS

-- get\_customers() function returns a REF CURSOR

-- that points to the rows in customers\_with\_varray

FUNCTION get\_customers

RETURN t\_ref\_cursor IS

--declare the REF CURSOR object

v\_customers\_ref\_cursor t\_ref\_cursor;

BEGIN

-- get the REF CURSOR

OPEN v\_customers\_ref\_cursor FOR

SELECT \*

FROM customers\_with\_varray;

-- return the REF CURSOR

RETURN customers\_ref\_cursor;

END get\_customers;

-- insert\_customer() procedure adds a row to

-- customers\_with\_varray

PROCEDURE insert\_customer(

p\_id IN customers\_with\_varray.id%TYPE,

p\_first\_name IN customers\_with\_varray.first\_name%TYPE,

p\_last\_name IN customers\_with\_varray.last\_name%TYPE,

p\_addresses IN customers\_with\_varray.addresses%TYPE

) IS

BEGIN

INSERT INTO customers\_with\_varray

VALUES (p\_id, p\_first\_name, p\_last\_name, p\_addresses);

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

END insert\_customer;

END varray\_package;

/

The following example calls insert\_customer() to add a new row to the customers\_with\_varraytable:

**CALL varray\_package.insert\_customer(**

**3, 'James', ’Red'**,

**t\_varray\_address(**

**'10 Main Street, Green Town, CA, 22212'**,

**'20 State Street, Blue Town, FL, 22213'**

**)**

**);**

Call completed.

The next example calls get\_products() to retrieve the rows from customers\_with\_varray:

**SELECT varray\_package.get\_customers**

**FROM dual;**

GET\_CUSTOMERS

--------------------

CURSOR STATEMENT : 1

CURSOR STATEMENT : 1

ID FIRST\_NAME LAST\_NAME

---------- ---------- ----------

ADDRESSES

---------------------------------------------------------

1 Steve Brown

T\_VARRAY\_ADDRESS('2 State Street, Beantown, MA, 12345',

'4 Hill Street, Lost Town, CA, 54321')

2 John Smith

T\_VARRAY\_ADDRESS('1 High Street, Newtown, CA, 12347',

'3 New Street, Anytown, MI, 54323',

'7 Market Street, Main Town, MA, 54323')

3 James Red

T\_VARRAY\_ADDRESS('10 Main Street, Green Town, CA, 22212',

'20 State Street, Blue Town, FL, 22213')

**Manipulating a Nested Table**

In this section, you’ll see a package named nested\_table\_package; this package contains the following items:

Image A REF CURSOR type named t\_ref\_cursor

Image A function named get\_customers(), which returns a t\_ref\_cursor object that points to the rows in customers\_with\_nested\_table

Image A procedure named insert\_customer(), which adds a row to customers\_with\_nested\_table

The collection\_schema.sql script contains the following package specification and body for nested\_table\_package:

CREATE PACKAGE nested\_table\_package AS

TYPE t\_ref\_cursor IS REF CURSOR;

FUNCTION get\_customers RETURN t\_ref\_cursor;

PROCEDURE insert\_customer(

p\_id IN customers\_with\_nested\_table.id%TYPE,

p\_first\_name IN customers\_with\_nested\_table.first\_name%TYPE,

p\_last\_name IN customers\_with\_nested\_table.last\_name%TYPE,

p\_addresses IN customers\_with\_nested\_table.addresses%TYPE

);

END nested\_table\_package;

/

CREATE PACKAGE BODY nested\_table\_package AS

-- get\_customers() function returns a REF CURSOR

-- that points to the rows in customers\_with\_nested\_table

FUNCTION get\_customers

RETURN t\_ref\_cursor IS

-- declare the REF CURSOR object

v\_customers\_ref\_cursor t\_ref\_cursor;

BEGIN

-- get the REF CURSOR

OPEN v\_customers\_ref\_cursor FOR

SELECT \*

FROM customers\_with\_nested\_table;

-- return the REF CURSOR

RETURN customers\_ref\_cursor;

END get\_customers;

-- insert\_customer() procedure adds a row to

-- customers\_with\_nested\_table

PROCEDURE insert\_customer(

p\_id IN customers\_with\_nested\_table.id%TYPE,

p\_first\_name IN customers\_with\_nested\_table.first\_name%TYPE,

p\_last\_name IN customers\_with\_nested\_table.last\_name%TYPE,

p\_addresses IN customers\_with\_nested\_table.addresses%TYPE

) IS

BEGIN

INSERT INTO customers\_with\_nested\_table

VALUES (p\_id, p\_first\_name, p\_last\_name, p\_addresses);

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

END insert\_customer;

END nested\_table\_package;

/

The following example calls insert\_customer() to add a new row to customers\_with\_nested\_table:

**CALL nested\_table\_package.insert\_customer(**

**3, 'James', ’Red'**,

**t\_nested\_table\_address(**

**t\_address('10 Main Street', 'Green Town', 'CA', '22212')**,

**t\_address('20 State Street', 'Blue Town', 'FL', '22213')**

**)**

**);**

Call completed.

The next example calls get\_customers() to retrieve the rows from customers\_with\_nested\_table:

**SELECT nested\_table\_package.get\_customers**

**FROM dual;**

GET\_CUSTOMERS

--------------------

CURSOR STATEMENT : 1

CURSOR STATEMENT : 1

ID FIRST\_NAME LAST\_NAME

---------- ---------- ----------

ADDRESSES(STREET, CITY, STATE, ZIP)

------------------------------------------------------------

1 Steve Brown

T\_NESTED\_TABLE\_ADDRESS(

T\_ADDRESS('2 State Street', 'Beantown', 'MA', '12345'),

T\_ADDRESS('4 Hill Street', 'Lost Town', 'CA', '54321'))

2 John Smith

T\_NESTED\_TABLE\_ADDRESS(

T\_ADDRESS('1 High Street', 'Newtown', 'CA', '12347'),

T\_ADDRESS('3 New Street', 'Anytown', 'MI', '54323'),

T\_ADDRESS('7 Market Street', 'Main Town', 'MA', '54323'))

3 James Red

T\_NESTED\_TABLE\_ADDRESS(

T\_ADDRESS('10 Main Street', 'Green Town', 'CA', '22212'),

T\_ADDRESS('20 State Street', 'Blue Town', 'FL', '22213'))

**PL/SQL Collection Methods**

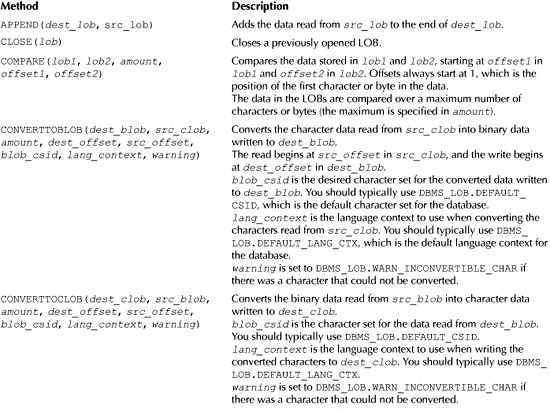
In this section, you’ll see the PL/SQL methods you can use with collections. [Table 13-3](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch14.html#table_13-3) summarizes the collection methods. These methods can be used only in PL/SQL.

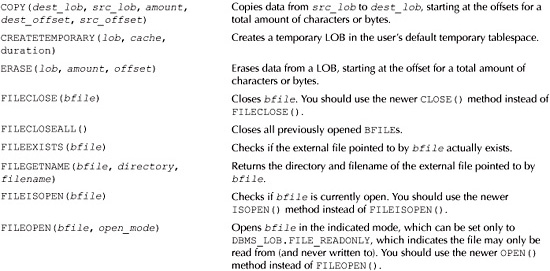
The following sections use a package named collection\_method\_examples; the examples illustrate the use of the methods shown in the previous table. The package is created by the collection\_schema.sqlscript, and you’ll see the individual methods defined in this package in the following sections.

**COUNT()**

COUNT returns the number of elements in a collection. Because a nested table can have individual elements that are empty, COUNT returns the number of non-empty elements in a nested table. For example, let’s say you have a nested table named v\_nested\_table that has its elements set as shown in the following table.







**TABLE 13-3** *PL/SQL Collection Methods*

Given this configuration, v\_nested\_table.COUNT returns 2, the number of non-empty elements.

COUNT is used in the get\_addresses() and display\_addresses() methods of the collection\_method\_examples package. The get\_addresses() function returns the specified customer’s addresses from customers\_with\_nested\_table, whose id is passed to the function:

FUNCTION get\_addresses(

p\_id customers\_with\_nested\_table.id%TYPE

) RETURN t\_nested\_table\_address IS

-- declare object named v\_addresses to store the

-- nested table of addresses

v\_addresses t\_nested\_table\_address;

BEGIN

-- retrieve the nested table of addresses into v\_addresses

SELECT addresses

INTO v\_addresses

FROM customers\_with\_nested\_table

WHERE id = p\_id;

-- display the number of addresses using v\_addresses.COUNT

DBMS\_OUTPUT.PUT\_LINE(

'Number of addresses = '|| **v\_addresses.COUNT**

);

-- return v\_addresses

RETURN v\_addresses;

END get\_addresses;

The following example sets the server output on and calls get\_addresses() for customer #1:

**SET SERVEROUTPUT ON**

**SELECT collection\_method\_examples.get\_addresses(1) addresses**

**FROM dual;**

ADDRESSES(STREET, CITY, STATE, ZIP)

--------------------------------------------------------

T\_NESTED\_TABLE\_ADDRESS(

T\_ADDRESS('2 State Street', 'Beantown', 'MA', '12345'),

T\_ADDRESS('4 Hill Street', 'Lost Town', 'CA', '54321'))

Number of addresses = 2

The following display\_addresses() procedure accepts a parameter named p\_addresses, which contains a nested table of addresses; the procedure displays the number of addresses in p\_addresses using COUNT, and then displays those addresses using a loop:

PROCEDURE display\_addresses(

p\_addresses t\_nested\_table\_address

) IS

v\_count INTEGER;

BEGIN

-- display the number of addresses in p\_addresses

DBMS\_OUTPUT.PUT\_LINE(

'Current number of addresses = '|| **p\_addresses.COUNT**

);

-- display the addresses in p\_addresses using a loop

FOR v\_count IN 1..**p\_addresses.COUNT** LOOP

DBMS\_OUTPUT.PUT\_LINE('Address #' || v\_count || ':');

DBMS\_OUTPUT.PUT(p\_addresses(v\_count).street || ', ');

DBMS\_OUTPUT.PUT(p\_addresses(v\_count).city || ', ');

DBMS\_OUTPUT.PUT(p\_addresses(v\_count).state || ', ');

DBMS\_OUTPUT.PUT\_LINE(p\_addresses(v\_count).zip);

END LOOP;

END display\_addresses;

You’ll see the use of display\_addresses () shortly.

**DELETE()**

DELETE removes elements from a collection. There are three forms of DELETE:

Image DELETE removes all elements.

Image DELETE (*n*) removes element *n*.

Image DELETE (*n, m*) removes elements *n* through *m*.

For example, let’s say you have a nested table named v\_nested\_table that has seven elements, then v\_nested\_table.DELETE(2, 5) removes elements 2 through 5.

The following delete\_address() procedure gets the addresses for customer #1 and then uses DELETE to remove the address whose index is specified by the p\_address\_num parameter:

PROCEDURE delete\_address(

p\_address\_num INTEGER

) IS

v\_addresses t\_nested\_table\_address;

BEGIN

v\_addresses := get\_addresses(1);

display\_addresses(v\_addresses);

DBMS\_OUTPUT.PUT\_LINE(’Deleting address #' || p\_address\_num);

-- delete the address specified by p\_address\_num

**v\_addresses.DELETE(p\_address\_num);**

display\_addresses(v\_addresses);

END delete\_address;

The following example calls delete\_address(2) to remove address #2 from customer #1:

**CALL collection\_method\_examples.delete\_address(2);**

Number of addresses = 2

Current number of addresses = 2

Address #1:

2 State Street, Beantown, MA, 12345

Address #2:

4 Hill Street, Lost Town, CA, 54321

Deleting address #2

Current number of addresses = 1

Address #1:

2 State Street, Beantown, MA, 12345

**EXISTS()**

EXISTS (*n* ) returns true if element *n* in a collection exists: EXISTS returns true for non-empty elements, and it returns false for empty elements of nested tables or elements beyond the range of a collection. For example, let’s say you have a nested table named v\_nested\_table that has its elements set as shown in the following table.



Given this configuration, v\_nested\_table.EXISTS(2) returns true (because element #2 is not empty), and v\_nested\_table.EXISTS(3) returns false (because element #3 is empty).

The following exist\_addresses() procedure gets the addresses for customer #1, uses DELETE to remove address #1, and then uses EXISTS to check whether addresses #1 and #2 exist (#1 does not exist because it has been deleted, #2 does exist):

PROCEDURE exist\_addresses IS

v\_addresses t\_nested\_table\_address;

BEGIN

v\_addresses := get\_addresses(1);

DBMS\_OUTPUT.PUT\_LINE(’Deleting address #1');

v\_addresses.DELETE(1);

-- use EXISTS to check if the addresses exist

IF **v\_addresses.EXISTS(1)** THEN

DBMS\_OUTPUT.PUT\_LINE('Address #1 does exist');

ELSE

DBMS\_OUTPUT.PUT\_LINE('Address #1 does not exist');

END IF;

IF **v\_addresses.EXISTS(2)** THEN

DBMS\_OUTPUT.PUT\_LINE('Address #2 does exist');

END IF;

END exist\_addresses;

The following example calls exist\_addresses():

**CALL collection\_method\_examples.exist\_addresses();**

Number of addresses = 2

Deleting address #1

Address #1 does not exist

Address #2 does exist

**EXTEND()**

EXTEND adds elements to the end of a collection. There are three forms of EXTEND:

Image EXTEND adds one element, which is set to null.

Image EXTEND(*n*) adds *n* elements, which are set to null.

Image EXTEND(*n, m*) adds *n* elements, which are set to a copy of the *m* element.

For example, let’s say you have a collection named v\_nested\_table that has seven elements, then v\_nested\_table.EXTEND(2, 5) adds element #5 twice to the end of the collection.

The following extend\_addresses() procedure gets the addresses for customer #1 into v\_addresses, then uses EXTEND to copy address #1 twice to the end of v\_addresses:

PROCEDURE extend\_addresses IS

v\_addresses t\_nested\_table\_address;

BEGIN

v\_addresses := get\_addresses(1);

display\_addresses(v\_addresses);

DBMS\_OUTPUT.PUT\_LINE('Extending addresses');

-- copy address #1 twice to the end of v\_addresses

**v\_addresses.EXTEND(2, 1);**

display\_addresses(v\_addresses);

END extend\_addresses;

The following example calls extend\_addresses():

**CALL collection\_method\_examples.extend\_addresses();**

Number of addresses = 2

Current number of addresses = 2

Address #1:

2 State Street, Beantown, MA, 12345

Address #2:

4 Hill Street, Lost Town, CA, 54321

Extending addresses

Current number of addresses = 4

Address #1:

2 State Street, Beantown, MA, 12345

Address #2:

4 Hill Street, Lost Town, CA, 54321

Address #3:

2 State Street, Beantown, MA, 12345

Address #4:

2 State Street, Beantown, MA, 12345

**FIRST()**

You use FIRST to get the index of the first element in a collection. If the collection is completely empty, FIRSTreturns null. Because a nested table can have individual elements that are empty, FIRST returns the lowest index of a non-empty element in a nested table. For example, let’s say you have a nested table named v\_nested\_table that has its elements set as shown in the following table.



Given this configuration, v\_nested\_table.FIRST returns 2, the lowest index containing a non-empty element.

The following first\_address() procedure gets the addresses for customer #1 into v\_addresses and then uses FIRST to display the index of the first address in v\_addresses; the procedure then deletes address #1 using DELETE and displays the new index returned by FIRST:

PROCEDURE first\_address IS

v\_addresses t\_nested\_table\_address;

BEGIN

v\_addresses := get\_addresses(1);

-- display the FIRST address

DBMS\_OUTPUT.PUT\_LINE('First address = ' || **v\_addresses.FIRST**);

DBMS\_OUTPUT.PUT\_LINE(’Deleting address #1');

v\_addresses.DELETE(1);

-- display the FIRST address again

DBMS\_OUTPUT.PUT\_LINE('First address = ' || **v\_addresses.FIRST**);

END first\_address;

The following example calls first\_address():

**CALL collection\_method\_examples.first\_address();**

Number of addresses = 2

First address = 1

Deleting address #1

First address = 2

**LAST()**

LAST returns the index of the last element in a collection. If the collection is completely empty, LAST returns null. Because a nested table can have individual elements that are empty, LAST returns the highest index of a non-empty element in a nested table. For example, let’s say you have a nested table named v\_nested\_tablethat has its elements set as shown in the following table.



Given this configuration, v\_nested\_table.LAST returns 4, the highest index containing a non-empty element.

The following last\_address() procedure gets the addresses for customer #1 into v\_addresses and then uses LAST to display the index of the last address in v\_addresses; the procedure then deletes address #2 using DELETE and displays the new index returned by LAST:

PROCEDURE last\_address IS

v\_addresses t\_nested\_table\_address;

BEGIN

v\_addresses := get\_addresses(1);

-- display the LAST address

DBMS\_OUTPUT.PUT\_LINE('Last address = ' || **v\_addresses.LAST**);

DBMS\_OUTPUT.PUT\_LINE(’Deleting address #2');

v\_addresses.DELETE(2);

-- display the LAST address again

DBMS\_OUTPUT.PUT\_LINE('Last address = ' || **v\_addresses.LAST**);

END last\_address;

The following example calls last\_address():

**CALL collection\_method\_examples.last\_address();**

Number of addresses = 2

Last address = 2

Deleting address #2

Last address = 1

**NEXT()**

NEXT(*n*) returns the index of the element after *n*. Because a nested table can have individual elements that are empty, NEXT returns the index of a non-empty element after *n*. If there are no elements after *n*, NEXT returns null. For example, let’s say you have a nested table named v\_nested\_table that has its elements set as shown in the following table.



Given this configuration, v\_nested\_table.NEXT(1) returns 4, the index containing the next non-empty element; v\_nested\_table.NEXT(4) returns null.

The following next\_address() procedure gets the addresses for customer #1 into v\_addresses and then uses NEXT(1) to get the index of the address after address #1 in v\_addresses; the procedure then uses NEXT(2) to attempt to get the index of the address after address #2 (there isn’t one, because customer #1 only has two addresses, so null is returned):

PROCEDURE next\_address IS

v\_addresses t\_nested\_table\_address;

BEGIN

v\_addresses := get\_addresses(1);

-- use NEXT(1) to get the index of the address

-- after address #1

DBMS\_OUTPUT.PUT\_LINE(

'v\_addresses.NEXT(1) = ' || **v\_addresses.NEXT(1)**

);

-- use NEXT(2) to attempt to get the index of

-- the address after address #2 (there isn’t one,

-- so null is returned)

DBMS\_OUTPUT.PUT\_LINE(

'v\_addresses.NEXT(2) = ' || **v\_addresses.NEXT(2)**

);

END next\_address;

The following example calls next\_address(); v\_addresses.NEXT(2) is null, and so no output is shown after the = for that element:

**CALL collection\_method\_examples.next\_address();**

Number of addresses = 2

v\_addresses.NEXT(1) = 2

v\_addresses.NEXT(2) =

**PRIOR()**

PRIOR(*n*) returns the index of the element before *n*. Because a nested table can have individual elements that are empty, PRIOR returns the index of a non-empty element before *n*. If there are no elements before *n*, PRIOR returns null. For example, let’s say you have a nested table named v\_nested\_table that has its elements set as shown in the following table.



Given this configuration, v\_nested\_table. PRIOR(4) returns 1, the index containing the prior non-empty element; v\_nested\_table. PRIOR(1) returns null.

The following prior\_address() procedure gets the addresses for customer #1 into v\_addresses and then uses PRIOR(2) to get the index of the address before address #2 in v\_addresses; the procedure then uses PRIOR(1) to attempt to get the index of the address before address #1 (there isn’t one, so null is returned):

PROCEDURE prior\_address IS

v\_addresses t\_nested\_table\_address;

BEGIN

v\_addresses := get\_addresses(1);

-- use PRIOR(2) to get the index of the address

-- before address #2

DBMS\_OUTPUT.PUT\_LINE(

'v\_addresses.PRIOR(2) = ' || **v\_addresses.PRIOR(2)**

);

-- use PRIOR(1) to attempt to get the index of

-- the address before address #1 (there isn’t one,

-- so null is returned)

DBMS\_OUTPUT.PUT\_LINE(

'v\_addresses.PRIOR(1) = ' || **v\_addresses.PRIOR(1)**

);

END prior\_address;

The following example calls prior\_address(); v\_addresses.PRIOR(1) is null, and so no output is shown after the = for that element:

**CALL collection\_method\_examples.prior\_address();**

Number of addresses = 2

v\_addresses.PRIOR(2) = 1

v\_addresses.PRIOR(1) =

**TRIM()**

TRIM removes elements from the end of a collection. There are two forms of TRIM:

Image TRIM removes one element from the end.

Image TRIM(*n* ) removes *n* elements from the end.

For example, let’s say you have a nested table named v\_nested\_table, then v\_nested\_table.TRIM(2) removes two elements from the end.

The following trim\_addresses() procedure gets the addresses of customer #1, copies address #1 to the end of v\_addresses three times using EXTEND(3, 1), and then removes two addresses from the end of v\_addresses using TRIM(2):

PROCEDURE trim\_addresses IS

v\_addresses t\_nested\_table\_address;

BEGIN

v\_addresses := get\_addresses(1);

display\_addresses(v\_addresses);

DBMS\_OUTPUT.PUT\_LINE('Extending addresses');

v\_addresses.EXTEND(3, 1);

display\_addresses(v\_addresses);

DBMS\_OUTPUT.PUT\_LINE('Trimming 2 addresses from end');

-- remove 2 addresses from the end of v\_addresses

-- using TRIM(2)

**v\_addresses.TRIM(2);**

display\_addresses(v\_addresses);

END trim\_addresses;

The following example calls trim\_addresses():

**CALL collection\_method\_examples.trim\_addresses();**

Number of addresses = 2

Current number of addresses = 2

Address #1:

2 State Street, Beantown, MA, 12345

Address #2:

4 Hill Street, Lost Town, CA, 54321

Extending addresses

Current number of addresses = 5

Address #1:

2 State Street, Beantown, MA, 12345

Address #2:

4 Hill Street, Lost Town, CA, 54321

Address #3:

2 State Street, Beantown, MA, 12345

Address #4:

2 State Street, Beantown, MA, 12345

Address #5:

2 State Street, Beantown, MA, 12345

Trimming 2 addresses from end

Current number of addresses = 3

Address #1:

2 State Street, Beantown, MA, 12345

Address #2:

4 Hill Street, Lost Town, CA, 54321

Address #3:

2 State Street, Beantown, MA, 12345

**MULTILEVEL COLLECTIONS**

With the release of Oracle Database 9*i*, you can create a collection in the database whose elements are also a collection. These "collections of collections" are known as *multilevel collections*. The following list shows the valid multilevel collections:

Image A nested table of nested tables

Image A nested table of varrays

Image A varray of varrays

Image A varray of nested tables

I’ve provided an SQL\*Plus script named collection\_schema2.sql in the SQL directory. This script creates a user named collection\_user2, with a password of collection\_password, along with the types and the table shown in this section. You can run this script if you are using Oracle Database 9*i* or higher. After the script completes, you will be logged in as collection\_user2.

Let’s say you wanted to store a set of phone numbers for each address of a customer. The following example creates a varray type of three VARCHAR2 strings named t\_varray\_phone to represent phone numbers:

CREATE TYPE t\_varray\_phone AS VARRAY(3) OF VARCHAR2(14);

/

Next, the following example creates an object type named t\_address that contains an attribute named phone\_numbers; this attribute is of type t\_varray\_phone:

CREATE TYPE t\_address AS OBJECT (

street VARCHAR2(15),

city VARCHAR2(15),

state CHAR(2),

zip VARCHAR2(5),

phone\_numbers t\_varray\_phone

);

/

The next example creates a nested table type of t\_address objects:

CREATE TYPE t\_nested\_table\_address AS TABLE OF t\_address;

/

The following example creates a table named customers\_with\_nested\_table, which contains a column named addresses of type t\_nested\_table\_address:

CREATE TABLE customers\_with\_nested\_table (

id INTEGER PRIMARY KEY,

first\_name VARCHAR2(10),

last\_name VARCHAR2(10),

addresses t\_nested\_table\_address

)

NESTED TABLE

addresses

STORE AS

nested\_addresses;

So, customers\_with\_nested\_table contains a nested table whose elements contain an address with a varray of phone numbers.

The following INSERT statement adds a row to customers\_with\_nested\_table; notice the structure and content of the INSERT statement, which contains elements for the nested table of addresses, each of which has an embedded varray of phone numbers:

INSERT INTO customers\_with\_nested\_table VALUES (

1, ’Steve', 'Brown',

t\_nested\_table\_address(

t\_address('2 State Street', 'Beantown', 'MA', '12345',

t\_varray\_phone(

'(800)-555-1211',

'(800)-555-1212',

'(800)-555-1213'

)

),

t\_address('4 Hill Street', 'Lost Town', 'CA', '54321',

t\_varray\_phone(

'(800)-555-1211',

'(800)-555-1212'

)

)

)

);

You can see that the first address has three phone numbers, while the second address has two. The following query retrieves the row from customers\_with\_nested\_table:

**SELECT \***

**FROM customers\_with\_nested\_table;**

ID FIRST\_NAME LAST\_NAME

---------- ---------- ----------

ADDRESSES(STREET, CITY, STATE, ZIP, PHONE\_NUMBERS)

------------------------------------------------------------------------

1 Steve Brown

T\_NESTED\_TABLE\_ADDRESS(

T\_ADDRESS('2 State Street', 'Beantown', 'MA', '12345',

T\_VARRAY\_PHONE('(800)-555-1211', '(800)-555-1212', '(800)-555-1213')),

T\_ADDRESS('4 Hill Street', 'Lost Town', 'CA', '54321',

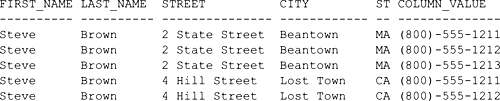
T\_VARRAY\_PHONE('(800)-555-1211', '(800)-555-1212')))

You can use TABLE() to treat the data stored in the collections as a series of rows, as shown in the following query:

**SELECT cn.first\_name, cn.last\_name, a.street, a.city, a.state, p.\***

**FROM customers\_with\_nested\_table cn**,

**TABLE(cn.addresses) a, TABLE(a.phone\_numbers) p;**



The following UPDATE statement shows how to update the phone numbers for the 2 State Street address; notice that TABLE() is used to get the addresses as a series of rows and that a varray containing the new phone numbers is supplied in the SET clause:

**UPDATE TABLE(**

**-- get the addresses for customer #1**

**SELECT cn.addresses**

**FROM customers\_with\_nested\_table cn**

**WHERE cn.id = 1**

**) addrs**

**SET addrs.phone\_numbers =**

**t\_varray\_phone(**

**'(800)-555-1214'**,

**'(800)-555-1215'**

**)**

**WHERE addrs.street = '2 State Street';**

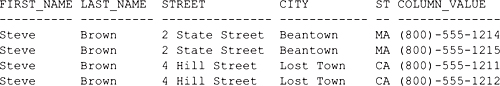
1 row updated.

The following query verifies the change:

**SELECT cn.first\_name, cn.last\_name, a.street, a.city, a.state, p.\***

**FROM customers\_with\_nested\_table cn**,

**TABLE(cn.addresses) a, TABLE(a.phone\_numbers) p;**



Support for multilevel collection types is a very powerful extension to the Oracle database software, and you might want to consider using them in any database designs you contribute to.

**ORACLE DATABASE 10*G* ENHANCEMENTS TO COLLECTIONS**

In this section, you’ll learn about the following enhancements made to collections in Oracle Database 10*g*:

Image Support for associative arrays

Image Ability to change the size or precision of an element type

Image Ability to increase the number of elements in a varray

Image Ability to use varray columns in temporary tables

Image Ability to use a different tablespace for a nested table’s storage table

Image ANSI support for nested tables

The various statements that create the items shown in this section are contained in the collection\_schema3.sql script. This script creates a user named collection\_user3 with a password of collection\_password and creates the collection types, tables, and PL/SQL code. You can run this script if you are using Oracle Database 10*g* or higher. After the script completes, you will be logged in as collection\_user3.

**Associative Arrays**

An associative array is a set of key and value pairs. You can get the value from the array using the key (which may be a string) or an integer that specifies the position of the value in the array. The following example procedure named customers\_associative\_array() illustrates the use of associative arrays:

CREATE PROCEDURE customers\_associative\_array AS

-- define an associative array type named t\_assoc\_array;

-- the value stored in each array element is a NUMBER,

-- and the index key to access each element is a VARCHAR2

TYPE t\_assoc\_array IS TABLE OF NUMBER INDEX BY VARCHAR2(15);

-- declare an object named v\_customer\_array of type t\_assoc\_array;

-- v\_customer\_array will be used to store the ages of customers

v\_customer\_array t\_assoc\_array;

BEGIN

-- assign the values to v\_customer\_array; the VARCHAR2 key is the

-- customer name and the NUMBER value is the age of the customer

v\_customer\_array('Jason') := 32;

v\_customer\_array(’Steve') := 28;

v\_customer\_array('Fred') := 43;

v\_customer\_array('Cynthia') := 27;

-- display the values stored in v\_customer\_array

DBMS\_OUTPUT.PUT\_LINE(

'v\_customer\_array[″Jason″] = ' || v\_customer\_array('Jason')

);

DBMS\_OUTPUT.PUT\_LINE(

'v\_customer\_array[″Steve″] = ' || v\_customer\_array(’Steve')

);

DBMS\_OUTPUT.PUT\_LINE(

'v\_customer\_array[″Fred″] = ' || v\_customer\_array('Fred')

);

DBMS\_OUTPUT.PUT\_LINE(

'v\_customer\_array[″Cynthia″] = ' || v\_customer\_array('Cynthia')

);

END customers\_associative\_array;

/

The following example sets the server output on and calls customers\_associative\_array():

**SET SERVEROUTPUT ON**

**CALL customers\_associative\_array();**

v\_customer\_array['Jason'] = 32

v\_customer\_array[’Steve'] = 28

v\_customer\_array['Fred'] = 43

v\_customer\_array['Cynthia'] = 27

**Changing the Size of an Element Type**

You can change the size of an element type in a collection when the element type is one of the character, numeric, or raw types (raw is used to store binary data—you’ll learn about this in the next chapter). Earlier in this chapter, you saw the following statement that creates a varray type named t\_varray\_address:

CREATE TYPE t\_varray\_address AS VARRAY(2) OF VARCHAR2(50);

/

The following example changes the size of the VARCHAR2 elements in t\_varray\_address to 60:

**ALTER TYPE t\_varray\_address**

**MODIFY ELEMENT TYPE VARCHAR2(60) CASCADE;**

Type altered.

The CASCADE option propagates the change to any dependent objects in the database, which, in the example, is the customers\_with\_varray table that contains a column named addresses of type t\_varray\_address. You can also use the INVALIDATE option to invalidate any dependent objects and immediately recompile the PL/SQL code for the type.

**Increasing the Number of Elements in a Varray**

You can increase the number of elements in a varray. The following example increases the number of elements in t\_varray\_address to 5:

**ALTER TYPE t\_varray\_address**

**MODIFY LIMIT 5 CASCADE;**

Type altered.

**Using Varrays in Temporary Tables**

You can use varrays in temporary tables, which are tables whose rows are temporary and are specific to a user session (temporary tables were covered in the section "Creating a Table" in [Chapter 10](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch10.html#ch10)). The following example creates a temporary table named cust\_with\_varray\_temp\_table that contains a varray named addresses of type t\_varray\_address:

CREATE GLOBAL TEMPORARY TABLE cust\_with\_varray\_temp\_table (

id INTEGER PRIMARY KEY,

first\_name VARCHAR2(10),

last\_name VARCHAR2(10),

addresses t\_varray\_address

);

**Using a Different Tablespace for a Nested Table’s Storage Table**

By default, a nested table’s storage table is created in the same tablespace as the parent table (a tablespace is an area used by the database to store objects such as tables—see the section "Creating a Table" in [Chapter 10](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch10.html#ch10) for details).

In Oracle Database 10*g* and higher, you can specify a different tablespace for a nested table’s storage table. The following example creates a table named cust\_with\_nested\_table that contains a nested table named addresses of type t\_nested\_table\_address; notice that the tablespace for the nested\_addresses2 storage table is the users tablespace:

CREATE TABLE cust\_with\_nested\_table (

id INTEGER PRIMARY KEY,

first\_name VARCHAR2(10),

last\_name VARCHAR2(10),

addresses t\_nested\_table\_address

)

NESTED TABLE

addresses

STORE AS

nested\_addresses2 TABLESPACE users;

You must have a tablespace named users in order for this example to work, and for this reason I’ve commented out the example in the collection\_schema3.sql script. You can see all the tablespaces you have access to by performing the following query:

**SELECT tablespace\_name**

**FROM user\_tablespaces;**

TABLESPACE\_NAME

---------------

SYSTEM

SYSAUX

UNDOTBS1

TEMP

USERS

EXAMPLE

If you want to run the previous CREATE TABLE statement, you can edit the example in the collection\_schema3.sql script to reference one of your tablespaces and then copy the statement into SQL\*Plus and run it.

**ANSI Support for Nested Tables**

The American National Standards Institute (ANSI) specification includes a number of operators that may be used with nested tables. You’ll learn about these operators in the following sections.

**Equal and Not-Equal Operators**

The equal (=) and not-equal (<>) operators compare two nested tables, which are considered equal when they satisfy all the following conditions:

Image The tables are the same type.

Image The tables are the same cardinality, that is, they contain the same number of elements.

Image All the elements of the table have the same value.

The following equal\_example() procedure illustrates the use of the equal and not-equal operators:

CREATE PROCEDURE equal\_example AS

-- declare a type named t\_nested\_table

TYPE t\_nested\_table IS TABLE OF VARCHAR2(10);

-- create t\_nested\_table objects named v\_customer\_nested\_table1,

-- v\_customer\_nested\_table2, and v\_customer\_nested\_table3;

-- these objects are used to store the names of customers

v\_customer\_nested\_table1 t\_nested\_table :=

t\_nested\_table('Fred', 'George', ’Susan');

v\_customer\_nested\_table2 t\_nested\_table :=

t\_nested\_table('Fred', 'George', ’Susan');

v\_customer\_nested\_table3 t\_nested\_table :=

t\_nested\_table('John', 'George', ’Susan');

v\_result BOOLEAN;

BEGIN

-- use = operator to compare v\_customer\_nested\_table1 with

-- v\_customer\_nested\_table2 (they contain the same names, so

-- v\_result is set to true)

v\_result := **v\_customer\_nested\_table1 = v\_customer\_nested\_table2**;

IF v\_result THEN

DBMS\_OUTPUT.PUT\_LINE(

'v\_customer\_nested\_table1 equal to v\_customer\_nested\_table2'

);

END IF;

-- use <> operator to compare v\_customer\_nested\_table1 with

-- v\_customer\_nested\_table3 (they are not equal because the first

-- names, 'Fred' and 'John', are different and v\_result is set

-- to true)

v\_result := **v\_customer\_nested\_table1 <> v\_customer\_nested\_table3**;

IF v\_result THEN

DBMS\_OUTPUT.PUT\_LINE(

'v\_customer\_nested\_table1 not equal to v\_customer\_nested\_table3'

);

END IF;

END equal\_example;

/

The following example calls equal\_example():

**CALL equal\_example();**

v\_customer\_nested\_table1 equal to v\_customer\_nested\_table2

v\_customer\_nested\_table1 not equal to v\_customer\_nested\_table3

**IN and NOT IN Operators**

The IN operator checks if the elements of one nested table appear in another nested table. Similarly, NOT INchecks if the elements of one nested table do not appear in another nested table. The following in\_example()procedure illustrates the use of IN and NOT IN:

CREATE PROCEDURE in\_example AS

TYPE t\_nested\_table IS TABLE OF VARCHAR2(10);

v\_customer\_nested\_table1 t\_nested\_table :=

t\_nested\_table('Fred', 'George', ’Susan');

v\_customer\_nested\_table2 t\_nested\_table :=

t\_nested\_table('John', 'George', ’Susan');

v\_customer\_nested\_table3 t\_nested\_table :=

t\_nested\_table('Fred', 'George', ’Susan');

v\_result BOOLEAN;

BEGIN

-- use IN operator to check if elements of v\_customer\_nested\_table3

-- are in v\_customer\_nested\_table1 (they are, so v\_result is

-- set to true)

v\_result := **v\_customer\_nested\_table3 IN**

**(v\_customer\_nested\_table1)**;

IF v\_result THEN

DBMS\_OUTPUT.PUT\_LINE(

'v\_customer\_nested\_table3 in v\_customer\_nested\_table1'

);

END IF;

-- use NOT IN operator to check if the elements of

-- v\_customer\_nested\_table3 are not in v\_customer\_nested\_table2

-- (they are not, so v\_result is set to true)

v\_result := **v\_customer\_nested\_table3 NOT IN**

**(v\_customer\_nested\_table2)**;

IF v\_result THEN

DBMS\_OUTPUT.PUT\_LINE(

'v\_customer\_nested\_table3 not in v\_customer\_nested\_table2'

);

END IF;

END in\_example;

/

The following example calls in\_example():

**CALL in\_example();**

v\_customer\_nested\_table3 in v\_customer\_nested\_table1

v\_customer\_nested\_table3 not in v\_customer\_nested\_table2

**SUBMULTISET Operator**

The SUBMULTISET operator checks whether the elements of one nested table are a subset of another nested table. The following submultiset\_example() procedure illustrates the use of SUBMULTISET:

CREATE PROCEDURE submultiset\_example AS

TYPE t\_nested\_table IS TABLE OF VARCHAR2(10);

v\_customer\_nested\_table1 t\_nested\_table :=

t\_nested\_table('Fred', 'George', ’Susan');

v\_customer\_nested\_table2 t\_nested\_table :=

t\_nested\_table('George', 'Fred', ’Susan', 'John', ’Steve');

v\_result BOOLEAN;

BEGIN

-- use SUBMULTISET operator to check if elements of

-- v\_customer\_nested\_table1 are a subset of v\_customer\_nested\_table2

-- (they are, so v\_result is set to true)

v\_result :=

**v\_customer\_nested\_table1 SUBMULTISET OF v\_customer\_nested\_table2**;

IF v\_result THEN

DBMS\_OUTPUT.PUT\_LINE(

'v\_customer\_nested\_table1 subset of v\_customer\_nested\_table2'

);

END IF;

END submultiset\_example;

/

The following example calls submultiset\_example():

**CALL submultiset\_example();**

customer\_nested\_table1 subset of customer\_nested\_table2

**MULTISET Operator**

The MULTISET operator returns a nested table whose elements are set to certain combinations of elements from two supplied nested tables. There are three MULTISET operators:

Image **MULTISET UNION** returns a nested table whose elements are set to the sum of the elements from two supplied nested tables.

Image **MULTISET INTERSECT** returns a nested table whose elements are set to the elements that are common to two supplied nested tables.

Image **MULTISET EXCEPT** returns a nested table whose elements are in the first supplied nested table but not in the second.

You may also use one of the following options with MULTISET:

Image **ALL** indicates that all the applicable elements are in the returned nested table. ALL is the default. For example, MULTISET UNION ALL returns a nested table whose elements are set to the sum of elements from two supplied nested tables, and all elements, including duplicates, are in the returned nested table.

Image **DISTINCT** indicates that only the non-duplicate (that is, distinct) elements are in the returned nested table. For example, MULTISET UNION DISTINCT returns a nested table whose elements are set to the sum of elements from two supplied nested tables, but duplicates are removed from the returned nested table.

The following multiset\_example() procedure illustrates the use of MULTISET:

CREATE PROCEDURE multiset\_example AS

TYPE t\_nested\_table IS TABLE OF VARCHAR2(10);

v\_customer\_nested\_table1 t\_nested\_table :=

t\_nested\_table('Fred', 'George', ’Susan');

v\_customer\_nested\_table2 t\_nested\_table :=

t\_nested\_table('George', ’Steve', 'Rob');

v\_customer\_nested\_table3 t\_nested\_table;

v\_count INTEGER;

BEGIN

-- use MULTISET UNION (returns a nested table whose elements

-- are set to the sum of the two supplied nested tables)

v\_customer\_nested\_table3 :=

**v\_customer\_nested\_table1 MULTISET UNION**

**v\_customer\_nested\_table2**;

DBMS\_OUTPUT.PUT('UNION: ');

FOR v\_count IN 1..v\_customer\_nested\_table3.COUNT LOOP

DBMS\_OUTPUT.PUT(v\_customer\_nested\_table3(v\_count) || ' ');

END LOOP;

DBMS\_OUTPUT.PUT\_LINE(' ');

-- use MULTISET UNION DISTINCT (DISTINCT indicates that only

-- the non-duplicate elements of the two supplied nested tables

-- are set in the returned nested table)

v\_customer\_nested\_table3 :=

**v\_customer\_nested\_table1 MULTISET UNION DISTINCT**

**v\_customer\_nested\_table2**;

DBMS\_OUTPUT.PUT('UNION DISTINCT: ');

FOR v\_count IN 1..v\_customer\_nested\_table3.COUNT LOOP

DBMS\_OUTPUT.PUT(v\_customer\_nested\_table3(v\_count) || ' ');

END LOOP;

DBMS\_OUTPUT.PUT\_LINE(' ');

-- use MULTISET INTERSECT (returns a nested table whose elements

-- are set to the elements that are common to the two supplied

-- nested tables)

v\_customer\_nested\_table3 :=

**v\_customer\_nested\_table1 MULTISET INTERSECT**

**v\_customer\_nested\_table2**;

DBMS\_OUTPUT.PUT('INTERSECT: ');

FOR v\_count IN 1..v\_customer\_nested\_table3.COUNT LOOP

DBMS\_OUTPUT.PUT(v\_customer\_nested\_table3(v\_count) || ' ');

END LOOP;

DBMS\_OUTPUT.PUT\_LINE(' ');

-- use MULTISET EXCEPT (returns a nested table whose

-- elements are in the first nested table but not in

-- the second)

v\_customer\_nested\_table3 :=

**v\_customer\_nested\_table1 MULTISET EXCEPT**

**v\_customer\_nested\_table2**;

DBMS\_OUTPUT.PUT\_LINE('EXCEPT: ');

FOR v\_count IN 1..v\_customer\_nested\_table3.COUNT LOOP

DBMS\_OUTPUT.PUT(v\_customer\_nested\_table3(v\_count) || ' ');

END LOOP;

END multiset\_example;

/

The following example calls multiset\_example():

**CALL multiset\_example();**

UNION: Fred George Susan George Steve Rob

UNION DISTINCT: Fred George Susan Steve Rob

INTERSECT: George

EXCEPT:

**CARDINALITY() Function**

The CARDINALITY() function returns the number of elements in a collection. The following cardinality\_example() procedure illustrates the use of CARDINALITY():

CREATE PROCEDURE cardinality\_example AS

TYPE t\_nested\_table IS TABLE OF VARCHAR2(10);

v\_customer\_nested\_table1 t\_nested\_table :=

t\_nested\_table('Fred', 'George', ’Susan');

v\_cardinality INTEGER;

BEGIN

-- call CARDINALITY() to get the number of elements in

-- v\_customer\_nested\_table1

v\_cardinality := **CARDINALITY(v\_customer\_nested\_table1)**;

DBMS\_OUTPUT.PUT\_LINE('v\_cardinality = ' || v\_cardinality);

END cardinality\_example;

/

The following example calls cardinality\_example():

**CALL cardinality\_example();**

v\_cardinality = 3

**MEMBER OF Operator**

The MEMBER OF operator checks whether an element is in a nested table. The following member\_of\_example() procedure illustrates the use of MEMBER OF:

CREATE PROCEDURE member\_of\_example AS

TYPE t\_nested\_table IS TABLE OF VARCHAR2(10);

v\_customer\_nested\_table1 t\_nested\_table :=

t\_nested\_table('Fred', 'George', ’Susan');

v\_result BOOLEAN;

BEGIN

-- use MEMBER OF to check if 'George' is in

-- v\_customer\_nested\_table1 (he is, so v\_result is set

-- to true)

v\_result := **'George' MEMBER OF v\_customer\_nested\_table1**;

IF v\_result THEN

DBMS\_OUTPUT.PUT\_LINE(″'George″ is a member');

END IF;

END member\_of\_example;

/

The following example calls member\_of\_example():

**CALL member\_of\_example();**

'George' is a member

**SET() Function**

The SET() function first converts a nested table into a set, then removes duplicate elements from the set, and finally returns the set as a nested table. The following set\_example() procedure illustrates the use of SET():

CREATE PROCEDURE set\_example AS

TYPE t\_nested\_table IS TABLE OF VARCHAR2(10);

v\_customer\_nested\_table1 t\_nested\_table :=

t\_nested\_table('Fred', 'George', ’Susan', 'George');

v\_customer\_nested\_table2 t\_nested\_table;

v\_count INTEGER;

BEGIN

-- call SET() to convert a nested table into a set,

-- remove duplicate elements from the set, and get the set

-- as a nested table

v\_customer\_nested\_table2 := **SET(v\_customer\_nested\_table1)**;

DBMS\_OUTPUT.PUT('v\_customer\_nested\_table2: ');

FOR v\_count IN 1..v\_customer\_nested\_table2.COUNT LOOP

DBMS\_OUTPUT.PUT(v\_customer\_nested\_table2(v\_count) || ' ');

END LOOP;

DBMS\_OUTPUT.PUT\_LINE(' ');

END set\_example;

/

The following example calls set\_example():

**CALL set\_example();**

v\_customer\_nested\_table2: Fred George Susan

**IS A SET Operator**

The IS A SET operator checks if the elements in a nested table are distinct. The following is\_a\_set\_example() procedure illustrates the use of IS A SET:

CREATE PROCEDURE is\_a\_set\_example AS

TYPE t\_nested\_table IS TABLE OF VARCHAR2(10);

v\_customer\_nested\_table1 t\_nested\_table :=

t\_nested\_table('Fred', 'George', ’Susan', 'George');

v\_result BOOLEAN;

BEGIN

-- use IS A SET operator to check if the elements in

-- v\_customer\_nested\_table1 are distinct (they are not, so

-- v\_result is set to false)

v\_result := **v\_customer\_nested\_table1 IS A SET**;

IF v\_result THEN

DBMS\_OUTPUT.PUT\_LINE('Elements are all unique');

ELSE

DBMS\_OUTPUT.PUT\_LINE('Elements contain duplicates');

END IF;

END is\_a\_set\_example;

/

The following example calls is\_a\_set\_example():

**CALL is\_a\_set\_example();**

Elements contain duplicates

**IS EMPTY Operator**

The IS EMPTY operator checks if a nested table doesn’t contain elements. The following is\_empty\_example() procedure illustrates the use of IS EMPTY:

CREATE PROCEDURE is\_empty\_example AS

TYPE t\_nested\_table IS TABLE OF VARCHAR2(10);

v\_customer\_nested\_table1 t\_nested\_table :=

t\_nested\_table('Fred', 'George', ’Susan');

v\_result BOOLEAN;

BEGIN

-- use IS EMPTY operator to check if

-- v\_customer\_nested\_table1 is empty (it is not, so

-- v\_result is set to false)

v\_result := **v\_customer\_nested\_table1 IS EMPTY**;

IF v\_result THEN

DBMS\_OUTPUT.PUT\_LINE('Nested table is empty');

ELSE

DBMS\_OUTPUT.PUT\_LINE('Nested table contains elements');

END IF;

END is\_empty\_example;

/

The following example calls is\_empty\_example():

**CALL is\_empty\_example();**

Nested table contains elements

**COLLECT() Function**

The COLLECT() function returns a nested table from a set of elements. The following query illustrates the use of COLLECT():

**SELECT COLLECT(first\_name)**

**FROM customers\_with\_varray;**

COLLECT(FIRST\_NAME)

----------------------------------------------

SYSTPfrFhAg+WRJGwW7ma9zy1KA == (’Steve', 'John')

You can use CAST() to convert the elements returned by COLLECT() to a specific type, as shown in the following query:

**SELECT CAST(COLLECT(first\_name) AS t\_table)**

**FROM customers\_with\_varray;**

CAST(COLLECT(FIRST\_NAME)AST\_TABLE)

----------------------------------

T\_TABLE(’Steve', 'John')

For your reference, the t\_table type used in the previous example is created by the following statement in the collection\_schema3.sql script:

CREATE TYPE t\_table AS TABLE OF VARCHAR2(10);

/

**POWERMULTISET() Function**

The POWERMULTISET() function returns all combinations of elements in a given nested table, as shown in the following query:

**SELECT \***

**FROM TABLE(**

**POWERMULTISET(t\_table('This', 'is', 'a', 'test'))**

**);**

COLUMN\_VALUE

----------------------------------

T\_TABLE('This')

T\_TABLE('is')

T\_TABLE('This', 'is')

T\_TABLE('a')

T\_TABLE('This', 'a')

T\_TABLE('is', 'a')

T\_TABLE('This', 'is', 'a')

T\_TABLE('test')

T\_TABLE('This', 'test')

T\_TABLE('is', 'test')

T\_TABLE('This', 'is', 'test')

T\_TABLE('a', 'test')

T\_TABLE('This', 'a', 'test')

T\_TABLE('is', 'a', 'test')

T\_TABLE('This', 'is', 'a', 'test')

**POWERMULTISET\_BY\_CARDINALITY() Function**

The POWERMULTISET\_BY\_CARDINALITY() function returns the combinations of elements in a given nested table that have a specified number of elements (or "cardinality"). The following query illustrates the use of POWERMULTISET\_BY\_CARDINALITY(), specifying a cardinality of 3:

**SELECT \***

**FROM TABLE(**

**POWERMULTISET\_BY\_CARDINALITY(**

**t\_table('This', 'is', 'a', 'test'), 3**

**)**

**);**

COLUMN\_VALUE

-----------------------------

T\_TABLE('This', 'is', 'a')

T\_TABLE('This', 'is', 'test')

T\_TABLE('This', 'a', 'test')

T\_TABLE('is', 'a', 'test')

**SUMMARY**

In this chapter, you have learned the following:

Image Collections allow you to store sets of elements.

Image There are three types of collections: varrays, nested tables, and associative arrays.

Image A varray is similar to an array in Java; you can use a varray to store an ordered set of elements with each element having an index associated with it. The elements in a varray are of the same type, and a varray has one dimension. A varray has a maximum size that you set when creating it, but you can change the size later.

Image A nested table is a table that is embedded within another table, and you can insert, update, and delete individual elements in a nested table. Because you can modify individual elements in a nested table, they are more flexible than a varray—a varray can be modified only as a whole. A nested table doesn’t have a maximum size, and you can store an arbitrary number of elements in a nested table.

Image An associative array is a set of key and value pairs. You can get the value from the array using the key (which may be a string) or an integer that specifies the position of the value in the array. An associative array is similar to a hash table in programming languages such as Java.

Image A collection may itself contain embedded collections. Such a collection is known as a multilevel collection.