CHAPTER 20
Managing PL/SQL Code

Writing the code for an application is just one step toward putting that application into production and then maintaining the code base. It is not possible within the scope of this book to fully address the entire life cycle of application design, development, and deployment. I do have room, however, to offer some ideas and advice about the following topics:

Managing and analyzing code in the database
: When you compile PL/SQL program units, the source code is loaded into the data dictionary in a variety of forms (the text of the code, dependency relationships, parameter information, etc.). You can then use SQL statements to retrieve information about those program units, making it easier to understand and manage your application code.

Using compile-time warnings
: Starting with Oracle Database 10*g*, Oracle has added significant new and transparent capabilities to the PL/SQL compiler. The compiler will now automatically optimize your code, often resulting in substantial improvements in performance. In addition, the compiler will provide warnings about your code that will help you improve its readability, performance, and/or functionality.

Managing dependencies and recompiling code
: Oracle automatically manages dependencies between database objects. It is very important to understand how these dependencies work, how to minimize invalidation of program units, and how best to recompile program units.

Testing PL/SQL programs
: Testing our programs to verify correctness is central to writing and deploying successful applications. You can strengthen your own homegrown tests with automated testing frameworks, both open source and commercial.

Tracing PL/SQL code
: Most of the applications we write are very complex—so complex, in fact, that we can get lost inside our own code. Code instrumentation (which means, mostly, inserting trace calls in your programs) can provide the additional information needed to make sense of what we write.

Debugging PL/SQL programs
: Many development tools now offer graphical debuggers based on Oracle's DBMS_DEBUG API. These provide the most powerful way to debug programs, but they are still just a small part of the overall debugging process. In this chapter I also discuss program tracing and explore some of the techniques and (dare I say) philosophical approaches you should utilize to debug effectively.

Protecting stored code
: Oracle offers a way to "wrap" source code so that confidential and proprietary information can be hidden from prying eyes (unless, of course, those eyes download an "unwrapper" utility). This feature is most useful to vendors who sell applications based on PL/SQL stored code.

Using edition-based redefinition
: New to Oracle Database 11*g* Release 2, this feature allows database administrators to "hot patch" PL/SQL application code. Prior to this release, if you needed to recompile a production package with *state* (package-level variables), you would risk the dreaded ORA-04068 error unless you scheduled downtime for the application —and that would require you to kick the users off the system. Now, new versions of code and underlying database tables can be compiled into the application while it is being used, reducing the downtime for Oracle applications. This is primarily a DBA feature, but it is covered lightly in this chapter.

Managing Code in the Database

When you compile a PL/SQL program unit, its source code is stored in the database itself. Information about that program unit is then made available through a set of data dictionary views. This approach to compiling and storing code information offers two tremendous advantages:

Information about that code is available via the SQL language
: You can write queries and even entire PL/SQL programs that read the contents of these data dictionary views, obtain lots of fascinating and useful information about your code, and even change the state of your application code.

The database manages dependencies between your stored objects
: In the world of PL/SQL, you don't have to "make" an executable that is then run by users. There is no "build process" for PL/SQL. The database takes care of all such

housekeeping details for you, letting you focus more productively on implementing business logic.

The following sections introduce you to some of the most commonly accessed sources of information in the data dictionary.

Overview of Data Dictionary Views

The Oracle data dictionary is a jungle—lushly full of incredible information, but often with less than clear pathways to your destination. There are hundreds of views built on hundreds of tables, many complex interrelationships, special codes, and, all too often, nonoptimized view definitions. A subset of this multitude is particularly handy to PL/SQL developers; I will take a closer look at the key views in a moment. First, it is important to know that there are three types or levels of data dictionary views:

*USER_**
: Views that show information about the database objects owned by the currently connected schema.

*ALL_**
: Views that show information about all of the database objects to which the currently connected schema has access (either because it owns them or because it has been granted access to them). Generally they have the same columns as the corresponding USER views, with the addition of an OWNER column in the ALL views.

*DBA_**
: Views that show information about all the objects in the database. Generally they have the same columns as the corresponding ALL views. Exceptions include the v$, gx$, and x$ views.

I'll work with the USER views in this chapter; you can easily modify any scripts and techniques to work with the ALL views by adding an OWNER column to your logic. The following are some views a PL/SQL developer is most likely to find useful:

USER_ARGUMENTS
: The arguments (parameters) in all the procedures and functions in your schema.

USER_DEPENDENCIES
: The dependencies to and from objects you own. This view is mostly used by Oracle to mark objects INVALID when necessary, and also by IDEs to display the dependency information in their object browsers. Note: for full dependency analysis, you will want to use ALL_DEPENDENCIES, in case your program unit is called by a program unit owned by a different schema.

USER_ERRORS
: The current set of compilation errors for all stored objects (including triggers) you own. This view is accessed by the SHOW ERRORS SQL*Plus command, described in Chapter 2. You can, however, write your own queries against it as well.

USER_IDENTIFIERS (PL/Scope)
: Introduced in Oracle Database 11*g* and populated by the PL/Scope compiler utility. Once populated, this view provides you with information about all the identifiers (program names, variables, etc.) in your code base. This is a very powerful code analysis tool.

USER_OBJECTS
: The objects you own (excepting database links). You can, for instance, use this view to see if an object is marked INVALID, find all the packages that have "EMP" in their names, etc.

USER_OBJECT_SIZE
: The size of the objects you own. Actually, this view will show you the source, parsed, and compile sizes for your code. Although it is used mainly by the compiler and the runtime engine, you can use it to identify the large programs in your environment, good candidates for pinning into the SGA.

USER_PLSQL_OBJECT_SETTINGS
: Information about the characteristics of a PL/SQL object that can be modified through the ALTER and SET DDL commands, such as the optimization level, debug settings, and more. Introduced in Oracle Database 10*g*.

USER_PROCEDURES
: Information about stored program units (*not* just procedures, as the name would imply), such as the AUTHID setting, whether the program was defined as DETERMINISTIC, and so on.

USER_SOURCE
: The text source code for all objects you own (in Oracle9*i* Database and above, including database triggers and Java source). This is a very handy view, because you can run all sorts of analyses of the source code against it using SQL and, in particular, Oracle Text.

USER_STORED_SETTINGS
: PL/SQL compiler flags. Use this view to discover which programs have been compiled using native compilation.

USER_TRIGGERS and USER_TRIGGER_COLS
: The database triggers you own (including source code and description of triggering event) and any columns identified with the triggers. You can write programs against this view to enable or disable triggers for a particular table.

You can view the structures of each of these views either with a DESCRIBE command in SQL*Plus or by referring to the appropriate Oracle documentation. The following sections provide some examples of the ways you can use these views.

Display Information About Stored Objects

The USER_OBJECTS view contains the following key information about an object:

OBJECT_NAME
: Name of the object

OBJECT_TYPE
: Type of the object (e.g., PACKAGE, FUNCTION, TRIGGER)

STATUS
: Status of the object: VALID or INVALID

LAST_DDL_TIME
: Timestamp indicating the last time that this object was changed

The following SQL*Plus script displays the status of PL/SQL code objects:

```
/* File on web: psobj.sql */
SELECT object_type, object_name, status
  FROM user_objects
 WHERE object_type IN (
    'PACKAGE', 'PACKAGE BODY', 'FUNCTION', 'PROCEDURE',
    'TYPE', 'TYPE BODY', 'TRIGGER')
 ORDER BY object_type, status, object_name
```

The output from this script will be similar to the following:

```
OBJECT_TYPE          OBJECT_NAME                    STATUS
-------------------- ------------------------------ ----------
FUNCTION             DEVELOP_ANALYSIS               INVALID
                     NUMBER_OF_ATOMICS              INVALID

PACKAGE              CONFIG_PKG                     VALID
                     EXCHDLR_PKG                    VALID
```

Notice that two of my modules are marked as INVALID. See the section "Recompiling Invalid Program Units" on page 773 for more details on the significance of this setting and how you can change it to VALID.

Display and Search Source Code

You should always maintain the source code of your programs in text files (or via a development tool specifically designed to store and manage PL/SQL code outside of the database). When you store these programs in the database, however, you can take ad-

vantage of SQL to analyze your source code across all modules, which may not be a straightforward task with your text editor.

The USER_SOURCE view contains all of the source code for objects owned by the current user. The structure of USER_SOURCE is as follows:

```
Name                           Null?    Type
------------------------------ -------- ----
NAME                           NOT NULL VARCHAR2(30)
TYPE                                    VARCHAR2(12)
LINE                           NOT NULL NUMBER
TEXT                                    VARCHAR2(4000)
```

where:

NAME
: Is the name of the object

TYPE
: Is the type of the object (ranging from PL/SQL program units to Java source to trigger source)

LINE
: Is the line number

TEXT
: Is the text of the source code

USER_SOURCE is a very valuable resource for developers. With the right kind of queries, you can do things like:

- Display source code for a given line number.
- Validate coding standards.
- Identify possible bugs or weaknesses in your source code.
- Look for programming constructs not identifiable from other views.

Suppose, for example, that I have set as a rule that individual developers should never hardcode one of those application-specific error numbers between –20,999 and –20,000 (such hardcodings can lead to conflicting usages and lots of confusion). I can't stop a developer from writing code like this:

```
RAISE_APPLICATION_ERROR (-20306, 'Balance too low');
```

but I can create a package that allows me to identify all the programs that have such a line in them. I call it my "validate standards" package; it is very simple, and its main procedure looks like this:

```
/* Files on web: valstd.* */
PROCEDURE progwith (str IN VARCHAR2)
IS
```

```
   TYPE info_rt IS RECORD (
      NAME   user_source.NAME%TYPE
    , text   user_source.text%TYPE
   );
   TYPE info_aat IS TABLE OF info_rt
      INDEX BY PLS_INTEGER;

   info_aa   info_aat;
BEGIN
   SELECT NAME || '-' || line
        , text
   BULK COLLECT INTO info_aa
     FROM user_source
    WHERE UPPER (text) LIKE '%' || UPPER (str) || '%'
      AND NAME <> 'VALSTD'
      AND NAME <> 'ERRNUMS';

   disp_header ('Checking for presence of "' || str || '"');

   FOR indx IN info_aa.FIRST .. info_aa.LAST
   LOOP
      pl (info_aa (indx).NAME, info_aa (indx).text);
   END LOOP;
END progwith;
```

Once this package is compiled into my schema, I can check for usages of –20,*NNN* numbers with this command:

```
SQL> EXEC valstd.progwith ('-20')
==================
VALIDATE STANDARDS
==================
Checking for presence of "-20"
CHECK_BALANCE - RAISE_APPLICATION_ERROR (-20306, 'Balance too low');
MY_SESSION -    PRAGMA EXCEPTION_INIT(dblink_not_open,-2081);
VSESSTAT - CREATE DATE    : 1999-07-20
```

Notice that the *–2081* and *1999-07-20* "hits" in my output are not really a problem; they show up only because I didn't define my filter narrowly enough.

This is a fairly crude analytical tool, but you could certainly make it more sophisticated. You could also have it generate HTML that is then posted on your intranet. You could then run the valstd scripts every Sunday night through a DBMS_JOB-submitted job, and each Monday morning developers could check the intranet for feedback on any fixes needed in their code.

Use Program Size to Determine Pinning Requirements

The USER_OBJECT_SIZE view gives you the following information about the size of the programs stored in the database:

SOURCE_SIZE
: Size of the source in bytes. This code must be in memory during compilation (including dynamic/automatic recompilation).

PARSED_SIZE
: Size of the parsed form of the object in bytes. This representation must be in memory when any object that references this object is compiled.

CODE_SIZE
: Code size in bytes. This code must be in memory when the object is executed.

Here is a query that allows you to show code objects that are larger than a given size. You might want to run this query to identify the programs that you will want to pin into the database using DBMS_SHARED_POOL (see Chapter 24 for more information on this package) in order to minimize the swapping of code in the SGA:

```
/* File on web: pssize.sql */
SELECT name, type, source_size, parsed_size, code_size
  FROM user_object_size
 WHERE code_size > &&1 * 1024
 ORDER BY code_size DESC
```

Obtain Properties of Stored Code

The USER_PLSQL_OBJECT_SETTINGS view (introduced in Oracle Database 10*g*) provides information about the following compiler settings of a stored PL/SQL object:

PLSQL_OPTIMIZE_LEVEL
: Optimization level that was used to compile the object

PLSQL_CODE_TYPE
: Compilation mode for the object

PLSQL_DEBUG
: Whether or not the object was compiled for debugging

PLSQL_WARNINGS
: Compiler warning settings that were used to compile the object

NLS_LENGTH_SEMANTICS
: NLS length semantics that were used to compile the object

Possible uses for this view include:

- Identifying any programs that are not taking full advantage of the optimizing compiler (an optimization level of 1 or 0):

```
/* File on web: low_optimization_level.sql */
SELECT owner, name
  FROM user_plsql_object_settings
 WHERE plsql_optimize_level IN (1,0);
```

- Determining if any stored programs have disabled compile-time warnings:

```
/* File on web: disable_warnings.sql */
SELECT NAME, plsql_warnings
  FROM user_plsql_object_settings
 WHERE plsql_warnings LIKE '%DISABLE%';
```

The USER_PROCEDURES view lists all functions and procedures, along with associated properties, including whether a function is pipelined, parallel enabled, or aggregate. USER_PROCEDURES will also show you the AUTHID setting for a program (DEFINER or CURRENT_USER). This can be very helpful if you need to see quickly which programs in a package or group of packages use invoker rights or definer rights. Here is an example of such a query:

```
/* File on web: show_authid.sql */
SELECT   AUTHID
       , p.object_name program_name
       , procedure_name subprogram_name
    FROM user_procedures p, user_objects o
   WHERE p.object_name = o.object_name
     AND p.object_name LIKE '<package or program name criteria>'
ORDER BY AUTHID, procedure_name;
```

Analyze and Modify Trigger State Through Views

Query the trigger-related views (USER_TRIGGERS, USER_TRIG_COLUMNS) to do any of the following:

- Enable or disable all triggers for a given table. Rather than writing this code manually, you can execute the appropriate DDL statements from within a PL/SQL program. See the section "Maintaining Triggers" on page 743 in Chapter 19 for an example of such a program.
- Identify triggers that execute only when certain columns are changed, but do not have a WHEN clause. A best practice for triggers is to include a WHEN clause to make sure that the specified columns actually *have* changed values (rather than simply writing the same value over itself).

Here is a query you can use to identify potentially problematic triggers lacking a WHEN clause:

```
/* File on web: nowhen_trigger.sql */
SELECT *
  FROM user_triggers tr
 WHERE when_clause IS NULL AND
       EXISTS (SELECT 'x'
                 FROM user_trigger_cols
                WHERE trigger_owner = USER
                AND trigger_name = tr.trigger_name);
```

Analyze Argument Information

A *very* useful view for programmers is USER_ARGUMENTS. It contains information about each of the arguments of each of the stored programs in your schema. It offers, simultaneously, a wealth of nicely parsed information about arguments and a bewildering structure that is very hard to work with.

Here is a simple SQL*Plus script to dump the contents of USER_ARGUMENTS for all the programs in the specified package:

```
/* File on web: desctest.sql */
SELECT object_name, argument_name, overload
     , POSITION, SEQUENCE, data_level, data_type
  FROM user_arguments
 WHERE package_name = UPPER ('&&1');
```

A more elaborate PL/SQL-based program for displaying the contents of USER_ARGUMENTS may be found in the *show_all_arguments.sp* file on the book's website.

You can also write more specific queries against USER_ARGUMENTS to identify possible quality issues with your code base. For example, Oracle recommends that you stay away from the LONG datatype and instead use LOBs. In addition, the fixed-length CHAR datatype can cause logic problems; you are much better off sticking with VARCHAR2. Here is a query that uncovers the usage of these types in argument definitions:

```
/* File on web: long_or_char.sql */
SELECT object_name, argument_name, overload
     , POSITION, SEQUENCE, data_level, data_type
  FROM user_arguments
 WHERE data_type IN ('LONG','CHAR');
```

You can even use USER_ARGUMENTS to deduce information about a package's program units that is otherwise not easily obtainable. Suppose that I want to get a list of all the procedures and functions defined in a package specification. You will say: "No problem! Just query the USER_PROCEDURES view." And that would be a fine answer, except that it turns out that USER_PROCEDURES doesn't tell you whether a program is a function or a procedure (in fact, it can be *both*, depending on how the program is overloaded!).

You might instead want to turn to USER_ARGUMENTS. It does, indeed, contain that information, but it is far less than obvious. To determine whether a program is a function or a procedure, you must check to see if there is a row in USER_ARGUMENTS for that package-program combination that has a POSITION of 0. That is the value Oracle uses to store the RETURN "argument" of a function. If it is not present, then the program must be a procedure.

The following function uses this logic to return a string that indicates the program type (if it is overloaded with both types, the function returns "FUNCTION, PROCEDURE"). Note that the list_to_string function used in the main body is provided in the file:

```
/* File on web: program_type.sf */
FUNCTION program_type (owner_in     IN VARCHAR2,
                                        package_in   IN VARCHAR2,
                                        program_in   IN VARCHAR2)
   RETURN VARCHAR2
IS
   c_function_pos   CONSTANT PLS_INTEGER := 0;

   TYPE type_aat IS TABLE OF all_objects.object_type%TYPE
      INDEX BY PLS_INTEGER;

   l_types                   type_aat;
   retval                    VARCHAR2 (32767);
BEGIN
     SELECT CASE MIN (position)
               WHEN c_function_pos THEN 'FUNCTION'
               ELSE 'PROCEDURE'
            END
       BULK COLLECT INTO l_types
       FROM all_arguments
      WHERE     owner = owner_in
            AND package_name = package_in
            AND object_name = program_in
   GROUP BY overload;

   IF l_types.COUNT > 0
   THEN
      retval := list_to_string (l_types, ',', distinct_in => TRUE);
   END IF;

   RETURN retval;
END program_type;
```

Finally, you should also know that the built-in package DBMS_DESCRIBE provides a PL/SQL API to provide much of the same information as USER_ARGUMENTS. There are differences, however, in the way these two elements handle datatypes.

Analyze Identifier Usage (Oracle Database 11g's PL/Scope)

It doesn't take long for the volume and complexity of a code base to present serious maintenance and evolutionary challenges. I might need, for example, to implement a new feature in some portion of an existing program. How can I be sure that I understand the impact of this feature *and* make all the necessary changes? Prior to Oracle Database 11*g*, the tools I could use to perform impact analysis were largely limited to queries against ALL_DEPENDENCIES and ALL_SOURCE. Now, with PL/Scope, I can perform much more detailed and useful analyses.

PL/Scope collects data about identifiers in PL/SQL source code when it compiles your code, and makes it available in static data dictionary views. This collected data, accessible through USER_IDENTIFIERS, includes very detailed information about the types and

usages (including declarations, references, assignments, etc.) of each identifier, plus information about the location of each usage in the source code.

Here is the description of the USER_IDENTIFIERS view:

```
Name                  Null?    Type
--------------------- -------- ---------------
NAME                           VARCHAR2(128)
SIGNATURE                      VARCHAR2(32)
TYPE                           VARCHAR2(18)
OBJECT_NAME           NOT NULL VARCHAR2(128)
OBJECT_TYPE                    VARCHAR2(13)
USAGE                          VARCHAR2(11)
USAGE_ID                       NUMBER
LINE                           NUMBER
COL                            NUMBER
USAGE_CONTEXT_ID               NUMBER
```

You can write queries against USER_IDENTIFIERS to mine your code for all sorts of information, including violations of naming conventions. PL/SQL editors such as Toad are likely to start offering user interfaces to PL/Scope soon, making it easy to analyze your code. Until that happens, you will need to construct your own queries (or use those produced and made available by others).

To use PL/Scope, you must first ask the PL/SQL compiler to analyze the identifiers of your program when it is compiled. You do this by changing the value of the PLSCOPE_SETTINGS compilation parameter. You can do this for a session or even an individual program unit, as shown here:

```
ALTER SESSION SET plscope_settings='IDENTIFIERS:ALL'
```

You can see the value of PLSCOPE_SETTINGS for any particular program unit with a query against USER_PLSQL_OBJECT_SETTINGS.

Once PL/Scope has been enabled, whenever you compile a program unit, Oracle will populate the data dictionary with detailed information about how each identifier in your program (variables, types, programs, etc.) is used.

Let's take a look at a few examples of using PL/Scope. Suppose I create the following package specification and procedure, with PL/Scope enabled:

```
/* File on web: 11g_plscope.sql */
ALTER SESSION SET plscope_settings='IDENTIFIERS:ALL'
/

CREATE OR REPLACE PACKAGE plscope_pkg
IS
   FUNCTION plscope_func (plscope_fp1 NUMBER)
      RETURN NUMBER;

   PROCEDURE plscope_proc (plscope_pp1 VARCHAR2);
END plscope_pkg;
```

```
/

CREATE OR REPLACE PROCEDURE plscope_proc1
IS
   plscope_var1   NUMBER := 0;
BEGIN
   plscope_pkg.plscope_proc (TO_CHAR (plscope_var1));
   DBMS_OUTPUT.put_line (SYSDATE);
   plscope_var1 := 1;
END plscope_proc1;
/
```

I can verify PL/Scope settings as follows:

```
SELECT name, plscope_settings
  FROM user_plsql_object_settings
 WHERE name LIKE 'PLSCOPE%'

NAME                           PLSCOPE_SETTINGS
------------------------------ ----------------
PLSCOPE_PKG                    IDENTIFIERS:ALL
PLSCOPE_PROC1                  IDENTIFIERS:ALL
```

Let's determine what has been *declared* in the process of compiling these two program units:

```
SELECT name, TYPE
  FROM user_identifiers
 WHERE name LIKE 'PLSCOPE%' AND usage = 'DECLARATION'ORDER BY type, usage_id

NAME             TYPE
---------------  -----------
PLSCOPE_FP1      FORMAL IN
PLSCOPE_PP1      FORMAL IN
PLSCOPE_FUNC     FUNCTION
PLSCOPE_PKG      PACKAGE
PLSCOPE_PROC1    PROCEDURE
PLSCOPE_PROC     PROCEDURE
PLSCOPE_VAR1     VARIABLE
```

Now I'll discover all locally declared variables:

```
SELECT a.name variable_name, b.name context_name, a.signature
  FROM user_identifiers a, user_identifiers b
 WHERE     a.usage_context_id = b.usage_id
       AND a.TYPE = 'VARIABLE'
       AND a.usage = 'DECLARATION'
       AND a.object_name = 'PLSCOPE_PROC1'
       AND a.object_name = b.object_nameORDER BY a.object_type, a.usage_id

VARIABLE_NAME  CONTEXT_NAME  SIGNATURE
-------------- ------------- --------------------------------
PLSCOPE_VAR1   PLSCOPE_PROC1 401F008A81C7DCF48AD7B2552BF4E684
```

Impressive, yet PL/Scope can do so much more. I would like to know all the locations in my program unit in which this variable is used, as well as the type of usage:

```
SELECT usage, usage_id, object_name, object_type
  FROM user_identifiers sig
     , (SELECT a.signature
         FROM user_identifiers a
         WHERE     a.TYPE = 'VARIABLE'
               AND a.usage = 'DECLARATION'
               AND a.object_name = 'PLSCOPE_PROC1') variables
WHERE sig.signature = variables.signature
ORDER BY object_type, usage_id

USAGE       USAGE_ID   OBJECT_NAME                    OBJECT_TYPE
----------- ---------- ------------------------------ -------------
DECLARATION 3          PLSCOPE_PROC1                  PROCEDURE
ASSIGNMENT  4          PLSCOPE_PROC1                  PROCEDURE
REFERENCE   7          PLSCOPE_PROC1                  PROCEDURE
ASSIGNMENT  9          PLSCOPE_PROC1                  PROCEDURE
```

You should be able to see, even from these simple examples, that PL/Scope offers enormous potential in helping you better understand your code and analyze the impact of change on that code.

Lucas Jellema of AMIS has produced more interesting and complex examples of using PL/Scope to validate naming conventions. You can find these queries in the *11g_plscope_amis.sql* file on the book's website.

In addition, I have created a helper package and demonstration scripts to help you get started with PL/Scope. Check out the *plscope_helper*.** files, as well as the other *plscope*.** files.

Managing Dependencies and Recompiling Code

Another very important phase of PL/SQL compilation and execution is the checking of program *dependencies*. A dependency (in PL/SQL) is a reference from a stored program to some database object outside that program. Server-based PL/SQL programs can have dependencies on tables, views, types, procedures, functions, sequences, synonyms, object types, package specifications, etc. Program units are not, however, dependent on package bodies or object type bodies; these are the "hidden" implementations.

Oracle's basic dependency principle for PL/SQL is, loosely speaking:

> Do not use the currently compiled version of a program if any of the objects on which it depends have changed since it was compiled.

The good news is that most dependency management happens automatically, from the tracking of dependencies to the recompilation required to keep everything synchron-

ized. You can't completely ignore this topic, though, and the following sections should help you understand how, when, and why you'll need to intervene.

In Oracle Database 10*g* and earlier, dependencies were tracked with the granularity of a program unit. So if a procedure was dependent upon a function within a package or a column within a table, the dependent unit was the package or the table. This granularity has been the standard from the dawn of PL/SQL—until recently.

Beginning with Oracle Database 11*g*, the granularity of dependency tracking has improved. Instead of tracking the dependency to the unit (for example, a package or a table), the grain is now the element within the unit (for example, the columns in a table and the formal parameters of a subprogram). This *fine-grained dependency tracking* means that your program will not be invalidated if you add a program or overload an existing program in an existing package. Likewise, if you add a column to a table, the database will not automatically invalidate all PL/SQL programs that reference the table —only those programs that reference all columns, as in a SELECT * or by using the anchored declaration %ROWTYPE, will be affected. The following sections explore this situation in detail.

The section "Qualify All References to Variables and Columns in SQL Statements" on page 59 in Chapter 3 provides an example of this fine-grained dependency management.

It would be nice to report on the fine-grained dependencies that Oracle Database 11*g* manages, but as of Oracle Database 11*g* Release 2, this data is not available in any of the data dictionary views. I hope that they will "published" for our use in the future.

If, however, you are not yet building and deploying applications on Oracle Database 11*g* or later, object-level dependency tracking means that almost any change to underlying database objects will cause a wide ripple effect of invalidations.

Analyzing Dependencies with Data Dictionary Views

You can use several of the data dictionary views to analyze dependency relationships.

Let's take a look at a simple example. Suppose that I have a package named bookworm on the server. This package contains a function that retrieves data from the books table. After I create the table and then create the package, both the package specification and the body are VALID:

```
SELECT object_name, object_type, status
  FROM USER_OBJECTS
 WHERE object_name = 'BOOKWORM';
```

```
OBJECT_NAME                    OBJECT_TYPE         STATUS
------------------------------ ------------------- -------
BOOKWORM                       PACKAGE             VALID
BOOKWORM                       PACKAGE BODY        VALID
```

Behind the scenes, when I compiled my PL/SQL program, the database determined a list of other objects that BOOKWORM needs in order to compile successfully. I can explore this dependency graph using a query of the data dictionary view USER_DEPENDENCIES:

```
SELECT name, type, referenced_name, referenced_type
  FROM USER_DEPENDENCIES
 WHERE name = 'BOOKWORM';

NAME            TYPE            REFERENCED_NAME REFERENCED_TYPE
--------------- --------------- --------------- ---------------
BOOKWORM        PACKAGE         STANDARD        PACKAGE
BOOKWORM        PACKAGE BODY    STANDARD        PACKAGE
BOOKWORM        PACKAGE BODY    BOOKS           TABLE
BOOKWORM        PACKAGE BODY    BOOKWORM        PACKAGE
```

Figure 20-1 illustrates this information as a directed graph, where the arrows indicate a "depends on" relationship. In other words, the figure shows that:

- The bookworm package specification and body both depend on the built-in package named STANDARD (see the sidebar "Flying the STANDARD" on page 767).
- The bookworm package body depends on its corresponding specification and on the books table.

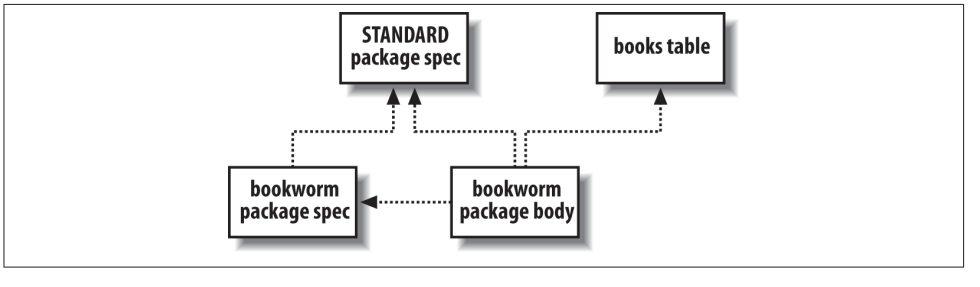

Figure 20-1. Dependency graph of the bookworm package

For the purpose of tracking dependencies, the database records a package specification and body as two different entities. Every package body will have a dependency on its corresponding specification, but the specification will never depend on its body. Nothing depends on the body. Hey, the package might not even have a body!

If you've been responsible for maintaining someone else's code during your career, you will know that performing impact analysis relies not so much on "depends-on" infor-

mation as it does on "referenced-by" information. Let's say that I'm contemplating a change in the structure of the books table. Naturally, I'd like to know everything that might be affected:

```
SELECT name, type
  FROM USER_DEPENDENCIES
 WHERE referenced_name = 'BOOKS'
   AND referenced_type = 'TABLE';

NAME                           TYPE
------------------------------ ------------
ADD_BOOK                       PROCEDURE
TEST_BOOK                      PACKAGE BODY
BOOK                           PACKAGE BODY
BOOKWORM                       PACKAGE BODY
FORMSTEST                      PACKAGE
```

As you can see, in addition to the bookworm package, there are some programs in my schema I haven't told you about—but fortunately, the database never forgets. Nice!

As clever as the database is at keeping track of dependencies, it isn't clairvoyant: in the data dictionary, the database can only track dependencies of local stored objects written with static calls. There are plenty of ways that you can create programs that do not appear in the USER_DEPENDENCIES view. These include external programs that embed SQL or PL/SQL, remote stored procedures or client-side tools that call local stored objects, and any programs that use dynamic SQL.

As I was saying, if I alter the table's structure by adding a column:

```
ALTER TABLE books MODIFY popularity_index NUMBER (8,2);
```

then the database will immediately and automatically invalidate all program units that depend on the books table (or, in Oracle Database 11*g* and later, only those program units that reference this column). Any change in the DDL time of an object—even if you just rebuild it with no changes—will cause the database to invalidate dependent program units (see the sidebar "Avoiding Those Invalidations" on page 777). Actually, the database's automatic invalidation is even more sophisticated than that; if you own a program that performs a particular DML statement on a table in another schema, and your privilege to perform that operation gets revoked, this action will also invalidate your program.

After the change, a query against USER_OBJECTS shows me the following information:

```
/* File on web: invalid_objects.sql */
SQL> SELECT object_name, object_type, status
     FROM USER_OBJECTS
     WHERE status = 'INVALID';

   OBJECT_NAME                    OBJECT_TYPE        STATUS
   ------------------------------ ------------------ -------
```

```
ADD_BOOK                       PROCEDURE          INVALID
BOOK                           PACKAGE BODY       INVALID
BOOKWORM                       PACKAGE BODY       INVALID
FORMSTEST                      PACKAGE            INVALID
FORMSTEST                      PACKAGE BODY       INVALID
TEST_BOOK                      PACKAGE BODY       INVALID
```

By the way, this again illustrates a benefit of the two-part package arrangement: for the most part, the package bodies have been invalidated, but not the specifications. As long as the specification doesn't change, program units that depend on the package will not be invalidated. The only specification that has been invalidated here is for FORMSTEST, which depends on the books table because (as I happen to know) it uses the anchored declaration books%ROWTYPE.

One final note: another way to look at programmatic dependencies is to use Oracle's DEPTREE_FILL procedure in combination with the DEPTREE or IDEPTREE views. As a quick example, if I run the procedure using:

```
BEGIN DEPTREE_FILL('TABLE', USER, 'BOOKS'); END;
```

I can then get a nice listing by selecting from the IDEPTREE view:

```
SQL> SELECT * FROM IDEPTREE;

DEPENDENCIES
-----------------------------------------
TABLE SCOTT.BOOKS
   PROCEDUE SCOTT.ADD_BOOK
   PACKAGE BODY SCOTT.BOOK
   PACKAGE BODY SCOTT.TEST_BOOK
   PACKAGE BODY SCOTT.BOOKWORM
   PACKAGE SCOTT.FORMSTEST
      PACKAGE BODY SCOTT.FORMSTEST
```

This listing shows the result of a recursive "referenced-by" query. If you want to use these objects yourself, execute the *$ORACLE_HOME/rdbms/admin/utldtree.sql* script to build the utility procedure and views in your own schema. Or, if you prefer, you can emulate it with a query such as:

```
SELECT RPAD (' ', 3*(LEVEL-1)) || name || ' (' || type || ') '
  FROM user_dependencies
     CONNECT BY PRIOR RTRIM(name || type) =
             RTRIM(referenced_name || referenced_type)
       START WITH referenced_name = 'name' AND referenced_type = 'type'
```

Now that you've seen how the server keeps track of relationships among objects, let's explore one way that the database takes advantage of such information.

Flying the STANDARD

All but the most minimal database installations will have a built-in package named STANDARD available in the database. This package gets created along with the data dictionary views from *catalog.sql* and contains many of the core features of the PL/SQL language, including:

- Functions such as INSTR and LOWER
- Comparison operators such as NOT, =, and >
- Predefined exceptions such as DUP_VAL_ON_INDEX and VALUE_ERROR
- Subtypes such as STRING and INTEGER

You can view the source code for this package by looking at the file *standard.sql*, which you will normally find in the *$ORACLE_HOME/rdbms/admin* subdirectory.

STANDARD's specification is the "root" of the PL/SQL dependency graph; that is, it depends upon no other PL/SQL programs, but most PL/SQL programs depend upon it. This package is explored in more detail in Chapter 24.

Fine-Grained Dependency (Oracle Database 11g)

One of the nicest features of PL/SQL is its automated dependency tracking. The Oracle database automatically keeps track of all database objects on which a program unit is dependent. If any of those objects are subsequently modified, the program unit is marked INVALID and must be recompiled. For example, in the case of the scope_demo package, the inclusion of the query from the employees table means that this package is marked as being dependent on that table.

As I mentioned earlier, prior to Oracle Database 11*g*, dependency information was recorded only with the granularity of the object as a whole. If *any* change at all is made to that object, all dependent program units are marked INVALID, *even if the change does not affect those program units*.

Consider the scope_demo package. It is dependent on the employees table, but it refers only to the department_id and salary columns. In Oracle Database 10*g*, I can change the size of the first_name column and this package will be marked INVALID.

In Oracle Database 11*g*, Oracle fine-tuned its dependency tracking down to the element within an object. In the case of tables, the Oracle database now records that a program unit depends on specific columns within a table. With this approach, the database can

avoid unnecessary recompilations, making it easier for you to evolve your application code base.

In Oracle Database 11*g* and later, I can change the size of my first_name column and this package is *not* marked INVALID, as you can see here:

```
ALTER TABLE employees MODIFY first_name VARCHAR2(2000)
/
Table altered.
SELECT object_name, object_type, status
  FROM all_objects
 WHERE owner = USER AND object_name = 'SCOPE_DEMO'/

OBJECT_NAME                    OBJECT_TYPE         STATUS
------------------------------ ------------------- -------
SCOPE_DEMO                     PACKAGE             VALID
SCOPE_DEMO                     PACKAGE BODY        VALID
```

Note, however, that unless you fully qualify all references to PL/SQL variables inside your embedded SQL statements, you will not be able to take full advantage of this enhancement.

Specifically, qualification of variable names will avoid invalidation of program units when new columns are *added* to a dependent table.

Consider that original, unqualified SELECT statement in set_global:

```
SELECT COUNT (*)
  INTO l_count
  FROM employees
 WHERE department_id = l_inner AND salary > l_salary;
```

As of Oracle Database 11*g*, fine-grained dependency means that the database will note that the scope_demo package is dependent only on department_id and salary.

Now suppose that the DBA adds a column to the employees table. Since there are unqualified references to PL/SQL variables in the SELECT statement, it is possible that the new column name will change the dependency information for this package. Namely, if the new column name is the same as an unqualified reference to a PL/SQL variable, the database will now resolve that reference to the *column name*. Thus, the database will need to update the dependency information for scope_demo, which means that it needs to invalidate the package.

If, conversely, you *do* qualify references to all your PL/SQL variables inside embedded SQL statements, then when the database compiles your program unit, it knows that there is no possible ambiguity. Even when columns are added, the program unit will remain VALID.

Note that the INTO list of a query is not actually a part of the SQL statement. As a result, variables in that list do not persist into the SQL statement that the PL/SQL compiler

derives. Consequently, qualifying (or not qualifying) that variable with its scope name will have no bearing on the database's dependency analysis.

Remote Dependencies

Server-based PL/SQL immediately becomes invalid whenever there is a change in a local object on which it depends. However, if it depends on an object in a remote database and that object changes, the local database does not attempt to invalidate the calling PL/SQL program in real time. Instead, the local database defers the checking until runtime.

Here is a program that has a remote dependency on the procedure recompute_prices, which lives across the database link *findat.ldn.world*:

```
PROCEDURE synch_em_up (tax_site_in IN VARCHAR2, since_in IN DATE)
IS
BEGIN
   IF tax_site_in = 'LONDON'
   THEN
      recompute_prices@findat.ldn.world(cutoff_time => since_in);
   END IF;
END;
```

If you recompile the remote procedure and some time later try to run synch_em_up, you are likely to get an ORA-04062 error with accompanying text such as *timestamp* (or *signature*) *of package "SCOTT.recompute_prices" has been changed*. If your call is still legal, the database will recompile synch_em_up, and if it succeeds, its next invocation should run without error. To understand the database's remote procedure call behavior, you need to know that the PL/SQL compiler always stores two kinds of information about each referenced remote procedure—its timestamp and its signature:

Timestamp
: The most recent date and time (down to the second) when an object's specification was reconstructed, as given by the TIMESTAMP column in the USER_OBJECTS view. For PL/SQL programs, this is not necessarily the same as the most recent compilation time because it's possible to recompile an object without reconstructing its specification. (Note that this column is of the DATE datatype, not the newer TIMESTAMP datatype.)

Signature
: A footprint of the actual shape of the object's specification. Signature information includes the object's name and the ordering, datatype family, and mode of each parameter.

So, when I compiled synch_em_up, the database retrieved both the timestamp and the signature of the remote procedure called recomputed_prices, and stored a representation of them with the bytecode of synch_em_up.

How do you suppose the database uses this information at runtime? The model is simple: it uses either the timestamp or the signature, depending on the current value of the parameter REMOTE_DEPENDENCIES_MODE. If that timestamp or signature information, which is stored in the local program's bytecode, doesn't match the actual value of the remote procedure at runtime, you get the ORA-04062 error.

Oracle's default remote dependency mode is the timestamp method, but this setting can sometimes cause unnecessary recompilations. The DBA can change the database's initialization parameter REMOTE_DEPENDENCIES_MODE, or you can change your session's setting, like this:

```
ALTER SESSION SET REMOTE_DEPENDENCIES_MODE = SIGNATURE;
```

or, inside PL/SQL:

```
EXECUTE IMMEDIATE 'ALTER SESSION SET REMOTE_DEPENDENCIES_MODE = SIGNATURE';
```

Thereafter, for the remainder of that session, every PL/SQL program run will use the signature method. As a matter of fact, Oracle's client-side tools always execute this ALTER SESSION...SIGNATURE command as the first thing they do after connecting to the database, overriding the database setting.

Oracle Corporation recommends using signature mode on client tools like Oracle Forms and timestamp mode on server-to-server procedure calls. Be aware, though, that signature mode can cause false negatives—situations where the runtime engine thinks that the signature hasn't changed, but it really has—in which case the database does not force an invalidation of a program that calls it remotely. You can wind up with silent computational errors that are difficult to detect and even more difficult to debug. Here are several risky scenarios:

- Changing only the default value of one of the called program's formal parameters. The caller will continue to use the old default value.
- Adding an overloaded program to an existing package. The caller will not bind to the new version of the overloaded program even if it is supposed to.
- Changing just the name of a formal parameter. The caller may have problems if it uses named parameter notation.

In these cases, you will have to perform a manual recompilation of the caller. In contrast, the timestamp mode, while prone to false positives, is immune to false negatives. In other words, it won't miss any needed recompilations, but it may force recompilation that is not strictly required. This safety is no doubt why Oracle uses it as the default for server-to-server RPCs.

If you do use the signature method, Oracle recommends that you add any new functions or procedures at the *end* of package specifications because doing so reduces false positives.

In the real world, minimizing recompilations can make a significant difference in application availability. It turns out that you can trick the database into thinking that a local call is really remote so that you can use signature mode. This is done using a loopback database link inside a synonym. Here is an example that assumes you have an Oracle Net service name "localhost" that connects to the local database:

```
CREATE DATABASE LINK loopback
     CONNECT TO bob IDENTIFIED BY swordfish USING 'localhost'
/
CREATE OR REPLACE PROCEDURE volatilecode AS
BEGIN
   -- whatever
END;
/
CREATE OR REPLACE SYNONYM volatile_syn FOR volatilecode@loopback
/
CREATE OR REPLACE PROCEDURE save_from_recompile AS
BEGIN
   ...
   volatile_syn;
   ...
END;
/
```

To take advantage of this arrangement, your production system would then include an invocation such as this:

```
BEGIN
   EXECUTE IMMEDIATE 'ALTER SESSION SET REMOTE_DEPENDENCIES_MODE=SIGNATURE';
   save_from_recompile;
END;
/
```

As long as you don't do anything that alters the signature of volatilecode, you can modify and recompile it without invalidating save_from_recompile or causing a runtime error. You can even rebuild the synonym against a different procedure entirely. This approach isn't completely without drawbacks; for example, if volatilecode outputs anything using DBMS_OUTPUT, you won't see it unless save_from_recompile retrieves it explicitly over the database link and then outputs it directly. But for many applications, such workarounds are a small price to pay for the resulting increase in availability.

Limitations of Oracle's Remote Invocation Model

Through Oracle Database 11*g* Release 2, there is no direct way for a PL/SQL program to use any of the following package constructs on a remote server:

- Variables (including constants)
- Cursors
- Exceptions

This limitation applies not only to client PL/SQL calling the database server, but also to server-to-server RPCs.

The simple workaround for variables is to use "get and set" programs to encapsulate the data. In general, you should be doing that anyway because it is an excellent programming practice. Note, however, that Oracle does not allow you to reference remote package public variables, *even indirectly*. So if a subprogram simply refers to a package public variable, you cannot invoke the subprogram through a database link.

The workaround for cursors is to encapsulate them with open, fetch, and close subprograms. For example, if you've declared a book_cur cursor in the specification of the book_maint package, you could put this corresponding package body on the server:

```
PACKAGE BODY book_maint
AS
   prv_book_cur_status BOOLEAN;

   PROCEDURE open_book_cur IS
   BEGIN
      IF NOT book_maint.book_cur%ISOPEN
      THEN
         OPEN book_maint.book_cur;
      END IF;
   END;
   FUNCTION next_book_rec RETURN books%ROWTYPE
   IS
      l_book_rec books%ROWTYPE;
   BEGIN
      FETCH book_maint.book_cur INTO l_book_rec;
      prv_book_cur_status := book_maint.book_cur%FOUND;
      RETURN l_book_rec;
   END;

   FUNCTION book_cur_is_found RETURN BOOLEAN
   IS
   BEGIN
      RETURN prv_book_cur_status;
   END;

   PROCEDURE close_book_cur IS
```

```
    BEGIN
       IF book_maint.book_cur%ISOPEN
       THEN
          CLOSE book_maint.book_cur;
       END IF;
    END;
END book_maint;
```

Unfortunately, this approach won't work around the problem of using remote exceptions; the exception "datatype" is treated differently from true datatypes. Instead, you can use the RAISE_APPLICATION_ERROR procedure with a user-defined exception number between −20000 and −20999. See Chapter 6 for a discussion of how to write a package to help your application manage this type of exception.

Recompiling Invalid Program Units

In addition to becoming invalid when a referenced object changes, a new program may be in an invalid state as the result of a failed compilation. In any event, no PL/SQL program marked as INVALID will run until a successful recompilation changes its status to VALID. Recompilation can happen in one of three ways:

Automatic runtime recompilation
: The PL/SQL runtime engine will, under many circumstances, automatically recompile an invalid program unit when that program unit is called.

ALTER...COMPILE recompilation
: You can use an explicit ALTER command to recompile the package.

Schema-level recompilation
: You can use one of many alternative built-ins and custom code to recompile all invalid program units in a schema or database instance.

Automatic runtime compilation

Since Oracle maintains information about the status of program units compiled into the database, it knows when a program unit is invalid and needs to be recompiled. When a user connected to the database attempts to execute (directly or indirectly) an invalid program unit, the database will automatically attempt to recompile that unit.

You might then wonder: why do we need to explicitly recompile program units at all? There are two reasons:

- In a production environment, "just in time" recompilation can have a ripple effect, in terms of both performance degradation and cascading invalidations of other database objects. You'll greatly improve the user experience by recompiling all invalid program units when users are not accessing the application (if at all possible).

- Recompilation of a program unit that was previously executed by another user connected to the same instance can and usually *will* result in an error that looks like this:

```
ORA-04068: existing state of packages has been discarded
ORA-04061: existing state of package "SCOTT.P1" has been invalidated
ORA-04065: not executed, altered or dropped package "SCOTT.P1"
ORA-06508: PL/SQL: could not find program unit being called
```

This type of error occurs when a package that has *state* (one or more variables declared at the package level) has been recompiled. All sessions that had previously initialized that package are now out of sync with the newly compiled package. When the database tries to reference or run an element of that package, it cannot "find program unit" and throws an exception.

The solution? Well, you (or the application) *could* trap the exception and then simply call that same program unit again. The package state will be reset (that's what the ORA-4068 error message is telling us), and the database will be able to execute the program. Unfortunately, the states of *all packages*, including DBMS_OUTPUT and other built-in packages, will also have been reset in that session. It is very unlikely that users will be able to continue running the application successfully.

What this means for users of PL/SQL-based applications is that whenever the underlying code needs to be updated (recompiled), all users must stop using the application. That is *not* an acceptable scenario in today's world of "always on" Internet-based applications. Oracle Database 11*g* Release 2 finally addressed this problem by offering support for "hot patching" of application code through the use of edition-based redefinition. This topic is covered briefly at the end of this chapter.

The bottom line on automatic recompilation bears repeating: prior to Oracle Database 11*g* Release 2, in live production environments, do not do *anything* that will invalidate or recompile (automatically or otherwise) any stored objects for which sessions might have instantiations that will be referred to again.

Fortunately, development environments don't usually need to worry about ripple effects, and automatic recompilation outside of production can greatly ease our development efforts. While it might still be helpful to recompile all invalid program units (explored in the following sections), it is not as critical a step.

ALTER...COMPILE recompilation

You can always recompile a program unit that has previously been compiled into the database using the ALTER...COMPILE command. In the case presented earlier, for example, I know by looking in the data dictionary that three program units were invalidated.

To recompile these program units in the hope of setting their status back to VALID, I can issue these commands:

```
ALTER PACKAGE bookworm COMPILE BODY REUSE SETTINGS;
ALTER PACKAGE book COMPILE BODY REUSE SETTINGS;
ALTER PROCEDURE add_book COMPILE REUSE SETTINGS;
```

Notice the inclusion of REUSE SETTINGS. This clause ensures that all the compilation settings (optimization level, warnings level, etc.) previously associated with this program unit will remain the same. If you do not include REUSE SETTINGS, then the current settings of the session will be applied upon recompilation.

Of course, if you have many invalid objects, you will not want to type ALTER COMPILE commands for each one. You *could* write a simple query, like the following one, to *generate* all the ALTER commands:

```
SELECT 'ALTER ' || object_type || ' ' || object_name
       || ' COMPILE REUSE SETTINGS;'
  FROM user_objects
 WHERE status = 'INVALID'
```

But the problem with this "bulk" approach is that as you recompile one invalid object, you may cause many others to be marked INVALID. You are much better off relying on more sophisticated methods for recompiling all invalid program units; these are covered next.

Schema-level recompilation

Oracle offers a number of ways to recompile all invalid program units in a particular schema. Unless otherwise noted, the following utilities must be run from a schema with SYSDBA authority. All files listed here may be found in the *$ORACLE_HOME/Rdbms/Admin* directory. The options include:

utlip.sql
: Invalidates and recompiles all PL/SQL code and views in the entire database. (Actually, it sets up some data structures, invalidates the objects, and prompts you to restart the database and run *utlrp.sql*.)

utlrp.sql
: Recompiles all of the invalid objects in serial. This is appropriate for single-processor hardware. If you have a multiprocessor machine, you'll probably want to use *utlrcmp.sql* instead.

utlrcmp.sql
: Like *utlrp.sql*, recompiles all invalid objects, but in parallel; it works by submitting multiple recompilation requests into the database's job queue. You can supply the "degree of parallelism" as an integer argument on the command line. If you leave it null or supply 0, then the script will attempt to select the proper degree of parallelism

on its own. However, even Oracle warns that this parallel version may not yield dramatic performance results because of write contention on system tables.

DBMS_UTILITY.RECOMPILE_SCHEMA
: This procedure has been around since Oracle8 Database and can be run from any schema; SYSDBA authority is *not* required. It will recompile program units in the specified schema. Its header is defined as follows:

```
DBMS_UTILITY.COMPILE_SCHEMA (
    schema VARCHAR2
  , compile_all BOOLEAN DEFAULT TRUE,
  , reuse_settings BOOLEAN DEFAULT FALSE
);
```

Prior to Oracle Database 10*g*, this utility was poorly designed and often invalidated as many program units as it recompiled to VALID status. Now, it seems to work as one would expect.

UTL_RECOMP
: This built-in package, first introduced in Oracle Database 10*g*, was designed for database upgrades or patches that require significant recompilation. It has two programs, one that recompiles invalid objects serially and one that uses DBMS_JOB to recompile in parallel. To recompile all of the invalid objects in a database instance in parallel, for example, a DBA needs only to run this single command:

```
UTL_RECOMP.recomp_parallel
```

When running this parallel version, it uses the DBMS_JOB package to queue up the recompile jobs. When this happens, all other jobs in the queue are temporarily disabled to avoid conflicts with the recompilation.

Here is an example of calling the serial version to recompile all invalid objects in the SCOTT schema:

```
CALL UTL_RECOMP.recomp_serial ('SCOTT');
```

If you have multiple processors, the parallel version may help you complete your recompilations more rapidly. As Oracle notes in its documentation of this package, however, compilation of stored programs results in updates to many catalog structures and is I/O-intensive; the resulting speedup is likely to be a function of the speed of your disks.

Here is an example of requesting recompilation of all invalid objects in the SCOTT schema, using up to four simultaneous threads for the recompilation steps:

```
CALL UTL_RECOMP.recomp_parallel ('SCOTT', 4);
```

Solomon Yakobson, an outstanding Oracle DBA and general technologist, has also written a recompile utility that can be used by non-DBAs to recompile all invalid program units in dependency order. It handles stored programs, views (including materialized views), triggers, user-defined object types, and dimensions. You can find the utility in a file named *recompile.sql* on the book's website.

Avoiding Those Invalidations

When a database object's DDL time changes, the database's usual *modus operandi* is to immediately invalidate all of its dependents on the local database.

In Oracle Database 10*g* and later releases, recompiling a stored program via its original creation script will not invalidate dependents. This feature does not extend to recompiling a program using ALTER...COMPILE or via automatic recompilation, which *will* invalidate dependents. Note that even if you use a script, the database is very picky; if you change anything in your source code—even just a single letter—that program's dependents will be marked INVALID.

Compile-Time Warnings

Compile-time warnings can greatly improve the maintainability of your code and reduce the chance that bugs will creep into it. Compile-time warnings differ from compile-time errors; with warnings, your program will still compile and run. You may, however, encounter unexpected behavior or reduced performance as a result of running code that is flagged with warnings.

This section explores how compile-time warnings work and which issues are currently detected. Let's start with a quick example of applying compile-time warnings in your session.

A Quick Example

A very useful compile-time warning is *PLW-06002: Unreachable code*. Consider the following program (available in the *cantgothere.sql* file on the book's website). Because I have initialized the salary variable to 10,000, the conditional statement will *always* send me to line 9. Line 7 will never be executed:

```
      /* File on web: cantgothere.sql */
1     PROCEDURE cant_go_there
2     AS
3        l_salary NUMBER := 10000;
4     BEGIN
5        IF l_salary > 20000
```

```
 6       THEN
 7          DBMS_OUTPUT.put_line ('Executive');
 8       ELSE
 9          DBMS_OUTPUT.put_line ('Rest of Us');
10       END IF;
11     END cant_go_there;
```

If I compile this code in any release prior to Oracle Database 10*g*, I am simply told "Procedure created." If, however, I have enabled compile-time warnings in my session in this release or later, when I try to compile the procedure I get this response from the compiler:

```
SP2-0804: Procedure created with compilation warnings

SQL> SHOW err
Errors for PROCEDURE CANT_GO_THERE:

LINE/COL ERROR
-------- -----------------------------------
7/7      PLW-06002: Unreachable code
```

Given this warning, I can now go back to that line of code, determine why it is unreachable, and make the appropriate corrections.

Enabling Compile-Time Warnings

Oracle allows you to turn compile-time warnings on and off, and also to specify the types of warnings that interest you. There are three categories of warnings:

Severe
: Conditions that could cause unexpected behavior or actual wrong results, such as aliasing problems with parameters.

Performance
: Conditions that could cause performance problems, such as passing a VARCHAR2 value to a NUMBER column in an UPDATE statement.

Informational
: Conditions that do not affect performance or correctness, but that you might want to change to make the code more maintainable.

Oracle lets you enable/disable compile-time warnings for a specific category, for all categories, and even for specific, individual warnings. You can do this with either the ALTER DDL command or the DBMS_WARNING built-in package.

To turn on compile-time warnings in your system as a whole, issue this command:

```
ALTER SYSTEM SET PLSQL_WARNINGS='string'
```

The following command, for example, turns on compile-time warnings in your system for all categories:

```
ALTER SYSTEM SET PLSQL_WARNINGS='ENABLE:ALL';
```

This is a useful setting to have in place during development because it will catch the largest number of potential issues in your code.

To turn on compile-time warnings in your session for severe problems only, issue this command:

```
ALTER SESSION SET PLSQL_WARNINGS='ENABLE:SEVERE';
```

And if you want to alter compile-time warnings settings for a particular, already compiled program, you can issue a command like this:

```
ALTER PROCEDURE hello COMPILE PLSQL_WARNINGS='ENABLE:ALL' REUSE SETTINGS;
```

Make sure to include REUSE SETTINGS to make sure that all *other* settings (such as the optimization level) are not affected by the ALTER command.

You can tweak your settings with a very high level of granularity by combining different options. For example, suppose that I want to see all performance-related issues, that I will not concern myself with server issues for the moment, and that I would like the compiler to treat *PLW-05005: function exited without a RETURN* as a compile error. I would then issue this command:

```
ALTER SESSION SET PLSQL_WARNINGS=
  'DISABLE:SEVERE'
 ,'ENABLE:PERFORMANCE'
 ,'ERROR:05005';
```

I especially like this "treat as error" option. Consider the *PLW-05005: function returns without value* warning. If I leave PLW-05005 simply as a warning, then when I compile my no_return function, shown next, the program does compile, and I *can* use it in my application:

```
SQL> CREATE OR REPLACE FUNCTION no_return
  2     RETURN VARCHAR2
  3  AS
  4  BEGIN
  5     DBMS_OUTPUT.PUT_LINE (
  6        'Here I am, here I stay');
  7  END no_return;
  8  /
SP2-0806: Function created with compilation warnings

SQL> SHOW ERR
Errors for FUNCTION NO_RETURN:

LINE/COL ERROR
```

```
-------- -----------------------------------------------------------------
1/1      PLW-05005: function NO_RETURN returns without value at line 7
```

If I now alter the treatment of that error with the ALTER SESSION command just shown and then recompile no_return, the compiler stops me in my tracks:

```
Warning: Procedure altered with compilation errors
```

By the way, I could also change the settings for that particular program only, to flag this warning as a "hard" error, with a command like this:

```
ALTER PROCEDURE no_return COMPILE PLSQL_WARNINGS = 'error:6002' REUSE SETTINGS
/
```

You can, in each of these variations of the ALTER command, also specify ALL as a quick and easy way to refer to all compile-time warning categories, as in:

```
ALTER SESSION SET PLSQL_WARNINGS='ENABLE:ALL';
```

Oracle also provides the DBMS_WARNING package, which provides the same capabilities to set and change compile-time warning settings through a PL/SQL API. DBMS_WARNING goes beyond the ALTER command, allowing you to make changes to those warning controls that you care about while leaving all the others intact. You can also easily restore the original settings when you're done.

DBMS_WARNING was designed to be used in install scripts in which you might need to disable a certain warning, or treat a warning as an error, for individual program units being compiled. You might not have any control over the scripts surrounding those for which you are responsible. Each script's author should be able to set the warning settings he wants, while inheriting a broader set of settings from a more global scope.

Some Handy Warnings

In the following sections, I present a subset of all the warnings Oracle has implemented, with an example of the type of code that will elicit the warning and descriptions of interesting behavior (where present) in the way that Oracle has implemented compile-time warnings.

To see the full list of warnings available in any given Oracle version, search for the PLW section of the *Error Messages* book of the Oracle documentation set.

PLW-05000: Mismatch in NOCOPY qualification between specification and body

The NOCOPY compiler hint tells the Oracle database that, if possible, you would like it to *not* make a copy of your IN OUT arguments. This can improve the performance of programs that pass large data structures, such as collections or CLOBs.

You need to include the NOCOPY hint in both the specification and the body of your program (relevant for packages and object types). If the hint is not present in both, the database will apply whatever is specified in the specification.

Here is an example of code that will generate this warning:

```
/* File on web: plw5000.sql */
PACKAGE plw5000
IS
   TYPE collection_t IS
      TABLE OF VARCHAR2 (100);

   PROCEDURE proc (
      collection_in IN OUT NOCOPY
         collection_t);
END plw5000;

PACKAGE BODY plw5000
IS
   PROCEDURE proc (
      collection_in IN OUT
         collection_t)
   IS
   BEGIN
      DBMS_OUTPUT.PUT_LINE ('Hello!');
   END proc;
END plw5000;
```

Compile-time warnings will display as follows:

```
SQL> SHOW ERRORS PACKAGE BODY plw5000
Errors for PACKAGE BODY PLW5000:

LINE/COL ERROR
-------- -----------------------------------------------------------------
3/20     PLW-05000: mismatch in NOCOPY qualification between specification
         and body

3/20     PLW-07203: parameter 'COLLECTION_IN' may benefit from use of the
         NOCOPY compiler hint
```

PLW-05001: Previous use of 'string' (at line string) conflicts with this use

This warning will make itself heard when you have declared more than one variable or constant with the same name. It can also pop up if the parameter list of a program defined in a package specification is different from that of the definition in the package body.

You may be saying to yourself: I've seen that error before, but it is a compilation error, not a warning. And, in fact, you are right, in that the following program simply will not compile:

```
/* File on web: plw5001.sql */
PROCEDURE plw5001
IS
   a   BOOLEAN;
```

```
   a   PLS_INTEGER;
BEGIN
   a := 1;
   DBMS_OUTPUT.put_line ('Will not compile');
END plw5001;
```

You will receive the following compile error: *PLS-00371: at most one declaration for 'A' is permitted in the declaration section.*

So why is there a *warning* for this situation? Consider what happens when I remove the assignment to the variable named a:

```
SQL> CREATE OR REPLACE PROCEDURE plw5001
  2  IS
  3     a   BOOLEAN;
  4     a   PLS_INTEGER;
  5  BEGIN
  6     DBMS_OUTPUT.put_line ('Will not compile?');
  7  END plw5001;
  8  /
Procedure created.
```

The program compiles! The database does not flag the PLS-00371 because I have not actually *used* either of the variables in my code. The PLW-05001 warning fills that gap by giving me a heads up if I have declared, but not yet used, variables with the same name, as you can see here:

```
SQL> ALTER PROCEDURE plw5001 COMPILE plsql_warnings = 'enable:all';
SP2-0805: Procedure altered with compilation warnings

SQL> SHOW ERRORS
Errors for PROCEDURE PLW5001:

LINE/COL ERROR
-------- -----------------------------------------------------------------
4/4      PLW-05001: previous use of 'A' (at line 3) conflicts with this use
```

PLW-05003: Same actual parameter (string and string) at IN and NOCOPY may have side effects

When you use NOCOPY with an IN OUT parameter, you are asking PL/SQL to pass the argument by reference, rather than by value. This means that any changes to the argument are made immediately to the variable in the outer scope. "By value" behavior (NOCOPY is not specified or the compiler ignores the NOCOPY hint), on the other hand, dictates that changes within the program are made to a local copy of the IN OUT parameter. When the program terminates, these changes are then copied to the actual parameter. (If an error occurs, the changed values are *not* copied back to the actual parameter.)

Use of the NOCOPY hint increases the possibility that you will run into the issue of argument aliasing, in which two different names point to the same memory location.

Aliasing can be difficult to understand and debug; a compile-time warning that catches this situation will come in very handy.

Consider this program:

```
/* File on web: plw5003.sql */
PROCEDURE very_confusing (
   arg1   IN              VARCHAR2
 , arg2   IN OUT          VARCHAR2
 , arg3   IN OUT NOCOPY   VARCHAR2
)
IS
BEGIN
   arg2 := 'Second value';
   DBMS_OUTPUT.put_line ('arg2 assigned, arg1 = ' || arg1);
   arg3 := 'Third value';
   DBMS_OUTPUT.put_line ('arg3 assigned, arg1 = ' || arg1);
END;
```

It's a simple enough program: pass in three strings, two of which are IN OUT; assign values to those IN OUT arguments; and display the value of the first IN argument's value after each assignment.

Now I will run this procedure, passing the very same local variable as the argument for each of the three parameters:

```
SQL> DECLARE
  2     str   VARCHAR2 (100) := 'First value';
  3  BEGIN
  4     DBMS_OUTPUT.put_line ('str before = ' || str);
  5     very_confusing (str, str, str);
  6     DBMS_OUTPUT.put_line ('str after = ' || str);
  7  END;
  8  /
str before = First value
arg2 assigned, arg1 = First value
arg3 assigned, arg1 = Third value
str after = Second value
```

Notice that while very_confusing is still running, the value of the arg1 argument was not affected by the assignment to arg2. Yet when I assigned a value to arg3, the value of arg1 (an IN argument) was changed to "Third value"! Furthermore, when very_confusing terminated, the assignment to arg2 was applied to the str variable. Thus, when control returned to the outer block, the value of the str variable was set to "Second value", effectively writing over the assignment of "Third value".

As I said earlier, parameter aliasing can be very confusing. So, if you enable compile-time warnings, programs such as plw5003 may be revealed to have potential aliasing problems:

```
SQL> CREATE OR REPLACE PROCEDURE plw5003
  2  IS
  3     str   VARCHAR2 (100) := 'First value';
  4  BEGIN
  5     DBMS_OUTPUT.put_line ('str before = ' || str);
  6     very_confusing (str, str, str);
  7     DBMS_OUTPUT.put_line ('str after = ' || str);
  8  END plw5003;
  9  /
SP2-0804: Procedure created with compilation warnings

SQL> SHOW ERR

Errors for PROCEDURE PLW5003:
LINE/COL ERROR
-------- -----------------------------------------------------------------
6/4      PLW-05003: same actual parameter(STR and STR) at IN and NOCOPY
         may have side effects
6/4      PLW-05003: same actual parameter(STR and STR) at IN and NOCOPY
         may have side effects
```

PLW-05004: Identifier string is also declared in STANDARD or is a SQL built-in

Many PL/SQL developers are unaware of the STANDARD package and its implications for their PL/SQL code. For example, it is common to find programmers who assume that names like INTEGER and TO_CHAR are reserved words in the PL/SQL language. That is not the case. They are, respectively, a datatype and a function *declared in the STANDARD package*.

STANDARD is one of the two default packages of PL/SQL (the other is DBMS_STANDARD). Because STANDARD is a default package, you do not need to qualify references to datatypes like INTEGER, NUMBER, PLS_INTEGER, and so on with "STANDARD"—but you could, if you so desired.

PLW-5004 notifies you if you happen to have declared an identifier with the same name as an element in STANDARD (or a SQL built-in; most built-ins—but not all—are declared in STANDARD).

Consider this procedure definition:

```
      /* File on web: plw5004.sql
 1    PROCEDURE plw5004
 2    IS
 3       INTEGER   NUMBER;
 4
 5       PROCEDURE TO_CHAR
 6       IS
 7       BEGIN
 8          INTEGER := 10;
 9       END TO_CHAR;
10    BEGIN
```

```
11       TO_CHAR;
12    END plw5004;
```

Compile-time warnings for this procedure will display as follows:

```
LINE/COL ERROR
-------- -----------------------------------------------------------------
3/4      PLW-05004: identifier INTEGER is also declared in STANDARD
         or is a SQL builtin
5/14     PLW-05004: identifier TO_CHAR is also declared in STANDARD
         or is a SQL builtin
```

You should avoid reusing the names of elements defined in the STANDARD package unless you have a very specific reason to do so.

PLW-05005: Function string returns without value at line string

This warning makes me happy. A function that does not return a value is a very badly designed function. This is a warning that I would recommend you ask the database to treat as an error with the "ERROR:5005" syntax in your PLSQL_WARNINGS setting.

You already saw one example of such a function: no_return. That was a very obvious chunk of code; there wasn't a single RETURN in the entire executable section. Your code will, of course, be more complex. The fact that a RETURN may not be executed could well be hidden within the folds of complex conditional logic.

At least in some of these situations, though, the database will *still* detect the problem. The following program demonstrates one such situation:

```
 1 FUNCTION no_return (
 2     check_in IN BOOLEAN)
 3     RETURN VARCHAR2
 4  AS
 5  BEGIN
 6     IF check_in
 7     THEN
 8        RETURN 'abc';
 9     ELSE
10        DBMS_OUTPUT.put_line (
11        'Here I am, here I stay');
12     END IF;
13  END no_return;
```

Oracle has detected a branch of logic that will not result in the execution of a RETURN, so it flags the program with a warning. The *plw5005.sql* file on the book's website contains even more complex conditional logic, demonstrating that the warning is raised for less trivial code structures as well.

PLW-06002: Unreachable code

The Oracle database now performs static (compile-time) analysis of your program to determine if any lines of code in your program will never be reached during execution. This is extremely valuable feedback to receive, but you may find that the compiler warns you of this problem on lines that do not, at first glance, seem to be unreachable. In fact, Oracle notes in the description of the action to take for this error that you should "disable the warning if much code is made unreachable intentionally and the warning message is more annoying than helpful."

You already saw an example of this compile-time warning in the section "A Quick Example" on page 777. Now consider the following code:

```
      /* File on web: plw6002.sql */
 1    PROCEDURE plw6002
 2    AS
 3       l_checking BOOLEAN := FALSE;
 4    BEGIN
 5       IF l_checking
 6       THEN
 7          DBMS_OUTPUT.put_line ('Never here...');
 8       ELSE
 9          DBMS_OUTPUT.put_line ('Always here...');
10          GOTO end_of_function;
11       END IF;
12       <<end_of_function>>
13       NULL;
14    END plw6002;
```

In Oracle Database 10*g* and later, you will see the following compile-time warnings for this program:

```
LINE/COL ERROR
-------- -----------------------------
5/7      PLW-06002: Unreachable code
7/7      PLW-06002: Unreachable code
13/4     PLW-06002: Unreachable code
```

I see why line 7 is marked as unreachable: l_checking is set to FALSE, and so line 7 can never run. But why is line 5 marked unreachable? It seems as though, in fact, that code will *always* be run! Furthermore, line 13 will always be run as well because the GOTO will direct the flow of execution to that line through the label. Yet it too is tagged as unreachable.

The reason for this behavior is that prior to Oracle Database 11*g*, the unreachable code warning is generated after optimization of the code. In Oracle Database 11*g* and later, the analysis of unreachable code is much cleaner and more helpful.

The compiler does *not* give you false positives; when it says that line *N* is unreachable, it is telling you that the line truly will never be executed, accurately reflecting the optimized code.

There are currently scenarios of unreachable code that are *not* flagged by the compiler. Here is one example:

```
/* File on web: plw6002.sql */
FUNCTION plw6002 RETURN VARCHAR2
AS
BEGIN
   RETURN NULL;
   DBMS_OUTPUT.put_line ('Never here...');
END plw6002;
```

Certainly, the call to DBMS_OUTPUT.PUT_LINE is unreachable, but the compiler does not currently detect that state—until 12.1.

PLW-07203: Parameter string may benefit from use of the NOCOPY compiler hint

As mentioned earlier in relation to PLW-05005, use of NOCOPY with complex, large IN OUT parameters can improve the performance of programs under certain conditions. This warning will flag programs whose IN OUT parameters might benefit from NOCOPY. Here is an example of such a program:

```
/* File on web: plw7203.sql */
PACKAGE plw7203
IS
   TYPE collection_t IS TABLE OF VARCHAR2 (100);

   PROCEDURE proc (collection_in IN OUT collection_t);
END plw7203;
```

This is another one of those warnings that will be generated for lots of programs and may become a nuisance. The warning/recommendation is certainly valid, but for most programs the impact of this optimization will not be noticeable. Furthermore, you are unlikely to switch to NOCOPY without making other changes in your code to handle situations where the program terminates before completing, possibly leaving your data in an uncertain state.

PLW-07204: Conversion away from column type may result in suboptimal query plan

This warning will surface when you call a SQL statement from within PL/SQL and rely on implicit conversions within that statement. Here is an example:

```
/* File on web: plw7204.sql */
FUNCTION plw7204
   RETURN PLS_INTEGER
AS
   l_count PLS_INTEGER;
BEGIN
```

```
   SELECT COUNT(*) INTO l_count
     FROM employees
    WHERE salary = '10000';
   RETURN l_count;
END plw7204;
```

Related tightly to this warning is *PLW-7202: bind type would result in conversion away from column type*.

PLW-06009: Procedure string OTHERS handler does not end in RAISE or RAISE_APPLICATION_ERROR

This warning (added in Oracle Database 11*g*) appears when your OTHERS exception handler does not execute some form of RAISE (re-raising the same exception or raising another) and does not call RAISE_APPLICATION_ERROR. In other words, there is a good possibility that you are "swallowing" the error and ignoring it. Under certain fairly rare circumstances, ignoring errors is the appropriate thing to do. Usually, however, you will want to pass an exception back to the enclosing block.

Here is an example:

```
/* File on web: plw6009.sql */
FUNCTION plw6009
   RETURN PLS_INTEGER
AS
   l_count   PLS_INTEGER;
BEGIN
   SELECT COUNT ( * ) INTO l_count
     FROM dual WHERE 1 = 2;

   RETURN l_count;
EXCEPTION
   WHEN OTHERS
   THEN
      DBMS_OUTPUT.put_line ('Error!');
      RETURN 0;
END plw6009;
```

Testing PL/SQL Programs

I get great satisfaction out of creating new things, and that is one of the reasons I so enjoy writing software. I love to take an interesting idea or challenge, and then come up with a way of using the PL/SQL language to meet that challenge.

I have to admit, though, that I don't really like having to take the time to test my software (nor do I like to write documentation for it). I do it, but I don't really do enough of it. And I have this funny feeling that I am not alone. The overwhelming reality is that developers generally perform an inadequate number of inadequate tests and figure that

if the users don't find a bug, there is no bug. Why does this happen? Let me count the ways...

Psychology of success and failure
: We are so focused on getting our code to work correctly that we generally shy away from bad news—or from taking the chance of getting bad news. Better to do some cursory testing, confirm that everything seems to be working OK, and then wait for others to find bugs, if there are any (as if there were any doubt).

Deadline pressures
: Hey, it's Internet time! Time to market determines all. We need everything yesterday, so let's release pre-beta software as production and let our users test/suffer through our applications.

Management's lack of understanding
: IT management is notorious for not really understanding the software development process. If we aren't given the time and authority to write (and I mean *write* in the broadest sense, including testing, documentation, refinement, etc.) code properly, we will always end up with buggy junk that no one wants to admit ownership of.

Overhead of setting up and running tests
: If it's a big deal to write and run tests, they won't get done. We'll decide that we don't have time; after all, there is always something else to work on. One consequence of this is that more and more of the testing is handed over to the QA department, if there is one. That transfer of responsibility is, on the one hand, positive. Professional quality assurance professionals can have a tremendous impact on application quality. Yet developers must take and exercise responsibility for unit testing their own code; otherwise, the testing/QA process is much more frustrating and extended.

The end result is that software almost universally needs more—much more—testing and fewer bugs. How can we test more effectively in the world of PL/SQL?

In the following sections, I answer that question by first taking a look at what I would consider to be a weak but typical manual testing process. Then I draw some conclusions about the key problems with manual testing. From there, I take a look at automated testing options for PL/SQL code.

Typical, Tawdry Testing Techniques

When testing the effect of a program, you need to identify what has been changed by that program: for example, the string returned by a function, the table updated by a procedure. Then you need to decide, in advance, what the correct behavior of the program for a given set of inputs and setup (a test case) would be. Then, after the program has run, you must compare the actual results (what was changed by the program) to the expected values. If they match, your program worked. If there is a discrepancy, the program failed.

That's a very general description of testing; the critical question is how you go about defining all needed test cases and implementing the tests. Let's start by looking at what I would consider to be a fairly typical (and typically bad) approach to testing.

Say that I am writing a big application with lots of string manipulation. I've got a "hangnail" called SUBSTR; this function bothers me, and I need to take care of it. What's the problem? SUBSTR is great when you know the starting location of a string and the number of characters you want. In many situations, though, I have only the start and end locations, and then I have to compute the number of characters. But which formula is it?

```
end - start
end - start + 1
end - start - 1
```

I can never remember (the correct answer is end – start + 1), so I write a program that will remember it for me—the betwnstr function:

```
/* File on web: betwnstr.sf */
FUNCTION betwnstr (string_in IN VARCHAR2
                 , start_in IN INTEGER
                 , end_in IN INTEGER
)
   RETURN VARCHAR2
IS
BEGIN
   RETURN (SUBSTR ( string_in,  start_in,  end_in - start_in + 1));
END betwnstr;
```

That was easy, and I am very certain that this formula is correct—I reverse engineered it from an example. Still, I should test it. The problem is that I am under a lot of pressure, and this is just one little utility among many other programs I must write and test. So I throw together a crude "test script" built around DBMS_OUTPUT.PUT_LINE, and run it:

```
BEGIN
   DBMS_OUTPUT.put_line (NVL (betwnstr ('abcdefg', 3, 5)
                            , '**Really NULL**'));
END;

cde
```

It worked—how exciting! But I should run more tests than that one. Let's change the end value to 500. It should return the rest of the string, just like SUBSTR would:

```
BEGIN
   DBMS_OUTPUT.put_line (NVL (betwnstr ('abcdefg', 3, 500)
                            , '**Really NULL**'));
END;

cdefg
```

It worked again! This is my lucky day. Now, let's make sure it handles NULLs properly:

```
BEGIN
   DBMS_OUTPUT.put_line (NVL (betwnstr ('abcdefg', NULL, 5)
                            , '**Really NULL**'));
END;

**Really NULL**
```

Three in a row. This is one very correct function, wouldn't you say? No, you are probably (or, at the very least, you *should* be) shaking your head and saying to yourself: "That's just pitiful. You haven't scratched the surface of all the scenarios you need to test. Why, you didn't even change the value of the first argument! Plus, every time you changed your input values you threw away your last test."

Good points, all. So, rather than just throw up some different argument values willy-nilly, I will come up with a list of test cases whose behavior I want to verify, as shown in following table.

String	Start	End	Result
abcdefg	1	3	abc
abcdefg	0	3	abc
<anything>	NULL	NOT NULL	NULL
<anything>	NOT NULL	NULL	NULL
NULL	*<anything>*	*<anything>*	NULL
abcdefg	Positive number	Smaller than start	NULL
abcdefg	1	Number larger than length of string	abcdefg

From this grid, I will then construct a simple test script like the following:

```
/* File on web: betwnstr.tst */
BEGIN
   DBMS_OUTPUT.put_line ('Test 1: ' || betwnstr (NULL, 3, 5));
   DBMS_OUTPUT.put_line ('Test 2: ' || betwnstr ('abcdefgh', 0, 5));
   DBMS_OUTPUT.put_line ('Test 3: ' || betwnstr ('abcdefgh', 3, 5));
   DBMS_OUTPUT.put_line ('Test 4: ' || betwnstr ('abcdefgh', -3, -5));
   DBMS_OUTPUT.put_line ('Test 5: ' || betwnstr ('abcdefgh', NULL, 5));
   DBMS_OUTPUT.put_line ('Test 6: ' || betwnstr ('abcdefgh', 3, NULL));
   DBMS_OUTPUT.put_line ('Test 7: ' || betwnstr ('abcdefgh', 3, 100));
END;
```

And now whenever I need to test betwnstr, I simply run this script and check the results. Based on that initial implementation, they are:

```
SQL> @betwnstr.tst
Test 1:
Test 2: abcdef
Test 3: cde
Test 4:
```

```
Test 5:
Test 6:
Test 7: cdefgh
```

Ah... "check the results." So easy to say, but how easy is it to do? Did this test work properly? I have to go through the results line by line and compare them to my grid. Plus, if I am going to test this code thoroughly, I will probably have more than 30 test cases (what about *negative* start and end values?). It will take me at least several minutes to scan the results of my test. And this is a ridiculously simple piece of code. The thought of extending this technique to any "real" code is frightening. Imagine if my program modified two tables and returned two OUT arguments. I might have hundreds of test cases, plus nontrivial setup tasks and the challenge of figuring out how to make sure the contents of my tables are correct.

Yet this is the approach many developers take routinely when "testing" their code. To conclude, almost all the code testing we do suffers from these key drawbacks:

Handwritten test code
: We write the test code ourselves, which severely limits how much testing we can do. Who has time to write all that code?

Incomplete testing
: If we were completely honest with ourselves, we would be forced to admit that we don't actually *test* most of our code. Rather, we try a few of the most obvious cases to reassure ourselves that the program is not obviously broken. That's a far cry from actual testing.

Throwaway testing
: Our tests are not repeatable. We are so focused on getting the program to work *right now* that we can't think ahead and realize that we—or someone else—will have to do the same tests, over and over again, in the future.

Manual verification
: If we rely on our own eyes and observational skills to verify test results, it will take way too much time and likely result in erroneous conclusions. We are so desperate for our programs to work that we overlook minor issues or apparent failures, and explain them away.

Testing after development
: I believe that most programmers say to themselves, "When I am done writing my program, I will test it." Sounds so reasonable, does it not? And yet it is a fatally flawed principle. First, we are never "done" writing our programs. So we inevitably run out of time for testing. Second, and more troubling, if we think about testing only after we finish implementing our programs, we will subconsciously choose to run those tests that are most likely to succeed, and avoid those that we are pretty sure will cause problems. It's the way our brains are wired.

Clearly, if we are going to test effectively and thoroughly, we need to take a different path. We need a way to define our tests so that they can easily be maintained over time. We need to be able to easily run our tests and then, most importantly, determine without lengthy analysis the outcome: success or failure. And we need to figure out a way to run tests without having to write enormous amounts of test code.

In the following sections, I first offer some advice on how to approach testing your code. Then I examine automated testing options for PL/SQL developers, with a focus on utPLSQL and Quest Code Tester for Oracle.

General Advice for Testing PL/SQL Code

Whatever tool you choose to help you test, you should take the following into consideration if you hope to successfully transform the quality of your testing:

Commit to testing
: The most important change to make is inside our heads. We have to change our perspective from "I sure hope this program will work" to "I want to be able to *prove* that my program works." Once you commit to testing, you will find yourself writing more modular code that can be more easily tested. You will also then have to find tools to help you test more efficiently.

Get those test cases out of your head before you start writing your program—and onto a piece of paper or into a tool that manages your tests
: The important thing is to externalize your belief of what needs to be tested; otherwise, you are likely to lose or ignore that information. On Monday, when I start to build my program, I can easily think of 25 different scenarios (requirements) that need to be covered (implemented). Three days later I have run out of time, so I switch to testing. Suddenly and very oddly, I can only remember five test cases (the most obvious ones). If you make a list of your known test cases at the very beginning of the development process, you are much more likely to remember and verify them.

Don't worry about 100% test coverage
: I doubt that there has ever been a nontrivial software program that was *completely* tested. You should not set as your objective 100% coverage of all possible test cases. It is very unlikely to happen and will serve only to discourage you. The most important thing about testing is to get started. So what if you only implement 10% of your test cases in phase 1? That's 10% more than you were testing before. And once your test cases (and associated code) are in place, it is much easier to add to them.

Integrate testing into development
: You cannot afford to put off testing until after you are "done" writing your software. Instead, you should think about testing as early as possible in the process. List out

your test cases, construct your test code, and then run those tests *as you implement, debug, and enhance your program*. After every change, run your test again to verify that you are making progress. If you need a fancy name (a *methodology*) to be convinced about the value of this approach, check out the widely adopted (in object-oriented circles) *test-driven development* (TDD).

Get those regression tests in place
: All of the preceding suggestions, plus the tools described next, will help you build a *regression test*. This kind of test is intended to make sure that your code does not regress or move backward. It's terribly embarrassing when we roll out v2 of our product and half the features of v1 are broken. "How can this happen?" wail our users. And if we gave them an honest answer, they would run screaming from the meeting room, because that answer would be: "Sorry, but we didn't have time to write a regression test. That means when we make a change in our spaghetti code we really don't have any idea what might have been broken." This is unacceptable, yes? Once you have a regression test in place, though, you can make changes and roll out new versions with confidence.

Automated Testing Options for PL/SQL

Today, PL/SQL developers can choose from the following automated frameworks and tools for testing their code:

utPLSQL
: The first framework for PL/SQL, utPLSQL is essentially "JUnit for PL/SQL." It implements *extreme programming* testing principles and automatically runs your handwritten test code, verifying the results. Note that while hundreds of organizations use this tool, it is not an active open source project at this time.

Code Tester for Oracle
: This commercial testing tool offers the highest level of test automation. It generates test code from UI-specified expected behaviors, runs those tests, and displays the results using a red light/green light format. It is offered by Dell as part of the Toad Development Suite.

Oracle SQL Developer
: Oracle SQL Developer is a free IDE from Oracle that offers integrated unit testing, very similar to Code Tester for Oracle.

PLUTO
: PLUTO is similar to utPLSQL, but it is implemented using Oracle object types.

dbFit
: The dbFit framework follows a very different approach to specifying tests: tabular scripts. It consists of a set of FIT fixtures that enable FIT/FitNesse tests to execute directly against a database.

Tracing PL/SQL Execution

You get your program to compile. You run your Quest Code Tester test definition—and it tells you that you have a failed test case: there's a bug somewhere in your program. How, then, do you find the cause of the problem? You can certainly dive right into your source code debugger (virtually all PL/SQL editors include visual debuggers with UI-settable breaks and watchpoints). You may, however, want to consider tracing execution of your program first.

Before exploring options for tracing, let's first look at the difference between debugging and tracing. Developers often conflate these two processes into a single activity, yet they are quite different. To summarize, you first trace execution to obtain in-depth information about application behavior, helping you isolate the source of the problem; you then use a debugger to find the specific lines of code that cause a bug.

A key distinction between tracing and debugging is that tracing is a "batch" process, while debugging is interactive. I turn on tracing and run my application code. When it is done, I open the trace log and use the information there to inform my debugging session. When I debug, I step through my code line by line (usually starting from a breakpoint that is close to the source of the problem, as indicated by trace data). A debug session is usually very time-consuming and tedious, so it makes an awful lot of sense to do everything I can to minimize the time spent debugging. Solid, proactive tracing will help me do this.

Every application should include programmer-defined tracing (also known as *instrumentation*). This section explores options for tracing, but first let's review some principles that we should follow when implementing tracing:

- Trace calls should remain in the code throughout all phases of development and deployment. In other words, do not insert trace calls while developing and then remove them when the application goes into production. Tracing is often the best opportunity you have to understand what is happening in your application when it is run by a real, live user in a production environment.
- Keep the overhead of calls to your trace utility to an absolute minimum. When tracing is disabled, the user should see no impact on application performance.
- Do not call the DBMS_OUTPUT.PUT_LINE program directly within your application code as the trace mechanism. This built-in is not flexible or powerful enough for high-quality tracing.
- Call DBMS_UTILITY.FORMAT_CALL_STACK or a subprogram of UTL_CALL_STACK (12*c* or higher) to save the execution call stack with your trace information.
- Make it easy for the end user to enable and disable tracing of your backend code. It should not require the intervention of the support organization to switch on

tracing. Nor should you have to provide a different version of the application that includes tracing.

- If someone else has already created a trace utility that you can use (and that meets these and your own principles), don't waste your time building your own trace mechanism.

Let's consider that last principle first. What tracing utilities already do exist?

DBMS_APPLICATION_INFO
: This built-in package offers an API that allows applications to "register" their current execution status with the Oracle database. This tracing utility writes trace information to V$ dynamic views. It is described in the next section.

Log4PLSQL
: This open source tracing framework is modeled after (and built upon) log4J, a very popular Java logging mechanism. You can get more information about Log4PLSQL at the Log 4 PL/SQL website.

opp_trace
: This package is available on the book's website; it offers basic, effective tracing functionality for PL/SQL applications (file: *opp_trace.sql*).

DBMS_TRACE
: This built-in utility traces the execution of PL/SQL code, but does not allow you to log any application data as part of your trace. You can, however, use this trace utility without making any changes to your source code. It is described in a later section.

You can also use one of Oracle's built-in PL/SQL profilers to obtain information about the *performance* profile of each line and subprogram in your application. The profilers are discussed in Chapter 21.

DBMS_UTILITY.FORMAT_CALL_STACK

The DBMS_UTILITY.FORMAT_CALL_STACK function returns a formatted string that shows the execution call stack: the sequence of invocations of procedures or functions that led to the point at which the function was called. In other words, this function answers the question "How did I get here?"

Here is a demonstration of using this function and what the formatted string looks like:

```
/* File on web: callstack.sql */
SQL> CREATE OR REPLACE PROCEDURE proc1
  2  IS
  3  BEGIN
  4     DBMS_OUTPUT.put_line (DBMS_UTILITY.format_call_stack);
```

```
  5  END;
  6  /

Procedure created.

SQL> CREATE OR REPLACE PACKAGE pkg1
  2  IS
  3     PROCEDURE proc2;
  4  END pkg1;
  5  /

Package created.

SQL> CREATE OR REPLACE PACKAGE BODY pkg1
  2  IS
  3     PROCEDURE proc2
  4     IS
  5     BEGIN
  6        proc1;
  7     END;
  8  END pkg1;
  9  /

Package body created.

SQL> CREATE OR REPLACE PROCEDURE proc3
  2  IS
  3  BEGIN
  4     FOR indx IN 1 .. 1000
  5     LOOP
  6        NULL;
  7     END LOOP;
  8
  9     pkg1.proc2;
 10  END;
 11  /

Procedure created.

SQL> BEGIN
  2     proc3;
  3  END;
  4  /
----- PL/SQL Call Stack -----
          object      line  object
          handle    number  name
000007FF7EA83240         4  procedure HR.PROC1
000007FF7E9CC3B0         6  package body HR.PKG1
000007FF7EA0A3B0         9  procedure HR.PROC3
000007FF7EA07C00         2  anonymous block
```

Here are some things to keep in mind about DBMS_UTILITY.FORMAT_CALL_STACK:

- If you call a subprogram in a package, the formatted call stack will only show the package name, not the subprogram name (once the code has been compiled all these names are "gone"—i.e., not available to the runtime engine).
- You can view the actual code that was being executed by querying the contents of ALL_SOURCE for the program unit name and the line number.
- Whenever you are tracing application execution (explored in more detail in the following sections) or logging errors, call this function and save the string in your trace/log repository for later viewing and analysis.
- Developers have for years been asking Oracle to provide a way to parse the contents of the string, and in Oracle Database 12*c*, it finally responded with a new package: UTL_CALL_STACK. (If you are not yet running 12.1 or higher, check out the *callstack.pkg* file for a similar kind of utility.)

UTL_CALL_STACK (Oracle Database 12c)

Oracle Database 12*c* provides the UTL_CALL_STACK package, which provides information about currently executing subprograms. Though the package name suggests that it only provides information about the execution call stack, it also offers access to the error stack *and* error backtrace data.

Each stack has a *depth* (an indicator of the location in the stack), and you can ask for the information at a certain depth in each of the three types of stacks made available through the package. One of the greatest improvements of UTL_CALL_STACK over DBMS_UTILITY.FORMAT_CALL_STACK is that you can obtain a *unit-qualified name*, which concatenates together the unit name, all lexical parents of the subprogram, and the subprogram name. This additional information is not available, however, for the backtrace.

Here is a table of the subprograms in this package, followed by a couple of examples.

Name	Description
BACKTRACE_DEPTH	Returns the number of backtrace items in the backtrace.
BACKTRACE_LINE	Returns the line number of the unit at the specified backtrace depth.
BACKTRACE_UNIT	Returns the name of the unit at the specified backtrace depth.
CONCATENATE_SUBPROGRAM	Returns a concatenated form of a unit-qualified name.

Name	Description

DYNAMIC_DEPTH	Returns the number of subprograms on the call stack, including SQL, Java, and other non-PL/SQL contexts invoked along the way. For example, if A calls B calls C calls B, then this stack, written as a line with dynamic depths underneath it, would look like:
	`A B C B`
	`4 3 2 1`

ERROR_DEPTH	Returns the number of errors on the call stack.
ERROR_MSG	Returns the error message of the error at the specified error depth.
ERROR_NUMBER	Returns the error number of the error at the specified error depth.
LEXICAL_DEPTH	Returns the lexical nesting level of the subprogram at the specified dynamic depth.
OWNER	Returns the owner name of the unit of the subprogram at the specified dynamic depth.
SUBPROGRAM	Returns the unit-qualified name of the subprogram at the specified dynamic depth.
UNIT_LINE	Returns the line number of the unit of the subprogram at the specified dynamic depth.

Here are just a few of the things you can call UTL_CALL_STACK for:

1. Get the unit-qualified name of the currently executing subprogram. The call to UTL_CALL_STACK.SUBPROGRAM returns an *array* of strings. The CONCATENATE_SUBPROGRAM function takes that array and returns a single string of "."-delimited names. Here, I pass 1 to SUBPROGRAM because I want to get information about the program at the top of the stack—the currently executing subprogram:

```
/* File on web: 12c_utl_call_stack.sql */
CREATE OR REPLACE PACKAGE pkg1
IS
   PROCEDURE proc1;
END pkg1;
/

CREATE OR REPLACE PACKAGE BODY pkg1
IS
   PROCEDURE proc1
   IS
      PROCEDURE nested_in_proc1
      IS
      BEGIN
         DBMS_OUTPUT.put_line (
            utl_call_stack.concatenate_subprogram (utl_call_stack.subprogram (1)));
      END;
   BEGIN
      nested_in_proc1;
   END;
END pkg1;
/

BEGIN
```

```
   pkg1.proc1;
END;
/

PKG1.PROC1.NESTED_IN_PROC1
```

2. Show the name of the program unit and the line number in that unit where the current exception was raised. First I create a function named BACKTRACE_TO that "hides" the calls to UTL_CALL_STACK subprograms. In each call to BACKTRACE_UNIT and BACKTRACE_LINE, I pass the value returned by the ERROR_DEPTH function. The depth value for errors is different from that of the call stack. With the call stack, 1 is the top of the stack (the currently executing subprogram). With the error backtrace, the location in my code where the error was raised is found at ERROR_DEPTH, not 1:

```
/* File on web: 12c_utl_call_stack.sql */
CREATE OR REPLACE FUNCTION backtrace_to
   RETURN VARCHAR2
IS
BEGIN
   RETURN    utl_call_stack.backtrace_unit (utl_call_stack.error_depth)
          || ' line '
          || utl_call_stack.backtrace_line (utl_call_stack.error_depth);
END;
/

CREATE OR REPLACE PACKAGE pkg1
IS
   PROCEDURE proc1;
   PROCEDURE proc2;
END;
/

CREATE OR REPLACE PACKAGE BODY pkg1
IS
   PROCEDURE proc1
   IS
      PROCEDURE nested_in_proc1
      IS
      BEGIN
         RAISE VALUE_ERROR;
      END;
   BEGIN
      nested_in_proc1;
   END;

   PROCEDURE proc2
   IS
   BEGIN
      proc1;
   EXCEPTION
```

```
      WHEN OTHERS THEN RAISE NO_DATA_FOUND;
   END;
END pkg1;
/

CREATE OR REPLACE PROCEDURE proc3
IS
BEGIN
   pkg1.proc2;
END;
/

BEGIN
   proc3;
EXCEPTION
   WHEN OTHERS
   THEN
      DBMS_OUTPUT.put_line (backtrace_to);
END;
/

HR.PKG1 line 19
```

Here are some things to keep in mind about UTL_CALL_STACK:

- Compiler optimizations can change lexical, dynamic, and backtrace depth, since the optimization process can result in subprogram invocations being skipped.
- UTL_CALL_STACK is not supported past remote procedure call (RPC) boundaries. For example, if proc1 calls remote procedure remoteproc2, remoteproc2 will not be able to obtain information about proc1 using UTL_CALL_STACK.
- Lexical unit information is available through the PL/SQL conditional compilation feature and is therefore not exposed through UTL_CALL_STACK.

UTL_CALL_STACK is a very handy utility, but for real-world use, you will likely need to build some utilities of your own around this package's subprograms. I have built a helper package (see *12c_utl_call_stack_helper.sql* and *12c_utl_call_stack_helper_demo.sql* on the book's website) with utilities I thought you might find helpful.

DBMS_APPLICATION_INFO

The DBMS_APPLICATION_INFO built-in package provides an API that allows applications to "register" their current execution status with the Oracle database. Once registered, information about the status of an application can be externally monitored through several of the V$ views (which are, in turn, based on the underlying X$ tables). Using the V$ views as the trace repository is what distinguishes this package from all other tracing alternatives.

The DBMS_APPLICATION_INFO package is used to develop applications that can be monitored in various ways, including:

- Module usage (where users spend their time in the application)
- Resource accounting by transaction and module
- End-user tracking and resource accounting in three-tier architectures
- Incremental recording of long-running process statistics

DBAs and developers can monitor applications registered using DBMS_APPLICATION_INFO for performance and resource consumption much more closely than is otherwise possible. This facilitates better application tuning as well as more accurate usage-based cost accounting.

The following table lists the subprograms in this package; all are procedures and none can be run in SQL.

Name	Description
DBMS_APPLICATION_INFO.SET_MODULE	Sets name of module executing
DBMS_APPLICATION_INFO.SET_ACTION	Sets action within module
DBMS_APPLICATION_INFO.READ_MODULE	Reads module and action for current session
DBMS_APPLICATION_INFO.SET_CLIENT_INFO	Sets client information for session
DBMS_APPLICATION_INFO.READ_CLIENT_INFO	Reads client information for session
DBMS_APPLICATION_INFO.SET_SESSION_LONGOPS	Sets row in LONGOPS table

For thorough coverage of this package, see the chapter from the book *Oracle Built-in Packages* that we have included on this book's website.

Here is a demonstration of DBMS_APPLICATION_INFO:

```
/* File on web: dbms_application_info.sql */
PROCEDURE drop_dept (
    deptno_IN IN employees.department_id%TYPE
  , reassign_deptno_IN IN employees.department_id%TYPE
)
IS
   l_count PLS_INTEGER;
BEGIN
   DBMS_APPLICATION_INFO.SET_MODULE
      (module_name => 'DEPARTMENT FIXES'
      ,action_name => null);
   DBMS_APPLICATION_INFO.SET_ACTION (action_name => 'GET COUNT IN DEPT');

   SELECT COUNT(*)
     INTO l_count
     FROM employees
    WHERE department_id = deptno_IN;
```

```
      DBMS_OUTPUT.PUT_LINE ('Reassigning ' || l_count || ' employees');

      IF l_count > 0
      THEN
         DBMS_APPLICATION_INFO.SET_ACTION (action_name => 'REASSIGN EMPLOYEES');

         UPDATE employees
            SET department_id = reassign_deptno_IN
          WHERE department_id = deptno_IN;
      END IF;

      DBMS_APPLICATION_INFO.SET_ACTION (action_name => 'DROP DEPT');

      DELETE FROM departments WHERE department_id = deptno_IN;

      COMMIT;

      DBMS_APPLICATION_INFO.SET_MODULE(null,null);

   EXCEPTION
      WHEN OTHERS THEN
         DBMS_APPLICATION_INFO.SET_MODULE(null,null);
   END drop_dept;
```

Notice in this example that DBMS_APPLICATION_INFO is called three times to distinguish between the three steps involved in the process of dropping the department. This gives a very fine granularity to the level at which the application can be tracked.

Be sure to set the action name to a name that can identify the current transaction or logical unit of work within the module.

When the transaction terminates, call DBMS_APPLICATION_INFO.SET_ACTION and pass a null value for the action_name parameter. This ensures that if subsequent transactions do not register using DBMS_APPLICATION_INFO, they are not incorrectly counted as part of the current action. As in the example, if the program handles exceptions, the exception handler should probably reset the action information.

Tracing with opp_trace

The opp_trace utility is available on the website for the 5th edition of this book. It implements a robust tracing mechanism that allows you to direct trace output to either the screen or a table (opp_trace). Run the *opp_trace.sql* script to create the database objects. Run *opp_trace_uninstall.sql* to remove these same database objects.

You have my permission to modify this utility as needed to use it in your environment (for example, you may have to change the needs of the database objects).

Using the opp_trace API, I can enable tracing for all calls to trace as follows:

```
opp_trace.turn_on_trace;
```

In the next call to set_tracing, I enable tracing only for contexts that contain the string "balance":

```
opp_trace.turn_on_trace ('balance');
```

Now let's take a look at how I make calls to opp_trace.trace_activity in my stored programs.

I almost never call q$error_manager.trace directly. Instead, I nest it inside a call to q$error_manager.trace_enabled, as you see here:

```
IF opp_trace.trace_is_on
THEN
   opp_trace.trace_activity (
         context_in => 'generate_test_code for program'
       , data_in    => qu_program_qp.name_for_id (l_program_key)
   );
END IF;
```

I call the trace program in this way to minimize the runtime overhead of tracing. The trace_enabled function returns the value of a single Boolean flag; it passes no actual arguments and finishes its work efficiently. If it returns TRUE, then the Oracle database will evaluate all the expressions in the parameter list and call the trace procedure, which will also make sure that tracing is enabled for this specific context.

If I call the trace procedure directly in my application code, then every time the runtime engine hits that line of code, it will evaluate all the actual arguments in the parameter list and call the trace procedure. The trace procedure will then make sure that tracing is enabled for this specific context. If tracing is disabled, then nothing more happens—but notice that the application will have wasted CPU cycles evaluating the arguments and passing them into trace.

Would a user ever notice the overhead of evaluating those arguments unnecessarily? Perhaps not, but as you add more and more trace calls to your code, you increase the probability of user impact. You should instead set as a habit and standard that you always hide your actual trace calls inside an IF statement that keeps overhead to a minimum. I use a code snippet so that I can insert the IF statement with a template call to the trace mechanism without a lot of typing.

The DBMS_TRACE Facility

The DBMS_TRACE built-in package provides programs to start and stop PL/SQL tracing in a session. When tracing is turned on, the engine collects data as the program executes. The data is then written out to the Oracle server trace file.

The PL/SQL trace facility provides a trace file that shows you the specific steps executed by your code. DBMS_PROFILER and DBMS_HPROF (hierarchical profiler), which are described in Chapter 21, offer more comprehensive analyses of your application, in-

cluding timing information and counts of the number of times a specific line was executed.

Installing DBMS_TRACE

This package may not have been installed automatically with the rest of the built-in packages. To determine whether DBMS_TRACE is present, connect to SYS (or another account with SYSDBA privileges) and execute this command in SQL*Plus:

```
DESCRIBE DBMS_TRACE
```

If you see this error:

```
ORA-04043: object dbms_trace does not exist
```

then you must install the package.

To install DBMS_TRACE, remain connected as SYS (or another account with SYSDBA privileges), and run the following files in the order specified:

$ORACLE_HOME/rdbms/admin/dbmspbt.sql
$ORACLE_HOME/rdbms/admin/prvtpbt.plb

DBMS_TRACE programs

The subprograms listed in the following table are available in the DBMS_TRACE package.

Name	Description
SET_PLSQL_TRACE	Starts PL/SQL tracing in the current session
CLEAR_PLSQL_TRACE	Stops the dumping of trace data for that session
PLSQL_TRACE_VERSION	Gets the major and minor version numbers of the DBMS_TRACE package

To trace execution of your PL/SQL code, you must first start the trace with a call to:

```
DBMS_TRACE.SET_PLSQL_TRACE (trace_level INTEGER);
```

in your current session, where *trace_level* is one of the following values:

- Constants that determine which elements of your PL/SQL program will be traced:

```
DBMS_TRACE.trace_all_calls          constant INTEGER := 1;
DBMS_TRACE.trace_enabled_calls      constant INTEGER := 2;
DBMS_TRACE.trace_all_exceptions     constant INTEGER := 4;
DBMS_TRACE.trace_enabled_exceptions constant INTEGER := 8;
DBMS_TRACE.trace_all_sql            constant INTEGER := 32;
DBMS_TRACE.trace_enabled_sql        constant INTEGER := 64;
DBMS_TRACE.trace_all_lines          constant INTEGER := 128;
DBMS_TRACE.trace_enabled_lines      constant INTEGER := 256;
```

- Constants that control the tracing process:

```
DBMS_TRACE.trace_stop              constant INTEGER := 16384;
DBMS_TRACE.trace_pause             constant INTEGER := 4096;
DBMS_TRACE.trace_resume            constant INTEGER := 8192;
DBMS_TRACE.trace_limit             constant INTEGER := 16;
```

By combining the DBMS_TRACE constants, you can enable tracing of multiple PL/SQL language features simultaneously. Note that the constants that control the tracing behavior (such as DBMS_TRACE.trace_pause) should not be used in combination with the other constants (such as DBMS_TRACE.trace_enabled_calls).

To turn on tracing from all programs executed in your session, issue this call:

```
DBMS_TRACE.SET_PLSQL_TRACE (DBMS_TRACE.trace_all_calls);
```

To turn on tracing for all exceptions raised during the session, issue this call:

```
DBMS_TRACE.SET_PLSQL_TRACE (DBMS_TRACE.trace_all_exceptions);
```

Then run your code. When you are done, stop the trace session by calling:

```
DBMS_TRACE.CLEAR_PLSQL_TRACE;
```

You can then examine the contents of the trace file. The names of these files are generated by the database; you would usually look at the modification dates to figure out which file to examine. The location of the trace files is discussed in the section "Format of collected data" on page 807. You can also set an identifier in your trace file so that you can find it more easily later, as in:

```
SQL> alter session set tracefile_identifier = 'hello_steven!';
Session altered.
SQL> select tracefile from v$process where tracefile like '%hello_steven!%';
TRACEFILE
--------------------------------------------------------------------------------
/u01/app/oracle/diag/rdbms/orcl/orcl/trace/orcl_ora_24446_hello_steven!.trc
```

Note that you cannot use PL/SQL tracing with the shared server (formerly known as the *multithreaded server*, or MTS).

Control trace file contents

The trace files produced by DBMS_TRACE can get *really* big. You can focus the output by enabling only specific programs for trace data collection. Note that you cannot use this approach with remote procedure calls.

To enable tracing for any programs that are created or replaced in the current session, you can alter the session as follows:

```
ALTER SESSION SET PLSQL_DEBUG=TRUE;
```

If you don't want to alter your entire session, you can recompile a specific program unit in debug mode as follows (not applicable to anonymous blocks):

```
ALTER [PROCEDURE | FUNCTION | PACKAGE BODY] program_name COMPILE DEBUG;
```

After you have enabled the programs you're interested in, issue the following call to initiate tracing just for those program units:

```
DBMS_TRACE.SET_PLSQL_TRACE (DBMS_TRACE.trace_enabled_calls);
```

You can also restrict the trace information to only those exceptions raised within enabled programs with this call:

```
DBMS_TRACE.SET_PLSQL_TRACE (DBMS_TRACE.trace_enabled_exceptions);
```

If you request tracing for all programs or exceptions and also request tracing only for enabled programs or exceptions, the request for "all" takes precedence.

Pause and resume the trace process

The SET_PLSQL_TRACE procedure can do more than just determine which information will be traced. You can also request that the tracing process be paused and resumed. The following statement, for example, requests that no information be gathered until tracing is resumed:

```
DBMS_TRACE.SET_PLSQL_TRACE (DBMS_TRACE.trace_pause);
```

DBMS_TRACE will write a record to the trace file to show when tracing was paused and/or resumed.

Use the DBMS_TRACE.trace_limit constant to request that only the last 8,192 trace events of a run be preserved. This approach helps ensure that you can turn tracing on without overwhelming the database with trace activity. When the trace session ends, only the last 8,192 records are saved.

Format of collected data

If you request tracing only for enabled program units, and the current program unit is not enabled, no trace data is written. If the current program unit is enabled, call tracing writes out the program unit type, name, and stack depth.

Exception tracing writes out the line number where an exception is raised. It also records trace information on whether the exception is user-defined or predefined, and records the exception number in the case of predefined exceptions. If you raise a user-defined exception, you will always see an error code of 1.

Here is an example of the output from a trace of the showemps procedure:

```
*** 1999.06.14.09.59.25.394
*** SESSION ID:(9.7) 1999.06.14.09.59.25.344
------------ PL/SQL TRACE INFORMATION -----------
```

```
Levels set :  1
Trace:  ANONYMOUS BLOCK: Stack depth = 1
Trace:  PROCEDURE SCOTT.SHOWEMPS: Call to entry at line 5 Stack depth = 2
Trace:  PACKAGE BODY SYS.DBMS_SQL: Call to entry at line 1 Stack depth = 3
Trace:  PACKAGE BODY SYS.DBMS_SYS_SQL: Call to entry at line 1 Stack depth = 4
Trace:  PACKAGE BODY SYS.DBMS_SYS_SQL: ICD vector index = 21 Stack depth = 4
Trace:  PACKAGE PLVPRO.P: Call to entry at line 26 Stack depth = 3
Trace:  PACKAGE PLVPRO.P: ICD vector index = 6 Stack depth = 3
Trace:  PACKAGE BODY PLVPRO.P: Call to entry at line 1 Stack depth = 3
Trace:  PACKAGE BODY PLVPRO.P: Call to entry at line 1 Stack depth = 3
Trace:  PACKAGE BODY PLVPRO.P: Call to entry at line 1 Stack depth = 4
```

Debugging PL/SQL Programs

When you test a program, you find errors in your code. When you debug a program, you uncover the cause of an error and fix it. These are two very different processes and should not be confused. Once a program is tested, and bugs are uncovered, it is certainly the responsibility of the developer to fix those bugs. And so the debugging begins!

Many programmers find that debugging is by far the hardest part of programming. This difficulty often arises from the following factors:

Lack of understanding of the problem being solved by the program
: Most programmers like to code. They tend to not like reading and understanding specifications, and will sometimes forgo this step so that they can quickly get down to writing code. The chance of a program meeting its requirements under these conditions is slim at best.

Poor programming practices
: Programs that are hard to read (lack of documentation, too much documentation, inconsistent use of whitespace, bad choices for identifier names, etc.), are not properly modularized, and try to be too clever present a much greater challenge to debug than programs that are well designed and structured.

The program simply contains too many errors
: Without the proper analysis and coding skills, you will create code with a much higher occurrence of bugs. When you compile a program and get back five screens of compile errors, do you just want to scream and hide? It is easy to be so overwhelmed by your errors that you don't take the organized, step-by-step approach needed to fix those errors.

Limited debugging skills
: There are many different approaches to uncovering the causes of your problems. Some approaches only make life more difficult for you. If you have not been trained in the best way to debug your code, you can waste many hours, raise your blood pressure, and upset your manager.

The following sections review the debugging methods that you will want to avoid at all costs, and then offer recommendations for more effective debugging strategies.

The Wrong Way to Debug

As I present the various ways you shouldn't debug your programs, I expect that just about all of you will say to yourselves, "Well, that sure is obvious. Of course you shouldn't do that. I never do that."

And yet the next time you sit down to do your work, you may very well follow some of these obviously horrible debugging practices.

If you happen to see little bits of yourself in the paragraphs that follow, I hope you will be inspired to mend your ways.

Disorganized debugging

When faced with a bug, you become a whirlwind of frenzied activity. Even though the presence of an error indicates that you did not fully analyze the problem and figure out how the program should solve it, you do not now take the time to understand the program. Instead, you place MESSAGE statements (in Oracle Forms) or SRW.MESSAGE statements (in Oracle Reports) or DBMS_OUTPUT.PUT_LINE statements (in stored modules) all over your program in the hopes of extracting more clues.

You do not save a copy of the program before you start making changes, because that would take too much time; you are under a lot of pressure right now, and you are certain that the answer will pop right out at you. You will just remove your debug statements later.

You spend lots of time looking at information that is mostly irrelevant. You question everything about your program, even though most of it uses constructs you've employed successfully for years.

You skip lunch but make time for coffee, lots of coffee, because it is free and you want to make sure your concentration is at the most intense level possible. Even though you have no idea what is causing the problem, you think that maybe if you try this one change, it might help. You make the change and take several minutes to compile, generate, and run through the test case, only to find that the change didn't help. In fact, it seemed to cause another problem because you hadn't thought through the full impact of the change on your application.

So you back out of that change and try something else in hopes that it might work. But several minutes later, you again find that it doesn't. A friend, noticing that your fingers are trembling, offers to help. But you don't know where to start explaining the problem because you don't really know what is wrong. Furthermore, you are kind of embarrassed about what you've done so far (turning the program into a minefield of tracing state-

ments) and realize you don't have a clean version to show your friend. So you snap at the best programmer in your group and call your family to let them know you aren't going to be home for dinner that night.

Why? Because you are determined to fix that bug!

Irrational debugging

You execute your report, and it comes up empty. You spent the last hour making changes both in the underlying data structures and in the code that queries and formats the data. You are certain, however, that your modifications could not have made the report disappear.

You call your internal support hotline to find out if there is a network problem, even though File Manager clearly shows access to network drives. You further probe as to whether the database has gone down, even though you just connected successfully. You spend another 10 minutes of the support analyst's time running through a variety of scenarios before you hang up in frustration.

"They don't know anything over there," you fume. You realize that you will have to figure this one out all by yourself. So you dive into the code you just modified. You are determined to check every single line until you find the cause of your difficulty. Over the course of the next two hours, you talk aloud to yourself—a lot.

"Look at that! I called the stored procedure inside an IF statement. I never did that before. Maybe I can't call stored programs that way." So you remove the IF statement and instead use a GOTO statement to perform the branching to the stored procedure. But that doesn't fix the problem.

"My code seems fine. But it calls this other routine that Joe wrote ages ago." Joe has since moved on, making him a ripe candidate for the scapegoat. "It probably doesn't work anymore; after all, we did upgrade to a new voicemail system." So you decide to perform a standalone test of Joe's routine, which hasn't changed for two years and has no interface to voicemail. But his program seems to work fine—when it's not run from your program.

Now you are starting to get desperate. "Maybe this report should only run on weekends. Hey, can I put a local module in an anonymous block? Maybe I can use only local modules in procedures and functions! I think maybe I heard about a bug in this tool. Time for a workaround..."

You get angry and begin to understand why your eight-year-old hits the computer monitor when he can't beat the last level of *Ultra Mystic Conqueror VII*. And just as you are ready to go home and take it out on your dog, you realize that you are connected to the development database, which has almost no data at all. You switch to the test instance, run your report, and everything looks just fine.

Except, of course, for that GOTO and all the other workarounds you stuck in the report...

Debugging Tips and Strategies

In this chapter, I do not pretend to offer a comprehensive primer on debugging. The following tips and techniques, however, should help you improve on your current set of error-fixing skills.

Use a source code debugger

The single most effective thing you can do to minimize the time spent debugging your code is to use a source code debugger. One is now available in just about every PL/SQL integrated development environment (IDE). If you are using Quest's Toad or SQL Navigator, Allround Automations's PL/SQL Developer, or Oracle SQL Developer (or any other such GUI tool), you will be able to set visual breakpoints in your code with the click of a mouse, step through your code line by line, watch variables as they change their values, and so on.

The other tips in this section apply whether or not you are using a GUI-based debugger, but there is no doubt that if you are still debugging the old-fashioned way (inserting calls to DBMS_OUTPUT.PUT_LINE in dozens of places in your code), you are wasting a lot of your time. (Unfortunately, if your code is deployed at some customer site, debugging with a GUI tool is not always possible, in which case you usually have to resort to some sort of logging mechanism.)

Gather data

Gather as much data as possible about when, where, and how the error occurred. It is very unlikely that the first occurrence of an error will give you all the information you will want or need to figure out the source of that error. Upon noticing an error, you might be tempted to show off your knowledge of the program by declaring, "Got it! I know what's going on and exactly how to fix it." This can be very gratifying when it turns out that you do have a handle on the problem, and that may be the case for simple bugs. Some problems can appear simple, however, and turn out to require extensive testing and analysis. Save yourself the embarrassment of pretending (or believing) that you know more than you actually do. Before rushing to change your code, take these steps:

1. Run the program again to see if the error is reproducible.

   This will be the first indication of the complexity of the problem. It is almost impossible to determine the cause of a problem if you are unable to get it to occur predictably. Once you work out the steps needed to get the error to occur, you will have gained much valuable information about its cause.

2. Narrow the test case needed to generate the error.

   I recently had to debug a problem in one of my Oracle Forms modules. A pop-up window would lose its data under certain circumstances. At first glance, the rule seemed to be: "For a new call, if you enter only one request, that request will be

lost." If I had stopped testing at that point, I would have had to analyze all code that initialized the call record and handled the INSERT logic. Instead, I tried additional variations of data entry and soon found that the data was lost only when I navigated to the pop-up window directly from a certain item. Now I had a very narrow test case to analyze, and it became very easy to uncover the error in logic.

3. Examine the circumstances under which the problem does not occur.

   "Failure to fail" can offer many insights into the reason an error does occur. It also helps you narrow down the sections of code and the conditions you have to analyze when you go back to the program.

The more information you gather about the problem at hand, the easier it will be to solve that problem. It is worth the extra time to assemble the evidence. So, even when you are absolutely sure you are on to that bug, hold off and investigate a little further.

Remain logical at all times

Symbolic logic is the lifeblood of programmers. No matter which programming language you use, the underlying logical framework is a constant. PL/SQL has one particular syntax. The C language uses different keywords, and the IF statement looks a little different. The elegance of LISP demands a very different way of building programs. But underneath it all, symbolic logic provides the backbone on which you hang the statements that solve your problems.

The reliance on logical and rational thought in programming is one reason that it is so easy for a developer to learn a new programming language. As long as you can take the statement of a problem and develop a logical solution step by step, the particulars of a language are secondary.

With logic at the core of our being, it amazes me to see how often we programmers abandon this logic and pursue the most irrational path to solving a problem. We engage in wishful thinking and highly superstitious, irrational, or dubious thought processes. Even though we know better—much better—we find ourselves questioning code that conforms to documented functionality, that has worked in the past, and that surely works at that moment. This irrationality almost always involves shifting the blame from oneself to the "other"—the computer, the compiler, Joe, the word processor, whatever. Anything and anybody but our own pristine selves!

When you attempt to shift blame, you only put off solving your problem. Computers and compilers may not be intelligent, but they're very fast and very consistent. All they can do is follow rules, and you write the rules in your program. So when you uncover a bug in your code, take responsibility for that error. Assume that *you* did something wrong—don't blame the PL/SQL compiler, Oracle Forms, or the text editor.

If you do find yourself questioning a basic element or rule in the compiler that has always worked for you in the past, it is time to take a break. Better yet, it is time to get someone

else to look at your code. It is amazing how another pair of eyes can focus your own analytical powers on the real causes of a problem.

Strive to be the Spock of programming. Accept only what is logical. Reject that which has no explanation.

Analyze instead of trying

So, you have a pile of data and all the clues you could ask for in profiling the symptoms of your problem. Now it is time to analyze that data. For many people, analysis takes the following form: "Hmm, this looks like it could be the answer. I'll make this change, recompile, and try it to see if it works."

What's wrong with this approach? When you try a solution to see what will happen, what you are really saying is:

- You are not sure that the change really is a solution. If you were sure, you wouldn't "try" it to see what would happen. You would make the change and then test that change.
- You have not fully analyzed the error to understand its causes. If you know why an error occurs, then you know if a particular change will fix that problem. If you are unsure about the source of the error, you will be tempted to simply try a change and examine the impact. This is, unfortunately, very faulty logic.
- Even if the change stops the error from occurring, you can't be sure that your "solution" really solved anything. Because you aren't sure why the problem occurred, the simple fact that the problem doesn't reappear in your particular tests doesn't mean that you fixed the bug. The most you can say is that your change stopped the bug from occurring under certain, perhaps even most, circumstances.

To truly solve a problem, you must completely analyze the cause of the problem. Once you understand why the problem occurs, you have found the root cause and can take the steps necessary to make the problem go away in all circumstances.

When you identify a potential solution, perform a walk-through of your code based on that change. Don't execute your form. Examine your program, and mentally try out different scenarios to test your hypothesis. Once you are certain that your change actually does address the problem, you can then perform a test of that solution. You won't be *trying* anything; you will be *verifying* a fix.

Analyze your bug fully before you test solutions. If you say to yourself, "Why don't I try this?" in the hope that it will solve the problem, then you are wasting your time and debugging inefficiently.

Take breaks, and ask for help

We are often our own biggest obstacles when it comes to sorting out our problems, whether a program bug or a personal crisis. When you are stuck on the inside of a problem, it is hard to maintain an objective distance and take a fresh look.

When you are making absolutely no progress and feel that you have tried everything, try these two radical techniques:

- Take a break.
- Ask for help.

When I have struggled with a bug for any length of time without success, I not only become ineffective, I also tend to lose perspective. I pursue irrational and superstitious leads. I lose track of what I have already tested and what I have assumed to be right. I get too close to the problem to debug it effectively.

My frustration level usually correlates closely to the amount of time I have sat in my ergonomic chair and perched over my wrist-padded keyboard and stared at my low-radiation screen. Often the very simple act of stepping away from the workstation will clear my head and leave room for a solution to pop into place. Did you ever wake up the morning after a very difficult day at work to find the elusive answer sitting there at the end of your dream?

Make it a rule to get up and walk around at least once an hour when you are working on a problem—heck, even when you are writing your programs. Give your brain a chance to let its neural networks make the connections and develop new options for your programming. There is a whole big world out there. Even when your eyes are glued to the monitor and your source code, the world keeps turning. It never hurts to remind yourself of the bigger picture, even if that only amounts to taking note of the weather outside your air-conditioned cocoon.

Even more effective than taking a break is asking another person to look at your problem. There is something entirely magical about the dynamic of adding another pair of eyes to the situation. You might struggle with a problem for an hour or two, and finally, at the exact moment that you break down and explain the problem to a coworker, the solution will jump out at you. It could be a mismatch on names, a false assumption, or a misunderstanding of the IF statement logic. Whatever the case, chances are that you yourself will find it (even though you couldn't for the last two hours) as soon as you ask someone else to find it for you.

And even if the error does not yield itself quite so easily, you still have lots to gain from the perspective of another person who (a) did not write the code and has no subconscious assumptions or biases about it, and (b) isn't mad at the program.

Other benefits accrue from asking for help. You improve the self-esteem and self-confidence of other programmers by showing that you respect their opinions. If you are one of the best developers in the group, then your request for help demonstrates that you, too, sometimes make mistakes and need help from the team. This builds the sense (and the reality) of teamwork, which will improve the overall development and testing efforts on the project.

Change and test one area of code at a time

One of my biggest problems when I debug my code is that I am overconfident about my development and debugging skills, so I try to address too many problems at once. I make 5 or 10 changes, rerun my test, and get very unreliable and minimally useful results. I find that my changes cause other problems (a common phenomenon until a program stabilizes, and a sure sign that lots more debugging and testing are needed); that some, but not all, of the original errors are gone; and that I have no idea which changes fixed which errors and which changes caused new errors.

In short, my debugging effort is a mess, and I have to back out of changes until I have a clearer picture of what is happening in my program.

Unless you are making very simple changes, you should fix one problem at a time and then test that fix. The amount of time it takes to compile, generate, and test may increase, but in the long run you will be much more productive.

Another aspect of incremental testing and debugging is performing unit tests on individual modules before you test a program that calls these various modules. If you test the programs separately and determine that they work, when you debug your application as a whole (in a system test), you do not have to worry about whether those modules return correct values or perform the correct actions. Instead, you can concentrate on the code that calls the modules. (See the section "Testing PL/SQL Programs" on page 788 for more on unit testing.)

You will also find it helpful to come up with a system for keeping track of your troubleshooting efforts. Dan Clamage, a reviewer for this book, reports that he maintains a simple text file with running commentary of his efforts to reproduce the problem and what he has done to correct it. This file will usually include any SQL written to analyze the situation, setup data for test cases, a list of the modules examined, and any other items that may be of interest in the future. With this file in place, it's much easier to return at any time (e.g., after you have had a good night's sleep and are ready to try again) and follow your original line of reasoning.

Using Whitelisting to Control Access to Program Units

Most PL/SQL-based applications are made up of many packages, some of which are the top-level API to be used by programmers to implement user requirements and others of which are helper packages that are to be used only by certain other packages.

Before Oracle Database 12*c*, PL/SQL could not prevent a session from using any and all subprograms in packages to which that session's schema had been granted EXECUTE authority.

As of Oracle Database 12*c*, all PL/SQL program units have an optional ACCESSIBLE BY clause that lets you specify a whitelist of other PL/SQL units that can access the PL/SQL unit that you are creating or altering.

Let's take a look at an example. First I create my "public" package specification, which is intended to be used by other developers to build the application:

```
/* File on web: 12c_accessible_by.sql */
CREATE OR REPLACE PACKAGE public_pkg
IS
   PROCEDURE do_only_this;
END;
/
```

Next, I create the specification of my "private" package; it is private in the sense that I want to make sure that it can *only* be invoked from within the public package. So I add the ACCESSIBLE_BY clause:

```
CREATE OR REPLACE PACKAGE private_pkg
   ACCESSIBLE BY (public_pkg)
IS
   PROCEDURE do_this;

   PROCEDURE do_that;
END;
/
```

Now it's time to implement the package bodies. public_pkg.do_only_this calls the private package subprograms:

```
CREATE OR REPLACE PACKAGE BODY public_pkg
IS
   PROCEDURE do_only_this
   IS
   BEGIN
      private_pkg.do_this;
      private_pkg.do_that;
   END;
END;
/
```

```
CREATE OR REPLACE PACKAGE BODY private_pkg
IS
   PROCEDURE do_this
   IS
   BEGIN
      DBMS_OUTPUT.put_line ('THIS');
   END;

   PROCEDURE do_that
   IS
   BEGIN
      DBMS_OUTPUT.put_line ('THAT');
   END;
END;
/
```

I can now run the public package's procedure without any problem:

```
BEGIN
   public_pkg.do_only_this;
END;
/
THIS
THAT
```

But if I try to call a subprogram in the private package in an anonymous block, I see this error:

```
BEGIN
   private_pkg.do_this;
END;
/

ERROR at line 2:
ORA-06550: line 2, column 1:
PLS-00904: insufficient privilege to access object PRIVATE_PKG
ORA-06550: line 2, column 1:
PL/SQL: Statement ignored
```

And the same error occurs if I try to compile a program unit that tries to call a subprogram in the private package:

```
SQL> CREATE OR REPLACE PROCEDURE use_private
  2  IS
  3  BEGIN
  4     private_pkg.do_this;
  5  END;
  6  /

Warning: Procedure created with compilation errors.

SQL> sho err
Errors for PROCEDURE USE_PRIVATE:
```

```
LINE/COL ERROR
-------- -----------------------------------------------------------------
4/4      PL/SQL: Statement ignored
4/4      PLS-00904: insufficient privilege to access object PRIVATE_PKG
```

As you can see by the PLS errors, this issue is caught at compilation time. There is no runtime "hit" for using this feature.

Protecting Stored Code

Virtually any application I write contains proprietary information. If I write my application in PL/SQL and sell it commercially, I really don't want to let customers (or worse, competitors) see my secrets. Oracle offers a program known as *wrap* that hides—or *obfuscates*—most, if not all, of these secrets.

Some people refer to "wrapping" code as "encrypting" code, but wrapping is not true encryption. If you need to deliver information, such as a password, that *really* needs to be secure, you should not rely upon this facility. Oracle does provide a way of incorporating true encryption into your own applications using the built-in package DBMS_CRYPTO (or DBMS_OBFUSCATION_TOOLKIT in releases before Oracle Database 10*g*). Chapter 23 describes encryption and other aspects of PL/SQL application security.

When you wrap PL/SQL source, you convert your readable ASCII text source code into unreadable ASCII text source code. This unreadable code can then be distributed to customers, regional offices, and more, for creation in new database instances. The Oracle database maintains dependencies for this wrapped code as it would for programs compiled from readable text. In short, a wrapped program is treated within the database just as normal PL/SQL programs are treated; the only difference is that prying eyes can't query the USER_SOURCE data dictionary to extract trade secrets.

Oracle has, for years, provided a *wrap* executable that performs the obfuscation of your code. Starting with Oracle Database 10*g* Release 2, you can also use the DBMS_DDL.WRAP and DBMS_DDL.CREATE_WRAPPED programs to wrap dynamically constructed PL/SQL code.

Restrictions on and Limitations of Wrapping

You should be aware of the following issues when working with wrapped code:

- Wrapping makes reverse engineering of your source code difficult, but you should still avoid placing passwords and other highly sensitive information in your code

(and remember that your wrapped code *can* be reverse engineered back to a readable format).

- You cannot wrap the source code in triggers. If it is critical that you hide the contents of triggers, move the code to a package and then call the packaged program from the trigger.
- Wrapped code cannot be compiled into databases of a version lower than that of the *wrap* program. Wrapped code is upward compatible only.
- You cannot include SQL*Plus substitution variables inside code that must be wrapped.

Using the Wrap Executable

To wrap PL/SQL source code, you run the *wrap* executable. This program, named *wrap.exe*, is located in the *bin* directory of the Oracle instance. The format of the wrap command is:

```
wrap iname=infile [oname=outfile]
```

where *infile* points to the original, readable version of your program, and *outfile* is the name of the file that will contain the wrapped version of the code. If *infile* does not contain a file extension, then the default of *.sql* is assumed.

If you do not provide an oname argument, then *wrap* creates a file with the same name as *infile* but with a default extension of *.plb*, which stands for "PL/SQL binary" (a misnomer, but it gets the idea across: binaries are, in fact, unreadable).

Here are some examples of how you can use the *wrap* executable:

- Wrap a program, relying on all the defaults:

```
wrap iname=secretprog
```

- Wrap a package body, specifying overrides of all the defaults (notice that the wrapped file doesn't have to have the same filename or extension as the original):

```
wrap iname=secretbody.spb oname=shhhhhh.bin
```

Dynamic Wrapping with DBMS_DDL

Oracle Database 10*g* Release 2 introduced a way to generate wrapped code dynamically —the WRAP and CREATE_WRAPPED programs of the DBMS_DDL package:

DBMS_DDL.WRAP
: Returns a string containing an obfuscated version of your code

DBMS_DDL.CREATE_WRAPPED
: Compiles an obfuscated version of your code into the database

Both programs are overloaded to work with a single string and with arrays of strings based on the DBMS_SQL.VARCHAR2A and DBMS_SQL.VARCHAR2S collection types. Here are two examples that use these programs:

- Obfuscate and display a string that creates a tiny procedure:

```
SQL> DECLARE
  2     l_program   VARCHAR2 (32767);
  3  BEGIN
  4     l_program :=
  5         'CREATE OR REPLACE PROCEDURE dont_look IS BEGIN NULL; END;';
  6     DBMS_OUTPUT.put_line (SYS.DBMS_DDL.wrap (l_program));
  7  END;
  8  /
```

  The output is:

```
CREATE OR REPLACE PROCEDURE dont_look wrapped

a000000
369
abcd
....
XtQ19EnOI8a6hBSJmk2NebMgPHswg5nnm7+fMr2ywFy4CP6Z9P4I/v4rpXQruMAy/tJepZmB
CC0r
uIHHLcmmpkOCnm4=
```

- Read a PL/SQL program definition from a file, obfuscate it, and compile it into the database:

```
/* File on web: obfuscate_from_file.sql */
PROCEDURE obfuscate_from_file (
   dir_in    IN   VARCHAR2
 , file_in   IN   VARCHAR2
)
IS
   l_file    UTL_FILE.file_type;
   l_lines   DBMS_SQL.varchar2s;

   PROCEDURE read_file (lines_out IN OUT NOCOPY DBMS_SQL.varchar2s)
   IS BEGIN ... not critical to the example ... END read_file;
BEGIN
   read_file (l_lines);
   SYS.DBMS_DDL.create_wrapped (l_lines, l_lines.FIRST, l_lines.LAST);
END obfuscate_from_file;
```

Guidelines for Working with Wrapped Code

I have found the following guidelines useful in working with wrapped code:

- Create batch files so that you can easily, quickly, and uniformly wrap one or more files. In Windows, I create *.bat* files that contain lines like this in my source code directories:

  ```
  c:\orant\bin\wrap iname=plvrep.sps oname=plvrep.pls
  ```

  Of course, you can also create parameterized scripts and pass in the names of the files you want to wrap.
- You can only wrap package specifications and bodies, object type specifications and bodies, and standalone functions and procedures. You can run the wrap binary against any other kind of SQL or PL/SQL statement, but those files will not be changed.
- You can tell that a program is wrapped by examining the program header. It will contain the keyword WRAPPED, as in:

  ```
  PACKAGE BODY package_name WRAPPED
  ```

  Even if you don't notice the keyword WRAPPED on the first line, you will immediately know that you are looking at wrapped code because the text in USER_SOURCE will look like this:

  ```
     LINE TEXT
  ------- ----------------------
       45 abcd
       46 95a425ff
       47 a2
       48 7 PACKAGE:
  ```

  and no matter how bad your coding style is, it surely isn't *that* bad!
- Wrapped code is much larger than the original source. In my experience, a 57 KB readable package body turns into a 153 KB wrapped package body, while an 86 KB readable package body turns into a 357 KB wrapped package body. These increases in file size do result in increased requirements for storing source code in the database. The size of compiled code stays the same, although the time it takes to compile may increase.

Introduction to Edition-Based Redefinition (Oracle Database 11g Release 2)

One of the most significant enhancements in Oracle Database 11*g* Release 2 was surely *edition-based redefinition*, a new element of Oracle's high-availability solution. This feature makes it possible to upgrade the database component of an application while it

is being used; that is, Oracle now supports "hot patching" of PL/SQL-based applications. Edition-based redefinition makes it possible to minimize or completely eliminate downtime for maintenance.

With edition-based redefinition, when you need to upgrade an application while it is in use, you make a copy of any affected database objects in the application and *redefine* the copied objects in isolation from the running application. Any changes you make are not visible to and do not have any effect on users. Users can continue to run the application as it existed before your changes (to this new edition). When you are certain that all changes are correct, you then make the upgraded application available to all users.

As you can imagine, the addition of this feature has had a sweeping impact on the Oracle database. For example, if you want to see a list of *all* the objects you have defined, instead of writing a query against ALL_OBJECTS, you can now query the contents of ALL_OBJECTS_AE ("All Editions"). The unique specifier for an object is now OWNER, OBJECT_NAME, and EDITION_NAME (assuming, in any case, that the owner is editions-enabled). This one aspect is just the tip of the iceberg of all the changes that edition-based redefinition has wrought in the Oracle database.

Other Oracle database capabilities in the high-availability space can be adopted and deployed at particular sites where an application is installed without that application's needing special preparation and without its developers even knowing about the high-availability capabilities that different sites use.

Edition-based redefinition is fundamentally different. Here's how to take advantage of this feature:

- The schema(s) that own the database objects that are the application's backend must be modified to prepare the application to use edition-based redefinition. This design work should be done by the application architect, and introduced into a new (or first) version of the application. Scripts need to be written to implement this preparatory upgrade step, and those scripts must run "old-style," which is to say, offline.
- Once the application is ready for edition-based redefinition, the development team programmers responsible for scripting patches and upgrades will then need to learn how to do edition-based redefinition and write their scripts in a new way.

Given the complexity of this feature and the fact that, strictly speaking, it extends well beyond the PL/SQL language, we can do little more in this book than offer a very simple demonstration to give you a sense of how edition-based redefinition works (all code is available in the *11gR2_editions.sql* file on the book's website).

Let's start by creating a new edition. Every edition must be defined as the child of an existing edition. Furthermore, all databases upgraded to or created in Oracle Database

11*g* Release 2 and later start with one edition named ora$base. This edition always must serve as the parent of the first edition created with a CREATE EDITION statement.

Suppose that I am enhancing my Human Resources application to reflect a change in the rule for displaying the full name of an employee. Historically, I displayed names in the format "first space last," as shown here:

```
/* File on web: 11gR2_editions.sql */
FUNCTION full_name (first_in IN employees.first_name%TYPE
                  , last_in IN employees.first_name%TYPE
                    )
   RETURN VARCHAR2
IS
BEGIN
   RETURN (first_in || ' ' || last_in);
END full_name;
```

This function is defined in the ora$base edition. When I call it, I see the following output:

```
SQL> BEGIN   DBMS_OUTPUT.put_line (full_name ('Steven', 'Feuerstein'));END;/
Steven Feuerstein
```

Unfortunately, our users have changed their minds (what a surprise!): they now want names displayed in the form "last comma first." Now, this function is called all day long in the application, and I don't want to have to force our users off that application. Thankfully, we recently upgraded to Oracle Database 11*g* Release 2. So, I first create an edition for the new version of my function:

```
CREATE EDITION NEW_HR_PATCH_NAMEFORMAT
/
```

I then make this edition current in my session:

```
ALTER SESSION SET edition = HR_PATCH_NAMEFORMAT
/
```

Since this edition was based on ora$base, it inherits all the objects defined in that parent edition. I can, therefore, still call my function and get the same answer as before:

```
SQL> BEGIN   DBMS_OUTPUT.put_line (full_name ('Steven', 'Feuerstein'));END;/
Steven Feuerstein
```

Now I change the implementation of this function to reflect the new rule:

```
CREATE OR REPLACE FUNCTION full_name (first_in IN employees.first_name%TYPE
                                    , last_in IN employees.first_name%TYPE
                                     )
   RETURN VARCHAR2
IS
BEGIN
   RETURN (last_in || ', ' || first_in);
END full_name;
/
```

Now when I run the function, I see a different result:

```
SQL> BEGIN   DBMS_OUTPUT.put_line (full_name ('Steven', 'Feuerstein'));END;/
Feuerstein, Steven
```

But if I change the edition back to the base edition, I see my old format:

```
SQL> ALTER SESSION SET edition = ora$base/
2 BEGIN   DBMS_OUTPUT.put_line (full_name ('Steven', 'Feuerstein'));END;/
Steven Feuerstein
```

That's the basic idea behind edition-based redefinition, but of course your application architect and your development team will need to explore the many aspects of this feature—especially cross-edition triggers and editioning views, both of which are needed when you change the structure of a table (which is *not* directly editionable).

You will find extensive documentation on edition-based redefinition in the Oracle's *Advanced Application Developer's Guide*.