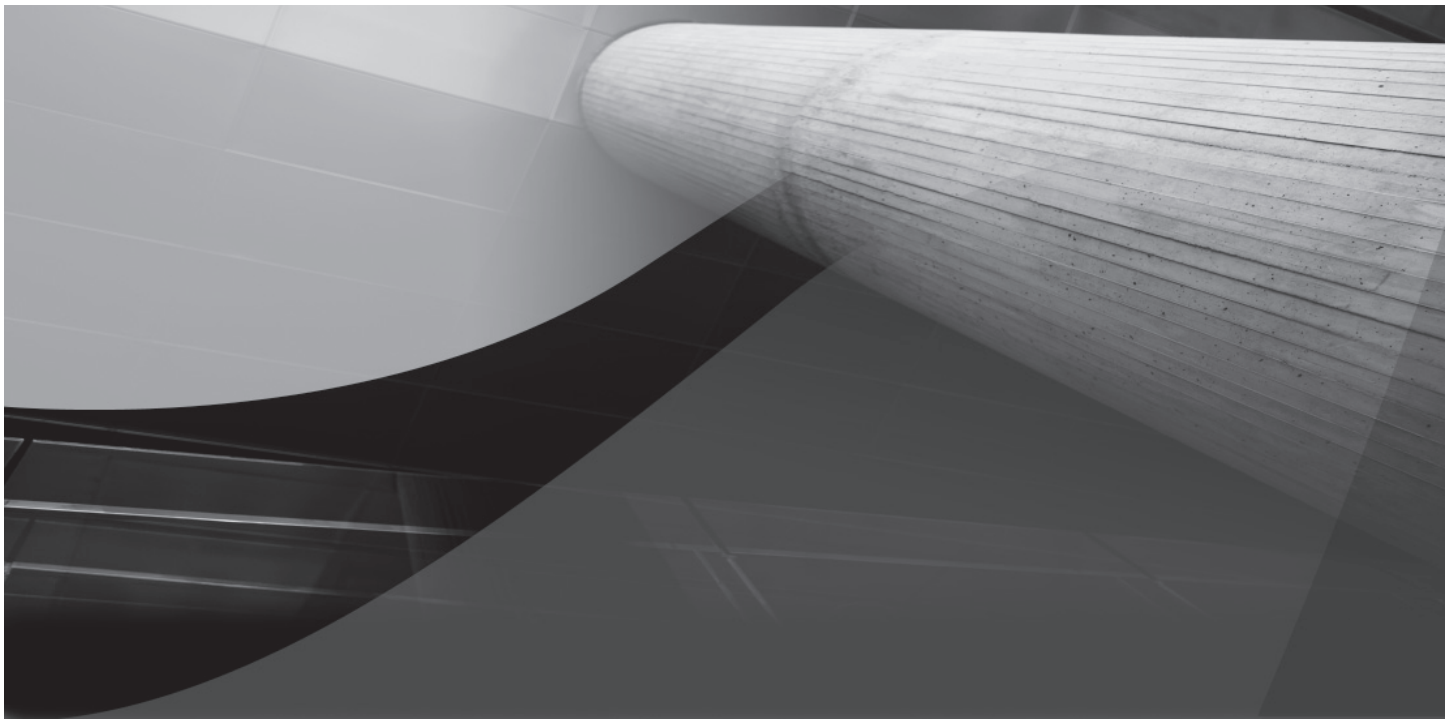


CHAPTER 2

PL/SQL Basics

common beginning place is a summary of language components. This chapter tours PL/SQL features. Subsequent chapters develop details that explain why the PL/SQL language is a robust tool with many options.

As an introduction to PL/SQL basics, this chapter introduces and briefly discusses

- Oracle PL/SQL block structure
- Variables, assignments, and operators
- Control structures
- Conditional structures
- Iterative structures
- Stored functions, procedures, and packages
- Transaction scope
- Database triggers

PL/SQL is a case-insensitive programming language, like SQL. While the language is case insensitive, there are many conventions applied to how people write their code. Most choose combinations of uppercase, lowercase, title case, or mixed case. Among these opinions there is no standard approach to follow.

PL/SQL Standard Usage for This Book
The PL/SQL code in this book uses uppercase for command words and lowercase for variables, column names, and stored program calls.

Oracle PL/SQL Block Structure

PL/SQL was developed by modeling concepts of structured programming, static data typing, modularity, and exception management. It extends the ADA programming language. ADA extended the Pascal programming language, including the assignment and comparison operators and single-quote string delimiters.

PL/SQL supports two types of programs: one is an anonymous-block program, and the other is a named-block program. Both types of programs have *declaration, execution,* and *exception* handling sections or blocks. Anonymous blocks support batch scripting, and named blocks deliver stored programming units.

The basic prototype for an anonymous-block PL/SQL programs is

```
[DECLARE]
   declaration_statements
 BEGIN
   execution_statements
[EXCEPTION]
   exception_handling_statements
END;
/
```

As shown in the prototype, PL/SQL requires only the execution section for an anonymous-block program. The execution section starts with a `BEGIN` statement and stops at the beginning of the optional `EXCEPTION` block or the `END` statement of the program. *A semicolon ends the anonymous PL/SQL block, and the forward slash executes the block.*

Declaration sections can contain variable definitions and declarations, user-defined PL/SQL type definitions, cursor definitions, reference cursor definitions, and local function or procedure definitions. *Execution* sections can contain variable assignments, object initializations, conditional structures, iterative structures, nested anonymous PL/SQL blocks, or calls to local or stored named PL/SQL blocks. *Exception* sections can contain error handling phrases that can use all of the same items as the execution section.

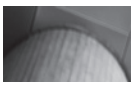
The simplest PL/SQL block does nothing. You must have a minimum of one statement inside any execution block, even if it's a `NULL` statement. The forward slash executes an anonymous PL/SQL block. The following illustrates the most basic anonymous-block program:

```
BEGIN
   NULL;
END;
/
```

You must enable the SQL*Plus `SERVEROUTPUT` variable to print content to the console. The `hello_world.sql` print a message to the console:

```
-- This is found in hello_world.sql on the publisher's web site.
SET SERVEROUTPUT ON SIZE 1000000
BEGIN
  dbms_output.put_line('Hello World.');
END;
/
```

The SQL*Plus `SERVEROUTPUT` environment variable opens an output buffer, and the `DBMS_OUTPUT.PUT_LINE()` function prints a line of output. All declarations, statements, and blocks are terminated by a semicolon.



NOTE
Every PL/SQL block must contain something, at least a NULL; statement, or it will fail run-time compilation, also known as parsing.

SQL*Plus supports the use of substitution variables in the interactive console, which are prefaced by an ampersand, `&`. Substitution variables are variable-length strings or numbers. You should never assign dynamic values in the declaration block, like substitution variables.

The following program defines a variable and assigns it a value:

```
-- This is found in substitution.sql on the publisher's web site.
DECLARE
  my_var VARCHAR2(30);
BEGIN
  my_var := '&input';
  dbms_output.put_line('Hello '|| my_var );
END;
/
```

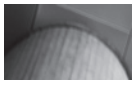
The assignment operator in PL/SQL is a colon plus an equal sign (`:=`). PL/SQL string literals are delimited by single quotes. Date, numeric, and string literals are covered in Chapter 3.

You run anonymous blocks by calling them from Oracle SQL*Plus. The `@` symbol in Oracle SQL*Plus loads and executes a script file. The default file extension is `.sql`, but you can override it with another extension. This means you can call a filename without its `.sql` extension.

The following demonstrates calling the `substitution.sql` file:

```
SQL> @substitution.sql
Enter value for input: Henry Wadsworth Longfellow
old   3:   my_var VARCHAR2(30) := '&input';
new   3:   my_var VARCHAR2(30) := 'Henry Wadsworth Longfellow';
Hello Henry Wadsworth Longfellow
PL/SQL procedure successfully completed.
```

The line starting with `old` designates where your program accepts a substitution, and `new` designates the run-time substitution. Assigning a string literal that is too large for the variable fires an exception. Exception blocks manage raised errors. A generic exception handler manages any raised error. You use a `WHEN` block to catch every raised exception with the generic error handler—`OTHERS`.



TIP
*You can suppress echoing the substitution by setting SQL*Plus VERIFY off.*

The following `exception.sql` program demonstrates how an exception block manages an error when the string is too long for the variable:

```
-- This is found in exception.sql on the publisher's web site.
DECLARE
  my_var VARCHAR2(10);
BEGIN
  my_var := '&input';
  dbms_output.put_line('Hello '|| my_var );
EXCEPTION
  WHEN others THEN
    dbms_output.put_line(SQLERRM);
END;
/
```

The anonymous block changed the definition of the string from 30 characters to 10 characters. The poet's name is now too long to fit in the target variable. Assigning the variable raises an exception. The console output shows the handled and raised exception:

```
SQL> @exception.sql
Enter value for input: Henry Wadsworth Longfellow
old   4:   my_var := '&input';
new   4:   my_var := 'Henry Wadsworth Longfellow';
ORA-06502: PL/SQL: numeric or value error: character string buffer too small
PL/SQL procedure successfully completed.
```

You can also have: (a) nested anonymous-block programs in the *execution* section of an anonymous block; (b) named-block programs in the *declaration* section that can in turn contain the same type of nested programs; and (c) calls to stored named-block programs.

The outermost programming block controls the total program flow, while nested programming blocks control their subordinate programming flow. Each anonymous- or named-block programming unit can contain an *exception* section. When a local exception handler fails to manage an exception, it throws the exception to a containing block until it reaches the SQL*Plus environment.

Error stack management is the same whether errors are thrown from called local or named PL/SQL blocks. Error are raised and put in a *first-in, last-out* queue, which is also known as a stack.

You have explored the basic structure of PL/SQL block programs and error stack management. The block structure is foundational knowledge to work in PL/SQL.

Variables, Assignments, and Operators

Datatypes in PL/SQL include all SQL datatypes and subtypes qualified in Table B-2 of Appendix B. Chapter 3 covers PL/SQL-specific datatypes. PL/SQL also supports scalar and composite variables. Scalar variables hold only one thing, while composite variables hold more than one thing. The preceding programs have demonstrated how you declare and assign values to scalar variables.

Variable names begin with letters and can contain alphabetical characters, ordinal numbers (0 to 9), the `$`, **_**, and **#** symbols. Variables have local scope only. This means they're available only in the scope of a given PL/SQL block. The exceptions to that rule are nested anonymous blocks. Nested anonymous blocks operate inside the defining block. They can thereby access variables from the containing block. That is, unless you've declared the same variable name as something else inside the nested anonymous block.

A declaration of a number variable without an explicit assignment makes its initial value null. The prototype shows that you can assign a value later in the execution block:

```
DECLARE
  variable_name NUMBER;
BEGIN
  variable_name := 1;
END;
/
```

An explicit assignment declares a variable with a not-null value. You can use the default value or assign a new value in the execution block. Both are demonstrated next. You can use an assignment operator or the `DEFAULT` reserved word interchangeably to assign initial values. The following shows a prototype:

```
DECLARE
  variable_name NUMBER [:= | DEFAULT ] 1;
BEGIN
  variable_name := 1;
END;
/
```

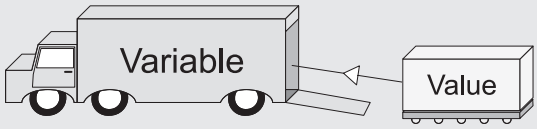
The Assignment Model and Language

All programming languages assign values to variables. They typically assign a value to a variable on the left.

The prototype for generic assignment in any programming language is

```
left_operand assignment_operator right_operand statement_terminator
```

This assigns the right operand to the left operand, as shown here:



You implement it in PL/SQL as follows:

```
left_operand := right_operand;
```

The left operand must always be a variable. The right operand can be a value, variable, or function. Functions must return a value when they're right operands. This is convenient in PL/SQL because all functions return values. Functions in this context are also known as expressions.

The trick is that only functions returning a SQL datatype can be called in SQL statements. Functions returning a PL/SQL datatype only work inside PL/SQL blocks.

Oracle 11*g* performs many implicit casting operations. They fail to follow the common rule of programming: *implicitly cast when there is no loss of precision.* This means you can assign a complex number like 4.67 to an integer and lose the 0.67 portion of the number. Likewise, there are a series of functions to let you *explicitly* cast when there is greater risk of losing precision. You should choose carefully when you *explicitly* downcast variables. Appendix J covers explicit casting functions.

There are also several product-specific datatypes. They support various component products in Oracle 11*g.* You can find these datatypes in the *Oracle Database PL/SQL Packages and Type Reference.*

The assignment operator is not the lone operator in the PL/SQL programming language. Chapter 3 covers all the comparison, concatenation, logical, and mathematical operators. In short, you use

- The equal (=) symbol to check matching values
- The standard greater or less than with or without an equal component (>, >=, <, or <=) as comparison operators to check for inequalities
- The negation (<>, !=, ~= or ^=) comparison operators to check for non-matching values

You define `CURSOR` statements in the declaration section. `CURSOR` statements let you bring data from tables and views into your PL/SQL programs. A `CURSOR` statement can have zero or many formal parameters. `CURSOR` parameters are *pass-by-value,* or `IN` mode–only variables. Chapter 4 covers `CURSOR` statements.

You have now reviewed variables, assignments, and operators. You have also been exposed to PL/SQL-specific user-defined types.

Control Structures

Control structures do two things. They check a logical condition and branch program execution, or they iterate over a condition until it is met or they are instructed to exit. The conditional structures section covers *if, elsif, else,* and *case* statements. The later section "Iterative Structures" covers looping with for and while structures.

Conditional Structures

Conditional statements check whether a value meets a condition before taking action. There are two types of conditional structures in PL/SQL. One is the `IF` statement, and the other is the `CASE` statement. The `IF` statement that has two subtypes, *if-then-else* and *if-then-elsif-then-else.* **The *elsif* is not a typo** but the correct reserved word in PL/SQL. This is another legacy from Pascal and ADA.

IF Statement

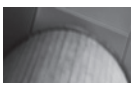
All `IF` statements are blocks in PL/SQL and end with the `END IF` phrase. `CASE` statements are also blocks that end with `END CASE` phrase. Semicolons follow the ending phrases and terminate all blocks in PL/SQL. The following is the basic prototype for an if-then-else PL/SQL block:

```
IF [NOT] left_operand1 = right_operand1 [[AND|OR]
   [NOT] left_operand2 = right_operand2 [[AND|OR]
   [NOT] boolean_operand ]] THEN
  NULL;
ELSE
  NULL;
END IF;
```

The foregoing if-then-else block prototype uses an equality comparison, but you can substitute any of the comparison operators for the equal symbol. You can also evaluate one or more conditions by using `AND` or `OR` to join statements. Boolean outcomes then apply to the combination of expressions. You can negate single or combined outcomes with the `NOT` operator.

Logical operators support conjoining and including operations. A conjoining operator, `AND`, means that *both statements must evaluate as true or false.* An include operator, `OR`, means that *one or the other must be true.* Include operators stop processing when one statement evaluates as true. Conjoining operators check that all statements evaluate as true.

`BOOLEAN` variables are comparisons in and of themselves. Other operands can be any valid datatype that works with the appropriate comparison operator, but remember variables must be initialized. Problems occur when you fail to initialize or handle non-initialized variables in statements.



TIP
You can check if a `BOOLEAN` *value is true by using a comparison operator and constant (like* `some_boolean = TRUE`*), but it isn't the best way to use a Boolean variable in a comparison operation.*

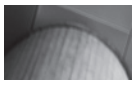
For example, when you use an IF statement to evaluate a non-initialized `BOOLEAN` as *true,* it fails and processes the `ELSE` block; however, when you use an `IF NOT` statement to evaluate a non-initialized `BOOLEAN` as *false,* it also fails and processes the `ELSE` block. This happens because a non-initialized `BOOLEAN` variable isn't *true* or *false.*

The solution to this problem is to use the SQL `NVL()` function. It lets you substitute a value for any *null* value variables. The `NVL()` function takes two parameters: the first is a variable, and the second is a literal, which can be a numeric, string, or constant value. The two parameters must share the same datatype. *You can access all the standard SQL functions natively in your PL/SQL programs.* The following program demonstrates how you use the `NVL()` against a non-initialized `BOOLEAN` variable:

```
-- This is found in if_then.sql on the publisher's web site.
DECLARE
  -- Define a Boolean variable.
  my_var BOOLEAN;
BEGIN
  -- Use an NVL function to substitute a value for evaluation.
  IF NOT NVL(my_var,FALSE) THEN
    dbms_output.put_line('This should happen!');
  ELSE
    dbms_output.put_line('This can''t happen!');
  END IF;
END;
/
```

The `IF NOT` statement would return *false* when the `BOOLEAN` variable isn't initialized. The preceding program finds the `NVL()` function value to be *false,* or `NOT` *true,* and it prints the following message:

```
This should happen!
```



NOTE
The `ELSE` *block contains a backquoted string. The single quote mark is a reserved character for delimiting strings. You backquote an apostrophe by using another apostrophe, or a single quote, inside a delimited string. You can also substitute another backquoting character, as covered in the Oracle 10g recent features section.*

The if-then-*elsif*-then-else statement works like the if-then-else statement but lets you perform multiple conditional evaluations in the same `IF` statement. The following is the basic prototype for an if-then-*elsif*-then-else PL/SQL block:

```
IF    [NOT] left_operand1 > right_operand2 [AND|OR]
  NULL;
ELSIF [NOT] left_operand1 = right_operand1 [[AND|OR]
      [NOT] left_operand2 = right_operand2 [[AND|OR]
      [NOT] boolean_operand ]] THEN
  NULL;
ELSE
  NULL;
END IF;
```

CASE Statement

The other conditional statement is a `CASE` statement. A `CASE` statement works like the if-then-*elsif*-then-else process. There are two types of `CASE` statements: one is a *simple* `CASE`, and the

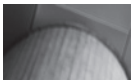
other is a *searched* CASE. A simple CASE statement takes a scalar variable as an expression and then evaluates it against a list of like scalar results. A searched CASE statement takes a BOOLEAN variable as an expression and then compares the Boolean state of the WHEN clause *results as an expression.*

The following is the generic prototype of the CASE statement:

```
CASE [ TRUE | [selector_variable]]
  WHEN [criterion1 | expression1] THEN
    criterion1_statements;
  WHEN [criterion2 | expression2] THEN
    criterion2_statements;
  WHEN [criterion(n+1) | expression(n+1)] THEN
    criterion(n+1)_statements;
  ELSE
    block_statements;
END CASE;
```

The next program demonstrates a searched CASE statement:

```
BEGIN
  CASE TRUE
  WHEN (1 > 3) THEN
    dbms_output.put_line('One is greater than three.');
  WHEN (3 < 5) THEN
    dbms_output.put_line('Three is less than five.');
  WHEN (1 = 2) THEN
    dbms_output.put_line('One equals two.');
  ELSE
    dbms_output.put_line('Nothing worked.');
  END CASE;
END;
/
```



TIP
You can leave TRUE *out (because it is the default selector), but don't. Putting it in adds clarity.*

The program evaluates WHEN clause results as expressions, finding that 3 is less than 5. It then prints

```
Three is less than five.
```

You can find out more about CASE statements in Chapter 4. This subsection has demonstrated the conditional expressions available to you in PL/SQL. It has also suggested an alternative to non-initialized variables.

Iterative Structures

PL/SQL supports FOR, SIMPLE, and WHILE loops. There's no syntax for a *repeat until* loop block, but you can still perform one. Loops typically work in conjunction with cursors but can work to solve other problems, like searching or managing Oracle collections.

FOR Loops

PL/SQL supports *numeric* and *cursor* `FOR` loops. The *numeric* `FOR` loop iterates across a defined range, while the *cursor* `FOR` loop iterates across rows returned by a `SELECT` statement cursor. `FOR` loops manage how they begin and end implicitly. You can override the *implicit* `END LOOP` phrase by using an explicit `CONTINUE` or `EXIT` statement to respectively skip an iteration or force a premature exit from the loop.

Numeric `FOR` loops take two implicit actions. They automatically declare and manager their own loop index, and they create and manage their exit from the loop. A *numeric* `FOR` loop has the following prototype:

```
FOR i IN starting_number..ending_number LOOP
  statement;
END LOOP;
```

The *starting_number* and *ending_number* must be integers. The *loop index* is the `i` variable, and the loop index scope is limited to the `FOR` loop. The index variable is a `PLS_INTEGER` datatype number. When you have previously defined or declared a variable `i`, the numeric loop will ignore the externally scoped variable and create a new locally scoped variable.

The following sample program prints index values from 1 to 10:

```
BEGIN
  FOR i IN 1..10 LOOP
    dbms_output.put_line('The index value is ['||i||']');
  END LOOP;
END;
/
```

The *cursor* `FOR` loop requires a locally defined `CURSOR`. You cannot use a *cursor* `FOR` loop to iterate across a reference cursor (`REF CURSOR`) because *reference cursors can only be traversed by using explicit loop structures, like simple and while loops.* The cursor `FOR` loop can also use a `SELECT` statement in lieu of a locally defined `CURSOR`, and has the following prototype:

```
FOR i IN {cursor_name[(parameter1,parameter(n+1))] | (sql_statement)} LOOP
  statement;
END LOOP;
```

The *cursor_name* can have an optional parameter list, which is enclosed in parentheses. A *cursor_name* without optional parameters does not require parentheses. You are using an *explicit* cursor when you call a *cursor_name* and an *implicit* cursor when you provide a `SELECT` statement.

The following demonstrates how you write an explicit cursor in a `FOR` loop and use the data seeded by the downloadable scripts:

```
DECLARE
  CURSOR c IS SELECT item_title FROM item;
BEGIN
  FOR i IN c LOOP
    dbms_output.put_line('The title is ['||i.item_title||']');
  END LOOP;
END;
/
```

The following demonstrates how you write an implicit cursor in a `FOR` loop and use the data seeded by the downloadable scripts:

```
BEGIN
  FOR i IN (SELECT item_title FROM item) LOOP
    dbms_output.put_line('The title is ['||i.item_title||']');
  END LOOP;
END;
/
```

The index variable is not a PLS_INTEGER number in a cursor FOR loop. It is a reference to the record structure returned by the cursor. You combine the cursor index variable and column name with a dot, also known as the component selector. In this case, the cursor index is the component. The component selector lets you select a column from the row returned by the cursor.

The *statement* must be a valid SELECT statement, but you can dynamically reference locally scoped variable names without any special syntax in all clauses except the FROM clause. Unless you override the exit criteria, the *cursor* FOR loop will run through all rows returned by the *cursor* or *statement.*

Simple Loops

Simple loops are explicit structures. They require that you manage both loop index and exit criteria. Typically, simple loops are used in conjunction with locally defined cursor statements and reference cursors (REF CURSOR).

Oracle provides six cursor attributes that help you manage activities in loops. The four cursor attributes are: %FOUND, %NOTFOUND, %ISOPEN, and %ROWCOUNT. Two others support bulk operations. They are all covered in Chapter 4.

The simple loops have a variety of uses. The following is the prototype for a simple loop, using an explicit CURSOR:

```
OPEN cursor_name [(parameter1,parameter(n+1))];
LOOP
  FETCH cursor_name
  INTO  row_structure_variable | column_variable1 [,column_variable(n+1)];
    EXIT WHEN cursor_name%NOTFOUND;
    statement;
END LOOP;
CLOSE cursor_name;
```

The prototype demonstrates that you OPEN a CURSOR before starting the simple loop, and then you FETCH a row. While rows are returned you process them, but when a FETCH fails to return a row, you exit the loop. Place the EXIT WHEN statement as the last statement in the loop when you want the behavior of a repeat until loop. Repeat until loops typically process statements in a loop at least once regardless of whether the CURSOR returns records.

The following mimics the cursor FOR loop against the ITEM table:

```
DECLARE
  title item.item_title%TYPE;
  CURSOR c IS SELECT item_title FROM item;
BEGIN
  OPEN c;
  LOOP
    FETCH c INTO title;
    EXIT WHEN c%NOTFOUND;
    dbms_output.put_line('The title is ['||title||']');
```

```
  END LOOP;
  CLOSE c;
END;
/
```

WHILE Loops

The WHILE loop differs from the simple loop because it guards entry to the loop, not exit. It sets the entry guard as a precondition expression. The loop is only entered when the guard condition is met. The basic syntax is

```
OPEN cursor_name [(parameter1,parameter(n+1))];
WHILE condition LOOP
  FETCH cursor_name
  INTO  row_structure_variable | column_variable1 [,column_variable(n+1)];
    EXIT WHEN cursor_name%NOTFOUND;
    statement;
END LOOP;
CLOSE cursor_name;
```

When the condition checks for an opened CURSOR, then the WHILE *condition* would be *cursor_name*%ISOPEN. There are many other possible condition values that you can use in WHILE loops. The following code demonstrates how you can use a cursor %ISOPEN attribute as the guard on entry condition:

```
DECLARE
  title item.item_title%TYPE;
  CURSOR c IS SELECT item_title FROM item;
BEGIN
  OPEN c;
  WHILE c%ISOPEN LOOP
    FETCH c INTO title;
    IF c%NOTFOUND THEN
      CLOSE c;
    END IF;
    dbms_output.put_line('The title is ['||title||']');
  END LOOP;
END;
/
```

The WHILE condition is true only until the IF statement closes the cursor inside the loop. You should note that repeating instructions come after the IF statement.

This section has demonstrated how you can use implicit and explicit looping structures. It has also introduced you to the management of the CURSOR statement in the execution section of PL/SQL programs. Chapter 4 covers the CONTINUE and GOTO statements.

Stored Functions, Procedures, and Packages

PL/SQL stored programming units are typically functions, procedures, packages, and triggers. You can also store object types, but that discussion is in Chapter 14.

Oracle maintains a unique list of stored object names for tables, views, sequences, stored programs, and types. This list is known as a namespace. Functions, procedures, packages, and objects are in this namespace. Another namespace stores triggers.

Stored functions, procedures, and packages provide a way to hide implementation details in a program unit. They also let you wrap the implementation from prying eyes on the server tier.

Stored Functions

Stored functions are convenient structures because you can call them directly from SQL statements or PL/SQL programs. All stored functions must return a value. You can also use them as right operands because they return a value. Functions are defined in local declaration blocks or the database. You frequently implement them inside stored packages.

The prototype for a stored *function* is

```
FUNCTION function_name
[( parameter1     [IN][OUT] [NOCOPY] sql_datatype | plsql_datatype
[, parameter2     [IN][OUT] [NOCOPY] sql_datatype | plsql_datatype
[, parameter(n+1) [IN][OUT] [NOCOPY] sql_datatype | plsql_datatype )]]]
RETURN [ sql_data_type | plsql_data_type ]
[ AUTHID [ DEFINER | CURRENT_USER ]]
[ DETERMINISTIC | PARALLEL_ENABLED ]
[ PIPELINED ]
[ RESULT_CACHE [ RELIES ON table_name ]] IS
  declaration_statements
BEGIN
  execution_statements
  RETURN variable;
[EXCEPTION]
  exception_handling_statements
END [function_name];
/
```

Functions can be used as right operands in PL/SQL assignments. You can also call them directly from SQL statements, provided they return a SQL datatype. Procedures cannot be right operands. Nor can you call them from SQL statements.

You can query a function that returns a SQL datatype by using the following prototype from the pseudotable `DUAL`:

```
SELECT   some_function[(actual_parameter1, actual_parameter2)]
FROM     dual;
```

You are *no longer limited to passing actual parameters by positional order* in SQL statements. This means that you can used PL/SQL named notation in SQL. Chapter 6 covers how named, positional, and mixed notation work.

The following is a *named notation* prototype for the same query of a PL/SQL function from the pseudotable `DUAL`:

```
SELECT   some_function[(formal_parameter => actual_parameter2)]
FROM     dual;
```

Named positional calls work best when default values exist for other parameters. There isn't much purpose in calling only some of the parameters when the call would fail. Formal parameters are optional parameters. Named positional calls work best with functions or procedures that have optional parameters.

You can also use the CALL statement to capture a return value from a function into a bind variable. The prototype for the CALL statement follows:

```
CALL some_function[(actual_parameter1, actual_parameter2)]
INTO some_session_bind_variable;
```

The following is a small sample case that concatenates two strings into one:

```
-- This is found in join_strings.sql on the publisher's web site.
CREATE OR REPLACE FUNCTION join_strings
( string1 VARCHAR2
, string2 VARCHAR2 ) RETURN VARCHAR2 IS
BEGIN
  RETURN string1 ||' '|| string2||'.';
END;
/
```

You can now query the function from SQL:

```
SELECT join_strings('Hello','World') FROM dual;
```

Likewise, you can define a session-level bind variable and then use the CALL statement to put a variable into a session-level bind variable:

```
VARIABLE session_var VARCHAR2(30)
CALL join_strings('Hello','World') INTO :session_var;
```

The CALL statement uses an INTO clause when working with stored functions. You dispense with the INTO clause when working with stored procedures.

Selecting the bind variable from the pseudo–DUAL table, like this

```
SELECT :session_var FROM dual;
```

you'll see

```
Hello World.
```

Functions offer a great deal of power to database developers. They are callable in both SQL statements and PL/SQL blocks.

Procedures

Procedures cannot be right operands. Nor can you use them in SQL statements. You move data into and out of PL/SQL stored procedures through their formal parameter list. As with stored functions, you can also define local named-block programs in the declaration section of procedures.

The prototype for a stored *procedure* is

```
PROCEDURE procedure_name
[( parameter1     [IN][OUT] [NOCOPY] sql_datatype | plsql_datatype
[, parameter2     [IN][OUT] [NOCOPY] sql_datatype | plsql_datatype
```

```
[, parameter(n+1) [IN][OUT] [NOCOPY] sql_datatype | plsql_datatype )]]]
[ AUTHID DEFINER | CURRENT_USER ] IS
  declaration_statements
BEGIN
  execution_statements
[EXCEPTION]
  exception_handling_statements
END [procedure_name];
/
```

You can define procedures with or without formal parameters. Formal parameters in procedures can be either *pass-by-value* or *pass-by-reference* variables in stored procedures. *Pass-by-reference* variables have both and `IN` and `OUT` modes. As in the case of functions, when you don't provide a parameter mode, the procedure creation assumes you want the mode to be a *pass-by-value.*

Procedures can't be used as right operands in PL/SQL assignments, nor called directly from SQL statements. The following implements a stored procedure that uses a *pass-by-reference* semantic to enclose a string in square brackets:

```
-- This is found in format_string.sql on the publisher's web site.
CREATE OR REPLACE PROCEDURE format_string
( string_in IN OUT VARCHAR2 ) IS
BEGIN
  string_in := '['||string_in||']';
END;
/
```

You can also use the `CALL` statement to call and pass variables into and out of a procedure. Like the earlier function example, this example uses the `CALL` statement and bind variable:

```
VARIABLE session_var VARCHAR2(30)
CALL join_strings('Hello','World') INTO :session_var;
CALL format_string(:session_var);
```

The first `CALL` statement calls the previously introduced function and populates the `:session_var` variable. You should note that the second `CALL` statement does not use an `INTO` clause when passing a variable into and out of a stored procedure. This differs from how it works with stored functions.

You also can use the `EXECUTE` statement with stored procedures. The following works exactly like the `CALL` statement:

```
EXECUTE format_string(:session_var);
```

When you select the bind variable from the pseudo–`DUAL` table,

```
SELECT :session_var FROM dual;
```

you'll see

```
[Hello World.]
```

unless you ran both examples, which means you'll see double brackets:

```
[[Hello World.]]
```

Procedures offer you the ability to use pass-by-value or pass-by-reference formal parameters. As you'll see in Chapters 6 and 16, stored procedures let you exchange values with external applications.

Packages

Packages are the backbone of stored programs in Oracle 11*g*. They act like libraries and are composed of functions and procedures. Unlike standalone functions and procedures, packages let you create overloaded functions and procedures. Chapter 9 covers these features of packages.

Packages have a published specification. The specification avoids single parser limitations because all functions and procedures are published. Publishing acts like forward referencing for local functions and procedures. Package bodies contain the hidden details of the functions and procedures rather than their defined signature.

Package bodies must mirror the function and procedure signatures provided in the package specifications. Package bodies may also contain locally defined types, functions, and procedures. These structures are only available inside the package body. They mimic the concept of private access variables in other modern programming languages, like C++ and Java.

Transaction Scope

Transaction scope is a thread of execution—a process. You establish a session when you connect to the Oracle 11*g* database. The session lets you set environment variables, like `SERVEROUTPUT`, which lets you print from your PL/SQL programs. What you do during your session is visible only to you until you commit the work. After you commit the changes, other sessions can see the changes you've made.

During a session, you can run one or more PL/SQL programs. They execute serially, or in sequence. The first program can alter the data or environment before the second runs, and so on. This is true because your session is the main transaction. All activities potentially depend on all the prior activities. You can commit work, making all changes permanent, or roll back to reject work, repudiating all or some changes.

The power to control the session rests with three commands. They were once called transaction control language (TCL) commands. Some documentation now speaks of them as data control language (DCL) commands. The book uses DCL to represent these three commands. The problem is trying to disambiguate this group of commands from Berkeley's Tcl. The commands are

- **The `COMMIT` statement** Commits all DML changes made from the beginning of the session or since the last `ROLLBACK` statement.
- **The `SAVEPOINT` statement** Divides two epochs. An epoch is defined by the transactions between two relative points of time. A `SAVEPOINT` delimits two epochs.
- **The `ROLLBACK` statement** Undoes all changes from now to an epoch or named `SAVEPOINT`, or now to the beginning of a SQL*Plus session.

These commands let you control what happens in your session and program routines. The beginning of a session is both the beginning of an epoch and an implicit `SAVEPOINT` statement. Likewise, the ending of a session is the ending of an epoch and implicit `COMMIT` statement.

How you manage transaction scope differs between a single transaction scope and multiple transaction scopes. You create multiple transaction scopes when a function or procedure is designated as an autonomous stored program unit.

Single Transaction Scope

A common business problem involves guaranteeing the sequential behavior of two or more DML statements. The idea is that they all must either succeed or fail. Partial success is not an option. DCL commands let you guarantee the behavior of sequential activities in a single transaction scope.

The following program uses DCL commands to guarantee both `INSERT` statements succeed or fail:

```
-- This is found in transaction_scope.sql on the publisher's web site.
BEGIN
  -- Set savepoint.
  SAVEPOINT new_member;
  -- First insert.
  INSERT INTO member VALUES
  ( member_s1.nextval, 1005,'D921-71998','4444-3333-3333-4444', 1006
  , 2, SYSDATE, 2, SYSDATE);

  -- Second insert.
  INSERT INTO contact VALUES
  ( contact_s1.nextval, member_s1.currval + 1, 1003
  ,'Bodwin','Jordan',''
  , 2, SYSDATE, 2, SYSDATE);
  -- Print success message and commit records.
  dbms_output.put_line('Both succeeded.');
  COMMIT;
EXCEPTION
  WHEN others THEN
    -- Roll back to savepoint, and raise exception message.
    ROLLBACK TO new_member;
    dbms_output.put_line(SQLERRM);
END;
/
```

The second `INSERT` statement fails because the foreign key constraint on `member_id` in the `member` table isn't met. The failure triggers an Oracle exception and shifts control to the exception block. The first thing the exception block does is roll back to the initial `SAVEPOINT` statement set by the anonymous-block program.

Multiple Transaction Scopes

Some business problems require that programs work independently. Independent programs run in discrete transaction scopes. When you call an autonomous program unit, it runs in another transaction scope.

You can build autonomous programs with the `AUTONOMOUS_TRANSACTION` precomplier instruction. A precompiler instruction is a `PRAGMA` and sets a specific behavior, like independent transaction scope. Only the following types of programs can be designated as autonomous routines:

- Top-level (not nested) anonymous blocks
- Local, standalone, package subroutines—functions and procedures
- Methods of the SQL object type
- Database triggers

The beginning transaction scope is known as the main routine. It calls an autonomous routine, which then spawns its own transaction scope. A failure in the main routine after calling an autonomous program can only roll back changes made in the main transaction scope. The autonomous transaction scope can succeed or fail independently of the main routine. However, the main routine can also fail when an exception is raised in an autonomous transaction.

Chapter 5 includes an example of this type of parallel activity. The primary `INSERT` statement fails because of activities in an autonomous database trigger. When the event fires the autonomous trigger, it writes the attempt to an error table, commits the write, and then raises an exception. The trigger exception causes the original `INSERT` statement to fail.

Multiple transaction scope programs are complex. You should be sure the benefits outweigh the risk when using multiple transaction scope solutions.

Database Triggers

Database *triggers* are specialized stored programs that are triggered by events in the database. They run between when you issue a command and when you perform the database management system action. Because they come in between, you cannot use SQL Data Control Language in triggers: `SAVEPOINT`, `ROLLBACK`, or `COMMIT`. You can define five types of triggers in the Oracle Database 11*g* family of products:

- *Data Definition Language (DDL) triggers* These triggers fire when you *create, alter, rename,* or *drop* objects in a database schema. They are useful to monitor poor programming practices, such as when programs *create* and *drop* temporary tables rather than use Oracle collections effectively in memory. Temporary tables can fragment disk space and over time degrade the database performance.

- *Data Manipulation Language (DML) or row-level triggers* These triggers fire when you *insert, update,* or *delete* data from a table. You can use these types of triggers to audit, check, save, and replace values before they are changed. Automatic numbering of pseudonumeric *primary keys* is frequently done by using a DML trigger.

- *Compound triggers* These triggers act as both statement- and row-level triggers when you *insert, update,* or *delete* data from a table. These triggers let you capture information at four timing points: (a) before the firing statement; (b) before each row change from the firing statement; (c) after each row change from the firing statement; and (d) after the firing statement. You can use these types of triggers to audit, check, save, and replace values before they are changed when you need to take action at both the statement and row event levels.

- *Instead of triggers* These triggers enable you to stop performance of a DML statement and redirect the DML statement. `INSTEAD OF` triggers are often used to manage how you write to views that disable a direct write because they're not updatable views. The `INSTEAD OF` triggers apply business rules and directly *insert, update,* or *delete* rows in appropriate tables related to these updatable views.

- *System or database event triggers* These triggers fire when a system activity occurs in the database, like the logon and logoff event triggers used in Chapter 13. These triggers enable you to track system events and map them to users.

We will cover all five trigger types in Chapter 10.

Summary

This chapter has reviewed the Procedural Language/Structured Query Language (PL/SQL) basics and explained how to jump-start your PL/SQL skills. The coverage should serve to whet your appetite for more.