**CHAPTER 11  
Introducing PL/SQL Programming**

Oracle added a procedural programming language known as PL/SQL (Procedural Language/SQL) to Oracle Database 6. PL/SQL enables you to write programs that contain SQL statements. In this chapter, you’ll learn about the following PL/SQL topics:

image Block structure

image Variables and types

image Conditional logic

image Loops

image Cursors, which allow PL/SQL to read the results returned by a query

image Procedures

image Functions

image Packages, which are used to group procedures and functions together in one unit

image Triggers, which are blocks of code that are run when a certain event occurs in the database

image Oracle Database 11*g* enhancements to PL/SQL

You can use PL/SQL to add business logic to a database application. This centralized business logic can be run by any program that can access the database, including SQL\*Plus, Java programs, C# programs, and more.

image

**NOTE**  
*For full details on how to access a database through Java, see my book* Oracle9*i* JDBC Programming *(Oracle Press, 2002). For details on how to access a database through C#, see my book* Mastering C# Database Programming *(Sybex, 2003)*.

**BLOCK STRUCTURE**

PL/SQL programs are divided up into structures known as *blocks*, with each block containing PL/SQL and SQL statements. A PL/SQL block has the following structure:

[DECLARE

*declaration\_statements*

]

BEGIN

*executable\_statements*

[EXCEPTION

*exception\_handling\_statements*

]

END;

/

where

image *declaration\_statements* declare the variables used in the rest of the PL/SQL block. DECLARE blocks are optional.

image *executable\_statements* are the actual executable statements, which may include loops, conditional logic, and so on.

image *exception\_handling\_statements* are statements that handle any execution errors that might occur when the block is run. EXCEPTION blocks are optional.

Every statement is terminated by a semicolon (;), and a PL/SQL block is terminated using the forward slash (/)character. Before I get into the details of PL/SQL, you’ll see a simple example to get a feel for the language. The following example (contained in the area\_example.sql script in the SQL directory) calculates the width of a rectangle given its area and height:

SET SERVEROUTPUT ON

DECLARE

v\_width INTEGER;

v\_height INTEGER := 2;

v\_area INTEGER := 6;

BEGIN

-- set the width equal to the area divided by the height

v\_width := v\_area / v\_height;

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

EXCEPTION

WHEN ZERO\_DIVIDE THEN

DBMS\_OUTPUT.PUT\_LINE(’Division by zero');

END;

/

The SET SERVEROUTPUT ON command turns the server output on so you can see the lines produced by DBMS\_OUTPUT.PUT\_LINE() on the screen when you run the script in SQL\*Plus. After this initial command comes the PL/SQL block itself, which is divided into the DECLARE, BEGIN, and EXCEPTION blocks.

The DECLARE block contains declarations for three INTEGER variables named v\_width, v\_ height, and v\_area (I always put v\_ at the start of variable names). The v\_height and v\_area variables are initialized to 2 and 6 respectively.

Next comes the BEGIN block, which contains three lines. The first line is a comment that contains the text "set the width equal to the area divided by the height." The second line sets v\_width to v\_area divided by v\_height; this means v\_width is set to 3 (= 6 / 2). The third line calls DBMS\_OUTPUT.PUT\_LINE() to display the value of v\_width on the screen. DBMS\_OUTPUT is a built-in package of code that comes with the Oracle database; among other items, DBMS\_OUTPUT contains procedures that allow you to output values to the screen.

Next, the EXCEPTION block handles any attempts to divide a number by zero. It does this by "catching" the ZERO\_DIVIDE exception; in the example, no attempt is actually made to divide by zero, but if you change v\_height to 0 and run the script you’ll see the exception.

At the very end of the script, the forward slash character (/) marks the end of the PL/SQL block.

The following listing shows the execution of the area\_example.sql script in SQL\*Plus:

SQL> **@ C:\SQL\area\_example.sql**

v\_width = 3

image

**NOTE**  
*If your* area\_example.sql *script is in a directory other than* C:\SQL, *use your own directory in the previous command*.

**VARIABLES AND TYPES**

Variables are declared within a DECLARE block. As you saw in the previous example, a variable declaration has both a name and a type. For example, the v\_width variable was declared as

v\_width INTEGER;

image

**NOTE**  
*The PL/SQL types are similar to the database column types. You can see all the types in the appendix*.

The following example shows more variable declarations (these variables could be used to store the column values from the products table):

v\_product\_id INTEGER;

v\_product\_type\_id INTEGER;

v\_name VARCHAR2(30);

v\_description VARCHAR2(50);

v\_price NUMBER(5, 2);

You may also specify a variable’s type using the %TYPE keyword, which tells PL/SQL to use the same type as a specified column in a table. The following example uses %TYPE to declare a variable of the same type as the price column of the products table, which is NUMBER(5, 2):

v\_product\_price products.price%TYPE;

**CONDITIONAL LOGIC**

You use the IF, THEN, ELSE, ELSIF, and END IF keywords to perform conditional logic:

*IF condition1* THEN

*statements1*

ELSIF *condition2* THEN

*statements2*

ELSE

*statements3*

END IF;

where

image *condition1* and *condition2* are Boolean expressions that evaluate to true or false.

image *statements1*, *statements2*, and *statements3* are PL/SQL statements.

The conditional logic flows as follows:

image If *condition1* is true, then *statements1* are executed.

image If *condition1* is false but *condition2* is true, then *statements2* are executed.

image If neither *condition1* nor *condition2* is true, then *statements3* are executed.

You can also embed an IF statement within another IF statement, as shown in the following example:

IF v\_count > 0 THEN

v\_message := 'v\_count is positive';

IF v\_area > 0 THEN

v\_message := 'v\_count and v\_area are positive';

END IF

ELSIF v\_count = 0 THEN

v\_message := 'v\_count is zero';

ELSE

v\_message := 'v\_count is negative';

END IF;

In this example, if v\_count is greater than 0, then v\_message is set to 'v\_count is positive'. If v\_count and v\_area are greater than 0, then v\_message is set to 'v\_count and v\_area are positive'. The rest of the logic is straightforward.

**LOOPS**

You use a loop to run statements zero or more times. There are three types of loops in PL/SQL:

image **Simple loops** run until you explicitly end the loop.

image **WHILE loops** run until a specified condition occurs.

image **FOR loops** run a predetermined number of times.

You’ll learn about these loops in the following sections.

**Simple Loops**

A simple loop runs until you explicitly end the loop. The syntax for a simple loop is as follows:

LOOP

*statements*

END LOOP;

To end the loop, you use either an EXIT or an EXIT WHEN statement. The EXIT statement ends a loop immediately; the EXIT WHEN statement ends a loop when a specified condition occurs.

The following example shows a simple loop. A variable named v\_counter is initialized to 0 prior to the beginning of the loop. The loop adds 1 to v\_counter and exits when v\_counter is equal to 5 using an EXIT WHEN statement.

v\_counter := 0;

LOOP

v\_counter := v\_counter + 1;

EXIT WHEN v\_counter = 5;

END LOOP;

image

**NOTE**  
*The* EXIT WHEN *statement can appear anywhere in the loop code*.

In Oracle Database 11*g* you can also end the current iteration of a loop using the CONTINUE or CONTINUE WHEN statement. The CONTINUE statement ends the current iteration of the loop unconditionally and continues with the next iteration; the CONTINUE WHEN statement ends the current iteration of the loop when a specified condition occurs and then continues with the next iteration. The following example shows the use of the CONTINUE statement:

v\_counter := 0;

LOOP

-- after the CONTINUE statement is executed, control returns here

v\_counter := v\_counter + 1;

IF v\_counter = 3 THEN

CONTINUE; -- end current iteration unconditionally

END IF;

EXIT WHEN v\_counter = 5;

END LOOP;

The next example shows the use of the CONTINUE WHEN statement:

v\_counter := 0;

LOOP

-- after the CONTINUE WHEN statement is executed, control returns here

v\_counter := v\_counter + 1;

CONTINUE WHEN v\_counter = 3; -- end current iteration when v\_counter = 3

EXIT WHEN v\_counter = 5;

END LOOP;

image

**NOTE**  
*A* CONTINUE *or* CONTINUE WHEN *statement cannot cross a procedure, function, or method boundary*.

**WHILE Loops**

A WHILE loop runs until a specified condition occurs. The syntax for a WHILE loop is as follows:

WHILE condition LOOP

statements

END LOOP;

The following example shows a WHILE loop that executes while the v\_counter variable is less than 6:

v\_counter := 0;

WHILE v\_counter < 6 LOOP

v\_counter := v\_counter + 1;

END LOOP;

**FOR Loops**

A FOR loop runs a predetermined number of times; you determine the number of times the loop runs by specifying the *lower* and *upper bounds* for a loop variable. The loop variable is then incremented (or decremented) each time around the loop. The syntax for a FOR loop is as follows:

FOR *loop\_variable* IN [REVERSE] *lower\_bound..upper\_bound* LOOP

*statements*

END LOOP;

where

image *loop\_variable* is the loop variable. You can use a variable that already exists as the loop variable, or you can just have the loop create a new variable for you (this occurs if the variable you specify doesn’t exist). The loop variable value is increased (or decreased if you use the REVERSE keyword) by 1 each time through the loop.

image REVERSE means that the loop variable value is to be decremented each time through the loop. The loop variable is initialized to the upper boundary, and is decremented by 1 until the loop variable reaches the lower boundary. You must specify the lower boundary before the upper boundary.

image *lower\_bound* is the loop’s lower boundary. The loop variable is initialized to this lower boundary provided REVERSE is not used.

image *upper\_bound* is the loop’s upper boundary. If REVERSE is used, the loop variable is initialized to this upper boundary.

The following example shows a FOR loop. Notice that the variable v\_counter2 isn’t explicitly declared—so the FOR loop automatically creates a new INTEGER variable named v\_counter2:

FOR v\_counter2 IN 1..5 LOOP

DBMS\_OUTPUT.PUT\_LINE(v\_counter2);

END LOOP;

The following example uses REVERSE:

FOR v\_counter2 IN REVERSE 1..5 LOOP

DBMS\_OUTPUT.PUT\_LINE(v\_counter2);

END LOOP;

In this example, v\_counter2 starts at 5, is decremented by 1 each time through the loop, and ends at 1.

**CURSORS**

You use a *cursor* to fetch rows returned by a query. You retrieve the rows into the cursor using a query and then fetch the rows one at a time from the cursor. You typically use the following five steps when using a cursor:

**1.** Declare variables to store the column values for a row.

**2.** Declare the cursor, which contains a query.

**3.** Open the cursor.

**4.** Fetch the rows from the cursor one at a time and store the column values in the variables declared in Step 1. You would then do something with those variables, such as display them on the screen, use them in a calculation, and so on.

**5.** Close the cursor.

You’ll learn the details of these five steps in the following sections, and you’ll see a simple example that gets the product\_id, name, and price columns from the products table.

**Step 1: Declare the Variables to Store the Column Values**

The first step is to declare the variables that will be used to store the column values. These variables must be compatible with the column types.

image

**TIP**  
*Earlier you saw that* %TYPE *may be used to get the type of a column. If you use* %TYPE *when declaring your variables, your variables will automatically be of the correct type*.

The following example declares three variables to store the product\_id, name, and price columns from the products table; notice that %TYPE is used to automatically set the type of the variables to the same type as the columns:

DECLARE

v\_product\_id products.product\_id%TYPE;

v\_name products.name%TYPE;

v\_price products.price%TYPE;

**Step 2: Declare the Cursor**

Step 2 is to declare the cursor. A cursor declaration consists of a name that you assign to the cursor and the query you want to execute. The cursor declaration, like all other declarations, is placed in the declaration section. The syntax for declaring a cursor is as follows:

CURSOR *cursor\_name* IS

*SELECT\_statement;*

where

image *cursor\_name* is the name of the cursor.

image *SELECT\_statement* is the query.

The following example declares a cursor named v\_product\_cursor whose query retrieves the product\_id, name, and price columns from the products table:

CURSOR v\_product\_cursor IS

SELECT product\_id, name, price

FROM products

ORDER BY product\_id;

The query isn’t executed until you open the cursor.

**Step 3: Open the Cursor**

Step 3 is to open the cursor. You open a cursor using the OPEN statement, which must be placed in the executable section of the block.

The following example opens v\_product\_cursor, which executes the query:

OPEN v\_product\_cursor;

**Step 4: Fetch the Rows from the Cursor**

Step 4 is to fetch the rows from the cursor, which you do using the FETCH statement. The FETCH statement reads the column values into the variables declared in Step 1. FETCH uses the following syntax:

FETCH *cursor\_name*

INTO *variable*[, *variable* …];

where

image *cursor\_name* is the name of the cursor.

image *variable* is the variable into which a column value from the cursor is stored. You need to provide matching variables for each column value.

The following FETCH example retrieves a row from v\_product\_cursor and stores the column values in the v\_product\_id, v\_name, and v\_price variables created earlier in Step 1:

FETCH v\_product\_cursor

INTO v\_product\_id, v\_name, v\_price;

Because a cursor may contain many rows, you need a loop to read them. To figure out when to end the loop, you can use the Boolean variable v\_product\_cursor%NOTFOUND. This variable is true when there are no more rows to read in v\_product\_cursor. The following example shows a loop:

LOOP

-- fetch the rows from the cursor

FETCH v\_product\_cursor

INTO v\_product\_id, v\_name, v\_price;

-- exit the loop when there are no more rows, as indicated by

-- the Boolean variable v\_product\_cursor%NOTFOUND (= true when

-- there are no more rows)

EXIT WHEN v\_product\_cursor%NOTFOUND;

-- use DBMS\_OUTPUT.PUT\_LINE() to display the variables

DBMS\_OUTPUT.PUT\_LINE(

'v\_product\_id = ' || v\_product\_id || ', v\_name = ' || v\_name ||

', v\_price = ' || v\_price

);

END LOOP;

Notice that I’ve used DBMS\_OUTPUT.PUT\_LINE() to display the v\_product\_id, v\_name, and v\_pricevariables that were read for each row. In a real application, you might use v\_price in a complex calculation.

**Step 5: Close the Cursor**

Step 5 is to close the cursor using the CLOSE statement. Closing a cursor frees up system resources. The following example closes v\_product\_cursor:

CLOSE v\_product\_cursor;

The following section shows a complete script that contains all five steps.

**COMPLETE EXAMPLE: PRODUCT\_CURSOR.SQL**

The following product\_cursor.sql script is contained in the SQL directory:

-- product\_cursor.sql displays the product\_id, name,

-- and price columns from the products table using a cursor

SET SERVEROUTPUT ON

DECLARE

-- step 1: declare the variables

v\_product\_id products.product\_id%TYPE;

v\_name products.name%TYPE;

v\_price products.price%TYPE;

-- step 2: declare the cursor

CURSOR v\_product\_cursor IS

SELECT product\_id, name, price

FROM products

ORDER BY product\_id;

BEGIN

-- step 3: open the cursor

OPEN v\_product\_cursor;

LOOP

-- step 4: fetch the rows from the cursor

FETCH v\_product\_cursor

INTO v\_product\_id, v\_name, v\_price;

-- exit the loop when there are no more rows, as indicated by

-- the Boolean variable v\_product\_cursor%NOTFOUND (= true when

-- there are no more rows)

EXIT WHEN v\_product\_cursor%NOTFOUND;

-- use DBMS\_OUTPUT.PUT\_LINE() to display the variables

DBMS\_OUTPUT.PUT\_LINE(

'v\_product\_id = ' || v\_product\_id || ', v\_name = ' || v\_name ||

', v\_price = ' || v\_price

);

END LOOP;

-- step 5: close the cursor

CLOSE v\_product\_cursor;

END;

/

To run this script, follow these steps:

**1.** Connect to the database as store with the password store\_password.

**2.** Run the product\_cursor.sql script using SQL\*Plus:

SQL> **@ C:\SQL\product\_cursor.sql**

image

**NOTE**  
*If your* product\_cursor.sql *script is in a different directory from* C:\SQL, *use your own directory in the previous command*.

The output from product\_cursor.sql is as follows:

v\_product\_id = 1, v\_name = Modern Science, v\_price = 19.95

v\_product\_id = 2, v\_name = Chemistry, v\_price = 30

v\_product\_id = 3, v\_name = Supernova, v\_price = 25.99

v\_product\_id = 4, v\_name = Tank War, v\_price = 13.95

v\_product\_id = 5, v\_name = Z Files, v\_price = 49.99

v\_product\_id = 6, v\_name = 2412: The Return, v\_price = 14.95

v\_product\_id = 7, v\_name = Space Force 9, v\_price = 13.49

v\_product\_id = 8, v\_name = From Another Planet, v\_price = 12.99

v\_product\_id = 9, v\_name = Classical Music, v\_price = 10.99

v\_product\_id = 10, v\_name = Pop 3, v\_price = 15.99

v\_product\_id = 11, v\_name = Creative Yell, v\_price = 14.99

v\_product\_id = 12, v\_name = My Front Line, v\_price = 13.49

**CURSORS AND FOR LOOPS**

You can use a FOR loop to access the rows in a cursor. When you do this, you don’t have to explicitly open and close the cursor—the FOR loop does this automatically for you. The following product\_cursor2.sql script uses a FOR loop to access the rows in v\_product\_cursor; notice that this script contains less code than product\_cursor.sql:

-- product\_cursor2.sql displays the product\_id, name,

-- and price columns from the products table using a cursor

-- and a FOR loop

SET SERVEROUTPUT ON

DECLARE

CURSOR v\_product\_cursor IS

SELECT product\_id, name, price

FROM products

ORDER BY product\_id;

BEGIN

FOR v\_product IN v\_product\_cursor LOOP

DBMS\_OUTPUT.PUT\_LINE(

'product\_id = ' || v\_product.product\_id ||

', name = ' || v\_product.name ||

', price = ' || v\_product.price

);

END LOOP;

END;

/

To run the product\_cursor2.sql script, you issue a command similar to the following:

SQL> **@ "C:\SQL\product\_cursor2.sql"**

The output from this script is as follows:

product\_id = 1, name = Modern Science, price = 19.95

product\_id = 2, name = Chemistry, price = 30

product\_id = 3, name = Supernova, price = 25.99

product\_id = 4, name = Tank War, price = 13.95

product\_id = 5, name = Z Files, price = 49.99

product\_id = 6, name = 2412: The Return, price = 14.95

product\_id = 7, name = Space Force 9, price = 13.49

product\_id = 8, name = From Another Planet, price = 12.99

product\_id = 9, name = Classical Music, price = 10.99

product\_id = 10, name = Pop 3, price = 15.99

product\_id = 11, name = Creative Yell, price = 14.99

product\_id = 12, name = My Front Line, price = 13.49

**OPEN-FOR Statement**

You may also use the OPEN-FOR statement with a cursor, which adds even more flexibility when processing cursors because you can assign the cursor to a different query. This is shown in the following product\_cursor3.sql script:

-- product\_cursor3.sql displays the product\_id, name,

-- and price columns from the products table using a cursor

-- variable and the OPEN-FOR statement

SET SERVEROUTPUT ON

DECLARE

-- declare a REF CURSOR type named t\_product\_cursor

TYPE t\_product\_cursor IS

REF CURSOR RETURN products%ROWTYPE;

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

v\_product\_cursor t\_product\_cursor;

-- declare an object to store columns from the products table

-- named v\_product (of type products%ROWTYPE)

v\_product products%ROWTYPE;

BEGIN

-- assign a query to v\_product\_cursor and open it using OPEN-FOR

OPEN v\_product\_cursor FOR

SELECT \* FROM products WHERE product\_id < 5;

-- use a loop to fetch the rows from v\_product\_cursor into v\_product

LOOP

FETCH v\_product\_cursor INTO v\_product;

EXIT WHEN v\_product\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE(

'product\_id = ' || v\_product.product\_id ||

', name = ' || v\_product.name ||

', price = ' || v\_product.price

);

END LOOP;

-- close v\_product\_cursor

CLOSE v\_product\_cursor;

END;

/

In the DECLARE block, the following statement declares a REF CURSOR type named t\_product\_cursor (I always put t\_ at the start of type names):

TYPE t\_product\_cursor IS

REF CURSOR RETURN products%ROWTYPE;

A REF CURSOR is a pointer to a cursor, and is similar to a pointer in the C++ programming language. The previous statement declares a user-defined type named t\_product\_cursor, and returns a row containing the various columns of the products table (this is indicated using %ROWTYPE). This user-defined type may be used to declare an actual object, as shown in the following statement, which declares an object named v\_product\_cursor:

v\_product\_cursor t\_product\_cursor;

The following statement declares an object to store columns from the products table named v\_product (of type products%ROWTYPE):

v\_product products%ROWTYPE;

In the BEGIN block, v\_product\_cursor is assigned a query and opened by the following OPEN-FORstatement:

OPEN v\_product\_cursor FOR

SELECT \* FROM products WHERE product\_id < 5;

After this statement is executed, v\_product\_cursor will be loaded with the first four rows in the products table. The query assigned to v\_product\_cursor can be any valid SELECT statement; this means you can re-use the cursor and assign another query to the cursor later in the PL/SQL code.

Next, the following loop fetches the rows from v\_product\_cursor into v\_product and displays the row details:

LOOP

FETCH v\_product\_cursor INTO v\_product;

EXIT WHEN v\_product\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE(

'product\_id = ' || v\_product.product\_id ||

', name = ' || v\_product.name ||

', price = ' || v\_product.price

);

END LOOP;

After the loop, v\_product\_cursor is closed using the following statement:

CLOSE v\_product\_cursor;

The output from this script is the same as the output from product\_cursor2.sql.

**Unconstrained Cursors**

The cursors in the previous section all have a specific return type; these cursors are known as constrained cursors. The return type for a constrained cursor must match the columns in the query that is run by the cursor. An unconstrained cursor has no return type, and can therefore run any query.

The use of an unconstrained cursor is shown in the following unconstrained\_cursor .sql script; notice v\_cursor in the code is used to run two different queries:

-- This script shows the use of unconstrained cursors

SET SERVEROUTPUT ON

DECLARE

-- declare a REF CURSOR type named t\_cursor (this has no return

-- type and can therefore run any query)

TYPE t\_cursor IS REF CURSOR;

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

v\_cursor t\_cursor;

-- declare an object to store columns from the products table

-- named v\_product (of type products%ROWTYPE)

v\_product products%ROWTYPE;

-- declare an object to store columns from the customers table

-- named v\_customer (of type customers%ROWTYPE)

v\_customer customers%ROWTYPE;

BEGIN

-- assign a query to v\_cursor and open it using OPEN-FOR

OPEN v\_cursor FOR

SELECT \* FROM products WHERE product\_id < 5;

-- use a loop to fetch the rows from v\_cursor into v\_product

LOOP

FETCH v\_cursor INTO v\_product;

EXIT WHEN v\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE(

' product\_id = ' || v\_product.product\_id ||

', name = ' || v\_product.name ||

', price = ' || v\_product.price

);

END LOOP;

-- assign a new query to v\_cursor and open it using OPEN-FOR

OPEN v\_cursor FOR

SELECT \* FROM customers WHERE customer\_id < 3;

-- use a loop to fetch the rows from v\_cursor into v\_product

LOOP

FETCH v\_cursor INTO v\_customer;

EXIT WHEN v\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE(

'customer\_id = ' || v\_customer.customer\_id ||

', first\_name = ' || v\_customer.first\_name ||

', last\_name = ' || v\_customer.last\_name

);

END LOOP;

-- close v\_cursor

CLOSE v\_cursor;

END;

/

To run the unconstrained\_cursor.sql script, you issue a command similar to the following:

SQL> **@ "C:\SQL\unconstrained\_cursor.sql"**

The output from this script is as follows:

product\_id = 1, name = Modern Science, price = 19.95

product\_id = 2, name = Chemistry, price = 30

product\_id = 3, name = Supernova, price = 25.99

product\_id = 4, name = Tank War, price = 13.95

customer\_id = 1, first\_name = John, last\_name = Brown

customer\_id = 2, first\_name = Cynthia, last\_name = Green

You’ll learn more about REF CURSOR variables later in this chapter and more about user-defined types in the next chapter.

**EXCEPTIONS**

Exceptions are used to handle run-time errors in your PL/SQL code. Earlier, you saw the following PL/SQL example that contains an EXCEPTION block:

DECLARE

v\_width INTEGER;

v\_height INTEGER := 2;

v\_area INTEGER := 6;

BEGIN

-- set the width equal to the area divided by the height

v\_width := v\_area / v\_height;

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

EXCEPTION

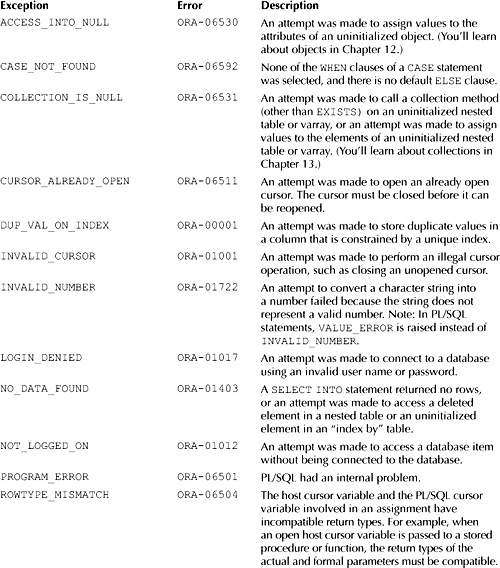
WHEN ZERO\_DIVIDE THEN

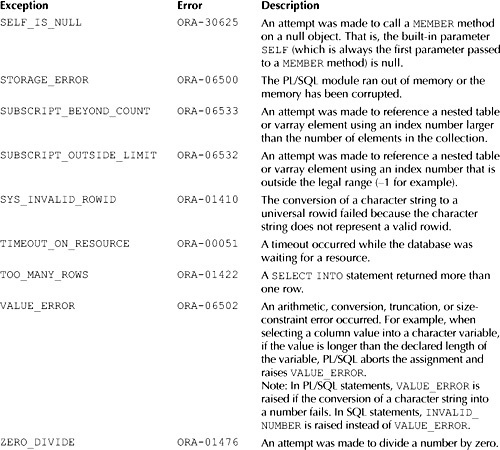
DBMS\_OUTPUT.PUT\_LINE(’Division by zero');

END;

/

The EXCEPTION block in this example handles an attempt to divide a number by zero. In PL/SQL terminology, the EXCEPTION block *catches* a ZERO\_DIVIDE exception that is *raised* in the BEGIN block (although in the example code, ZERO\_DIVIDE is never actually raised). The ZERO\_DIVIDE exception and the other common exceptions are shown in [Table 11-1](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch11.html#table_11-1).





**TABLE 11-1** *Predefined Exceptions*

The following sections show examples that raise some of the exceptions shown in [Table 11-1](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch11.html#table_11-1).

**ZERO\_DIVIDE Exception**

The ZERO\_DIVIDE exception is raised when an attempt is made to divide a number by zero. The following example attempts to divide 1 by 0 in the BEGIN block and therefore raises the ZERO\_DIVIDE exception:

**BEGIN**

**DBMS\_OUTPUT.PUT\_LINE(1 / 0);**

**EXCEPTION**

**WHEN ZERO\_DIVIDE THEN**

**DBMS\_OUTPUT.PUT\_LINE(’Division by zero');**

**END;**

**/**

Division by zero

When an exception is raised, program control passes to the EXCEPTION block and the WHEN clause is examined for a matching exception; the code inside the matching clause is then executed. In the previous example, the ZERO\_DIVIDE exception is raised in the BEGIN block, and program control then passes to the EXCEPTIONblock; a matching exception is found in the WHEN clause, and the code inside the clause is executed.

If no matching exception is found, the exception is propagated to the enclosing block. For example, if the EXCEPTION block was omitted from the previous code, the exception is propagated up to SQL\*Plus:

**BEGIN**

**DBMS\_OUTPUT.PUT\_LINE(1 / 0);**

**END;**

BEGIN

\*

ERROR at line 1:

ORA-01476: divisor is equal to zero

ORA-06512: at line 2

As you can see, SQL\*Plus displays a default error that shows the line numbers, the Oracle error codes, and a simple description.

**DUP\_VAL\_ON\_INDEX Exception**

The DUP\_VAL\_ON\_INDEX exception is raised when an attempt is made to store duplicate values in a column that is constrained by a unique index. The following example attempts to insert a row in the customers table with a customer\_id of 1; this causes DUP\_VAL\_ON\_INDEX to be raised, because the customers table already contains a row with a customer\_id of 1:

**BEGIN**

**INSERT INTO customers (**

**customer\_id, first\_name, last\_name**

**) VALUES (**

**1, 'Greg', 'Green'**

**);**

**EXCEPTION**

**WHEN DUP\_VAL\_ON\_INDEX THEN**

**DBMS\_OUTPUT.PUT\_LINE(’Duplicate value on an index');**

**END;**

**/**

Duplicate value on an index

**INVALID\_NUMBER Exception**

The INVALID\_NUMBER exception is raised when an attempt is made to convert an invalid character string into a number. The following example attempts to convert the string 123X to a number that is used in an INSERT, which causes INVALID\_NUMBER to be raised because 123X is not a valid number:

**BEGIN**

**INSERT INTO customers (**

**customer\_id, first\_name, last\_name**

**) VALUES (**

**'123X', 'Greg', 'Green'**

**);**

**EXCEPTION**

**WHEN INVALID\_NUMBER THEN**

**DBMS\_OUTPUT.PUT\_LINE('Conversion of string to number failed');**

**END;**

**/**

Conversion of string to number failed

**OTHERS Exception**

You can use the OTHERS exception to handle all exceptions, as shown here:

**BEGIN**

**DBMS\_OUTPUT.PUT\_LINE(1 / 0);**

**EXCEPTION**

**WHEN OTHERS THEN**

**DBMS\_OUTPUT.PUT\_LINE('An exception occurred');**

**END;**

**/**

An exception occurred

Because OTHERS matches all exceptions, you must list it after any specific exceptions in your EXCEPTIONblock. If you attempt to list OTHERS elsewhere, the database returns the error PLS-00370; for example:

SQL> **BEGIN**

2 **DBMS\_OUTPUT.PUT\_LINE(1 / 0);**

3 **EXCEPTION**

4 **WHEN OTHERS THEN**

5 **DBMS\_OUTPUT.PUT\_LINE('An exception occurred');**

6 **WHEN ZERO\_DIVIDE THEN**

7 **DBMS\_OUTPUT.PUT\_LINE(’Division by zero');**

8 **END;**

9 **/**

WHEN OTHERS THEN

\*

ERROR at line 4:

ORA-06550: line 4, column 3:

PLS-00370: OTHERS handler must be last among the exception

handlers of a block

ORA-06550: line 0, column 0:

PL/SQL: Compilation unit analysis terminated

**PROCEDURES**

A procedure contains a group of SQL and PL/SQL statements. Procedures allow you to centralize your business logic in the database and may be used by any program that accesses the database.

In this section, you’ll learn how to

image Create a procedure.

image Call a procedure.

image Get information on procedures.

image Drop a procedure.

image View errors in a procedure.

**Creating a Procedure**

You create a procedure using the CREATE PROCEDURE statement. The simplified syntax for the CREATE PROCEDURE statement is as follows:

CREATE [OR REPLACE] PROCEDURE *procedure\_name*

[(*parameter\_name* [IN | OUT | IN OUT] *type* [, …])]

{IS | AS}

BEGIN

*procedure\_body*

END *procedure\_name;*

where

image OR REPLACE means the procedure is to replace an existing procedure.

image *procedure\_name* is the name of the procedure.

image *procedure\_name* is the name of the procedure.

image *parameter\_name* is the name of a parameter that is passed to the procedure. You may pass multiple parameters to a procedure.

image IN | OUT | IN OUT is the *mode* of the parameter. You may pick one of the following modes for each parameter:

image IN, which is the default mode for a parameter. An IN parameter must be set to a value when the procedure is run. The value of an IN parameter cannot be changed in the procedure body.

image OUT, which means the parameter is set to a value in the procedure body.

image IN OUT, which means the parameter can have a value when the procedure is run, and the value can be changed in the body.

image *type* is the type of the parameter.

image *procedure\_body* contains the actual code for the procedure.

The following example creates a procedure named update\_product\_price()—this procedure, and the other PL/SQL code shown in the rest of this chapter, was created when you ran the store\_schema.sqlscript. The update\_product\_price() procedure multiplies the price of a product by a factor; the product ID and the factor are passed as parameters to the procedure. If the product exists, the procedure multiplies the product price by the factor and commits the change.

CREATE PROCEDURE update\_product\_price(

p\_product\_id IN products.product\_id%TYPE,

p\_factor IN NUMBER

) AS

v\_product\_count INTEGER;

BEGIN

-- count the number of products with the

-- supplied product\_id (will be 1 if the product exists)

SELECT COUNT(\*)

INTO v\_product\_count

FROM products

WHERE product\_id = p\_product\_id;

-- if the product exists (v\_product\_count = 1) then

-- update that product’s price

IF v\_product\_count = 1 THEN

UPDATE products

SET price = price \* p\_factor

WHERE product\_id = p\_product\_id;

COMMIT;

END IF;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

END update\_product\_price;

/

The procedure accepts two parameters named p\_product\_id and p\_factor (I always put p\_ at the start of parameter names). Both of these parameters use the IN mode, which means that their values must be set when the procedure is run and that the parameter values cannot be changed in the procedure body.

The declaration section contains an INTEGER variable named v\_product\_count:

v\_product\_count INTEGER;

The body of the procedure starts after BEGIN. The SELECT statement in the body gets the number of rows from the products table whose product\_id is equal to p\_product\_id:

SELECT COUNT(\*)

INTO v\_product\_count

FROM products

WHERE product\_id = p\_product\_id;

image

**NOTE**  
COUNT(\*) *returns number of rows found*.

If the product is found, v\_product\_count will be set to 1; otherwise, v\_product\_count will be set to 0. If v\_product\_count is 1, the price column is multiplied by p\_factor using the UPDATE statement, and the change is committed:

IF v\_product\_count = 1 THEN

UPDATE products

SET price = price \* p\_factor

WHERE product\_id = p\_product\_id;

COMMIT;

END IF;

The EXCEPTION block performs a ROLLBACK if an exception is raised:

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

Finally, the END keyword is used to mark the end of the procedure:

END update\_product\_price;

/

image

**NOTE**  
*The repetition of the procedure name after the* END *keyword is not required, but it is good programming practice to put it in*.

**Calling a Procedure**

You run (or *call*) a procedure using the CALL statement. The example you’ll see in this section will multiply the price of product #1 by 1.5 using the procedure shown in the previous section. First, the following query retrieves the price of product #1 so you can compare it with the modified price later:

**SELECT price**

**FROM products**

**WHERE product\_id = 1;**

PRICE

----------

19.95

The following statement calls update\_product\_price(), passing the parameter values 1 (the product\_id) and 1.5 (the factor by which the product price is multiplied):

**CALL update\_product\_price(1, 1.5);**

Call completed.

This statement shows the use of *positional notation* to indicate the values to be passed to the procedure or function. In positional notation, the position of parameters is used to assign the values passed to the procedure. In the example, the first value in the call is 1, and this is passed to the first parameter in the procedure (p\_product\_id); the second value in the call is 1.5, and this is passed to the second parameter (p\_factor). In Oracle Database 11*g*, you can also use named and mixed notation in addition to positional notation, and you’ll learn about these types of notation shortly.

The next query retrieves the details for product #1 again; notice the price has been multiplied by 1.5:

**SELECT price**

**FROM products**

**WHERE product\_id = 1;**

PRICE

----------

29.93

In Oracle Database 11*g* you can pass parameters using named and mixed notation. In *named notation*, you include the name of the parameter when calling a procedure. For example, the following statement calls update\_product\_price() using named notation; notice that the values for the p\_factor and p\_product\_id parameters are indicated using =>:

**CALL update\_product\_price(p\_factor => 1.3, p\_product\_id => 2);**

image

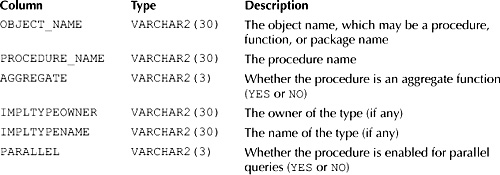
**TIP**  
*Named notation makes your code easier to read and maintain because the parameters are explicitly shown*.

In *mixed notation*, you use both positional and named notation; you use positional notation for the first set of parameters and named notation for the last set of parameters. Mixed notation is useful when you have procedures and functions that have both required and optional parameters; you use positional notation for the required parameters, and named notation for the optional parameters. The following example uses mixed notation; notice that positional notation comes before named notation when specifying the parameter values:

**CALL update\_product\_price(3, p\_factor => 1.7);**

**Getting Information on Procedures**

You can get information on your procedures from the user\_procedures view. [Table 11-2](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch11.html#table_11-2) describes some of the columns in user\_procedures.



**TABLE 11-2** *Some Columns in the user\_procedures View*

image

**NOTE**  
*You can get information on all the procedures you have access to using* all\_procedures.

The following example retrieves the object\_name, aggregate, and parallel columns from user\_procedures for update\_product\_price():

**SELECT object\_name, aggregate, parallel**

**FROM user\_procedures**

**WHERE object\_name = 'UPDATE\_PRODUCT\_PRICE';**

OBJECT\_NAME AGG PAR

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

UPDATE\_PRODUCT\_PRICE NO NO

**Dropping a Procedure**

You drop a procedure using DROP PROCEDURE. For example, the following statement drops update\_product\_price():

DROP PROCEDURE update\_product\_price;

**Viewing Errors in a Procedure**

If the database reports an error when you create a procedure, you can view the errors using the SHOW ERRORScommand. For example, the following CREATE PROCEDURE statement attempts to create a procedure that has a syntax error at line 6 (the parameter should be p\_dob, not p\_dobs):

SQL> **CREATE PROCEDURE update\_customer\_dob (**

2 **p\_customer\_id INTEGER, p\_dob DATE**

3 **) AS**

4 **BEGIN**

5 **UPDATE customers**

6 **SET dob = p\_dobs**

7 **WHERE customer\_id = p\_customer\_id;**

8 **END update\_customer\_dob;**

9 **/**

Warning: Procedure created with compilation errors.

As you can see, there is a compilation error. To view the errors, you use SHOW ERRORS:

SQL> **SHOW ERRORS**

Errors for PROCEDURE UPDATE\_CUSTOMER\_DOB:

LINE/COL ERROR

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

5/3 PL/SQL: SQL Statement ignored

6/13 PL/SQL: ORA-00904: invalid column name

Line 5 was ignored because an invalid column name was referenced in line 6. You can fix the error by issuing an EDIT command to edit the CREATE PROCEDURE statement, changing p\_dobs to p\_dob, and rerunning the statement by entering/.

**FUNCTIONS**

A *function* is similar to a procedure, except that a function must return a value. Together, stored procedures and functions are sometimes referred to as *stored subprograms* because they are small programs.

In this section, you’ll learn how to

image Create a function.

image Call a function.

image Get information on functions.

image Drop a function.

**Creating a Function**

You create a function using the CREATE FUNCTION statement. The simplified syntax for the CREATE FUNCTION statement is as follows:

CREATE [OR REPLACE] FUNCTION *function\_name*

[(*parameter\_name* [IN | OUT | IN OUT] *type* [, …])]

RETURN *type*

{IS | AS}

BEGIN

*function\_body*

END *function\_name;*

where

image OR REPLACE means the procedure is to replace an existing function.

image *function\_name* is the name of the function.

image *parameter\_name* is the name of a parameter that is passed to the function. You may pass multiple parameters to a function.

image IN |OUT | IN OUT is the mode of the parameter.

image *type* is the type of the parameter.

image *function\_body* contains actual code for the function. Unlike a procedure, the body of a function must return a value of the type specified in the RETURN clause.

The following example creates a function named circle\_area(), which returns the area of a circle. The radius of the circle is passed as a parameter named p\_radius to circle\_area(); notice that circle\_area() returns a NUMBER:

CREATE FUNCTION circle\_area (

p\_radius IN NUMBER

) RETURN NUMBER AS

v\_pi NUMBER := 3.1415926;

v\_area NUMBER;

BEGIN

-- circle area is pi multiplied by the radius squared

v\_area := v\_pi \* POWER(p\_radius, 2);

RETURN v\_area;

END circle\_area;

/

The next example creates a function named average\_product\_price(), which returns the average price of products whose product\_type\_id equals the parameter value:

CREATE FUNCTION average\_product\_price (

p\_product\_type\_id IN INTEGER

) RETURN NUMBER AS

v\_average\_product\_price NUMBER;

BEGIN

SELECT AVG(price)

INTO v\_average\_product\_price

FROM products

WHERE product\_type\_id = p\_product\_type\_id;

RETURN v\_average\_product\_price;

END average\_product\_price;

/

**Calling a Function**

You call your own functions as you would call any of the built-in database functions; you saw how to call built-in functions in [Chapter 4](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch04.html#ch04). (Just to refresh your memory, you can call a function using a SELECT statement that uses the dual table in the FROM clause.) The following example calls circle\_area(), passing a radius of 2 meters to the function using positional notation:

**SELECT circle\_area(2)**

**FROM dual;**

CIRCLE\_AREA(2)

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

12.5663704

In Oracle Database 11*g*, you can also use named and mixed notation when calling functions. For example, the following query uses named notation when calling circle\_area():

**SELECT circle\_area(p\_radius => 4)**

**FROM dual;**

CIRCLE\_AREA(P\_RADIUS=>4)

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

50.2654816

The next example calls average\_product\_price(), passing the parameter value 1 to the function to get the average price of products whose product\_type\_id is 1:

**SELECT average\_product\_price(1)**

**FROM dual;**

AVERAGE\_PRODUCT\_PRICE(1)

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

29.965

**Getting Information on Functions**

You can get information on your functions from the user\_procedures view; this view was covered earlier in the section "Getting Information on Procedures." The following example retrieves the object\_name, aggregate, and parallel columns from user\_procedures for circle\_area() and average\_product\_price():

**SELECT object\_name, aggregate, parallel**

**FROM user\_procedures**

**WHERE object\_name IN ('CIRCLE\_AREA', 'AVERAGE\_PRODUCT\_PRICE');**

OBJECT\_NAME AGG PAR

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

AVERAGE\_PRODUCT\_PRICE NO NO

CIRCLE\_AREA NO NO

**Dropping a Function**

You drop a function using DROP FUNCTION. For example, the following statement drops circle\_area():

DROP FUNCTION circle\_area;

**PACKAGES**

In this section, you’ll learn how to group procedures and functions together into *packages*. Packages allow you to encapsulate related functionality into one self-contained unit. By modularizing your PL/SQL code through the use of packages, you build up your own libraries of code that other programmers can reuse. In fact, the Oracle database comes with a library of packages, which allow you to access external files, manage the database, generate HTML, and much more; to see all the packages, you should consult the *Oracle Database PL/SQL Packages and Types Reference* manual from Oracle Corporation.

Packages are typically made up of two components: a *specification* and a *body*. The package specification lists the available procedures, functions, types, and objects. You can make the items listed in the specification available to all database users, and I refer to these items as being *public* (although only users you have granted privileges to access your package can use it). The specification doesn’t contain the code that makes up the procedures and functions; the code is contained in the package body.

Any items in the body that are not listed in the specification are *private* to the package. Private items can be used only inside the package body. By using a combination of public and private items, you can build up a package whose complexity is hidden from the outside world. This is one of the primary goals of all programming: hide complexity from your users.

**Creating a Package Specification**

You create a package specification using the CREATE PACKAGE statement. The simplified syntax for the CREATE PACKAGE statement is as follows:

CREATE [OR REPLACE] PACKAGE *package\_name*

{IS | AS}

*package\_specification*

END *package\_name;*

where

image *package\_name* is the name of the package.

image *package\_specification* lists the public procedures, functions, types, and objects available to your package’s users.

The following example creates a specification for a package named product\_package:

CREATE PACKAGE product\_package AS

TYPE t\_ref\_cursor IS REF CURSOR;

FUNCTION get\_products\_ref\_cursor RETURN t\_ref\_cursor;

PROCEDURE update\_product\_price (

p\_product\_id IN products.product\_id%TYPE,

p\_factor IN NUMBER

);

END product\_package;

/

The t\_ref\_cursor type is a PL/SQL REF CURSOR type. A REF CURSOR is similar to a pointer in the C++ programming language, and it points to a cursor; as you saw earlier, a cursor allows you to read the rows returned by a query. The get\_products\_ref\_cursor() function returns a t\_ref\_cursor, and, as you’ll see in the next section, it points to a cursor that contains the rows retrieved from the products table.

The update\_product\_price() procedure multiplies the price of a product and commits the change.

**Creating a Package Body**

You create a package body using the CREATE PACKAGE BODY statement. The simplified syntax for the CREATE PACKAGE BODY statement is as follows:

CREATE [OR REPLACE] PACKAGE BODY *package\_name*

{IS | AS}

*package\_body*

END *package\_name;*

where

image *package\_name* is the name of the package, which must match the package name in the specification.

image *package\_body* contains the code for the procedures and functions.

The following example creates the package body for product\_package:

CREATE PACKAGE BODY product\_package AS

FUNCTION get\_products\_ref\_cursor

RETURN t\_ref\_cursor IS

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

BEGIN

-- get the REF CURSOR

OPEN v\_products\_ref\_cursor FOR

SELECT product\_id, name, price

FROM products;

-- return the REF CURSOR

RETURN v\_products\_ref\_cursor;

END get\_products\_ref\_cursor;

PROCEDURE update\_product\_price (

p\_product\_id IN products.product\_id%TYPE,

p\_factor IN NUMBER

) AS

v\_product\_count INTEGER;

BEGIN

-- count the number of products with the

-- supplied product\_id (will be 1 if the product exists)

SELECT COUNT(\*)

INTO v\_product\_count

FROM products

WHERE product\_id = p\_product\_id;

-- if the product exists (v\_product\_count = 1) then

-- update that product’s price

IF v\_product\_count = 1 THEN

UPDATE products

SET price = price \* p\_factor

WHERE product\_id = p\_product\_id;

COMMIT;

END IF;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

END update\_product\_price;

END product\_package;

/

The get\_products\_ref\_cursor() function opens the cursor and retrieves the product\_id, name, and price columns from the products table The reference to this cursor (the REF CURSOR) is stored in v\_products\_ref\_cursor and returned by the function.

The update\_product\_price() procedure multiplies the price of a product and commits the change. This procedure is identical to the one shown earlier in the section "Creating a Procedure," so I won’t go into the details on how it works again.

**CALLING FUNCTIONS AND PROCEDURES IN A PACKAGE**

When calling functions and procedures in a package, you must include the package name in the call. The following example calls product\_package.get\_products\_ref\_cursor(), which returns a reference to a cursor containing the product\_id, name and price for the products:

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

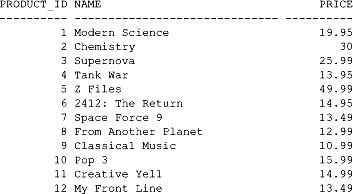
**FROM dual;**

GET\_PRODUCTS\_REF\_CUR

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

CURSOR STATEMENT : 1

CURSOR STATEMENT : 1



The next example calls product\_package.update\_product\_price() to multiply product #3’s price by 1.25:

**CALL product\_package.update\_product\_price(3, 1.25);**

The next query retrieves the details for product #3; notice that the price has increased:

**SELECT price**

**FROM products**

**WHERE product\_id = 3;**

PRICE

----------

32.49

**Getting Information on Functions and Procedures in a Package**

You can get information on your functions and procedures in a package from the user\_procedures view; this view was covered earlier in the section "Getting Information on Procedures." The following example retrieves the object\_name and procedure\_name columns from user\_procedures for product\_package:

**SELECT object\_name, procedure\_name**

**FROM user\_procedures**

**WHERE object\_name = 'PRODUCT\_PACKAGE';**

OBJECT\_NAME PROCEDURE\_NAME

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

PRODUCT\_PACKAGE GET\_PRODUCTS\_REF\_CURSOR

PRODUCT\_PACKAGE UPDATE\_PRODUCT\_PRICE

**Dropping a Package**

You drop a package using DROP PACKAGE. For example, the following statement drops product\_package:

DROP PACKAGE product\_package;

**TRIGGERS**

A *trigger* is a procedure that is run (or *fired*) automatically by the database when a specified DML statement (INSERT, UPDATE, or DELETE) is run against a certain database table. Triggers are useful for doing things like advanced auditing of changes made to column values in a table.

**When a Trigger Fires**

A trigger may fire before or after a DML statement runs. Also, because a DML statement can affect more than one row, the code for the trigger may be run once for every row affected (a *row-level trigger*), or just once for all the rows (a *statement-level trigger*). For example, if you create a row-level trigger that fires for an UPDATE on a table, and you run an UPDATE statement that modified ten rows of that table, then that trigger would run ten times. If, however, your trigger was a statement-level trigger, the trigger would fire once for the whole UPDATEstatement, regardless of the number of rows affected.

There is another difference between a row-level trigger and a statement-level trigger: A row-level trigger has access to the old and new column values when the trigger fires as a result of an UPDATE statement on that column. The firing of a row-level trigger may also be limited using a trigger *condition*; for example, you could set a condition that limits the trigger to fire only when a column value is less than a specified value.

**Set Up for the Example Trigger**

As mentioned, triggers are useful for doing advanced auditing of changes made to column values. In the next section, you’ll see a trigger that records when a product’s price is lowered by more than 25 percent; when this occurs, the trigger will add a row to the product\_price\_audit table. The product\_price\_audit table is created by the following statement in the store\_schema.sql script:

CREATE TABLE product\_price\_audit (

product\_id INTEGER

CONSTRAINT price\_audit\_fk\_products

REFERENCES products(product\_id),

old\_price NUMBER(5, 2),

new\_price NUMBER(5, 2)

);

As you can see, the product\_id column of the product\_price\_audit table is a foreign key to the product\_id column of the products table. The old\_price column will be used to store the old price of a product prior to the change, and the new\_price column will be used to store the new price after the change.

**Creating a Trigger**

You create a trigger using the CREATE TRIGGER statement. The simplified syntax for the CREATE TRIGGERstatement is as follows:

CREATE [OR REPLACE] TRIGGER *trigger\_name*

{BEFORE | AFTER | INSTEAD OF | FOR} *trigger\_event*

ON *table\_name*

[FOR EACH ROW]

[{FORWARD | REVERSE} CROSSEDITION]

[{FOLLOWS | PRECEDES} *schema.other\_trigger*}

[{ENABLE | DISABLE}]

[WHEN *trigger\_condition*]]

BEGIN

*trigger\_body*

END *trigger\_name;*

where

image OR REPLACE means the trigger is to replace an existing trigger, if present.

image *trigger\_name* is the name of the trigger.

image BEFORE means the trigger fires before the triggering event is performed. AFTER means the trigger fires after the triggering event is performed. INSTEAD OF means the trigger fires instead of performing the triggering event. FOR, which is new for Oracle Database 11*g*, allows you to create a compound trigger consisting of up to four sections in the trigger body.

image *trigger\_event* is the event that causes the trigger to fire.

image *table\_name* is the table that the trigger references.

image FOR EACH ROW means the trigger is a row-level trigger, that is, the code contained within *trigger\_body*is run for each row when the trigger fires. If you omit FOR EACH ROW, the trigger is a statement-level trigger, which means the code within *trigger\_body* is run once when the trigger fires.

image {FORWARD | REVERSE} CROSSEDITION is new for Oracle Database 11*g* and will typically be used by database administrators or application administrators. A FORWARD cross edition trigger is intended to fire when a DML statement makes a change in the database while an online application currently accessing the database *is being patched or upgraded* (FORWARD is the default); the code in the trigger body must be designed to handle the DML changes when the application patching or upgrade is complete. A REVERSE cross edition trigger is similar, except it is intended to fire and handle DML changes made *after the online application has been patched or upgraded*.

image {FOLLOWS | PRECEDES} *schema.other\_trigger* is new for Oracle Database 11*g* and specifies whether the firing of the trigger follows or precedes the firing of another trigger specified in *schema.other\_trigger*. You can create a series of triggers that fire in a specific order.

image {ENABLE | DISABLE} is new for Oracle Database 11*g* and indicates whether the trigger is initially enabled or disabled when it is created (the default is ENABLE). You enable a disabled trigger by using the ALTER TRIGGER *trigger\_name* ENABLE statement or by enabling all triggers for a table using ALTER TABLE*table\_name* ENABLE ALL TRIGGERS.

image *trigger\_condition* is a Boolean condition that limits when a trigger actually runs its code.

image *trigger\_body* contains the code for the trigger.

The example trigger you’ll see in this section fires before an update of the price column in the productstable; therefore, I’ll name the trigger before\_product\_price\_update. Also, because I want to use the price column values before and after an UPDATE statement modifies the price column’s value, I must use a row-level trigger. Finally, I want to audit a price change when the new price is lowered by more than 25 percent of the old price; therefore, I’ll need to specify a trigger condition to compare the new price with the old price. The following statement creates the before\_product\_price\_update trigger:

CREATE TRIGGER before\_product\_price\_update

BEFORE UPDATE OF price

ON products

FOR EACH ROW WHEN (new.price < old.price \* 0.75)

BEGIN

dbms\_output.put\_line('product\_id = ' || :old.product\_id);

dbms\_output.put\_line('Old price = ' || :old.price);

dbms\_output.put\_line('New price = ' || :new.price);

dbms\_output.put\_line('The price reduction is more than 25%');

-- insert row into the product\_price\_audit table

INSERT INTO product\_price\_audit (

product\_id, old\_price, new\_price

) VALUES (

:old.product\_id, :old.price, :new.price

);

END before\_product\_price\_update;

/

There are five things you should notice about this statement:

image BEFORE UPDATE OF price means the trigger fires before an update of the price column.

image FOR EACH ROW means this as a row-level trigger, that is, the trigger code contained within the BEGIN and END keywords runs once for each row modified by the update.

image The trigger condition is (new.price < old.price \* 0.75), which means the trigger fires only when the new price is less than 75 percent of the old price (that is, when the price is reduced by more than 25 percent).

image The new and old column values are accessed using the :old and :new aliases in the trigger.

image The trigger code displays the product\_id, the old and new prices, and a message stating that the price reduction is more than 25 percent. The code then adds a row to the product\_price\_audit table containing the product\_id and the old and new prices.

**Firing a Trigger**

To see the output from the trigger, you need to run the SET SERVEROUTPUT ON command:

**SET SERVEROUTPUT ON**

To fire the before\_product\_price\_update trigger, you must reduce a product’s price by more than 25 percent. Go ahead and perform the following UPDATE statement to reduce the price of products #5 and #10 by 30 percent (this is achieved by multiplying the price column by .7). The following UPDATE statement causes the before\_product\_price\_update trigger to fire:

**UPDATE products**

**SET price = price \* .7**

**WHERE product\_id IN (5, 10);**

product\_id = 10

Old price = 15.99

New price = 11.19

The price reduction is more than 25%

product\_id = 5

Old price = 49.99

New price = 34.99

The price reduction is more than 25%

2 rows updated.

As you can see, the trigger fired for products #10 and #5. You can see that the trigger did indeed add the two required rows containing the product\_ids, along with the old and new prices, to the product\_price\_audit table using the following query:

**SELECT \***

**FROM product\_price\_audit**

**ORDER BY product\_id;**

PRODUCT\_ID OLD\_PRICE NEW\_PRICE

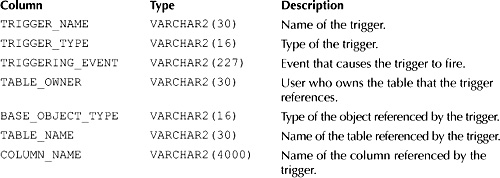
---------- ---------- ----------

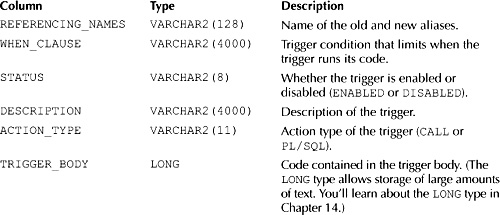
5 49.99 34.99

10 15.99 11.19

**Getting Information on Triggers**

You can get information on your triggers from the user\_triggers view. [Table 11-3](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch11.html#table_11-3) describes some of the columns in user\_triggers.





**TABLE 11-3** *Some Columns in the user\_triggers View*

image

**NOTE**  
*You can get information on all the triggers you have access to using* all\_triggers.

The following example retrieves the details of the before\_product\_price\_update trigger from user\_triggers (the output is printed pretty for clarity):

**SELECT trigger\_name, trigger\_type, triggering\_event, table\_owner**

**base\_object\_type, table\_name, referencing\_names, when\_clause, status,**

**description, action\_type, trigger\_body**

**FROM user\_triggers**

**WHERE trigger\_name = 'BEFORE\_PRODUCT\_PRICE\_UPDATE';**

TRIGGER\_NAME TRIGGER\_TYPE

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

BEFORE\_PRODUCT\_PRICE\_UPDATE BEFORE EACH ROW

TRIGGERING\_EVENT

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

UPDATE

TABLE\_OWNER BASE\_OBJECT\_TYPE TABLE\_NAME

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

STORE TABLE PRODUCTS

REFERENCING\_NAMES

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

REFERENCING NEW AS NEW OLD AS OLD

WHEN\_CLAUSE

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

new.price < old.price \* 0.75

STATUS

--------

ENABLED

DESCRIPTION

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

before\_product\_price\_update

BEFORE UPDATE OF

price

ON

products

FOR EACH ROW

ACTION\_TYPE

-----------

PL/SQL

TRIGGER\_BODY

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

BEGIN

dbms\_output.put\_line('product\_id = ' || :old.product\_id);

dbms\_output…

image

**NOTE**  
*You can see all the code for the trigger using the SQL\*Plus* SET LONG *command, for example*, SET LONG 1000.

**Disabling and Enabling a Trigger**

You can stop a trigger from firing by disabling it by using the ALTER TRIGGER statement. The following example disables the before\_product\_price\_update trigger:

ALTER TRIGGER before\_product\_price\_update DISABLE;

The next example enables the before\_product\_price\_update trigger:

ALTER TRIGGER before\_product\_price\_update ENABLE;

**Dropping a Trigger**

You drop a trigger using DROP TRIGGER. The following example drops the before\_product\_price\_update trigger:

DROP TRIGGER before\_product\_price\_update;

**NEW ORACLE DATABASE 11*G* PL/SQL FEATURES**

In this section, you’ll see some of the new PL/SQL features introduced in Oracle Database 11*g*. Specifically, the following will be discussed:

image The SIMPLE\_INTEGER type

image Support for sequences in PL/SQL

image PL/SQL native machine code generation

**SIMPLE\_INTEGER Type**

The SIMPLE\_INTEGER type is a subtype of BINARY\_INTEGER; the SIMPLE\_INTEGER can store the same range as BINARY\_INTEGER, except SIMPLE\_INTEGER cannot store a NULL value. The range of values SIMPLE\_INTEGER can store is −231 (−2,147,483,648) to 231 (2,147,483,648).

Arithmetic overflow is truncated when using SIMPLE\_INTEGER values; therefore, calculations don’t raise an error when overflow occurs. Because overflow errors are ignored, the values stored in a SIMPLE\_INTEGER can wrap from positive to negative and from negative to positive, as, for example:

230 + 230 = 0x40000000 + 0x40000000 = 0x80000000 = – 231

−231 + − 231 = 0x80000000 + 0x80000000 = 0x00000000 = 0

In the first example, two positive values are added, and a negative total is produced. In the second example, two negative values are added, and zero is produced.

Because overflow is ignored and truncated when using SIMPLE\_INTEGER values in calculations, SIMPLE\_INTEGER offers much better performance than BINARY\_INTEGER when the DBA configures the database to compile PL/SQL to native machine code. Because of this benefit, you should use SIMPLE\_INTEGER in your PL/SQL code when you don’t need to store a NULL and you don’t care about overflow truncation occurring in your calculations; otherwise, you should use BINARY\_INTEGER.

The following get\_area() procedure shows the use of the SIMPLE\_INTEGER type; get\_area()calculates and displays the area of a rectangle:

CREATE PROCEDURE get\_area

AS

v\_width SIMPLE\_INTEGER := 10;

v\_height SIMPLE\_INTEGER := 2;

v\_area SIMPLE\_INTEGER := v\_width \* v\_height;

BEGIN

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

END get\_area;

/

image

**NOTE**  
*You’ll find this example, and the other examples in this section, in a script named*plsql\_11g\_examples.sql *in the* SQL *directory. You may run this script if you are using Oracle Database* 11*g*.

The following example shows the execution of get\_area():

**SET SERVEROUTPUT ON**

**CALL get\_area();**

v\_area = 20

As expected, the calculated area is 20.

**SEQUENCES IN PL/SQL**

In the previous chapter you saw how to create and use sequences of numbers in SQL. In Oracle Database 11*g*, you can also use sequences in PL/SQL code.

As a reminder, a sequence generates a series of numbers. When you create a sequence in SQL, you can specify its initial value and an increment for the series of subsequent numbers.

You use the currval pseudo column to get the current value in the sequence and nextval to generate the next number. Before you access currval, you must first use nextval to generate an initial number.

The following statement creates a table named new\_products; this table will be used shortly:

CREATE TABLE new\_products (

product\_id INTEGER CONSTRAINT new\_products\_pk PRIMARY KEY,

name VARCHAR2(30) NOT NULL,

price NUMBER(5, 2)

);

The next statement creates a sequence named s\_product\_id:

CREATE SEQUENCE s\_product\_id;

The following statement creates a procedure named add\_new\_products, which uses s\_product\_id to set the product\_id column in a row added to the new\_products table; notice the use of the nextval and currval pseudo columns in the PL/SQL code (this is new for Oracle Database 11*g*):

CREATE PROCEDURE add\_new\_products

AS

v\_product\_id BINARY\_INTEGER;

BEGIN

-- use nextval to generate the initial sequence number

v\_product\_id := s\_product\_id.nextval;

DBMS\_OUTPUT.PUT\_LINE('v\_product\_id = ' || v\_product\_id);

-- add a row to new\_products

INSERT INTO new\_products

VALUES (v\_product\_id, 'Plasma Physics book', 49.95);

DBMS\_OUTPUT.PUT\_LINE(’s\_product\_id.currval = ' || s\_product\_id.currval);

-- use nextval to generate the next sequence number

v\_product\_id := s\_product\_id.nextval;

DBMS\_OUTPUT.PUT\_LINE('v\_product\_id = ' || v\_product\_id);

-- add another row to new\_products

INSERT INTO new\_products

VALUES (v\_product\_id, 'Quantum Physics book', 69.95);

DBMS\_OUTPUT.PUT\_LINE(’s\_product\_id.currval = ' || s\_product\_id.currval);

END add\_new\_products;

/

The following example runs add\_new\_products() and shows the contents of the new\_products table:

**SET SERVEROUTPUT ON**

**CALL add\_new\_products();**

v\_product\_id = 1

s\_product\_id.currval = 1

v\_product\_id = 2

s\_product\_id.currval = 2

**SELECT \* FROM new\_products;**

PRODUCT\_ID NAME PRICE

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

1 Plasma Physics book 49.95

2 Quantum Physics book 69.95

As expected, two rows were added to the table.

**PL/SQL Native Machine Code Generation**

By default, each PL/SQL program unit is compiled into intermediate form, machine-readable code. This machine-readable code is stored in the database and interpreted every time the code is run. With PL/SQL native compilation, the PL/SQL is turned into native code and stored in shared libraries. Native code runs much faster than intermediate code because native code doesn’t have to be interpreted before it runs.

In certain versions of the database prior to Oracle Database 11*g*, you can compile PL/SQL code to C code, and then compile the C code into machine code; this is a very laborious and problematic process. In Oracle Database 11*g*, the PL/SQL complier can generate native machine code directly. Setting up the database to generate native machine code should be done only by an experienced DBA (as such, its coverage is beyond the scope of this book). You can read all about PL/SQL native machine code generation in the *PL/SQL User’s Guide and Reference* manual from Oracle Corporation.

**SUMMARY**

In this chapter, you learned the following:

image PL/SQL programs are divided up into blocks containing PL/SQL and SQL statements.

image A loop, such as a WHILE or FOR loop, runs statements multiple times.

image A cursor allows PL/SQL to read the rows returned by a query.

image Exceptions are used to handle run-time errors that occur in your PL/SQL code.

image A procedure contains a group of statements. Procedures allow you to centralize your business logic in the database and may be run by any program that accesses the database.

image A function is similar to a procedure except that a function must return a value.

image You can group procedures and functions together into packages, which encapsulate related functionality into one self-contained unit.

image A trigger is a procedure that is run automatically by the database when a specific INSERT, UPDATE, or DELETE statement is run. Triggers are useful for doing things like advanced auditing of changes made to column values in a table.