CHAPTER 3

GENERAL PROGRAMMING LANGUAGE FUNDAMENTALS

Chapter Objectives

In this Chapter, you will learn about:

In the first two chapters you learned about the difference between machine language and a programming language. You have also learned how PL/SQL is different from SQL and about the PL/SQL basic block structure. This is similar to learning the history behind a foreign language and in what context it is used. In order to use the PL/SQL language, you will have to learn the key words, what they mean, and when and how to use them. First, you will encounter the different types of key words and then their full syntax. Finally, in this chapter, you will expand on simple block structure with an exploration of scope and nesting blocks.

LAB 3.1

PL/SQL PROGRAMMING FUNDAMENTALS

LAB OBJECTIVES

After this Lab, you will be able to:

- ✔ Make Use of PL/SQL Language Components
- ✔ Make Use of PL/SQL Variables
- ✔ Handle PL/SQL Reserved Words
- ✔ Make Use of Identifiers in PL/SQL
- ✔ Make Use of Anchored Data types
- ✔ Declare and Initialize Variables
- ✔ Understand the Scope of a Block, Nested Blocks, and Labels

In most languages, you have only two sets of characters: numbers and letters. Some languages, such as Hebrew or Tibetan, have specific characters for vowels that are not placed in line with consonants. Additionally, other languages, such as Japanese, have three character sets: one for words originally taken from the Chinese language, another set for native Japanese words, and then a third for other foreign words. In order to speak any foreign language, you have to begin by learning these character sets. Then you progress to learn how to make words from these character sets. Finally, you learn the parts of speech and you can begin talking. You can think of PL/SQL as being a more complex language because it has many character types and, additionally, many types of words or lexical units that are made from these character sets. Once you learn these, you can progress to learn the structure of the PL/SQL language.

CHARACTER TYPES

The PL/SQL engine accepts four types of characters: letters, digits, symbols (*, +, -, =, etc.), and white space. When elements from one or more of these character types are joined together, they will create a lexical unit (these lexical units can be

a combination of character types). The lexical units are the words of the PL/SQL language. First you need to learn the PL/SQL vocabulary, and then you will move on to the syntax, or grammar. Soon you can start talking in PL/SQL.

Although PL/SQL can be considered a language, don't try talking to your fellow programmers in PL/SQL. For example, at a dinner table of programmers, if you say, "BEGIN, LOOP FOR PEAS IN PLATE EXECUTE EAT PEAS, END LOOP, EXCEPTION WHEN BROCCOLI FOUND EXECUTE SEND TO PRESIDENT BUSH, END EAT PEAS," you may not be considered human. This type of language is reserved for Terminators and the like.

LEXICAL UNITS

A language such as English contains different parts of speech. Each part of speech, such as a verb or noun, behaves in a different way and must be used according to specific rules. Likewise, a programming language has lexical units that are the building blocks of the language. PL/SQL lexical units fall within one of the following five groups:

1. *Identifiers*. Identifiers must begin with a letter and may be up to 30 characters long. See a PL/SQL manual for a more detailed list of restrictions; generally, if you stay with characters, numbers, and " ", and avoid reserved words, you will not run into problems.
2. *Reserved words*. Reserved words are words that PL/SQL saves for its own use (e.g., BEGIN, END, SELECT).
3. *Delimiters*. These are characters that have special meaning to PL/SQL, such as arithmetic operators and quotation marks.
4. *Literals*. A literal is any value (character, numeric, or Boolean [true/false]) that is not an identifier. 123, "Declaration of Independence," and FALSE are examples of literals.
5. *Comments*. These can be either single-line comments (i.e., --) or multiline comments (i.e., /* */).

See Appendix B, "PL/SQL Formatting Guide," for details on formatting.

In the following exercises, you will practice putting these units together.

LAB 3.1 EXERCISES

3.1.1 MAKE USE OF PL/SQL LANGUAGE COMPONENTS

Now that you have the character types and the lexical units, it is equivalent to knowing the alphabet and how to spell out words.

a) Why does PL/SQL have so many different types of characters? What are they used for?

b) What would be the equivalent of a verb and a noun in English in PL/SQL? Do you speak PL/SQL?

3.1.2 MAKE USE OF PL/SQL VARIABLES

Variables may be used to hold a temporary value.

```
Syntax : <variable-name> <data type> [optional default
assignment]
```

Variables may also be known as identifiers. There are some restrictions that you need to be familiar with: Variables must begin with a letter and may be up to 30 characters long. Consider the following example:

■ *FOR EXAMPLE*

This example contains a list of valid identifiers:

```
v_student_id
v_last_name
V_LAST_NAME
apt_#
```

It is important to note that the identifiers `v_last_name` and `V_LAST_NAME` are considered identical because PL/SQL is not case sensitive.

Next, consider an example of illegal identifiers:

■ *FOR EXAMPLE*

```
X+Y
1st_year
student ID
```

Identifier `X+Y` is illegal because it contains the "+" sign. This sign is reserved by PL/SQL to denote an addition operation, and it is referred to as a mathematical

symbol. Identifier, `1st_year` is illegal because it starts with a number. Finally, identifier `student ID` is illegal because it contains a space.

Next, consider another example:

■ *FOR EXAMPLE*

```
SET SERVEROUTPUT ON;
DECLARE
   first&last_names VARCHAR2(30);
BEGIN
   first&last_names := 'TEST NAME';
   DBMS_OUTPUT.PUT_LINE(first&last_names);
END;
```

In this example, you declare a variable called `first&last_names`. Next, you assign a value to this variable and display this value on the screen. When run, the example produces the following output:

```
Enter value for last_names: Elena
old   2:    first&last_names VARCHAR2(30);
new   2:    firstElena VARCHAR2(30);
Enter value for last_names: Elena
old   4:    first&last_names := 'TEST NAME';
new   4:    firstElena := 'TEST NAME';
Enter value for last_names: Elena
old   5:    DBMS_OUTPUT.PUT_LINE(first&last_names);
new   5:    DBMS_OUTPUT.PUT_LINE(firstElena);
TEST NAME
PL/SQL procedure successfully completed.
```

Consider the output produced. Because there is an ampersand (&) present in the name of the variable `first&last_names`, the portion of the variable is considered to be a substitution variable (you learned about substitution variables in Chapter 2). In other words, the portion of the variable name after the ampersand (`last_names`) is treated by the PL/SQL compiler as a substitution variable. As a result, you are prompted to enter the value for the `last_names` variable every time the compiler encounters it.

It is important to realize that while this example does not produce any syntax errors, the variable `first&last_names` is still an invalid identifier because the ampersand character is reserved for substitution variables. To avoid this problem, change the name of the variable from `first&last_names` to `first_and_last_names`. Therefore, *you should use an ampersand sign in the name of a variable only when you use it as a substitution variable in your program.*

■ *FOR EXAMPLE*

```
-- ch03_1a.pls
SET SERVEROUTPUT ON
DECLARE
   v_name VARCHAR2(30);
   v_dob DATE;
   v_us_citizen BOOLEAN;
BEGIN
   DBMS_OUTPUT.PUT_LINE(v_name||'born on'||v_dob);
END;
```

a) If you ran the previous example in a SQL*Plus, what would be the result?

b) Run the example and see what happens. Explain what is happening as the focus moves from one line to the next.

3.1.3 HANDLE PL/SQL RESERVED WORDS

Reserved words are ones that PL/SQL saves for its own use (e.g., BEGIN, END, and SELECT). You cannot use reserved words for names of variables, literals, or user-defined exceptions.

■ *FOR EXAMPLE*

```
SET SERVEROUTPUT ON;
DECLARE
   exception VARCHAR2(15);
BEGIN
   exception := 'This is a test';
   DBMS_OUTPUT.PUT_LINE(exception);
END;
```

a) What would happen if you ran the preceding PL/SQL block? Would you receive an error message? If so, explain.

3.1.4 MAKE USE OF IDENTIFIERS IN PL/SQL

Take a look at the use of identifiers in the following example:

■ *FOR EXAMPLE*

```
SET SERVEROUTPUT ON;
DECLARE
   v_var1 VARCHAR2(20);
   v_var2 VARCHAR2(6);
   v_var3 NUMBER(5,3);
BEGIN
   v_var1 := 'string literal';
   v_var2 := '12.345';
   v_var3 := 12.345;
   DBMS_OUTPUT.PUT_LINE('v_var1: '||v_var1);
   DBMS_OUTPUT.PUT_LINE('v_var2: '||v_var2);
   DBMS_OUTPUT.PUT_LINE('v_var3: '||v_var3);
END;
```

In this example, you declare and initialize three variables. The values that you assign to them are literals. The first two values, 'string literal' and '12.345' are string literals because they are enclosed by single quotes. The third value, 12.345, is a numeric literal. When run, the example produces the following output:

```
v_var1: string literal
v_var2: 12.345
v_var3: 12.345
PL/SQL procedure successfully completed.
```

Consider another example that uses numeric literals:

■ *FOR EXAMPLE*

```
SET SERVEROUTPUT ON;
DECLARE
   v_var1 NUMBER(2) := 123;
   v_var2 NUMBER(3) := 123;
   v_var3 NUMBER(5,3) := 123456.123;
BEGIN
   DBMS_OUTPUT.PUT_LINE('v_var1: '||v_var1);
   DBMS_OUTPUT.PUT_LINE('v_var2: '||v_var2);
   DBMS_OUTPUT.PUT_LINE('v_var3: '||v_var3);
END;
```

a) What would happen if you ran the preceding PL/SQL block?

3.1.5 Make Use of Anchored Data Types

The data type that you assign to a variable can be based on a database object. This is called an *anchored declaration* since the variable's data type is dependent on that of the underlying object. It is wise to make use of anchored data types when possible so that you do not have to update your PL/SQL when the data types of base objects change.

```
Syntax:  <variable_name> <type attribute>%TYPE
```

The type is a direct reference to a database column.

■ *FOR EXAMPLE*

```
-- ch03_2a.pls
SET SERVEROUTPUT ON
DECLARE
   v_name student.first_name%TYPE;
   v_grade grade.numeric_grade%TYPE;
BEGIN
   DBMS_OUTPUT.PUT_LINE(NVL(v_name, 'No Name ')||
      ' has grade of '||NVL(v_grade, 0));
END;
```

a) In the previous example, what has been declared? State the data type and value.

3.1.6 Declare and Initialize Variables

In PL/SQL, variables must be declared in order to be referenced. This is done in the initial declarative section of a PL/SQL block. Remember that each declaration must be terminated with a semicolon. Variables can be assigned using the assignment operator ":=". If you declare a variable to be a constant, it will retain the same value throughout the block; in order to do this, you must give it a value at declaration.

Type the following into a text file and run the script from a SQL*Plus session.

```
-- ch03_3a.pls
SET SERVEROUTPUT ON
DECLARE
   v_cookies_amt NUMBER := 2;
   v_calories_per_cookie CONSTANT NUMBER := 300;
BEGIN
   DBMS_OUTPUT.PUT_LINE('I ate ' || v_cookies_amt ||
      ' cookies with ' ||  v_cookies_amt *
      v_calories_per_cookie || ' calories.');
   v_cookies_amt := 3;
   DBMS_OUTPUT.PUT_LINE('I really ate ' ||
      v_cookies_amt
      || ' cookies with ' ||  v_cookies_amt *
      v_calories_per_cookie || ' calories.');
   v_cookies_amt := v_cookies_amt + 5;
   DBMS_OUTPUT.PUT_LINE('The truth is, I actually ate '
      || v_cookies_amt || ' cookies with ' ||
   v_cookies_amt * v_calories_per_cookie
      || ' calories.');
END;
```

a) What will the output be for the preceding script? Explain what is being declared and what the value of the variable is throughout the scope of the block.

■ *FOR EXAMPLE*

```
-- ch03_3a.pls
SET SERVEROUTPUT ON
DECLARE
   v_lname VARCHAR2(30);
   v_regdate DATE;
   v_pctincr CONSTANT NUMBER(4,2) := 1.50;
   v_counter NUMBER := 0;
   v_new_cost course.cost%TYPE;
   v_YorN BOOLEAN := TRUE;
BEGIN
       DBMS_OUTPUT.PUT.PUT_LINE(V_COUNTER);
       DBMS_OUTPUT.PUT_LINE(V_NEW_COST);
END;
```

b) In the previous example, add the following expressions to the beginning of the procedure (immediately after the BEGIN in the previous example), then explain the values of the variables at the beginning and at the end of the script.

```
v_counter := NVL(v_counter, 0) + 1;
v_new_cost := 800 * v_pctincr;
```

PL/SQL variables are held together with expressions and operators. An expression is a sequence of variables and literals, separated by operators. These expressions are then used to manipulate data, perform calculations, and compare data.

Expressions are composed of a combination of operands and operators. An *operand* is an argument to the operator; it can be a variable, a constant, a function call. An *operator* is what specifies the action (+, **, /, OR, etc.).

You can use parentheses to control the order in which Oracle evaluates an expression. Continue to add the following to your SQL script the following:

```
v_counter := ((v_counter + 5)*2) / 2;
v_new_cost := (v_new_cost * v_counter)/4;
```

c) What will the values of the variables be at the end of the script?

3.1.7 Understand the Scope of a Block, Nested Blocks, and Labels

SCOPE OF A VARIABLE

The scope, or existence, of structures defined in the declaration section are local to that block. The block also provides the scope for exceptions that are declared and raised. Exceptions will be covered in more detail in Chapters 7, 10, and 11.

The scope of a variable is the portion of the program in which the variable can be accessed, or where the variable is visible. It usually extends from the moment of declaration until the end of the block in which the variable was declared. The visibility of a variable is the part of the program where the variable can be accessed.

```
BEGIN    -- outer block
         BEGIN -- inner block
                  …;
         END;  -- end of inner block
END;     -- end of outer block
```

LABELS AND NESTED BLOCKS

Labels can be added to a block in order to improve readability and to qualify the names of elements that exist under the same name in nested blocks. The name of

the block must precede the first line of executable code (either the BEGIN or DECLARE) as follows:

FOR EXAMPLE

```
-- ch03_4a.pls
set serveroutput on
   <<find_stu_num>>
   BEGIN
      DBMS_OUTPUT.PUT_LINE('The procedure
                  find_stu_num has been executed.');
   END find_stu_num;
```

The label optionally appears after END. In SQL*Plus, the first line of a PL/SQL block cannot be a label. For commenting purposes, you may alternatively use "- -" or /*, ending with */.

Blocks can be nested in the main section or in an exception handler. A *nested block* is a block that is placed fully within another block. This has an impact on the scope and visibility of variables. The scope of a variable in a nested block is the period when memory is being allocated for the variable and extends from the moment of declaration until the END of the nested block from which it was declared. The visibility of a variable is the part of the program where the variable can be accessed.

FOR EXAMPLE

```
-- ch03_4b.pls
SET SERVEROUTPUT ON
<< outer_block >>
DECLARE
   v_test NUMBER := 123;
BEGIN
   DBMS_OUTPUT.PUT_LINE
      ('Outer Block, v_test: '||v_test);
   << inner_block >>
   DECLARE
      v_test NUMBER := 456;
   BEGIN
      DBMS_OUTPUT.PUT_LINE
         ('Inner Block, v_test: '||v_test);
      DBMS_OUTPUT.PUT_LINE
         ('Inner Block, outer_block.v_test: '||
           outer_block.v_test);
   END inner_block;
END outer_block;
```

This example produces the following output:

```
Outer Block, v_test: 123
Inner Block, v_test: 456
Inner Block, outer_block.v_test: 123
```

a) If the following example were run in SQL*Plus, what do you think would be displayed?

```
-- ch03_5a.pls
SET SERVEROUTPUT ON
DECLARE
   e_show_exception_scope EXCEPTION;
   v_student_id           NUMBER := 123;
BEGIN
  DBMS_OUTPUT.PUT_LINE('outer student id is '
     ||v_student_id);
   DECLARE
     v_student_id    VARCHAR2(8) := 125;
   BEGIN
      DBMS_OUTPUT.PUT_LINE('inner student id is '
         ||v_student_id);
      RAISE e_show_exception_scope;
   END;
EXCEPTION
   WHEN e_show_exception_scope
   THEN
      DBMS_OUTPUT.PUT_LINE('When am I displayed?');
      DBMS_OUTPUT.PUT_LINE('outer student id is '
         ||v_student_id);
END;
```

b) Now run the example and see if it produces what you expected. Explain how the focus moves from one block to another in this example.

LAB 3.1 EXERCISE ANSWERS

This section gives you some suggested answers to the questions in Lab 3.1, with discussion related to how those answers resulted. The most important thing to realize is whether your answer works. You should figure out the implications of the answers here and what the effects are from any different answers you may come up with.

3.1.1 ANSWERS

a) Why does PL/SQL have so many different types of characters? What are they used for?

Answer: The PL/SQL engine recognizes different characters as having different meaning and therefore processes them differently. PL/SQL is neither a pure mathematical language nor a spoken language, yet it contains elements of both. Letters will form various lexical units such as identifiers or key words, mathematic symbols will form lexical units known as delimiters that will perform an operation, and other symbols, such as /, indicate comments that should not be processed.*

b) What would be the equivalent of a verb and a noun in English in PL/SQL? Do you speak PL/SQL?

Answer: A noun would be similar to the lexical unit known as an identifier. A verb would be similar to the lexical unit known as a delimiter. Delimiters can simply be quotation marks, but others perform a function such as to multiply "".*

3.1.2 ANSWERS

a) If you ran the previous example in a SQL*Plus, what would be the result?

Answer: Assuming `SET SERVEROUTPUT ON` *had been issued, you would get only* `born on`. *The reason is that the variables* `v_name` *and* `v_dob` *have no values.*

b) Run the example and see what happens. Explain what is happening as the focus moves from one line to the next.

Answer: Three variables are declared. When each one is declared, its initial value is null. `v_name` *is set as a varchar2VARCHAR2 with a length of 30,* `v_dob` *is set as a character type date, and* `v_us_citizen` *is set to BOOLEAN. Once the executable section begins, the variables have no value and, therefore, when the DBMS_OUTPUT is told to print their values, it prints nothing.*

This can be seen if the variables were replaced as follows: Instead of `v_name`, use `NVL(v_name, 'No Name')` and instead of `v_dob` use `NVL (v_dob, '01-Jan-1999')`. Then run the same block and you will get

```
No Name born on 01-Jan-1999
```

In order to make use of a variable, you must declare it in the declaration section of the PL/SQL block. You will have to give it a name and state its data type. You also have the option to give your variable an initial value. Note that if you do not assign a variable an initial value, it will be null. It is also possible to constrain the declaration to "not null," in which case you must assign an initial value. Variables must first be declared and then they can be referenced. PL/SQL does not allow forward references. You can set the variable to be a constant, which means it cannot change.

3.1.3 ANSWERS

a) What would happen if you ran the above PL/SQL block? Would you receive an error message? If so, explain.

Answer: In this example, you declare a variable called `exception`*. Next, you initialize this variable and display its value on the screen.*

This example illustrates an invalid use of reserved words. To the PL/SQL compiler, "exception" is a reserved word and it denotes the beginning of the exception-handling section. As a result, it cannot be used to name a variable. Consider the huge error message produced by this tiny example.

```
exception VARCHAR2(15);
   *
ERROR at line 2:
ORA-06550: line 2, column 4:
PLS-00103: Encountered the symbol "EXCEPTION" when
expecting one of the following:
begin function package pragma procedure subtype type use
<an identifier> <a double-quoted delimited-identifier>
 cursor
form current
The symbol "begin was inserted before "EXCEPTION"
to continue.
ORA-06550: line 4, column 4:
PLS-00103: Encountered the symbol "EXCEPTION" when
expecting one of the following:
begin declare exit for goto if loop mod null pragma
 raise
return select update while <an identifier>
<a double-quoted delimited-identifier> <a bin
ORA-06550: line 5, column 25:
PLS-00103: Encountered the symbol "EXCEPTION" when
expecting one of the following:
( ) - + mod not null others <an identifier>
<a double-quoted delimited-identifier> <a bind variable>
avg
count current exists max min prior sql s
ORA-06550: line 7, column 0:
PLS-00103: Encountered the symbol "end-of-file" when
expecting one of the following:
begin declare end exception exit for goto if loop
```

Here is a question you should ask yourself: If you did not know that the word "exception" is a reserved word, do you think you would attempt to debug the preceding script after looking at this error message? I know I would not.

3.1.4 ANSWERS

a) What would happen if you ran the preceding PL/SQL block?

Answer: In this example, you declare and initialize three numeric variables. The first declaration and initialization (`v_var1 NUMBER(2) := 123`*) causes an error because the value 123 exceeds the specified precision. The second variable declaration and initialization (*`v_var2 NUMBER(3) := 123`*) does not cause any errors because the value 123 corresponds to the specified precision. The last declaration and initialization (*`v_var3 NUMBER(5,3) := 123456.123`*) causes an error because the value 123456.123 exceeds the specified precision. As a result, this example produces the following output:*

```
DECLARE
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error
ORA-06512: at line 2
```

3.1.5 ANSWERS

a) In the previous example, what has been declared? State the data type and value.

Answer: The variable v_name was declared with the identical data type as the column first_name from the database table STUDENT - varchar2(25). Additionally, the variable v_grade was declared the identical data type as the column grade_numeric on the grade database table – number NUMBER(3). Each has a value of null.

Most Common Data Types

VARCHAR2(maximum_length)

- Stores variable-length character data.
- Takes a required parameter that specifies a maximum length up to 32,767 bytes.
- Does not use a constant or variable to specify the maximum length; an integer literal must be used.
- The maximum width of a VARCHAR2 database column is 4000 bytes.

CHAR[(maximum_length)]

- Stores fixed-length (blank-padded if necessary) character data.
- Takes an optional parameter that specifies a maximum length up to 32,767 bytes.
- Does not use a constant or variable to specify the maximum length; an integer literal must be used. If maximum length is not specified, it defaults to 1.

(*cont'd.*)

- The maximum width of a CHAR database column is 2000 bytes; the default is 1 byte.

NUMBER[(precision, scale)]

- Stores fixed or floating-point numbers of virtually any size.
- Precision is the total number of digits.
- Scale determines where rounding occurs.
- It is possible to specify precision and omit scale, in which case scale is 0 and only integers are allowed.
- Constants or variables cannot be used to specify precision and scale; integer literals must be used.
- Maximum precision of a NUMBER value is 38 decimal digits.
- Scale can range from –84 to 127.
- For instance, a scale of 2 rounds to the nearest hundredth (3.456 becomes 3.46).
- Scale can be negative, which causes rounding to the left of the decimal point. For example, a scale of -3 rounds to the nearest thousandth (3456 becomes 3000). A scale of zero rounds to the nearest whole number. If you do not specify the scale, it defaults to zero.

BINARY_INTEGER

- Stores signed integer variables.
- Compares to the NUMBER data type. BINARY_INTEGER variables are stored in the binary format, which takes less space.
- Calculations are faster.
- Can store any integer value in the range –2,147,483,747 through 2,147,483,747.
- This data type is primarily used for indexing a PL/SQL table. This will be explained in more depth in Chapter 16, "PL/SQL Tables." You cannot create a column in a regular table of binary_integer type.

DATE

- Stores fixed-length date values.
- Valid dates for DATE variables include January 1, 4712 B.C. to December 31, A.D. 9999.
- When stored in a database column, date values include the time of day in seconds since midnight. The date portion defaults to the first day of the current month; the time portion defaults to midnight.
- Dates are actually stored in binary format and will be displayed according to the default format.

TIMESTAMP

- This is a new data type introduced with Oracle 9i. It is an extension of the DATE data type. It stores fixed-length date values with preci-

sion down to a fraction of a second with up to 9 places after the decimal (the default is 6). Here is an example of the default this displays for this data type: '12-JAN-2002 09.51.44.000000 PM'

- The "with timezone" or "with local timezone" option allows the TIMESTAMP to be related to a particular time zone. This will then be adjusted to the time zone of the database. For example, this would allow a global database to have an entry in London and New York recorded as being the same time even though it will display as noon in New York and 5 P.M. in London.

BOOLEAN

- Stores the values TRUE and FALSE and the nonvalue NULL. Recall that NULL stands for a missing, unknown, or inapplicable value.
- Only the values TRUE and FALSE and the nonvalue NULL can be assigned to a BOOLEAN variable.
- The values TRUE and FALSE cannot be inserted into a database column.

LONG

- Stores variable-length character strings.
- The LONG data type is like the VARCHAR2 data type, except that the maximum length of a LONG value is 2 gigabytes.
- You cannot select a value longer than 4000 bytes from a LONG column into a LONG variable.
- LONG columns can store text, arrays of characters, or even short documents. You can reference LONG columns in UPDATE, INSERT, and (most) SELECT statements, but not in expressions, SQL function calls, or certain SQL clauses, such as WHERE, GROUP BY, and CONNECT BY.

LONG RAW

- Stores raw binary data of variable length up to 2 gigabytes.

LOB (Large Object)

- There are four types of LOBS: BLOB, CLOB, NCLOB, and BFILE. These can store binary objects, such as image or video files, up to 4 gigabytes in length.
- A BFILE is a large binary file stored outside the database. The maximum size is 4 gigabytes.

ROWID

- Internally, every Oracle database table has a ROWID pseudocolumn, which stores binary values called rowids.
- Rowids uniquely identify rows and provide the fastest way to access particular rows.

(*cont'd.*)

- Use the ROWID data type to store rowids in a readable format.
- When you select or fetch a rowid into a ROWID variable, you can use the function ROWIDTOCHAR, which converts the binary value into an 18-byte character string and returns it in that format.
- Extended rowids use a base 64 encoding of the physical address for each row. The encoding characters are A–Z, a–z, 0–9, +, and /. Row ID in Oracle 9i is as follows: OOOOOOFFFBBBBBBRRR. Each component has a meaning. The first section, OOOOOO, signifies the database segment. The next section, FFF, indicates the tablespace-relative datafile number of the datafile that contains the row. The following section, BBBBBB, is the data block that contains the row. The last section, RRR, is the row in the block (keep in mind that this may change in future versions of Oracle).

3.1.6 ANSWERS

a) What will the output be for the preceding script? Explain what is being declared and what the value of the variable is throughout the scope of the block.

Answer: The server output will be

```
I ate 2 cookies with 600 calories.
I really ate 3 cookies with 900 calories.
The truth is, I actually ate 8 cookies with
2400 calories.
PL/SQL procedure successfully completed.
```

Initially the variable `v_cookies_amt` *is declared to be a NUMBER with the value of 2, and the variable* `v_calories_per_cookie` *is declared to be a CONSTANT NUMBER with a value of 300 (since it is declared to be a tCONSTANT, it will not change its value). In the course of the procedure, the value of* `v_cookies_amt` *is later set to be 3, and then finally it is set to be its current value, 3 plus 5, thus becoming 8.*

b) In the previous example, add the following expressions to the beginning of the procedure, then explain the values of the variables at the beginning and at the end of the script.

Answer: Initially the variable `v_lname` *is declared as a data type VARCHAR2 with a length of 30 and a value of null. The variable* `v_regdate` *is declared as data type date with a value of null. The variable* `v_pctincr` *is declared as CONSTANT*

NUMBER with a length of 4 and a precision of 2 and a value of 1.15. The variable `v_counter` *is declared as NUMBER with a value of 0. The variable* `v_YorN` *is declared as a variable of BOOLEAN data type and a value of TRUE.*

*The output of the procedure will be as follows (make sure you have entered SET SERVEROUTPUT ON earlier on in your SQL*Plus session):*

```
1
1200
PL/SQL procedure successfully completed.
```

Once the executable section is complete, the variable `v_counter` *will be changed from null to 1. The value of* `v_new_cost` *will change from null to 1200 (800 times 1.50).*

Note that a common way to find out the value of a variable at different points in a block is to add a DBMS_OUTPUT.PUT_LINE(v_variable_name); throughout the block.

c) What will the values of the variables be at the end of the script?

Answer: The value of `v_counter` *will then change from 1 to 6, which is ((1 + 5) *2))/2, and the value of* `new_cost` *will go from 1200 to 1800, which is (800 * 6)/4. The output from running this procedure will be:*

```
6
1800
PL/SQL procedure successfully completed.
```

Operators (Delimiters): the Separators in an Expression

Arithmetic (** , * , / , + , -)
Comparison(=, <> , != , < , > , <= , >= , LIKE , IN , BETWEEN , IS NULL)
Logical (AND, OR, NOT)
String (||, LIKE)
Expressions
Operator Precedence
    ** , NOT
    +, - (arithmetic identity and negation) *, / + , - , || =, <>, != , <= , >= , < , > , LIKE, BETWEEN, IN, IS NULL
  AND—logical conjunction
  OR—logical inclusion

3.1.7 ANSWERS

a) If the following example were run in SQL*Plus, what do you think would be displayed?

Answer: The following would result:

```
outer student id is 123
inner student id is 125
When am I displayed?
outer student id is 123
PL/SQL procedure successfully completed.
```

b) Now run the example and see if it produces what you expected. Explain how the focus moves from one block to another in this example.

Answer: The variable `e_Show_Exception_Scope` *is declared as an exception type in the declaration section of the block. There is also a declaration of the variable called* `v_student_id` *of data type NUMBER that is initialized to the number 123. This variable has a scope of the entire block, but it is visible only outside of the inner block. Once the inner block begins, another variable, named* `v_student_id`*, is declared. This time it is of data type VARCHAR2(8) and is initialized to 125. This variable will have a scope and visibility only within the inner block. The use of DBMS_OUTPUT helps to show which variable is visible. The inner block raises the exception* `e_Show_Exception_Scope`*; this means that the focus will move out of the execution section and into the exception section. The focus will look for an exception named* `e_Show_Exception_Scope`*. Since the inner block has no exception with this name, the focus will move to the outer block's exception section and it will find the exception. The inner variable* `v_student_id` *is now out of scope and visibility. The outer variable* `v_student_id` *(which has always been in scope) now regains visibility. Because the exception has an IF/THEN construct, it will execute the DBMS_OUTPUT call. This is a simple use of nested blocks. Later in the book you will see more complex examples. Once you have covered exception handling in depth in Chapters 7, 10, and 11, you will see that there is greater opportunity to make use of nested blocks.*

LAB 3.1 SELF-REVIEW QUESTIONS

In order to test your progress, you should be able to answer the following questions.

1) If a variable is declared as follows, what are the results?

```
v_fixed_amount CONSTANT NUMBER;
```

a) _____ A NUMBER variable called `v_fixed_amount` has been declared (it will remain as a constant once initialized).

b) _____ A NUMBER variable called `v_fixed_amount` has been declared (it will remain as null).

c) _____ An error message will result because constant initialization must be done in the executable section of the block.

d) _____ An error message will result because the declaration for the CONSTANT is missing an assignment to a NUMBER.

2) Which of the following are valid character types for PL/SQL?

a) _____ Numbers
b) _____ English letters
c) _____ Paragraph returns
d) _____ Arithmetic symbols
e) _____ Japanese Kanji

3) A variable may be used for which of the following?

a) _____ To hold a constant, such as the value of π
b) _____ To hold the value of a counter that keeps changing
c) _____ To place a value that will be inserted into the database
d) _____ To hold onto the function of an operand
e) _____ To hold any value as long as you declare it

4) Which of the following will declare a variable that is of the identical data type as the `student_id` in the database table STUDENT in the CTA database?

a) _____ v_id student_id := 123;
b) _____ v_id binary integer;
c) _____ v_id number`NUMBER` := 24;
d) _____ v_id student_id%type;

5) The value of a variable is set to null after the 'end;' of the block is issued.

a) _____ True
b) _____ False

Answers appear in Appendix A, Section 3.1.

CHAPTER 3

TEST YOUR THINKING

Before starting these projects, take a look at the formatting guidelines in Appendix B. Make your variable names conform to the standard. At the top of the *declaration section,* put a comment stating which naming standard you are using.

1) Write a PL/SQL block
- **a)** That includes declarations for the following variables:
  - A VARCHAR2 data type that can contain the string 'Introduction to Oracle PL/SQL'
  - A NUMBER that can be assigned 987654.55, but not 987654.567 or 9876543.55
  - A CONSTANT (you choose the correct data type) that is autoinitialized to the value '603D'
  - A BOOLEAN
  - A DATE data type autoinitialized to one week from today
- **b)** In the body of the PL/SQL block, put a DBMS_OUTPUT.PUT_LINE message for each of the variables that received an autoinitialization value.
- **c)** In a comment at the bottom of the PL/SQL block, state the value of your NUMBER data type.

2) Alter the PL/SQL block you created in Project 1 to conform to the following specs:
- **a)** Remove the DBMS_OUTPUT.PUT_LINE messages.
- **b)** In the body of the PL/SQL block, write a selection test (IF) that does the following (use a nested IF statement where appropriate):
  - **i)** Check whether the VARCHAR2 you created contains the course named 'Introduction to Underwater Basketweaving'.
  - **ii)** If it does, then put a DBMS_OUTPUT.PUT_LINE message on the screen that says so.
  - **iii)** If it does not, then test to see if the CONSTANT you created contains the room number 603D.
  - **iv)** If it does, then put a DBMS_OUTPUT.PUT_LINE message on the screen that states the course name and the room number that you've reached in this logic.

   v) If it does not, then put a DBMS_OUTPUT.PUT_LINE Message on the screen that states that the course and location could not be determined.

c) Add a WHEN OTHERS EXCEPTION that puts a DBMS_OUTPUT.PUT_LINE message on the screen that says that an error occurred.

The projects in this section are meant to have you utilize all of the skills that you have acquired throughout this chapter. The answers to these projects can be found in Appendix D and at the companion Web site to this book, located at http://authors.phptr.com/rosenzweig3e. Visit the Web site periodically to share and discuss your answers.