Class Note: Chapter 5

Writing Database Access Program with PL/SQL

(Updated on June 23, 2017)

[The “class note” is the typical material I would prepare for my face-to-face class. Since this is an Internet based class, I am sharing the notes with everyone assuming you are in the class.]

SQL is nothing more than a data access language that allows applications (computer programs) to put data into and get data out of an Oracle database. In other words, SQL by itself is not a full-featured programming language that you can use to develop powerful database applications. To build a database application, you must use a procedural language that encompasses SQL to interact with an Oracle database. This chapter explains Oracle's very own procedural language, PL/SQL, which you can use to program an Oracle database server and associated applications. The following topics will be covered:

- PL/SQL coding basics
- Anonymous PL/SQL blocks
- Stored procedures, functions, and packages
- Database triggers

NOTE
By no means is this chapter a complete guide to PL/SQL. However, this chapter does provide an intermediate- level tutorial of PL/SQL’s capabilities, so that you can get started programming an Oracle database server.

Chapter Prerequisites

To practice the hands-on exercises in this chapter, you need to start SQL*Plus and run the following command script

location*\Sql\chap05.sql*

where location is the file directory where you expanded the support file downloaded from the course site. For example, after starting SQL*Plus and connecting as SCOTT, you can run this chapter's SQL command script using the SQL*Plus command @, as in the following example (assuming that your chap05.sql file is in C:\temp\Sql).

```
SQL > @C:\temp\Sql\chap05.sql;
```

Once the script completes successfully, leave the current SQL*Plus session open and use it to perform this chapter's exercises in the order that they appear. Refer to chapter 4 for more details for downloading the script files.

5.1 What Is PL/SQL?

PL/SQL is a procedural programming language that's built into most Oracle products. With PL/SQL, you can build programs to process information by combining PL/SQL procedural statements that control program flow with SQL statements that access an Oracle database. For example, the following is a very simple PL/SQL program that updates a part's UNITPRICE, given the part's ID number.

```
CREATE OR REPLACE PROCEDURE updatePartPrice (
 partId IN INTEGER,
 newPrice IN NUMBER )
IS
 invalidPart EXCEPTION;
BEGIN
--HERE'S AN UPDATE STATEMENT TO UPDATE A DATABASE RECORD
 UPDATE parts
  SET unitprice = newPrice
  WHERE id = partId;
--HERE'S AN ERROR-CHECKING STATEMENT
 IF SQL%NOTFOUND THEN
  RAISE invalidPart;
 END IF;
EXCEPTION
--HERE'S AM ERROR-HANDLING ROUTINE
 WHEN invalidPart THEN
 raise_application_error(-20000,'Invalid Part ID');
END updatePartPrice;
/
```

This example program is a procedure that is stored as a program unit in a database. Using PL/SQL, you can build many types of database access program units, including anonymous PL/SQL blocks, procedures, functions, and packages. All of the sections in this chapter include examples of PL/SQL programs. But before learning about full-blown PL/SQL programs, you need to understand the basic programmatic constructs and commands that the PL/SQL language offers. If you run this script at this point, procedure will be created but compiler may complain about the warnings.

NOTE
PL/SQL is a procedural language that's very similar to Ada. PL/SQL has statements that allow you to declare variables and constants, control program flow, assign and manipulate data, and more.

5.2 PL/SQL Blocks

A PL/SQL program is structured using distinct blocks that group related declarations and statements. Each block in a PL/SQL program has a specific task and solves a particular problem. Consequently, you can organize a PL/SQL program so that it is easy to understand.

A PL/SQL block can include three sections as the following pseudo-code illustrates: program declarations, the main program body, and exception handlers.

```
DECLARE
 -- program declarations are optional
 BEGIN
   -- program body is required
 EXCEPTION
 -- exception handlers are optional
END;
```

Most of the examples in this class/chapter ask you to use SQL*Plus and interactively type and execute anonymous PL/SQL blocks while learning the fundamentals of PL/SQL. An anonymous PL/SQL block has no name and is not stored permanently as a file or in an Oracle database. An application, such as SQL*Plus, simply sends the PL/SQL block to the database server for processing at run time. Once Oracle executes an anonymous PL/SQL block, the block ceases to exist.

5.2.1 Program Declarations

The declaration section of a PL/SQL block is where the block declares all variables, constants, exceptions, and so on, that are then accessible to all other parts of the same block. The declarative section of a PL/SQL block starts with the DECLARE keyword and implicitly ends with the BEGIN keyword of the program body. If the program does not need to make any declarations, the declaration section is not necessary.

5.2.2 The Program Body

The main program body of a PL/SQL block contains the executable statements for the block. In other words, the body is where the PL/SQL block defines its functionality. The body of a PL/SQL block begins with the BEGIN keyword and ends with the EXCEPTION keyword that starts the exception handling section of the block; if the block does not include any exception handlers, the program body ends with the END keyword that ends the block altogether.

5.2.3 Exception Handlers

The optional exception handling section of a PL/SQL block contains the exception handlers (error handling routines) for the block. When a statement in the block's body raises an exception (detects an error), it transfers program control to a corresponding exception handler in the exception section for further processing. The exception handling section of a PL/SQL block begins with the EXCEPTION keyword and ends with the END keyword. If a program does not need to define any exception handlers, the exception handling section of the block is not necessary.

5.2.4 Program Comments

All blocks of a PL/SQL program should include comments that document program declarations and functionality. Comments clarify the purpose of specific programs and code segments.

PL/SQL supports two different styles for comments, as the following code segment shows.

```
-- PRECEDE A SINGLE-LINE COMMENT WITH A DOUBLE-HYPHEN.
/* DELIMIT A MULTI-LINE COMMENT WITH /*" AS A PREFIX AND " * /" AS
A SUFFIX. A MULTI-LINE COMMENT CAN CONTAIN ANY NUMBER OF LINES. */
```

The examples in the remainder of this chapter often use comments to help explain the functionality of code listings.

5.3 The Fundamentals of PL/SQL Coding

All procedural languages, such as PL/SQL, have fundamental language elements and functionality that you need to learn about before you can build programs using the language. The following sections introduce the basic elements of PL/SQL, including the following:

- How to declare program variables and assign them values?
- How to control program flow with loops and conditional logic?
- How to embed SQL statements and interact with Oracle databases?
- How to declare and use subprograms (procedures and functions) within PL/SQL blocks?
- How to declare user-defined types, such as records and nested tables?
- How to declare and use cursors to process queries that return multiple rows?
- How to use exception handlers to handle error conditions?

5.3.1 Working with Program Variables

All procedural programs typically declare one or more program variables and use them to hold temporary information for program processing. The next two exercises teach you how to declare program variables (and constants), initialize them, and assign them values in the body of a PL/SQL program.

EXERCISE 5.1: Declaring Variables and Constants with Basic Data types

The declaration section of a PL/SQL program can include variable and constant declarations. The general syntax that you use to declare a scalar variable or constant is as follows:

```
Variable [CONSTANT] datatype [[NOT NULL] {DEFAULT/:=} expression];
```

NOTE
To declare a constant rather than a variable, include the CONSTANT keyword in the declaration. You must initialize a constant, after which the constant's value cannot change.

When you declare a variable, you can choose to initialize it immediately or wait until later in the body of the program; however, if you include the NOT NULL constraint, you must initialize the variable as part of its declaration. By default, PL/SQL initializes a variable as null unless the program explicitly initializes the variable.

A program can declare a variable or constant using any Oracle or ANSI/ISO datatype or subtype listed in Table 5-1.

NOTE
A subtype is a constrained version of its base type

Data type	**Subtype**	**Description**
BINARY_INTEGER	NATURAL, NATURALN, POSITOIVE, POSITIVEN, SIGNTYPE	Stores signed integers in the range -2,147,483,647 and 2,147,483,647, Store only nonnegative integers; the latter disallows nulls. POSITIVE and POSITIVEN store only positive integers; the latter disallows nulls. SIGNTYPE stores only -1, 0, and 1
CHAR (*size*)	CHARACTER (*size*)	Fixed-length character data of length *size* bytes. Fixed for every row in the table (with trailing blanks); maximum size is 2000 bytes per row, default size is 1 byte per row. Consider the character set (one-byte or multibyte) before setting *size*.
VARCHAR2 (*size*)	VARCHAR (*size*), STRING	Variable-length character data. A maximum *size* must be specified. Variable for each row, up to 4000 bytes per row. Consider the character set (one-byte or multibyte) before setting *size*.
NCHAR(*size*)		Fixed-length character data of length *size* characters or bytes, depending on the national character set. Fixed for every row in the table (with trailing blanks). Column *size* is the number of characters for a fixed-width national character set or the number of bytes for a varying-width national character set. Maximum *size* is determined by the number of bytes required to store

		one character, with an upper limit of 2000 bytes per row. Default is 1 character or 1 byte, depending on the character set.
NVARCHAR2 (*size*)		Variable-length character data of length *size* characters or bytes, depending on national character set. A maximum *size* must be specified. Variable for each row. Column *size* is the number of characters for a fixed-width national character set or the number of bytes for a varying-width national character set. Maximum *size* is determined by the number of bytes required to store one character, with an upper limit of 4000 bytes per row. Default is 1 character or 1 byte, depending on the character set.
LONG		Variable-length character data. Variable for each row in the table, up to 2^31 - 1 bytes, or 2 gigabytes, per row.
NUMBER (*p, s*)	DEC, DECIMAL, DOUBLE PRECISION, FLOAT (*precision*), INTEGER, INT, NUMERIC, REAL, SMALLINT	Variable-length numeric data. Maximum precision *p* and/or scale *s* is 38. Variable for each row. The maximum space required for a given column is 21 bytes per row.
DATE		Fixed-length date and time data, ranging from January 1, 4712 B.C. to December 31, 4712 A.D. Fixed at 7 bytes for each row in the table. Default format is a string (such as DD-MON-YY) specified by NLS_DATE_FORMAT parameter.
RAW (*size*)		Variable-length raw binary data. A maximum *size* must be specified. Variable for each row in the table, up to 2000 bytes per row.
LONG RAW		Variable-length raw binary data. Variable for each row in the table, up to 2^31 - 1 bytes, or 2 gigabytes, per row.
BLOB		Binary data. Up to 2^32 - 1 bytes, or 4 gigabytes.
CLOB		Single-byte character data. Up to 2^32 - 1 bytes, or 4 gigabytes.
NCLOB		Single-byte or fixed-length multibyte national character set (NCHAR) data. Up to 2^32 - 1 bytes, or 4 gigabytes.
BFILE		Binary data stored in an external file. Up to 2^32 - 1 bytes, or 4 gigabytes.
ROWID		Binary data representing row addresses. Fixed at 10 bytes (extended ROWID) or 6 bytes (restricted ROWID) for each row in the table.

TABLE 5-1. PL/SQL Scalar Datatypes and Related Subtypes

For the first hands-on exercise in this chapter, use SQL*Plus to enter the following anonymous PL/SQL block that declares and initializes several variables of different datatypes and then outputs their current values to standard output.

```
DECLARE
    outputstring VARCHAR2(20) := 'Hello World';
    todaysDate DATE := SYSDATE;
    pi CONSTANT NUMBER := 3.14159265359;
```

```
BEGIN
    DBMS_OUTPUT.PUT_LINE(outputstring);
    DBMS_OUTPUT.PUT_LINE( todaysDate);
    DBMS_OUTPUT.PUT_LINE (pi);
END;
/
```

NOTE
*To end a PL/SQL program in SQL*Plus, you must terminate the code with a line that includes only the backslash (/) character*.

The output of the program should be similar to the following:

```
Hello World
16-AUG-99
3.14159265359

PL/SQL procedure successfully completed.
```

There are a couple of subtle points to understand about this first example PL/SQL program.

- When a program declares a variable using a datatype that requires a constraint specification (such as VARCHAR2), the declaration must specify the constraint.
- The body of the example program uses the DBMS_OUTPUT.PUT_LINE procedure to direct output to the standard output. The DBMS_OUTPUT. PUT_LINE procedure is analogous in functionality to the println procedure in C, and the System.outprintln method of Java. However, you will not see the output of a call to DBMS_OUTPUT.PUT_LINE when using SQL*Plus unless you enable the SQL*Plus environment setting SERVEROUTPUT. The prerequisite command script for this chapter executes a SET SERVEROUTPUT ON statement for you.

EXERCISE 5.2: Assigning Values to Variables

Value assignment is one of the most common operations within any type of procedural program. The general syntax that you use to assign a value to a variable is as follows:

```
variable:= expression
```

An assignment statement in a PL/SQL program assigns the value that results from an expression to a PL/SQL construct, such as a variable, using the assignment operator (: =). For example, enter the following PL/SQL block, which declares some variables, assigns them values in the program body, and then uses the DBMS_ OUTPUT.PUT_LINE procedure to output their current values to standard output.

```
DECLARE
  outputString VARCHAR2(20) ;
  todaysDate DATE;
  tomorrowsDate DATE;
  lastDayOfTheMonth DATE;
BEGIN
  outputString := ‘Hello’ ||‘World’;
  todaysDate := SYSDATE;
  tomorrowsDate := SYSDATE + 1;
  lastDayOfTheMonth := LAST_DAY(SYSDATE);
  DBMS_OUTPUT.PUT_LINE(outputString);
   DBMS_OUTPUT.PUT_LINE(todaysDate);
   DBMS_OUTPUT.PUT_LINE(tomorrowsDate);
  DBMS_OUTPuT.PUT_LINE(lastDayOfTheMonth);
END;
/
```

The program output should be similar to the following:

```
Hello World
16-MAY-16
17-MAY-16
31-MAY-16
PL/SQL procedure successfully completed.
```

Notice that the assignment statements in the body of the program use different types of expressions that build values to assign to variables. Several sections in the previous chapter teach you how to build expressions for a query's SELECT clause using literals, operators, and SQL functions. Similarly, you can build expressions for the right side of an assignment statement in a PL/SQL program. For example, the expressions in the example program use literals, the concatenation and addition operators (||and +), and the LAST_DAY and SYSDATE functions.

NOTE
Most SQL functions are also built-in and supported in PL/SQL statements.

5.3.2 Controlling Program Flow

Typical procedural programs have flow. That is, a program uses some sort of logic to control whether and when the program executes given statements. PL/SQL programs can control program flow using iterative logic (loops), conditional logic (if-then-else), and sequential logic (goto). The following exercises teach you how to use the different program flow control statements that PL/SQL offers.

EXERCISE 5.3: Using PL/SQL Loops

A PL/SQL program can use a loop to iterate the execution of a series of statements a certain number of times. Enter the following anonymous PL/SQL block, which teaches you how to use a basic loop. The beginning of a basic loop starts with a LOOP statement and ends with an END LOOP statement.

```
DECLARE
 loopCounter INTEGER := 0;
BEGIN
LOOP
  DBMS_OUTPUT.PUT(loopCounter ||‘  ‘);
  loopCounter := loopCounter + 1;
 EXIT WHEN loopCounter = 10;
END LOOP;
DBMS_OUTPUT.PUT_LINE(‘Loop Exited.’);
END;
/
```

The program output should look like the following:

```
 0 1 2 3 4 5 6 7 8 9 Loop Exited.
PL/SQL procedure successfully completed.
```

It is important to understand that every basic loop definition should use either an *EXIT WHEN* or *EXIT* statement to terminate the loop—otherwise the loop executes infinitely!

NOTE
The preceding example uses the DBMS_OUTPUT.PUT *procedure to place display output in a temporary buffer. The subsequent call to* DBMS_OUTPUT.PUT_LINE *prints the entire contents of the session's display buffer.*

Next, enter the following anonymous PL/SQL block, which teaches you how to use a different type of loop, a WHILE loop. The LOOP statement of a WHILE loop begins with a WHILE condition and ends with an END LOOP statement.

```
DECLARE
  loopCounter INTEGER := 0;
BEGIN
   WHILE loopCounter < 10 LOOP
  DBMS_OUTPUT.PUT(loopCounter || ‘ ’);
 loopCounter := loopCounter + 1;
  END LOOP;
DBMS_OUTPUT.PUT_LINE('Loop Exited.');
END;
/
```

The program output should look like the following:

```
0 1 2 3 4 5 6 7 8 9  Loop  Exited.
PL/SQL procedure successfully completed.
```

Notice that the definition of a WHILE loop requires that you specify a condition to describe how the loop terminates.

Finally, enter the following anonymous PL/SQL block, which teaches you how to use a third type of loop, a FOR loop. The LOOP statement of a FOR loop begins with a FOR clause, and ends with an END LOOP statement.

```
BEGIN
   «outer_loop»
   FOR outerLoopCounter IN 0 .. 25 LOOP
   DBMS_OUTPUT.PUT_LINE('Outer Loop: '||outerLoopCounter);
  DBMS_OUTPUT.PUT(' Inner Loop: ');
    «inner_loop»
    FOR innerLoopCounter IN REVERSE 1 .. 3 LOOP
    DBMS_OUTPUT.PUT(innerLoopCounter ||' ');
    EXIT inner_loop
    WHEN ((outerLOopCounter = 2) AND(innerLoopCounter =3));
    EXIT outer_loop
    WHEN ((outerLoopCounter = 5) AND(innerLoopCounter = 2));
    END LOOP inner_loop;
    DBMS_OUTPUT.PUT_LINE('Inner Loop Exited.');
  DBMS_OUTPUT.PUT_LINE('-----------------');
  END LOOP outer_loop;
DBMS_OUTPUT.PUT_LINE('Outer Loop Exited.');
END;
/
```

The program output should look like the following: *(Be careful if you copy and paste this code in SQL*Plus, sometimes << >> are printed as quotes and it might lead you to compiler giving errors on your code. Make sure that the code in SQL*Plus is exactly as shown here)*

```
Outer Loop: 0
Inner Loop: 321 inner Loop Exited.
--------------------------------------
Outer Loop: 1
Inner Loop: 321 Inner Loop Exited.
--------------------------------------
Outer Loop: 2
Inner Loop: 3 Inner Loop Exited.
--------------------------------------
Outer Loop: 3
Inner Loop: 321 inner Loop Exited.
--------------------------------------
Outer Loop: 4
Inner Loop: 321 inner Loop Exited.
--------------------------------------
Outer Loop: 5
Inner Loop: 3 2 Outer Loop Exited.

PL/SQL procedure successfully completed.
```

The preceding example includes several important points that you should understand about FOR loops and loops in general:

- You can nest loops within loops.
- You can use loop labels to name loops, and then reference specific loops by name in EXIT and END LOOP statements.
- A FOR loop can declare its integer counter variable as part of the FOR ... LOOP statement.
- A FOR loop automatically increments or decrements its counter variable so that you do not have to explicitly do so in the body of the loop.

EXERCISE 5.4: Using the PL/SQL Command IF ... ELSIF ... ELSE

An IF statement in a PL/SQL program evaluates a Boolean condition, and if the condition is TRUE executes one or more statements. You can also use the optional ELSIF and ELSE clauses to enter subsequent conditions to be evaluated if the first condition is not TRUE. Enter the following anonymous PL/SQL block, which teaches you how to use the PL/SQL command IF ... ELSIF ... ELSE for conditions.

```
BEGIN
FOR i IN 1 .. 20  LOOP
IF ((i mod 3 = 0) AND (i mod 5 = 0))THEN
    DBMS_OUTPUT.PUT('multipleOfBoth');
    ELSIF i mod 3 = 0 THEN
    DBMS_OUTPUT.PUT( 'multipleOf3');
         ELSIF  i  mod  5 = 0  THEN
DBMS_OUTPUT.PUT('multipleOf5');
    ELSE
DBMS_OUTPUT.PUT(i);
END IF;
DBMS_OUTPUT.PUT(' ');
END LOOP;
DBMS_OUTPUT.PUT_LINE(' ');
END;
/
```

The program output should look like the following:

```
1 2 multipleOf3 4 multipleOf5 multipleOf3 7 8
multipleOf3
multipleOf5 11 multipleOf3 13 14 multipleOfBoth 16 17
multipleOf3 19 multipleOf5

PL/SQL procedure successfully completed.
```

EXERCISE 5.5: Using the PL/SQL Command GOTO

Unlike conditional and iterative flow control, sequential control or branching is rarely necessary in PL/SQL programs; however, PL/SQL provides the GOTO command should the need arise. For example, enter the following anonymous PL/SQL block, which teaches you how to use a GOTO statement to branch to a program label that precedes an executable statement in a PL/SQL program.

```
BEGIN
  --set the loop to iterate 10 times
  FOR i IN 1 .. 10 LOOP
  DBMS_OUTPUT.PUT ( i ||‘ ’);
   IF i = 5 THEN
    GOTO message1;
   END IF;
  END LOOP;
DBMS_OUTPUT.PUT_LINE (‘All loop iterations printed.');
  «message1»
   DBMS_OUTPUT.PUT_LINE (‘Only 5 loop iterations printed.’);
END;
/
```

The program output is as follows:

```
1 2 3 4 5 Only 5 loop iterations printed.
PL/SQL procedure successfully completed.
```

5.3.3 Interacting with Databases

The previous example exercises are PL/SQL programs that generate simple output to demonstrate some basics of PL/SQL. However, the primary reason for using PL/SQL is to create database access programs. A PL/SQL program can interact with an Oracle database only through the use of SQL. The following exercises show you how a PL/SQL program can manipulate database information using standard SQL DML (data modification language) statements and cursors.

EXERCISE 5.6: Manipulating Table Data with DML Statements

PL/SQL programs can include any valid INSERT, UPDATE, or DELETE statement to modify the rows in a database table. For example, enter the following anonymous PL/SQL block, which inserts a new record into the PARTS table.

```
DECLARE
  newId INTEGER := 6;
  newDesc VARCHAR2(250) := ‘Mouse’;
    BEGIN
    INSERT INTO parts
    VALUES(newId, newDesc, 49, 1200, 500);
END;
/
```

The output produced by this program is simply the following line:

```
PL/SQL procedure successfully completed.
```

A variable or constant in a PL/SQL block can satisfy the requirement for an expression in a DML statement. For example, the previous example program uses program variables to supply the first two values in the VALUES clause of the INSERT statement.

Now query the PARTS table to see the new record.

```
SELECT * FROM parts
WHERE id = 6;
```

The result set is as follows:

```
 ID DESCRIPTION UNITPRICE ONHAND REORDER
--- ----------- --------- ------ -------
  6     Mouse       49      1200     500
```

All data modifications made by INSERT, UPDATE, and DELETE statements inside of a PL/SQL block are part of your session's current transaction. Although you can include COMMIT and ROLLBACK statements inside many types of PL/SQL blocks, transaction control is typically controlled outside of PL/SQL blocks, so that transaction boundaries are clearly visible to those using your PL/SQL programs.

EXERCISE 5.7: Assigning a Value to a Variable with a Query

PL/SQL programs often use the INTO clause of the SQL command SELECT to assign a specific database value to a program variable. Oracle supports the use of a SELECT... INTO statement only inside PL/SQL programs. Try out this type of assignment statement by entering the following anonymous PL/SQL block, which uses a SELECT ... INTO statement to assign a value to a program variable.

```
DECLARE
 partDesc VARCHAR2(250);
 BEGIN
   SELECT description INTO partDesc
   FROM parts
   WHERE  id  =  3;
DBMS_OUTPUT.PUT_LINE(‘Part 3 is a ' ||partDesc);
END;
/
```

The program output is as follows:

```
 Part3 is a Laptop PC

PL/SL procedure successfully completed.
```

A SELECT ... INTO command must have a result set with only one row—if the result set of a SELECT ... INTO statement contains more than one row, Oracle raises the TOO_MANY_ROWS exception. To process a query that returns more than one row, a PL/SQL program must use a cursor. You'll learn more about cursors later in this chapter.

5.3.4 Declaring and Using Subprograms: Procedures and Functions

The declarative section of a PL/SQL block can declare a common subtask as a named subprogram (or subroutine). Subsequent statements in the main program body can then call (execute) the subprogram to perform work whenever necessary.

PL/SQL supports two types of subprograms: procedures and functions. A procedure is a subprogram that performs an operation. A function is a subprogram that computes a value and returns it to the program that called the function.

EXERCISE 5.8: Declaring and Using a Procedure

The general syntax for declaring a procedure is as follows:

```
PROCEDURE procedure
  [(parameter[IN|OUT|IN OUT]datatype[{DEFAULTl:=expression]
  [,...]  ) ]
 declarations  ..
 (IS|AS)
  BEGIN
  statements...
END [procedure];
```

When you declare a subprogram, such as a procedure, you can pass values into and out of the subprogram using parameters. Typically, a calling program passes one or more variables as parameters to a subprogram.

For each parameter, you must specify a datatype in an unconstrained form. Furthermore, you should indicate the mode of each parameter as IN, OUT, or IN OUT:

- An IN parameter passes a value into a subprogram, but a subprogram cannot change the value of the external variable that corresponds to an IN parameter.
- An OUT parameter cannot pass a value into a subprogram, but a subprogram can manipulate an OUT parameter to change the value of the corresponding variable in the outside calling environment.
- An IN OUT parameter combines the capabilities of IN and OUT parameters.

Enter the following anonymous PL/SQL block, which declares a procedure to print horizontal lines of a specified width.

```
DECLARE
 PROCEDURE printLine(width IN INTEGER, chr IN CHAR DEFAULT '-') IS
  BEGIN
   FOR i IN 1 .. width LOOP
   DBMS_OUTPUT.PUT(chr);
  END LOOP;
 DBMS_OUTPUT.PUT_LINE('');
END printLine;
 BEGIN
 printLine(40, '*');                 -- print a line of 40 *s
 printLine(width => 20, chr => '=');  -- print a line of 20 =s
 printLine(10);                       -- print a line of 10 -s
END;
/
```

The program output is as follows:

```
****************************************
====================
----------

PL/SQL procedure successfully completed.
```

The body of the example in this exercise calls the printLine procedure three times.

- The first procedure call provides values for both parameters of the procedure using positional notation—each parameter value that you specify in a procedure call corresponds to the procedure parameter declared in the same position.
- The second procedure call provides values for both parameters of the procedure using named notation—the name of a parameter and the association operator (=>) precedes a parameter value.
- The third procedure call demonstrates that you must provide values for all procedure parameters without default values, but can optionally omit values for parameters with a default value.

EXERCISE 5.9: Declaring and Using a Function

The general syntax for declaring a function is as follows:

```
FUNCTION function
  [(parameter[IN|OUT|IN OUT]datatype[(DEFAULTl:=}
expression]
     [,...] )]
     RETURN datatype
     declarations ...
{IS|AS} BEGIN
statements   .. .
END[function];
```

Notice that a function differs from a procedure in that it returns a value to its calling environment. The specification of a function declares the type of the return value.

Furthermore, the body of a function must include one or more RETURN statements to return a value to the calling environment.

Enter the following anonymous PL/SQL block, which declares and uses a function.

```
DECLARE
 tempTotal NUMBER;
 FUNCTION orderTotal(orderId IN INTEGER)
 RETURN NUMBER
  IS
   orderTotal NUMBER;
    tempTotal NUMBER;
      BEGIN
      SELECT SUM(i.quantity * p.unitprice) INTO orderTotal
      FROM items i, parts p
      WHERE i.o_id = orderid
      AND i.p_id = p.id
     GROUP BY i.o_id;
    RETURN orderTotal;
  END orderTotal;
BEGIN
 DBMS_OUTPUT.PUT_LINE('Order 1 Total:'|| orderTotal(1));
 tempTotal := orderTotal(2) ;
 DBMS_OUTPUT.PUT_LINE('Order2Total:'|| tempTotal);
END;
/
```

The program output is as follows:

```
Order 1 Total: 7094
Order 2 Total: 3196

PL/SQL procedure successfully completed.
```

The main program body of the PL/SQL block calls the orderTotal function twice. A program can call a function anywhere an expression is valid—for example, as a parameter for a procedure call or on the right side of an assignment statement. A SQL statement can also reference a user-defined function in the condition of a WHERE clause.

5.3.5 Working with Record Types

So far in this chapter, you've seen how to declare simple, scalar variables and constants based on Oracle datatypes (for example, NUMBER) and subtypes of the base datatypes (for example, INTEGER]. A block in a PL/SQL program can also declare user-defined types and then use the user-defined types to declare corresponding program variables. This section teaches you how to declare and use an elementary user-defined type, a record type. A record type consists of a group of one or more related fields, each of which has its own name and datatype. Typically, PL/SQL programs use a record type to create variables that match the structure of a record in a table. For example, you might declare a record type called partRecord and then use the type to create a record variable that holds a part's ID, DESCRIPTION, UNITPRICE, ONHAND, and REORDER fields.

After you declare a record variable, you can manipulate the individual fields of the record or pass the entire record to subprograms as a unit.

The general syntax for declaring a record type is as follows:

```
TYPE recordType IS RECORD
[field datatype [NOT NULL]{DEFAULT|:=}expression]
[,field.. .]
 )
```

The specification of an individual field in a record type is similar to declaring a scalar variable—you must specify the field's datatype, and you can specify an optional not null constraint, as well as initialize the field.

EXERCISE 5.10: Declaring and Using Record Types

Enter the following anonymous PL/SQL block, which demonstrates how to declare and use a record type. The declaration section of the block declares a user-defined record type to match the attributes of the PARTS table, and then declares two record variables using the new type. The body of the program demonstrates how to do the following:

- Reference individual fields of record variables using dot notation
- Assign values to the fields of a record variable
- Pass a record variable as a parameter to a procedure call
- Copy the field values of one record variable to the fields of another variable of the same record type
- Use the fields of a record variable as expressions in an INSERT statement

```
DECLARE
 TYPE partRecord IS RECORD (
 id INTEGER,
 description VARCHAR2(250),
 unitprice NUMBER(10,2),
 onhand INTEGER,
 reorder INTEGER
 ) ;
 SelectedPart partRecord;
 CopiedPart partRecord;
 PROCEDURE printPart (title IN VARCHAR2, thisPart IN partRecord)
 IS
   BEGIN
   FOR i IN 1 .. 50 LOOP
   DBMS_OUTPUT.PUT('-');
   END LOOP;
 DBMS_OUTPUT.PUT_LINE('');
 DBMS_OUTPUT.PUT_LINE(title||‘: ID: ’||thisPart.id ||
 ‘DESCRIPTION: ‘ ||thisPart.description);
 END printPart;
   BEGIN
   /* Assign values to the fields of a record variable
   || using a SELECT .. INTO statement.
   */
 SELECT id, description,unitprice,onhand, reorder INTO
  selectedPart
  FROM parts WHERE id = 3;
  printPart(‘ selectedPart Info',selectedPart);

   /* Assign the field values of one record variable to
   || the corresponding fields in another record
   || variable of the same type. Then assign new
   || values to the fields of original record variable
   || I to demonstrate that record copies are not by reference.
   */
  copiedPart := selectedPart;

  selectedPart.id := 7;
  selectedPart.description := ‘Laser Printer’;
  selectedPart.unitprice := 399;
  selectedPart.onhand := 730;
  selectedPart.reorder := 500;

  printPart(‘newPart Info’, selectedPart);
  printPart(‘copiedPart Info’, copiedPart);

   /* Use the fields of a record variable as expressions
   ||in the VALUES clause of an INSERT statement.
   */
 INSERT INTO parts
 VALUES (selectedPart.id,selectedPart.description,
  selectedPart.unitprice,selectedPart.onhand,
 selectedPart.reorder);
 END;
 /
```

The program output is as follows:

```
------------------------------------------------
selectedPart Info: ID: 3 DESCRIPTION: Laptop PC
------------------------------------------------
newPart Info: ID: 7 DESCRIPTION: Laser Printer
------------------------------------------------
CopiedPart Info: ID: 3 DESCRIPTION: Laptop PC

PL/SQL procedure successfully completed.
```

NOTE
As an additional exercise, build a query of the PARTS table to view the new pan inserted by the example program.

5.4 Using the %TYPE and %ROWTYPE Attributes

A PL/SQL program can use the %TYPE and %ROWTYPE attributes to declare variables, constants, individual fields in records, and record variables that match the properties of database columns and tables or other program constructs. Not only do attributes simplify the declaration of program constructs, but their use also makes programs flexible for database modifications. For example, after an administrator modifies the PARTS table to add a new column, a record variable declared using the %ROWTYPE attribute automatically adjusts to account for the new column at run time, without any modification of the program.

EXERCISE 5.11: Using the %TYPE Attribute

The declaration of a PL/SQL variable, constant, or field in a record variable can use the %TYPE attribute to capture the datatype of another program construct or column in a database table at run time. For example, enter the following anonymous PL/SQL block, which uses the %TYPE attribute to reference the columns in the PARTS table when declaring the partRecord type.

```
DECLARE
 TYPE partRecord IS RECORD(
 id parts.id%TYPE,
 description parts.description%TYPE,
 unitprice parts.unitprice%TYPE,
 onhand parts.onhand%TYPE,
 reorder parts.reorder%TYPE
 ) ;
 selectedPart partRecord;
   BEGIN
    SELECT id, description,unitprice,onhand,reorder INTO selectedPart
    FROM parts WHERE id = 3;
    DBMS_OUTPUT.PUT_LINE('ID:' || selectedPart.id);
    DBMS_OUTPUT.PUT_LINE('DESCRIPTION:'||  SelectedPart.description);
    DBMS_OUTPUT.PUT_LINE('UNIT PRICE;'||selectedPart.unitprice);
    DBMS_OUTPUT.PUT_LINE('CURRENTLY ONHAND:'|| selectedpart.onhand);
    DBMS_OUTPUT.PUT_LINE('REORDER AT:'||selectedPart.reorder);
   END;
/
```

The program output is as follows:

```
ID: 3
DESCRIPTION: Laptop PC
UNIT PRICE: 2100
CURRENTLY ONHAND: 7631
REORDER AT: 1000

PL/SQL procedure successfully completed.
```

EXERCISE 5.12: Using the %ROWTYPE Attribute

A PL/SQL program can use the %ROWTYPE attribute to easily declare record variables and other constructs at run time. For example, enter the following anonymous

PL/SQL block, which shows how to use the %ROWTYPE attribute to simplify the declaration of a record variable that corresponds to the fields in the PARTS table.

```
DECLARE
 selectedPart parts%ROWTYPE;
  BEGIN
  SELECT id,description,unitprice,onhand,reorder INTO selectedPart
  FROM parts WHERE id = 3;
  DBMS_OUTPUT.PUT_LINE('ID:'|| selectedPart.id);
  DBMS_OUTPUT.PUT_LINE('DESCRIPTION:'||selectedPart.description);
  DBMS_OUTPUT.PUT_LINE('UNIT PRICE:'||SelectedPart.unitprice);
  DBMS_OUTPUT.PUT_LINE('CURRENTLY ONHAND:'||selectedPart.onhand);
  DBMS_OUTPUT.PUT_LINE('REORDER AT:'||SelectedPart.reorder);
 END;
/
```

The program output is as follows:

```
ID : 3
DESCRIPTION: Laptop PC
UNIT PRICE: 2100
CURRENTLY ONHAND: 7631
REORDER AT: 1000

PL/SQL procedure successfully completed.
```

Notice that a record variable that you declare using the %ROWTYPE attribute automatically has field names that correspond to the fields in the referenced table.

5.4.1 Working with Cursors

Whenever an application submits a SQL statement to Oracle, the server opens at least one cursor to process the statement. A cursor is essentially a work area for a SQL statement. When a PL/SQL program (or any other application) submits an INSERT, UPDATE, or DELETE statement, Oracle automatically opens a cursor to process the statement. Oracle also can automatically process SELECT statements that return just one row. However, database access programs frequently must process a query that returns a set of database records, rather than just one row. To process the rows of a query that correspond to a multirow result set, a PL/SQL program must explicitly declare a cursor with a name, and then reference the cursor by its name to process rows one at a time. The steps for using cursors inside PL/SQL programs include the following:

- Declare the cursor. You use a query to define the columns and rows of a cursor.
- Open the cursor. You use the PL/SQL command OPEN to open a declared cursor.
- Fetch rows from the cursor. You use the PL/SQL command FETCH to fetch rows, one-by-one, from an open cursor.

- Close the cursor. You use the PL/SQL command CLOSE to close an open cursor.

EXERCISE 5.13: Declaring and Using a Simple Cursor

To familiarize yourself with the steps necessary to declare and use a cursor, enter the following anonymous PL/SQL block, which declares and uses a very simple cursor to print out selected columns in all the rows of the PARTS table.

```
DECLARE
 -- Step 1: Declare the cursor.
  CURSOR nextPartsRow IS
  SELECT * FROM parts
  ORDER BY id;
   currentPart parts%ROWTYPE;  --record variable that matches the cursor
   BEGIN
   -- Step 2. Open the cursor.
   OPEN nextPartsRow;
   FETCH nextPartsRow INTO currentPart;
   -- Step 3. Using a WHILE loop, fetch individual rows from the cursor.
   WHILE nextPartsRow%FOUND LOOP
   DBMS_OUTPUT.PUT_LINE(currentPart.id ||''||CurrentPart.description);
   FETCH nextPartsRow INTO CurrentPart;
   END LOOP;
  -- Step 4. Close the cursor.
 CLOSE nextPartsRow;
END;
/
```

The program output is as follows:

```
1 Fax Machine
2 Copy Machine
3 Laptop PC
4 Desktop PC
5 Scanner
6 Mouse
7 Laser Printer

PL/SQL procedure successfully completed.
```

Notice that after fetching the first row from the cursor, the program uses a WHILE loop to fetch subsequent rows from the open cursor, one-by-one. A PL/SQL program can use a cursor attribute to make decisions when processing cursors. The condition of the WHILE loop in the example program uses the %FOUND cursor attribute to detect when the last row of the cursor has been fetched. Table 5-2 lists the cursor attributes available with PL/SQL.

EXERCISE 5.14: Using Cursor FOR Loops

Because cursors are designed to process queries with multiple-row result sets, programs almost always process cursors using loops. To simplify the steps necessary to set up and process a cursor, a PL/SQL program can use a cursor FOR loop. A cursor FOR loop automatically declares a variable or record capable of receiving the rows in the cursor, opens the cursor, fetches rows from the cursor, and closes the cursor when the last row is fetched from the cursor. Enter the following anonymous PL/SQL block,

Explicit Cursor Attribute	**Description**
cursor%FOUND	The %FOUND attribute evaluates to TRUE when the preceding FETCH statement corresponds to at least one row in a database; otherwise, %FOUND evaluates to FALSE.
cursor%NOTFOUND	The %NOTFOUND attribute evaluates to TRUE when the preceding FETCH statement does not correspond to at least one row in a database; otherwise, %NOTFOUND evaluates to FALSE
cursor%ISOPEN	The %ISOPEN attribute evaluates to TRUE when the target cursor is open; otherwise, %ISOPEN evaluates to FALSE.
cursor %ROWCOUNT	The %ROWCOUNT attribute reveals the number of rows fetched so far for an explicitly declared cursor.

TABLE 5-2. *Attributes of Explicitly Declared Cursors*

which is a revision of the block in the previous exercise, and notice how a cursor FOR loop can simplify the steps necessary to process a cursor?

```
DECLARE
  CURSOR partsRows IS
    SELECT * FROM parts
    ORDER BY id;
 BEGIN
   FOR currentPart IN partsRows LOOP
   DBMS_OUTPUT.PUT_LINE(currentPart.id ||’’ ||currentPart.description);
 END LOOP;
END;
/
```

The program output is as follows:

```
1     Fax machine
2     Copy Machine
3     Laptop PC
4     Desktop PC
```

```
      5    Scanner
      6    Mouse
      7    Laser Printer

      PL/SQL procedure successfully completed.
```

If you do not see the above output, chances are you exited the SQL*Plus and login as practice05/password. You may issue the following SQL command to enable the server output.

```
SQL>set serveroutput on;
```

EXERCISE 5.15: Declaring Cursors with Parameters

When you declare a cursor, you can also declare one or more cursor parameters that the PL/SQL program uses to define the cursor's record selection criteria at run time. When a program opens a cursor that has a cursor parameter, the program can indicate a value for each cursor parameter. To learn how to declare and use a cursor that has a cursor parameter, enter the following anonymous PL/SQL block.

```
DECLARE
 CURSOR customersRows (salesRepId INTEGER) IS
 SELECT id, firstname ||'' ||lastname AS name
 FROM customers
 WHERE s_id = salesRepId;
BEGIN
 DBMS_OUTPUT.PUT_LINE('Sales Rep #1''s Customers');
 FOR currentCustomer IN customersRows (1) LOOP
 DBMS_OUTPUT.PUT_LINE('ID: '||currentCustomer.id ||
 ', NAME: ' ||currentCustomer.name);
END LOOP;
END;
/
```

The program output is as follows:

```
Sales Rep #1's Customers
ID: 2, NAME: Bill Musial
ID: 3, NAME: Danielle Sams
ID: 7, NAME: Joseph Wiersbicki
ID: 9, NAME: Dorothy Clay
ID: 10, NAME: Dave Haagensen

PL/SQL procedure successfully completed.
```

EXERCISE 5.16: Manipulating a Cursor's Current Row

As a program fetches individual rows from a cursor's result set, it is accessing the cursor's current row. A PL/SQL program can take advantage of the special

CURRENT OF clause in the WHERE condition of an UPDATE or DELETE statement that must process the current row of a cursor. Enter the following anonymous PL/SQL block, which teaches you how to use the CURRENT OF clause in the WHERE condition of a DELETE statement. This block deletes the two new records inserted into the PARTS table by previous example programs in this chapter.

```
DECLARE
 CURSOR partsRows (partId INTEGER) IS
 SELECT * FROM parts
 WHERE id >= partId
  FOR UPDATE;
   BEGIN
    FOR currentPart IN partsRows(6)LOOP--selects
                                   --parts 6 and 7
    DELETE FROM parts
    WHERE CURRENT OF partsRows;
     DBMS_OUTPUT.PUT_LINE('Deleted part' ||currentPart.id
||',' || currentPart.description);
END LOOP;
END;
/
```

Assuming that your PARTS table has rows for parts 6 and 7 (a Mouse and a Laser Printer, respectively), which were inserted by earlier exercises in this chapter, the program output should be as follows:

```
Deleted part 5, Mouse
Deleted part 7, Laser Printer

PL/SQL procedure successfully completed.
```

Notice in the declaration of the cursor in the example program that when you declare a cursor with the intention of updating or deleting rows fetched by the cursor, you must declare the cursor's defining query with the FOR UPDATE keywords. This requirement forces Oracle to lock the rows in the cursor's result set, which prevents other transactions from updating or deleting the same rows until your transaction commits or rolls back.

5.4.2 Working with Collections

PL/SQL blocks can also declare and use collections. A collection in a PL/SQL program is a variable that is made up of an ordered set of like elements. To create a collection, you must first declare either a nested table type or a varray (varying array) type, and then declare a variable of the collection type. The next few sections explain more about nested tables and varrays.

NOTE
Oracle's PL/SQL also supports a third collection type, PL/SQL tables (index-by

tables), for backward compatibility with previous versions of Oracle. However, you should use nested tables rather than PL/SQL tables when developing new applications, to gain additional functionality. This chapter does not explain PL/SQL table types further.

5.4.3 Nested Tables

A PL/SQL program can use a nested table type to create variables that have one or more columns and an unlimited number of rows, just like tables in a database. The general syntax for declaring a nested table type is as follows:

```
TYPE tableType IS TABLE OF
(datatype |{variable|table.column}%TYPE | table%ROWTYPE)
[NOT NULL];
```

The following exercises teach you how to declare, use, and manipulate nested tables.

EXERCISE 5.17: Declaring and Initializing a Nested Table

Enter the following anonymous PL/SQL block, which demonstrates how to declare a nested table type and then use the type to declare a new collection variable. The following example also demonstrates how to initialize a collection with its constructor method and then reference specific elements in the nested table collection by subscript.

```
DECLARE
 --declare a nested table type of INTEGERs
   TYPE integerTable IS TABLE OF INTEGER;
 -- declare and initialize a collection with its
constructor
   tempIntegers integerTable := integerTable(1, 202, 451) ;
   BEGIN
   FOR i IN 1 .. 3 LOOP
   DBMS_OUTPUT.PUT_LINE('Element # '||i ||' is '   ||
tempIntegers(i));
  END LOOP;
END;
/
```

The program output is as follows:

```
Element #1 is 1
Element #2 is 202
Element #3 is 451

PL/SQL procedure successfully completed.
```

This very simple example demonstrates a few fundamental points you will need to understand about nested tables.

- Before you can use a collection, such as a nested table, you must initialize it using the type's corresponding constructor. PL/SQL automatically provides a constructor with the same name as the collection type. When you call a constructor to initialize a collection variable, you can specify a comma-separated list of initial elements for the collection, or a set of empty parentheses to initialize the collection as NULL.
- A nested table collection can have any number of rows. The size of a table can increase or decrease dynamically, as necessary. The next practice exercise will show how to add and delete elements in a nested table.
- Nested tables are initially dense—initially, all elements have consecutive subscripts.

EXERCISE 5.18: Using Collection Methods with a Nested Table

PL/SQL supports several different collection methods that you can use to manipulate collections. To use a collection method, an expression in a PL/SQL program names the collection with the collection method as a suffix, using dot notation. Table 5-3 lists the collection methods that are available in PL/SQL.

Collection Method	Description
COUNT	Returns the number of elements currently in the collection
DELETE[(x[,y] ...)]	Deletes some or all of the collection's elements without deallocating the space used by the elements.
EXISTS(x)	Returns TRUE if the xth element in the collection exists. Otherwise, the method returns FALSE.
EXTEND[(x[,y])]	Appends x copies of the yth element to the tail end of the collection. If y is omitted, appends x null elements to the collection. If both x and y are omitted, appends a single null element to the collection
FIRST	Returns the index number of the first element in the collection.
LAST	Returns the index number of the last element in the collection.
LIMIT	Returns the maximum number of elements that a varray's collection can contain.
NEXT(x)	Returns the index number of the element after the xth element of the collection.
PRIOR(x)	Returns the index number of the element before the xth element of the collection.
TRIM(x)	Trims x elements from the end of the collection.

TABLE 5-3. *Collection Methods*

Enter the following anonymous PL/SQL block, which demonstrates how to use several collection methods with a nested table of records.

```
DECLARE
  -- declare a nested table type of PARTS
   TYPE partsTable IS TABLE OF parts%ROWTYPE;
   -- declare and initialize a collection with its constructor
   tempParts partsTable := partsTable( );
   -- cursor to fetch rows from PARTS table
      CURSOR nextPartsRow IS
         SELECT * FROM parts ORDER BY id;
          currentElement INTEGER;

BEGIN
   /* ADD ELEMENTS TO COLLECTION
  -- Create 10 new elements (rows) in the collection
   -- using the collection's EXTEND method.
  */
  tempParts.EXTEND(10);

   /* POPULATE COLLECTION
   -- Use a cursor to populate every even numbered element
   -- in the tempParts nested table collection with the
   -- five rows in the PARTS table.
   */
 FOR currentPart IN nextPartsRow LOOP
 tempParts(nextPartsRow%ROWCOUNT * 2) := currentPart;
END LOOP;

/* OUTPUT COLLECTION ELEMENT FIELDS
  -- Output the ID and DESCRIPTION fields of all elements
  -- in the tempParts nested table collection; if the
  -- element is NULL, indicate this.
*/
  DBMS_OUTPUT.PUT_LINE( ‘Densely populated tempParts
elements:’||tempParts.COUNT);
  currentElement := tempParts.FIRST;
      FOR i IN 1 .. tempParts.COUNT LOOP
      DBMS_OUTPUT.PUT('Element #' || currentElement || 'is');
      IF tempParts(currentElement).id IS NULL THEN
      DBMS_OUTPUT.PUT_LINE('an empty element.');
          ELSE
          DBMS_OUTPUT.PUT_LINE(‘ID:‘||tempParts(currentElement).id ||
‘,’ ||‘DESCRIPTION:’ ||tempParts(currentElement).description);
      END IF;
 currentElement := tempParts .NEXT (currentElement);
 END LOOP;
  /* DELETE EMPTY COLLECTION ELEMENTS
   -- Use the collection's DELETE method to delete the
   -- empty elements (rows) from the nested table.
  */
 FOR i IN 1 .. tempParts.COUNT LOOP
 IF tempParts(i).id IS NULL THEN
```

```
  tempParts.DELETE(i);
 END IF;
END LOOP;

   /* OUTPUT SPARSE VERSION OF THE COLLECTION
   -- Print out ID and DESCRIPTIONS of the elements in
   -- the sparse version of the nested table.
   */
  DBMS_OUTPUT.PUT_LINE( 'Sparsely populated tempParts elements: '||
tempParts.COUNT); currentElement := tempParts.FIRST;
  FOR i IN 1 .. tempParts.COUNT LOOP
   DBMS_OUTPUT.PUT('Element # '||currentElement || ' is ');
      IF tempParts(currentElement).id IS NULL THEN
      DBMS_OUTPUT.PUT_LINE('an empty element.');
      ELSE
      DBMS_OUTPUT.PUT_LINE('ID:'||tempParts(currentElement).id ||
',' ||' DESCRIPTION: '||tempParts(currentElement).description);
      END IF;
  currentElement := tempPartS.NEXT(currentElement);
 END LOOP;
END;
/
```

The program output is as follows:

```
Densely populated tempParts elements: 10
Element #1 is an empty element.
Element #2 is ID: 1, DESCRIPTION: Fax Machine
Element #3 is an empty element.
Element #4 is ID: 2, DESCRIPTION: Copy Machine
Element #5 is an empty element.
Element #6 is ID: 3, DESCRIPTION: Laptop PC
Element #7 is an empty element.
Element #8 is ID: 4, DESCRIPTION: Desktop PC
Element #9 is an empty element.
Element #10 is ID: 5, DESCRIPTION; Scanner
Sparsely populated tempParts elements: 5
Element #2 is ID: 1, DESCRIPTION: Fax Machine
Element #4 is ID: 2, DESCRIPTION: Copy Machine
Element #6 is ID: 3, DESCRIPTION: Laptop PC
Element #8 is ID; 4, DESCRIPTION: Desktop PC
Element 110 is ID: 5, DESCRIPTION: Scanner

PL/SQL procedure successfully completed.
```

Besides demonstrating how to use the EXTEND, COUNT, FIRST, NEXT, and DELETE collection methods, this example shows that while nested tables are initially dense (elements have consecutive subscripts), they can later become sparse (elements can have nonconsecutive subscripts) if the program deletes elements from the collection. With this possibility in mind, the loops that output elements in the example program do not rely on the loop's counter variable to reference collection elements.

5.4.4 Varying Arrays

A program can also declare a varying array (varray) type to create table-like variables that have one or more columns and a limited number of rows. The general syntax for declaring a varray type is as follows:

```
TYPE varrayType IS(VARRAY | VARYING ARRAY)(size)OF
(datatype | {variableltable.colujnn}%TYPE | table%ROWTYPE)
[NOT NULL];
```

For the most part, varray collections are similar to nested table collections, with the following important differences.

- When you declare a varray type, you must declare the number of elements in the varray, which remains constant.
- Varrays must remain dense. A program must insert members into a varray using consecutive subscripts, and cannot delete elements from a varray.

5.4.5 Handling Program Exceptions

A program is not complete unless it contains routines to process the errors that can occur during program execution. Rather than embed error-handling routines into the body of a program, a PL/SQL program addresses error-handling requirements using exceptions and associated exception handlers. An exception is a named error condition. A PL/SQL program raises a named exception when it detects an error, and then it passes control to an associated exception handler routine that is separate from the main program body. The next two exercises teach you more about exceptions and exception handling, including predefined and user-defined exceptions.

EXERCISE 5.19: Handling Predefined Exceptions

PL/SQL includes many predefined exceptions that correspond to several common Oracle errors. When a program encounters a predefined exception, it automatically transfers program control to the associated exception handler—a program does not have to explicitly perform checks for predefined exceptions.

PL/SQL identifies almost 20 predefined exceptions. For example, enter the following anonymous PL/SQL block, which includes exception handlers to handle the NO_DATA_FOUND and TOO_MANY_ROWS predefined PL/SQL exceptions:

- A PL/SQL program automatically raises the NO_DATA_FOUND exception when a SELECT INTO statement has a result set with no rows.
- A PL/SQL program automatically raises the TOO_MANY_ROWS exception when a SELECT INTO statement has a result set with more than one row.

NOTE
For a complete list of predefined exceptions, see your Oracle documentation.

```
DECLARE
 PROCEDURE printOrder(thisOrderDate IN DATE)IS
 thisId INTEGER;
   BEGIN
   SELECT id INTO thisId FROM orders
   WHERE orderdate = thisOrderDate;
   DBMS_OUTPUT.PUT_LINE(‘Order ID’ ||thisId
   ||' on ' ||thisOrderDate);
  EXCEPTION
   WHEN no_data_found THEN
   DBMS_OUTPUT.PUT_LINE('No data found for SELECT.. INTO’);
  END printOrder;
 BEGIN
 printOrder('23-JUN-99');
 printOrder('24-JUN-99');
 printOrder(‘18-JUN-99');
 printOrder(‘19-JUN-99');
  EXCEPTION
  WHEN too_many_rows THEN
     DBMS_OUTPUT.PUT_LINE('Too many rows found for SELECT ..
INTO');
END;
/
```

The program output is as follows:

```
Order ID 14 on 23-JUN-99
No data found for SELECT .. INTO
Too many rows found for SELECT .. INTO

PL/SQL procedure successfully completed.
```

There are several important points that this example demonstrates about exception handling in general:

- You can include an exception-handling section in any PL/SQL block—both the anonymous PL/SQL block and its subprogram (the printOrder procedure) have their own exception-handling sections.
- The first call to the printOrder procedure does not raise any exceptions and prints the ID of the only order placed on 23-Jun-99.
- The second call to the printOrder procedure raises the predefined exception NO_DATA_FOUND, because the SELECT ... INTO statement in the procedure does not retrieve any rows. The NO_DATA_FOUND exception handler local to the procedure handles the exception by printing a message, and then returns control to the calling program, the anonymous PL/SQL block, which then calls the printOrder procedure a third time.
- The third call to the printOrder procedure raises the predefined exception TOO_MANY_ROWS, because the SELECT ... INTO statement in the

procedure returns more than one row. The procedure's exception-handling section does not handle the TOO_MANY_ROWS exception locally, so the exception propagates to the calling program, the anonymous PL/SQL block, which does have an exception handler for TOO_MANY_ROWS. The exception handler prints a message, and then passes control to the calling program, which in this case, is SQL*Plus. Execution of the anonymous PL/SQL block stops, and the fourth call to the printOrder procedure never executes.

NOTE
If a statement in a PL/SQL block raises an exception and the block does not have an associated exception handler, the program stops execution and returns the error number and message that correspond to the exception to the calling program.

EXERCISE 5.20: Declaring and Handling User-Defined Exceptions

A program can also declare user-defined exceptions in the declarative section of a block. However, a program must perform explicit checks for a user-defined exception that then raise the exception. Enter the following anonymous PL/SQL block, which demonstrates the use of user-defined exceptions and corresponding exception handlers.

```
DECLARE
 partNum INTEGER := 10;
 errNum INTEGER;
 errMsg VARCHAR2 (2000);
 invalidPart EXCEPTION;
BEGIN
 UPDATE parts
  SET description = 'Test'
   WHERE id = partNum;
--Explicitly check for the user-defined exception
IF SQL%NOTFOUND THEN
  RAISE invalidPart;
 END IF;
 DBMS_OUTPUT.PUT_LINE(‘Part updated.’);
EXCEPTION
 WHEN invalidPart THEN
  raise_application_error(-20003,'Invalid Part ID#' ||partNum);
WHEN OTHERS THEN
   errNum := SQLCODE;
   errMsg := SUBSTR(SQLERRM, 1, 100);
  raise_application_error (-20000, errNum ||‘’ ||errMsg);
END;
/
```

The program output is as follows:

```
DECLARE
*
```

```
ERROR at line 1:
ORA-20003: Invalid Part ID #10
ORA-06512: at line 17
```

The example in this section introduces several interesting points about exception handling:

- You declare a user-defined exception in the declarative section of a PL/SQL block with the *EXCEPTION keyword*.
- You raise a user-defined exception with the PL/SQL command RAISE.
- A PL/SQL program can use the RAISE__APPLICATION_ERROR procedure to return a user-defined error number and message to the calling environment. All user-defined error messages must be in the range -20000 to -20999.
- A PL/SQL program can use the WHEN OTHERS exception handler to handle all exceptions that do not have a specific handler.
- A PL/SQL program can use the special SQLCODE and SQLERRM functions to return the most recent Oracle error number and message.

5.5 Types of PL/SQL Programs

Now that you understand the basics of the PL/SQL language, it's time to learn more about the different types of programs you can create with PL/SQL, including anonymous PL/SQL blocks, procedures, functions, packages, and database triggers.

5.5.1 Anonymous PL/SQL Blocks

The previous examples in this chapter are all anonymous PL/SQL blocks. An anonymous block is a PL/SQL block that appears within your application. An anonymous PL/SQL block has no name and is not stored for subsequent reuse. The application simply sends the block of code to the database server for processing at run time. Once the server executes an anonymous block, the block ceases to exist.

5.5.2 Stored Procedures and Functions

Several exercises in this chapter taught you how to declare and use PL/SQL subprograms (procedures and functions) within PL/SQL blocks to encapsulate frequently used tasks. A subprogram is a named PL/SQL program that can take parameters and be called again and again to perform work. You can also store procedures and functions as compiled bits of application logic inside an Oracle database as named schema objects. By centralizing common procedures and functions in the database, any application can make use of them to perform work. Judicious use of stored procedures and functions can increase developer productivity and simplify application development. The next two practice exercises demonstrate how to create stored procedures and functions in an Oracle database.

EXERCISE 5.21: Creating and Using Stored Procedures

To create a stored procedure in an Oracle database, use the SQL command CREATE PROCEDURE. Specification of a stored PL/SQL subprogram is basically the same as when you declare a subprogram in the declarative section of a PL/SQL block. However, when you declare a stored procedure, you can use the AUTHID CURRENT_USER or AUTHID DEFINER options to indicate the privilege domain that Oracle uses when executing the procedure.

- If you create the procedure with the AUTHID CURRENT_USER option, Oracle executes the procedure (when called) using the privilege domain of the user calling the procedure. To execute the procedure successfully, the caller must have the privileges necessary to access all database objects referenced in the body of the stored procedure.
- If you create the procedure with the default AUTHID DEFINER option, Oracle executes the procedure using the privilege domain of the owner of the procedure. To execute the procedure successfully, the procedure owner must have the privileges necessary to access all database objects referenced in the body of the stored procedure. To simplify privilege management for application users, the default AUTHID DEFINER option should be your typical choice when creating a stored procedure—this way, you do not have to grant privileges to all the users that need to call the procedure.

NOTE
For more information about database access privileges, see chapter 9.

Enter the following example, which demonstrates how to create and store the familiar printLine procedure in an Oracle database.

```
CREATE OR REPLACE PROCEDURE printLine(
width IN INTEGER,
Chr IN CHAR DEFAULT '-')
AUTHID DEFINER
IS
BEGIN
FOR i IN 1 .. width LOOP
DBMS_OUTPUT.PUT(chr);
 END LOOP;
DBMS_OUTPUT.PUT_LINE('');
END printLine;
/
```

Now, you can use the printLine procedure in any other PL/SQL program just by calling the stored procedure in the database. For example, try entering the following anonymous PL/SQL blocks, which use the printLine stored procedure.

```
BEGIN
Printline (40, '*');                        --print a line of 40*s
END;
/
BEGIN
```

```
printLine (width => 20, chr => ' =' ) ; -- print a line of 20 =s
END;
/
BEGIN
printLine(10);                          -- print a line of 10 -s
END;
/
```

The program outputs are as follows:

```
***************************************
PL/SQL procedure successfully completed.
====================
PL/SQL procedure successfully completed.
----------
PL/SQL procedure successfully completed.
```

EXERCISE 5.22: Creating and Using Stored Functions

To create a stored function in an Oracle database, use the SQL command CREATE *FUNCTION*. Specify a function just as you would in the declarative section of a PL/SQL block—do not forget to declare the function's return type, and to use one or more RETURN statements in the body of the function to return the function's return value. Enter the following example, which demonstrates how to create and store the orderTotal function in an Oracle database.

```
CREATE OR REPLACE FUNCTION orderTotal(orderId IN INTEGER)
RETURN NUMBER
IS
 orderTotal NUMBER;
  tempTotal NUMBER;
BEGIN
 SELECT SUM(i.quantity * p.unitprice) INTO orderTotal
 FROM items i, parts p
 WHERE i.o_id = orderId
 AND i.p_id = p.id
 GROUP BY i.o_id;
 RETURN orderTotal;
END orderTotal;
/
```

Now, enter the following query, which uses the orderTotal function to return the IDs and order dates of all orders that total more than $5,000.

```
SELECT id, orderdate
FROM orders
WHERE orderTotal(id) > 5000;
```

The results are as follows:

```
ID       ORDERDATE
-----    -----------
```

```
1          18-JUN-99
5          19-JUN-99
10         21-JUN-99
11         22-JUN-99
```

5.6 Packages

A package is a group of procedures, functions, and other PL/SQL constructs, all stored together in a database as a unit. Packages are especially useful for organizing a number of PL/SQL procedures and functions that relate to a particular database application.

A package has two parts: a specification and a body,

- A package *specification* defines the interface to the package. In a package specification, you declare all package variables, constants, cursors, procedures, functions, and other constructs that you want to make available to programs outside the package. In other words, everything that you declare in a package's specification is public. You declare a package specification with the SQL command *CREATE PACKAGE*.
- A package body defines all public procedures and functions declared in the package specification. Additionally, a package body can include other construct definitions not in the specification; such package constructs are private (available only to programs within the package). You declare a package body with the SQL command *CREATE PACKAGE BODY.*

All variables, constants, and cursors declared in either a package specification or body outside of a subprogram are considered global. Unlike private variables, constants, and cursors declared within specific procedures and functions, global constructs are available to all package procedures and functions and have a state that persists independent of any particular package subprogram on a per-session basis.

EXERCISE 5.23: Declaring and Using a Package

Enter the following example to create a very simple package called partMgmt, which demonstrates some of the functionality available with PL/SQL packages.

```
CREATE OR REPLACE PACKAGE partMgmt IS
 -- Public subprograms
  PROCEDURE insertPart (partRecord IN parts%ROWTYPE);
  PROCEDURE updatePart (partRecord IN parts%ROWTYPE);
  PROCEDURE deletePart (partId IN INTEGER);
  PROCEDURE printPartsProcessed;
END partMgmt;
/

CREATE OR REPLACE PACKAGE BODY partMgmt AS
 -- Private global variable
 rowsProcessed INTEGER := 0;
```

```
   -- Public subprograms
   PROCEDURE insertPart (partRecOrd IN parts%ROWTYPE) IS
    BEGIN
    INSERT INTO parts
     VALUES (partRecord.id, partRecord.description,
partRecord.unitprice, parCRecord.onhand, partRecord.reorder);
    rowsProcessed := rowsProcessed + 1;
     END insertPart;

   PROCEDURE updatePart (partRecord IN parts%ROWTYPE) IS
     BEGIN
     UPDATE parts
     SET description = partRecord.description,
     unitprice = partRecord.unitprice,
     onhand = partRecord.onhand,
   reorder = partRecord.reorder
  WHERE id = partRecord.id;
  rowsProcessed := rowsProcessed + 1;
 END updatePart;

   PROCEDURE deletePart (partId IN INTEGER) IS
    BEGIN
    DELETE FROM parts
    WHERE id = partId;
    rowsProcessed := rowsProcessed + 1;
  END deletePart;

PROCEDURE printPartsProcessed IS
  BEGIN
  DBMS_OUTPUT.PUT_LINE( 'Parts processed this session: '||rowsProcessed);
   END printPartsProcessed;
END partMgmt;
/
```

Now, enter the following anonymous PL/SQL block, which uses the insertPart procedure of the partMgmt package to insert a new part in the PARTS table.

```
DECLARE
newPart parts%ROWTYPE;
BEGIN
newPart.id := 6;
newPart.description := 'Mouse';
newPart.unitprice := 49; newPart.onhand := 1200;
newPart.reorder := 500;

partMgmt.insertPart(newPart);
END;
/
```

Notice that when you reference a package object (for example, a global variable or subprogram), you must use dot notation to qualify the package object with its package name. Now, enter the following anonymous PL/SQL block to update the new part's ONHAND quantity using the updatePart procedure in the partMgmt package.

```
DECLARE
aPart parts%ROWTYPE;
BEGIN
SELECT * INTO aPart FROM parts
WHERE id = 6; aPart.onhand := 1123;

partMgmt.updatePart(aPart);
END;
/
```

Now use the SQL*Plus command EXECUTE to execute the deletePart procedure of the partMgmt package to delete the newest part. The EXECUTE command is equivalent to surrounding a procedure call with the BEGIN and END keywords that delimit the start and end of an anonymous PL/SQL block.

```
EXECUTE partMgmt.deletePart(6);
```

Finally, enter the following EXECUTE command to output the number of rows in the PARTS table processed by the current session using the partMgmt package.

```
EXECUTE partMgmt.printPartsProcessed;
```

The program output is as follows:

```
 Parts processed this session: 3

PL/SQL procedure successfully completed.
```

The printPartsProcessed procedure demonstrates that the rowsProcessed global variable retains its state, independent of calls to individual package subprograms.

5.6.1 Prebuilt Utility Packages

Oracle includes several prebuilt utility packages that provide additional functionality not available with SQL or PL/SQL. Table 5-4 lists several of the prebuilt packages available with Oracle.

Package Name	Description
DBMS_ALERT	Procedures and functions that allow applications to name and signal alert conditions without polling
DBMS_AQ DBMS_AQADM	Procedures and functions to queue the execution of transactions and administer queuing mechanisms
DBMS_DDL DBMS_UTILITY	Procedures that provide access to a limited number of DDL statements inside PL/SQL programs

DBMS_DESCRIBE	Procedures that describe the API for stored procedures and functions
DBMS_JOB	Procedures and functions to manage a database's job queuing mechanisms
DBMS_LOB	Procedures and functions to manipulate BLOBs, CLOBs, NCLOBs, and BFILEs
DBMS_LOCK	Procedures and functions that allow applications to coordinate access to shared resources
DBMS_OUTPUT	Procedures and functions that allow a PL/SQL program to generate terminal output
DBMS_PIPE	Procedures and functions that allow database sessions to communicate using pipes (communication channels)
DBM5_ROWID	Procedures and functions that allow applications to easily interpret a base-64 character external ROWID
DBMS_SESSION	Procedures and functions to control an application user's session
DBMS_SQL	Procedures and functions to perform dynamic SQL from within a PL/SQL program
DBM5_TRANSACTION	Procedures to perform a limited amount of transaction control
UTL_FILE	Procedures and functions that allow a PL/SQL program to read and write text files to the server's file system

TABLE 5-4. *Some of the Many Prebuilt Packages Available with Oracle (continued)*

NOTE
See your Oracle documentation for complete information about the APIs for all prebuilt packages and examples of their use.

5.7 Database Triggers

A *database trigger* is a stored procedure that you associate with a specific table. When applications target the table with a SQL DML (data modification language) statement that meets the trigger's execution conditions, Oracle automatically fires (executes) the trigger to perform work. Therefore, you can use triggers to customize an Oracle database server's reaction to application events.

To create a database trigger, you use the SQL command CREATE TRIGGER. The simplified syntax (not complete) for the CREATE TRIGGER command is as follows:

```
CREATE[OR REPLACE]TRIGGER trigger
(BEFORE |AFTER)
(DELETE/|INSERT|UPDATE [OF Column[,column]...    ])
[OR{DELETE|INSERT|UPDATE[OF column[,column]...]} ] ...
ON table}
FOR EACH ROW[WHEN condition]]
...PL/SQL block...
END[trigger]
```

A trigger definition includes the following unique parts:

- A trigger's definition includes a list of trigger statements, including INSERT, UPDATE, and/or DELETE, that fire the trigger. A trigger is associated with one, and only one, table.
- A trigger can be set to fire before or after the trigger statement to provide specific application logic.
- A trigger's definition indicates whether the trigger is a statement trigger or a row trigger.
- A statement trigger fires only once, no matter how many rows the trigger statement affects.

EXERCISE 5.24: Creating and Using Database Triggers

Enter the following CREATE TRIGGER statement to create a trigger that automatically logs some basic information about the DML changes made to the PARTS table. The logPartChanges trigger is an after-statement trigger that fires once after the triggering statement, no matter how many rows the trigger statement affects.

```
CREATE OR REPLACE TRIGGER logPartChanges
AFTER INSERT OR UPDATE OR DELETE ON Parts
DECLARE
statementType CHAR(l);
BEGIN
IF INSERTING THEN
statementType := ‘I’;
ELSIF UPDATING THEN
statementType := ‘U’;
ELSE
statementType := ‘D’
END IF;
INSERT INTO partsLog
VALUES (SYSDATE, statementType, USER);
END logPartChanges;
/
```

Notice in the logPartChanges trigger that when a trigger allows different types of statements to fire the trigger, the INSERTING, UPDATING, and DELETING predicates allow conditional statements to identify the type of statement that actually fired the trigger.

Now create the logDetailedPartChanges trigger, which is a before-row trigger that logs more detailed information about DML modifications that target the PARTS table.

```
CREATE OR REPLACE TRIGGER logDetailedPartChanges
BEFORE INSERT OR UPDATE OR DELETE ON parts
FOR EACH ROW
BEGIN
INSERT INTO detailedpartslog
VALUES (SYSDATE, USER,
:new.id, :new.description, :new.unitprice,
:new.onhand, :new.reorder,
:old.id, :old.description, :old.unitprice.
:old.onhand, :old.reorder
) ;
END logDetailedPartChanges;
/
```

NOTE
Optionally, a row trigger can include a trigger restriction—a Boolean condition that determines when to fire the trigger.

Notice in the logDetailedPartChanges trigger that :new and :old correlation values allow a row trigger to access new and old field values of the current row. When a trigger statement is an INSERT statement, all old field values are null. Similarly, when a trigger statement is a DELETE statement, all new field values are null.

Finally, let's test out our new triggers and see what they actually do. Enter the following SQL statements that insert, update, and delete rows in the PARTS table, and then query the PARTSLOG and DETAILEDPARTSLOG tables.

```
INSERT INTO parts
VALUES(6, ’Mouse’, 49, 1200, 500);

UPDATE parts
SET onhand= onhand-1O;
DELETE FROM parts
WHERE    id=6;


SELECT * FROM partsLog;


SELECT newid, newonhand, oldid, oldonhand FROM detailedpartslog;
```

The result sets for the queries should be similar to the following:

```
CHANGEDATE   C       USERID
----------   --   ----------
16-AUG-99    I     PRACTICE05
16-AUG-99    U     PRACTICE05
```

```
16-AUG-99    D     PRACTICE05


NEWID NEWONHAND OLDID OLDONHAND
----- --------- ----- ----------
6     1200
1     267         1        277
2     133         2        143
3     7521        3        7631
4     5893        4        5903
5     480         5        490
6     1190        6        1200
                  6        1190
```

An important point to understand about database triggers is that triggers execute within the context of the current transaction. Therefore, if you were to roll back the current transaction, you would get the "no rows selected" message when you subsequently queried the PARTSLOG and DETAILEDPARTSLOG tables.

Chapter Summary

This class/chapter has provided you with a broad overview of the extended capabilities that PL/SQL offers for creating powerful database access programs. You've learned about the basics of the language itself, as well as how to create PL/SQL programs using stored procedures, functions, packages, and database triggers.