CHAPTER 26

Object-Oriented Aspects of PL/SQL

PL/SQL has always been a language that supports traditional procedural programming styles such as structured design and functional decomposition. Using PL/SQL packages, you can also take an object-based approach, applying principles such as abstraction and encapsulation to the business of manipulating relational tables. Recent versions of the Oracle database have introduced direct support for *object-oriented programming* (OOP), providing a rich and complex type system, complete with support for type hierarchies and "substitutability."

In the interest of summarizing this book-length topic into a modest number of pages, this chapter presents a few choice code samples to demonstrate the most significant aspects of object programming with PL/SQL. These cover the following areas:

- Creating and using object types
- Using inheritance and substitutability
- Type evolution
- Pointer (REF)-based retrieval
- Object views, including INSTEAD OF views

Among the things you will *not* find in this chapter are:

- Comprehensive syntax diagrams for SQL statements dealing with object types
- Database administration topics such as importing and exporting object data
- Low-level considerations such as physical data storage on disk

I'd like to introduce the topic with a brief history.

Introduction to Oracle's Object Features

First released in 1997 as an add-on to the Oracle8 Database (the so-called "object-relational database"), the Objects Option allowed developers to extend Oracle's built-in datatypes to include *abstract datatypes*. The introduction of programmer-defined *collections* (described in Chapter 12) in that release also proved useful, not only because application developers had been looking for ways to store and retrieve arrays in the database, but also because PL/SQL provided a new way to query collections as if they were tables. While there were other interesting aspects of the new Oracle object model, such as pointer-based navigation, there was no notion of inheritance or dynamic polymorphism, making the object-relational features of the Oracle8 Database an option that drew few converts from (or into) the camp of true OOP believers. The complexity of the object features, plus a perceived performance hit, also limited uptake in the relational camp.

The Oracle8*i* Database introduced support for Java Stored Procedures, which not only provided the ability to program the server using a less proprietary language than PL/SQL, but also made it easier for the OOP community to consider using stored procedures. Oracle provided a way to translate object type definitions from the server into Java classes, making it possible to share objects across the Java/database boundary. Oracle released the Oracle8*i* Database during a peak of market interest in Java, so hardly anyone really noticed that the database's core object features were not much enhanced, except that Oracle Corporation quietly began bundling the object features with the core database server. Around this time, I asked an Oracle representative about the future of object programming in PL/SQL, and the response was, "If you want real object-oriented programming in the database, use Java."

Nevertheless, with the Oracle9*i* Database release, Oracle significantly extended the depth of its native object support, becoming a more serious consideration for OOP purists. Inheritance and polymorphism became available in the database, and PL/SQL gained new object features. Does it finally make sense to extend the object model of your system into the structure of the database itself? Should you now repartition existing middleware or client applications to take advantage of "free stuff" in the database server? As Table 26-1 shows, Oracle has made great strides, and the move may be tempting. The table also shows that a few desirable features still aren't available.[1]

> Oracle Database 10*g*, although introducing several useful enhancements to collections (see Chapter 12), included only one new feature unique to object types: it is described in the sidebar "The OBJECT_VALUE Pseudocolumn" on page 1180.

1. Perhaps I should say *arguably* desirable features. The missing features are unlikely to be showstoppers.

Table 26-1. Significant object programming features in the Oracle database

Feature	8.0	8.1	9.1	9.2 and later	11*g* and later
Abstract datatypes as first-class database entity	✓	✓	✓	✓	✓
Abstract datatypes as PL/SQL parameter	✓	✓	✓	✓	✓
Collection-typed attributes	✓	✓	✓	✓	✓
REF-typed attributes for intra-database object navigation	✓	✓	✓	✓	✓
Implementing method logic in PL/SQL or C	✓	✓	✓	✓	✓
Programmer-defined object comparison semantics	✓	✓	✓	✓	✓
Views of relational data as object-typed data	✓	✓	✓	✓	✓
Compile-time or static polymorphism (method overloading)	✓	✓	✓	✓	✓
Ability to "evolve" type by modifying existing method logic (but not signature), or by adding methods	✓	✓	✓	✓	✓
Implementing method logic in Java		✓	✓	✓	✓
"Static" methods (execute without having object instance)		✓	✓	✓	✓
Relational primary key can serve as persistent object identifier, allowing declarative integrity of REFs		✓	✓	✓	✓
Inheritance of attributes and methods from a user-defined type			✓	✓	✓
Dynamic method dispatch			✓	✓	✓
Noninstantiable supertypes, similar to Java-style "abstract classes"			✓	✓	✓
Ability to evolve type by removing methods (and adding to change signature)			✓	✓	✓
Ability to evolve type by adding and removing attributes, automatically propagating changes to associated physical database structures			✓	✓	✓
"Anonymous" types: ANYTYPE, ANYDATA, ANYDATASET			✓	✓	✓
Downcast operator (TREAT) and type detection operator (IS OF) available in SQL			✓	✓	✓
TREAT and IS OF available in PL/SQL				✓	✓
User-defined constructor functions				✓	✓
Supertype method invocation in a subtype					✓
"Private" attributes, variables, constants, and methods					
Inheritance from multiple supertypes					
Sharing of object types or instances across distributed databases without resorting to object views					

Unless you're already a practicing object-oriented programmer, many of the terms in this table probably don't mean much to you. However, the remainder of this chapter should shed some light on these terms and give some clues about the larger architectural decisions you may need to make.

Object Types by Example

In keeping with the sample general application area explored in the introductory book *Learning Oracle PL/SQL*, I'd like to build an Oracle system that will use an object-oriented approach to modeling a trivial library catalog. The catalog can hold books, serials (such as magazines, proceedings, or newspapers), and, eventually, other artifacts.

A graphic portrayal of the top-level types appears in Figure 26-1. Later on, I might want to add to the type hierarchy, as the dotted-line boxes imply.

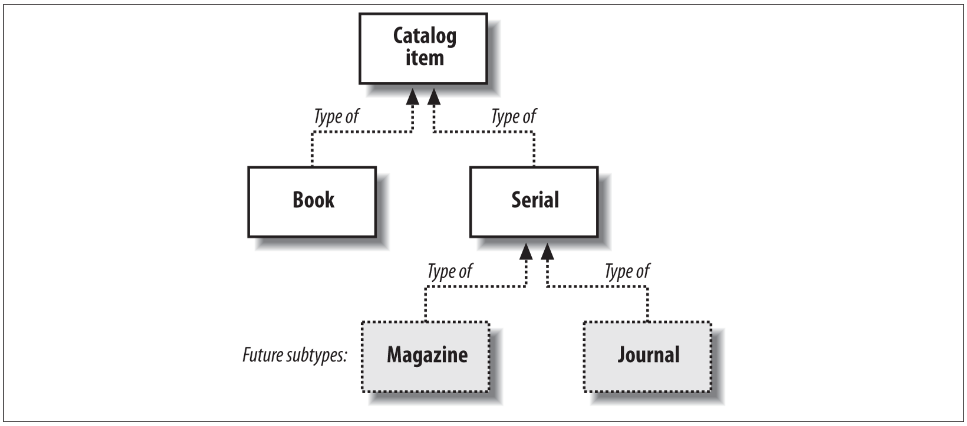

Figure 26-1. Type hierarchy for a trivial library catalog

Creating a Base Type

The "root," or top, of the hierarchy represents the common characteristics of all the subtypes. For now, let's assume that the only things that books and serials have in common are a library-assigned identification number and some kind of filing title. I can create an object type for catalog items using the following SQL statement from SQL*Plus:

```
CREATE OR REPLACE TYPE catalog_item_t AS OBJECT (
   id INTEGER,
   title VARCHAR2(4000),
   NOT INSTANTIABLE MEMBER FUNCTION ck_digit_okay
      RETURN BOOLEAN,
   MEMBER FUNCTION print
      RETURN VARCHAR2
) NOT INSTANTIABLE NOT FINAL;
```

This statement creates an object type, which is similar to a Java or C++ class. In relational terms, an object type is akin to a record type bundled with related functions and procedures. These subprograms are known collectively as *methods*.

The NOT FINAL keyword at the end flags the datatype as being able to serve as the *base type* or supertype from which you can derive other types. I needed to include NOT FINAL because I want to create subtypes for books and serials; if this keyword is omitted, the Oracle database defaults to FINAL—that is, no subtypes allowed.

Notice also that I've marked this type specification NOT INSTANTIABLE. Although PL/SQL will let me declare a variable of type catalog_item_t, I won't be able to give it a value—not directly, anyway. Similar to a Java *abstract class*, this kind of type exists only to serve as a base type from which to create subtypes, and objects of the subtype will, presumably, be instantiable.

For demonstration and debugging purposes, I've included a print method ("print" is *not* a reserved word, by the way) as a way to describe the object in a single string. When I create a subtype, it can (and probably should) override this method—in other words, the subtype will include a method with the same name, but will also print the subtype's attributes. Notice that instead of making print a procedure, which would have hardcoded a decision to use something like DBMS_OUTPUT.PUT_LINE, I decided to make it a function whose output can be redirected later. This decision isn't particularly object oriented, just good design.

I've also defined a ck_digit_okay method that will return TRUE or FALSE depending on whether the "check digit" is OK. The assumption here (which is a bad one, I admit) is that all subtypes of catalog_item_t will be known by some identifier other than their library-assigned ID, and these other identifiers include some concept of a check digit.[2] I'm only going to be dealing with books and serials, normally identified with an ISBN or ISSN, so the check digit concept applies to all the subtypes.

Here are a few further comments before we move on to the next part of the example:

- The preceding CREATE TYPE statement creates only an object type specification. The corresponding body, which implements the methods, will be created separately via CREATE TYPE BODY.
- Object types live in the same namespace as tables and top-level PL/SQL programs. This is one of the reasons I use the "_t" naming convention with types.
- Object types are owned by the Oracle user (schema) that created them, and this user may grant the EXECUTE privilege to other users.
- You can attempt to create synonyms on object types, but unless you're using Oracle9*i* Database Release 2 or later, the synonyms won't work.

2. A *check digit* is a number incorporated into an identifier that is mathematically derived from the identifier's other digits. Its accuracy yields a small amount of confidence that the overall identifier has been correctly transcribed. The ISBN (International Standard Book Number) and ISSN (International Standard Serial Number)—identifiers assigned by external authorities—both contain check digits. So do most credit card numbers.

- As with conventional PL/SQL programs, you can create an object type using either definer rights (the default) or invoker rights (described in Chapter 24).
- Unlike some languages' object models, Oracle's model doesn't define a master root-level class from which all programmer-defined classes derive. Instead, you can create any number of standalone root-level datatypes such as catalog_item_t.
- If you see the compiler error *PLS-00103: Encountered the symbol ";" when expecting one of the following...*, you have probably made the common mistake of terminating the methods with a semicolon. The correct token in the type specification is a comma.

Creating a Subtype

I made catalog_item_t impossible to instantiate, so now would be a good time to show how to create a subtype for book objects. In the real world, a book is a type of catalog item. This is also true in my example, in which all instances of this book_t subtype will have four attributes:

id
: Inherited from the base catalog_item_t type

title
: Also inherited from the base type

isbn
: Corresponds to the book's assigned ISBN, if any

pages
: An integer giving the number of pages in the book

In code, I can make the equivalent statement as follows:

```
TYPE book_t UNDER catalog_item_t (
  isbn VARCHAR2(13),
  pages INTEGER,

  CONSTRUCTOR FUNCTION book_t (id IN INTEGER DEFAULT NULL,
     title IN VARCHAR2 DEFAULT NULL,
     isbn IN VARCHAR2 DEFAULT NULL,
     pages IN INTEGER DEFAULT NULL)
     RETURN SELF AS RESULT,

   OVERRIDING MEMBER FUNCTION ck_digit_okay
      RETURN BOOLEAN,

   OVERRIDING MEMBER FUNCTION print
      RETURN VARCHAR2
);
```

The interesting portions of this code are described in the following table.

Line(s)	Description
1	You can see that the syntax for indicating a subtype is the keyword UNDER in line 1, which makes a certain amount of intuitive sense. Oracle doesn't use the phrase AS OBJECT here because it would be redundant; the only thing that can exist "under" an object type is another object type.
2–3	I need to list only those attributes that are unique to the subtype; those in the parent type are implicitly included. Oracle orders the attributes with the base type first, then the subtype, in the same order as defined in the specification.
5–15	Here are the method declarations. I'll look at these methods more closely in the next section.

Methods

I've used two kinds of methods in the previous type definition:

Constructor method
: A function that accepts values for each attribute and assembles them into a typed object. Declared in lines 5–9 of the example.

Member method
: A function or procedure that executes in the context of an object instance—that is, it has access to the current values of each of the attributes. Declared in lines 11–12 and 14–15 of the example.

My example shows a user-defined constructor, a feature that was introduced in Oracle9*i* Database Release 2. Earlier versions provided only a system-defined constructor. Creating your own constructor for each type gives you precise control over what happens at instantiation. That control can be very useful for doing extra tasks like validation and introducing controlled side effects. In addition, you can use several overloaded versions of a user-defined constructor, allowing it to adapt to a variety of calling circumstances.

To see some types and methods in action, take a look at this anonymous block:

```
 1   DECLARE
 2      generic_item catalog_item_t;
 3      abook book_t;
 4   BEGIN
 5      abook := NEW book_t(title => 'Out of the Silent Planet',
 6         isbn => '0-6848-238-02');
 7      generic_item := abook;
 8      DBMS_OUTPUT.PUT_LINE('BOOK: ' || abook.print());
 9      DBMS_OUTPUT.PUT_LINE('ITEM: ' || generic_item.print());
10    END;
```

Interestingly, the objects' print invocations (lines 8 and 9) yield identical results for both a book and generic_item:

```
BOOK: id=; title=Out of the Silent Planet; isbn=0-6848-238-02; pages=
ITEM: id=; title=Out of the Silent Planet; isbn=0-6848-238-02; pages=
```

The following table walks you through the code.

Line(s)	Description
5–6	The constructor assembles a new object and puts it into a book. My example takes advantage of PL/SQL's named notation. It supplies values for only two of the four attributes, but the constructor creates the object anyway, which is what I asked it to do. The syntax to use any constructor follows this pattern: `[ NEW ] typename ( arg1, arg2, ... );` The NEW keyword, introduced in Oracle9*i* Database Release 2, is optional, but is nevertheless useful as a visual cue that the statement will create a new object.
7	Even though a catalog item is not instantiable, I can assign to it an instance of a subtype, and it will even hold all the attributes that are unique to the subtype. This demonstrates one nifty aspect of "substitutability" that Oracle supports in PL/SQL, which is that, by default, an object variable may hold an instance of any of its subtypes. Note to programmers of other languages: the assignment in line 7 is not simply creating a second reference to one object; instead, it's making a complete copy. In English, it certainly makes sense to regard a book as a catalog item. In computerese, it's a case of *widening* or *upcasting* the generic item by adding attributes from a more specific subtype. The converse operation, *narrowing*, is trickier but nevertheless possible, as you'll see later.
8–9	Notice that the calls to print use the graceful object-style invocation: `object.methodname(arg1, arg2, ...)` because it is a member method executing on an already declared and instantiated object. Which version of the print method executes for objects of different types? The one in the *most specific subtype* associated with the currently instantiated object. The selection of the method gets deferred until runtime, in a feature known as *dynamic method dispatch*. This can be very handy, although it may incur a performance cost.

Let's turn now to the body of the book_t method, so you can better understand the result you've just seen. The implementation holds two important new concepts, which I'll describe afterward:

```
 1   TYPE BODY book_t
 2   AS
 3      CONSTRUCTOR FUNCTION book_t (id IN INTEGER,
 4         title IN VARCHAR2,
 5         isbn IN VARCHAR2,
 6         pages IN INTEGER)
 7         RETURN SELF AS RESULT
 8      IS
 9      BEGIN
10          SELF.id := id;
11          SELF.title := title;
12          SELF.isbn := isbn;
13          SELF.pages := pages;
14          IF isbn IS NULL OR SELF.ck_digit_okay
15          THEN
16             RETURN;
17          ELSE
18             RAISE_APPLICATION_ERROR(-20000, 'ISBN ' || isbn
19                || ' has bad check digit');
```

```
20          END IF;
21       END;
22
23       OVERRIDING MEMBER FUNCTION ck_digit_okay
24          RETURN BOOLEAN
25       IS
26          subtotal PLS_INTEGER := 0;
27          isbn_digits VARCHAR2(10);
28       BEGIN
29          /* remove dashes and spaces */
30          isbn_digits := REPLACE(REPLACE(SELF.isbn, '-'), ' ');
31          IF LENGTH(isbn_digits) != 10
32          THEN
33             RETURN FALSE;
34          END IF;
35
36          FOR nth_digit IN 1..9
37          LOOP
38             subtotal := subtotal +
39                (11 - nth_digit) *
40                  TO_NUMBER(SUBSTR(isbn_digits, nth_digit, 1));
41          END LOOP;
42
43          /* check digit can be 'X', which has value of 10 */
44          IF UPPER(SUBSTR(isbn_digits, 10, 1)) = 'X'
45          THEN
46             subtotal := subtotal + 10;
47          ELSE
48             subtotal := subtotal + TO_NUMBER(SUBSTR(isbn_digits, 10, 1));
49          END IF;
50
51          RETURN MOD(subtotal, 11) = 0;
52
53       EXCEPTION
54          WHEN OTHERS
55          THEN
56             RETURN FALSE;
57       END;
58
59       OVERRIDING MEMBER FUNCTION print
60          RETURN VARCHAR2
61       IS
62       BEGIN
63         RETURN 'id=' || id || '; title=' || title
64             || '; isbn=' || isbn || '; pages=' || pages;
65       END;
66    END;
```

Note the following about lines 3–21:

- A user-defined constructor has several rules to follow:
  — It must be declared with the keywords CONSTRUCTOR FUNCTION (line 3).

  - The return clause must be RETURN SELF AS RESULT (line 7).
  - It assigns values to any of the current object's attributes (lines 10–13).
  - It ends with a bare RETURN statement or an exception (line 16; lines 18–19).

- A constructor will typically assign values to as many of the attributes as it knows about. As you can see from line 14, my constructor tests the check digit before completing the construction. You will notice, if you skip ahead to line 30, that object attributes (such as SELF.isbn) are accessible even before validation is complete, an interesting and useful feature.
- Lines 18–19 are merely a placeholder; you should definitely take a more comprehensive approach to application-specific exceptions, as discussed in "Use Standardized Error Management Programs" on page 163.

Next, let's look at the use of the SELF keyword that appears throughout the type body. SELF is akin to Java's this keyword. Translation for non-Java programmers: SELF is merely a way to refer to the invoking (current) object when writing implementations of member methods. You can use SELF by itself when referring to the entire object, or you can use dot notation to refer to an attribute or a method:

```
IF SELF.id ...

IF SELF.ck_digit_okay() ...
```

The SELF keyword is not always required inside a member method, as you can see in lines 63–64, because the current object's attribute identifiers are always in scope. Using SELF can provide attribute visibility (as in lines 10–13, where the PL/SQL compiler interprets those unqualified identifiers as the formal parameters) and help to make your code SELF-documenting. (Ugh, sorry about that.)

There are a few more rules to note about this keyword:

- SELF isn't available inside static method bodies because static methods have no "current object." (I'll define static methods later in this section.)
- By default, SELF is an IN variable in functions and an IN OUT variable in procedures and constructor functions.
- You can change the default mode by including SELF as the first formal parameter.

Lines 23–57 of the previous example show the computing of the check digit, which is kind of fun, but my algorithm doesn't really exploit any new object-oriented features. I will digress to mention that the exception handler is quite important here; it responds to a multitude of problems such as the TO_NUMBER function encountering a character instead of a digit.

Next, on to creating a subtype for serials:

```
TYPE serial_t UNDER catalog_item_t (
   issn VARCHAR2(10),
   open_or_closed VARCHAR2(1),

   CONSTRUCTOR FUNCTION serial_t (id IN INTEGER DEFAULT NULL,
      title IN VARCHAR2 DEFAULT NULL,
      issn IN VARCHAR2 DEFAULT NULL,
      open_or_closed IN VARCHAR2 DEFAULT NULL)
      RETURN SELF AS RESULT,

   OVERRIDING MEMBER FUNCTION ck_digit_okay
      RETURN BOOLEAN,

   OVERRIDING MEMBER FUNCTION print
      RETURN VARCHAR2
) NOT FINAL;
```

Again, no new features appear in this type specification, but it does give another example of subtyping. A serial item in this model will have its own constructor, its own version of validating the check digit, and its own way to print itself.[3]

In addition to constructor and member methods, Oracle supports two other categories of methods:

Static methods
: Functions or procedures invoked independently of any instantiated objects. Static methods behave a lot like conventional PL/SQL procedures or functions.

Comparison methods (MAP and ORDER)
: Special member methods that let you program what Oracle should do when it needs to compare two objects of a particular datatype—for example, in an equality test in PL/SQL or when sorting objects in SQL.

One final point before moving on: objects follow PL/SQL's general convention that uninitialized variables are null;[4] the precise term for objects is *atomically null* (see Chapter 13 for more information).

As with collections, when an object is null, you cannot simply assign values to its attributes. Take a look at this short example:

```
DECLARE
   mybook book_t;      -- declared, but not initialized
BEGIN
   IF mybook IS NULL   -- this will be TRUE; it is atomically null
   THEN
```

3. In case you're curious, the open_or_closed attribute will be either (O)pen, meaning that the library can continue to modify the catalog entry (perhaps it does not own all the issues); (C)losed, meaning that the catalog entry is complete; or NULL, meaning we just don't know at the moment.

4. Associative arrays are a significant exception; they are non-null but empty when first declared.

```
      mybook.title := 'Learning Oracle PL/SQL'; -- this line raises...
   END IF;
EXCEPTION
   WHEN ACCESS_INTO_NULL    -- ...this predefined exception
   THEN
      ...
END;
```

Before assigning values to the attributes, you *must* initialize (instantiate) the entire object in one of three ways: by using a constructor method, via direct assignment from another object, or via a fetch from the database, as described in "Storing, Retrieving, and Using Persistent Objects" on page 1154.

Invoking Supertype Methods in Oracle Database 11g and Later

One restriction in Oracle's object-oriented functionality that was lifted in Oracle Database 11*g* was the inability to invoke a method of a supertype that is overridden in the current (or higher-level) subtype.

Prior to Oracle Database 11*g*, if I overrode a supertype's method in a subtype, there was no way that I could call the supertype's method in an instance of the subtype. This is now possible, as I demonstrate next.

Suppose I create a root type to manage and display information about food (my favorite topic!):

```
/* File on web: 11g_gen_invoc.sql */
CREATE TYPE food_t AS OBJECT (
   NAME         VARCHAR2 (100),
   food_group   VARCHAR2 (100),
   grown_in     VARCHAR2 (100),
   MEMBER FUNCTION to_string RETURN VARCHAR2
)
NOT FINAL;
/
CREATE OR REPLACE TYPE BODY food_t
IS
   MEMBER FUNCTION to_string RETURN VARCHAR2
   IS
   BEGIN
      RETURN 'FOOD! ' || self.name || ' - '
             || self.food_group || ' - ' || self.grown_in;
   END;
END;
/
```

I then create a subtype of food, dessert, that overrides the to_string method. Now, when I display information about a dessert I would like to include both dessert-specific information and the more general food attributes, but I don't want to copy and paste the code from the food type. I want to reuse it. Prior to Oracle Database 11*g*, this was not possible. With the new *general invocation* feature (SELF AS *supertype*), however, I can define the type as follows:

```
CREATE TYPE dessert_t UNDER food_t (
    contains_chocolate CHAR (1)
  , year_created NUMBER (4)
  , OVERRIDING MEMBER FUNCTION to_string RETURN VARCHAR2
);
/

CREATE OR REPLACE TYPE BODY dessert_t
IS
   OVERRIDING MEMBER FUNCTION to_string  RETURN VARCHAR2
   IS
   BEGIN
      /* Add the supertype (food) string to the subtype string */
      RETURN    'DESSERT! With Chocolate? '
             || contains_chocolate
             || ' created in '
             || SELF.year_created
             || chr(10)
             || (SELF as food_t).to_string;
   END;
END;
/
```

Now, when I display the "to string" representation of a dessert, I see the food information as well:

```
DECLARE
   TYPE foodstuffs_nt IS TABLE OF food_t;

   fridge_contents foodstuffs_nt
         := foodstuffs_nt (
               food_t ('Eggs benedict', 'PROTEIN', 'Farm')
             , dessert_t ('Strawberries and cream'
                        , 'FRUIT', 'Backyard', 'N', 2001)
            );
BEGIN
   FOR indx in 1 .. fridge_contents.COUNT
   LOOP
      DBMS_OUTPUT.put_line (RPAD ('=', 60, '='));
      DBMS_OUTPUT.put_line (fridge_contents (indx).to_string);
   END LOOP;
END;
/
```

The output is:

```
============================================================
FOOD! Eggs benedict - PROTEIN - Farm
============================================================
DESSERT! With Chocolate? N created in 2001
FOOD! Strawberries and cream - FRUIT - Backyard
```

In Oracle's implementation of supertype invocation, you don't simply refer to the supertype with a generic SUPERTYPE keyword, as is done in some other object-oriented languages. Instead, you must specify the supertype from the hierarchy. This is more flexible (you can invoke whichever supertype method you like), but it also means that you must hardcode the name of the supertype in your subtype's code.

Storing, Retrieving, and Using Persistent Objects

Thus far, I've been discussing the definition of datatypes and the instantiation of objects only in the memory of running programs. Fortunately, that's not even half the story! Oracle wouldn't be Oracle if there were no way to store an object in the database.

There are at least two main ways that I could physically store the library catalog as modeled thus far: either as one big table of catalog objects or as a series of smaller tables, one for each subtype. I'll show the former arrangement, which could begin as follows:

```
CREATE TABLE catalog_items OF catalog_item_t
   (CONSTRAINT catalog_items_pk PRIMARY KEY (id));
```

This statement tells Oracle to build an *object table* called catalog_items, each row of which will be a *row object* of type catalog_item_t. An object table generally has one column per attribute:

```
SQL > DESC catalog_items
 Name                                Null?    Type
 ----------------------------------- -------- ------------------------
 ID                                  NOT NULL NUMBER(38)
 TITLE                                        VARCHAR2(4000)
```

Remember, though, that catalog_item_t isn't instantiable, and each row in the table will actually be of a subtype such as book or serial item. So where do the extra attributes go? Consider that these are legal statements:[5]

```
INSERT INTO catalog_items
   VALUES (NEW book_t(10003, 'Perelandra', '0-684-82382-9', 222));
INSERT INTO catalog_items
   VALUES (NEW serial_t(10004, 'Time', '0040-781X', 'O'));.
```

In fact, Oracle put the ISBN, ISSN, and so on into hidden columns on the catalog_items table. From an object programming point of view, that's pretty cool because it helps preserve the abstraction of the catalog item, yet provides a way to expose the additional subtype information when needed.

5. I would prefer to use named notation in these static function calls, but that was not supported until Oracle Database 11*g*, which supports named notation for any user-defined PL/SQL function called within a SQL statement.

One more thing about the catalog_items table: the aforementioned CONSTRAINT clause designates the id column as the primary key. Yes, object tables can have primary keys too. And, if you exclude such a CONSTRAINT clause, Oracle will instead create a system-generated object identifier (OID), as described next.

Method Chaining

An object whose type definition looks like this:

```
CREATE OR REPLACE TYPE chaindemo_t AS OBJECT (
    x NUMBER, y VARCHAR2(10), z DATE,
    MEMBER FUNCTION setx (x IN NUMBER) RETURN chaindemo_t,
    MEMBER FUNCTION sety (y IN VARCHAR2) RETURN chaindemo_t,
    MEMBER FUNCTION setz (z IN DATE) RETURN chaindemo_t);
```

provides the ability to "chain" its methods together. For example:

```
DECLARE
    c chaindemo_t := chaindemo_t(NULL, NULL, NULL);
BEGIN
    c := c.setx(1).sety('foo').setz(sysdate);  -- chained invocation
```

The preceding executable statement really just acts as the equivalent of:

```
c := c.setx(1);
c := c.sety('foo');
c := c.setz(sysdate);
```

Each function returns a typed object as the input to the next function in the chain. The implementation of one of the methods appears in the following code (the others are similar):

```
MEMBER FUNCTION setx (x IN NUMBER) RETURN chaindemo_t IS
   l_self chaindemo_t := SELF;
BEGIN
   l_self.x := x;
   RETURN l_self;
END;
```

Here are some rules about chaining:

- You cannot use a function's return value as an IN OUT parameter to the next function in the chain. Functions return read-only values.
- Methods are invoked in order from left to right.
- The return value of a chained method must be of the object type expected by the method to its right.
- A chained call can include at most a single procedure.

If your chained call includes a procedure, it must be the rightmost method in the chain.

Object identity

If you're a relational database programmer, you know that conventional tables have a unique identifier for every row. If you're an object-oriented programmer, you know that OOP environments generally assign unique arbitrary identifiers that serve as object handles. If you're a programmer using object-relational features of the database, you can use a mix of both approaches. The following table summarizes where you will find object identifiers.

What and where	Has object identifier?
Row object in object table	Yes
Column object in any table (or fetched into PL/SQL program)	No; use row's primary key instead
Transient object created in PL/SQL program	No; use entire object instead
Row object fetched from object table into PL/SQL program	Yes, but available in program only if you explicitly fetch the "REF" (see the section "Using REFs" on page 1165)

Here is an example of a table that can hold column objects:

```
CREATE TABLE my_writing_projects (
   project_id INTEGER NOT NULL PRIMARY KEY,
   start_date DATE,
   working_title VARCHAR2(4000),
   catalog_item catalog_item_t  -- this is a "column object"
);
```

Oracle Corporation takes the view that a column object is dependent on the row's primary key, and should not be independently identified.[6]

For any object table, the Oracle database can base its object identifier on one of two things:

The primary key value
: To use this feature, use the clause OBJECT IDENTIFIER IS PRIMARY KEY at the end of the CREATE TABLE statement.

A system-generated value
: If you omit the PRIMARY KEY clause, Oracle adds a hidden column named SYS_NC_OID$ to the table and populates it with a unique 16-byte RAW value for each row.

Which kind of OID should you use? Primary-key-based OIDs typically use less storage than system-generated OIDs, provide a means of enforcing referential integrity, and allow for much more convenient loading of objects. System-generated OIDs have the

6. A contrary view is held by relational database experts, who assert that OIDs should not be used for row identification and that *only* column objects should have OIDs. See Hugh Darwen and C. J. Date, "The Third Manifesto," *SIGMOD Record* 24, no. 1 (March 1995).

advantage that REFs to them cause "scoped" or limited values from only one table. For a more complete discussion of the pros and cons of these two approaches, check out Oracle's *Application Developer's Guide—Object-Relational Features*. For now, you should know that a system-generated OID is:

Opaque
: Although your programs can use the OID indirectly, you don't typically see its value.

Potentially globally unique across databases
: The OID space makes provisions for up to 2^{128} objects (definitely "many" by the reckoning of the Hottentots).[7] In theory, these OIDs could allow object navigation across distributed databases without embedding explicit database links.

Immutable
: Immutable in this context means incapable of update. Even after export and import, the OID remains the same, unlike a ROWID. To "change" an OID, you would have to delete and recreate the object.

The VALUE function

To retrieve an object from the database, Oracle provides the VALUE function in SQL. VALUE accepts a single argument, which must be a table alias in the current FROM clause, and returns an object of the type on which the table is defined. It looks like this in a SELECT statement:

```
SELECT VALUE(c)
  FROM catalog_items c;
```

I like short abbreviations as table aliases, which explains the *c*. The VALUE function returns an opaque series of bits to the calling program rather than a record of column values. SQL*Plus, however, has built-in features to interpret these bits, returning the following result from that query:

```
VALUE(C)(ID, TITLE)
-------------------------------------------------
BOOK_T(10003, 'Perelandra', '0-684-82382-9', 222)
SERIAL_T(10004, 'Time', '0040-781X', 'O')
```

PL/SQL also has features to deal with fetching objects. Start with a properly typed local variable named catalog_item:

```
DECLARE
   catalog_item catalog_item_t;
   CURSOR ccur IS
      SELECT VALUE(c)
        FROM catalog_items c;
BEGIN
```

7. The Hottentots had a four-valued counting system: 1, 2, 3, and "many."

```
   OPEN ccur;
   FETCH ccur INTO catalog_item;
   DBMS_OUTPUT.PUT_LINE('I fetched item #' || catalog_item.id);
   CLOSE ccur;
END;
```

The argument to PUT_LINE uses *variable.attribute* notation to yield the attribute value, resulting in the output:

```
I fetched item #10003
```

The fetch assigns the object to the local variable catalog_item, which is of the base type; this makes sense because I don't know in advance which subtype I'll be retrieving. My fetch simply assigns the object into the variable.

In addition to substitutability, the example also illustrates (by displaying catalog_item.id) that I have direct access to the base type's attributes.

In case you're wondering, normal cursor attribute tricks work too; the previous anonymous block is equivalent to:

```
DECLARE
   CURSOR ccur IS
      SELECT VALUE(c) obj
        FROM catalog_items c;
   arec ccur%ROWTYPE;
BEGIN
   OPEN ccur;
   FETCH ccur INTO arec;
   DBMS_OUTPUT.PUT_LINE('I fetched item #' || arec.obj.id);
   CLOSE ccur;
END;
```

If I just wanted to print out all of the object's attributes, I could, of course, use the print method I've already defined. It's legal to use this because it has been defined at the root type level and implemented in the subtypes; at runtime, the database will find the appropriate overriding implementations in each subtype. Ah, the beauty of dynamic method dispatch.

As a matter of fact, the VALUE function supports dot notation, which provides access to attributes and methods—but only those specified on the base type. For example, the following:

```
SELECT VALUE(c).id, VALUE(c).print()
  FROM catalog_items c;
```

yields this:

```
VALUE(C).ID VALUE(C).PRINT()
----------- ----------------------------------------------------------
      10003 id=10003; title=Perelandra; isbn=0-684-82382-9; pages=222
      10004 id=10004; title=Time; issn=0040-781X; open_or_closed=Open
```

If I happen to be working in a client environment that doesn't understand Oracle objects, I might want to take advantage of such features.

But what if I want to read only the attribute(s) unique to a particular subtype? I might first try something like this:

```
SELECT VALUE(c).issn   /* Error; subtype attributes are inaccessible */
  FROM catalog_items c;
```

This gives me *ORA-00904: invalid column name*. The Oracle database is telling me that an object of the parent type provides no direct access to subtype attributes. I might try declaring book of type book_t and assigning the subtyped object to it, hoping that it will expose the "hidden" attributes:

```
book := catalog_item;  /* Error; Oracle won't do implied downcasts */
```

This time I get *PLS-00382: expression is of wrong type*. What's going on? The nonintuitive answer to that mystery appears in the next section.

Before I move on, here are a few final notes about performing DML on object relational tables:

- For object tables built on object types that lack subtypes, it is possible to select, insert, update, and delete all column values using conventional SQL statements. In this way, some object-oriented and relational programs can share the same underlying data.
- You cannot perform conventional relational DML on hidden columns that exist as a result of subtype-dependent attributes. You must use an "object DML" approach.
- To update an entire persistent object from a PL/SQL program, you can use an object DML statement such as:

  ```
  UPDATE catalog_items c SET c = object_variable WHERE ...
  ```

  This updates all the attributes (columns), including those unique to a subtype.
- The only good way I have found to update a specific column that is unique to a subtype is to update the entire object. For example, to change the page count to 1,000 for the book with id 10007:

  ```
  UPDATE catalog_items c
     SET c = NEW book_t(c.id, c.title, c.publication_date, c.subject_refs,
                   (SELECT TREAT(VALUE(y) AS book_t).isbn
                      FROM catalog_items y
                     WHERE id = 10007),
                          1000)
   WHERE id = 10007;
  ```

Now let's go back and take a look at that last problem I mentioned.

The TREAT function

If I'm dealing with a PL/SQL variable typed as a supertype, and it's populated with a value of one of its subtypes, how can I gain access to the subtype-specific attributes and methods? In my case, I want to treat a generic catalog item as the more narrowly defined book. This operation is called *narrowing* or *downcasting*, and is something the compiler can't, or won't, do automatically. What I need to use is the Oracle function called TREAT:

```
DECLARE
   book book_t;
   catalog_item catalog_item_t := NEW book_t();
BEGIN
   book := TREAT(catalog_item AS book_t);   /* Using 9i R2 or later */
END;
```

or in SQL (note that in releases prior to Oracle9*i* Database Release 2 PL/SQL doesn't directly support TREAT):

```
DECLARE
   book book_t;
   catalog_item catalog_item_t := book_t(NULL, NULL, NULL, NULL);
BEGIN
   SELECT TREAT (catalog_item AS book_t)
     INTO book
     FROM DUAL;
END;
```

The general syntax of the TREAT function is:

```
TREAT (object_instance AS subtype) [ . { attribute | method( args...) } ]
```

where *object_instance* is any object with *subtype* as the name of one of its subtypes. Calls to TREAT won't compile if you attempt to treat one type as another from a different type hierarchy. One notable feature of TREAT is that if you have supplied an object from the correct type hierarchy, it will return either the downcasted object or NULL—but not an error.

As with VALUE, you can use dot notation with TREAT to specify an attribute or method of the TREATed object. For example:

```
DBMS_OUTPUT.PUT_LINE(TREAT (VALUE(c) AS serial_t).issn);
```

If I want to iterate over all the objects in the table in a type-aware fashion, I can do something like this:

```
DECLARE
   CURSOR ccur IS
      SELECT VALUE(c) item FROM catalog_items c;
   arec ccur%ROWTYPE;
BEGIN
   FOR arec IN ccur
   LOOP
      CASE
```

```
            WHEN arec.item IS OF (book_t)
            THEN
               DBMS_OUTPUT.PUT_LINE('Found a book with ISBN '
                  || TREAT(arec.item AS book_t).isbn);
            WHEN arec.item IS OF (serial_t)
            THEN
               DBMS_OUTPUT.PUT_LINE('Found a serial with ISSN '
                  || TREAT(arec.item AS serial_t).issn);
            ELSE
               DBMS_OUTPUT.PUT_LINE('Found unknown catalog item');
         END CASE;
      END LOOP;
   END;
```

This block introduces the IS OF predicate to test an object's type. Although the syntax is somewhat exciting:

```
    object IS OF ( [ ONLY ] typename )
```

the IS OF operator is much more limited than one would hope: it works only on object types, not on any of Oracle's core datatypes like NUMBER or DATE. Also, it will return an error if the *object* is not in the same type hierarchy as *typename*.

Notice the ONLY keyword. The default behavior—without ONLY—is to return TRUE if the object is of the given type *or any of its subtypes*. If you use ONLY, the expression won't check the subtypes and returns TRUE only if the type is an exact match.

Syntactically, you must always use the output from any TREAT expression as a function, even if you just want to call TREAT to invoke a member procedure. For example, you'd expect that if there were a set_isbn member procedure in book_t, you could do this:

```
    TREAT(item AS book_t).set_isbn('0140714154'); -- wrong
```

But that gives the curious compiler error *PLS-00363: expression 'SYS_TREAT' cannot be used as an assignment target.*

Instead, you need to store the item in a temporary variable, and then invoke the member procedure:

```
    book := TREAT(item AS book_t);
    book.set_isbn('0140714154');
```

The IS OF predicate, like TREAT itself, became available in Oracle9*i* Database Release 1 SQL, although direct support for it in PL/SQL didn't appear until Oracle9*i* Database Release 2. As a Release 1 workaround, I could define one or more additional methods in the type tree, taking advantage of dynamic method dispatch to perform the desired operation at the correct level in the hierarchy. The "correct" solution to the narrowing problem depends not just on the version number, though, but also on what my application is supposed to accomplish.

For the moment, I'd like to move on to another interesting area: exploring the features Oracle offers when (not if!) you have to deal with changes in application design.

Evolution and Creation

Oracle9*i* Database and later versions are light years beyond the Oracle8*i* Database in the area known as *type evolution*—that is, the later versions let you make a variety of changes to object types, even if you have created tables full of objects that depend on those types. Yippee!

Earlier in this chapter, I did a quick-and-dirty job of defining catalog_item_t. As almost any friendly librarian would point out, it might also be nice to carry publication date information[8] about all the holdings in the library. So I just hack out the following (no doubt while my DBA cringes):

```
ALTER TYPE catalog_item_t
   ADD ATTRIBUTE publication_date VARCHAR2(400)
   CASCADE INCLUDING TABLE DATA;
```

Et voilà! Oracle propagates this change to perform the needed physical alterations in the corresponding table(s). It appends the attribute to the bottom of the supertype's attributes and adds a column after the last column of the supertype in the corresponding object table. A DESCRIBE of the type now looks like this:

```
SQL> DESC catalog_item_t
 catalog_item_t is NOT FINAL
 catalog_item_t is NOT INSTANTIABLE
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 ID                                                 NUMBER(38)
 TITLE                                              VARCHAR2(4000)
 PUBLICATION_DATE                                   VARCHAR2(400)

METHOD
------
 MEMBER FUNCTION CK_DIGIT_OKAY RETURNS BOOLEAN
 CK_DIGIT_OKAY IS NOT INSTANTIABLE

METHOD
------
 MEMBER FUNCTION PRINT RETURNS VARCHAR2
```

And a DESCRIBE of the table now looks like this:

```
SQL> DESC catalog_items
 Name                                      Null?    Type
```

8. I can't make this attribute an Oracle DATE type, though, because sometimes it's just a year, sometimes a month or a quarter, and occasionally something completely offbeat. I might get really clever and make this a nifty object type... well, maybe in the movie version.

```
 ------------------------------------------ -------- ----------------------------
 ID                                         NOT NULL NUMBER(38)
 TITLE                                               VARCHAR2(4000)
 PUBLICATION_DATE                                    VARCHAR2(400)
```

In fact, the ALTER TYPE statement fixes *nearly* everything—though, alas, it isn't smart enough to rewrite my methods. My constructors are a particular issue because I need to alter their signature. Hey, no problem! I can change a method signature by dropping and then recreating the method.

> When evolving object types, you may encounter the message *ORA-22337: the type of accessed object has been evolved*. This condition may prevent you from doing a DESCRIBE on the type. You might think that recompiling it will fix the problem, but it won't. Moreover, if you have hard dependencies on the type, the Oracle database won't let you recompile the object type specification. To get rid of this error, disconnect and then reconnect your Oracle session. This clears various buffers and enables DESCRIBE to see the new version.

To drop the method from the book type specification, specify:

```
ALTER TYPE book_t
   DROP CONSTRUCTOR FUNCTION book_t (id INTEGER DEFAULT NULL,
      title VARCHAR2 DEFAULT NULL,
      isbn VARCHAR2 DEFAULT NULL,
      pages INTEGER DEFAULT NULL)
      RETURN SELF AS RESULT
   CASCADE;
```

Notice that I supply the full function specification. That will guarantee that I'm dropping the correct method because multiple overloaded versions of it might exist. (Strictly speaking, the DEFAULTs are not required, but I left them in because I'm usually just cutting and pasting this stuff.)

The corresponding add-method operation is easy:

```
ALTER TYPE book_t
   ADD CONSTRUCTOR FUNCTION book_t (id INTEGER DEFAULT NULL,
      title VARCHAR2 DEFAULT NULL,
      publication_date VARCHAR2 DEFAULT NULL,
      isbn VARCHAR2 DEFAULT NULL,
      pages INTEGER DEFAULT NULL)
      RETURN SELF AS RESULT
   CASCADE;
```

Easy for me, anyway; the database is doing a lot more behind the scenes than I will probably ever know.

The next steps (not illustrated in this chapter) would be to alter the serial_t type in a similar fashion and then rebuild the two corresponding object type bodies with the

CREATE OR REPLACE TYPE BODY statement. I would also want to inspect all the methods to see whether any changes would make sense elsewhere (for example, it would be a good idea to include the publication date in the print method).

By the way, you can drop a type using the statement:

```
DROP TYPE typename [ FORCE ];
```

Use the FORCE option (available only in Oracle Database 11*g* Release 2 and later) with care, because it cannot be undone. Any object types or object tables that depend on a force-dropped type will be rendered permanently useless. If there are any columns defined on a force-dropped type, the database marks them as UNUSED and makes them inaccessible. If your type is a subtype, and you have used the supertype in any table definitions, you might benefit from this form of the statement:

```
DROP TYPE subtypename VALIDATE;
```

VALIDATE causes the database to look through the table and drop the type only as long as there are no instances of the subtype, avoiding the disastrous consequences of the FORCE option.

Now let's visit the strange and fascinating world of *object referencing*.

Back to Pointers?

The object-relational features in Oracle include the ability to store an *object reference* or *REF value*. A REF is a *logical pointer* to a particular row in an object table. The Oracle database stores inside each reference the following information:

- The target row's primary key or system-generated object identifier
- A unique identifier to designate the table
- At the programmer's discretion, a hint on the row's physical whereabouts on disk, in the form of its ROWID

The literal contents of a REF are not terribly useful unless you happen to like looking at long hex strings:

```
SQL> SELECT REF(c) FROM catalog_items c WHERE ROWNUM = 1;
REF(C)
-------------------------------------------------------------------------------
00002802099FC431FBE5F20599E0340003BA0F1F139FC431FBE5F10599E0340003BA0F1F13024000
```

However, your queries and programs can use a REF to retrieve a row object without having to name the table where the object resides. Huh? Queries without table names? A pointer in a relational database? Let's take a look at how this feature might work in my library catalog.

Using REFs

Libraries classify their holdings within a strictly controlled set of subjects. For example, the Library of Congress might classify the book you're reading now in the following three subjects:

- Oracle (Computer file)
- PL/SQL (Computer program language)
- Relational databases

The Library of Congress uses a hierarchical subject tree: "Computer file" is the broader subject or parent of "Oracle," and "Computer program language" is the broader subject for "PL/SQL."

When classifying things, any number of subjects may apply to a particular catalog item in a many-to-many (M:M) relationship between subjects and holdings. In my simple library catalog, I will make one long list (table) of all available subjects. While a relational approach to the problem would then establish an "intersection entity" to resolve the M:M relationship, I have other options out here in object-relational land.

I will start with an object type for each subject:

```
CREATE TYPE subject_t AS OBJECT (
   name VARCHAR2(2000),
   broader_term_ref REF subject_t
);
```

Each subject has a name and a broader term. However, I'm not going to store the term itself as a second attribute, but instead a reference to it. The third line of this type definition shows that I've typed the broader_term_ref attribute as a REF to a same-typed object. It's kind of like Oracle's old EMP table, with an MGR column whose value identifies the manager's record in the same table.

I now create a table of subjects:

```
CREATE TABLE subjects OF subject_t
  (CONSTRAINT subject_pk PRIMARY KEY (name),
   CONSTRAINT subject_self_ref FOREIGN KEY (broader_term_ref)
      REFERENCES subjects);
```

The foreign key begs a bit of explanation. Even though it references a table with a relational primary key, because the foreign key datatype is a REF, Oracle knows to use the table's object identifier instead. This support for the REF-based foreign key constraint is a good example of Oracle's bridge between the object and relational worlds.

Here are a few unsurprising inserts into this table (just using the default constructor):

```
INSERT INTO subjects VALUES (subject_t('Computer file', NULL));
INSERT INTO subjects VALUES (subject_t('Computer program language', NULL));
INSERT INTO subjects VALUES (subject_t('Relational databases', NULL));
INSERT INTO subjects VALUES (subject_t('Oracle',
   (SELECT REF(s) FROM subjects s WHERE name = 'Computer file')));
INSERT INTO subjects VALUES (subject_t('PL/SQL',
   (SELECT REF(s) FROM subjects s WHERE name = 'Computer program language')));
```

For what it's worth, you can list the contents of the subjects table, as shown here:

```
SQL> SELECT VALUE(s) FROM subjects s;

VALUE(S)(NAME, BROADER_TERM_REF)
--------------------------------------------------------------------------------
SUBJECT_T('Computer file', NULL)
SUBJECT_T('Computer program language', NULL)
SUBJECT_T('Oracle', 00002202089FC431FBE6FB0599E0340003BA0F1F139FC431FBE6690599E03
40003BA0F1F13)

SUBJECT_T('PL/SQL', 00002202089FC431FBE6FC0599E0340003BA0F1F139FC431FBE6690599E03
40003BA0F1F13)

SUBJECT_T('Relational databases', NULL)
```

Even if that's interesting, it's not terribly useful. However, what's both interesting and useful is that I can easily have Oracle automatically "resolve" or follow those pointers. For example, I can use the DEREF function to navigate those ugly REFs back to their target rows in the table:

```
SELECT s.name, DEREF(s.broader_term_ref).name bt
  FROM subjects s;
```

Dereferencing is like an automatic join, although it's more of an outer join than an equijoin. In other words, if the reference is null or invalid, the driving row will still appear, but the target object (and column) will be null.

Oracle has introduced a dereferencing shortcut that is really quite elegant. You only need to use dot notation to indicate what attribute you wish to retrieve from the target object:

```
SELECT s.name, s.broader_term_ref.name bt FROM subjects s;
```

Both queries produce the following output:

```
NAME                           BT
------------------------------ ------------------------------
Computer file
Computer program language
Oracle                         Computer file
PL/SQL                         Computer program language
Relational databases
```

As a point of syntax, notice that both forms require a table alias, as in the following:

```
SELECT table_alias.ref_column_name.column_name
   FROM tablenametable_alias
```

You can also use REF-based navigation in the WHERE clause. To show all the subjects whose broader term is "Computer program language," specify:

```
SELECT VALUE(s).name FROM subjects s
 WHERE s.broader_term_ref.name = 'Computer program language';
```

Although my example table uses a reference to itself, in reality a reference can point to an object in any object table in the same database. To see this in action, let's return to the definition of the base type catalog_item_t. I can now add an attribute that will hold a collection of REFs, so that each cataloged item can be associated with any number of subjects. First, I'll create a collection of subject references:

```
CREATE TYPE subject_refs_t AS TABLE OF REF subject_t;
```

Now I'll allow every item in the catalog to be associated with any number of subjects:

```
ALTER TYPE catalog_item_t
   ADD ATTRIBUTE subject_refs subject_refs_t
   CASCADE INCLUDING TABLE DATA;
```

And now (skipping gleefully over the boring parts about modifying any affected methods in the dependent types), I might insert a catalog record using the following exotic SQL statement:

```
INSERT INTO catalog_items
VALUES (NEW book_t(10007,
   'Oracle PL/SQL Programming',
   'Sept 1997',
    CAST(MULTISET(SELECT REF(s)
                    FROM subjects s
                   WHERE name IN ('Oracle', 'PL/SQL', 'Relational databases'))
      AS subject_refs_t),
   '1-56592-335-9',
   987));
```

The CAST/MULTISET clause performs an on-the-fly conversion of the subject REFs into a collection, as explained in the section "Working with Collections" on page 365.

Here is a slightly more understandable PL/SQL equivalent:

```
DECLARE
   subrefs subject_refs_t;
BEGIN
   SELECT REF(s)
     BULK COLLECT INTO subrefs
     FROM subjects s
    WHERE name IN ('Oracle', 'PL/SQL', 'Relational databases');

   INSERT INTO catalog_items VALUES (NEW book_t(10007,
      'Oracle PL/SQL Programming', 'Sept 1997', subrefs, '1-56592-335-9', 987));
END;
```

In English, that code says "grab the REFs to three particular subjects, and store them with this particular book."

REF-based navigation is so cool that I'll show another example using some more of that long-haired SQL:

```
SELECT VALUE(s).name
   || ' (' || VALUE(s).broader_term_ref.name || ')' plsql_subjects
  FROM TABLE(SELECT subject_refs
               FROM catalog_items
              WHERE id=10007) s;
```

This example retrieves values from the subjects table, including the name of each broader subject term, without ever mentioning the subjects table by name. (The TABLE function converts a collection into a virtual table.) Here are the results:

```
PLSQL_SUBJECTS
------------------------------------
Relational databases ()
PL/SQL (Computer program language)
Oracle (Computer file)
```

Other than automatic navigation from SQL, what *else* does all this effort offer the PL/SQL programmer? Er, well, not a whole lot. References have a slight edge, at least because as theory goes, they are *strongly typed*—that is, a REF-typed column can point only to an object that is defined on the same object type as the REF. Contrast this behavior with conventional foreign keys, which can point to any old thing as long as the target is constrained to be a primary key or has a unique index on it.

The UTL_REF package

The UTL_REF built-in package performs the dereferencing operation without an explicit SQL call, allowing your application to perform a programmatic lock, select, update, or delete of an object given only its REF. As a short example, I can add a method such as the following to the subject_t type:

```
MEMBER FUNCTION print_bt (str IN VARCHAR2)
   RETURN VARCHAR2
IS
   bt subject_t;
BEGIN
   IF SELF.broader_term_ref IS NULL
   THEN
      RETURN str;
  ELSE
      UTL_REF.SELECT_OBJECT(SELF.broader_term_ref, bt);
      RETURN bt.print_bt(NVL(str,SELF.name)) || ' (' || bt.name || ')';
   END IF;
END;
```

This recursive procedure walks the hierarchy from the current subject to the "topmost" broader subject.

When using the procedures in UTL_REF, the REF argument you supply must be typed to match your object argument. The complete list of subprograms in UTL_REF follows:

UTL_REF.SELECT_OBJECT (`obj_ref` IN, `object_variable` OUT);
: Finds the object to which *obj_ref* points and retrieves a copy in *object_variable*.

UTL_REF.SELECT_OBJECT_WITH_CR (`obj_ref` IN, `object_variable` OUT);
: Like SELECT_OBJECT, but makes a copy ("snapshot") of the object. This version exists to avoid a mutating table error (ORA-4091), which can occur if you are updating an object table and setting the value to a function, but the function uses UTL_REF to dereference an object from the same table you're updating.

UTL_REF.LOCK_OBJECT (`obj_ref` IN);
: Locks the object to which *obj_ref* points but does not fetch it yet.

UTL_REF.LOCK_OBJECT (`obj_ref` IN, `object_variable` OUT);
: Locks the object to which *obj_ref* points and retrieves a copy in *object_variable*.

UTL_REF.UPDATE_OBJECT (`obj_ref` IN, `object_variable` IN);
: Replaces the object to which *obj_ref* points with the value supplied in *object_variable*. This operation updates all of the columns in the corresponding object table.

UTL_REF.DELETE_OBJECT (`obj_ref` IN);
: Deletes the object to which *obj_ref* points.

In C, Better Support for REFs

While PL/SQL offers few overwhelming reasons to program with object references, you'll find more benefits to this programming style with the Oracle Call Interface (OCI), Oracle's C/C++ language interface, or even with Pro*C. In addition to the ability to navigate REFs, similar to what you find in PL/SQL, OCI provides *complex object retrieval* (COR). With COR, you can retrieve an object and all its REFerenced neighbors in a single call. Both OCI and Pro*C support a client-side object cache, allowing an application to load objects into client memory and to manipulate (select, insert, update, merge, delete) them as if they were in the database. Then, in a single call, the application can flush all the changes back to the server. In addition to improving the programmer's functional repertoire, these features reduce the number of network roundtrips, improving overall performance. The downside: creating a cache of Oracle data outside the server invites a host of challenges relating to concurrency and locking.

REFs and type hierarchies

All of the UTL_REF subprograms are procedures, not functions,[9] and the parameters have the unique characteristic of being semiweakly typed. In other words, the database doesn't need to know at compile time what the precise datatypes are, as long as the REF matches the object variable.

I'd like to mention a few more technical points about REFs when dealing with type hierarchies. Assume the following program declarations:

```
DECLARE
   book book_t;
   item catalog_item_t;
   itemref REF catalog_item_t;
   bookref REF book_t;
```

As you have seen, assigning a REF to an "exactly typed" variable works fine:

```
SELECT REF(c) INTO itemref
  FROM catalog_items c WHERE id = 10007;
```

Similarly, you can dereference an object into the exact type, using:

```
UTL_REF.select_object(itemref, item);
```

or:

```
SELECT DEREF(itemref) INTO item FROM DUAL;
```

However, you cannot directly narrow a REF:

```
SELECT REF(c)
  INTO bookref    /* Error */
  FROM catalog_items c WHERE id = 10007;
```

One way to narrow a REF would be to use TREAT, which understands how to narrow references:

```
SELECT TREAT(REF(c) AS REF book_t)
  INTO bookref
  FROM catalog_items c WHERE id = 10007;
```

You can always widen or upcast while dereferencing, whether you are using:

```
UTL_REF.select_object(TREAT(bookref AS ref catalog_item_t), item);
```

(notice the explicit upcast) or:

```
SELECT DEREF(bookref) INTO item FROM DUAL;
```

And, although you cannot narrow or downcast while dereferencing with DEREF, as shown here:

9. I'm somewhat mystified by this; it would be a lot handier if at least SELECT_OBJECT were a function.

```
SELECT DEREF(itemref)
  INTO book   /* Error */
  FROM DUAL;
```

TREAT can again come to the rescue:

```
SELECT DEREF(TREAT(itemref AS REF book_t))
  INTO book
  FROM catalog_items c WHERE id = 10007;
```

Or, amazingly enough, you can perform an implicit downcast with UTL_REF:

```
UTL_REF.select_object(itemref, book);
```

Got all that?

Dangling REFs

Here are a few final comments about object references:

- A REF may point to nothing, in which case it's known as a *dangling REF*. This can happen when you store a reference to an object and then delete the object. Oracle permits such nonsense if you fail to define a foreign key constraint that would prevent it.
- To locate references that point to nothing, use the IS DANGLING operator:

  ```
  SELECT VALUE(s) FROM subjects s
  WHERE broader_term_ref IS DANGLING;
  ```

Now let's move on and take a look at some Oracle features for dealing with data whose type is either unknown or varying.

Generic Data: The ANY Types

As discussed in Chapter 13, Oracle provides the ANYDATA type, which can hold data in any other built-in or user-defined type. With ANYDATA, a PL/SQL program could, for instance, store, retrieve, and operate on a data item declared on any SQL type in the database—without having to create dozens of overloaded versions. Sounds pretty good, right? This feature was tailor-made for advanced queuing, where an application needs to put a "thing" in the queue, and you don't want the queue to have to know what the datatype of each item is.

The built-in packages and types in this family are:

ANYDATA type
: Encapsulation of any SQL-datatyped item in a self-descriptive data structure.

ANYTYPE type
: When used with ANYDATA, reads the description of the data structure. Can be used separately to create transient object types.

DBMS_TYPES package
: A package consisting only of constants that help interpret which datatype is being used in the ANYDATA object.

ANYDATASET type
: Similar to ANYDATA, but the contents are one or more instances of a datatype (like a collection).

Preview: What ANYDATA is not

If I wanted to write a function that would print anything (that is, convert it to a string), I might start with this spec:

```
FUNCTION printany (whatever IN ANYDATA) RETURN VARCHAR2;
```

and hope to invoke the function like this:

```
DBMS_OUTPUT.PUT_LINE(printany(SYSDATE));          -- nope
DBMS_OUTPUT.PUT_LINE(printany(NEW book_t(111)));  -- nada
DBMS_OUTPUT.PUT_LINE(printany('Hello world'));    -- nyet
```

Unfortunately, those calls won't work. ANYDATA is actually an encapsulation of other types, and you must first *convert* the data into the ANYDATA type using one of its built-in static methods:

```
DBMS_OUTPUT.PUT_LINE(printany(ANYDATA.ConvertDate(SYSDATE));
DBMS_OUTPUT.PUT_LINE(printany(ANYDATA.ConvertObject(NEW book_t(12345)));
DBMS_OUTPUT.PUT_LINE(printany(ANYDATA.ConvertVarchar2('Hello world')));
```

Don't think of ANYDATA as an exact replacement for overloading.

Dealing with ANYDATA

Let's take a look at an implementation of the printany program and see how it figures out how to deal with data of different types. This code is not comprehensive—it deals only with numbers, strings, dates, objects, and REFs—but you could extend it to almost any other datatype:

```
/* File on web: printany.fun */
FUNCTION printany (adata IN ANYDATA)
   RETURN VARCHAR2
AS
   aType ANYTYPE;
   retval VARCHAR2(32767);
   result_code PLS_INTEGER;
BEGIN
   CASE adata.GetType(aType)
   WHEN DBMS_TYPES.TYPECODE_NUMBER THEN
       RETURN 'NUMBER: ' || TO_CHAR(adata.AccessNumber);
    WHEN DBMS_TYPES.TYPECODE_VARCHAR2 THEN
       RETURN 'VARCHAR2: ' || adata.AccessVarchar2;
    WHEN DBMS_TYPES.TYPECODE_CHAR THEN
```

```
14          RETURN 'CHAR: ' || RTRIM(adata.AccessChar);
15       WHEN DBMS_TYPES.TYPECODE_DATE THEN
16          RETURN 'DATE: ' || TO_CHAR(adata.AccessDate,
17                                'YYYY-MM-DD hh24:mi:ss');
18       WHEN DBMS_TYPES.TYPECODE_OBJECT THEN
19          EXECUTE IMMEDIATE 'DECLARE ' ||
20              '   myobj ' || adata.GetTypeName || '; ' ||
21              '   myad anydata := :ad; ' ||
22              'BEGIN ' ||
23              '   :res := myad.GetObject(myobj); ' ||
24              '   :ret := myobj.print(); ' ||
25              'END;'
26               USING IN adata, OUT result_code, OUT retval;
27          retval := adata.GetTypeName || ': ' || retval;
28       WHEN DBMS_TYPES.TYPECODE_REF THEN
29          EXECUTE IMMEDIATE 'DECLARE ' ||
30              '   myref ' || adata.GetTypeName || '; ' ||
31              '   myobj ' || SUBSTR(adata.GetTypeName,
32                           INSTR(adata.GetTypeName, ' ')) || '; ' ||
33              '   myad anydata := :ad; ' ||
34              'BEGIN ' ||
35              '   :res := myad.GetREF(myref); ' ||
36              '   UTL_REF.SELECT_OBJECT(myref, myobj);' ||
37              '   :ret := myobj.print(); ' ||
38              'END;'
39               USING IN adata, OUT result_code, OUT retval;
40          retval := adata.GetTypeName || ': ' || retval;
41       ELSE
42          retval := '<data of type ' || adata.GetTypeName ||'>';
43       END CASE;
44
45       RETURN retval;
46
47    EXCEPTION
48       WHEN OTHERS
49       THEN
50          IF INSTR(SQLERRM, 'component ''PRINT'' must be declared') > 0
51          THEN
52             RETURN adata.GetTypeName || ': <no print() function>';
53          ELSE
54             RETURN 'Error: ' || SQLERRM;
55          END IF;
56    END;
```

The following table describes just a few highlights.

Line(s)	Description
5	In cases where I need a temporary variable to hold the result, I assume that 32 KB will be big enough. Remember that PL/SQL dynamically allocates memory for large VARCHAR2s, so it won't be a memory pig unless required.
6	The value of result_code (see lines 26 and 39) is irrelevant for the operations in this example, but is required by the ANYDATA API.

Line(s)	Description

8	The ANYDATA type includes a method called GetType that returns a code corresponding to the datatype. Here is its specification:
	`MEMBER FUNCTION ANYDATA.GetType`
	`  (OUT NOCOPY ANYTYPE)`
	`  RETURN` *`typecode_integer`*`;`
	To use this method, though, you have to declare an ANYTYPE variable into which Oracle will store detailed information about the type that you've encapsulated.

9, 11, 13, 15, 18, 28	These expressions rely on the constants that Oracle provides in the built-in package DBMS_TYPES.
10, 12, 14, 16	These statements use the ANYDATA.Access*NNN* member functions introduced in Oracle9*i* Database Release 2. In Release 1, you had to use the Get*NNN* member procedures for a similar result, although they required the use of a temporary local variable.
19–26	To get an object to print itself without doing a lot of data dictionary contortions, this little dynamic anonymous block will construct an object of the correct type and invoke its print() member method. You did give it a print(), didn't you?
29–39	The point of this is to dereference the pointer and return the referenced object's content. Well, it will work if there's a print().
45–52	In the event that I'm trying to print an object with no print member method, the compiler will return an error at runtime that I can detect in this fashion. In this case, the code will just punt and return a generic message.

Running my earlier invocations:

```
DBMS_OUTPUT.PUT_LINE(printany(ANYDATA.ConvertDate(SYSDATE));
DBMS_OUTPUT.PUT_LINE(printany(ANYDATA.ConvertObject(NEW book_t(12345)));
DBMS_OUTPUT.PUT_LINE(printany(ANYDATA.ConvertVarchar2('Hello world')));
```

yields:

```
DATE: 2005-03-10 16:00:25
SCOTT.BOOK_T: id=12345; title=; publication_date=; isbn=; pages=
VARCHAR2: Hello world
```

As you can see, using ANYDATA isn't as convenient as true inheritance hierarchies because ANYDATA requires explicit conversions. However, it does let you create a table column or object attribute that will hold any type of data.[10]

Creating a transient type

Although PL/SQL still does not support defining new object types inside a program's declaration section, it is possible to use these ANY built-ins to create this kind of "transient" type—that is, one that exists only at runtime. Wrapped up as an ANYTYPE, you can even pass such a type as a parameter and create an instance of it as an ANYDATA. Here is an example:

10. As of this writing, it is impossible to store in a table an ANYDATA encapsulating an object that has evolved or that is part of a type hierarchy.

```
/* Create (anonymous) transient type with two attributes: number, date */
FUNCTION create_a_type
   RETURN ANYTYPE
AS
   mytype ANYTYPE;
BEGIN
   ANYTYPE.BeginCreate(typecode => DBMS_TYPES.TYPECODE_OBJECT,
                       atype => mytype);
   mytype.AddAttr(typecode => DBMS_TYPES.TYPECODE_NUMBER,
                  aname => 'just_a_number',
                  prec => 38,
                  scale => 0,
                  len => NULL,
                  csid => NULL,
                  csfrm => NULL);
   mytype.AddAttr(typecode => DBMS_TYPES.TYPECODE_DATE,
                  aname => 'just_a_date',
                  prec => 5,
                  scale => 5,
                  len => NULL,
                  csid => NULL,
                  csfrm => NULL);
   mytype.EndCreate;
   RETURN mytype;
END;
```

As you can see, there are three main steps:

1. Begin the creation by calling the static procedure BeginCreate. This returns an initialized ANYTYPE.
2. One at a time, add the desired attributes using the AddAttr member procedure.
3. Call the member procedure EndCreate.

Similarly, when you wish to use the type, you will need to assign attribute values in a piecewise manner:

```
DECLARE
   ltype ANYTYPE := create_a_type;
   l_any ANYDATA;
BEGIN
   ANYDATA.BeginCreate(dtype => ltype, adata => l_any);
   l_any.SetNumber(num => 12345);
   l_any.SetDate(dat => SYSDATE);
   l_any.EndCreate;
END;
```

If you don't know the structure of the datatype in advance, you can discover it using ANYTYPE methods (such as GetAttrElemInfo) in combination with a piecewise application of the ANYDATA.Get methods. (See the *anyObject.sql* script on the book's website for an example.)

I Can Do It Myself

In object-oriented design, there is a school of thought that wants each object type to have the intelligence necessary to be self-sufficient. If the object needs to be stored persistently in a database, it should know how to save itself; similarly, it should include methods for update, delete, and retrieval. If I subscribed to this philosophy, here is one of the methods I would want to add to my type:

```
ALTER TYPE catalog_item_t
   ADD MEMBER PROCEDURE remove
   CASCADE;

TYPE BODY catalog_item_t
AS
   ...
   MEMBER PROCEDURE remove
   IS
   BEGIN
      DELETE catalog_items
       WHERE id = SELF.id;
      SELF := NULL;
   END;
END;
```

(Oracle does not offer a destructor method, by the way.) By defining this method at the supertype level, I take care of all my subtypes, too. This design assumes that corresponding objects will live in a single table; some applications might need some additional logic to locate the object. (Also, a real version of this method might include logic to perform ancillary functions like removing dependent objects and/or archiving the data before removing the object permanently.)

Assuming that my applications would always modify a transient object in memory before writing it to disk, I could combine insert and update into a single method I'll call "save":

```
ALTER TYPE catalog_item_t
   ADD MEMBER PROCEDURE save,
   CASCADE;

TYPE BODY catalog_item_t
AS
   ...
   MEMBER PROCEDURE save
   IS
   BEGIN
      UPDATE catalog_items c
         SET c = SELF
       WHERE id = SELF.id;
       IF SQL%ROWCOUNT = 0
       THEN
          INSERT INTO catalog_items VALUES (SELF);
```

```
        END IF;
    END;
```

You may correctly point out that this will replace all of the column values in the table even if they are unchanged, which could cause triggers to fire that shouldn't and results in needless I/O. Alas, this is one of the unfortunate by-products of an object approach. It is true that with careful programming, you could avoid modifying columns from the supertype that haven't changed, but columns from any subtype are not individually accessible from any variation on the UPDATE statement that Oracle currently offers.

Retrieval is the most difficult operation to encapsulate because of the many WHERE-clause permutations and the multiset nature of the result. The specification of the query criteria can be a real rat's nest, as anyone who has ever built a custom query screen will attest. If we consider only the result side, the options for what to return include:

- A collection of objects
- A collection of REFs
- A pipelined result set
- A cursor variable (strongly or weakly typed)

The requirements of the application and its programming environment will have the largest influence on how to choose from these options. Here's a stripped-down example that uses the fourth approach, a cursor variable:

```
ALTER TYPE catalog_item_t
   ADD STATIC FUNCTION cursor_for_query (typename IN VARCHAR2 DEFAULT NULL,
      title IN VARCHAR2 DEFAULT NULL,
      att1 IN VARCHAR2 DEFAULT NULL,
      val1 IN VARCHAR2 DEFAULT NULL)
      RETURN SYS_REFCURSOR
   CASCADE;
```

I use a static method that returns the built-in SYS_REFCURSOR type, which is a weak cursor type that Oracle provides (just something of a convenience feature), allowing the client program to iterate over the results. The att1 and val1 parameters provide a means of querying subtype-specific attribute/value pairs; a real version of this program would be better off accepting a collection of such attribute/value pairs to allow queries on multiple attributes of a given subtype.

Jumping ahead to how you might execute a query, let's look at this example:

```
DECLARE
   catalog_item catalog_item_t;
   l_refcur SYS_REFCURSOR;
BEGIN
   l_refcur := catalog_item_t.cursor_for_query(
      typename => 'book_t',
      title => 'Oracle PL/SQL Programming');
   LOOP
      FETCH l_refcur INTO catalog_item;
      EXIT WHEN l_refcur%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE('Matching item:' || catalog_item.print);
   END LOOP;
   CLOSE l_refcur;
END;
```

which yields:

```
Matching item:id=10007; title=Oracle PL/SQL Programming;
 publication_date=Sept 1997;
isbn=1-56592-335-9; pages=987
```

The implementation is:

```
MEMBER PROCEDURE save
IS
BEGIN
   UPDATE catalog_items c
      SET c = SELF
    WHERE id = SELF.id;
    IF SQL%ROWCOUNT = 0
    THEN
       INSERT INTO catalog_items VALUES (SELF);
     END IF;
 END;

STATIC FUNCTION cursor_for_query (typename IN VARCHAR2 DEFAULT NULL,
    title IN VARCHAR2 DEFAULT NULL,
    att1 IN VARCHAR2 DEFAULT NULL,
    val1 IN VARCHAR2 DEFAULT NULL)
    RETURN SYS_REFCURSOR
 IS
    l_sqlstr VARCHAR2(1024);
    l_refcur SYS_REFCURSOR;
 BEGIN
    l_sqlstr := 'SELECT VALUE(c) FROM catalog_items c WHERE 1=1 ';
    IF title IS NOT NULL
    THEN
      l_sqlstr := l_sqlstr || 'AND title = :t ';
    END IF;

    IF typename IS NOT NULL
    THEN
       IF att1 IS NOT NULL
       THEN
          l_sqlstr := l_sqlstr
             || 'AND TREAT(SELF AS '
               || typename || ').' || att1 || ' ';
          IF val1 IS NULL
          THEN
             l_sqlstr := l_sqlstr || 'IS NULL ';
         ELSE
```

```
               l_sqlstr := l_sqlstr || '=:v1 ';
            END IF;
         END IF;
         l_sqlstr := l_sqlstr || 'AND VALUE(c) IS OF
           (' || typename ||') ';
      END IF;

      l_sqlstr := 'BEGIN OPEN :lcur FOR ' || l_sqlstr || '; END;';

      IF title IS NULL AND att1 IS NULL
      THEN
         EXECUTE IMMEDIATE l_sqlstr USING IN OUT l_refcur;
      ELSIF title IS NOT NULL AND att1 IS NULL
      THEN
         EXECUTE IMMEDIATE l_sqlstr USING IN OUT l_refcur, IN title;
      ELSIF title IS NOT NULL AND att1 IS NOT NULL
      THEN
         EXECUTE IMMEDIATE l_sqlstr
              USING IN OUT l_refcur, IN title, IN att1;
      END IF;

      RETURN l_refcur;
   END;
```

Because dynamic SQL is a little tricky to follow, here is what the function would have generated internally with the previous query:

```
BEGIN
   OPEN :lcur FOR
      SELECT VALUE(c)
        FROM catalog_items c
       WHERE 1=1
         AND title = :t
         AND VALUE(c) IS OF (book_t);
END;
```

One nice thing about this approach is that you don't have to modify the query code every time you add a subtype to the inheritance tree.

Comparing Objects

So far, my examples have used object tables—tables in which each row constitutes an object built with the CREATE TABLE...OF type statement. As I've illustrated, such an arrangement enjoys some special features, such as REF-based navigation and the treatment of entire objects (rather than individual column values) as the unit of I/O.

You can also use an object type as the datatype for individual columns in a table (the relevant nomenclature is *column objects*, as mentioned earlier). For example, imagine that I want to create a historical record of changes in the catalog_items table, capturing all inserts, updates, and deletes. I can do this as follows:

```
CREATE TABLE catalog_history (
   id INTEGER NOT NULL PRIMARY KEY,
   action CHAR(1) NOT NULL,
   action_time TIMESTAMP DEFAULT (SYSTIMESTAMP) NOT NULL,
   old_item catalog_item_t,
   new_item catalog_item_t)
   NESTED TABLE old_item.subject_refs STORE AS catalog_history_old_subrefs
   NESTED TABLE new_item.subject_refs STORE AS catalog_history_new_subrefs;
```

As soon as you start populating a table with column objects, though, you raise some questions about how Oracle should behave when you ask it to do things like sort or index on one of those catalog_item_t columns. There are four ways you can compare objects, some of which are more useful than others:

Attribute-level comparison
: You can include the relevant attribute(s) when sorting, creating indexes, or comparing.

Default SQL
: Oracle's SQL knows how to do a simple equality test. In this case, two objects are considered equal if they are defined on exactly the same type and every corresponding attribute is equal. This will work if the objects have only scalar attributes (no collections or LOBs) and if you haven't already defined a MAP or ORDER member method on the object type.

MAP member method
: You can create a special function method that returns a "mapping" of the object value onto a datatype that Oracle already knows how to compare, such as a number or a date. This will work only if no ORDER method exists.

ORDER member method
: This is another special function that compares two objects and returns a flag value that indicates their relative ordering. This will work only if no MAP method exists.

Default SQL comparison is not terribly useful, so I won't say any more about it. The following sections describe the other, more useful ways to compare objects.

The OBJECT_VALUE Pseudocolumn

Curious readers may wonder how, precisely, one could automatically populate an audit-style table such as catalog_history, which includes column objects defined on a type that has subtypes. You might hope that it could be done with a table-level trigger.

The difficult question is how to capture the values of the attributes for all the subtypes. There is no obvious way to refer to them generically. No problem... Pseudocolumn Man comes to the rescue! Ponder this:

```
   TRIGGER catalog_hist_upd_trg
   AFTER UPDATE ON catalog_items
   FOR EACH ROW
   BEGIN
      INSERT INTO catalog_history (id,
         action,
         action_time,
         old_item,
         new_item)
      VALUES (catalog_history_seq.NEXTVAL,
         'U',
         SYSTIMESTAMP,
         :OLD.OBJECT_VALUE,
         :NEW.OBJECT_VALUE);
   END;
```

Oracle provides access to the fully attributed subtypes via the pseudocolumn OBJECT_VALUE. However, this works only if you have Oracle Database 10*g* or later; it's true that a similar pseudocolumn SYS_NC_ROWINFO$ is available in earlier versions, but I have found that it does not work in this particular application.

OBJECT_VALUE can also be used for other purposes and is not limited to circumstances involving subtypes; for example, it can be useful when you are creating object views using the WITH OBJECT IDENTIFIER clause (discussed later in this chapter).

Attribute-level comparison

Attribute-level comparison may not be precisely what you want, but it is fairly easy in PL/SQL, or even in SQL if you remember to use a table alias in the SQL statement. Oracle lets you expose attributes via dot notation:

```
SELECT * FROM catalog_history c
 WHERE c.old_item.id > 10000
 ORDER BY NVL(TREAT(c.old_item as book_t).isbn, TREAT
(c.old_item AS serial_t).issn)
```

Attribute-level index creation is equally easy:

```
CREATE INDEX catalog_history_old_id_idx ON catalog_history c (c.old_item.id);
```

The MAP method

Both the MAP and the ORDER methods make it possible to perform statements such as the following:

```
SELECT * FROM catalog_history
 ORDER BY old_item;

IF old_item > new_item
THEN ...
```

First let's look at MAP. I can add a trivial MAP method to catalog_item_t as follows:

```
ALTER TYPE catalog_item_t
   ADD MAP MEMBER FUNCTION mapit RETURN NUMBER
   CASCADE;

TYPE BODY catalog_item_t
AS ...
   MAP MEMBER FUNCTION mapit RETURN NUMBER
   IS
   BEGIN
      RETURN id;
   END;
   ...
END;
```

Assuming, of course, that ordering by id makes sense, now I can order and compare catalog items to my heart's content, and the Oracle database will call this method automatically whenever necessary. The function needn't be so simple; for example, it could return a scalar value computed from all the object attributes, melded together in some way that actually might be of some value to librarians.

Creating a MAP method like this has a side effect, though: the equality comparison gets defined in a way you might not like. "Equality" now becomes a matter of the mapped value being equal for the objects you're comparing. If you want an easy way to compare two objects for attribute-by-attribute equality, you will want to either create your own (non-MAP) method and invoke it by name when needed, or use an ORDER method.

The ORDER method

The alternative to MAP is an ORDER member function, which compares two objects: SELF, and another object of the same type that you supply as an argument. You want to program the function to return an integer that is positive, zero, or negative, indicating the ordering relationship of the second object to SELF. Table 26-2 illustrates the behavior you need to incorporate.

Table 26-2. Desired behavior of ORDER member functions

For these desired semantics...	Your ORDER member function must return
SELF < *argumentObject*	Any negative number (typically –1)
SELF = *argumentObject*	0
SELF > *argumentObject*	Any positive number (typically +1)
Undefined comparison	NULL

Let's take a look at a nontrivial example of an ORDER method:

```
1   ALTER TYPE catalog_item_t
2      DROP MAP MEMBER FUNCTION mapit RETURN NUMBER
3      CASCADE;
```

```

ALTER TYPE catalog_item_t
   ADD ORDER MEMBER FUNCTION orderit (obj2 IN catalog_item_t)
      RETURN INTEGER
   CASCADE;

 TYPE BODY catalog_item_t
 AS ...
    ORDER MEMBER FUNCTION orderit (obj2 IN catalog_item_t)
       RETURN INTEGER
    IS
       self_gt_o2 CONSTANT PLS_INTEGER := 1;
       eq CONSTANT PLS_INTEGER := 0;
       o2_gt_self CONSTANT PLS_INTEGER := -1;
       l_matching_count NUMBER;
    BEGIN
       CASE
          WHEN obj2 IS OF (book_t) AND SELF IS OF (serial_t) THEN
             RETURN o2_gt_self;
          WHEN obj2 IS OF (serial_t) AND SELF IS OF (book_t) THEN
             RETURN self_gt_o2;
          ELSE
             IF obj2.title = SELF.title
                AND obj2.publication_date = SELF.publication_date
             THEN
                IF obj2.subject_refs IS NOT NULL
                   AND SELF.subject_refs IS NOT NULL
                   AND obj2.subject_refs.COUNT = SELF.subject_refs.COUNT
                THEN
                   SELECT COUNT(*) INTO l_matching_count FROM
                   (SELECT * FROM TABLE
                    (SELECT CAST(SELF.subject_refs AS subject_refs_t)
                                    FROM dual)
                      INTERSECT
                      SELECT *FROM TABLE
                       (SELECT CAST(obj2.subject_refs AS subject_refs_t)
                                     FROM dual));
                   IF l_matching_count = SELF.subject_refs.COUNT
                   THEN
                      RETURN eq;
                   END IF;
                END IF;
             END IF;
             RETURN NULL;
       END CASE;
    END;
    ...
 END;
```

The following table describes the important things to note.

Line(s)	Description
21–24	This means that "books sort higher than serials."
26–46	This is an equality test that uses a very cool feature. Because Oracle doesn't know how to compare collections, this code uses Oracle's ability to select from a collection as if it were a table. By checking to make sure that the relational intersection of these two collections has the expected number of elements, I can determine whether every element in the first collection has an equal counterpart in the second (which is my definition of *equality*).

Overall, my ORDER method is still inadequate because it fails to treat the subtype-specific attributes, but anything longer would just be too unwieldy for this book.

Additional comparison recommendations

To close out this discussion, here are a few additional rules and recommendations for comparison methods:

- MAP and ORDER cannot coexist in the same object type; use one or the other.
- Oracle recommends using MAP when you have a large number of objects to sort or compare, as in a SQL statement. This is because of an internal optimization that reduces the number of function calls. With ORDER, the function must run once for every comparison.
- Oracle ignores the method names; you can call them anything you want.
- Subtypes can include MAP methods, but only if the supertype also has one.
- Subtypes cannot have ORDER methods; you'll have to put all the comparison "smarts" into the supertype.

Object Views

Although Oracle's object extensions offer PL/SQL programmers rich possibilities for the design of new systems, it's unlikely that you will want to completely reengineer your existing systems to use objects. In part to allow established applications to take advantage of the new object features over time, Oracle provides *object views*. This feature offers several unique advantages:

"Object-ification" of remote data
: It's not yet possible to use the object tables and physical REFs across a distributed database, but you can create object views and virtual REFs that cast remote relational data as objects.

Virtual denormalization
: In a relational database or even an object-relational database, you will usually find relationships modeled in only one direction. For example, a book has some number of subjects. With an object view, it's easy to associate a column that provides the

inverse mapping; for example, a subject object could include a collection of REFs that point to all of the books in that subject.

Efficiency of object access
In Oracle Call Interface (OCI) applications, object programming constructs provide for the convenient retrieval, caching, and updating of object data. By reducing trips between the application and the database server, these programming facilities may provide performance improvements, with the added benefit that application code can be more succinct.

Greater flexibility to change the object model
Although newer versions of Oracle have tremendous abilities in the area of type evolution, adding and removing object attributes still cause table bits to move around on the disk, which administrators may be loath to do. Recompiling object views suffers no such consequences.

On the other hand, there are some disadvantages to using object views:

View performance
Object views are still views, and some Oracle shops are generally leery of the performance of any view.

No virtual REFs
You cannot store virtual REFs in the database; instead, they get constructed on the fly. This may present some challenges if you someday want to convert those object views into object tables.

Other features of Oracle can improve the expressiveness of any types of views, not just object views. Two such features that are not strictly limited to object views are collections and INSTEAD OF triggers:

Collections
Consider two relational tables with a simple master-detail relationship. You can create a view portraying the detail records as a single nonscalar attribute (collection) of the master.

INSTEAD OF triggers
In addition, by using INSTEAD OF triggers, you can tell the Oracle database exactly how to perform inserts, updates, and deletes on the view.

From an object perspective, object views have one slight disadvantage when compared to comprehensive reengineering: object views cannot retrofit any benefits of encapsulation. Insofar as any applications apply INSERT, UPDATE, MERGE, and DELETE statements directly to the underlying relational data, they may subvert the benefits of encapsulation normally provided by an object approach. Object-oriented designs typically prevent free-form access directly to data. However, because Oracle supports neither private attributes nor private methods, the incremental sacrifice here is small.

If you do choose to layer object views on top of an existing system, it may be possible for new applications to enjoy incremental benefits, and your legacy systems will be no worse off than they were before. Figure 26-2 illustrates this use of object views.

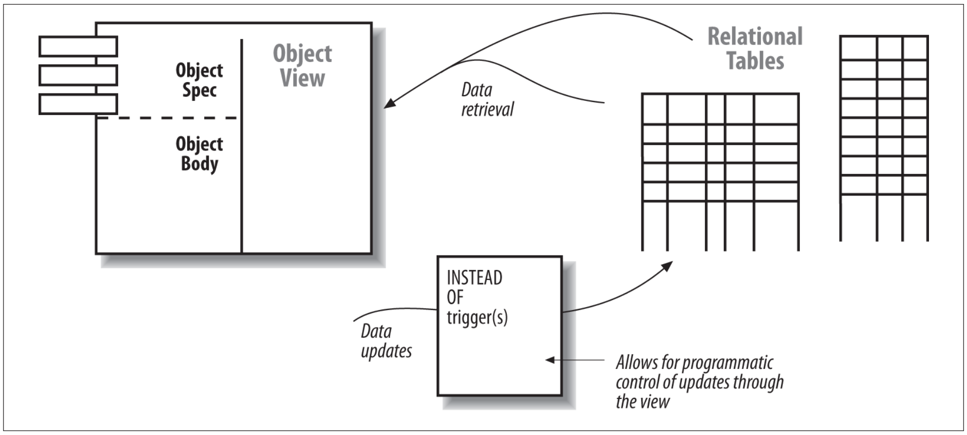

Figure 26-2. Object views allow you to bind an object type definition to (existing) relational tables

The following sections discuss aspects of using object views (including differences between object tables and object views) that PL/SQL programmers should find particularly useful and interesting.

A Sample Relational System

For this chapter's second major example, let's look at how object views might be used in a database application that supports a graphic design firm. Their relational application includes information about images (GIF, JPEG, etc.) that appear on websites they design. These images are stored in files, but data about them is stored in relational tables. To help the graphic artists locate the right images, each image has one or more associated keywords stored in a straightforward master-detail relationship.

The legacy system has a table of suppliers:

```
CREATE TABLE suppliers (
   id INTEGER NOT NULL PRIMARY KEY,
   name VARCHAR2(400) NOT NULL
);
```

Here is the table for image metadata:

```
CREATE TABLE images (
   image_id INTEGER NOT NULL PRIMARY KEY,
   file_name VARCHAR2(512) NOT NULL,
   file_type VARCHAR2(12) NOT NULL,
   supplier_id INTEGER REFERENCES suppliers (id),
   supplier_rights_descriptor VARCHAR2(256),
   bytes INTEGER
);
```

Not all images originate from suppliers; if the supplier id is null, then the image was created in-house.

Finally, there is one table for the keywords associated with the images:

```
   CREATE TABLE keywords (
      image_id INTEGER NOT NULL REFERENCES images (image_id),
      keyword VARCHAR2(45) NOT NULL,
      CONSTRAINT keywords_pk PRIMARY KEY (image_id, keyword)
   );
```

Let's assume that the following data exists in the underlying tables:

```
   INSERT INTO suppliers VALUES (101, 'Joe''s Graphics');
   INSERT INTO suppliers VALUES (102, 'Image Bar and Grill');
   INSERT INTO images VALUES (100001, '/files/web/60s/smiley_face.png',
     'image/png', 101, 'fair use', 813);
   INSERT INTO images VALUES (100002, '/files/web/60s/peace_symbol.gif',
     'image/gif', 101, 'fair use', 972);
   INSERT INTO images VALUES (100003, '/files/web/00s/towers.jpg',
    'image/jpeg', NULL,
      NULL, 2104);
   INSERT INTO KEYWORDS VALUES (100001, 'SIXTIES');
   INSERT INTO KEYWORDS VALUES (100001, 'HAPPY FACE');
   INSERT INTO KEYWORDS VALUES (100002, 'SIXTIES');
   INSERT INTO KEYWORDS VALUES (100002, 'PEACE SYMBOL');
   INSERT INTO KEYWORDS VALUES (100002, 'JERRY RUBIN');
```

In the next few sections, you'll see several object views defined on this data:

- The first view is defined on an image type that includes the keywords as a collection attribute.
- The second view is a *subview*—that is, it's defined on a subtype in an object type hierarchy. It will include characteristics for images that originate from suppliers.
- The final view includes keywords and their inverse references back to the relevant images.

Object View with a Collection Attribute

Before creating an underlying type for the first view, I need a collection type to hold the keywords. Use of a nested table makes sense here, because keyword ordering is unimportant and because there is no logical maximum number of keywords:[11]

```
CREATE TYPE keyword_tab_t AS TABLE OF VARCHAR2(45);
```

At this point, it's a simple matter to define the image object type:

```
CREATE TYPE image_t AS OBJECT (
    image_id INTEGER,
    image_file BFILE,
    file_type VARCHAR2(12),
    bytes INTEGER,
    keywords keyword_tab_t
);
```

Assuming that the image files and the database server are on the same machine, I can use an Oracle BFILE datatype rather than the filename. I'll need to create a "directory"—that is, an alias by which the database will know the directory that contains the images. In this case, I use the root directory (on the target Unix system, this is represented by a single forward slash), because I happen to know that the file_name column includes full pathnames:

```
CREATE DIRECTORY rootdir AS '/';
```

You likely will not have privileges to work with files in the root directory; set your directory to a folder in which you can work.

So far, I have not defined a connection between the relational tables and the object type. They are independent organisms. It is in building the object view that I overlay the object definition onto the tables, as the next statement illustrates:

```
CREATE VIEW images_v
   OF image_t
   WITH OBJECT IDENTIFIER (image_id)
AS
   SELECT i.image_id, BFILENAME('ROOTDIR', i.file_name),
      i.file_type, i.bytes,
      CAST (MULTISET (SELECT keyword
                        FROM keywords k
                       WHERE k.image_id = i.image_id)
```

11. If ordering were important or if there were a (small) logical maximum number of keywords per image, a VARRAY collection would be a better choice.

```
          AS keyword_tab_t)
       FROM images i;
```

There are two components of this statement that are unique to object views:

OF image_t
: This means that the view will return objects of type image_t.

WITH OBJECT IDENTIFIER (image_id)
: To behave like a "real" object instance, data returned by the view will need some kind of object identifier. By designating the primary key as the basis of a virtual OID, I can enjoy the benefits of REF-based navigation to objects in the view.

In addition, the select list of an object view must correspond in number, position, and datatype with the attributes in the associated object type.

OK, now that I've created an object view, what can I do with it? Most significantly, I can retrieve data from it just as if it were an object table. So, from SQL*Plus, a query like the following:

```
SQL> SELECT image_id, keywords FROM images_v;
```

yields:

```
  IMAGE_ID KEYWORDS
---------- -------------------------------------------------------
    100003 KEYWORD_TAB_T()
    100001 KEYWORD_TAB_T('HAPPY FACE', 'SIXTIES')
    100002 KEYWORD_TAB_T('JERRY RUBIN', 'PEACE SYMBOL', 'SIXTIES')
```

In the interest of deepening the object appearance, I could also add methods to the type definition. Here, for example, is a print method:

```
ALTER TYPE image_t
   ADD MEMBER FUNCTION print RETURN VARCHAR2
   CASCADE;

CREATE OR REPLACE TYPE BODY image_t
AS
   MEMBER FUNCTION print
      RETURN VARCHAR2
   IS
      filename images.file_name%TYPE;
      dirname VARCHAR2(30);
      keyword_list VARCHAR2(32767);
   BEGIN
      DBMS_LOB.FILEGETNAME(SELF.image_file, dirname, filename);
      IF SELF.keywords IS NOT NULL
      THEN
         FOR key_elt IN 1..SELF.keywords.COUNT
         LOOP
            keyword_list := keyword_list || ', ' || SELF.keywords(key_elt);
         END LOOP;
      END IF;
      RETURN 'Id=' || SELF.image_id || '; File=' || filename
         || '; keywords=' || SUBSTR(keyword_list, 3);
   END;
END;
```

This example illustrates a way to "flatten" the keyword list by iterating over the virtual collection of keywords.

Is It Null, or Is It Not?

A null collection is not the same thing as an initialized collection with zero elements. Image 100003 has no keywords, but the object view is mistakenly returning an empty but initialized collection. To get a true NULL instead, I can use a DECODE to test the number of keywords:

```
CREATE OR REPLACE VIEW images_v
   OF image_t
   WITH OBJECT IDENTIFIER (image_id)
AS
   SELECT i.image_id, BFILENAME('ROOTDIR', i.file_name),
          i.file_type, i.bytes,
          DECODE((SELECT COUNT(*)
                    FROM keywords k2
                  WHERE k2.image_id = i.image_id),
               0, NULL,
               CAST (MULTISET (SELECT keyword
                                 FROM keywords k
                              WHERE k.image_id = i.image_id)
                 AS keyword_tab_t))
   FROM images i;
```

In other words, if there are no keywords, return NULL; otherwise, return the CAST/MULTISET expression. From this view, "SELECT...WHERE image_id=100003" properly yields the following:

```
  IMAGE_ID KEYWORDS
---------- -------------------------------------------------------
    100003
```

But you might conclude that this amount of conceptual purity is not worth the extra I/O (or having to look at the convoluted SELECT statement).

Other things you can do with object views include the following:

Use virtual REFs
: These are pointers to virtual objects. They are discussed in detail in the section "Differences Between Object Views and Object Tables" on page 1196.

Write INSTEAD OF triggers

These allow direct manipulation of the view's contents. You can read more about this topic in the section "INSTEAD OF Triggers" on page 1193.

Object Subview

In the case where I want to treat certain images differently from others, I might want to create a subtype. In my example, I'm going to create a subtype for those images that originate from suppliers. I'd like the subtype to include a REF to a supplier object, which is defined by:

```
CREATE TYPE supplier_t AS OBJECT (
   id INTEGER,
   name VARCHAR2(400)
);
```

and by a simple object view:

```
CREATE VIEW suppliers_v
   OF supplier_t
   WITH OBJECT IDENTIFIER (id)
AS
   SELECT id, name
     FROM suppliers;
```

I will need to alter or recreate the base type to be NOT FINAL:

```
ALTER TYPE image_t NOT FINAL CASCADE;
```

so that I can create the subtype under it:

```
CREATE TYPE supplied_images_t UNDER image_t (
   supplier_ref REF supplier_t,
   supplier_rights_descriptor VARCHAR2(256)
);
```

After all this preparation, I make the subview of this subtype and declare it to be UNDER the images_v view using the following syntax:

```
CREATE VIEW supplied_images_v
        OF supplied_images_t                UNDER images_v
AS
   SELECT i.image_id, BFILENAME('ROOTDIR', i.file_name),
          i.file_type, i.bytes,
          CAST (MULTISET (SELECT keyword
                            FROM keywords k
                           WHERE k.image_id = i.image_id)
            AS keyword_tab_t),
          MAKE_REF(suppliers_v, supplier_id),
          supplier_rights_descriptor
     FROM images i
    WHERE supplier_id IS NOT NULL;
```

Oracle won't let a subview query through the superview, so this view queries the base table, adding the WHERE clause to restrict the records retrieved. Also notice that subviews don't use the WITH OBJECT IDENTIFIER clause, because they inherit the same OIDs as their superviews.

I have introduced the MAKE_REF function in this query, which Oracle provides as a way to compute a REF to a virtual object. Here, the virtual object is the supplier, as conveyed through suppliers_v. The specification of MAKE_REF is:

```
FUNCTION MAKE_REF (view, value_list) RETURN ref;
```

where:

view
: Is the object view to which you want *ref* to point

value_list
: Is a comma-separated list of column values whose datatypes must match one-for-one with the OID attributes of *view*

You should realize that MAKE_REF does not actually select through the view; it merely applies an internal Oracle algorithm to derive a REF. And, as with "real" REFs, virtual REFs may not point to actual objects.

Now I come to a surprising result. Although it seems that I have not changed the superview, images from suppliers now appear twice in the superview—that is, as duplicates:

```
SQL> SELECT COUNT(*), image_id FROM images_v GROUP BY image_id;

  COUNT(*)   IMAGE_ID
---------- ----------
         2     100001
         2     100002
         1     100003
```

The Oracle database is returning a logical UNION ALL of the query in the superview and that of the subview. This does sort of make sense; an image from a supplier is still an image. To eliminate the duplicates, add a WHERE clause on the parent that excludes records returned in the subview:

```
CREATE OR REPLACE VIEW images_v AS
   ...
   WHERE supplier_id IS NULL;
```

Object View with Inverse Relationship

To demonstrate virtual denormalization, I can create a keyword type for a view that links keywords back to the images they describe:

```
CREATE TYPE image_refs_t AS TABLE OF REF image_t;

CREATE TYPE keyword_t AS OBJECT (
   keyword VARCHAR2(45),
   image_refs image_refs_t);
```

And here is a keywords view definition:

```
CREATE OR REPLACE VIEW keywords_v
   OF keyword_t
   WITH OBJECT IDENTIFIER (keyword)
AS
   SELECT keyword, CAST(MULTISET(SELECT MAKE_REF(images_v, image_id)
                                   FROM keywords
                                  WHERE keyword = main.keyword)
                     AS image_refs_t)
     FROM (SELECT DISTINCT keyword FROM keywords) main;
```

Now, I don't promise that queries on this view will run *fast*; the query is compensating for the fact that the database lacks a reference table of keywords by doing a SELECT DISTINCT operation, and even if I weren't using any object features, that would be an expensive query.

You may correctly point out that using MAKE_REF is not mandatory here; I could have retrieved a REF by making the inner query on images_v rather than on the keywords table. In general, MAKE_REF should be faster than a lookup through an object view; on occasion, you may not have the luxury of being able to perform that lookup.

Anyway, at this point I can run such pithy queries as this one:

```
SQL> SELECT DEREF(VALUE(i)).print()
  2    FROM keywords_v v, TABLE(v.image_refs) i
  3   WHERE keyword = 'SIXTIES';

DEREF(VALUE(I)).PRINT()
------------------------------------------------------------------------------------
Id=100001; File=/files/web/60s/smiley_face.gif; keywords=HAPPY FACE, SIXTIES
Id=100002; File=/files/web/60s/peace_symbol.gif; keywords=JERRY RUBIN, PEACE,
SIXTIES
```

That is, I can show a list of all the images tagged with the keyword SIXTIES, along with their other keywords and attributes. I admit that I'm not sure how groovy that really is!

INSTEAD OF Triggers

Since Chapter 19 covered the syntax and use of INSTEAD OF triggers, I'm not going to discuss their mechanics here. Instead, I'll explore whether they are a good fit for the problem of updating object views. If your goal is to migrate toward an object approach, you may ask whether INSTEAD OF triggers are just a relational throwback that facilitates a free-for-all in which any application can perform DML.

Well, they are and they aren't.

Let's examine the arguments for both sides, and come up with some considerations so you can decide what's best for your application.

The case against

On the one hand, you could use PL/SQL programs such as packages and object methods to provide a more comprehensive technique than triggers for encapsulating DML. It is nearly trivial to take the logic from my INSTEAD OF trigger and put it into an alternate PL/SQL construct that has more universal application. In other words, if you've already standardized on some combination of packages and methods as the means of performing DML, you could keep your environment consistent without using view triggers. You might conclude that view triggers just add complexity in an increasingly confusing equation.

Moreover, even Oracle cautions against the "excessive use" of triggers because they can cause "complex interdependencies." Imagine if your INSTEAD OF triggers performed DML on tables that had other triggers, which performed DML on still other tables with triggers... it's easy to see how this could get impossible to debug.

The case for

On the other hand, you can put much of the necessary logic that you would normally put into a package or method body into an INSTEAD OF trigger instead. Doing this in combination with a proper set of privilege restrictions could protect your data just as well as, or even better than, using methods or packages.

If you happen to use a client tool such as Oracle Forms, INSTEAD OF triggers allow you to use much more of the product's default functionality when you create a Forms "block" against a view rather than a table.

Finally, if you use OCI, INSTEAD OF triggers are *required* if the object view is not inherently modifiable and you want to be able to easily "flush" cached object view data back to the server.

The bigger question

The bigger question is this: what's the best place for the SQL statements that insert, update, and delete data, especially when using object views? Assuming that you want to localize these operations on the server side, you have at least three choices: PL/SQL packages, object methods, and INSTEAD OF triggers.

Table 26-3 summarizes some of the major considerations of the three techniques. Note that this table is not meant to compare these approaches for general-purpose use, but only as they apply to localizing DML on object views.

Table 26-3. Assessment of techniques for encapsulating DML on object views

Consideration	PL/SQL package	Object method	INSTEAD OF trigger
Consistency with object-oriented approach	Potentially very good	Excellent	Potentially very good
Ability to modify when underlying schema changes	Excellent; can be easily altered and recompiled independently	Excellent in Oracle9*i* Database and later	Excellent
Risk of unexpected interactions	Low	Low	High; triggers may have unpredictable interactions with each other
Ease of use with client tool default functionality (specifically Oracle Developer)	Acceptable; programmer must add code for all client-side transactional triggers	Acceptable; programmer must add code for all client-side transactional triggers	Excellent for top-level types (however, there is no INSTEAD OF LOCK server-side trigger)
Can be turned on and off at will	No	No	Yes (by disabling and enabling the trigger)

As you can see, there is no clear "winner." Each technique has benefits that may be of more or less importance to your application.

One important point about using INSTEAD OF triggers in view hierarchies is that you will need a separate trigger for each level of the hierarchy. When you perform DML through a subview, the subview's trigger will fire; when you perform DML through the superview, the superview's trigger will fire.

And of course, you may decide that INSTEAD OF triggers make sense in combination with PL/SQL packages and/or object methods to provide layers of encapsulation. For example:

```
TRIGGER images_v_insert
INSTEAD OF INSERT ON images_v
FOR EACH ROW
BEGIN
   /* Call a packaged procedure to perform the insert. */
   manage_image.create_one(:NEW.image_id, :NEW.file_type,
      :NEW.file_name, :NEW.bytes, :NEW.keywords);
END;
```

In an ideal world, developers would select an overall architecture and design approach before hurling every Oracle feature at their application. Use a feature only if it makes sense for your design. I agree with Oracle's advice that if you do use triggers, you should use them in moderation.

Differences Between Object Views and Object Tables

In addition to the obvious difference between an object view and an object table, PL/SQL programmers should be aware of the more subtle differences. Areas of difference include the following:

- OID uniqueness
- "Storeability" of physical versus virtual REFs
- REFs to nonunique OIDs

Let's look at each difference in turn.

OID uniqueness

An object table will always have a unique object identifier, either system-generated or derived from the primary key. It is technically possible—though poor practice—to create an object table with duplicate rows, but the instances will still be unique in their object identifiers. This can happen in two different ways:

Duplicate OIDs in a single view
: An object view can easily contain multiple object instances (rows) for a given OID. You've already seen a case where the superview can accidentally contain duplicates.

Duplicate OIDs across multiple views
: If your object view is defined on an underlying object table or view *and* you use the DEFAULT keyword to specify the OID, the view will contain OIDs that match the OIDs of the underlying structure.

It seems more likely that this second possibility of duplication will be legitimate in your application, because separate views are just separate stored queries.

"Storeability" of physical versus virtual REFs

If you've built an application with physical object tables, you can store REFs to those objects persistently in other tables. A REF is a binary value that the database can use as a pointer to an object.

However, the database returns an error if you attempt to store a virtual REF—that is, a REF to a row of an object view—in an actual table. Because the reference depends on some column value(s), you will need to save the underlying column value(s) instead of the virtual reference. From one perspective, this is an irritant rather than a major obstacle. Still, it's a bit unpleasant that I cannot intermingle object tables with object views, or perform a simple transformation from an object view into an object table. I would like to be able to create an object table:

```
CREATE TABLE images2 OF image_t
   NESTED TABLE keywords STORE AS keyword_tab;
```

and then populate it from the view:

```
    INSERT INTO images2       /* invalid because images_v includes a REF */
     SELECT VALUE(i) FROM images_v i;
```

Alas, Oracle tells me *ORA-22979: cannot INSERT object view REF or user-defined REF*. Life goes on, however.

REFs to nonunique OIDs

I don't believe that it is possible to have a REF to a nonunique OID when dealing with object tables. You may want to consider what will happen if you create a REF to an object in an object view, but the view has multiple object instances for the OID in question. Granted, this is a pretty weird case; you shouldn't be creating object views with ambiguous OIDs.

In my testing, DEREFing this type of virtual REF did indeed return an object—apparently, the first one Oracle found that matched.

Maintaining Object Types and Object Views

If you work much with object types, you will learn a number of ways to get information about the types and views that you have created. Once you reach the limits of the SQL*Plus DESCRIBE command, this could involve direct queries from the Oracle data dictionary.

Data Dictionary

The data dictionary term for user-defined types (objects and collections) is simply TYPE. Object type definitions and object type bodies are both found in the USER_SOURCE view (or DBA_SOURCE, or ALL_SOURCE), just as package specifications and bodies are. Table 26-4 lists a number of helpful queries you can use.

Table 26-4. Data dictionary entries for object types

To answer the question...	Use a query such as

What object and collection types have I created?	`SELECT * FROM user_types;`
	`SELECT * FROM user_objects`
	`    WHERE object_type = 'TYPE';`

What do my object type hierarchies look like?	`SELECT RPAD(' ', 3*(LEVEL-1)) || type_name`
	`  FROM user_types`
	`WHERE typecode = 'OBJECT'`
	`  CONNECT BY PRIOR type_name = supertype_name;`

What are the attributes of type foo?	`SELECT * FROM user_type_attrs`
	`  WHERE type_name = 'FOO';`

To answer the question...	Use a query such as

What are the methods of type foo?	`SELECT * FROM user_type_methods`
	`  WHERE type_name = 'FOO';`

What are the parameters of foo's methods?	`SELECT * FROM user_method_params`
	`  WHERE type_name = 'FOO';`

What datatype is returned by foo's method called bar?	`SELECT * FROM user_method_results`
	`  WHERE type_name = 'FOO' AND method_name = 'BAR';`

What is the source code for foo, including all ALTER statements?	`SELECT text FROM user_source`
	`  WHERE name = 'FOO'`
	`    AND type = 'TYPE'   /* or 'TYPE BODY' */`
	`  ORDER BY line;`

What are the object tables that implement foo?	`SELECT table_name FROM user_object_tables`
	`  WHERE table_type = 'FOO';`

What are all the columns in an object table foo_tab, including the hidden ones?	`SELECT column_name, data_type, hidden_column,`
	`    virtual_column`
	`  FROM user_tab_cols`
	`  WHERE table_name = 'FOO_TAB';`

What columns implement foo?	`SELECT table_name, column_name`
	`  FROM user_tab_columns`
	`  WHERE data_type = 'FOO';`

What database objects depend on foo?	`SELECT name, type FROM user_dependencies`
	` WHERE referenced_name = 'FOO';`

What object views have I created, using what OIDs?	`SELECT view_name, view_type, oid_text`
	`  FROM user_views`
	` WHERE type_text IS NOT NULL;`

What does my view hierarchy look like? (Requires a temporary table in Oracle versions that can't use a subquery with CONNECT BY.)	`CREATE TABLE uvtemp AS`
	`    SELECT v.view_name, v.view_type,`
	`      v.superview_name, v1.view_type superview_type`
	`      FROM user_views v, user_views v1`
	`     WHERE v.superview_name = v1.view_name (+);`
	`SELECT RPAD(' ', 3*(LEVEL-1)) || view_name`
	`     || ' (' || view_type || ') '`
	`  FROM uvtemp`
	`  CONNECT BY PRIOR view_type = superview_type;`
	`DROP TABLE uvtemp;`

What is the query on which I defined the foo_v view?	`SET LONG 1000 -- or greater`
	`SELECT text FROM user_views`
	`  WHERE view_name = 'FOO_V';`

What columns are in view foo_v?	`SELECT column_name, data_type_mod, data_type`
	`  FROM user_tab_columns`
	` WHERE table_name = 'FOO_V';`

One potentially confusing thing Oracle has done in the data dictionary is to make object tables invisible from the USER_TABLES view. Instead, a list of object tables appears in USER_OBJECT_TABLES (as well as in USER_ALL_TABLES).

Privileges

There are a handful of system-level privileges associated with object types, summarized here:

CREATE [ANY] TYPE
: Create, alter, and drop object types and type bodies. ANY means in any schema.

CREATE [ANY] VIEW
: Create and drop views, including object views. ANY means in any schema.

ALTER ANY TYPE
: Use ALTER TYPE facilities on types in any schema.

EXECUTE ANY TYPE
: Use an object type from any schema for purposes including instantiating, executing methods, referencing, and dereferencing.

UNDER ANY TYPE
: Create a subtype in one schema under a type in any other schema.

UNDER ANY VIEW
: Create a subview in one schema under a view in any other schema.

There are three kinds of object-level privileges on object types: EXECUTE, UNDER, and DEBUG. It is also important to understand how the conventional DML privileges apply to object tables and views.

The EXECUTE privilege

If you want your associate Joe to use one of your types in his own PL/SQL programs or tables, you can grant the EXECUTE privilege to him:

```
GRANT EXECUTE on catalog_item_t TO joe;
```

If Joe has the privilege needed to create synonyms and is running Oracle9*i* Database Release 2 or later, he will be able to create a synonym:

```
CREATE SYNONYM catalog_item_t FOR scott.catalog_item_t;
```

and use it like this:

```
CREATE TABLE catalog_items OF catalog_item_t;
```

and/or like this:

```
DECLARE
  an_item catalog_item_t;
```

Joe can also use a qualified reference to the type scott.catalog_item_t.

If you refer to an object type in a stored program and grant the EXECUTE privilege on that program to a user or role, having that privilege on the type is not required, even if the program is defined with invoker rights (described in Chapter 24). Similarly, if a user has a DML privilege on a view that has an INSTEAD OF trigger for that DML operation, that user doesn't need explicit EXECUTE privileges if the trigger refers to the object type because triggers run under the definer rights model. However, the EXECUTE privilege is required by users who need to run anonymous blocks that use the object type.

The UNDER privilege

The UNDER privilege gives the grantee the right to create a subtype. You can grant it as follows:

```
GRANT UNDER ON image_t TO scott;
```

For a schema to be able to create a subtype, you must define the supertype using invoker rights (AUTHID CURRENT_USER).

This privilege can also grant the recipient the right to create a subview:

```
GRANT UNDER ON images_v TO scott;
```

The DEBUG privilege

If one of your associates is using a PL/SQL debugger to analyze code that uses a type you have created, you may want to grant her the DEBUG privilege:

```
GRANT DEBUG ON image_t TO joe;
```

Doing so will enable the grantee to look "under the covers" to examine the variables used in the type and to set breakpoints inside methods.

The DEBUG privilege also applies to object views, providing a way to debug the PL/SQL source code of INSTEAD OF triggers.

The DML privileges

For object tables, the traditional SELECT, INSERT, UPDATE, and DELETE privileges still have some meaning. A user with only the SELECT privilege on the object table may retrieve any relational columns in the base type on which the table is defined, but cannot retrieve the object as an object. That is, VALUE, TREAT, REF, and DEREF are not available. Similarly, the other DML privileges—INSERT, UPDATE, and DELETE—also apply only to the relational interpretation of the table.

In the same fashion, the grantee will not have permission to use the constructor or other object methods unless the object type owner has granted the user the EXECUTE privilege on the object type. Any columns defined on subtypes will be invisible.

Concluding Thoughts from a (Mostly) Relational Developer

Over the years, I've seen no compelling evidence that any particular programming style has a monopoly on the fundamental things we care about—fidelity to requirements, performance efficiency, developer effectiveness, and system reliability. I have seen a lot of fads, bandwagons, handwaving, and unsupported assumptions (OK, I'm probably not entirely innocent myself), and object-oriented programming seems to attract quite a lot of it. That isn't to say that OOP fails to help you solve problems; it's just that OOP is not the magic bullet that many would have you believe.

Take, for example, the principle of object-based decomposition, particularly as it tends to generate inheritance hierarchies. Because they accurately model objects as they exist in the real world, software artifacts should be easier to comprehend, faster to assemble, and more amenable to large-scale system development. Sounds fabulous, doesn't it? Well, there are a lot of different ways to decompose something drawn from the real world. It is a rare taxonomy that can exist in a simple hierarchy. My library catalog hierarchy could have been decomposed according to, say, media (print versus audio tape versus digital format...). And, although Oracle provides wonderful tools for type evolution, it may still be so painful to make sweeping changes in a type hierarchy that it will never happen. This isn't really the tool's fault; reality has a way of circumventing even the best-laid plans.

Nor is it even clear that collocating the programming logic (methods) with the data (attributes) in an abstract datatype yields any measurable benefits. It looks reasonable and makes for some great sound bites, but how exactly will coupling data and behavior be better than keeping data structures (logical and physical table design) separate from processes (procedures, functions, packages)? Many development methods acknowledge that an organization's business data structures have a much slower rate of change than do the algorithms that manipulate them. It is a design truism (even for OOP) that the more volatile elements of a system should be kept separate from the more stable elements.

There is considerable inconsistency on this last point. Rich and famous object evangelists, while emphasizing the value of bundling data with behaviors, simultaneously promote a model-view-controller approach that "separates business logic from data." Are these emperors wearing clothes, or not?

Many OOP proponents have argued for years that its greatest benefit is the reuse of software. It has been said so many times that it must be true! Unfortunately, few observers have hard evidence for this, in part because there is no consensus on what constitutes "reuse." Even object apologists began promoting higher-level "components" (whatever those may be) as a preferred unit of reuse precisely because objects proved very difficult to fit into situations beyond those for which they were designed. My sense is that OOP results in no more code reuse than well-designed subroutines.

It is certainly possible to use object-oriented approaches with PL/SQL and achieve reuse of software. Fellow author Don Bales, an accomplished object-oriented programmer, has been using PL/SQL packages as "types" for about a decade, and he says that he has been able to take entire packages (and any accompanying tables) and drop them into new software development projects without modification. He believes that the missing ingredient in most object approaches is an accurate model of the person who is actually executing the software—the user—whom Don models as an object with behaviors implemented in the actual program that is run.

Regardless of development method, some of the critical ingredients of software success are having prior expertise with similar problems, being able to employ seasoned project leaders, and incorporating a conscious software design phase. Introducing object methods or any other approach is likely to produce more positive results than an unplanned, organically grown system.

A few final thoughts on when to best use Oracle's object features:

- If you use the Oracle Call Interface (OCI), it's possible that the client-side cache and complex object retrieval will tip the scales in favor of heavy use of Oracle's object features. I'm not an OCI programmer, though, so I can't speak from experience in this regard.
- If your organization already uses object programming across the board, Oracle's object features will probably make it easier and more graceful to introduce database technology into your systems.
- Don't throw the collections baby out with the objects bathwater. Remember that you don't need to use object types or object views to take advantage of collections.
- If you've never used OOP before, these object features may seem quite complicated. I would encourage you to play around quite a bit before committing to an object approach. In particular, try out object views in conjunction with an existing system.
- I would caution against rejecting object types and object views on a vague performance argument. Oracle has made continuous progress in reducing overhead. If you perform some actual measurements, you might find OOP within acceptable bounds for your application.
- It turns out that Oracle delivers some of its built-in functionality—most notably, XML_TYPE, but also Advanced Queuing, Oracle Spatial, and Rules Manager—using object types. As we have often learned in the past, once Oracle starts using some of its own features, bugs are more quickly fixed, performance is enhanced, and usability is improved. That has happened with object types as well. More than

that, however, it means that if you are going to fully leverage the Oracle feature set, you should at least become familiar with the object type syntax and basic features.