PART III

Create Programs Using PL/SQL

CHAPTER 8

Introduction to PL/SQL

Storing and retrieving information is just one part of any real-life application. Even the simplest applications need to do some processing that is difficult or impossible using SQL alone. Just think of how complex the computations are when the government's share of your earnings needs to be computed every year! OK, maybe you want to think of another example instead. In any case, SQL alone isn't up to the task.

What Is PL/SQL?

You may ask why SQL doesn't have features that allow you to do more sophisticated computations on data. The reason is partly historical: SQL came into existence as a database query language (Structured Query Language) and has evolved and been optimized for doing exactly that: querying databases. Different providers of database software have agreed to certain SQL standards, but they did not agree on how to give users more sophisticated SQL-oriented programming capabilities. Thus, each database software provider has come up with proprietary or semi-proprietary products. Oracle calls its solution *PL/SQL*. You can think of this as standing for "Programming Language for SQL."

In this chapter you will be introduced to the basics of PL/SQL. You will learn the difference between SQL, SQL*Plus®, and PL/SQL. You will also start writing simple PL/SQL procedures, as well as functions using basic PL/SQL constructs like variables, loops, and cursors. Then you will learn about the important art of handling errors in a way the user can easily understand.

If you just started reading in this chapter and have not done any of the exercises in the preceding chapters, you will need to create the sample tables built in prior chapters before you can do the exercises in this chapter. To accomplish this, enter the following SQL commands:

L 8-1

```
DROP TABLE plsql101_purchase;
DROP TABLE plsql101_product;
DROP TABLE plsql101_person;
DROP TABLE plsql101_old_item;
DROP TABLE plsql101_purchase_archive;

CREATE TABLE plsql101_person (
     person_code VARCHAR2(3) PRIMARY KEY,
     first_name  VARCHAR2(15),
     last_name   VARCHAR2(20),
     hire_date   DATE
     )
;
```

```
CREATE INDEX plsql101_person_name_index
ON plsql101_person(last_name, first_name);

ALTER TABLE plsql101_person
ADD CONSTRAINT plsql101_person_unique UNIQUE (
     first_name,
     last_name,
     hire_date
     )
;

INSERT INTO plsql101_person VALUES
     ('CA', 'Charlene', 'Atlas', '01-FEB-05');
INSERT INTO plsql101_person VALUES
     ('GA', 'Gary', 'Anderson', '15-FEB-05');
INSERT INTO plsql101_person VALUES
     ('BB', 'Bobby', 'Barkenhagen', '28-FEB-05');
INSERT INTO plsql101_person VALUES
     ('LB', 'Laren', 'Baxter', '01-MAR-05');
INSERT INTO plsql101_person VALUES
     ('LN', 'Linda', 'Norton', '01-JUN-06');

CREATE TABLE plsql101_product (
     product_name     VARCHAR2(25) PRIMARY KEY,
     product_price    NUMBER(4,2),
     quantity_on_hand NUMBER(5,0),
     last_stock_date  DATE
     )
;

ALTER TABLE plsql101_product ADD CONSTRAINT positive_quantity CHECK(
     quantity_on_hand IS NOT NULL
     AND
     quantity_on_hand >=0
     )
;

INSERT INTO plsql101_product VALUES
     ('Small Widget', 99, 1, '15-JAN-06');
INSERT INTO plsql101_product VALUES
     ('Medium Wodget', 75, 1000, '15-JAN-05');
INSERT INTO plsql101_product VALUES
     ('Chrome Phoobar', 50, 100, '15-JAN-06');
INSERT INTO plsql101_product VALUES
     ('Round Chrome Snaphoo', 25, 10000, null);
```

```
INSERT INTO plsql101_product VALUES
     ('Extra Huge Mega Phoobar +',9.95,1234,'15-JAN-07');
INSERT INTO plsql101_product VALUES ('Square Zinculator',
     45, 1, TO_DATE('December 31, 2005, 11:30 P.M.',
                    'Month dd, YYYY, HH:MI P.M.')
     )
;
INSERT INTO plsql101_product VALUES (
     'Anodized Framifier', 49, 5, NULL);
INSERT INTO plsql101_product VALUES (
     'Red Snaphoo', 1.95, 10, '31-DEC-04');
INSERT INTO plsql101_product VALUES (
     'Blue Snaphoo', 1.95, 10, '30-DEC-04')
;

CREATE TABLE plsql101_purchase (
     product_name  VARCHAR2(25),
     salesperson   VARCHAR2(3),
     purchase_date DATE,
     quantity      NUMBER(4,2)
     )
;

ALTER TABLE plsql101_purchase
ADD PRIMARY KEY (product_name,
                 salesperson,
                 purchase_date
                 )
;

ALTER TABLE plsql101_purchase
ADD CONSTRAINT reasonable_date CHECK(
     purchase_date IS NOT NULL
     AND
     TO_CHAR(purchase_date, 'YYYY-MM-DD') >= '2003-06-30'
     )
;

ALTER TABLE plsql101_purchase
ADD CONSTRAINT plsql101_purchase_fk_product FOREIGN KEY
     (product_name) REFERENCES plsql101_product;

ALTER TABLE plsql101_purchase
ADD CONSTRAINT plsql101_purchase_fk_person FOREIGN KEY
     (salesperson) REFERENCES plsql101_person;

CREATE INDEX plsql101_purchase_product
ON plsql101_purchase(product_name);
```

```
CREATE INDEX plsql101_purchase_salesperson
ON plsql101_purchase(salesperson);

INSERT INTO plsql101_purchase VALUES
     ('Small Widget', 'CA', '14-JUL-06', 1);
INSERT INTO plsql101_purchase VALUES
     ('Medium Wodget', 'BB', '14-JUL-06', 75);
INSERT INTO plsql101_purchase VALUES
     ('Chrome Phoobar', 'GA', '14-JUL-06', 2);
INSERT INTO plsql101_purchase VALUES
     ('Small Widget', 'GA', '15-JUL-06', 8);
INSERT INTO plsql101_purchase VALUES
     ('Medium Wodget', 'LB', '15-JUL-06', 20);
INSERT INTO plsql101_purchase VALUES
     ('Round Chrome Snaphoo', 'CA', '16-JUL-06', 5);
INSERT INTO plsql101_purchase VALUES (
     'Small Widget', 'CA', '17-JUL-06', 1)
;

UPDATE plsql101_product
SET    product_price = product_price * .9
WHERE  product_name NOT IN (
       SELECT DISTINCT product_name
       FROM   plsql101_purchase
       )
;

CREATE TABLE plsql101_purchase_archive (
     product_name  VARCHAR2(25),
     salesperson   VARCHAR2(3),
     purchase_date DATE,
     quantity      NUMBER(4,2)
     )
;

INSERT INTO plsql101_purchase_archive VALUES
     ('Round Snaphoo', 'BB', '21-JUN-04', 10);
INSERT INTO plsql101_purchase_archive VALUES
     ('Large Harflinger', 'GA', '22-JUN-04', 50);
INSERT INTO plsql101_purchase_archive VALUES
     ('Medium Wodget', 'LB', '23-JUN-04', 20);
INSERT INTO plsql101_purchase_archive VALUES
     ('Small Widget', 'ZZ', '24-JUN-05', 80);
INSERT INTO plsql101_purchase_archive VALUES
     ('Chrome Phoobar', 'CA', '25-JUN-05', 2);
INSERT INTO plsql101_purchase_archive VALUES
     ('Small Widget', 'JT', '26-JUN-05', 50);
```

Describe PL/SQL

PL/SQL provides the features that allow you to do sophisticated information processing. Let's say that every night you want to transfer the day's business summary into a day's summary table—*PL/SQL packages* can help you do this. Or, you want to know whether you need to arrange for extra supplies for purchase orders that are really large—PL/SQL provides *triggers* that will notify you as soon as any order placed is found to be larger than certain limits decided by you. In addition, you can use *PL/SQL stored procedures* to compute your employees' performance to help you decide about bonuses. Plus, a nice *PL/SQL function* can calculate the tax withholdings for an employee.

PL/SQL lets you use all the SQL data manipulation, cursor control, and transaction control commands, as well as all the SQL functions and operators. So, you can manipulate Oracle data flexibly and safely. Also, PL/SQL fully supports SQL datatypes. This reduces the need to convert data passed between your applications and the database. PL/SQL also supports dynamic SQL, an advanced programming technique that makes your applications more flexible and versatile. Your programs can build and process SQL data definition, data control, and session control statements "on the fly" at run time.

Before we proceed to learn more about some of these power tools, I will give you some idea about how SQL, SQL*Plus®, and PL/SQL relate to each other.

Who's Who in SQL, SQL*Plus®, and PL/SQL

Think of a restaurant. You go in and hopefully a well-trained waiter or waitress waits on you. You look through the menu and place an order. The waiter writes down your order and takes it into the kitchen. The kitchen is huge—there are many chefs and assistants. You can see a lot of food—cooked, partially cooked, and uncooked—stored in the kitchen. You can also see people with various jobs: they take the food in and out of storage, prepare a particular type of food (just soups or just salads, for instance), and so forth. Depending on what menu items you ordered, the waiter takes the order to different chefs. Some simple orders are completed by one chef, while more complex orders may require help from assistants, or even multiple chefs. In addition, some orders are standard items—a waiter can just tell a chef "mushroom pizza"—while other orders are custom creations requiring a detailed list of exactly what ingredients you want.

Now alter this scenario a little. Think of an Oracle database as the restaurant's kitchen, with SQL*Plus® serving as the waiter taking our orders—scripts, commands, or programs—to the kitchen, or database. Inside the kitchen are two main chefs: SQL and PL/SQL. Like a waiter, SQL*Plus® knows what orders it can process on its own, as well as what orders to take to specific chefs. In the same way that a waiter can bring you a glass of water without having to get it from a chef, SQL*Plus® can adjust the width of the lines shown on its screen without needing to go to the database.

The commands or programs you enter and execute at the SQL*Plus® prompt are somewhat like your special-order pizza. For custom orders the chefs have to do some thinking each time. Just like the chef has the recipe for cheese pizza stored in his or her brain, you can have PL/SQL store "recipes" for your favorite orders. These stored PL/SQL elements are called triggers, stored functions, stored procedures, and packages. You will learn more about them soon.

As I mentioned earlier, some orders require more than one chef to prepare them. Most of the interesting and useful database applications you create will have SQL and PL/SQL working together, passing information back and forth between them to process a script or program. In a restaurant, after an order is prepared it goes to a waiter to be taken to your table. Similarly, when SQL and PL/SQL process commands, the results go to SQL*Plus® (or a custom front-end form) to be displayed to the user.

Stored Procedures, Stored Functions, and Triggers

PL/SQL procedures, functions, and triggers all help you build complex business logic easily and in a *modular* fashion (meaning piece by piece, with the pieces being reusable by other pieces). Storing these in the Oracle server provides two immediate benefits: they can be used over and over with predictable results, and they execute very rapidly because server operations involve little or no network traffic.

Stored Procedures

A *stored procedure* is a defined set of actions written using the PL/SQL language. When a procedure is called, it performs the actions it contains. The procedure is stored in the database, which is the reason it is called a stored procedure.

A stored procedure can execute SQL statements and manipulate data in tables. It can be called to do its job from within another PL/SQL stored procedure, stored function, or trigger. A stored procedure can also be called directly from an SQL*Plus® prompt. As you read through the pages that follow, you will learn how to employ each of these methods for calling a stored procedure.

A procedure consists of two main parts: the specification and the body. The *procedure specification* contains the procedure's name and a description of its inputs and outputs. The inputs and outputs we are talking about are called the procedure's *formal parameters* or *formal arguments*. If a call to a procedure includes command-line parameters or other inputs, those values are called *actual parameters* or *actual arguments*.

Now let's take a look at some samples of procedure specifications. (Remember, the specification doesn't contain any code; it just names the procedure and defines any inputs and outputs the procedure can use.)

L 8-2
```
run_ytd_reports
```

This simple specification contains only the procedure's name. It has no parameters.

L 8-3
```
increase_prices (percent_increase NUMBER)
```

A value can be passed to this procedure when it is called. Within the procedure, the value will be addressed as PERCENT_INCREASE. Note that the value's datatype has been specified: NUMBER.

L 8-4
```
increase_salary_find_tax (increase_percent IN     NUMBER := 7,
                          sal              IN OUT NUMBER,
                          tax                 OUT NUMBER
                          )
```

Here we have a procedure with three formal parameters. The word IN after a parameter's name indicates that the procedure can read an incoming value from that parameter when the procedure is called. The word OUT after a parameter's name indicates that the procedure can use that parameter to send a value back to whatever called it. Having IN OUT after a parameter's name says that the parameter can bring a value into the procedure and also be used to send a value back out.

The INCREASE_PERCENT parameter in this example gets assigned a *default value* of 7 by including **:= 7** after the datatype. Because of this, if the procedure is called without specifying any increase percentage, it will increase the salary given by 7 percent and calculate the tax based on the new salary.

NOTE
Datatypes in a procedure cannot have size specifications. For instance, you can specify that a parameter is a NUMBER datatype, but not a NUMBER(10,2) datatype.

The *procedure body* is a block of PL/SQL code, which you will learn about in the section entitled Structure of a PL/SQL Block.

Stored Functions

A PL/SQL function is similar to a PL/SQL procedure: It has function specification and a function body. The main difference between a procedure and a function is that a function is designed to return a value that can be used within a larger SQL statement.

For instance, think for a moment about a function designed to calculate the percentage difference between two numbers. Ignoring the code that would perform this calculation, the function specification would look like this:

L 8-5
```
calc_percent(value_1 NUMBER,
             value_2 NUMBER) return NUMBER
```

This function accepts two numbers as input, referring to them internally as VALUE_1 and VALUE_2. Once the body of this function was written, it could be referred to in an SQL statement in the following way:

L 8-6
```
INSERT INTO employee VALUES (3000, CALC_PERCENT(300, 3000));
```

Triggers

A *trigger* is a PL/SQL procedure that gets executed automatically whenever some event defined by the trigger—the *triggering event*—happens. You can write triggers that fire when an INSERT, UPDATE, or DELETE statement is performed on a table; when DDL statements are issued; when a user logs on or off; or when the database starts, encounters an error, or shuts down.

Triggers differ from PL/SQL procedures in three ways:

- You cannot call a trigger from within your code. Triggers are called automatically by Oracle in response to a predefined event.
- Triggers do not have a parameter list.
- The specification for a trigger contains different information than a specification for a procedure.

You will learn more about triggers and their uses in Chapter 9.

Stored Procedures and SQL Scripts

While SQL scripts reside on your computer's hard disk, stored procedures reside within your Oracle database. An SQL script contains a series of SQL commands that are executed, one by one, when you invoke the script. In contrast, a stored procedure can contain flow-control commands allowing it to iterate through a particular section of code over and over; branch to another code section when particular situations occur; and respond to error conditions in a way you specify.

Structure of a PL/SQL Block

In this section you will learn about the PL/SQL *basic block*. Everything in PL/SQL that actually does work is made up of basic blocks. After learning about the basic blocks, you will see examples of complete procedures, functions, and triggers.

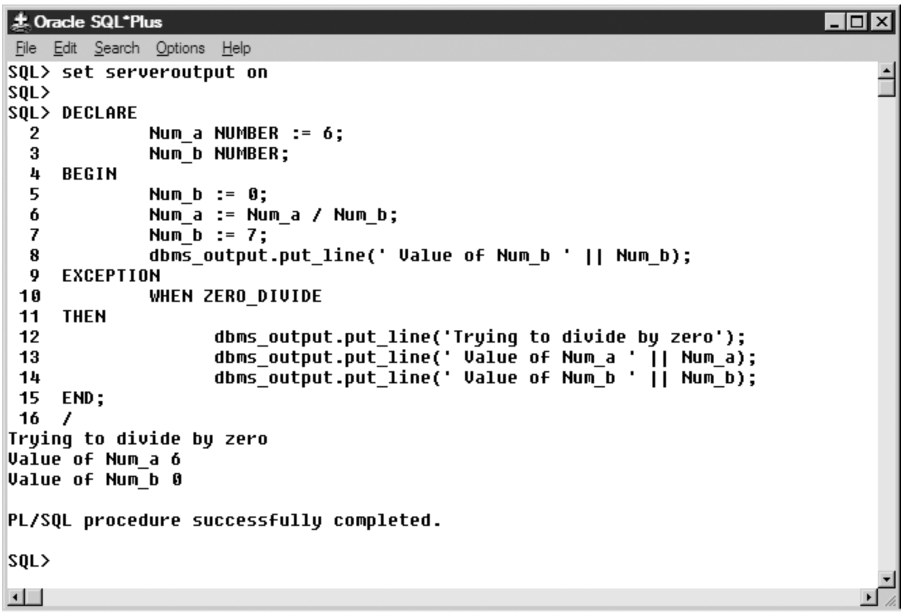
A PL/SQL basic block is made up of four sections: the *header section*, an optional *declaration section*, the *execution section*, and the optional *exception section*.

An *anonymous block* is a PL/SQL block with no header or name section, hence the term anonymous block. Anonymous blocks can be run from SQL*Plus® and they can be used within PL/SQL functions, procedures, and triggers. Recall that PL/SQL procedures, functions, and triggers are all made up of basic blocks themselves. What this means is that you can have a basic block within a basic block. Perhaps the best way to begin to understand a basic block is to examine a sample. First, type the following command so that information printed by programs can be made visible in SQL*Plus®:

L 8-7
```
set serveroutput on
```

Now try the following sample code to create an anonymous block. Compare your results with Figure 8-1.

L 8-8
```
DECLARE
        Num_a NUMBER := 6;
        Num_b NUMBER;
BEGIN
        Num_b := 0;
        Num_a := Num_a / Num_b;
        Num_b := 7;
        dbms_output.put_line(' Value of Num_b ' || Num_b);
EXCEPTION
        WHEN ZERO_DIVIDE
THEN
              dbms_output.put_line('Trying to divide by zero');
              dbms_output.put_line(' Value of Num_a ' || Num_a);
              dbms_output.put_line(' Value of Num_b ' || Num_b);
END;
/
```



```
SQL> set serveroutput on
SQL>
SQL> DECLARE
  2          Num_a NUMBER := 6;
  3          Num_b NUMBER;
  4  BEGIN
  5          Num_b := 0;
  6          Num_a := Num_a / Num_b;
  7          Num_b := 7;
  8          dbms_output.put_line(' Value of Num_b ' || Num_b);
  9  EXCEPTION
 10          WHEN ZERO_DIVIDE
 11  THEN
 12                dbms_output.put_line('Trying to divide by zero');
 13                dbms_output.put_line(' Value of Num_a ' || Num_a);
 14                dbms_output.put_line(' Value of Num_b ' || Num_b);
 15  END;
 16  /
Trying to divide by zero
Value of Num_a 6
Value of Num_b 0

PL/SQL procedure successfully completed.

SQL>
```

FIGURE 8-1. *Example of an anonymous PL/SQL block*

Header Section

The header section for a block varies based on what the block is part of. Recall that procedures, functions, triggers, and anonymous blocks are made up of basic blocks. In fact, each has one basic block that makes up its body. This body block may contain more basic blocks inside it. The header for this top-level basic block of a function, procedure, or trigger is the specification for that function, procedure, or trigger. For anonymous blocks, the header contains only the keyword DECLARE. For labeled blocks, the header contains the name of the label enclosed between << and >>, followed by the keyword DECLARE, as shown in the following code:

L 8-9

```
<<just_a_label>>
DECLARE
```

Block labels help make it easier to read code. In a procedure using nested blocks (blocks inside other blocks), you can refer to an item in a specific block by preceding the item's name with the name of the block (for example, *block_label.item_label*).

Declaration Section

The declaration section is optional. When used, it begins after the header section and ends at the keyword BEGIN. The declaration section contains the declarations for PL/SQL variables, constants, cursors, exceptions, functions, and procedures that will be used by the execution and exception sections of the block. All variable and constant declarations must come before any function or procedure declarations within the declaration section. You will learn more about PL/SQL variables and constants in the following section of the same name ("PL/SQL Variables and Constants"). A declaration tells PL/SQL to create a variable, constant, cursor, function, or procedure as specified in the declaration.

The declaration section (beginning with the word DECLARE) in the example shown in Figure 8-1 tells PL/SQL to create two number type variables called Num_a and Num_b. It also assigns a value of 6 by default to Num_a.

When a basic block has finished its run, everything declared within the declaration section stops existing. Things declared within the declaration section of a basic block can be used only within the same block. Thus, after running the example block in SQL*Plus® there is no way to pass Num_a to another PL/SQL procedure. Num_a and Num_b just go out of existence as soon as the block finishes its run. However, if you call a PL/SQL function or procedure within the execution or exception section of the block, you can pass Num_a and Num_b to them as actual parameters.

The long and short of the story is, whatever is in the declaration section is the private property of the block—to be used by and visible only to itself. Thus, what is in the declaration section of the block only lives as long as the block. In technical terms, Num_a and Num_b are said to have the *scope* of the block in which they are declared. The scope of the block starts at the beginning of the block's declaration section and ends at the end of its exception section.

Execution Section

The execution section starts with the keyword BEGIN and ends in one of two ways. If there is an exception section, the execution section ends with the keyword EXCEPTION. If no exception section is present, the execution section ends with the keyword END, followed optionally by the name of the function or procedure, and a semicolon. The execution section contains one or more PL/SQL statements that are executed when the block is run. The structure for the executable section is shown below:

```
BEGIN
    one or more PL/SQL statements
[exception section]
END [name of function or procedure];
```

The executable section, in the example block of Figure 8-1, contains three PL/SQL assignment statements. The assignment statement is the most commonly seen statement in PL/SQL code. The first statement assigns the value of zero to Num_b. The colon followed by an equal sign (:=) is the assignment operator. The assignment operator tells PL/SQL to compute whatever is on its right-hand side and place the result in whatever is on its left-hand side.

The second statement assigns Num_a the value of Num_a divided by Num_b. Note that after this statement is executed successfully the value of Num_a will be changed.

The third statement assigns the value of 7 to Num_b.

Exception Section

It is possible that during the execution of PL/SQL statements in the execution section, an error will be encountered that makes it impossible to proceed with the execution. These error conditions are called *exceptions*. The procedure's user should be informed when an exception occurs and told why it has occurred. You may want to issue a useful error to the user or you may want to take some corrective action and retry whatever the procedure was attempting before the error happened. You may want to roll back changes done to the database before the error occurred.

For all these situations PL/SQL helps you by providing *exception handling* capabilities. Exceptions are so important for good applications that I have a special section at the end of this chapter where you will learn more about them (see "Exceptions" within the "Error Handling" section). As an introduction, the following is the structure for the exception section:

```
EXCEPTION
  WHEN exception_name
  THEN
    actions to take when this exception occurs
  WHEN exception_name
  THEN
    actions to take when this exception occurs
```

The exception section begins at the keyword EXCEPTION and ends at the end of the block. For each exception there is a WHEN *exception_name* statement that specifies what should be done when a specific exception occurs. Our example has three funny-looking statements that have the effect of making text display on your SQL*Plus® screen. A little more information is needed to understand what they are doing, but we will leave the details for Chapter 9. The DBMS_OUTPUT package and PUT_LINE procedure are part of the Oracle database; together they cause text to display on your SQL*Plus® screen one line at a time.

All the statements between the statement that causes the exception and the exception section will be ignored. So, in the case of the example block in Figure 8-1, the assigning of 7 to Num_b is not executed. You can verify this by looking at the value for Num_b that the example code prints out.

When a statement in the exception section deals with an exception, we refer to that action as *exception handling*.

Detecting that the error occurred and which exception best describes it and then taking appropriate steps to inform PL/SQL about it so as to make it possible for PL/SQL to find the exception section for that exception is called *raising exception*. In the example code in Figure 8-1, the exception is raised by PL/SQL on its own by detecting that there is an attempt at division by zero. PL/SQL has a predefined name for this exception—ZERO_DIVIDE. In many situations the error must be detected by your code, not by PL/SQL.

Create a Simple PL/SQL Procedure

We have all the ingredients to try writing a complete PL/SQL procedure. You know about the basic block and you have learned about procedure specifications. Now try the following code:

L 8-10
```
CREATE PROCEDURE my_first_proc IS
        greetings VARCHAR2(20);
BEGIN
        greetings := 'Hello World';
        dbms_output.put_line(greetings);
END my_first_proc;
/
```

The syntax for creating a stored procedure is the following:

CREATE PROCEDURE procedure_specification IS procedure_body

In our sample, the procedure specification is just the name of the procedure, and the body is everything after it up to the last semicolon. For functions, you will use the keyword FUNCTION instead of PROCEDURE:

CREATE FUNCTION function_specification IS function_body

The forward slash (/) tells SQL*Plus® to go ahead and process the commands in the program. You can re-create the same procedure or function by changing the CREATE command to CREATE OR REPLACE. This will destroy the old definition of the procedure or function and replace it with the new one. If there is no old definition it will simply create a new one.

CREATE OR REPLACE PROCEDURE procedure_specification
IS procedure_body

Now let us see how this procedure can be called from SQL*Plus®:

L 8-11
```
set serveroutput on
EXECUTE my_first_proc;
```

SERVEROUTPUT ON allows you to see the printed output. The EXECUTE command actually executes the procedure. You can call the procedure from within an anonymous block as follows. Compare your results with those shown in Figure 8-2.

L 8-12
```
BEGIN
        my_first_proc;
END;
/
```

Call Procedures and Functions

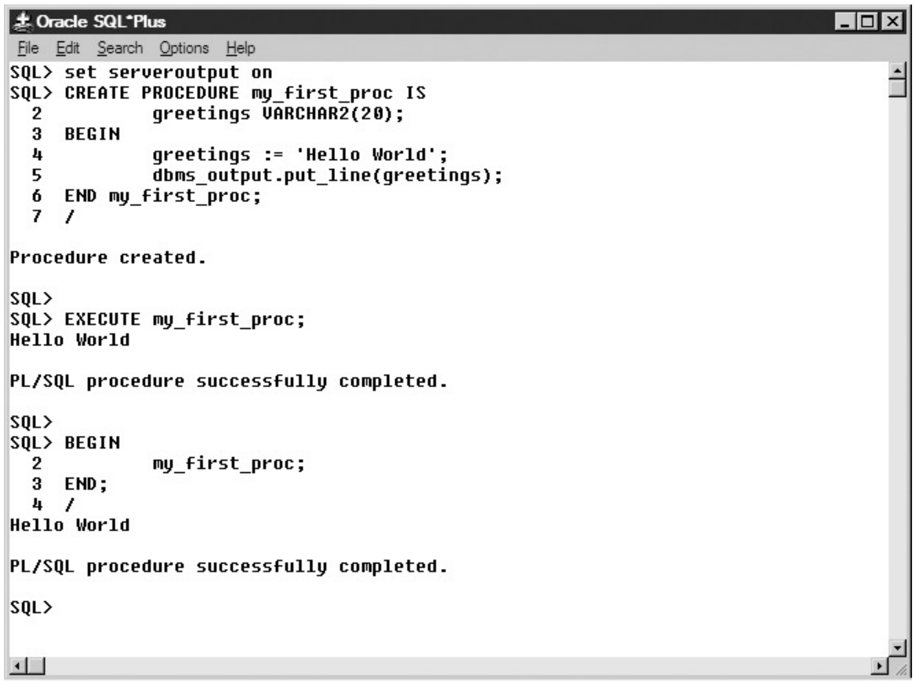
A procedure or function may or may not have formal parameters with default values. In fact, it may not have any formal parameters at all. For each case, the way the procedure or function is called is different. However, the following applies regardless of the parameters:

- The datatypes for the actual parameters must match or should be convertible by PL/SQL to the datatypes of corresponding formal parameters.
- Actual parameters must be provided for all formal parameters that do not have default values.

When calling a function without any parameters, you can just use the name with or without parentheses, like the following:

procedure_name();
or procedure_name;

The same syntax is used when dealing with a function, except a semicolon will not be used when the function is called as part of an expression.



```
SQL> set serveroutput on
SQL> CREATE PROCEDURE my_first_proc IS
  2          greetings VARCHAR2(20);
  3  BEGIN
  4          greetings := 'Hello World';
  5          dbms_output.put_line(greetings);
  6  END my_first_proc;
  7  /

Procedure created.

SQL>
SQL> EXECUTE my_first_proc;
Hello World

PL/SQL procedure successfully completed.

SQL>
SQL> BEGIN
  2          my_first_proc;
  3  END;
  4  /
Hello World

PL/SQL procedure successfully completed.

SQL>
```

FIGURE 8-2. *Simple "Hello World" PL/SQL procedure*

When a procedure has formal parameters with default values and when they are all at the end of the list of formal parameters in the procedure specification, the procedure may be called without specifying values for the last few formal parameters for which default values exist. However, all the formal parameters for which actual parameters are being supplied at call time must be listed before all of the formal parameters for which no actual parameters are being supplied. The call will then look like the following:

```
procedure_name(actual_param1,
               actual_param2,
               ...
               actual_paramN);
```

N may be less than or equal to the number of formal parameters for the procedure and *N* must be greater than or equal to the number of formal parameters for which default values do not exist.

When the default-valued formal parameters are not the last parameters in the specification, or when you wish to avoid having PL/SQL figure out which actual parameter corresponds to which formal parameter using its order in the list, you can specifically tell PL/SQL which actual parameter is for which formal parameter using the following syntax:

```
procedure_name(formal_param1 => actual_param1,
               formal_param2 => actual_param2,
                ...
                )
;
```

This is called *named notation* for calling functions and procedures. The earlier notation is called *positional notation* as the parameters are matched by their position in the list.

The same calling methods apply to functions. Functions, however, can appear within other expressions and may not have any semicolon at the end. You will see an example for named notation in the next section. It is possible to mix two notations, but the positional list must precede the notational list in the call.

PL/SQL Variables and Constants

You have seen some examples of PL/SQL variables in previous sections. Now we will discuss them in greater detail. Variables are essentially containers with name tags. They can contain or hold information or data of different kinds. Based on the kind of data they can hold, they have different datatypes and to distinguish them from one another they have names. Just as oil comes in a bottle and flour in a paper bag, PL/SQL will store numbers in variables of the NUMBER datatype and text in variables of the CHAR or VARCHAR2 datatypes. Taking it a step further, imagine the refrigerator in your company's break room. It's filled with brown paper bags that contain your lunch and the lunches of your co-workers. How will you find your noontime feast amongst all the other bags? Right! You'd put your name on the bag. Variables are given names, too, in order to avoid confusion. Further, if your lunch consisted of only bananas you may eat them and put the peels back into the brown paper bag. Now the contents of the bag have changed. Similarly, the contents of variables can be changed during the execution of PL/SQL statements.

Declare PL/SQL Variables

The syntax for declaring a variable in PL/SQL is either of the following:

```
variable_name data_type  [ [NOT NULL]  := default_value_expression];
variable_name data_type [ [NOT NULL] DEFAULT default_value_expression];
```

variable_name is any valid *PL/SQL identifier*. A valid PL/SQL identifier has the following properties:

Up to 30 characters long and has no white space of any form in it (as space or tabs).

- Made up of letters, digits 0 to 9, underscore (_), dollar ($), and pound (#) signs.
- Starts with a letter.
- Is not the same as a *PL/SQL* or an *SQL reserved word,* which has special meaning for PL/SQL or SQL. For example, a variable name cannot be BEGIN. BEGIN has a special meaning telling PL/SQL that here starts the beginning of a basic block execution section.

The use of NOT NULL requires that the variable have a value and, if specified, the variable must be given a default value.

When a variable is created it can be made to have a value specified by the default value expression. It is just a shorthand way to assign values to variables.

You already know about SQL datatypes—NUMBER, VARCHAR2, and DATE. PL/SQL shares them with SQL. PL/SQL has additional datatypes that are not in SQL. For a complete list, please refer to Oracle PL/SQL references.

Declare PL/SQL Constants

The syntax for declaring a constant is the following:

```
variable_name data_type  CONSTANT  := constant_value_expression;
```

Unlike variables, constants must be given a value and that value cannot change during the life or scope of the constant. Constants are very useful for enforcing safe and disciplined code development in large and complex applications. For example, if you want to ensure that the data passed to a PL/SQL procedure is not modified by the procedure, you can make that data a constant. If the procedure tries to modify it, PL/SQL will return with an exception.

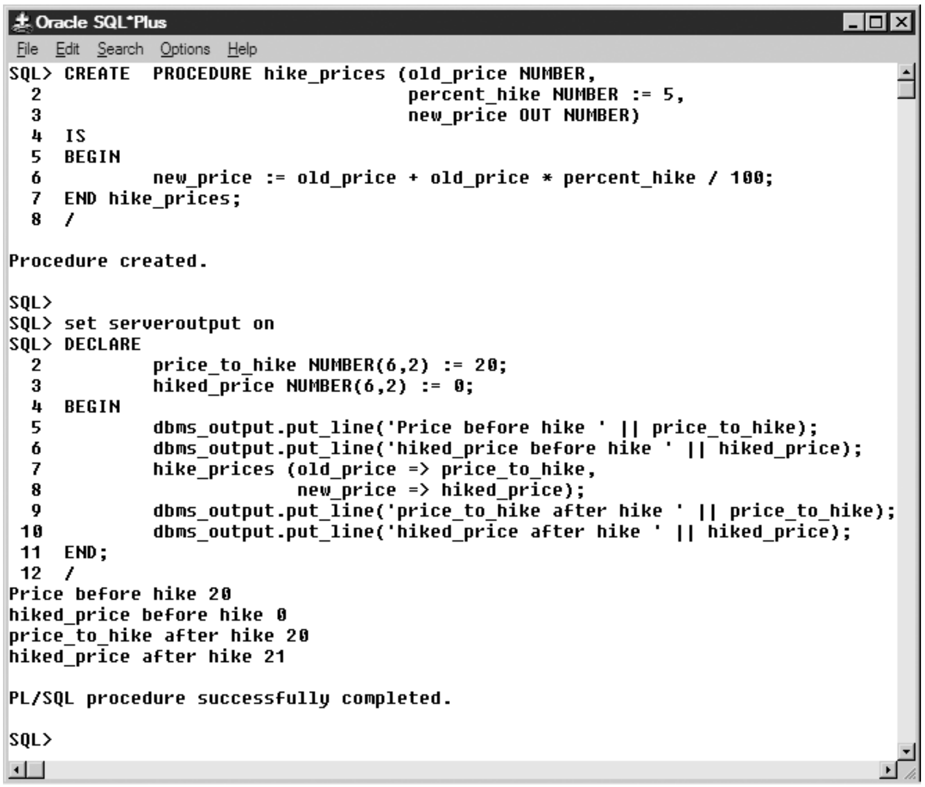
Assign Values to Variables

There are three ways a variable can get its value changed. First is the assignment of a valid expression to it using the PL/SQL assignment operator. You have seen a number of examples of this kind. The syntax is the following:

```
variable_name := expression ;
```

Second, a variable can be passed as the actual parameter corresponding to some IN OUT or OUT formal parameter when calling a PL/SQL procedure. After the procedure is finished the value of the variable may change. The following example shows the named notation for calling procedures. Refer to Figure 8-3 for the expected output.

L 8-13
```
CREATE  PROCEDURE hike_prices (old_price NUMBER,
                               percent_hike NUMBER := 5,
                               new_price OUT NUMBER)
IS
BEGIN
        new_price := old_price + old_price * percent_hike / 100;
END hike_prices;
/
```



```
SQL> CREATE  PROCEDURE hike_prices (old_price NUMBER,
                                    percent_hike NUMBER := 5,
                                    new_price OUT NUMBER)
     IS
     BEGIN
             new_price := old_price + old_price * percent_hike / 100;
     END hike_prices;
     /

Procedure created.

SQL>
SQL> set serveroutput on
SQL> DECLARE
             price_to_hike NUMBER(6,2) := 20;
             hiked_price NUMBER(6,2) := 0;
     BEGIN
             dbms_output.put_line('Price before hike ' || price_to_hike);
             dbms_output.put_line('hiked_price before hike ' || hiked_price);
             hike_prices (old_price => price_to_hike,
                          new_price => hiked_price);
             dbms_output.put_line('price_to_hike after hike ' || price_to_hike);
             dbms_output.put_line('hiked_price after hike ' || hiked_price);
     END;
     /
Price before hike 20
hiked_price before hike 0
price_to_hike after hike 20
hiked_price after hike 21

PL/SQL procedure successfully completed.

SQL>
```

FIGURE 8-3. *Assign values to PL/SQL variables by using them as actual parameters*

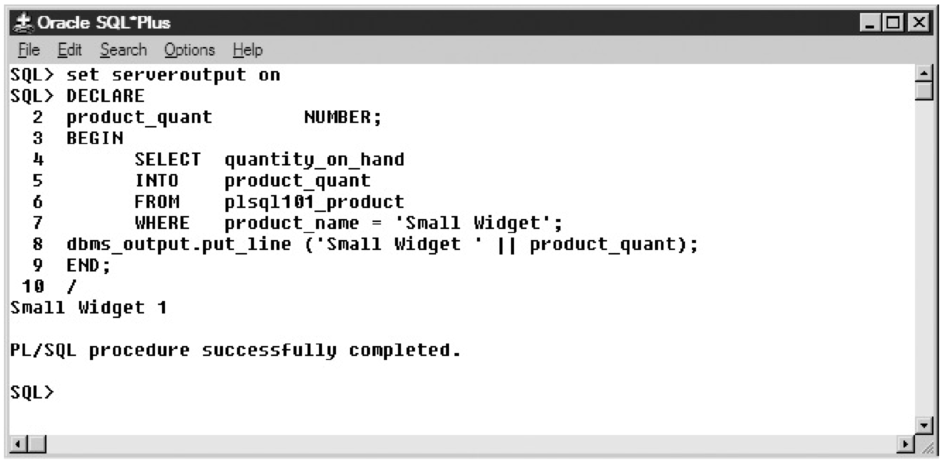
The following procedure shows the variables changing their values:

L 8-14
```
set serveroutput on
DECLARE
        price_to_hike NUMBER(6,2) := 20;
        hiked_price NUMBER(6,2) := 0;
BEGIN
        dbms_output.put_line('Price before hike ' || price_to_hike);
        dbms_output.put_line('hiked_price before hike ' || hiked_price);
        hike_prices (old_price => price_to_hike,
                     new_price => hiked_price);
        dbms_output.put_line('price_to_hike after hike ' || price_to_hike);
        dbms_output.put_line('hiked_price after hike ' || hiked_price);
END;
/
```

The following is a quick example with Figure 8-4 showing the results:

L 8-15
```
set serveroutput on
DECLARE
product_quant        NUMBER;
BEGIN
      SELECT  quantity_on_hand
      INTO    product_quant
      FROM    plsql101_product
      WHERE   product_name = 'Small Widget';
dbms_output.put_line ('Small Widget ' || product_quant);
END;
/
```

product_quant is assigned the value equal to the quantity of small widgets.



```
SQL> set serveroutput on
SQL> DECLARE
  2  product_quant        NUMBER;
  3  BEGIN
  4        SELECT  quantity_on_hand
  5        INTO    product_quant
  6        FROM    plsql101_product
  7        WHERE   product_name = 'Small Widget';
  8  dbms_output.put_line ('Small Widget ' || product_quant);
  9  END;
 10  /
Small Widget 1

PL/SQL procedure successfully completed.

SQL>
```

FIGURE 8-4. *Assign values to PL/SQL variables using SQL*

Use Variables

Variables are the very basic units of PL/SQL programs. They are used to hold results of computations, to return values from function calls, as actual parameters for calling functions and procedures, and so on. Variables should be used to make your application clean and easy to read, thereby creating a lower maintenance, more efficient program.

Suppose you want to perform a number of calculations using the current quantity of small widgets—compare it with the quantity from three months ago, or to the quantity of medium widgets. By using the variable to hold the value, you avoid the delay that would come from getting the quantity from the table again and again.

By naming variables in a way that makes sense to you, you can make your code easy to read and understand. The same principle applies when you use variables to hold the results of some very complex expressions instead of repeating the expressions in the code in multiple places.

Control Structures in PL/SQL

Many times you want to do one thing if something is true and something else if it is not true. For example, if a purchase order exceeds a certain dollar amount you would like to take 5 percent off the order, and maybe 10 percent off if the order exceeds some other amount. This kind of logic may be required inside your application that prints out the final invoice for your customers. This is *conditional processing* of data. Based on the condition, different parts of the code need to be executed.

Recall the case where you need to compute income tax for each employee. You need to complete a function for each employee, such as finding the earnings and filing status, and then apply the correct formula to find the tax. The correct formula differs for each employee based on filing status and all of the other factors. This is an example of an *iterative operation*.

PL/SQL provides you with the ability to do conditional and iterative processing. The constructs it provides are said to cause change of *program flow* and so control *the flow of the execution*.

IF Statement

The syntax for an IF statement is as follows:

```
IF condition_1 THEN
    actions_1;
[ELSIF condition_2 THEN
    actions_2;]
...
[ELSE
    actions_last;]
END IF;
```

actions_1 to *actions_last* represent one or more PL/SQL statements. Each set of statements gets executed only if its corresponding condition is true. When one of the IF conditions is determined to be true, the rest of the conditions are not checked.

Enter the following example and see that your results match those of Figure 8-5:

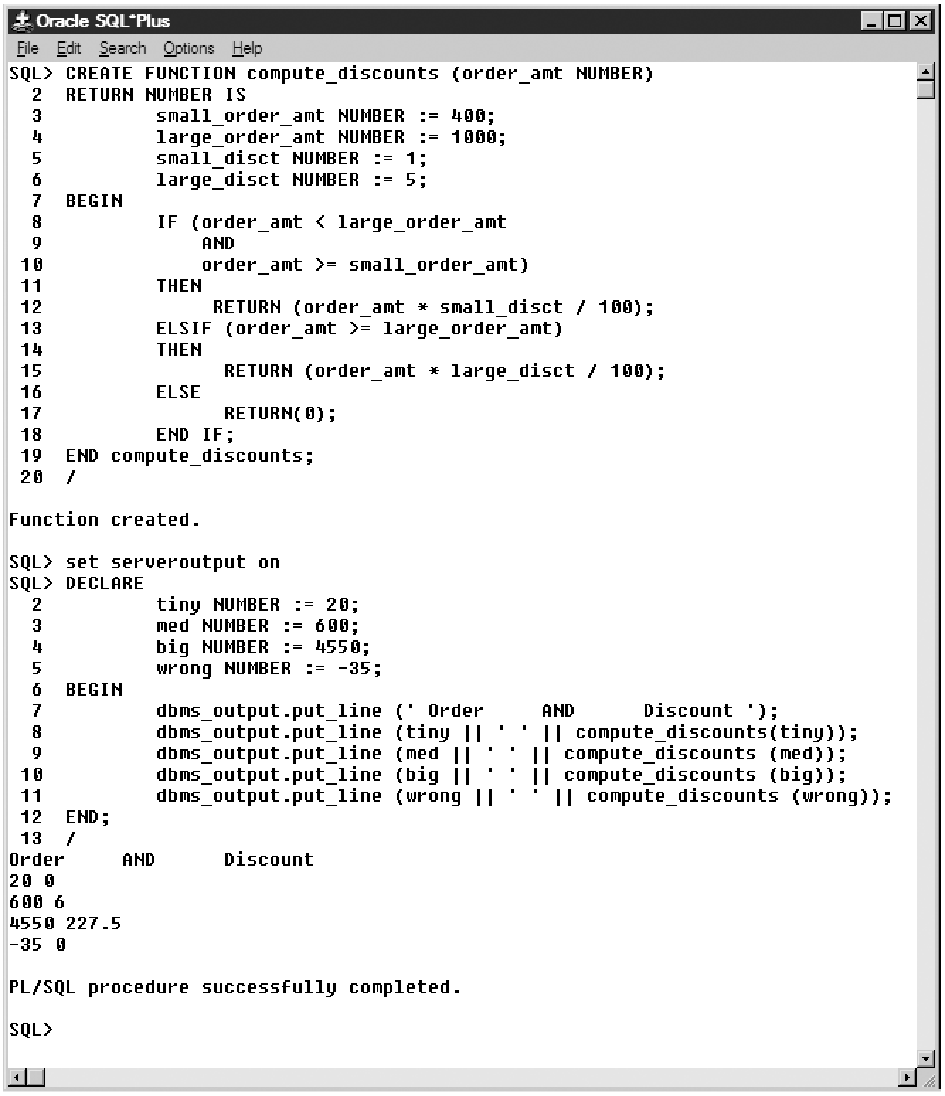
L 8-16
```
-- Compute discounts on orders.
-- Input order amount. Returns discount amount (zero for wrong inputs).
CREATE FUNCTION compute_discounts (order_amt NUMBER)
RETURN NUMBER IS
        small_order_amt NUMBER := 400;
        large_order_amt NUMBER := 1000;
        small_disct NUMBER := 1;
        large_disct NUMBER := 5;
BEGIN
        IF (order_amt < large_order_amt
            AND
            order_amt >= small_order_amt)
        THEN
             RETURN (order_amt * small_disct / 100);
        ELSIF (order_amt >= large_order_amt)
        THEN
              RETURN (order_amt * large_disct / 100);
        ELSE
              RETURN(0);
        END IF;
END compute_discounts;
/
```

This function will give a 1 percent discount for orders between 400 and 1000 and a 5 percent discount on orders above 1000. It will return zero for all other amounts including wrong values. For example, someone may try to use a negative value for *order_amt*, which is meaningless.

Observe at the start how the function is clearly documented. You should always consider all possibilities when writing your code and either clearly state in your documentation what you are going to do about error conditions or, if the conditions are severe enough, give appropriate error messages. Suppose in our case—however unimaginable it is—that this function may be called with a negative value for the *order_amt*, we have documented what the function will do in such a case.

You can test the function by calling it in an anonymous block. Be sure you have serveroutput on. Refer once again to Figure 8-5 for this example.

L 8-17
```
set serveroutput on
DECLARE
        tiny NUMBER := 20;
        med NUMBER := 600;
        big NUMBER := 4550;
        wrong NUMBER := -35;
BEGIN
        dbms_output.put_line (' Order     AND      Discount ');
        dbms_output.put_line (tiny || ' ' || compute_discounts(tiny));
        dbms_output.put_line (med || ' ' || compute_discounts (med));
        dbms_output.put_line (big || ' ' || compute_discounts (big));
        dbms_output.put_line (wrong || ' ' || compute_discounts (wrong));
END;
/
```



```
SQL> CREATE FUNCTION compute_discounts (order_amt NUMBER)
  2  RETURN NUMBER IS
  3          small_order_amt NUMBER := 400;
  4          large_order_amt NUMBER := 1000;
  5          small_disct NUMBER := 1;
  6          large_disct NUMBER := 5;
  7  BEGIN
  8          IF (order_amt < large_order_amt
  9              AND
 10              order_amt >= small_order_amt)
 11          THEN
 12               RETURN (order_amt * small_disct / 100);
 13          ELSIF (order_amt >= large_order_amt)
 14          THEN
 15                RETURN (order_amt * large_disct / 100);
 16          ELSE
 17                RETURN(0);
 18          END IF;
 19  END compute_discounts;
 20  /

Function created.

SQL> set serveroutput on
SQL> DECLARE
  2          tiny NUMBER := 20;
  3          med NUMBER := 600;
  4          big NUMBER := 4550;
  5          wrong NUMBER := -35;
  6  BEGIN
  7          dbms_output.put_line (' Order     AND      Discount ');
  8          dbms_output.put_line (tiny || ' ' || compute_discounts(tiny));
  9          dbms_output.put_line (med || ' ' || compute_discounts (med));
 10          dbms_output.put_line (big || ' ' || compute_discounts (big));
 11          dbms_output.put_line (wrong || ' ' || compute_discounts (wrong));
 12  END;
 13  /
Order     AND      Discount
20 0
600 6
4550 227.5
-35 0

PL/SQL procedure successfully completed.

SQL>
```

FIGURE 8-5. *Example of an IF statement*

Loops

PL/SQL provides three different iteration constructs. Each allows you to repeatedly execute a set of PL/SQL statements. You stop the repeated executions based on some condition.

LOOP

The syntax for the LOOP construct is as follows:

<<*loop_name*>>
LOOP
    statements;
    EXIT *loop_name* [WHEN *exit_condition_expression*];
    statements;
END LOOP ;

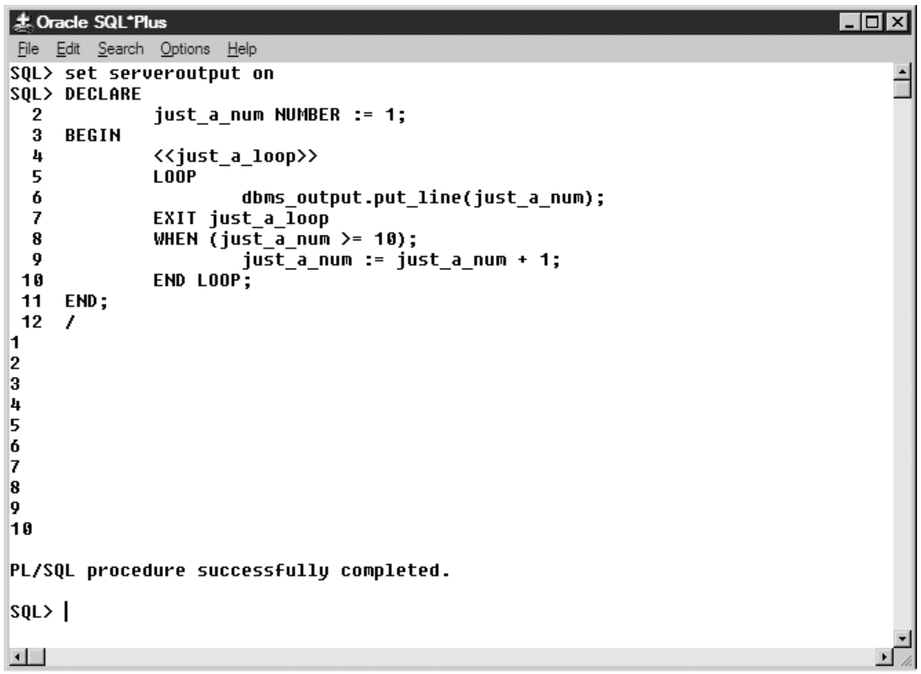
All the statements within the loop are executed repeatedly. During each repetition or *iteration* of the loop, the exit condition expression is checked for a positive value if the WHEN condition is present. If the expression is true, then the execution skips all statements following the EXIT and jumps to the first statement after END LOOP within the code. No more iterations are done. If the WHEN condition is not present, the effect is to execute statements between LOOP and EXIT only once. You will obviously be doing something illogical if you are not using the WHEN condition. After all, the idea of a loop is to potentially loop through the code.

Try out the following loop example and compare the results with Figure 8-6. It simply prints out the first ten numbers.

As usual, do not forget to set serveroutput on to see the output.

L 8-18
```
set serveroutput on
DECLARE
        just_a_num NUMBER := 1;
BEGIN
        <<just_a_loop>>
        LOOP
                dbms_output.put_line(just_a_num);
        EXIT just_a_loop
        WHEN (just_a_num >= 10);
                just_a_num := just_a_num + 1;
        END LOOP;
END;
/
```

Each iteration increments the variable *just_a_num* by 1. When 10 is reached, the exit condition is satisfied and the loop is exited.



```
SQL> set serveroutput on
SQL> DECLARE
            just_a_num NUMBER := 1;
     BEGIN
            <<just_a_loop>>
            LOOP
                    dbms_output.put_line(just_a_num);
            EXIT just_a_loop
            WHEN (just_a_num >= 10);
                    just_a_num := just_a_num + 1;
            END LOOP;
     END;
     /
1
2
3
4
5
6
7
8
9
10

PL/SQL procedure successfully completed.

SQL>
```

FIGURE 8-6. *Example of a simple LOOP*

WHILE Loop

Another type of loop is the WHILE loop. A WHILE loop is well suited for situations when the number of loop iterations is not known in advance, but rather is determined by some external factor. The syntax for a WHILE loop is as follows:

```
WHILE while_condition_expression
LOOP
      statements;
END LOOP;
```

Practice creating a WHILE loop by entering the following code. Your results should match those of Figure 8-7.

L 8-19

```
set serveroutput on
DECLARE
     just_a_num NUMBER := 1;
```

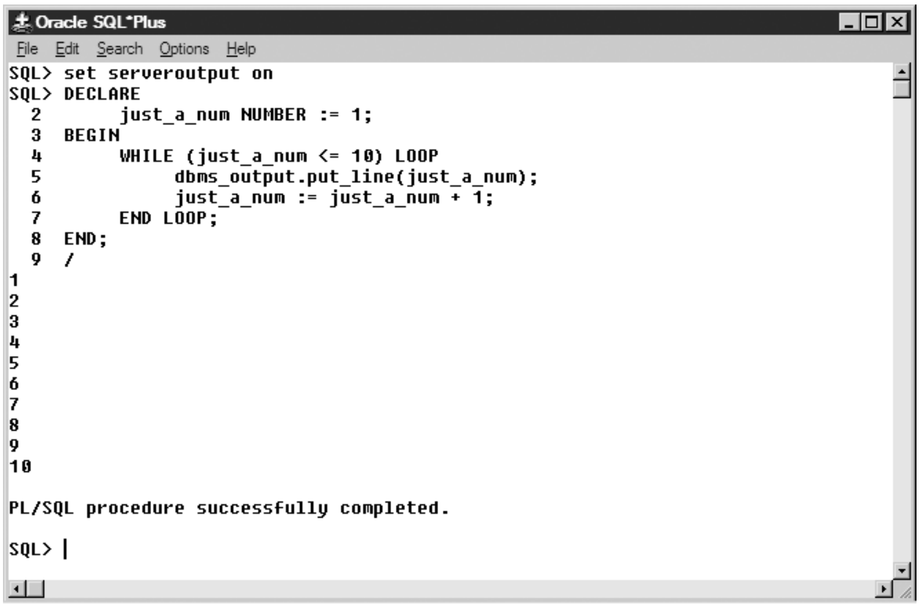
```
BEGIN
     WHILE (just_a_num <= 10) LOOP
          dbms_output.put_line(just_a_num);
          just_a_num := just_a_num + 1;
     END LOOP;
END;
/
```

NOTE
The condition for the WHILE must be true every time before entering the loop.

FOR Loop

The FOR loop uses a counter variable, also called a *loop index*, to count the number of iterations. The counter is incremented, starting from the lower limit specified, or decremented, starting from the upper limit specified, at the end of each iteration or loop. If it is out of the range, the looping stops. The syntax for the FOR loop is as follows:

FOR *counter* IN [REVERSE] *lower_bound .. upper_bound*
LOOP
    statements;
END LOOP;

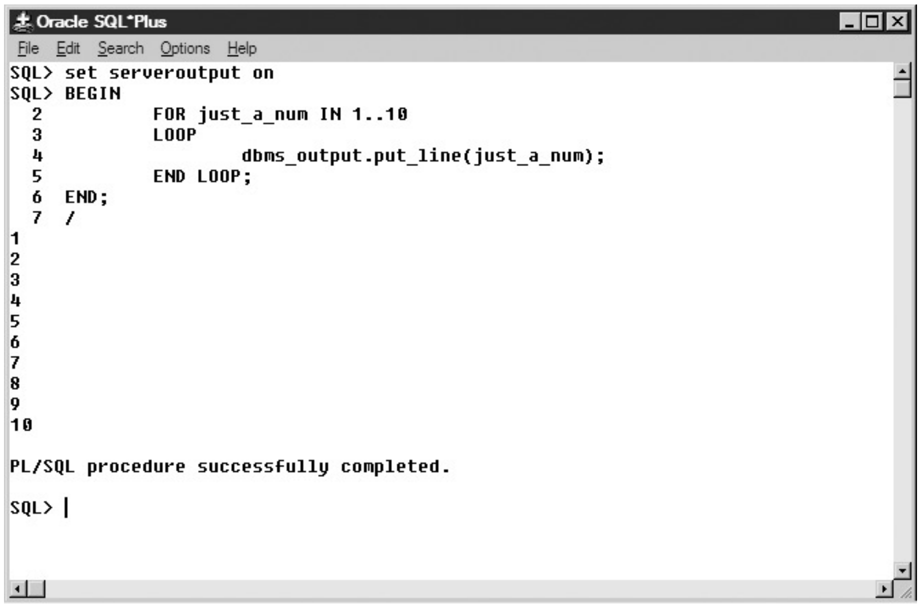


```
SQL> set serveroutput on
SQL> DECLARE
  2       just_a_num NUMBER := 1;
  3  BEGIN
  4       WHILE (just_a_num <= 10) LOOP
  5            dbms_output.put_line(just_a_num);
  6            just_a_num := just_a_num + 1;
  7       END LOOP;
  8  END;
  9  /
1
2
3
4
5
6
7
8
9
10

PL/SQL procedure successfully completed.

SQL> |
```

FIGURE 8-7. *Example of a WHILE loop*



```
SQL> set serveroutput on
SQL> BEGIN
  2          FOR just_a_num IN 1..10
  3          LOOP
  4                  dbms_output.put_line(just_a_num);
  5          END LOOP;
  6  END;
  7  /
1
2
3
4
5
6
7
8
9
10

PL/SQL procedure successfully completed.

SQL> |
```

FIGURE 8-8. *Example of a FOR loop*

Now create your first FOR loop using the following code. Your results should match those of Figure 8-8.

L 8-20

```
set serveroutput on
BEGIN
        FOR just_a_num IN 1..10
        LOOP
                dbms_output.put_line(just_a_num);
        END LOOP;
END;
/
```

Now for fun and experience, try using the REVERSE command in your FOR loop. Your results should show the numbers in reverse order from 10 to 1.

Cursors

The cursor is an extremely important PL/SQL construct. It is the heart of PL/SQL and SQL cooperation and stands for "current set of records." A *cursor* is a special PL/SQL element that has an associated SQL SELECT statement. Using a cursor, each row of the SQL statement associated with the cursor can be processed one at a time. A cursor is declared in the declaration section of a basic block. A cursor is opened using the command OPEN and rows can be fetched using the command FETCH. After all processing is done, the cursor is closed using the CLOSE command. Closing the cursor releases all of the system resources that were used while the cursor was open. You can lock the rows selected by a cursor to prevent other people from modifying them while you are using them. Closing the cursor or executing an explicit COMMIT or ROLLBACK will unlock the rows.

PL/SQL uses hidden or *implicit cursors* for SQL statements within PL/SQL code. We discuss them more in Chapter 9. In this section we will focus on *explicit cursors*, which simply means cursors that have been assigned a name.

We will write a simple procedure that uses a cursor to compute the commissions for all salespersons. Before we do that, however, take a look at the syntax for an explicit cursor.

Cursor Declaration and Cursor Attributes

A cursor is declared within a PL/SQL procedure in the following manner:

```
CURSOR cursor_name [( [parameter1 [, parameter2 ...])]
[RETURN return_specification]
IS
    select_statement
        [FOR UPDATE
            [OF table_or_col1
                [, table_or_col2 ...]
            ]
        ]
;
```

The parameters are similar to procedure parameters, but they are all IN parameters. They cannot be OUT or IN OUT because the cursor cannot modify them. The parameters are used in the WHERE clause of the cursor SELECT statement. The return specification tells what type of records will be selected by the SELECT statement. You will learn more about PL/SQL records in Chapter 9. The *table_or_col* is a column name you intend to update or a table name from which you intend to delete or update rows; it must be taken from the names of tables and columns used within the cursor SELECT statement. It is used to clearly document what may potentially be modified by the code that uses this cursor. The FOR UPDATE commands lock the rows selected by the SELECT statement when the cursor is opened, and they remain locked until you close the cursor in the ways already discussed.

A cursor has some indicators to show its state and they are called *attributes of the cursor*. The attributes are shown in Table 8-1.

Attribute	**Description**
cursor_name%ISOPEN	Checks if the cursor is open. It returns TRUE if the cursor *cursor_name* is already open.
cursor_name%ROWCOUNT	The number of table rows returned by the cursor SELECT statement.
cursor_name%FOUND	Checks whether the last attempt to get a record from the cursor succeeded. It returns TRUE if a record was fetched.
cursor_name%NOTFOUND	Opposite of the FOUND attribute. It returns TRUE when no more records are found.

TABLE 8-1. *Cursor Attributes*

PL/SQL Records

Although PL/SQL records will be discussed in greater detail in Chapter 9, you'll need to know a little something about them before we proceed. So let's start with a brief introduction in this chapter.

A PL/SQL record is a collection of basic types of data and can be accessed as a single unit. You access the individual fields of the record using the *record_name.field_name* notation you are already familiar with for use with table columns. Records are of three types and you can declare variables of record types. The three types of records are the following:

- **Table-Based** This record has fields that match the names and types of the table columns. So if a cursor selects the entire row—by using SELECT* from *some_table,* for example—the records it returns can be directly copied into the variable of the table-based record type for *some_table*.
- **Cursor-Based** This record has fields that match in name, datatype, and order to the final list of columns in the cursor's SELECT statement.
- **Programmer-Defined** These are records in which you define a record type.

Use OPEN, FETCH, and CLOSE Cursor

The following is the syntax for opening, fetching from, and closing a cursor:

```
OPEN cursor_name;
FETCH cursor_name INTO record_var_or_list_of_var;
CLOSE cursor_name;
```

When opened, a cursor contains a set of records if the cursor's SELECT statement was successful and resulted in fetching selected rows from the database. Each FETCH then removes a record from the open cursor and moves the record's contents into either a PL/SQL variable—of a record type that matches the record type of the cursor record—or into a different set of PL/SQL variables such that each variable in the list matches in type with the corresponding field in the cursor record.

You will check if there are any more records left in the cursor before trying to fetch one from the cursor using the FOUND and NOTFOUND attributes of the cursor. Fetching from an empty cursor will fetch the last fetched record over and over again and will not give you any error. So make sure you use FOUND or NOTFOUND if you are using FETCH.

The actual processing of records from a cursor usually occurs within a loop. When writing the loop, it's a good idea to start by checking whether a record has been found in the cursor. If so, the code proceeds to perform whatever processing you need; if not, the code exits from the loop. There is a more compact way to do the same where PL/SQL takes care of opening, fetching, and closing without your needing to do it—the cursor FOR loop.

Cursor FOR Loop

The syntax for the cursor FOR loop is the following:

```
FOR cursor_record IN cursor_name LOOP
    statements;
END LOOP;
```

This cursor FOR loop continues fetching records from the cursor into the *cursor_record* record type variable. You can use *cursor_record* fields to access the data within your PL/SQL statements in the loop. When all the records are done, the loop ends. The cursor is automatically opened and closed for your convenience by PL/SQL.

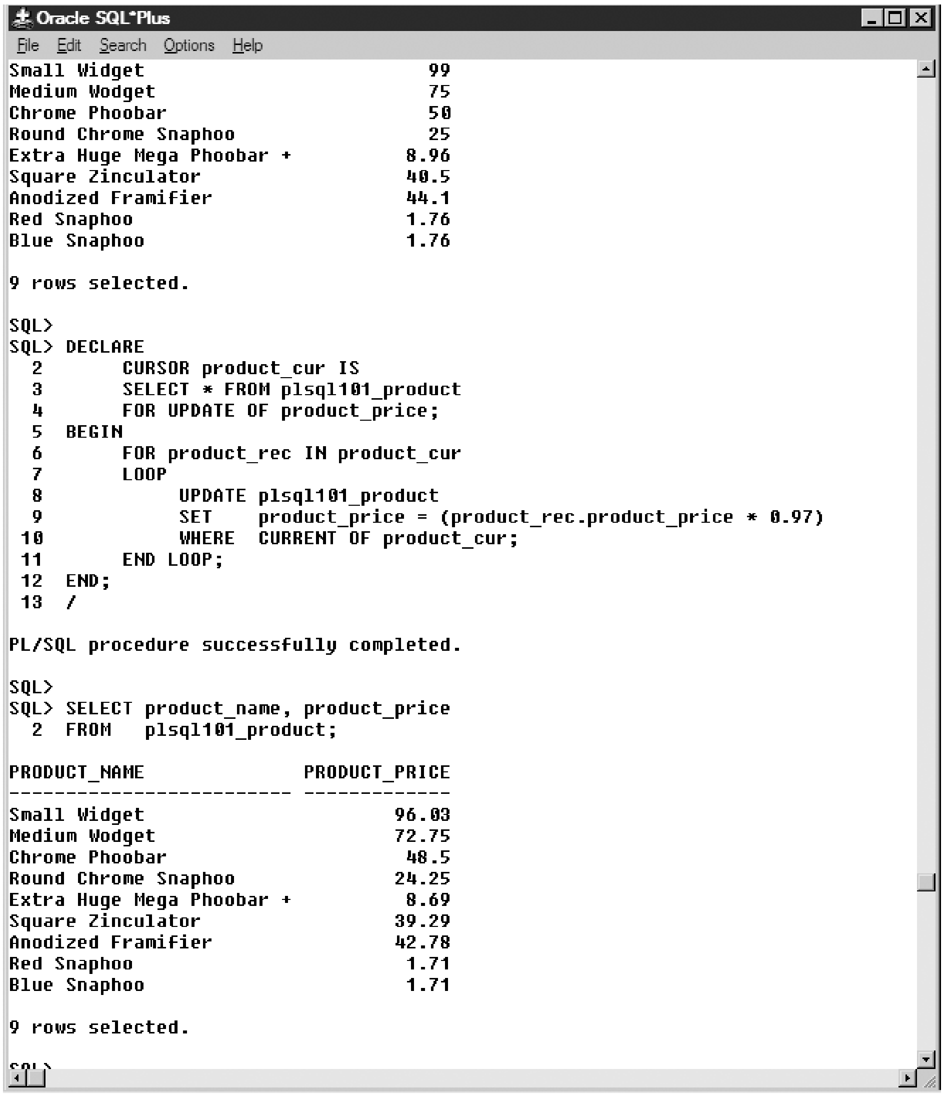
You will receive an *invalid cursor* message if you try to fetch from a cursor that is not open. If you do not close cursors, you may end up eventually running into the maximum number of open cursors that the system allows.

WHERE CURRENT OF

When the cursor is opened in order to update or delete the rows it selects, you can use

```
WHERE CURRENT OF cursor_name
```

to access the table and row corresponding to the most recently fetched record in the WHERE clause of the UPDATE or DELETE statement. For example, to reduce the prices in the PLSQL101_PRODUCT table by 3 percent, type the following code and check your results against those in Figure 8-9.



```
Small Widget                         99
Medium Wodget                        75
Chrome Phoobar                       50
Round Chrome Snaphoo                 25
Extra Huge Mega Phoobar +          8.96
Square Zinculator                  40.5
Anodized Framifier                 44.1
Red Snaphoo                        1.76
Blue Snaphoo                       1.76

9 rows selected.

SQL>
SQL> DECLARE
  2       CURSOR product_cur IS
  3       SELECT * FROM plsql101_product
  4       FOR UPDATE OF product_price;
  5  BEGIN
  6       FOR product_rec IN product_cur
  7       LOOP
  8            UPDATE plsql101_product
  9            SET    product_price = (product_rec.product_price * 0.97)
 10            WHERE  CURRENT OF product_cur;
 11       END LOOP;
 12  END;
 13  /

PL/SQL procedure successfully completed.

SQL>
SQL> SELECT product_name, product_price
  2  FROM   plsql101_product;

PRODUCT_NAME              PRODUCT_PRICE
------------------------- -------------
Small Widget                      96.03
Medium Wodget                     72.75
Chrome Phoobar                     48.5
Round Chrome Snaphoo              24.25
Extra Huge Mega Phoobar +          8.69
Square Zinculator                 39.29
Anodized Framifier                42.78
Red Snaphoo                        1.71
Blue Snaphoo                       1.71

9 rows selected.
```

FIGURE 8-9. *Examples of a cursor FOR loop and WHERE CURRENT OF clause*

L 8-21
```
SELECT product_name, product_price
FROM   plsql101_product;

DECLARE
     CURSOR product_cur IS
     SELECT * FROM plsql101_product
     FOR UPDATE OF product_price;
BEGIN
     FOR product_rec IN product_cur
     LOOP
          UPDATE plsql101_product
          SET    product_price = (product_rec.product_price * 0.97)
          WHERE  CURRENT OF product_cur;
     END LOOP;
END;
/

SELECT product_name, product_price
FROM   plsql101_product;
```

Nested Loops and Cursor Example

The following code demonstrates complete use of cursors and loops within loops or *nested loops*. Enter the following code:

L 8-22
```
-- This procedure computes the commissions for salespersons.
-- It prints out the salesperson's code, his or her total sales,
-- and corresponding commission.
-- No inputs. No errors are reported and no exceptions are raised.
/* Logic: A cursor to create a join between PLSQL101_PRODUCT and
PLSQL101_PURCHASE on PRODUCT_NAME column is done.
The result is ordered by salesperson.
Outer loop starts with a new salesperson and inner loop
processes all rows for one salesperson.
*/
CREATE OR REPLACE PROCEDURE do_commissions IS
     commission_rate NUMBER  := 2   ;
     total_sale      NUMBER  := 0   ;
     current_person  CHAR(3) := ' ' ;
     next_person     CHAR(3)        ;
     quantity_sold   NUMBER  := 0   ;
     item_price      NUMBER  := 0   ;
     CURSOR sales_cur IS
          SELECT purc.salesperson,
                 purc.quantity,
                 prod.product_price
          FROM   plsql101_purchase purc,
                 plsql101_product  prod
          WHERE  purc.product_name = prod.product_name
          ORDER BY salesperson;
BEGIN
     OPEN sales_cur;
     LOOP
          FETCH sales_cur INTO
                next_person, quantity_sold, item_price;
          WHILE (next_person = current_person
                 AND
                 sales_cur%FOUND)
          LOOP
               total_sale :=
                     total_sale + (quantity_sold * item_price);
                FETCH sales_cur INTO
                     next_person, quantity_sold, item_price;
          END LOOP;
          IF (sales_cur%FOUND)
          THEN
               IF (current_person != next_person)
               THEN
                    IF (current_person != ' ' )
                    THEN
                         dbms_output.put_line
                            (current_person ||
                             ' ' ||
                             total_sale ||
                             ' ' ||
                             total_sale * commission_rate / 100);
                    END IF;
                    total_sale := quantity_sold * item_price;
                    current_person := next_person;
               END IF;
          ELSE IF (current_person != ' ')
          THEN
                 dbms_output.put_line(current_person ||
                               ' ' ||
                               total_sale ||
                               ' ' ||
                               total_sale * commission_rate / 100);
               END IF;
          END IF;
          EXIT WHEN sales_cur%NOTFOUND;
          END LOOP;
          CLOSE sales_cur;
     END do_commissions;
/
```

First look at the cursor's SELECT statement. It lists, from the PLSQL101_PURCHASE table, the quantities of items sold. It also shows their corresponding prices from the PLSQL101_PRODUCT table. This is achieved by creating a join. The result is ordered by salesperson so that we have all of the records for a particular salesperson together.

Once the cursor is opened and the first row is fetched, the condition for WHILE is checked. The first FETCH command for the current person has a value that cannot match any salesperson code; recall that its default value is a single space ( ). Therefore, the loop is skipped and we jump to the first IF statement. The IF statement checks to see if the last FETCH command returned any records. If records were returned, a check is made to see whether the *current_person* and *next_person* values match. If they don't match, we know that the last FETCH is the start of a new salesperson and it is time to print out the commissions for the current salesperson. Note that the first record for *current_person* is not valid; therefore, the IF check will fail and nothing will print.

The next statement sets the value for *total_sale* to be equal to the cost for the very first product. The statement after that stores the *next_person* value into the *current_person* variable. Now we will be back to the first FETCH in the loop as the loop's EXIT condition is not yet true. This FETCH may result in the same value for *next_person* as the *current_person*, in which case we have more than one entry for the current person in our list of sales. When that is the case, the WHILE loop is entered and the cost for items is added to the total sale amount. This loop will keep adding costs by fetching new records from the cursor until a new salesperson is identified. This process repeats over and over until there are no records left in the cursor. At that point, the validity of the *current_person* is checked. If the *current_person* is valid, then the very last IF statement prints out sales and commissions for that person; the commissions are calculated using the value in the *commission_rate* constant.

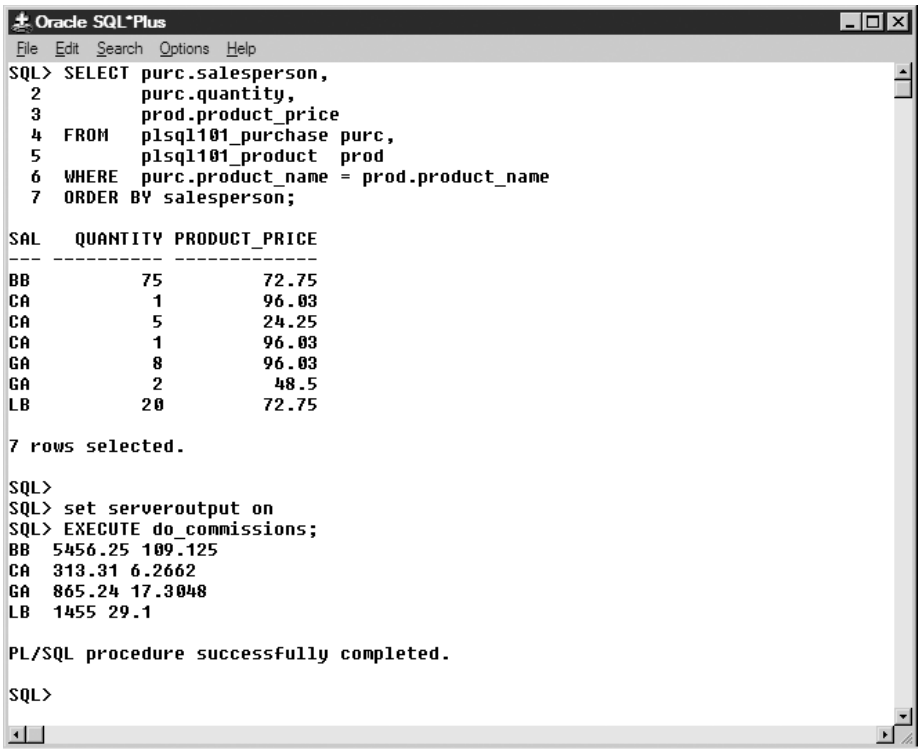
To test the procedure, enter the following commands and compare your results with those in Figure 8-10. The first command shows the raw records in the PLSQL101_ PURCHASE table, while the second command causes the DO_COMMISSIONS procedure to subtotal the sales in those records and calculate appropriate commissions for each salesperson.

L 8-23
```
SELECT purc.salesperson,
       purc.quantity,
       prod.product_price
FROM   plsql101_purchase purc,
       plsql101_product  prod
WHERE  purc.product_name = prod.product_name
ORDER BY salesperson;

set serveroutput on
EXECUTE do_commissions;
```

Error Handling

It is important to issue user-friendly error messages when error conditions occur. Earlier in this chapter, the section on basic PL/SQL blocks included a mention of exceptions; now it is time to get into more detail.



```
SQL> SELECT purc.salesperson,
  2         purc.quantity,
  3         prod.product_price
  4  FROM   plsql101_purchase purc,
  5         plsql101_product  prod
  6  WHERE  purc.product_name = prod.product_name
  7  ORDER BY salesperson;

SAL   QUANTITY PRODUCT_PRICE
--- ---------- -------------
BB          75         72.75
CA           1         96.03
CA           5         24.25
CA           1         96.03
GA           8         96.03
GA           2          48.5
LB          20         72.75

7 rows selected.

SQL>
SQL> set serveroutput on
SQL> EXECUTE do_commissions;
BB  5456.25 109.125
CA  313.31 6.2662
GA  865.24 17.3048
LB  1455 29.1

PL/SQL procedure successfully completed.

SQL>
```

FIGURE 8-10. *Example of nested loops*

Exceptions

An exception is an error state that is activated—or *raised*—when a specific problem occurs. There are many different exceptions, each relating to a different type of problem. When an exception is raised, the code execution stops at the statement that raised the exception, and control is passed to the exception-handling portion of the block. If the block does not contain an executable section, PL/SQL tries to find an executable section in the *enclosing basic block,* which is an outer block of code surrounding the block in which the exception was raised. If the immediate enclosing block does not have an exception handler to accommodate the raised exception, then the search continues to the next enclosing block and so on until a proper exception handler is found or, if not found, execution is halted with an unhandled exception error.

The exception-handling portion of a block is the perfect opportunity to issue meaningful error messages and clean up anything that could cause confusion or trouble later. A typical cleanup could involve issuing the ROLLBACK statement if an exception is raised during a procedure that has inserted rows into a table.

Once control is passed to the exception handler, control is not returned to the statement that caused the exception. Instead, control is passed to the enclosing basic block at the point just after the enclosed block or procedure/function was called.

System-Defined Exceptions

You are familiar with the ZERO_DIVIDE exception predefined by PL/SQL. There are quite a few other system-defined exceptions that are detected and raised by PL/SQL or Oracle. Table 8-2 provides a more complete list of system-defined exceptions.

System-Defined Exception	**Description**
CURSOR_ALREADY_OPEN	Tried to open an already open cursor.
DUP_VAL_ON_INDEX	Attempted to insert duplicate value in column restricted by unique index to be unique.
INVALID_CURSOR	Tried to FETCH from cursor that was not open or tried to close a cursor that was not open.
NO_DATA_FOUND	Tried to SELECT INTO when the SELECT returns no rows (as well as other conditions that are outside the scope of this book).
PROGRAM_ERROR	Internal error. Usually means you need to contact Oracle support.
STORAGE_ERROR	Program ran out of system memory.
TIME_OUT_ON_RESOURCE	Program waited too long for some resource to be available.
TOO_MANY_ROWS	SELECT INTO in PL/SQL returns more than one row.
VALUE_ERROR	PL/SQL encountered invalid data conversions, truncations, or constraints on data.
ZERO_DIVIDE	Attempt at division by zero.
OTHERS	All other exceptions or internal errors not covered by the exceptions defined in the basic block. Used when you are not sure which named exception you are handling, but you do want to handle whichever exception was raised.

TABLE 8-2. *System-Defined Exceptions*

PL/SQL has two ways of showing information to a user about an error. One option is the use of the command SQLCODE, which returns the error code. An error code is a negative value that usually equals the value of the corresponding ORA error that would be issued if the exception remains unhandled when an application terminates. The other option returns a text message regarding the error. Not surprisingly, this command is SQLERRM. You can use both SQLCODE and SQLERRM in the exception handler. Note: not all system-defined exceptions are named.

Now try the previous example again, but this time use SQLCODE and SQLERRM. Enter the following code and compare your results with those shown in Figure 8-11.

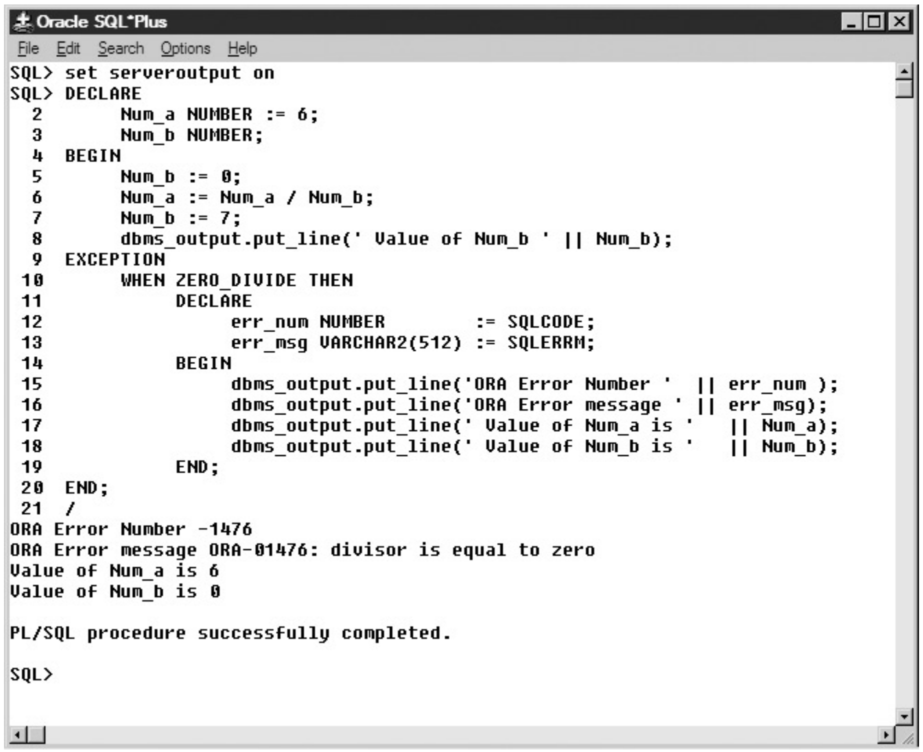
L 8-24
```
set serveroutput on
DECLARE
     Num_a NUMBER := 6;
     Num_b NUMBER;
BEGIN
     Num_b := 0;
     Num_a := Num_a / Num_b;
     Num_b := 7;
     dbms_output.put_line(' Value of Num_b ' || Num_b);
EXCEPTION
     WHEN ZERO_DIVIDE THEN
          DECLARE
               err_num NUMBER        := SQLCODE;
               err_msg VARCHAR2(512) := SQLERRM;
          BEGIN
               dbms_output.put_line('ORA Error Number '  || err_num );
               dbms_output.put_line('ORA Error message ' || err_msg);
               dbms_output.put_line(' Value of Num_a is '   || Num_a);
               dbms_output.put_line(' Value of Num_b is '   || Num_b);
          END;
END;
/
```

Programmer-Defined Exceptions

One handy feature of PL/SQL is that it allows you to create your own exception conditions and names. When raising and handling your own exceptions, they must be named and declared just like any other PL/SQL entity.

Here is a complete example of how to name and define your own exception. Enter the following code and compare your results with those shown in Figure 8-12:

L 8-25
```
set serveroutput on
DECLARE
     quantity1 NUMBER := -2;
     quantity2 NUMBER := 3;
     total NUMBER := 0;
     quantity_must_positive EXCEPTION;
     FUNCTION find_cost (quant NUMBER) RETURN NUMBER IS
```



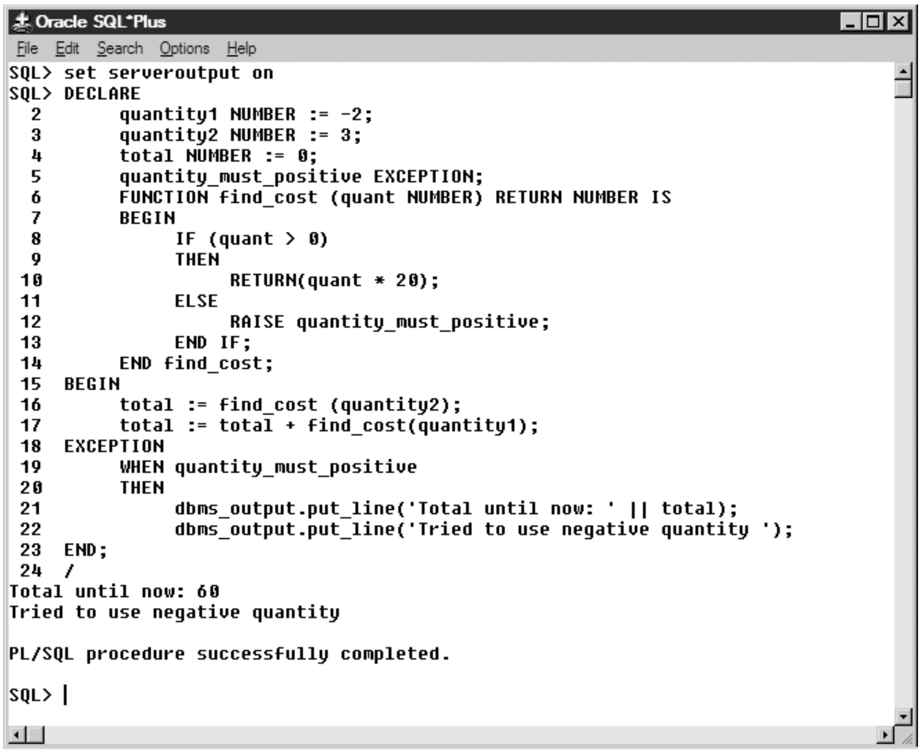
```
SQL> set serveroutput on
SQL> DECLARE
       Num_a NUMBER := 6;
       Num_b NUMBER;
  BEGIN
       Num_b := 0;
       Num_a := Num_a / Num_b;
       Num_b := 7;
       dbms_output.put_line(' Value of Num_b ' || Num_b);
  EXCEPTION
       WHEN ZERO_DIVIDE THEN
            DECLARE
                 err_num NUMBER        := SQLCODE;
                 err_msg VARCHAR2(512) := SQLERRM;
            BEGIN
                 dbms_output.put_line('ORA Error Number '  || err_num );
                 dbms_output.put_line('ORA Error message ' || err_msg);
                 dbms_output.put_line(' Value of Num_a is '   || Num_a);
                 dbms_output.put_line(' Value of Num_b is '   || Num_b);
            END;
  END;
  /
ORA Error Number -1476
ORA Error message ORA-01476: divisor is equal to zero
Value of Num_a is 6
Value of Num_b is 0

PL/SQL procedure successfully completed.

SQL>
```

FIGURE 8-11. *Using SQLCODE and SQLERRM for system-defined exceptions*

```
     BEGIN
          IF (quant > 0)
          THEN
               RETURN(quant * 20);
          ELSE
               RAISE quantity_must_positive;
          END IF;
     END find_cost;
BEGIN
     total := find_cost (quantity2);
     total := total + find_cost(quantity1);
EXCEPTION
     WHEN quantity_must_positive
     THEN
          dbms_output.put_line('Total until now: ' || total);
          dbms_output.put_line('Tried to use negative quantity ');
END;
/
```



```
SQL> set serveroutput on
SQL> DECLARE
       quantity1 NUMBER := -2;
       quantity2 NUMBER := 3;
       total NUMBER := 0;
       quantity_must_positive EXCEPTION;
       FUNCTION find_cost (quant NUMBER) RETURN NUMBER IS
       BEGIN
            IF (quant > 0)
            THEN
                 RETURN(quant * 20);
            ELSE
                 RAISE quantity_must_positive;
            END IF;
       END find_cost;
  BEGIN
       total := find_cost (quantity2);
       total := total + find_cost(quantity1);
  EXCEPTION
       WHEN quantity_must_positive
       THEN
            dbms_output.put_line('Total until now: ' || total);
            dbms_output.put_line('Tried to use negative quantity ');
  END;
  /
Total until now: 60
Tried to use negative quantity

PL/SQL procedure successfully completed.

SQL>
```

FIGURE 8-12. *Programmer-defined exception*

The exception is declared in the declaration section. Just like any other PL/SQL variable declared there, the life of the exception is valid only for this block. Since *find_cost* is also in this block, or is enclosed by this block, it can use the exception name. If the same function was defined as, say, a stored function, then you could not use the same exception name.

You can use your own exceptions for application-specific exception conditions that otherwise cannot be detected by the system or have no meaning for the system. For example, the system does not know that quantities ordered must be positive integer values. Your application should know this, however, and you can enforce it by catching values that are not positive integers as an exception while doing computations based on quantities. This is a very simple example, but you can imagine and will certainly come across more complex cases in real-life applications.

Summary

This chapter served as an introduction to the wonderful world of PL/SQL—a powerful programming language that works hand in hand with SQL. We explored PL/SQL variables. PL/SQL variables are used to hold the results of computations and to carry those results from one computation task to another.

We discussed the aspects of the PL/SQL basic block. All PL/SQL program units are made up of one or more basic blocks. A basic block is made up of header, declaration, execution, and exception sections. The header section contains identifying information for the block. For anonymous blocks, the header section is empty. The declaration section contains declarations for variables, constants, exceptions, cursors, functions, and procedures to be used within the block's execution and exception sections; if none of these are used, the declaration section will be empty. The execution section contains PL/SQL executable statements. The execution section is not optional and must be present to form a block. The exception section is used to handle error, or exception, conditions occurring within the execution section. This includes exceptions that may not be handled within any enclosed or nested blocks and within functions or procedures called.

We discussed how to create and call functions and procedures. You learned the meaning and use of formal and actual parameters.

Recall that PL/SQL program flow control constructs allow you to conditionally execute a piece of code once or repeatedly. The IF statement allows conditional execution once. The LOOP, WHILE loop, and FOR loop allow for repeated execution of the same set of statements. Cursors are the means for PL/SQL to communicate with SQL and hence the database. The cursor FOR loop allows you to process rows of tables one at a time.

Finally, you learned how to define your own exceptions and to raise or handle them by issuing friendly error messages. Another option is to remove what's causing the error and try again to create a successful program.

We have covered a lot of ground in this chapter and it has been the foundation for Chapter 9. You are probably eager to play with PL/SQL's powerful features. Have some fun, and then jump right into the next chapter.

Chapter Questions

1. Which of the following are true about PL/SQL procedures and functions?

   A. There is no difference between the two.

   B. A function has a return type in its specification and must return a value specified in that type. A procedure does not have a return type in its specification and should not return any value, but it can have a return statement that simply stops its execution and returns to the caller.

C. Both may have formal parameters of OUT or IN OUT modes, but a function should not have OUT or IN OUT mode parameters.

D. Both can be used in a WHERE clause of a SQL SELECT statement.

2. Which is the correct output for a run of the following code sample?

```
<<outer_block>>
DECLARE
    scope_num NUMBER := 3;
BEGIN
    DECLARE
        scope_num NUMBER := 6;
        Num_a     NUMBER := outer_block.scope_num;
    BEGIN
        dbms_output.put_line(scope_num);
        dbms_output.put_line(Num_a);
    END;
    dbms_output.put_line(scope_num);
END;
```

A. 6 3 3

B. Gives error saying duplicate declaration and aborts execution

C. 3 3 3

D. 6 3 6

3. Which of the following is true about IF statements?

A. At most, one set of executable statements gets executed corresponding to the condition that is TRUE. All other statements are not executed.

B. It depends. Sometimes more than one set of statements gets executed as multiple conditions may be TRUE and then statements for each of them should get executed.

4. Which of the following LOOPs will be entered at least once?

A. Simple LOOP

B. WHILE loop

C. FOR loop

D. Cursor FOR loop

5. Which one of the following is not true about exceptions?
   - **A.** Exceptions raised within the declaration section are to be handled in the enclosing block, if you want to handle them.
   - **B.** Statements in the execution section just after the statement responsible for raising exceptions are executed once the exception handler has finished execution.
   - **C.** When the system raises exceptions and they are not handled by the programmer, all the committed changes made to database objects—like tables by the execution section within which the exception occurred—are not automatically rolled back by the system.
   - **D.** Exceptions raised within a called procedure not handled by the procedure will roll back the changes done to IN OUT or OUT parameters by the procedure before the exception occurred.

Answers to Chapter Questions

1. **B, C.** A function has a return type in its specification and must return a value specified in that type. A procedure does not have a return type in its specification and should not return any value, but it can have a return statement that simply stops its execution and returns to the caller.
   Both may have formal parameters of OUT or IN OUT modes, but a function should not have OUT or IN OUT mode parameters.

Explanation A is certainly not true. D is false because procedures do not return values whereas functions do by utilizing the WHERE clause. Functions compute and return a single value, but do not modify its inputs so C is true. B is true as a matter of PL/SQL legal syntax.

2. **A.** 6 3 3

Explanation This is an example of scope. The outer block *scope_num* is overridden inside the inner block by its own *scope_num*. Thus the value for *scope_num* inside the inner block is 6. This inner *scope_num* is not visible to the outer block so once the inner block has run, the *scope_num* used is the outer *scope_num* resulting in the last value of 3. To get at the outer *scope_num,* we use the label name inside the inner block while assigning value to Num_a.

3. **A.** At most, one set of executable statements gets executed corresponding to the condition that is TRUE. All other statements are not executed.

Explanation The IF, ELSE, and ELSIF constrain the execution so that the conditions are mutually exclusive. Only one statement can be true. When no statements are true, no statements are executed.

4. **A.** Simple LOOP

Explanation The simple LOOP checks the condition for exiting the loop inside the loop body so it must get inside at least once. All other loops check the condition before entering the loop.

5. **B.** Statements in the execution section just after the statement responsible for raising exceptions are executed once the exception handler has finished execution.

Explanation If not handled, exceptions will abort the execution of the execution section and return control to the enclosing block. Therefore, the statements in the culprit execution section after the statement that raises exception will not be executed. All other choices are true.