CHAPTER 18
Packages

A *package* is a grouping or packaging together of elements of PL/SQL code into a named scope. Packages provide a structure (both logically and physically) in which you can organize your programs and other PL/SQL elements such as cursors, TYPEs, and variables. They also offer significant, unique functionality, including the ability to hide logic and data from view and to define and manipulate "global" or session-persistent data.

Why Packages?

The package is a powerful and important element of the PL/SQL language. It should be the cornerstone of any application development project. What makes packages so powerful and important? Consider their advantages. With packages, you can:

Enhance and maintain applications more easily

As more and more of the production PL/SQL code base moves into maintenance mode, the quality of PL/SQL applications will be measured as much by the ease of maintenance as by overall performance. Packages can make a substantial difference in this regard. From data encapsulation (hiding all calls to SQL statements behind a procedural interface to avoid repetition), to enumerating constants for literal or "magic" values, to grouping together logically related functionality, package-driven design and implementation lead to reduced points of failure in an application.

Improve overall application performance

By using packages, you can improve the performance of your code in a number of ways. Persistent package data can dramatically improve the response time of queries by caching static data, thereby avoiding repeated queries of the same information. Oracle's memory management also optimizes access to code defined in packages (see Chapter 24 for more details).

Shore up built-in weaknesses
: It is quite straightforward to construct a package on top of existing functionality where there are drawbacks. (Consider, for example, the UTL_FILE and DBMS_OUTPUT built-in packages, in which crucial functionality is badly or partially implemented.) You don't have to accept these weaknesses; instead, you can build your own package on top of Oracle's to correct as many of the problems as possible. For example, the *do.pkg* script I described in Chapter 17 offers a substitute for the DBMS_OUTPUT.PUT_LINE built-in that adds an overloading for the XMLType datatype. Sure, you can get some of the same effect with standalone procedures or functions, but overloading and other package features make this approach vastly preferable.

Minimize the need to recompile code
: As you will read in this chapter, a package usually consists of two pieces of code: the specification and the body. External programs (not defined in the package) can call only programs listed in the specification. If you change and recompile the package body, those external programs are not invalidated. Minimizing the need to recompile code is a critical factor in administering large bodies of application logic.

Packages are conceptually very simple. The challenge, I have found, is figuring out how to fully exploit them in an application. As a first step, I'll take a look at a simple package and see how, even in that basic code, we can reap many of the benefits of packages. Then I'll look at the special syntax used to define packages.

Before diving in, I would like to make an overall recommendation. *Always* construct your application around packages; avoid standalone (a.k.a. "schema-level") procedures and functions. Even if today you think that only one procedure is needed for a certain area of functionality, in the future you will almost certainly have two, then three, and then a dozen. At that point, you will find yourself saying, "Gee, I should really collect those together in a package!" That's fine, except that then you'll have to go back to all the invocations of those unpackaged procedures and functions and add in the package name. So start with a package and save yourself the trouble!

Demonstrating the Power of the Package

A package consists of up to two chunks of code: the *specification* (required) and the *body* (optional, but almost always present). The specification defines how a developer can use the package: which programs can be called, what cursors can be opened, and so on. The body contains the implementation of the programs (and, perhaps, cursors) listed in the specification, plus other code elements as needed.

Suppose that I need to write code to retrieve the "full name" of an employee whose name is in the form "last, first." That seems easy enough to write:

```
PROCEDURE process_employee (
   employee_id_in IN employees.employee_id%TYPE)
IS
   l_fullname VARCHAR2(100);
BEGIN
   SELECT last_name || ',' || first_name
     INTO l_fullname
     FROM employees
    WHERE employee_id = employee_id_in;
    ...
END;
```

Yet there are many problems lurking in this seemingly transparent code:

- I have hardcoded the length of the l_fullname variable. I did this because it is a *derived* value, the concatenation of two column values. I did not, therefore, have a column against which I could %TYPE the declaration. This could cause difficulties over time if the size of last_name and/or first_name columns is expanded.
- I have also hardcoded or explicitly placed in this block the *formula* (an application rule, really) for creating a full name. What's wrong with that, you wonder? What if next week I get a call from the users: "We want to see the names in first-space-last format." Yikes! Time to hunt through all my code for the last-comma-first constructions.
- Finally, this very common query will likely appear in a variety of formats in multiple places in my application. This SQL redundancy can make it very hard to maintain my logic—and optimize its performance.

What's a developer to do? I would like to be able to change the way I write my code to avoid the preceding hardcodings. To do that, I need to write these things once (one definition of a "full name" datatype, one representation of the formula, one version of the query) and then call them wherever needed. Packages to the rescue!

Consider the following package specification:

```
    /* Files on web: fullname.pkg, fullname.tst */
 1  PACKAGE employee_pkg
 2  AS
 3      SUBTYPE fullname_t IS VARCHAR2 (200);
 4
 5      FUNCTION fullname (
 6         last_in  employees.last_name%TYPE,
 7         first_in  employees.first_name%TYPE)
 8         RETURN fullname_t;
 9
10      FUNCTION fullname (
11         employee_id_in IN employees.employee_id%TYPE)
```

```
12          RETURN fullname_t;
13    END employee_pkg;
```

What I have done here is essentially *list* the different elements I want to use. The following table summarizes the important elements of the code.

Line(s)	Description
3	Declare a "new" datatype using SUBTYPE called fullname_t. It is currently defined to have a maximum of 200 characters, but that can easily be changed if needed.
5–8	Declare a function called fullname. It accepts a last name and a first name and returns the full name. Notice that the way the full name is constructed is not visible in the package specification. That's a good thing, as you will soon see.
15–18	Declare a second function, also called fullname; this version accepts a primary key for an employee and returns the full name for that employee. This repetition is an example of *overloading*, which I explored in Chapter 17.

Now, before I even show you the implementation of this package, let's rewrite the original block of code using my packaged elements (notice the use of dot notation, which is very similar to its use in the form *table.column*):

```
DECLARE
   l_name employee_pkg.fullname_t;
   employee_id_in employees.employee_id%TYPE := 1;
BEGIN
   l_name := employee_pkg.fullname (employee_id_in);
   ...
END;
```

I declare my variable using the new datatype, and then simply call the appropriate function to do all the work for me. The name formula and the SQL query have been moved from my application code to a separate "container" holding employee-specific functionality. The code is cleaner and simpler. If I need to change the formula for last name or expand the total size of the full name datatype, I can go to the package specification or body, make the changes, and recompile any affected code, and the code will automatically take on the updates.

Speaking of the package body, here is the implementation of employee_pkg:

```
 1    PACKAGE BODY employee_pkg
 2    AS
 3       FUNCTION fullname (
 4          last_in employee.last_name%TYPE,
 5          first_in employee.first_name%TYPE
 6       )
 7          RETURN fullname_t
 8       IS
 9       BEGIN
10          RETURN last_in || ', ' || first_in;
11       END;
12
13       FUNCTION fullname (employee_id_in IN employee.employee_id%TYPE)
14          RETURN fullname_t
```

```
15       IS
16          retval   fullname_t;
17       BEGIN
18          SELECT fullname (last_name, first_name) INTO retval
19            FROM employee
20           WHERE employee_id = employee_id_in;
21
22          RETURN retval;
23       EXCEPTION
24          WHEN NO_DATA_FOUND THEN RETURN NULL;
25
26          WHEN TOO_MANY_ROWS THEN errpkg.record_and_stop;
27       END;
28    END employee_pkg;
```

The following table describes the important elements of this code.

Line(s)	Description
3–11	These lines are nothing but a function wrapper around the last-comma-first formula.
13–27	Showcase a typical single-row query lookup built around an implicit query.
18	Here, though, the query calls that selfsame fullname function to return the combination of the two name components.

So now if my users call and say, "First-space-last, please!" I will not groan and work late into the night, hunting down occurrences of || ', ' ||. Instead, I will change the implementation of my employee_pkg.fullname in about five seconds flat and astound my users by announcing that they are ready to go.

And that, dear friends, gives you some sense of the beauty and power of packages.

Some Package-Related Concepts

Before diving into the details of package syntax and structure, you should be familiar with a few concepts:

Information hiding
: Information hiding is the practice of removing from view information about one's system or application. Why would a developer ever want to hide information? Couldn't it get lost? Information hiding is actually quite a valuable principle and coding technique. First of all, humans can deal with only so much complexity at a time. A number of researchers have demonstrated that remembering more than seven (plus or minus two) items in a group, for example, is challenging for the average human brain (this is known as the "human hrair limit," an expression that comes from the book *Watership Down*). By hiding unnecessary detail, you can focus on the important stuff. Second, not everyone needs to know—or should be allowed to know—all the details. I might need to call a function that calculates CEO compensation, but the formula itself could very well be confidential. In addition, if the formula changes, the code is insulated from that change.

Public and private

Closely related to information hiding is the fact that packages are built around the concepts of public and private elements. *Public* code is defined in the package specification and is available to any schema that has EXECUTE authority on the package. *Private* code, on the other hand, is defined in and visible only from within the package. External programs using the package cannot see or use private code.

When you build a package, you decide which of the package elements are public and which are private. You also can hide all the details of the package body from the view of other schemas/developers. In this way, you use the package to hide the implementation details of your programs. This is most important when you want to isolate the most volatile aspects of your application, such as platform dependencies, frequently changing data structures, and temporary workarounds.

In early stages of development you can also implement programs in the package body as "stubs," containing just enough code to allow the package to compile. This technique allows you to focus on the interfaces of your programs and the way they connect to each other.

Package specification

The package specification contains the definition or specification of all the publicly available elements in the package that may be referenced outside of the package. The specification is like one big declaration section; it does not contain any PL/SQL blocks or executable code. If a specification is well designed, a developer can learn from it everything necessary to use the package. There should never be any need to go "behind" the interface of the specification and look at the implementation, which is in the body.

Package body

The body of the package contains all the code required to implement elements defined in the package specification. The body may also contain private elements that do not appear in the specification and therefore cannot be referenced outside of the package. The body of the package resembles a standalone module's declaration section. It contains both declarations of variables and the definitions of all package modules. The package body may also contain an execution section, which is called the *initialization section* because it is run only once, to initialize the package.

Initialization

Initialization should not be a new concept for a programmer. In the context of packages, however, it takes on a specific meaning. Rather than initializing the value of a single variable, you can initialize the entire package with arbitrarily complex code. Oracle takes responsibility for making sure that the package is initialized only once per session.

Session persistence

As a database programmer, the concept of persistence should also be familiar. After all, a database is all about persistence: I insert a row into the database on Monday, fly to the Bahamas for the rest of the week, and when I return to work on the following Monday, my row is still in the database. It persisted!

Another kind of persistence is *session persistence*. This means that if I connect to the Oracle database (establish a session) and execute a program that assigns a value to a package-level variable (i.e., a variable declared in a package specification or body, outside of any program in the package), that variable is set to persist for the length of my session, and it retains its value even if the program that performed the assignment has ended.

It turns out that the package is the construct that offers support in the PL/SQL language for session-persistent data structures.

Diagramming Privacy

Let's go back to the public/private dichotomy for a moment. The distinction drawn between public and private elements in a package gives PL/SQL developers unprecedented control over their data structures and programs. A fellow named Grady Booch came up with a visual way to describe this aspect of a package (now called, naturally, the *Booch diagram*).

Take a look at Figure 18-1. Notice the two labels Inside and Outside. Outside consists of all the programs you write that are *not* a part of the package at hand (the *external programs*). Inside consists of the package body (the internals or implementation of the package).

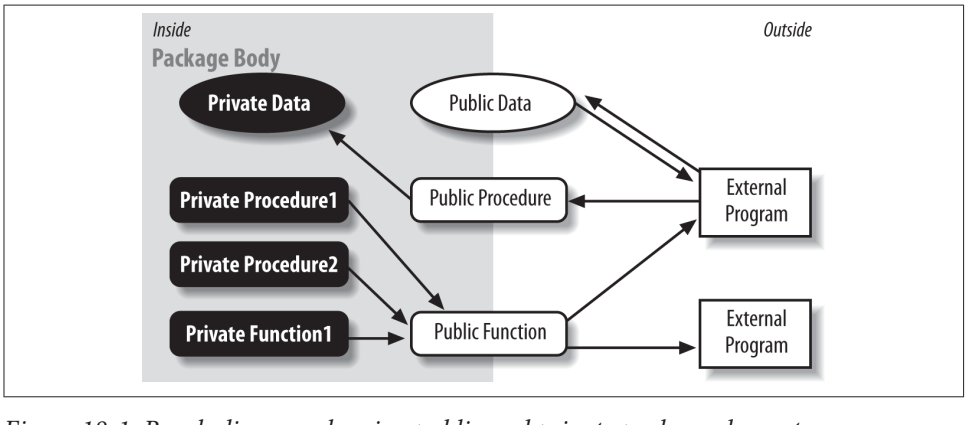

Figure 18-1. Booch diagram showing public and private package elements

Here are the conclusions we can draw from the Booch diagram:

- External programs cannot cross the boundary from outside to inside; that is, an external program may not reference or call any elements defined inside the package body. They are private and invisible outside of the package.
- Those elements defined in the package specification (labeled Public in the figure) straddle the boundary between inside and outside. These programs can be called by an external program (from the outside), can be called or referenced by a private program, and can, in turn, call or reference any other element in the package.
- Public elements of the package therefore offer the only path to the inside of the package. In this way, the package specification acts as a control mechanism for the package as a whole.
- If you find that a formerly private object (such as a module or a cursor) should instead be made public, simply add that object to the package specification and recompile. It will then be visible outside of the package.

Rules for Building Packages

The package is a deceptively simple construct. In a small amount of time, you can learn all the basic elements of package syntax and rules, but you can spend weeks (or more) uncovering all the nuances and implications of the package structure. In this section, I review the rules you need to know in order to build packages. Later in the chapter, I will take a look at the circumstances under which you will want to build packages.

To construct a package, you must build a specification and, in almost every case, a package body. You must decide which elements go into the specification and which are hidden away in the body. You also can include a block of code that the database will use to initialize the package.

The Package Specification

The specification of a package lists all the elements in that package that are available for use in applications, and provides all the information a developer needs in order to use elements defined in the package (often referred to as an *API*, or *application programming interface*). A developer should never have to look at the implementation code in a package body to figure out how to use an element in the specification.

Here are some rules to keep in mind for package specification construction:

- You can declare elements of almost any datatype, such as numbers, exceptions, types, and collections, at the package level (i.e., not within a particular procedure or function in the package). This is referred to as *package-level data*. Generally, you should avoid declaring variables in the package specification, although constants are always "safe."

You cannot declare cursor variables (variables defined from a REF CURSOR type) in a package specification (or body). Cursor variables are not allowed to *persist* at the session level (see "Working with Package Data" on page 667 for more information about package data persistence).

- You can declare almost any type of data structure in the package specification, such as a collection type, a record type, or a REF CURSOR type.
- You can declare procedures and functions in a package specification, but you can include only the header of the program (everything up to but not including the IS or AS keyword). The header must end with a semicolon.
- You can include explicit cursors in the package specification. An explicit cursor can take one of two forms: it can include the SQL query as a part of the cursor declaration, or you can "hide" the query inside the package body and provide only a RETURN clause in the cursor declaration. This topic is covered in more detail in the section "Packaged Cursors" on page 669.
- If you declare any procedures or functions in the package specification or if you declare a CURSOR without its query, then you *must* provide a package body in order to implement those code elements.
- You can include an AUTHID clause in a package specification, which determines whether any references to data objects will be resolved according to the privileges of the owner of the package (AUTHID DEFINER) or of the invoker of the package (AUTHID CURRENT_USER). See Chapter 24 for more information on this feature.
- You can include an optional package name label after the END statement of the package specification, as in:

```
END my_package;
```

Here is a simple package specification illustrating these rules:

```
    /* File on web: favorites.sql */
1   PACKAGE favorites_pkg
2      AUTHID CURRENT_USER
3   IS /* or AS */
4      -- Two constants; notice that I give understandable
5      -- names to otherwise obscure values.
6
7      c_chocolate CONSTANT PLS_INTEGER := 16;
8      c_strawberry CONSTANT PLS_INTEGER := 29;
9
10     -- A nested table TYPE declaration.
11     TYPE codes_nt IS TABLE OF INTEGER;
12
13     -- A nested table declared from the generic type.
14     my_favorites codes_nt;
15
```

```
16      -- A REF CURSOR returning favorites information.
17      TYPE fav_info_rct IS REF CURSOR RETURN favorites%ROWTYPE;
18
19      -- A procedure that accepts a list of favorites
20      -- (using a type defined above) and displays the
21      -- favorite information from that list.
22      PROCEDURE show_favorites (list_in IN codes_nt);
23
24      -- A function that returns all the information in
25      -- the favorites table about the most popular item.
26      FUNCTION most_popular RETURN fav_info_rct;
27
28  END favorites_pkg; -- End label for package
```

As you can see, a package specification is, in structure, essentially the same as a declaration section of a PL/SQL block. One difference, however, is that a package specification may *not* contain any implementation code.

The Package Body

The package body contains all the code required to implement the package specification. A package body is not always needed; see "When to Use Packages" on page 677 for examples of package specifications without bodies. A package body is required when any of the following conditions are true:

The package specification contains a cursor declaration with a RETURN clause
: You will then need to specify the SELECT statement in the package body.

The package specification contains a procedure or function declaration
: You will then need to complete the implementation of that module in the package body.

You want to execute code in the initialization section of the package
: The package specification does not support an execution section (executable statements within a BEGIN-END block); you can do this only in the body.

Structurally, a package body is very similar to a procedure definition. Here are some rules particular to package bodies:

- A package body can have declaration, execution, and exception sections. The declaration section contains the complete implementation of any cursors and programs defined in the specification, and also the definition of any private elements (not listed in the specification). The declaration section can be empty as long as there is an initialization section.
- The execution section of a package is known as the *initialization section*; this optional code is executed when the package is instantiated for a session. I discuss this topic in the following section.

- The exception section handles any exceptions raised in the initialization section. You can have an exception section at the bottom of a package body only if you have defined an initialization section.
- A package body may consist of the following combinations: only a declaration section; only an execution section; execution and exception sections; or declaration, execution, and exception sections.
- You may not include an AUTHID clause in the package body; it must go in the package specification. Anything declared in the specification may be referenced (used) within the package body.
- The same rules and restrictions for declaring package-level data structures apply to the body as to the specification—for example, you cannot declare a cursor variable.
- You can include an optional package name label after the END statement of the package body, as in:

```
END my_package;
```

Here is an implementation of the favorites_pkg body:

```
/* File on web: favorites.sql */
PACKAGE BODY favorites_pkg
IS
   -- A private variable
   g_most_popular   PLS_INTEGER := c_strawberry;

   -- Implementation of the function
   FUNCTION most_popular RETURN fav_info_rct
   IS
      retval fav_info_rct;
      null_cv fav_info_rct;
   BEGIN
      OPEN retval FOR
         SELECT *
           FROM favorites
          WHERE code = g_most_popular;
      RETURN retval;
   EXCEPTION
      WHEN NO_DATA_FOUND THEN RETURN null_cv;
   END most_popular;

  -- Implementation of the procedure
   PROCEDURE show_favorites (list_in IN codes_nt) IS
   BEGIN
      FOR indx IN list_in.FIRST .. list_in.LAST
      LOOP
         DBMS_OUTPUT.PUT_LINE (list_in (indx));
      END LOOP;
   END show_favorites;
```

```
END favorites_pkg; -- End label for package
```

See "When to Use Packages" on page 677 for other examples of package bodies.

Initializing Packages

Packages can contain data structures that persist for your entire session (this topic is covered in more detail in "Working with Package Data" on page 667). The first time your session uses a package (whether by calling a program defined in the package, reading or writing a variable, or using a locally declared variable TYPE), the database initializes that package. This involves one or all of the following:

- Instantiate any package-level data (such as a number variable or a string constant).
- Assign default values to variables and constants as specified in their declarations.
- Execute a block of code, called the *initialization section*, which is specifically designed to initialize the package, complementing the preceding steps.

Oracle executes these steps just once per session, and not until you need that information (i.e., on the "first touch" of that package).

A package may be reinitialized in a session if that package was recompiled since its last use or if the package state for your entire session was reset, as is indicated by the following error:

```
ORA-04068: existing state of packages has been discarded
```

The initialization section of a package consists of all the statements following the BEGIN statement at the end of the package declaration (and outside any procedure or function's definitions) and through to the END statement for the entire package body. Here is what an initialization section in favorites_pkg might look like:

```
/* File on web: favorites.sql */
PACKAGE BODY favorites_pkg
IS
   g_most_popular   PLS_INTEGER;

   PROCEDURE show_favorites (list_in IN codes_nt) ... END;

   FUNCTION most_popular RETURN fav_info_rct ... END;

   PROCEDURE analyze_favorites (year_in IN INTEGER) ... END;

-- Initialization section
BEGIN
   g_most_popular := c_chocolate;
```

```
    -- Use EXTRACT to get year number from SYSDATE!
    analyze_favorites (EXTRACT (YEAR FROM SYSDATE));
END favorites_pkg;
```

The initialization section is a powerful mechanism: PL/SQL automatically detects when this code should be run. You do not have to explicitly execute the statements, and you can be sure that they are run only once. Why would you use an initialization section? The following sections explore some specific reasons.

Execute complex initialization logic

You can, of course, assign default values to package data directly in the declaration statement. But this approach has a few possible problems:

- The logic required to set the default value may be quite complex and not easily invoked as a default value assignment.
- If the assignment of the default value raises an exception, that exception cannot be trapped within the package: it will instead propagate out unhandled. This issue is covered in more detail in "When initialization fails" on page 664.

Using the initialization section to initialize data offers several advantages over default value assignments. For one thing, you have the full flexibility of an execution section in which to define, structure, and document your steps, and if an exception is raised, you can handle it within the initialization section's exception section.

Cache static session information

Another great motivation for including an initialization section in your package is to cache information that is static (unchanging) throughout the duration of your session. If the data values don't change, why endure the overhead of querying or recalculating those values again and again?

In addition, if you want to make sure that the information is retrieved just once in your session, then the initialization section is an ideal, automatically managed way to get this to happen.

There is an important and typical tradeoff when working with cached package data: memory versus CPU. By caching data in package variables, you can improve the elapsed time performance of data retrieval. You accomplish this by moving the data "closer" to the user, into the program global area (PGA) of *each* session. If there are 1,000 distinct sessions, then there are 1,000 copies of the cached data. This technique decreases the CPU usage, but consumes more—sometimes *much* more—memory.

See "Cache Static Session Data" on page 684 for more details on this technique.

Avoid side effects when initializing

Avoid setting the values of global data in other packages within the initialization section (or any other values in other packages, for that matter). This precaution can prevent havoc in code execution and potential confusion for maintenance programmers. Keep the initialization section code focused on the current package. Remember that this code is executed whenever your application first tries to use a package element. You don't want your users sitting idle while the package performs some snazzy, expensive setup computations that could be parceled out to different packages or triggers in the application. Here's an example of the kind of code you should avoid:

```
PACKAGE BODY company IS
BEGIN
   /*
   || Initialization section of company_pkg updates the global
   || package data of a different package. This is a no-no!
  */
   SELECT SUM (salary)
     INTO employee_pkg.max_salary
     FROM employees;
END company;
```

If your initialization requirements seem different from those we've illustrated, you should consider alternatives to the initialization section, such as grouping your startup statements together into a procedure in the package. Give the procedure a name like init_environment; then, at the appropriate initialization point in your application, call the init_environment procedure to set up your session.

When initialization fails

There are several steps to initializing a package: declare data, assign default values, run the initialization section (if present). What happens when an error occurs, causing the failure of this initialization process? It turns out that even if a package fails to complete its initialization steps, the database marks the package as having been initialized and does *not* attempt to run the startup code again during that session. To verify this behavior, consider the following package:

```
/* File on web: valerr.pkg */
PACKAGE valerr
IS
   FUNCTION get RETURN VARCHAR2;
END valerr;

PACKAGE BODY valerr
IS
   -- A package-level, but private, global variable
   v VARCHAR2(1) := 'ABC';

   FUNCTION get RETURN VARCHAR2
   IS
```

```
   BEGIN
      RETURN v;
   END;
BEGIN
   DBMS_OUTPUT.PUT_LINE ('Before I show you v...');
EXCEPTION
  WHEN OTHERS
  THEN
    DBMS_OUTPUT.PUT_LINE ('Trapped the error!');
END valerr;
```

Suppose that I connect to SQL*Plus and try to run the valerr.get function (for the first time in that session). This is what I see:

```
SQL> EXEC DBMS_OUTPUT.PUT_LINE (valerr.get) *
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error: character string buffer too small
```

In other words, my attempt in the declaration of the v variable to assign a value of "ABC" caused a VALUE_ERROR exception. The exception section at the bottom of the package did *not* trap the error; it can trap only errors raised in the initialization section itself. And so the exception goes unhandled. Notice, however, that when I call that function a second time in my session, I do not get an error:

```
SQL> BEGIN
  2     DBMS_OUTPUT.PUT_LINE ('V is set to ' || NVL (valerr.get, 'NULL'));
  3  END;
  4  /
V is set to NULL
```

How curious! The statement "Before I show you v..." is never displayed; in fact, it is never executed. This packaged function fails the first time, but not the second or any subsequent times. Here I have one of those classic "unreproducible errors," and within the PL/SQL world, this is the typical cause of such a problem: a failure in package initialization.

These errors are very hard to track down. The best way to avoid such errors and also aid in detection is to move the assignments of default values to the initialization section, where the exception section can gracefully handle errors and report on their probable cause, as shown here:

```
PACKAGE BODY valerr
IS
   v VARCHAR2(1);
   FUNCTION get RETURN VARCHAR2 IS BEGIN ... END;
BEGIN
   v := 'ABC';

EXCEPTION
  WHEN OTHERS
  THEN
```

```
    DBMS_OUTPUT.PUT_LINE ('Error initializing valerr:');
    DBMS_OUTPUT.PUT_LINE (DBMS_UTILITY.FORMAT_ERROR_STACK);
    DBMS_OUTPUT.PUT_LINE (DBMS_UTILITY.FORMAT_ERROR_BACKTRACE);
END valerr;
```

You may even want to standardize your package design to *always* include an initialization procedure to remind developers on your team about this issue. Here's an example:

```
/* File on web: package_template.sql */
PACKAGE BODY <package_name>
IS
   -- Place private data structures below.
   -- Avoid assigning default values here.
   -- Instead, assign in the initialization procedure and
   -- verify success in the verification program.

   -- Place private programs here.

   -- Initialization section (optional)
   PROCEDURE initialize IS
   BEGIN
      NULL;
   END initialize;

   PROCEDURE verify_initialization (optional)
   -- Use this program to verify the state of the package.
   -- Were default values assigned properly? Were all
   -- necessary steps performed?
   IS
   BEGIN
      NULL;
   END verify_initialization;

   -- Place public programs here.

BEGIN
   initialize;
   verify_initialization;
END <package_name>;
/
```

Rules for Calling Packaged Elements

It doesn't really make any sense to talk about running or executing a package (after all, it is just a container for code elements). However, you will certainly want to run or reference those elements defined in a package.

A package owns its objects, just as a table owns its columns. To reference an element defined in the package specification *outside of* the package itself, you must use the same dot notation to fully specify the name of that element. Let's look at some examples.

The following package specification declares a constant, an exception, a cursor, and several modules:

```
PACKAGE pets_inc
IS
   max_pets_in_facility CONSTANT INTEGER := 120;
   pet_is_sick EXCEPTION;

   CURSOR pet_cur (pet_id_in IN pet.id%TYPE) RETURN pet%ROWTYPE;

   FUNCTION next_pet_shots (pet_id_in IN pet.id%TYPE) RETURN DATE;
   PROCEDURE set_schedule (pet_id_in IN pet.id%TYPE);

END pets_inc;
```

To reference any of these objects, I preface the object name with the package name, as follows:

```
DECLARE
   -- Base this constant on the id column of the pet table.
   c_pet CONSTANT pet.id%TYPE:= 1099;
   v_next_appointment DATE;
BEGIN
   IF pets_inc.max_pets_in_facility > 100
   THEN
      OPEN pets_inc.pet_cur (c_pet);
   ELSE
      v_next_appointment:= pets_inc.next_pet_shots (c_pet);
   END IF;
EXCEPTION
   WHEN pets_inc.pet_is_sick
   THEN
      pets_inc.set_schedule (c_pet);
END;
```

To summarize, there are two rules to follow in order to reference and use elements in a package:

- When you reference elements defined in a package specification from outside of that package (an external program), you must use dot notation in the form *package_name.element_name*.
- When you reference package elements from within the package (specification or body), you do not need to include the name of the package. PL/SQL will automatically resolve your reference within the scope of the package.

Working with Package Data

Package data consists of variables and constants that are defined at the *package level*—that is, not within a particular function or procedure in the package. The scope of the

package data is therefore not a single program, but rather the package as a whole. In the PL/SQL runtime architecture, package data structures *persist* (hold their values) for the duration of a session (rather than the duration of execution for a particular program).

If package data is declared inside the package body, then that data persists for the session but can be accessed only by elements defined in the package itself (private data).

If package data is declared inside the package specification, then that data persists for the session and is directly accessible (to both read and modify the value) by any program that has EXECUTE authority on that package (public data). Public package data is very similar to and potentially as dangerous as global variables in Oracle Forms.

If a packaged procedure opens a cursor, that cursor remains open and is available throughout the session. It is not necessary to define the cursor in each program. One module can open a cursor while another performs the fetch. Additionally, package variables can carry data across the boundaries of transactions because they are tied to the session rather than to a single transaction.

Global Within a Single Oracle Session

Package data structures act like globals within the PL/SQL environment. Remember, however, that they are accessible only within a single Oracle session or connection; package data is not shared across sessions. If you need to share data between different Oracle sessions, you can use the DBMS_PIPE package or Oracle Advanced Queuing. See the Oracle documentation or the book *Oracle Built-in Packages* for more information about these facilities.

You need to be careful about assuming that different parts of your application maintain a single Oracle database connection. There are times when a tool may establish a new connection to the database to perform an action. If this occurs, the data you have stored in a package in the first connection will not be available.

For example, suppose that an Oracle Forms application has saved values to data structures in a package. When the form calls a stored procedure, this stored procedure can access the same package-based variables and values as the form can because they share a single database connection. But now suppose that the form kicks off a report using Oracle Reports. By default, Oracle Reports uses a second connection to the database (with the same username and password) to run the report. Even if this report accesses the same package and data structures, the values in those data structures will not match those used by the form. The report is using a different database connection and a new instantiation of the package data structures.

Just as there are two types of data structures in a package (public and private), there are also two types of global package data to consider: global public data and global private data. The next three sections explore the various ways that package data can be used.

Global Public Data

Any data structure declared in the specification of a package is a global public data structure, meaning that any program outside of the package can access it. You can, for example, define a PL/SQL collection in a package specification and use it to keep a running list of all employees selected for a raise. You can also create a package of constants that are used throughout all your programs. Other developers will then reference the packaged constants instead of hardcoding the values in their programs. You are also allowed to change global public data structures, unless they are declared as CONSTANTs in the declaration statement.

Global data is the proverbial "loose cannon" of programming. It is very convenient to declare and is a great way to have all sorts of information available at any point in time. However, reliance on global data structures leads to unstructured code that is full of side effects.

Recall that the specification of a module should give you all the information you need to understand how to call and use that module. However, it is not possible to determine if a package reads and/or writes to global data structures from the package's specification. Because of this, you cannot be sure of what is happening in your application and which program changes what data.

It is always preferable to pass data as parameters in and out of modules. That way, reliance on those data structures is documented in the specification and can be accounted for by developers. On the other hand, you should create named global data structures for information that truly is global to an application, such as constants and configuration information.

You can put all such data into a single, central package, which would be easiest to manage. Note, however, that such a design also builds a "single point of recompilation" into your application: every time you make a change to the package and recompile the specification, you will cause many programs in your application to be invalidated.

Packaged Cursors

One particularly interesting type of package data is the explicit cursor, which was introduced in Chapter 14. I can declare a cursor in a package, in either the body or the specification. The state of this cursor (i.e., whether it is opened or closed and the pointer to the location in the result set) persists for the session, just like any other packaged data. This means that it is possible to open a packaged cursor in one program, fetch from it in a second, and close it in a third. This flexibility can be an advantage and also a potential problem.

Let's first look at some of the nuances of declaring packaged cursors, and then move on to how you can open, fetch, and close such cursors.

Declaring packaged cursors

If you are declaring an explicit cursor in a package specification, you have two options:

- Declare the entire cursor, including the query, in the specification. This is exactly the same as if you were declaring a cursor in a local PL/SQL block.
- Declare only the header of the cursor and do not include the query itself. In this case, the query is defined in the package body only. You have, in effect, hidden the implementation of the cursor.

If you declare only the header, then you must add a RETURN clause to the cursor definition that indicates the data elements returned by a fetch from the cursor. Of course, these data elements are actually determined by the SELECT statement for that cursor, but the SELECT statement appears only in the body, not in the specification.

The RETURN clause may be made up of either of the following datatype structures:

- A record defined from a database table using the %ROWTYPE attribute
- A record defined from a programmer-defined record type

If you declare a cursor in a package body, the syntax is the same as if you were declaring it in a local PL/SQL block.

Here is a simple package specification that shows both of these approaches:

```
    /* File on web: pkgcur.sql */
 1  PACKAGE book_info
 2  IS
 3     CURSOR byauthor_cur (
 4        author_in   IN   books.author%TYPE
 5     )
 6     IS
 7        SELECT *
 8          FROM books
 9         WHERE author = author_in;
10
11     CURSOR bytitle_cur (
12        title_filter_in  IN   books.title%TYPE
13     ) RETURN books%ROWTYPE;
14
15     TYPE author_summary_rt IS RECORD (
16        author                        books.author%TYPE,
17        total_page_count              PLS_INTEGER,
18        total_book_count              PLS_INTEGER);
19
20     CURSOR summary_cur (
21        author_in   IN   books.author%TYPE
22     ) RETURN author_summary_rt;
23  END book_info;
```

The following table describes the logic of this program.

Line(s)	Description
3–9	This is a very typical explicit cursor definition, fully defined in the package specification.
11–13	Define a cursor without a query. In this case, I am telling whoever is looking at the specification that if they open and fetch from this cursor, they will receive a single row from the books table for the specified "title filter," the implication being that wildcards are accepted in the description of the title.
15–18	Define a new record type to hold summary information for a particular author.
20–22	Declare a cursor that returns summary information (just three values) for a given author.

Let's take a look at the package body and then see what kind of code needs to be written to work with these cursors:

```
 1    PACKAGE BODY book_info
 2    IS
 3       CURSOR bytitle_cur (
 4          title_filter_in   IN   books.title%TYPE
 5       ) RETURN books%ROWTYPE
 6       IS
 7           SELECT *
 8           FROM books
 9           WHERE title LIKE UPPER (title_filter_in);
10
11       CURSOR summary_cur (
12          author_in   IN   books.author%TYPE
13       ) RETURN author_summary_rt
14       IS
15          SELECT author, SUM (page_count), COUNT (*)
16            FROM books
17           WHERE author = author_in;
18    END book_info;
```

Because I had two cursors with a RETURN clause in my book information package specification, I must finish defining those cursors in the body. The select list of the query that I now add to the header must match, in number of items and datatype, the RETURN clause in the package specification; in this case, they do. If they do not match or the RETURN clause is not specified in the body, then the package body will fail to compile with one of the following errors:

```
PLS-00323: subprogram or cursor '<cursor>' is declared in a
package specification and must be defined in the package body

PLS-00400: different number of columns between cursor SELECT
statement and return value
```

Working with packaged cursors

Now let's see how you can take advantage of packaged cursors. First of all, you do not need to learn any new syntax to open, fetch from, and close packaged cursors; you just

have to remember to prepend the package name to the name of the cursor. So if I want to get information about all the books having to do with PL/SQL, I can write a block like this:

```
DECLARE
   onebook   book_info.bytitle_cur%ROWTYPE;
BEGIN
   OPEN book_info.bytitle_cur ('%PL/SQL%');

   LOOP
      FETCH book_info.bytitle_cur INTO onebook;
      EXIT WHEN book_info.bytitle_cur%NOTFOUND;
      book_info.display (onebook);
   END LOOP;

   CLOSE book_info.bytitle_cur;
END;
```

As you can see, I can %ROWTYPE a packaged cursor and check its attributes just as I would with a locally defined explicit cursor. Nothing new there!

There are some hidden issues lurking in this code, however. Because my cursor is declared in a package specification, its scope is not bound to any given PL/SQL block. Suppose that I run this code:

```
BEGIN -- Only open...
   OPEN book_info.bytitle_cur ('%PEACE%');
END;
```

and then, in the same session, I run the anonymous block with the LOOP just shown. I will then get this error:

```
ORA-06511: PL/SQL: cursor already open
```

This happened because in my "only open" block, I neglected to close the cursor. Even though the block terminated, my packaged cursor did not close.

Given the persistence of packaged cursors, you should always keep the following rules in mind:

- Never assume that a packaged cursor is closed (and ready to be opened).
- Never assume that a packaged cursor is opened (and ready to be closed).
- Always be sure to explicitly close your packaged cursor when you are done with it. You also will need to include this logic in exception handlers; make sure the cursor is closed through all exit points in the program.

If you neglect these rules, you might well execute an application that makes certain assumptions and then pays the price in unexpected and unhandled exceptions. So the question then becomes: how best can you remember and follow these rules? My sug-

gestion is to build procedures that perform the open and close operations for you—and take all these nuances and possibilities into account.

The following package offers an example of this technique:

```
/* File on web: openclose.sql */
PACKAGE personnel
IS
   CURSOR emps_for_dept (
      department_id_in_in IN employees.department_id%TYPE)
   IS
      SELECT * FROM employees
       WHERE department_id = department_id_in;

   PROCEDURE open_emps_for_dept(
      department_id_in IN employees.department_id%TYPE,
      close_if_open IN BOOLEAN := TRUE
      );

   PROCEDURE close_emps_for_dept;

END personnel;
```

I have a packaged cursor along with procedures to open and close the cursor. So, if I want to loop through all the rows in the cursor, I would write code like this:

```
DECLARE
   one_emp personnel.emps_for_dept%ROWTYPE;
BEGIN
   personnel.open_emps_for_dept (1055);

   LOOP
      EXIT WHEN personnel.emps_for_dept%NOTFOUND;
      FETCH personnel.emps_for_dept INTO one_emp;
      ...
   END LOOP;

   personnel.close_emps_for_dept;
END;
```

I don't use explicit OPEN and CLOSE statements; instead, I call the corresponding procedures, which handle complexities related to packaged cursor persistence. I urge you to examine the *openclose.sql* file available on the book's website to study the implementation of these procedures.

You have a lot to gain by creating cursors in packages and making those cursors available to the developers on a project. Crafting precisely the data structures you need for your application is hard and careful work. These same structures—and the data in them—are used in your PL/SQL programs, almost always via a cursor. If you do not package up your cursors and provide them "free of charge and effort" to all developers, they will each write their own variations of these cursors, leading to all sorts of performance and

maintenance issues. Packaging cursors is just one example of using packages to encapsulate access to data structures, which is explored further in "When to Use Packages" on page 677.

One of the technical reviewers of this book, JT Thomas, offers the following alternative perspective:

> Rather than working with packaged cursors, you can get exactly the same effect by encapsulating logic and data presentation into views and publishing these to the developers. This allows the developers to then be responsible for properly maintaining their own cursors; the idea is that it is not possible to enforce proper maintenance given the toolset available with publicly accessible package cursors. Specifically, as far as I know, there is no way to enforce the usage of the open/close procedures, but the cursors will always remain visible to the developer directly opening/closing it; thus, this construct is still vulnerable. To make matters worse, however, the acceptance of publicly accessible packaged cursors and the open/close procedures might lull a team into a false sense of security and reliability.

Serializable Packages

As you have seen, package data by default persists for your entire session (or until the package is recompiled). This is an incredibly handy feature, but it has some drawbacks:

- Globally accessible (public *and* private) data structures persist, and that can cause undesired side effects. In particular, I can inadvertently leave packaged cursors open, causing "already open" errors in other programs.
- My programs can suck up lots of real memory (package data is managed in the user's memory area or user global area [UGA]) and then not release it if that data is stored in a package-level structure.

To help you manage the use of memory in packages, PL/SQL offers the SERIALLY_REUSABLE pragma. This pragma, which must appear in both the package specification and the body (if one exists), marks that package as *serially reusable*. For such packages, the duration of package state (the values of variables, the open status of a packaged cursor, etc.) can be reduced from a whole session to a single call of a program in the package.

To see the effects of this pragma, consider the following book_info package. I have created two separate programs, one to fill a list of books and another to show that list:

```
/* File on web: serialpkg.sql */
PACKAGE book_info
```

```
IS
   PRAGMA SERIALLY_REUSABLE;
   PROCEDURE fill_list;

   PROCEDURE show_list;
END;
```

As you can see in the following package body, that list is declared as a private, but global, associative array:

```
/* File on web: serialpkg.sql */
PACKAGE BODY book_info
IS
  PRAGMA SERIALLY_REUSABLE;

  TYPE book_list_t
  IS
     TABLE OF books%ROWTYPE
        INDEX BY PLS_INTEGER;
     my_books   book_list_t;

  PROCEDURE fill_list
  IS
  BEGIN
     FOR rec IN (SELECT *
                   FROM books
                  WHERE author LIKE '%FEUERSTEIN%')
     LOOP
        my_books (my_books.COUNT + 1) := rec;
     END LOOP;
  END fill_list;

  PROCEDURE show_list
  IS
  BEGIN
     IF my_books.COUNT = 0
     THEN
        DBMS_OUTPUT.PUT_LINE ('** No books to show you...');
     ELSE
        FOR indx IN 1 .. my_books.COUNT
        LOOP
           DBMS_OUTPUT.PUT_LINE (my_books (indx).title);
        END LOOP;
     END IF;
  END show_list;
END;
```

To see the effect of this pragma, I fill and then show the list. In my first approach, these two steps are done in the same block, so the collection is still loaded and can be displayed:

```
SQL> BEGIN
  2     DBMS_OUTPUT.PUT_LINE (
  3        'Fill and show in same block:'
```

```
  4     );
  5     book_info.fill_list;
  6     book_info.show_list;
  7  END;
  8  /

 Fill and show in same block:

   Oracle PL/SQL Programming
   Oracle PL/SQL Best Practices
   Oracle PL/SQL Built-in Packages
```

In my second attempt, I fill and show the list in two separate blocks. As a result, my collection is now empty:

```
SQL> BEGIN
   2      DBMS_OUTPUT.PUT_LINE ('Fill in first block');
   3      book_info.fill_list;
   4  END;
   5  /

Fill in first block

SQL> BEGIN
   2     DBMS_OUTPUT.PUT_LINE ('Show in second block:');
   3     book_info.show_list;
   4  END;
   5  /

Show in second block:
** No books to show you...
```

Here are some things to keep in mind for serialized packages:

- The global memory for serialized packages is allocated in the SGA, not in the user's UGA. This approach allows the package work area to be reused. Each time the package is reused, its package-level variables are initialized to their default values or to NULL, and its initialization section is reexecuted.
- The maximum number of work areas needed for a serialized package is determined by the number of concurrent users of that package. The increased use of SGA memory is offset by the decreased use of UGA or program memory. Finally, the database ages out work areas not in use if it needs to reclaim memory from the SGA for other requests.

When to Use Packages

By now, I've covered the rules, syntax, and nuances of constructing packages. Let's now return to the list of reasons you might want to use PL/SQL packages and explore them in more detail. These scenarios include:

Encapsulating (hiding) data manipulation
: Rather than have developers write SQL statements (leading to inefficient variations and maintenance nightmares), provide an interface to those SQL statements. This interface is known as a *table API* or *transaction API*.

Avoiding the hardcoding of literals
: Use a package with constants to give a name to each literal ("magic") value and avoid hardcoding it into individual (and multiple) programs. You can, of course, declare constants within procedures and functions as well. The advantage of a constant defined in a package specification is that it can be referenced outside of the package.

Improving the usability of built-in features
: Some of Oracle's own utilities, such as UTL_FILE and DBMS_OUTPUT, leave lots to be desired. Build your own package on top of Oracle's to correct as many of the problems as possible.

Grouping together logically related functionality
: If you have a dozen procedures and functions that all revolve around a particular aspect of your application, put them all into a package so that you can manage (and find) that code more easily.

Caching session-static data to improve application performance
: Take advantage of persistent package data to improve the response time of your application by caching (and not requerying) static data.

The following sections describe each of these motivations.

Encapsulate Data Access

Rather than have developers write their own SQL statements, you should provide an interface to those SQL statements. This is one of the most important motivations for building packages, yet it's only rarely employed by developers.

With this approach, PL/SQL developers as a rule will not write SQL in their applications. Instead, they will call predefined, tested, and optimized code that does all the work for them; for example, an "add" procedure (overloaded to support records) that issues the INSERT statement and follows standard error-handling rules, a function to retrieve a single row for a primary key, and a variety of cursors that handle the common requests

against the data structure (which could be a single table or a "business entity" consisting of multiple tables).

If you take this approach, developers will not necessarily need to understand how to join three or six different highly normalized tables to get the right set of data. They can just pick a cursor and leave the data analysis to someone else. They will not have to figure out what to do when they try to insert and the row already exists. The procedure has this logic inside it.

Perhaps the biggest advantage of this approach is that as your data structures change, the maintenance headaches of updating application code are both minimized and centralized. The person who is expert at working with that table or object type makes the necessary changes within that single package, and the changes are then "rolled out" more or less automatically to all programs relying on that package.

Data encapsulation is a big topic and can be very challenging to implement in a comprehensive way. You will find an example of a table encapsulation package (built around the employees table) in the *employee_tp.pks*, *employee_qp.**, *employee_cp.**, *department_tp.pks*, and *department_qp.** files on the book's website (these files were generated by the Quest CodeGen Utility, available from the Download page of ToadWorld.

Let's take a look at what kind of impact this use of packages can have on your code. The *givebonus1.sp* file on the book's website contains a procedure that gives the same bonus to each employee in the specified department, but only if that employee has been with the company for at least six months. Here are the parts of the give_bonus program that contain the SQL (see *givebonus1.sp* for the complete implementation):

```
/* File on web: givebonus1.sp */
PROCEDURE give_bonus (
   dept_in IN employees.department_id%TYPE,
   bonus_in IN NUMBER)
/*
|| Give the same bonus to each employee in the
|| specified department, but only if he has
|| been with the company for at least 6 months.
*/
IS
   l_name VARCHAR2(50);
   CURSOR by_dept_cur
   IS
      SELECT * FROM employees
      WHERE department_id = dept_in;

   fdbk INTEGER;
BEGIN
   /* Retrieve all information for the specified department. */
   SELECT department_name INTO l_name
     FROM departments
    WHERE department_id = dept_in;
```

```
  /* Make sure the department ID was valid. */
  IF l_name IS NULL
  THEN
     DBMS_OUTPUT.PUT_LINE (
        'Invalid department ID specified: ' || dept_in);
  ELSE
     /* Display the header. */
     DBMS_OUTPUT.PUT_LINE (
        'Applying Bonuses of ' || bonus_in ||
        ' to the ' || l_name || ' Department');
  END IF;
  /* For each employee in the specified department... */
  FOR rec IN by_dept_cur
  LOOP
     IF employee_rp.eligible_for_bonus (rec)
     THEN
        /* Update this column. */

        UPDATE employees
           SET salary = rec.salary + bonus_in
         WHERE employee_id = rec.employee_id;
         END IF;
  END LOOP;
END;
```

Now let's compare that to the encapsulation alternative, which you will find in its entirety in *givebonus2.sp*:

```
    /* File on web: givebonus2.sp */
 1  PROCEDURE give_bonus (
 2     dept_in    IN   employee_tp.department_id_t
 3   , bonus_in   IN   employee_tp.bonus_t
 4  )
 5  IS
 6     l_department     department_tp.department_rt;
 7     l_employees      employee_tp.employee_tc;
 8     l_rows_updated   PLS_INTEGER;
 9  BEGIN
10     l_department := department_tp.onerow (dept_in);
11     l_employees := employee_qp.ar_fk_emp_department (dept_in);
12
13     FOR l_index IN 1 .. l_employees.COUNT
14     LOOP
15        IF employee_rp.eligible_for_bonus (rec)
16        THEN
17           employee_cp.upd_onecol_pky
18              (colname_in          => 'salary'
19             , new_value_in        =>   l_employees (l_index).salary
20                                       + bonus_in
21             , employee_id_in      => l_employees (l_index).employee_id
22             , rows_out            => l_rows_updated
23              );
```

```
24         END IF;
25      END LOOP;
26
27      ... more processing with name and other elements...
28   END;
```

The following table gives an explanation of the changes made in this second version.

Line(s)	Significance
2–7	Declarations based on the underlying tables no longer use %TYPE and %ROWTYPE. Instead, a "types package" is provided that offers SUBTYPEs, which in turn rely on %TYPE and %ROWTYPE. When we take this approach, the application code no longer needs directly granted access to underlying tables (which would be unavailable in a fully encapsulated environment).
10	Replace the SELECT INTO with a call to a function that returns "one row" of information for the primary key.
11	Call a function that retrieves all the employee rows for the department ID foreign key. This function utilizes BULK COLLECT and returns a collection of records. This demonstrates how encapsulated code allows you to more easily take advantage of new features in PL/SQL.
13–25	The cursor FOR loop is replaced with a numeric FOR loop through the contents of the collection.
17–23	Use dynamic SQL to update any single column for the specified primary key.

Overall, the SQL statements have been removed from the program and have been replaced with calls to reusable procedures and functions. This optimizes the SQL in my application and allows me to write more robust code in a more productive manner.

It is by no means a trivial matter to build (or generate) such packages, and I recognize that most of you will not be willing or able to adopt a 100% encapsulated approach. You can, however, gain many of the advantages of data encapsulation without having to completely revamp your coding techniques. At a minimum, I suggest that you:

- Hide all your single-row queries behind a function interface. That way, you can make sure that error handling is performed and can choose the best implementation (implicit or explicit cursors, for example).
- Identify the tables that are most frequently and directly manipulated by developers and build layers of code around them.
- Create packaged programs to handle complex transactions. If "add a new order" involves inserting two rows, updating six others, and so on, make sure to embed this logic inside a procedure that handles the complexity. Don't rely on individual developers to figure it out (and write it more than once!).

Avoid Hardcoding Literals

Virtually any application has a variety of *magic values*—literal values that have special significance in a system. These values might be type codes or validation limits. Your users will tell you that these magic values never change. "I will *always* have only 25 line

items in my profit-and-loss," one will say. "The name of the parent company," swears another, "will *always* be ATLAS HQ." Don't take these promises at face value, and never code them into your programs. Consider the following IF statements:

```
IF footing_difference BETWEEN 1 and 100
THEN
   adjust_line_item;
END IF;

IF cust_status = 'C'
THEN
   reopen_customer;
END IF;
```

You are begging for trouble if you write code like this. You will be a much happier developer if you instead build a package of named constants as follows:

```
PACKAGE config_pkg
IS
   closed_status     CONSTANT VARCHAR2(1) := 'C';
   open_status       CONSTANT VARCHAR2(1) := 'O';
   active_status     CONSTANT VARCHAR2(1) := 'A';
   inactive_status   CONSTANT VARCHAR2(1) := 'I';

   min_difference    CONSTANT PLS_INTEGER := 1;
   max_difference    CONSTANT PLS_INTEGER := 100;

   earliest_date     CONSTANT DATE := SYSDATE;
   latest_date       CONSTANT DATE := ADD_MONTHS (SYSDATE, 120);

END config_pkg;
```

Using this package, my two preceding IF statements now become:

```
IF footing_difference
   BETWEEN config_pkg.min_difference and config_pkg.max_difference
THEN
   adjust_line_item;
END IF;

IF cust_status = config_pkg.closed_status
THEN
   reopen_customer;
END IF;
```

If any of my magic values ever change, I simply modify the assignment to the appropriate constant in the configuration package. I do not need to change a single program module. Just about every application I have reviewed (and many that I have written) mistakenly included hardcoded magic values in some of the programs. In every single case (including, of course, my own), the developer had to make repeated changes to the programs, during both development and maintenance phases. It was often a headache, and

sometimes a nightmare; I cannot emphasize strongly enough the importance of consolidating all magic values into one or more packages.

You will find another example of such a package in the *utl_file_constants.pkg* file. This package takes a different approach to that just shown. All values are hidden in the package body. The package specification consists only of functions, which return the values. This way, if and when I need to change a value, I do not have to recompile the package specification, and I avoid the need to recompile dependent programs.

Finally, if *you* get to choose the literal values that you plan to hide behind constants, you might consider using outlandish values that will further discourage any use of the literals. Suppose, for example, that you need to return a status indicator from a procedure: success or failure? Typical values for such flags include 0 and 1, S and F, etc. The problem with such values is that they are intuitive and brief, making it easy for undisciplined programmers to "cheat" and directly use the literals in their code. Consider the following:

```
PACKAGE do_stuff
IS
   c_success CONSTANT PLS_INTEGER := 0;
   c_failure CONSTANT PLS_INTEGER := 1;
   PROCEDURE big_stuff (stuff_key_in IN PLS_INTEGER, status_out OUT PLS_INTEGER);
END do_stuff;
```

With this definition, it is very likely indeed that you will encounter usages of big_stuff as follows:

```
do_stuff.big_stuff (l_stuff_key, l_status);

IF l_status = 0
THEN
   DBMS_OUTPUT.PUT_LINE ('Stuff went fine!');
END IF;
```

If, on the other hand, my package specification looks like this:

```
PACKAGE do_stuff
IS
   /* Entirely arbitrary literal values! */
   c_success CONSTANT PLS_INTEGER := -90845367;
   c_failure CONSTANT PLS_INTEGER := 55338292;
   PROCEDURE big_stuff (stuff_key_in IN PLS_INTEGER, status_out OUT PLS_INTEGER);
END do_stuff;
```

I predict that you will *never* see code like this:

```
do_stuff.big_staff (l_stuff_key, l_status);
IF l_status = -90845367
THEN
   DBMS_OUTPUT.PUT_LINE ('Stuff went fine!');
END IF;
```

It would be too embarrassing to write such code.

Improve Usability of Built-in Features

Some of Oracle's own supplied packages, such as UTL_FILE and DBMS_OUTPUT, either contain very bothersome bugs or reflect design choices that are undesirable. We all have our pet peeves, and not just about how Oracle builds utilities for us. What about that "ace" consultant who blew into town last year? Are you still trying to deal with the code mess he left behind? Maybe you can't *replace* any of this stuff, but you can certainly consider building your own packages on top of others' packages, poorly designed data structures, etc., to correct as many of the problems as possible.

Rather than fill up the pages of this book with examples, I've listed the filenames of a number of packages available on the book's website as companion code to this text. These demonstrate this use of packages and also offer some useful utilities. I suggest that you look through all the **.pkg* files on the site for other code you might find handy in your applications, but here are a few good starting points:

filepath.pkg
: Adds support for a path to UTL_FILE. This allows you to search through multiple specified directories to find the desired file.

xfile.pkg and JFile.java (alternatively, sf_file.pks/pkb and sf_file.java)
: Extend the reach of UTL_FILE by providing a package that is built on top of a Java class that performs many tasks unsupported by UTL_FILE.The xfile ("eXtra File stuff") package also offers 100% support of the UTL_FILE interface. This means that you can do a global search and replace of "UTL_FILE" with "xfile" in your code, and it will continue to work as it did before!

sf_out.pks/pkb, bpl.sp, do.pkg
: Substitutes for the "print line" functionality of DBMS_OUTPUT that help you avoid the nuisances of its design drawbacks (inability to display Booleans or—prior to Oracle Database 10*g*—strings longer than 255 bytes, for instance).

Group Together Logically Related Functionality

If you have a dozen procedures and functions that all revolve around a particular feature or aspect of your application, put them into a package so that you can manage (and find) that code more easily. This is most important when coding the business rules for your application. When implementing business rules, follow these important guidelines:

- Don't hardcode them (usually repeatedly) into individual application components.
- Don't scatter them across many different standalone, hard-to-manage programs.

Before you start building an application, construct a series of packages that encapsulate all of its rules. Sometimes these rules are part of a larger package, such as a table encapsulation package. In other cases, you might establish a package that contains nothing *but* the key rules. Here is one example:

```
/* File on web: custrules.pkg */
PACKAGE customer_rules
IS
    FUNCTION min_balance RETURN PLS_INTEGER;

    FUNCTION eligible_for_discount
       (customer_in IN customer%ROWTYPE)
       RETURN BOOLEAN;

    FUNCTION eligible_for_discount
       (customer_id_in IN customer.customer_id%TYPE)
       RETURN BOOLEAN;

END customer_rules;
```

The "eligible for discount" function is hidden away in the package so that it can be easily managed. I also use overloading to offer two different interfaces to the formula: one that accepts a primary key and establishes eligibility for that customer in the database, and a second that applies its logic to customer information already loaded into a %ROWTYPE record. Why did I do this? Because if a person has already queried the customer information from the database, she can use the %ROWTYPE overloading and avoid a second query.

Of course, not all "logically related functionality" has to do with business rules. For example, suppose I need to add to the built-in string manipulation functions of PL/SQL. Rather than create 12 different standalone functions, I will create a "string enhancements" package and put all of the functions there. Then I and others know where to go to access that functionality.

Cache Static Session Data

Take advantage of persistent package data to improve the response time of your application by caching (and not requerying) static data. You can do this at a number of different levels. For each of the following items, I've listed a few helpful code examples available on the book's website:

- Cache a single value, such as the name of the current user (returned by the USER function). Examples: *thisuser.pkg* and *thisuser.tst*.
- Cache a single row or set of information, such as the configuration information for a given user. Examples: *init.pkg* and *init.tst*.

- Cache a whole list of values, such as the contents of a static reference code lookup table. Examples: *emplu.pkg* (employee lookup) and *emplu.tst*.
- Use the *.tst* files to compare cached and noncached performance.

Package-based caching is just one type of caching available to PL/SQL developers. See Chapter 21 for a more detailed presentation of all of your caching options.

If you decide to take advantage of package-based caching, remember that this data is cached separately for each session that references the package (in the program global area). This means that if your cache of a row in a table consumes 2 MB and you have 1,000 simultaneously connected sessions, then you have just used up 2 GB of memory in your system—in addition to all the other memory consumed by the database.

Packages and Object Types

Packages are containers that allow you to group together data and code elements. Object types are containers that allow you to group together data and code elements. Do you need both? Do object types supersede packages, especially now that Oracle has added support for inheritance? When should you use a package and when should you use an object type? All very interesting and pertinent questions.

It is true that packages and object types share some features:

- Each can contain one or more programs and data structures.
- Each can (and usually does) consist of both a specification and a body.

There are, however, key differences between the two, including:

- An object type is a *template* for data; you can instantiate multiple object type instances (a.k.a. "objects") from that template. Each one of those instances has associated with it all of the attributes (data) and methods (procedures and functions) from the template. These instances can be stored in the database. A package, on the other hand, is a one-off structure and, in a sense, a static object type: you cannot declare instances of it.
- Object types offer inheritance. That means that I can declare an object type to be "under" another type, and it *inherits* all the attributes and methods of that supertype. There is no concept of hierarchy or inheritance in packages. See Chapter 26 for lots more information about this.

- With packages, you can create private, hidden data and programs. This is not supported in object types, in which everything is publicly declared and accessible (although you can still hide the implementation of methods in the object type body).

So when should you use object types and when should you use packages? First of all, very few people use object types, and even fewer attempt to take advantage of Oracle's "object relational" model. For them, packages will remain the core building blocks of their PL/SQL-based applications.

If you do plan to exploit object types, I recommend that you consider putting much of your complex code into packages that are then called by methods in the object type. You then have more flexibility in designing the code that implements your object types, and you can share that code with other elements of your application.