



# PART I

## Language Syntax





# CHAPTER 1

## Overview of PL/SQL



To understand Procedural Language/Structured Query Language (PL/SQL) program units, you must first understand the basic language syntax of PL/SQL. That is the subject of this chapter. These topics are tested in the first exam of the Oracle Internet Application Developer's certification, but these same topics are the backdrop and foundation of the entire exam on program units. Many issues of working with program units, such as privileges, permissions, interdependencies, and parameter syntax, are built upon the fundamentals that are reviewed in this chapter. So, although these topics are not explicitly tested in the exam, a complete understanding of them is required to perform successfully on the exam. Even if you already know this material, it is advised that you skim it before moving to the next chapter.

In this chapter, you will review and understand the following topics:

- An introduction to the PL/SQL language
- PL/SQL syntax fundamentals
- Declaration section
- Processing section
- Cursors
- Advanced datatype declaration
- Exception handling
- Working with blocks
- An introduction to program units

## An Introduction to the PL/SQL Language

PL/SQL is a proprietary computer programming language that is owned by the Oracle Corporation. It combines the power of SQL with the best and most commonly used features of some popular third-generation languages, such as C, FORTRAN, COBOL, Ada, Pascal, and PL/1. PL/SQL's features include variable declaration, assignment statements, conditional logic, loops, subprogram and parameter passing, and error handling, all integrated with the full power of the SQL language. Using PL/SQL, application developers can create programs that process database records individually with far more control than a stand-alone SQL script will enable, such as fully controllable conditional logic and looping.

SQL's Data Manipulation Language (DML) commands, including SELECT, INSERT, UPDATE, and DELETE, can be used in PL/SQL statements. Some SQL statements, such as SELECT, require some minor accommodations (such as the INTO clause). Most of Oracle's SQL functions can be used from within PL/SQL expressions inside of PL/SQL programs.

PL/SQL programs can be structured as program units, such as procedures and functions. Program units can be placed on either the server side or the client side. On the server side, program units can be stored in the database and be called to execute from other PL/SQL applications as well as from simple SQL\*Plus command-line statements. On the client side, Oracle Corporation has built support for PL/SQL program units into various Oracle development tools, such as Form Builder and Report Builder, to extend the capability of those tools and provide capabilities for detailed data manipulation. By having the flexibility to store program units on either the client or the server, the developer is empowered to choose either implementation for each program unit in order to tune the application for optimal performance; in other words, by being able to move the program unit around, the developer can choose to put program units that frequently interact with the database on the server side and move those program units that perform more direct interaction with the user on the client side. The choice is up to the developer, and the result is an optimally tuned application.

Procedures and functions can be combined into larger program units called *packages*. There are several advantages to using packages; they offer some security and performance improvement advantages, not to mention the simple benefit of organizing a set of program units in one place. When several applications are stored in the same database, it's a lot easier to see which program units support which applications when they are collected in a package than if they are thrown into a schema's alphabetical listing of objects. Even if only one application is stored in a schema, some of the program units might be intended for one work purpose, perhaps accounting, and another set of program units might be meant for something else, such as application maintenance and user account tracking. By storing the relevant program units in separate packages, it's easier for developers to understand and maintain the application.

Oracle Corporation provides some prewritten PL/SQL packages to extend the capability of the PL/SQL language, and Oracle also provides support for special features and techniques, such as scheduling batch jobs, working with operating system files, or working with Oracle database snapshots. PL/SQL program units can be stored inside the database in the form of database triggers, which are associated with events and are "fired" in response to various activities, such as SQL UPDATES and DELETES, whether the triggering SQL transaction originated from another PL/SQL program or not. In addition, PL/SQL program units can be created for use from within SQL statements or be used to "wrap" Java programs for storage in the database.

## 6 OCP Developer PL/SQL Program Units Exam Guide

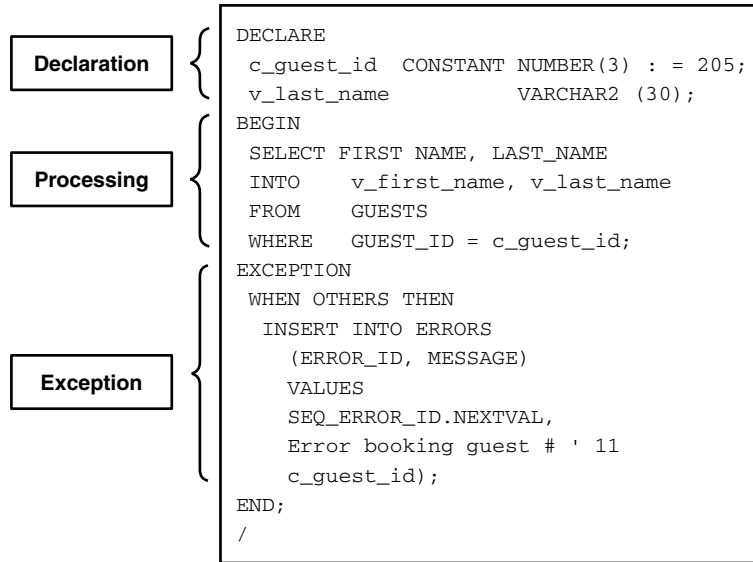
There are two major categories of PL/SQL programs: *anonymous* and *named*. Generally, *program units* refer to named PL/SQL programs. An example of an anonymous PL/SQL program, generally known as a *block*, is shown here:

```
DECLARE
  v_counter NUMBER(3);
  v_user     ALL_USERS.USERNAME%TYPE;
  v_today    DATE;
BEGIN
  -- First, get the current date and time,
  -- and the schema of the user running
  -- this program. Print the results on the screen.
  SELECT    SYSDATE, USER
  INTO      v_today, v_user
  FROM      DUAL;

  DBMS_OUTPUT.PUT_LINE(
    'Today : ' ||
    TO_CHAR(v_today, 'FMDay, Month DD, YYYY'));
  DBMS_OUTPUT.PUT_LINE('Schema: ' || v_user);
  -- Next, set up a loop
  v_counter := 0;
  LOOP
    v_counter := v_counter + 1;
    EXIT WHEN v_counter > 10;
    DBMS_OUTPUT.PUT_LINE('Line: ' || v_counter);
  END LOOP;
EXCEPTION
  WHEN OTHERS THEN
    INSERT INTO ERRORS
      VALUES (SEQ_ERROR_ID.NEXTVAL, 'Something went wrong. ');
    COMMIT;
END;
/
```

A PL/SQL block consists of the following sections:

- **Declaration section** Includes the declaration of variables, constants, cursors, and user-defined exceptions for use in the processing section. This section is only required if there is anything to declare. If the processing section doesn't require any user-declared elements, the declaration section is not required.
- **Processing section** This is the main body of the block and is the only required section of the block. It includes assignment statements, conditional logic, loops, and more.



**FIGURE 1-1.** *The structure of a PL/SQL block*

- **Exception-handling section** This is an optional section that will handle any exceptions that may be raised in the processing section. Exceptions may or may not be raised in the processing section during execution, but regardless of whether they are raised or not, this section is optional.

Figure 1-1 shows an example of a PL/SQL block and identifies the various sections.

## PL/SQL Syntax Fundamentals

In this section, you will cover

- Statements
- Identifiers
- Comments, single-line and multiline
- Literals

### Statements

PL/SQL programs are built as a combination of statements, and statements consist of a combination of identifiers, expressions, operators, and PL/SQL reserved words. Semicolons terminate these statements. Furthermore, white space between the elements within expressions is ignored. In other words, statements can continue across many lines until a semicolon is encountered, which terminates the statement and opens the way for the next statement. The job of understanding PL/SQL consists of understanding the rules of the language elements and understanding how to put the elements together into functional programs.

### Identifiers

Identifiers are used in PL/SQL to define the names of various elements. Identifiers are words that you make up. They must start with a character, can be up to 30 characters in length, and cannot include spaces, but can include dollar signs (\$), pound signs (#), and underscores (\_). Identifiers are used to name variables, constants, explicit cursors, user-named exceptions, and program units such as procedures, functions, database triggers, and packages.

PL/SQL is a case-insensitive language, meaning that the language elements are not case sensitive. However, data is case sensitive; in other words, string literals and the values of variables and constants that are defined with character-based datatypes are case sensitive.


### Comments

Comments can be included in PL/SQL code by preceding the comments with special characters. These special characters tell the PL/SQL parser to ignore the line, or lines, that follow.

Single-line comments are preceded by two dashes in succession. A single-line comment can be placed on a line by itself or at the end of a line of valid PL/SQL code.

A multiline comment starts with the characters, `/*`, and is terminated with the characters, `*/`. A multiline comment can span as many lines as you wish. It can begin on a line of existing PL/SQL code and at its termination can be followed by PL/SQL on the same line as the termination characters.

The following code listing shows two examples of single-line comments and one of a multiline comment:



```
DECLARE
  v_schema VARCHAR2(30); -- This is a single line comment
BEGIN
  -- This is a single line comment
  SELECT USER
```

```

INTO    v_schema
FROM    DUAL;
/* This is a multi line comment. It can continue
   on as many consecutive lines as you wish,
   as long as you end it with the proper
   characters. */ DBMS_OUTPUT.PUT_LINE(v_schema);
DBMS_OUTPUT.PUT_LINE('All done. ');
END;
/

```

## Literals

The rules for literals in PL/SQL are the same as the rules used in SQL. Character literals, also known as string literals, are delimited with single quotes. Numeric literals have no quotes, commas, dollar signs, or any other special characters, but they should include any necessary decimal points. Date literals are delimited with single quotes. The date value itself must be in the appropriate format for the database and the format mask that may apply. The default format for dates is 'DD-MON-YY', as in '01-JAN-03' for January 3, 2003. Most Oracle installations also accept the alternative format of 'DD-MON-YYYY' as a default, as in '01-JAN-2003'. A date literal can be expressed in a format that varies from the default, provided the appropriate format mask, according to the rules of SQL, accompanies them.

When using string literals, sometimes it's necessary to define a value that includes the single quote symbol, which is the string delimiter itself. This requires special attention. To include a single quote within the string without delimiting the end of the string, you must include two single quote characters in succession within the string literal. In other words, to specify my last name as a string literal, I have to use the following syntax:

```
'O''Hearn'
```

This will be seen within PL/SQL as the following literal value:

```
O'Hearn
```

One important datatype that PL/SQL uses is the BOOLEAN datatype. This datatype doesn't exist in the database and isn't formally recognized in SQL, but is used in PL/SQL. The BOOLEAN datatype is quite simple; the options for values are either TRUE or FALSE. (NULL, of course, is an option as well.) BOOLEAN literal values are expressed with the reserved words TRUE and FALSE without any quotes.

Finally, the reserved word NULL is considered a valid value of any variable of any datatype. The definition of NULL is properly understood as "the absence of information." In other words, it is not a blank, nor is it a zero; a blank and a zero are

defined values. NULL indicates that the PL/SQL program just doesn't know what the value is.

Literal values can be used throughout PL/SQL, as we shall see in the following sections.

# Declaration Section

The declaration section is where you set up variables, constants, and other elements for use in the processing section. The declaration section is only required if the processing section requires any of these elements. The declaration section can be used to declare variables, constants, cursors, and user-defined exceptions. Each is described in the following sections.

## Variables

Variables are declared with the following format:

```
VARIABLE_NAME DATATYPE;
```

For example, the following code declares a variable called `v_last_name` and declares it with the variable character datatype at a maximum length of 30 characters:

```
v_last_name VARCHAR2(30);
```

Variable names must start with a letter and can be up to 30 characters with no spaces. Names can include the following special characters:

```
#$_
```

A variable's datatype must be declared. The list of available datatypes follows this section.

As an option, a variable can be initialized at declaration with the assignment statement, as in the following examples:

```
v_last_name VARCHAR2(30) := 'Hoddlestein';  
v_found      BOOLEAN      := TRUE;  
v_order      NUMBER(5,2)  := 150.57;
```

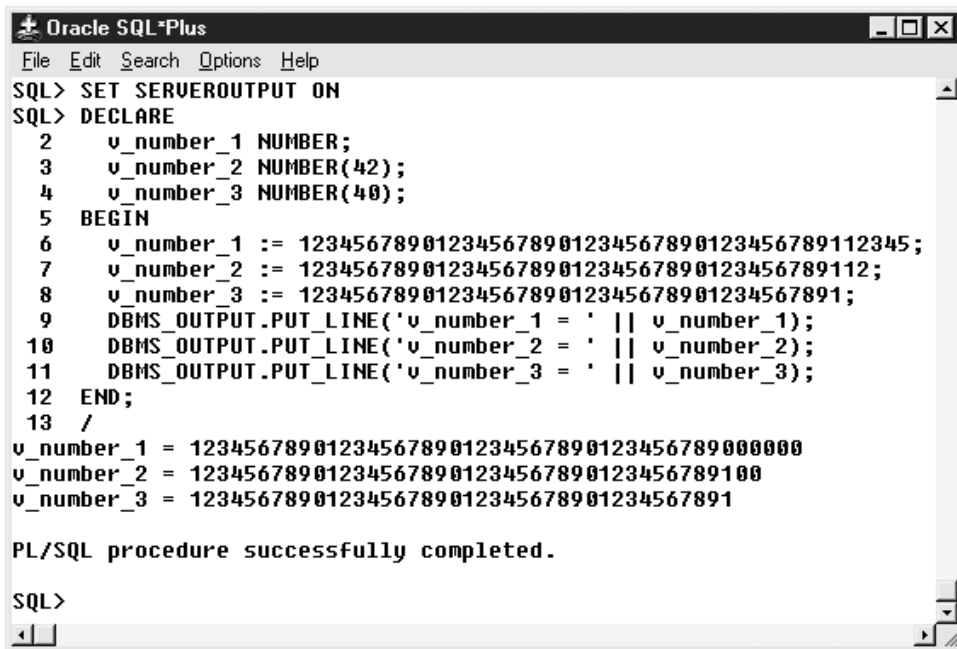
Without initialization, a variable's value at the beginning of the processing section will be NULL.

## Datatypes

To declare variables, constants, and, as we shall see later, parameters, you must be familiar with PL/SQL's datatypes. They are explained here:

- **CHAR(*n*)** Accepts alphanumeric data up to *n* characters in length. The *n* indicator is required. The CHAR(*n*) right pads the data with blanks to ensure that a total of *n* characters will be stored no matter what; for example, if you store the string "Joe" in a variable whose datatype is CHAR(5), then "Joe " is stored—two extra blanks are automatically added to force the string to be five characters long. The maximum length is 32,767 characters. CHAR is short for CHARACTER, which is also accepted, that is, CHARACTER(*n*). In practice, CHAR is not used nearly as much as the alternative, which is VARCHAR2, described later in this list. The length specifier, *n*, is not required. It defaults to 1.
- **VARCHAR2(*n*)** Accepts alphanumeric data up to *n* characters in length. As with CHAR(*n*), the *n* indicator is required, but contrary to CHAR(*n*), VARCHAR2(*n*) will not pad values, but will instead store only as much data as required by the entered value. For example, for a variable declared as VARCHAR2(30), if the provided value is Smith, then only 5 characters are stored, even though up to 30 characters are allowed. The longest VARCHAR2 length allowed is 32,767 characters.
- **DATE** Accepts date descriptions in the same date format required by the database. The default format is 'DD-MON-YY', where DD is the numeric day of the month, MON is the first three letters, capitalized, of the month, and YY is the last two digits of the year. For example, September 28, 1989 is represented as '28-SEP-89'.
- **NUMBER(*n,m*)** Accepts numeric data only. The value for *n* is the size specification, and the value of *m* is the precision. Both are optional, but *m* cannot be used without the presence of *n*. Up to *n* significant digits are accepted. For example, the declaration NUMBER(3) will accept numbers no higher than the number 999. The value for *m* is used to specify the portion of *n* that is defined to the right of the decimal point. In other words, NUMBER(*n,m*) will accept up to *n* significant digits of which *m* are to the right of the decimal point. For example, the declaration NUMBER(3,1) will accept numbers no higher than 99.9. See Figure 1-2 for some examples of variables declared with NUMBER.

The variable `v_number_1` is declared with NUMBER and no size or precision. If both *n* and *m* are left out, then the 40 most significant digits are



```

Oracle SQL*Plus
File Edit Search Options Help
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
  2   v_number_1 NUMBER;
  3   v_number_2 NUMBER(42);
  4   v_number_3 NUMBER(40);
  5 BEGIN
  6   v_number_1 := 123456789012345678901234567890123456789112345;
  7   v_number_2 := 123456789012345678901234567890123456789112;
  8   v_number_3 := 1234567890123456789012345678901234567891;
  9   DBMS_OUTPUT.PUT_LINE('v_number_1 = ' || v_number_1);
 10   DBMS_OUTPUT.PUT_LINE('v_number_2 = ' || v_number_2);
 11   DBMS_OUTPUT.PUT_LINE('v_number_3 = ' || v_number_3);
 12 END;
 13 /
v_number_1 = 123456789012345678901234567890123456789000000
v_number_2 = 123456789012345678901234567890123456789100
v_number_3 = 1234567890123456789012345678901234567891

PL/SQL procedure successfully completed.

SQL>

```

**FIGURE 1-2.** Three examples of variables declared with *NUMBER*

stored. The resulting variable can receive numbers that are larger than 40 significant digits, but will only record the first 40 most significant digits, rounded off. The next variable, `v_number_2`, demonstrates that the maximum allowable value for `n` is documented at 38, but a declared value of 42 works in PL/SQL 8.1.5, and a variable declared as `NUMBER(42)` accepts up to 40 actual significant digits. Finally, Figure 1-2 shows `v_number_3` as an example of a declaration of `NUMBER(40)`, which stores precisely 40 significant digits.

- **BOOLEAN** Stores values corresponding to the keywords `TRUE` or `FALSE` and, of course, can also be `NULL`. For example, an acceptable declaration is as follows:

```
v_found BOOLEAN := TRUE;
```

- **LONG** Accepts very large blocks of alphanumeric data, up to 32,760 bytes.

- **RAW** Accepts binary data, such as multimedia files, and accepts lengths of up to 32,767 characters. PL/SQL will not interpret the contents of a RAW variable.
- **LONG RAW** Accepts very large blocks of binary data, up to 32,760 bytes. Note that this is actually less than the ultimate length accepted by RAW.
- **MLSLABEL** A secure operating system label, used in Trusted Oracle.
- **ROWID** A value that is defined by the Oracle system to uniquely identify the physical storage address of a row in the Oracle database. The value is obtained from the pseudocolumn that is automatically added to every Oracle table by the Oracle system.

There are several datatypes for working with large objects (LOBs):

- **BLOB** A Binary Large Object that accepts up to 4 gigabytes of binary data.
- **CLOB** A Character Large Object that accepts up to 4 gigabytes of alphanumeric data.
- **NCLOB** The multibyte character set alternative to CLOB.
- **BFILE** A pointer to a binary operating system file.

There are several numeric datatypes that store whole number integers:

- **BINARY\_INTEGER and PLS\_INTEGER** Both enable the storage of integers ranging from  $(-2^{31} + 1)$  to  $(2^{31} - 1)$ . The difference between the two is that PLS\_INTEGER will raise an exception if an overflow occurs, whereas BINARY\_INTEGER may not do the same when the result is assigned to a NUMBER datatype.
- **INT, also INTEGER** The same as NUMBER, but it does not accept decimal values. In other words, INT does not accept a precision component, only a size component. For example, INTEGER(5) is accepted.
- **SMALLINT** The same as NUMBER(38).
- **POSITIVE** The same as BINARY\_INTEGER, but with a more limited range, from (1) to  $(2^{31})$ .
- **NATURAL** The same as BINARY\_INTEGER, but with a more limited range, from (0) to  $(2^{31})$ .

## 14 OCP Developer PL/SQL Program Units Exam Guide

There are several numeric datatypes that store real numbers:

- **NUMERIC, DEC, and DECIMAL** The same as NUMBER, except that they declare fixed-point numbers.
- **REAL, FLOAT, and DOUBLE\_PRECISION** The same as NUMBER, but these declare floating-point numbers.
- **TABLE** This is a complex datatype, also known as a composite datatype. A PL/SQL table is similar to an array structure that most third-generation programming languages support.
- **RECORD** This is another composite datatype. A variable declared with a RECORD datatype is a single variable that represents a combination of scalar datatypes (all of the other datatypes mentioned above, except for TABLE) as well as other RECORD variables. A RECORD variable, for example, might be declared that is called PHONE\_NUMBER but represents a combination of VARCHAR2 variables AREA\_CODE, PHONE, and EXTENSION.

### Constants

A PL/SQL constant is a declared element with a value that is assigned when it's declared in the declaration section. Its value cannot be changed by any code that references it, such as that found in the processing or exception-handling section.

The syntax to declare a constant is similar to that used for a variable. The only differences are the use of the reserved word **CONSTANT**, and the fact that the assignment statement that initializes the value is required, not optional, for a constant. Here's an example:

```
 c_tax_rate CONSTANT NUMBER(4,3) := 0.045;
```

The previous sample declares a constant called `c_tax_rate`. Constants, as with variables, require that a datatype be used in the declaration. The same datatypes that can be used to declare a variable are also available for use with constants.

Once the initialized value is set, it cannot be changed. In other words, in the course of the PL/SQL block's execution, the processing section cannot assign a new value to the constant; neither can the exception-handling section change the constant's value.

### Other Declared Elements

You can declare more than just variables and constants in the declaration section. You can also declare explicit cursors and user-defined exceptions. These will be discussed at a later point in this chapter.

## Processing Section

The processing section contains the executable statements that form the main body of your PL/SQL block. A PL/SQL block must have a processing section; it is the only required section in the block.

The processing section includes the following elements:

- Expressions
- Assignment statements
- Conditional statements
- Loops
- Cursor control statements

## Expressions

Expressions are not stand-alone statements, but are instead small units of code that are included in other stand-alone statements. Uses of expressions include assignment statements and conditional statements.

There are two general types of expressions: arithmetic expressions and comparison expressions. Arithmetic expressions combine variables, constants, and literal values with arithmetic operators or SQL functions to produce a single result. For example, consider the following expression:

```
v_wholesale_price + v_admin_expense
```

This expression takes two variables and adds them together, producing a single answer. This expression, as with all expressions, is not a complete stand-alone PL/SQL statement, but could be used within an assignment statement or elsewhere to build a complete statement. The arithmetic operators that PL/SQL uses are the same that SQL uses (see Table 1-1).

Expressions are subject to the rule of operator precedence. The rule of operator precedence dictates which operator is given priority over another operator within an expression.

For example, consider the following expression:

```
3 + 4 * 5
```

The rules of operator precedence dictate that the multiplication operator is evaluated first, which produces an interim result as follows:

```
3 + 20
```

---

Operator	Description	Operator Precedence
()	Parentheses	1
*	Multiplication	2
/	Division	
+	Addition	3
-	Subtraction	

---

**TABLE I-1.** *Arithmetic Operators*

Finally, this remaining expression is evaluated, and the result is computed to be 23. In other words



```
3 + 4 * 5 = 23
```

However, parentheses can be used to override the rules of operator precedence, as follows:




```
(3 + 4) * 5 = 35
```

By overriding the rule of operator precedence, we have altered the expression.

Nested parentheses are allowed, and the innermost set of parentheses will be evaluated before the others. In situations where multiple operators are within the same expression and are at the same level of operator precedence, the leftmost operators are evaluated first. The PL/SQL rules of operator precedence are the same rules that SQL uses.

## SQL Functions

Almost all of the SQL functions can be used in PL/SQL expressions. For example, this expression is considered valid in PL/SQL:



```
SUBSTR(v_lastname, 1, 1)
```

This expression uses the SQL function SUBSTR, or substring, and in this example, produces the first single character from within the variable v\_lastname.

The only SQL function that is not allowed in PL/SQL expressions is the DECODE function. The SQL DECODE function is essentially a conditional logic function, and PL/SQL has the more comprehensive alternative of the IF-THEN-END

IF statement, which we will address later. As a result of the presence of IF statements in PL/SQL, the DECODE function is not required, nor is it allowed. (Note that although DECODE cannot be used in PL/SQL expressions, it can be used in cursors, which are addressed later.)

## NULL

The presence of NULL requires special attention. Consider the following expression:

```
v_salary + v_commission
```

If the value of `v_salary` is 100, and the value of `v_commission` is NULL, then the evaluated result is NULL. After all, what is the result of 100 plus “I don’t know.”? The answer is, obviously, “I don’t know.” This is an important point. The presence of a NULL value in an arithmetic expression will produce a NULL result.

On the other hand, the presence of a NULL value in a character expression will not necessarily produce a NULL result. Here’s an example:

```
v_first_name := "James"
v_middle_name := NULL;
v_last_name := "Waters"
v_full_name := v_first_name || ' ' || v_middle_name || ' ' || v_last_name
```

The value for `v_full_name` will be “James Waters”. The NULL value for `v_middle_name` will simply be ignored in a character expression.

For those situations where the possible presence of a NULL value is unacceptable, you can use the SQL function NVL, which will replace any occurrences of NULL values with some specified value of your choosing. For example, in the case of the PL/SQL expression we looked at earlier, we can use NVL to ensure that a NULL value will not NULL out the entire result:

```
NVL(v_salary,0) + NVL(v_commission,0)
```

This expression now will always produce some sort of answer. If the commission value is NULL, then the NVL function will replace the presence of NULL with the zero value we provided as the second parameter. On the other hand, if the `v_commission` variable holds a non-NULL value, then the value will be left alone, and the equation will simply use the assigned value for `v_commission`.

## Comparison Operators and BOOLEAN Expressions

BOOLEAN expressions use comparison operators to obtain a TRUE or FALSE result. BOOLEAN expressions are used in IF statements and elsewhere, and are equivalent to the SQL WHERE clause.

## 18 OCP Developer PL/SQL Program Units Exam Guide

An example of a BOOLEAN expression is shown here:

```
v_last_name = 'Smith'
```

This expression will compare the value contained in the `v_last_name` variable with the literal value of 'Smith'. If the two values are identical, the expression will evaluate to TRUE. Otherwise, the expression evaluates to FALSE.

Comparison operators can be used to compare two arithmetic expressions. Here's an example:

```
(3+4)*5 >= (v_answer / v_rate);
```

In this example, the two expressions on either side of the comparison operator will be evaluated, and the results will be compared to determine if the expression on the left side is greater than or equal to the expression on the right side. If yes, then a value TRUE for the entire expression is the result. Otherwise, a FALSE results. The two sides of the comparison operator must represent the same general datatypes, meaning they must both be DATES, character strings, or numeric datatypes.

Furthermore, PL/SQL honors the same rules that SQL uses for the comparison for datatypes:

- **DATE** Earlier dates are *lesser* dates, and later dates are *greater* dates.
- **VARCHAR2** A is less than Z. Adam is less than Albert. Uppercase Z is less than lowercase a. The number 2 is greater than 10. If this seems strange, remember that when numbers are stored as character strings, PL/SQL won't recognize them as numbers and will treat them as the equivalent of dictionary entries.
- **Numeric datatypes, such as NUMBER and INTEGER** The number 1 is less than the number 2. The number 10 is less than the number 100.

Comparison operators are all collectively at the same level of operator precedence as each other and are considered inferior to the arithmetic operators. In other words, arithmetic operators are evaluated before comparison operators in any given expression, excluding, of course, the effect of any parentheses, which can be used to override default precedence rules.

---

Operator	Description
=	Equals
>=	Greater than or equal to
>	Greater than
<=	Less than or equal to
<	Less than
!=	Not equal
<>	Not equal
^=	Not equal
IN	Compares one value on the left with a set of values on the right. The set is enclosed in parentheses, with each value separated by commas, as in ('Smith', 'Jones', 'Grant', 'Waters') or (1999, 2000, 2001).
LIKE	<p>Activates wildcard characters. The two wildcard characters are as follows:</p> <ul style="list-style-type: none"> <li>_ An underscore represents one single unknown character.</li> <li>% A percent sign represents an unknown number of unknown characters, from zero to infinity.</li> </ul> <p>Here's an example:</p> <pre>IF ('_c%') LIKE v_lastname THEN</pre> <p>The '_c%', combined with the LIKE keyword, is interpreted to refer to any string with a second character that is a lowercase c and that's followed by any characters, from zero to an infinite number. Possible matches might include "McLean", "Acktinson", and "Ochs". If the reserved word LIKE is replaced with a standard comparison operator, such as the equals sign ("="), then the wildcard characters are no longer recognized as wildcards, but as literal values. In other words, without LIKE in the previous example, we'd be looking for someone whose last name really is "_c%", and I'm not sure what planet that person would be on, but chances are this isn't what we meant.</p>

---

**TABLE I-2.** Comparison Operators

## Logical Operators

Finally, multiple BOOLEAN expressions can be combined with logical operators. Here's an example:

```
IF ((3+4)*5 = v_answer) AND
   (v_last_name = 'Smith') THEN
```

In this example, the logical operator AND combines the two BOOLEAN expressions, which are evaluated first, to create an overall TRUE or FALSE for the entire IF condition.

The logical operators are shown in Table 1-3.

The rules of operator precedence dictate that expressions combined with AND are resolved before expressions combined with OR. Also, logical operators are evaluated after arithmetic and comparison operators.

For example, consider the following code sample:

```
DECLARE
  v_last_name  VARCHAR2(30) := 'Waters';
  v_department VARCHAR2(10) := 'President';
  v_order_total NUMBER(5,2) := 49.99;
BEGIN
  IF (v_department = 'President') OR
     (v_department = 'Marketing') AND
     (v_order_total > 79.84) THEN
    DBMS_OUTPUT.PUT_LINE('Found it');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Did not find it.');
```

END IF;

END;

/

---

Operator	Description	Operator Precedence
AND	Requires both sides of the logical operator expression to be TRUE. Otherwise, it returns a FALSE.	1
OR	Requires only one of the sides of the logical operator expression to be TRUE. If both sides are FALSE, the result is FALSE.	2

---

**TABLE 1-3.** *Logical Operators*

The result: Contrary to what you might think, it's 'Found it'. The reason is operator precedence of logical operators. Consider each expression:

- `(v_department = 'President')` evaluates to TRUE.
- `(v_department = 'Marketing')` evaluates to FALSE.
- `(v_order_total > 79.84)` evaluates to FALSE.

As a result, we have

```
TRUE OR FALSE AND FALSE
```

Operator precedence says that the AND comparison should be evaluated first. A comparison of FALSE AND FALSE produces a result of FALSE, resulting in

```
TRUE OR FALSE
```

And this, at long last, evaluates to TRUE.

If this is not what is desired, then once again parentheses can be used to override the behavior of operator precedence. For example, this is the same code sample, but with a careful placement of parentheses to alter the logic:

```
DECLARE
  v_last_name   VARCHAR2(30) := 'Waters';
  v_department  VARCHAR2(10) := 'President';
  v_order_total NUMBER(5,2)  := 49.99;
BEGIN
  IF ((v_department = 'President') OR
      (v_department = 'Marketing')) AND
      (v_order_total > 79.84) THEN
    DBMS_OUTPUT.PUT_LINE('Found it');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Did not find it.');
```

Now the result is 'Did not find it.' The default behavior of operator precedence has been overridden with parentheses.

## Assignment Statements

Assignment statements are built with the assignment symbol, `:=`, and are used to change the value of the element on the left side of the assignment symbol to be the same as the value resulting from the expression on the right side. The left side of the

## 22 OCP Developer PL/SQL Program Units Exam Guide

assignment statement is one variable. The right side of the assignment can be anything from one variable or constant, a literal value, or an expression with an evaluated result that is a single value.

The datatype on the right side of the assignment statement must match the datatype of the variable on the left. However, Oracle does perform automatic datatype conversions in many situations when it is appropriate. But proper design dictates that you as the developer should ensure, through the use of conversion functions and whatever else is appropriate, that the expression on the right produces a datatype that matches the datatype of the variable on the left.

For example, consider the following assignment statement:

```
v_lastname := 'Smith';
```

This assignment statement changes the value of the variable `v_lastname` to be equal to the character string `Smith`.

Here is an example of an assignment statement that uses an expression on the right side:

```
v_order_total := v_order_subtotal + v_tax + v_shipping;
```

Upon the execution of this statement, the value of `v_order_total` will be equal to the result of the expression on the right, which is the addition of the three variables `v_order_subtotal`, `v_tax`, and `v_shipping`.

## Conditional Statements

Conditional statements enable you to control the flow of execution through your code by testing for conditions and branching the process flow in different ways depending on the outcome. In PL/SQL, as in most languages, conditional statements are **If** statements.

Consider the following code sample:

```
DECLARE
    v_day_of_week VARCHAR2(30);
BEGIN
    v_day_of_week := TO_CHAR(SYSDATE, 'DY');
    IF (v_day_of_week IN ('SAT', 'SUN')) THEN
        DBMS_OUTPUT.PUT_LINE('The office is closed today');
    ELSE
        DBMS_OUTPUT.PUT_LINE('The office is open today.');
```

In this example, either one of two different sentences will be printed, depending on the value of the variable `v_day_of_week`. If the value is either SAT or SUN, then the sentence, The office is closed today, will print. Otherwise, the sentence, The office is open today, will print.

The syntax of a simple PL/SQL IF statement is as follows:

- The reserved word IF.
- Some expression that evaluates to a BOOLEAN value, in other words, either TRUE or FALSE. This expression can use literals, variables, constants, SQL functions, PL/SQL program units, arithmetic and logical operators, and, of course, comparison operators.
- The reserved word THEN.
- One or more PL/SQL statements, which are only executed when the IF expression evaluates to TRUE.
- The reserved words END IF and the semicolon, which mark the conclusion of this IF block.

The IF statement is considered a single statement, so the semicolon only appears at the end, which is at the end of END IF. However, the IF statement nests other complete statements within itself. You can place any number of valid PL/SQL statements after the THEN keyword, including assignment statements, SQL statements, and even other IF statements.

The previous code sample shows the IF statement in its simplest form. You may optionally extend the logic of this IF statement by including one or more occurrences of the reserved word ELSIF followed by another BOOLEAN expression and then a set of PL/SQL executable statements. Finally, you can optionally include the reserved word ELSE followed by a set of PL/SQL executable statements.

The following code sample demonstrates all of these options:

```

DECLARE
CURSOR cur_employees IS
  SELECT   E.EMPLOYEE_ID,
           E.POSITION,
           EC.SALARY
  FROM     EMPLOYEES E,
           EMP_COMPENSATION EC
  WHERE    E.EMPLOYEE_ID = EC.EMPLOYEE_ID
  AND      EC.START_DATE =
           (SELECT MAX(EC2.START_DATE)
            FROM   EMP_COMPENSATION EC2
            WHERE  EC2.EMPLOYEE_ID = E.EMPLOYEE_ID)

```

## 24 OCP Developer PL/SQL Program Units Exam Guide

```
ORDER BY POSITION;
rec_employees cur_employees%ROWTYPE;
v_salary_increase NUMBER(4,2);
BEGIN
OPEN cur_employees;
LOOP
FETCH cur_employees INTO rec_employees;
EXIT WHEN cur_employees%NOTFOUND;
IF (rec_employees.POSITION = 'Executive')
THEN
v_salary_increase := 0;
ELSIF (rec_employees.POSITION = 'Manager')
THEN
v_salary_increase := .02;
ELSIF (rec_employees.POSITION = 'Engineer')
THEN
v_salary_increase := .04;
ELSE -- for all other positions, do this
v_salary_increase := .06;
END IF;
INSERT INTO EMP_COMPENSATION
(EMP_COMPENSATION_ID,
EMPLOYEE_ID,
SALARY,
START_DATE)
VALUES
(SEQ_EMP_COMPENSATION_ID.NEXTVAL,
rec_employees.EMPLOYEE_ID,
rec_employees.SALARY * v_salary_increase,
SYSDATE);
END LOOP;
COMMIT;
END;
/
```

This code sample uses an IF statement that employs two ELSIF clauses and one single ELSE clause. In this example, if the value for `rec_employees.POSITION` is determined to be equal to the literal string `Executive`, then the first set of processing statements will be executed through to completion; in other words, the value for `v_salary_increase` will be set to zero, and the section that starts with the ELSIF that tests for the literal string `Manager` will be skipped, as will the rest of the IF block. Execution will pick up after the END IF keywords.

On the other hand, if the value for `rec_employees.POSITION` is `Manager`, then the second set of statements will be executed, and processing will jump past the `Engineer` section and pick up with the first executable statement after END IF.

Finally, if none of the specified values for `rec_employees.POSITION` are found—in other words, if it's not Executive, Manager, or Engineer—then the ELSE clause will capture control and set the value of `v_salary_increase` to `.06`.

In any IF statement, if an ELSIF clause is included, it must include a BOOLEAN expression, and the THEN keyword. The ELSIF clause is not required, but you may include as many as you wish.

If the ELSE clause is included, it cannot have a BOOLEAN expression, nor does it use a THEN keyword. There can only be one ELSE clause, although it is not required. The ELSE clause, if used, must be last, after any and all ELSIF clauses.

Remember that the first IF expression that tests for TRUE is where control will stop and process, and the rest of the ELSIF expressions that follow, whether they might be TRUE or not, will not even be evaluated. The first one that is TRUE is the first one that is processed.

The expressions in IF statements can use comparison and logical operators, variables, constants, SQL functions, and all of the other features available for use with expressions.

## Loops

Loops are structures that “nest” other PL/SQL statements inside. All statements in a loop will execute all the way through the loop, after which control will return to the top of the loop and repeat the entire set of statements again. Any variables with values that are changed as the loop executes will retain those changed values. In other words, the results of the first loop's execution is retained and built upon as the loop repeats.

Consider the following code sample:

```
DECLARE
    v_loop_counter NUMBER(3) := 1;
BEGIN
    LOOP
        DBMS_OUTPUT.PUT_LINE('Loop ' || v_loop_counter);
        v_loop_counter := v_loop_counter + 1;
        EXIT WHEN v_loop_counter >= 10;
    END LOOP;
END;
/
```

This loop will print the word Loop nine times, each of which will be followed with a number corresponding to the value of `v_loop_counter`. The keywords EXIT WHEN determine when the loop will terminate, and control will leave the loop and

be passed to the first executable statement after the END LOOP statement. The syntax is

- The reserved word LOOP
- Any number of valid PL/SQL statements
- The reserved words END LOOP, followed by a semicolon

The LOOP . . . END LOOP statement is one single statement. Just as the IF . . . END IF statement nests statements within itself, so does the LOOP . . . END LOOP statement. In fact, you can nest LOOP statements within other LOOP statements.

Also notice that the LOOP statement requires an EXIT to avoid behaving as an infinite loop. Without an EXIT statement of some kind, the loop will never terminate. It's the developer's responsibility to define the conditions upon which the loop will be eventually exited and include the EXIT statement.

The EXIT statement can be on a line by itself, as in the following code snippet:

```
IF (v_condition = TRUE)
THEN
    EXIT;
END IF;
```

The EXIT statement can also be used with a WHEN clause to define a BOOLEAN expression of some sort, as in the following:

```
EXIT WHEN v_condition = TRUE;
```

Either is acceptable, but the LOOP . . . END LOOP requires some sort of EXIT somewhere. You can even provide multiple EXIT statements if you wish. The first one encountered in the execution of the block is the one that will force control to leave the loop.

The LOOP statement has a few variations, such as the WHILE LOOP, the Numeric FOR LOOP, and the Cursor FOR LOOP. These are described in the next few sections.

### WHILE LOOP

The WHILE LOOP statement uses a conditional expression to test whether or not control should enter the loop. Furthermore, this expression is checked again at the top of every pass through the loop.

The following is an example of a WHILE LOOP:

```
DECLARE
    v_loop_limit NUMBER(3) := 0;
BEGIN
```

```

WHILE (v_loop_limit < 10)
LOOP
    v_loop_limit := v_loop_limit + 1;
    DBMS_OUTPUT.PUT_LINE('In the loop: ' || v_loop_limit);
END LOOP;
DBMS_OUTPUT.PUT_LINE('Out of the loop.');
```

The primary difference between the simple loop and the WHILE LOOP is that the WHILE LOOP could theoretically not execute the first time through the loop, although the simple loop will always enter the loop at least once. The expression that the WHILE LOOP tests can be the same sort of expression that an IF statement uses, with comparison and logical operators.

## Numeric FOR LOOP

The Numeric FOR LOOP uses a predefined numeric range to determine how many passes through the loop will be performed. The following code sample will repeat 10 times:

```

BEGIN
    FOR v_loop_limit IN 1..10
    LOOP
        DBMS_OUTPUT.PUT_LINE('In the FOR loop: ' || v_loop_limit);
    END LOOP;
END;
```

Notice that the variable `v_loop_limit` is not declared here. This is not a mistake. The Numeric FOR LOOP automatically declares and initializes the counter variable and automatically increments it by a value of one (1) each pass through the loop.

The downside is that the `v_loop_limit` variable will not be available upon the completion of the loop. The scope of this variable is the inner portion of the Numeric FOR LOOP. Once the loop concludes, you cannot reference the `v_loop_limit` variable again for any reason.

Note that the EXIT statement is not required here in order to define a logical exit point for the loop, but the PL/SQL syntax parser will accept it. If an EXIT is used, and if it executes before FOR loop completes, the Numeric FOR LOOP will exit according to what the EXIT statement requires and not complete the predefined cycle.

## Cursor FOR LOOP

To understand Cursor FOR LOOPS, you must first understand cursors, which are discussed in the next section. At the end of the next section, Cursor FOR LOOPS are addressed.

## Cursors

Contrary to popular belief, cursors are *not* Oracle programmers whose code won't work. (Well, maybe that's true some of the time.)

The PL/SQL explicit cursor element is the single most important feature of the PL/SQL language and is the entire reason for using the language. It empowers the developer to obtain record-level control over database queries, stepping any query through its returned data one row at a time and pausing for as long as required between rows to perform other processing—even to the point of interacting with the database objects and choosing to terminate the query at any moment, if the need arises.

There are two types of cursors in PL/SQL: implicit cursors and explicit cursors. First, let's look at what an implicit cursor is; then we'll discuss explicit cursors, which is the more powerful form.

## Implicit Cursors

Implicit cursors are not declared. They are DML statements, such as the SELECT statement, that are used in the processing section of a PL/SQL block.

For example, consider the following code sample:

```

DECLARE
    v_port_name VARCHAR2(80);
BEGIN
    SELECT    PORT_NAME
    INTO      v_port_name
    FROM      PORTS
    WHERE     PORT_ID = 101;
    DBMS_OUTPUT.PUT_LINE(v_port_name);
END;
```

The SELECT statement is an example of an implicit cursor. Notice the presence of the reserved word INTO. Implicit cursors built on the SELECT statement must use the INTO reserved word to define the list of variables that will capture the data from the columns of the SELECT statement. For each column selected, there must be a corresponding variable to receive the data.

Here's an example:

```

DECLARE
    v_ship_name  VARCHAR2(80);
    v_capacity   NUMBER(10);
    v_length     NUMBER(10);
BEGIN
    SELECT SHIP_NAME, CAPACITY, LENGTH
    INTO   v_ship_name, v_capacity, v_length
```

```

FROM   SHIPS
WHERE  SHIP_ID = 2;
DBMS_OUTPUT.PUT_LINE('Ship information:');
DBMS_OUTPUT.PUT_LINE('Name      : ' || v_ship_name);
DBMS_OUTPUT.PUT_LINE('Capacity: ' || v_capacity);
DBMS_OUTPUT.PUT_LINE('Length   : ' || v_length);
END;
```

The previous example shows an implicit cursor that obtains three values and SELECTs all three columns INTO the three variables. The first column's data is stored in the first variable, the second column's data is stored in the second variable, and the third column's data is stored in the third variable.

Implicit cursors are, in essence, SQL statements that are embedded in PL/SQL blocks. Embedded SQL statements can be any valid SQL statement that would work in the SQL\*Plus interface. This includes SELECT, INSERT, UPDATE, and DELETE statements. Any SQL function can be used, including DECODE, which is not allowed in PL/SQL expressions but is accepted in embedded SQL.

Implicit cursors, by definition, can only work with individual variables. The implication of this is that the SELECT statement defined in an implicit cursor must return one and only one record. Otherwise, PL/SQL will encounter an error condition, known as an *exception*, which is covered later.

The important point here is that SELECT statements written as implicit cursors must define a query that results in the returning of one single record—no more, no less—or else an exception is raised, and processing in the block will halt. This isn't necessarily an undesirable situation, as is discussed in the exception-handling section, but is something that you must consider when building your code.

The intention behind this language feature is simple: A SELECT statement written as an implicit cursor performs its query on the database in total and stores the entirety of its results into the variables you define in the INTO clause. Variables can only handle one value at a time. Therefore, the SELECT statement better not return something other than one value per variable. If you think this could happen—that is, if you think the SELECT statement could theoretically return more than one row—and if this condition is acceptable, then don't use an implicit cursor; declare an explicit cursor instead. On the other hand, if the presence of more than one row is a bad sign, then use an implicit cursor, and the database will force an exception to be raised when this bad sign occurs.

For example, if your SELECT statement is querying for records based on a primary key, then it better return one row or else the primary key constraint isn't performing its UNIQUE constraint correctly in the database. Perhaps someone has disabled the constraint and bad data has entered the database. This would be bad, and an implicit cursor that queries on a primary key but that returns multiple rows is definitely a problem you want to be notified about. So, use an implicit cursor and you'll get the exception raised if and when the database becomes problematic.

On the other hand, there are many situations where it's OK to return some number of records from the database other than one single record. For these situations, explicit cursors are the choice.

## Explicit Cursors

Explicit cursors are declared in the declaration section and are used in the processing section. They are called explicit cursors largely because you have to name them when you declare them; thus, they are “explicitly” referenced.

When you work with explicit cursors, you need to work with four statements, with the following keywords:

- **CURSOR** The keyword **CURSOR** is used to declare an explicit cursor in the declaration section. To declare a **CURSOR**, you use the keyword **CURSOR**, followed by a name you make up, and then the keyword **IS**, followed by a complete valid **SELECT** statement. **SELECT** statements used in a cursor can include anything that would execute in the database on its own. In other words, you can use joins, functions, subqueries, and so on.
- **OPEN** The keyword **OPEN** is used in the processing section to parse the **SELECT** statement that is declared in the **CURSOR** statement and prepare it for execution.
- **FETCH . . . INTO** These reserved words are used to obtain one and only one of the rows that the explicit cursor's **SELECT** statement returns. The **FETCH** statement can be invoked multiple times to return each of the **SELECT** statement's rows, one by one. However, the **FETCH** statement does not necessarily have to be used until all of the rows are returned. It can be used as often as is desired and stop short of returning all of the rows. The sequence in which the rows are returned is defined by the **SELECT** statement's **ORDER BY** clause. The **FETCH** syntax requires a reference to the name of the cursor as declared in the **CURSOR** declaration, followed by the reserved word **INTO** as well as by a list of variables with numbers that must match the number of columns selected in the declared **CURSOR** and with datatypes that must match as well.
- **CLOSE** This keyword is used to terminate the use of the explicit cursor. The PL/SQL parser will not issue a compilation error if the **CLOSE** statement is left out. But use of the **CLOSE** keyword is not only considered good design because it frees up memory allocation for other uses, it also empowers your PL/SQL block to re-open the cursor and begin all over again, should that be required by your code. However, if you leave out the **CLOSE** keyword, the Oracle processes will eventually figure it out and free up memory automatically.

Consider the following code listing:

```

DECLARE
  v_today DATE;
  -- declare the explicit cursor
  CURSOR cur_cruises IS
    SELECT C.CAPTAIN_ID,
           E.LAST_NAME || ', ' || E.FIRST_NAME FULL_NAME
    FROM   CRUISES C,
           EMPLOYEES E
    WHERE  C.CAPTAIN_ID = E.EMPLOYEE_ID
           AND START_DATE >= v_today
           AND END_DATE   <= v_today;
  v_captain_id NUMBER(3);
  v_full_name  VARCHAR2(40);
BEGIN
  -- use an implicit cursor
  SELECT   TRUNC(SYSDATE)
  INTO     v_today
  FROM     DUAL;
  DBMS_OUTPUT.PUT_LINE(
    'Captains currently at sea include:');
  OPEN cur_cruises; -- open the explicit cursor
  LOOP
    -- fetch the explicit cursor
    FETCH cur_cruises INTO v_captain_id, v_full_name;
    EXIT WHEN cur_cruises%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Captain ID: ' || v_captain_id);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || v_full_name);
  END LOOP;
  CLOSE cur_cruises; -- close the explicit cursor
END;
```

The previous code listing is an example of both an explicit cursor, declared with the name `cur_cruises`, and an implicit cursor, which immediately follows the `BEGIN` statement that starts the processing section. The explicit cursor `cur_cruises` is a named `SELECT` statement. Explicit cursors can be defined with any valid `SELECT` statement that would execute on its own. For example, if you can execute the `SELECT` statement in a SQL\*Plus window, you can use that `SELECT` statement to build an explicit cursor.

However, explicit cursors can also include variables that have already been defined from within PL/SQL. In the previous example, notice that the variable `v_today` is declared prior to the declaration of the `cur_cruises` explicit cursor. This means that the `cur_cruises` can include the variable in its own definition, even though the value for the `v_today` variable is not yet defined. In order for the PL/SQL

parser to function correctly, the variable must be declared before the cursor. But it does not need to be initialized at declaration. Instead, the moment the OPEN statement is issued on the cursor is when the variable's definition will be identified and the cursor's definition will be based on the value of the variable at the time of the OPEN statement.

## Cursor Attributes

Cursors have four values defined by the system that you can optionally choose to reference in your code. These are the four attributes:

- **NOTFOUND** A BOOLEAN value that is TRUE if the most recent FETCH statement issued for the associated cursor did not return a row.
- **FOUND** A BOOLEAN value that is TRUE if the most recent FETCH statement for the associated cursor returned a record.
- **ISOPEN** A BOOLEAN value that is TRUE if the associated cursor is currently open.
- **ROWCOUNT** A numeric value that indicates the total number of rows affected so far by the associated cursor.

The syntax for using cursor attributes is to indicate the cursor's name, followed by a percent sign (%) and the attribute. For example, a typical combination of statements is shown here:

```

FETCH cur_employees INTO rec_employees;
IF (cur_employees%NOTFOUND) THEN
    ...

```

The cursor `cur_employees`, once fetched, either found a row or it didn't. The story is in the value of `cur_employees%NOTFOUND`, which is TRUE if no row was found.

This works for explicit cursors, where you have a declared name of a cursor. For implicit cursors, there is no name, so instead we use the generic name `SQL`, which refers to the most recently executed implicit cursor in the code.

Here is an example of an implicit cursor attribute:

```

DECLARE
    UPDATE EMPLOYEES
        SET SALARY = SALARY * 1.05;
    DBMS_OUTPUT.PUT_LINE('Number of raises: ' ||
                          SQL%ROWCOUNT);
END;
/

```

The text will print out the number of rows affected by the UPDATE statement, which is an implicit cursor.

## Loops Revisited: The Cursor FOR Loop

Now that we know what cursors are, we can revisit the Cursor FOR LOOP. The Cursor FOR LOOP is similar to the Numeric FOR LOOP, but uses a PL/SQL cursor instead. The cursor is considered an anonymous explicit cursor. It's anonymous because we don't know its name, but it is explicit because it will fetch automatically through the set of records that the query returns.

The following code demonstrates a simple Cursor FOR LOOP:

```
BEGIN
  FOR rec_employees IN (SELECT EMPLOYEE_ID FROM EMPLOYEES)
  LOOP
    DBMS_OUTPUT.PUT_LINE('In the FOR loop: ' || rec_employees.EMPLOYEE_ID);
  END LOOP;
END;
/\
```

Notice that the FOR LOOP counter variable is `rec_employees`, which in the Cursor FOR LOOP is a `%ROWTYPE` variable. The presence of the SELECT statement after the IN keyword is what makes this a Cursor FOR LOOP.

The Cursor FOR LOOP automatically does the following:

- Declares the `%ROWTYPE` variable `rec_employees`. (The `%ROWTYPE` is discussed in the next section.)
- Declares an anonymous cursor for the SELECT statement and opens the cursor upon the first pass through the loop.
- Performs a fetch from the anonymous cursor for each pass through the loop and repeats the loop until the condition of `SQL%NOTFOUND` exists with the anonymous cursor.
- Upon exiting the loop, a `CLOSE` statement is issued on the cursor.

Just as with the Numeric FOR LOOP, the Cursor FOR LOOP does not require an EXIT statement, but it is accepted.

The Cursor FOR LOOP automatically declares a lot of elements that could be declared individually. However, when you declare a cursor explicitly, you have much more control. Just as with the Numeric FOR LOOP, the Cursor FOR LOOP's loop counter variable, which is `rec_employees` in the previous example, is local to the loop, so it cannot be referenced by any code outside the loop.

## Advanced Datatype Declaration

As you know, PL/SQL code is generally used to interact with the database, and the database already has tables and columns with declared datatypes. Furthermore, most PL/SQL variables are used to hold data obtained from the database through SELECT statements and related commands. As a result, most of the variables you will declare should have the same datatype as the columns in the tables of the database that you will be working with.

For example, if you are obtaining data from a table called CRUISES, and you are intending to SELECT data from that table, including the START\_DATE, then you'll probably want to have a variable that is already declared with the same datatype as START\_DATE to hold the data that you'll be fetching from the database.

The PL/SQL language features some special capabilities for declaring variables based on the datatypes of existing table columns from the database. The %TYPE declaration is used to declare a single variable based on a database object, and the %ROWTYPE can be used to declare multiple variables within a single declarative statement.

### %TYPE

The %TYPE declaration lets you declare a variable at design time with a datatype that isn't decided until execution time, and the datatype is automatically extracted from the Oracle database data dictionary from the column of a table that you specify.

For example, if you want to declare a variable with the purpose of receiving data from the CRUISES table's CRUISE\_NAME column, you want your variable to have the same datatype as that particular column as it is declared in the database. You could look it up and manually match them up, but the %TYPE declaration will do that for you. Better still, if the column's datatype is modified in the future, your %TYPE declared variable will ensure that future executions of your PL/SQL code will use the correct datatype. However, this does run the risk that a significant datatype change, such as a change from a VARCHAR2 to a NUMBER, may cause other problems in the processing section, but the risk is considered worth the benefit that the %TYPE declaration provides. In other words, %TYPE is considered good design.

The format for the %TYPE declaration is as follows:

```
variable_name TABLE.COLUMN%TYPE;
```

Following the example of a variable that will receive data from the CRUISES table's CRUISE\_NAME column, the code would look like this:

```
v_cruise_name CRUISES.CRUISE_NAME%TYPE;
```

The declaration for the `v_cruise_name` variable will be resolved at execution time by the PL/SQL block, which will inspect the data dictionary and locate the datatype for the `CRUISE` table's `CRUISE_NAME` column, and that will be the datatype for the `v_cruise_name` variable.

There is no requirement that you, as the developer, must associate the `%TYPE` declaration with the table and column that you later will fetch data from for the declared variable. PL/SQL will not bother to follow up and ensure that you have in fact tied these two elements together, the table's column and the variable, in your processing section. Technically, any table and column can be used, regardless of what you intend to do with the variable, but naturally the table and column selection is probably going to be relevant to your use of the variable in the processing section.


## **%ROWTYPE**

The `%ROWTYPE` declaration builds on the concept of the `%TYPE` declaration. A single declaration with `%ROWTYPE` can actually result in more than one variable.

The `%ROWTYPE` declaration ties the variable declaration to the declaration of a `CURSOR` in a PL/SQL block. The resulting declaration will produce, in a single statement, one variable for each of the columns selected in the `CURSOR` declaration. The `%ROWTYPE` declaration can therefore conceivably declare many variables in a single statement.

The resulting variables have a slightly unusual naming convention. Their names will be a combination of the `%ROWTYPE` variable name, followed by a period, followed by the name of the column as defined in the `CURSOR` that is referenced by the `%ROWTYPE` declaration.

Consider the following code sample:



```

DECLARE
    CURSOR cur_employees IS
        SELECT EMPLOYEE_ID,
               LAST_NAME || ', ' || FIRST_NAME FULL_NAME,
               POSITION
        FROM   EMPLOYEES
        ORDER BY LAST_NAME;
    rec_employees cur_employees%ROWTYPE;
BEGIN
    OPEN cur_employees;
    LOOP
        FETCH cur_employees INTO rec_employees;
        EXIT WHEN cur_employees%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(rec_employees.EMPLOYEE_ID);
        DBMS_OUTPUT.PUT_LINE(rec_employees.FULL_NAME);
    
```

## 36 OCP Developer PL/SQL Program Units Exam Guide

```
        DBMS_OUTPUT.PUT_LINE(rec_employees.POSITION);
    END LOOP;
    CLOSE cur_employees;
END;
```

The variable `rec_employees` is declared as a `%ROWTYPE` variable and is based on the cursor `cur_employees`. This means that the resulting variables will be as follows:

- **rec\_employees.EMPLOYEE\_ID** The datatype is inherited from the `EMPLOYEES` table's `EMPLOYEE_ID` datatype.
- **rec\_employees.FULL\_NAME** This represents the column alias name of the concatenated result of the `LAST_NAME` and the `FIRST_NAME` columns, along with the string literal `' , '`. The datatype of `rec_employees.FULL_NAME` is the datatype of the expression, which is determined by combining the datatypes of the elements within the expression. For example, if `LAST_NAME` is `VARCHAR2(30)` and `FIRST_NAME` is `VARCHAR2(20)`, then `FULL_NAME` will be `VARCHAR2(52)`, including the `' , '` string literal. Since these columns are combined using SQL functions in the original `SELECT` statement, then a column alias must be used for the `rec_employees %ROWTYPE` variable to be named.
- **rec\_employees.POSITION** The datatype is inherited from the `EMPLOYEES` table's `POSITION` datatype.

Also, note how the `%ROWTYPE` variable works in a `FETCH` statement. Although the use of standard variables requires a `FETCH` statement that specifies each column in the explicit cursor, and each variable that is receiving the fetched data, the use of a `%ROWTYPE` variable greatly simplifies the `FETCH` statement:

```
 FETCH cur_employees INTO rec_employees;
```

This format automatically fetches all of the columns defined in the cursor into all of the appropriate variables in the `%ROWTYPE` variable.

Once the `FETCH` statement has executed, the `%ROWTYPE` variables can be used. Here's an example:

```
 DBMS_OUTPUT.PUT_LINE(rec_employees.EMPLOYEE_ID);
```

In short, `%ROWTYPE` variables are the ideal choice for creating variables to be used with an explicit cursor.

## Exception-Handling Section

PL/SQL programs are primarily designed to interact with the database. The database is a large and potentially complex environment and its condition at any given moment is beyond the control of the PL/SQL parser. In other words, while the PL/SQL parser can analyze the syntax of your PL/SQL program when you compile it, there is no guarantee that the database with which it interacts will even be up and running later when the PL/SQL program executes or that the database objects that are named within your PL/SQL program still exist, let alone reflect the structure that your program requires.

This poses a problem. Although you can compile your PL/SQL program today and produce an accurately written program unit, you have no assurance that the database elements that your code interacts with, such as the relationship of data in one table to another, are going to remain unchanged in the future when your PL/SQL program is executed.

In order to handle these unpredictable situations, PL/SQL provides a mechanism known as exceptions. Exceptions are not errors, but instead are show-stopping problems based on circumstances beyond the control of the PL/SQL parser that result in your code being unable to continue.

When an exception occurs in the course of your PL/SQL program unit's execution, the exception is said to be raised. After the exception is raised, execution within the particular PL/SQL block terminates. Any open cursors are automatically closed, and any loops are exited. Execution control leaves the processing section of the block.

Furthermore, when execution exits the block due to a raised exception, control will pass into the exception-handling section of the block, if such a section has been included with the block. The exception-handling section is an optional section in which you can declare *exception handlers*. An exception handler is a section of PL/SQL code that is included with the PL/SQL block, and that will only execute if and when the associated exception is raised within the processing section.

If your PL/SQL program is executed and an exception is raised, and if you've included an exception-handling section, and if that exception-handling section includes an exception handler that is appropriate for the particular raised exception, then the code you've provided for your exception will be executed, and the PL/SQL block will terminate with a message indicating that the PL/SQL program was successfully completed.

On the other hand, if the same situation arises, but you have not provided an exception handler appropriate for the exception, then the PL/SQL program will still terminate, but with an error message indicating that an unhandled exception was raised in your PL/SQL program.

Note that the exception-handling section is not required. Exceptions may be raised whether you have provided an exception-handling section or not.

There are two general categories of exceptions: system-defined exceptions and user-defined exceptions.

## System-Defined Exceptions

System-defined exceptions are exceptions that you, as a developer, do not declare, but that are raised automatically by the system. There are several predefined exceptions:

- **CURSOR\_ALREADY\_OPEN** This is raised when your code attempts to issue an OPEN statement on a cursor that is already opened.
- **DUP\_VAL\_ON\_INDEX** You've attempted to INSERT or UPDATE a database table that includes a UNIQUE constraint on at least one column, and the UNIQUE constraint has been violated in the INSERT or UPDATE issued from your code.
- **INVALID\_CURSOR** This results from an attempt to fetch or close a cursor that hasn't been opened.
- **INVALID\_NUMBER** An automatic datatype conversion was attempted on a literal that is being passed to some mechanism in PL/SQL (such as a parameter) that's expecting a numeric value, and Oracle attempted the conversion but failed.
- **LOGIN\_DENIED** An attempt to log in to the database from your program was rejected by the database, probably due to an invalid user ID or password.
- **NO\_DATA\_FOUND** Generally, this exception indicates that you've attempted a SELECT . . . INTO command on a table that returned no rows, either as a result of the logic in the WHERE clause, or because the table truly has no rows. This exception can also result from an attempt to read past the end of file marker using the UTL\_FILE package or from an attempt to refer to a row in a PL/SQL table that hasn't been initialized.
- **NOT\_LOGGED\_ON** Your program has attempted some sort of database access, such as a SELECT statement, at a time that it is not logged on to the database.
- **PROGRAM\_ERROR** This exception generally is serious and is accompanied with a Contact Oracle Support message.
- **STORAGE\_ERROR** Generally, this indicates a memory problem.
- **TIMEOUT\_ON\_RESOURCE** While attempting to access some resource in the database, a timeout was experienced.

- **TOO\_MANY\_ROWS** This results from a `SELECT . . . INTO` statement that returns more than one row, as determined by the `WHERE` clause.
- **TRANSACTION\_BACKED\_OUT** A remote transaction is rolled back for some reason beyond the control of your program.
- **VALUE\_ERROR** This results from a variety of problems, generally related to attempts to convert values from one variable to another, such as truncation, constraint violations, or something related.
- **ZERO\_DIVIDE** Your code attempted to divide by zero. This can result from equations that divide by some variable with a value you can't necessarily predict, such as a parameter or some value fetched from a database object, and that turns out to be zero in some particular execution of your code.

Some of the more commonly encountered system-defined exceptions are `NO_DATA_FOUND` and `TOO_MANY_ROWS`, both of which result from the use of the `SELECT . . . INTO` statement, which must always return a single row from the database in order to populate the single value variable (or variables) defined in the `INTO` clause. However, an easy way to avoid receiving the `NO_DATA_FOUND` or the `TOO_MANY_ROWS` system-defined exceptions is to simply use explicit cursors all of the time. Explicit cursors cannot experience either of these exceptions.

## User-Defined Exceptions

User-defined exceptions are exceptions that you, the developer, declare in the declaration section and then explicitly raise with the `RAISE` statement somewhere in the processing section.

Consider the following code sample:

```

DECLARE
  CURSOR cur_work_schedule IS
    SELECT WORK_SCHEDULE_ID, START_DATE, END_DATE
    FROM   WORK_SCHEDULE;
  rec_work_schedule cur_work_schedule%ROWTYPE;
  ex_dates_incorrect EXCEPTION;
BEGIN
  OPEN cur_work_schedule;
  LOOP
    FETCH cur_work_schedule INTO rec_work_schedule;
    EXIT WHEN cur_work_schedule%NOTFOUND;
    IF rec_work_schedule.START_DATE >
       rec_work_schedule.END_DATE
    THEN
      RAISE ex_dates_incorrect;
    
```

```

        END IF;
    END LOOP;
    CLOSE cur_work_schedule;
EXCEPTION
    WHEN ex_dates_incorrect THEN
        INSERT INTO ERRORS
            (ERROR_ID, MESSAGE)
        VALUES
            (SEQ_ERROR_ID.NEXTVAL,
             rec_work_schedule.WORK_SCHEDULE_ID ||
             'START_DATE follows END_DATE');
END;
```

The user-defined exception `ex_dates_incorrect` is raised in the processing section explicitly with the `RAISE` command. In this example, the exception is only raised when the `START_DATE` value is determined to be greater than the `END_DATE`. When this condition is found to be true with a particular record, the processing section will immediately terminate, and control will pass to the exception-handling section, where the exception is handled with the `WHEN ex_dates_incorrect` exception handler. Information is then recorded in the `ERRORS` table, along with the primary key of the bad record, for future reference. The block will terminate with a PL/SQL procedure successfully completed message.

If there is no exception-handling section for the `ex_dates_incorrect` exception, then the generic `WHEN OTHERS` exception, if it exists, will execute, and the “successfully completed” message will display upon the completed execution of the PL/SQL code. Otherwise, the block will exit with an error message indicating that an unhandled exception was raised.

## Working with Blocks

The statement that builds a PL/SQL block must include, as a minimum, a `BEGIN` and an `END`, as the following demonstrates:

```

BEGIN
    ... processing statements go here ...
END;
```

If the block includes a declaration section, then the `DECLARE` statement is required:

```

DECLARE
    ... declarations go here ...
BEGIN
    ... processing statements go here ...
END;
```

If the block includes an exception-handling section, then the `EXCEPTION` keyword is used:

```
BEGIN
    ... processing statements go here ...
EXCEPTION
    ... exception handling statements go here ...
END;
```

Finally, a block that includes all sections uses all four keywords:

```
DECLARE
    ... declarations go here ...
BEGIN
    ... processing statements go here ...
EXCEPTION
    ... exception handling statements go here ...
END;
```

## Nested Blocks

Blocks can be nested within other blocks. Consider that the `DECLARE . . . BEGIN . . . EXCEPTION . . . END` statement is a single statement, and that the `BEGIN . . . END` that delineates the processing section can contain any other valid PL/SQL statement. It stands to reason therefore that one of those processing section statements can be another `DECLARE . . . BEGIN . . . EXCEPTION . . . END` statement. Furthermore, the nested block can contain another nested block and so on. Here's an example:

```
DECLARE
    v_highest_salary PAY_HISTORY.SALARY%TYPE;
BEGIN
    -- get the current highest salary
    SELECT MAX(SALARY)
    INTO   v_highest_salary
    FROM   PAY_HISTORY
    WHERE  END_DATE IS NULL;
    DECLARE
        c_highest_salary CONSTANT
            PAY_HISTORY.SALARY%TYPE := v_highest_salary;
    CURSOR cur_pay_history IS
        SELECT PAY_HISTORY_ID, SALARY
        FROM   PAY_HISTORY
        WHERE  END_DATE IS NULL;
        rec_pay_history cur_pay_history%ROWTYPE;
    BEGIN
        OPEN cur_pay_history;
```

## 42 OCP Developer PL/SQL Program Units Exam Guide

```
LOOP
    FETCH cur_pay_history INTO rec_pay_history;
    EXIT WHEN cur_pay_history%NOTFOUND;
    IF (rec_pay_history.SALARY/c_highest_salary < .10)
    THEN
        -- Give a 6 percent pay raise to the lowest
        -- compensated employees
        UPDATE PAY_HISTORY
            SET SALARY = SALARY * 1.06
            WHERE PAY_HISTORY_ID = rec_pay_history.PAY_HISTORY_ID;
    END IF;
END LOOP;
CLOSE cur_pay_history;
COMMIT;
END;
END;
/
```

The previous block is a block within a nested block. The outer block declares a variable and then uses that variable to fetch a value from the database. The nested block, also known as an inner block, uses the value that's been fetched from the database to declare a constant `c_highest_salary`, which is then used in the inner block.

Nested blocks can declare their own elements, including variables, constants, exceptions, and cursors. Nested blocks must subscribe to all of the rules required of any block. The `DECLARE` is optional, and the optional `EXCEPTION` reserved word can be used to handle any exceptions that are raised within the block.

However, once nested blocks are brought into an outer block, there are some issues of variable scope and exception scope that must be considered, which are discussed in the next few sections.

### Scope: Declared Elements

A declared element, such as a variable, constant, user-defined exception, or cursor, is recognized within the block that declares it. The declared element is said to be "local" to its own block.

But when nested blocks are involved, the elements declared in one block may or may not be available for another block, depending on the scope of the variable.

For example, consider the following code sample:


```
DECLARE -- Outer block
    v_employee_id NUMBER(3) := 101;
    v_last_name    VARCHAR2(30) := 'Smith';
BEGIN
    DECLARE -- Inner block
```

```

v_last_name VARCHAR2(30) := 'Jones';
v_hire_date DATE;
BEGIN
  DBMS_OUTPUT.PUT_LINE('Employee ID = ' || v_employee_id);
  DBMS_OUTPUT.PUT_LINE('Last Name (Inner) = ' || v_last_name);
END; -- End of inner block
  DBMS_OUTPUT.PUT_LINE('Last Name (Outer)= ' || v_last_name);
END; -- End of outer block
/

```

The output from this block will be as follows:



```

Employee ID = 101
Last Name (Inner) = Jones
Last Name (Outer) = Smith

```

Let's consider the different variables declared in this example.


The variable `v_employee_id` is declared only in the outer block. It's considered local to the outer block and global to the inner block. The variable can be referenced in either block. Any changes made to the variable in one block will be understood by the other one.

The variable `v_last_name` is declared in the outer block, and another is declared in the inner block. This results in two different variables. The `v_last_name` that is declared in the outer block behaves just like the `v_employee_id` variable, but there's a twist: The inner block, which would otherwise recognize the outer block's `v_last_name` variable, has its own variable of the same name, which overrides the attempt in this example to reference the variable of the same name in the outer block. The result: Any reference in the inner block to `v_last_name` is directed to the local block's variable of that name.

The variable `v_hire_date` is declared in the inner block. Therefore, it can be used in the inner block where `v_hire_date` is considered a local variable, but it cannot be referenced in this example in the outer block.

The general rule is that variables that are declared in a block are local to that block and global to any nested blocks. Local variables that have the same name as global variables take precedence in any variable references.

Block labels can be used to resolve conflicts between outer block and inner block elements. Here's an example:



```

<<outer_block>>
DECLARE
  v_lastname VARCHAR2(30) := 'Nader';
BEGIN
  <<inner_block>>
  DECLARE

```

## 44 OCP Developer PL/SQL Program Units Exam Guide


```
    v_lastname VARCHAR2(30) := 'Keyes';
BEGIN
    DBMS_OUTPUT.PUT_LINE(outer_block.v_lastname);
END;
END;
```

In this example, each block is given a label. The `DBMS_OUTPUT.PUT_LINE` statement uses the label prefix to directly reference the outer block variable, bypassing the inner block variable, and resulting in the string “Nader” printing out on the screen.

### Exception Scope and Exception Propagation

Remember that there are two kinds of exceptions: system-defined and user-defined. System-defined exceptions are raised automatically by the system. Their scope is not an issue. User-defined exceptions are exceptions that you define in the declaration section. When nested blocks are involved, the same rules of scope that apply to variables also apply to exceptions.

Consider the following code sample:




```
DECLARE
    ex_no_cabins EXCEPTION;
    v_cabins      BOOLEAN := FALSE;
    v_cruises     BOOLEAN := TRUE;
BEGIN
    DECLARE
        ex_no_cabins EXCEPTION;
        ex_no_cruises EXCEPTION;
    BEGIN
        IF NOT (v_cabins)
        THEN
            RAISE ex_no_cabins;
        END IF;
        IF NOT (v_cruises)
        THEN
            RAISE ex_no_cruises;
        END IF;
    EXCEPTION
        WHEN ex_no_cruises THEN
            DBMS_OUTPUT.PUT_LINE('No cruises!');
    END;
EXCEPTION
    WHEN ex_no_cabins THEN
        DBMS_OUTPUT.PUT_LINE('No cabins!');
END;
/
```

Notice that there are three exceptions declared in this code sample:

- **ex\_no\_cabins, defined in the outer block** This exception is local to the outer block and global to the inner block.
- **ex\_no\_cabins, defined in the inner block** This exception is local to the inner block. It has the same name as the outer block's exception, but PL/SQL recognizes it as a different exception.
- **ex\_no\_cruises, defined in the inner block** This exception is local to the inner block and is unknown to the outer block, as are all inner-block-declared elements.

The previous code sample will produce the following output:

```

 DECLARE
*
ORA-06510: PL/SQL: unhandled user-defined exception
ORA-06512: at line 12

```

The reason: The inner block will raise the exception `ex_no_cabins`. But this is the inner block's exception that is being raised.

At the moment that this exception is raised, control will automatically exit the inner block's processing section and be sent to the exception-handling section of the inner block. But there is no appropriate exception handler; there is neither a specific `WHEN ex_no_cabins` handler nor is there the generic `WHEN OTHERS` exception handler.

Note that had there been an appropriate exception handler in the inner block, then the inner block's exception handler would execute, and control would be passed to the next executable statement after the inner block within the outer block. However, there is no inner block exception handler. Therefore, control is passed to the outer block's exception handler, as PL/SQL looks for an exception-handling section.

In the outer block's exception-handling section, there is a `WHEN ex_no_cabins` handler, but this `ex_no_cabins` that is referenced is the outer block's `ex_no_cabins`, which is seen by PL/SQL as a different declared element from the inner block's `ex_no_cabins` exception. Therefore, PL/SQL does not recognize this exception handler as being relevant to the inner block's `ex_no_cabins` exception that was raised. As a result, the message that an unhandled exception was raised is the message displayed to the user.

Inner-block exception handlers can be used to handle exceptions that would otherwise terminate all processing. The inner block can handle the exception and move on if you set it up correctly.

Consider the following example:

```

DECLARE
CURSOR cur_cruises IS
    SELECT    CRUISE_ID, CRUISE_NAME, START_DATE, END_DATE, STATUS
    FROM      CRUISES
    ORDER BY  START_DATE;
rec_cruises cur_cruises%ROWTYPE;
BEGIN
    OPEN cur_cruises;
    LOOP
        FETCH cur_cruises INTO rec_cruises;
        EXIT WHEN cur_cruises%NOTFOUND;
        --
        IF (TRUNC(rec_cruises.START_DATE) <= TRUNC(SYSDATE))
        AND (rec_cruises.STATUS          = 'Scheduled')
        THEN
            BEGIN -- the start of the nested block
                UPDATE CRUISES
                    SET STATUS = 'Disembarked'
                WHERE CRUISE_ID = rec_cruises.CRUISE_ID;
            EXCEPTION
                WHEN OTHERS THEN
                    INSERT INTO ERRORS
                        (ERROR_ID, MESSAGE)
                    VALUES
                        (SEQ_ERROR_ID.NEXTVAL,
                         'Procedure CRUISE_LIST: UPDATE statement.');
            END; -- the end of the nested block
        END IF;
    END LOOP;
    CLOSE cur_cruises;
EXCEPTION
    WHEN OTHERS THEN
        INSERT INTO ERRORS
            (ERROR_ID, MESSAGE)
        VALUES
            (SEQ_ERROR_ID.NEXTVAL,
             'Procedure CRUISE_LIST experienced an error.');
```

The previous code sample shows an outer block and an inner block. The inner block is nested inside a loop of the outer block. The inner block also is set up to handle any exception, using the WHEN OTHERS exception handler.

Notice the placement of this block: It is within a loop of the outer block. This loop is structured to fetch one row at a time from a declared cursor.

Normally, if an exception were to be raised at any time during these fetches, the outer block would terminate the loop and close the cursor, regardless of where it might be in the series of records it is returning from the database.

However, by placing a block around the code that might raise the exception, the exception could now be handled within the loop. Once the exception is handled in the inner block, the inner block completes and passes control to the next executable statement in the outer block, which is still within the loop, enabling the fetching of rows from the database to continue.

This design choice of using inner blocks is just one alternative available to you, as the developer. Whether you choose to use it or not is not an issue of good design or proper structure, but rather a business rule decision driven by whether or not an exception raised during the fetch could truly be a show-stopping problem or merely a flag that one record is bad, but not so bad that the processing of other records should be terminated. The choice is up to you.

## An Introduction to Program Units

PL/SQL blocks are also known as program units. Program units can be unnamed—in other words, anonymous—or they can be named. This chapter, up to this point, has dealt with anonymous PL/SQL blocks. However, the rest of this book deals with named program units. This section discusses the difference between anonymous blocks and named program units.

### Anonymous Blocks

Anonymous blocks are blocks that are not named. Any PL/SQL block without a name is considered an anonymous block. Anonymous blocks are often used in files that are submitted directly to the database. For example, the SQL\*Plus command-line interface easily accepts anonymous PL/SQL blocks for execution. Also, anonymous blocks can be included within a larger PL/SQL block. The larger block may be anonymous or named.

### Named Program Units

The alternative to anonymous blocks is named program units. Named program units include procedures, functions, packages, and database triggers. The 1Z0-101 Test 2 exam focuses on named program units.

The name given to a PL/SQL program unit must follow the standard rules that Oracle requires for naming any database object. Names must be 30 characters or less in length and can include any letter or number, as well as the underscore (`_`),

pound sign (#), and dollar sign (\$). Names cannot start with a number and must not include any spaces.

Named PL/SQL program units that are stored in the database are automatically logged and tracked by Oracle's Data Dictionary. The Data Dictionary will store everything from the actual source code to information about the named program unit (that is, metadata) that will be discussed later.

It's important to realize that although the PL/SQL language is case insensitive, the Oracle Data Dictionary stores everything about the PL/SQL program unit, such as the name, in uppercase letters. Note that the source code itself is not converted to uppercase letters; it is stored exactly as it is presented to the database. But the name of the program unit and other information about it are stored in uppercase letters. Therefore, although you could create a program unit with a name such as `CreateShipment`, as you might do in Java, it will end up being stored in the database as `CREATESHIPMENT`. As a result, the general practice is to use underscore characters to separate words, as in `CREATE_SHIPMENT`, so that the uppercase conversion will not produce a confusing name.

The categories of named program units include the following:

- Procedure
- Function
- Package
- Database trigger

Each of these is introduced in the following sections and is discussed in much more detail in the chapters that follow.

### Procedure

A PL/SQL procedure is a named program unit that can be called on directly by its name. It can optionally receive values in the form of IN parameters and can even return values back to the calling source in the form of OUT parameters. Procedures can be stored on either the client or the server. They can be called by name from within other PL/SQL program units. Procedures that are stored in the database can be called from anywhere on the network where PL/SQL program units can execute.

### Functions

A PL/SQL function is a named program unit that returns a single value. Functions can take parameters, but only for incoming values. The only value a function returns, and it must always return a single value, is sent back in the syntax of the function call itself. As a result, functions cannot be called in the same way that procedures are called and vice versa. Functions, for example, can be called from

within a SQL statement, such as the column specification of a SELECT statement, whereas a procedure cannot.

The choice therefore of whether to create a PL/SQL program unit as a procedure or a function is not so much a choice based on what you, as the developer, want to accomplish within the program unit, but rather a decision based on how you want the program unit to be invoked, or called upon, and therefore how the data may or may not be returned to the calling source. This and all other issues of procedures and functions are addressed in detail in upcoming chapters.

## Package

A PL/SQL package is a combination of procedures and/or functions. Packages may optionally include other declared elements, such as PL/SQL constants, or exceptions.

The reasons for combining procedures and functions are many and include the following:

- The logical association of a particular set of procedures and functions that support a particular application
- The performance benefits of loading multiple program units simultaneously rather than individually
- The security benefits uniquely associated with packages that enable the developer to selectively grant execute privileges on program units without necessarily publishing the actual source code

## Database Trigger

A PL/SQL database trigger is a program unit that is not executed explicitly by the developer, but instead is associated with certain predetermined events that, if and when those events occur in the database, fire the program unit automatically. Database triggers are generally associated with DML functions on a given table. For example, you can create a database trigger that will always execute when anyone performs an UPDATE or DELETE on the CRUISES table.

The execution of the database trigger is unknown to the user, or process, that triggers its execution. Database triggers can invoke other PL/SQL program units, such as procedures and functions.

## Chapter Summary

This chapter is an overview of the PL/SQL language. PL/SQL programs are structured in blocks. Each block can have a declaration section, a processing section, and an exception-handling section. The only required section is the processing section. The keywords DECLARE . . . BEGIN . . . EXCEPTION . . . END form a complete block.

In the declaration section, you can declare variables, constants, cursors, and exceptions. Variables and constants can be assigned the same datatypes used in SQL, plus `BOOLEAN`. Variables may be initialized when declared; constants must be initialized when declared. Variables and constants can make use of the dynamic datatype declaratives `%TYPE` and `%ROWTYPE`. `%TYPE` bases the definition of a variable on a named database table and column, while `%ROWTYPE` is associated with an explicit cursor and defines one variable for each column selected in that explicit cursor.

In the processing section, you can use assignment statements, conditional logic statements, and loops to work with the declared elements of your block. You can open a cursor, so that you can then fetch as many records as you wish from it, and close the cursor when you are done with it. Loops can be simple loops, which will only terminate if and when an `EXIT` statement inside the loop is invoked. Numeric `FOR` `LOOP`s will pass through a predetermined set of times, based on the way you declare the loop. The loop counter in the Numeric `FOR` `LOOP` is automatically declared in the opening Numeric `FOR` `LOOP` statement. Cursor `FOR` `LOOP`s use cursors to determine the number of passes through the loop and automatically declare a `%ROWTYPE` style variable to automatically receive data that is fetched from the database.

Exceptions may be raised in the processing section. Exceptions are show-stoppers; that is, they are conditions that prevent processing from continuing in any sort of logical fashion. When an exception is raised, the processing section of the block is terminated at the point of the exception, any loops are exited, cursors are closed, and all required cleanup is performed automatically.

The optional exception-handling section is where you can choose to handle exceptions, including system-defined exceptions that are raised automatically by the system, or it is where you can choose to handle user-defined exceptions, which you explicitly raise with the `RAISE` statement somewhere in the processing section. Exceptions may or may not be raised, regardless of whether or not there is a corresponding exception handler. If they are raised and not handled, then an unhandled exception message will be returned from the database when the PL/SQL block is executed. But if an exception is raised and handled, then a success message will be returned. Exception handlers name the exception they are handling, as in `WHEN NO_DATA_FOUND`, and a series of exception handlers can be included in an exception-handling section. But the generic `WHEN OTHERS` can handle any exception that isn't otherwise handled in an exception-handling section and should be included last, if it's included at all.

A block can be nested in another block. One of the principle advantages of nesting blocks is the capability to handle exceptions within an inner set of code, enabling the outer block's code to continue processing. Any declared elements, such as variables, that are declared in the outer block's declaration section are local to the outer block and global to the inner block. This means that they can be

understood in both places, but may be overridden in the inner block since the inner block can declare its own variables, and those variables will be local to the inner block. Local variables take precedence over global variables, but block labels allow inner blocks to reference outer block elements in the event of a naming conflict. A PL/SQL block that starts with the reserved word DECLARE is an anonymous block. The rest of the book looks at named program units, such as procedures, functions, packages, and database triggers.

## Two-Minute Drill

- Character literals are delineated in single quotes, such as 'Sandy', as are date literals, such as '15-JUN-2004'. Numeric literals have no quotes, such as 1126.
- Single-line comments are indicated with a pair of successive dashes: --. Multiline comments start with a slash-asterisk, as in /\*, and continue until the first occurrence of the opposite, the asterisk-slash, as in \*/. The parser ignores comments.
- Variables may be declared for use in your processing section. Constants may also be declared with the reserved word CONSTANT. Both variables and constants must be given datatypes. The same datatypes used in SQL, such as NUMBER, NUMBER(n), NUMBER(n,m), VARCHAR2(n), and DATE, can be used, as can the new datatype BOOLEAN.
- To optionally initialize the value of a variable, use the assignment operator, :=. Constants must be initialized.
- Assignment statements in the processing section change the value of the specified variable on the left to be equal to whatever value will result from the evaluation of the expression on the right.
- SQL functions, such as LAST\_DAY, SUBSTR, INSTR, ABS, and more can be used in PL/SQL blocks. However, DECODE cannot be used in PL/SQL expressions. Instead, use the PL/SQL conditional statement IF . . . THEN . . . END IF.
- The IF . . . THEN . . . END IF statement can include the optional ELSIF clause as many times as you wish and up to one single ELSE clause, which, if included, must be at the end.
- Expressions in IF statements must evaluate to either TRUE or FALSE. Expressions may use comparison operators such as equals (=), greater than (>), and others. Also, logical operators such as AND and OR can be used.

- The LOOP . . . END LOOP statement doesn't formally include an EXIT statement, but you should always include at least one EXIT statement, and be sure to define logic to invoke the EXIT statement at some eventual iteration of the loop.
- In the Numeric FOR . . . LOOP . . . END LOOP, the loop counter variable is first identified in the FOR loop and is automatically declared and incremented as the loop progresses. The lower limit and upper limit of the counter are specified in the FOR loop. The EXIT is not required either, since the FOR loop automatically exits when the upper limit is reached. An example would be FOR I IN 1..10 LOOP . . . END LOOP.
- An explicit cursor can be declared as any valid SELECT statement and can optionally include references to declared elements, such as variables, provided that the variable is declared in a statement that precedes the statement declaring the cursor. The value of the variable that will be used in the cursor is whatever the value is at the time the OPEN statement is issued on the cursor.
- In the Cursor FOR LOOP, the loop counter is actually a %ROWTYPE variable, and the lower and upper limits are replaced with a SELECT statement, which is an automatically declared anonymous explicit cursor. Furthermore, the loop automatically performs an OPEN, FETCH, and CLOSE. The loop will pass through once for each record automatically fetched from the database and will exit when no more records remain.
- A variable declared in a block is considered local to the block and global to any nested blocks. If the nested block makes any declarations, the declared elements are unknown to the outer block and will override references to any outer-block-declared elements of the same name, unless block labels are used.
- All cursors have four attributes that are system-defined variables describing the state of the cursor. The four attributes are %FOUND, %NOTFOUND, %ISOPEN, and %ROWCOUNT. An attribute for an explicit cursor is identified by appending the name of the explicit cursor at the beginning of the attribute, such as cur\_employees%NOTFOUND, where cur\_employees is the name of the cursor. Implicit cursors are anonymous, so the generic name SQL is used to refer to the most recently executed implicit cursor to get its attributes. An example would be SQL%ROWCOUNT.
- The exception-handling section is always at the end of the block and always starts with the reserved word EXCEPTION, followed by one or more exception handlers. Exception handlers start with the reserved word WHEN, followed by the name of the exception, the reserved word THEN, and then one or more lines of executable statements.

- System-defined exceptions have predetermined names. The most commonly encountered system-defined exceptions are `VALUE_ERROR` and `ZERO_DIVIDE`, or in the case of implicit cursors, `NO_DATA_FOUND` and `TOO_MANY_ROWS`.
- User-defined exceptions are declared in the declaration section and are then raised with the `RAISE` statement somewhere in the processing section. You define the names of these exceptions and should create exception handlers for your user-defined exceptions.
- Exceptions that are raised, but not handled, will propagate out of the block and look for an outer block. If there is an outer block, then the outer block's exception handler will be investigated for an appropriate exception handler. If there isn't one, then the exception will propagate out again and continue until the exception either locates an exception handler or until there are no more outer blocks. The block's execution is terminated with an unhandled exception message when no exception handler is found.

## Chapter Questions

1. Which of the following keywords are required in any PL/SQL block? (Choose all that apply.)
  - A. DECLARE
  - B. BEGIN
  - C. EXCEPTION
  - D. END
2. What is wrong with the following code? (Choose all that apply.)

```
DECLARE
    v_answer VARCHAR2(30);
BEGIN
    v_answer + v_result := v_one / v_two;
END;
```

- A. Three variables need to be declared.
- B. You cannot use two variables on the left side of an assignment statement.
- C. You cannot use the slash in an expression.
- D. Nothing is wrong with this code.

**3. Assuming the existence of an explicit cursor named ships and a declared %ROWTYPE variable named current\_ship, which of the following FETCH statements are correct? (Choose all that apply.)**

- A.** FETCH ships INTO current\_ship%ROWTYPE;
- B.** FETCH ships INTO current\_ship.SHIP\_ID;
- C.** FETCH current\_ship INTO ships;
- D.** None of the above.

**4. What will happen when the following code is executed? (Choose all that apply.)**

```

DECLARE
    CURSOR cur_cruises IS
        SELECT CRUISE
        FROM CRUISES;
    rec_cruises cur_cruises%ROWTYPE;
BEGIN
    OPEN cur_cruises;
    SELECT CRUISE
        INTO rec_cruises
        FROM CRUISES;
    CLOSE cur_cruises;
EXCEPTION
    WHEN OTHERS THEN
        INSERT INTO ERRORS
            (ERROR_ID, MESSAGE)
        VALUES
            (SEQ_ERROR_ID.NEXTVAL, 'Something is wrong');
    COMMIT;
END;
```

- A.** Nothing, the code will not parse correctly.
  - B.** The TOO\_MANY\_ROWS exception will be raised, which is not handled, and the PL/SQL code will terminate with an error message.
  - C.** It depends on how many rows are in the database.
  - D.** The message, 'Something is wrong', will definitely be inserted into the ERRORS table.
- 5. Which of the following is not a cursor attribute? (Choose all that apply.)**
- A.** %ISOPEN
  - B.** %ISNOTOPEN

- C. %FOUND
- D. %NOTFOUND

**6. What will happen when the following code is executed?**

```
BEGIN
  CREATE TABLE HOLIDAYS
    (HOLIDAY_ID NUMBER(3),
     HOLIDAY     VARCHAR2(30));
END;
```

- A. The HOLIDAYS table will be created.
- B. The TOO\_MANY\_ROWS exception will be raised, but since it's not handled, nothing will happen.
- C. A parsing error will occur.
- D. The table will be created, but won't be declared since it isn't created in the DECLARE section.

**7. Which of the following are valid names for program units? (Choose all that apply.)**

- A. PROCESS\_ORDER
- B. SETUP-THE-DATABASE
- C. 100\_SAMPLE\_RECORDS
- D. GET\_ORDER\_#

**8. Consider the following code sample:**

```
DECLARE
  total_cabins  NUMBER(5) := 250;
  booked_cabins NUMBER(5);
  no_vacancies EXCEPTION;
BEGIN
  booked_cabins := 250;
  DECLARE
    booked_cabins CONSTANT NUMBER(5) := 199;
  BEGIN
    IF (booked_cabins >= total_cabins)
    THEN
      RAISE no_vacancies;
    END IF;
  EXCEPTION
```

## 56 OCP Developer PL/SQL Program Units Exam Guide

```
        WHEN no_vacancies THEN
            DBMS_OUTPUT.PUT_LINE('Inner block says: no vacancies');
        END;
        DBMS_OUTPUT.PUT_LINE('Complete');
    EXCEPTION
        WHEN no_vacancies THEN
            DBMS_OUTPUT.PUT_LINE('Outer block says: no vacancies.');
```

END;  
/

**What will be the result when this code is executed?**

- A. Inner block says: no vacancies.
  - B. Outer block says: no vacancies.
  - C. Complete.
  - D. An unhandled exception message.
- 9. You have created a PL/SQL block that declares variables using the %TYPE feature. When you created the block, the column upon which you based the %TYPE declaration had a datatype of NUMBER, but since you first created the block, it has been altered to have a datatype of VARCHAR2. Which of the following system-defined exceptions is most likely to occur in your block if you were you to execute the block now without modification?**
- A. NO\_DATA\_FOUND
  - B. VALUE\_ERROR
  - C. PROGRAM\_ERROR
  - D. VALUE\_CONFLICT
- 10. How many passes through the loop will occur? (Choose all that apply.)**

```
DECLARE
    FOR rec_ships IN (SELECT SHIP_ID FROM SHIPS)
    LOOP
        IF (rec_ships.SHIP_ID = 1)
            THEN
                EXIT;
            END IF;
        END LOOP;
    END;
```

/

- A. Once for each record in the SHIPS table.
- B. Only one. The EXIT will execute on the first record returned.
- C. None, this code won't parse since you can't have an EXIT statement in a Cursor FOR LOOP.
- D. None of the above.

## Answers to Chapter Questions

1. B, D. BEGIN and END

**Explanation** Only BEGIN and END are required. DECLARE is only used in anonymous blocks that require elements to be declared. EXCEPTION is only used when exception handlers are specified.

2. A, B. A: Three variables need to be declared. B: You cannot use two variables on the left side of an assignment statement.

**Explanation** Any and all variables and constants that are used in a PL/SQL block must be declared. In this example, only the variable `v_answer` is declared. The others, `v_result`, `v_one`, and `v_two`, all need to be declared. The left side of the assignment statement is the target of the assignment, and therefore must be one single variable. The slash is the symbol for division and is completely acceptable.

3. D. None of the above.

**Explanation** The proper FETCH statement is `FETCH ships INTO current_ship;`

4. C. It depends on how many rows are in the database.

**Explanation** Note that the explicit cursor, which is opened and closed, is never fetched. Instead, an implicit cursor has been slipped in between the OPEN and CLOSE statements. If there are no rows in the CRUISES table, then the NO\_DATA\_FOUND exception will be automatically raised by the system. That exception is not handled, but the WHEN OTHERS exception handler will catch the exception, and the message, 'Something is wrong', will be stored in the ERRORS table. On the other hand, if there is one single row in the CRUISES table, then the SELECT . . . INTO will execute fine, placing the single row's column values into the `rec_cruises` variable, and no exception is raised. Finally, if the table has more than one

row, then the `TOO_MANY_ROWS` exception will be raised, which is not handled, but once again the `WHEN OTHERS` exception handler will catch the exception, and the message, 'Something is wrong', will be stored in the `ERRORS` table.

5. B. `%ISNOTOPEN`

**Explanation** The four attributes are `%ISOPEN`, `%FOUND`, `%NOTFOUND`, and `%ROWCOUNT`. There is no such thing as `%ISNOTOPEN`.

6. C. A parsing error will occur.

**Explanation** The block is attempting to execute a DDL statement, and DDL statements aren't allowed in PL/SQL. Note, however, that there is a special package in PL/SQL that will support the execution of DDL statements by building them as text strings and sending them off to the database in another manner. But that is not what is being performed here.

7. A, D. `PROCESS_ORDER` and `GET_ORDER_#`

**Explanation** B is not allowed because of the use of hyphens. C is not allowed because names of program units cannot begin with numbers.

8. C. Complete.

**Explanation** A would have been correct had the `IF` statement evaluated to `TRUE`. Even though the user-defined exception is defined in the outer block, it is global to the inner block, and the inner block's exception handler will recognize the `RAISE` statement. If the `RAISE` statement would be executed, then the inner block's exception handler would have handled the exception, and the outer block's exception handler would never have been relevant, so B is not true. And since the user-defined exceptions are handled in both blocks, then D is not true. But C is true because the `IF` statement considers `booked_cabins` as the locally declared constant with a value of 199 and `total_cabins` as the globally declared variable with a value of 250. The comparison in the `IF` statement evaluates to `FALSE`, and the inner block completes, followed by the statement that prints the word Complete.

9. B. `VALUE_ERROR`

**Explanation** A occurs when you write an implicit cursor that attempts to `SELECT . . . INTO` a variable, but the `SELECT` returns no rows. C generally indicates some sort of major system bug. D is not a system-defined exception; I just made that up, but get used to that. Oracle loves to make up bogus names like this on the real OCP. The answer is B because `VALUE_ERROR` occurs when datatype conflicts exist, and if you've written code that was expecting a `NUMBER` datatype that is now a `VARCHAR2` datatype, then datatype conflicts within your code are possible.

10. D. None of the above.

**Explanation** You really can't tell how many passes through the loop will occur. A is wrong. The IF statement will EXIT upon the first occurrence of a `rec_ships.SHIP_ID` of 1, but you cannot be sure of B since you really have no way of knowing when that will occur. And C is just completely wrong. Although there are some people in the business who say that the use of an EXIT in a Cursor FOR LOOP is ill-advised, it's accepted by the parser. D is the answer, since the number of rows returned will be all of the rows defined in the SELECT statement until the first occurrence of `rec_ships.SHIP_ID` is equal to 1, which could be any time, if ever; perhaps that value never occurs. D is the answer.

