CHAPTER 3
Language Fundamentals

Every language—whether human or computer—has a syntax, a vocabulary, and a character set. In order to communicate within that language, you have to learn the rules that govern its usage. Many of us are wary of learning a new computer language. Change is often scary, but in general programming languages are very simple, and PL/SQL is no exception. The difficulty of conversing in languages based on bytes is not with the language itself, but with the compiler or computer with which we are having the discussion. Compilers are, for the most part, rather dull-witted. They are not creative, sentient beings. They are not capable of original thought. Their vocabulary is severely limited. Compilers just happen to think their dull thoughts very, very rapidly—and very inflexibly.

If I hear someone ask "gottabuck?" I can readily interpret that sentence and decide how to respond. On the other hand, if I instruct PL/SQL to "gimme the next half-dozen records," I will not get very far in my application. To use the PL/SQL language, you must dot your *i*'s and cross your *t*'s—syntactically speaking. So, this chapter covers the fundamental language rules that will help you converse with the PL/SQL compiler—the PL/SQL block structure, character set, lexical units, and PRAGMA keyword.

PL/SQL Block Structure

In PL/SQL, as in most other procedural languages, the smallest meaningful grouping of code is known as a *block*. A block is a unit of code that provides execution and scoping boundaries for variable declarations and exception handling. PL/SQL allows you to create *anonymous blocks* (blocks of code that have no name) and *named blocks*, which may be packages, procedures, functions, triggers, or object types.

A PL/SQL block has up to four different sections, only one of which is mandatory:

Header
: Used only for named blocks. The header determines the way the named block or program must be called. Optional.

Declaration section
: Identifies variables, cursors, and subblocks that are referenced in the execution and exception sections. Optional.

Execution section
: Contains statements the PL/SQL runtime engine will execute at runtime. Mandatory.

Exception section
: Handles exceptions to normal processing (warnings and error conditions). Optional.

Figure 3-1 shows the structure of the PL/SQL block for a procedure.

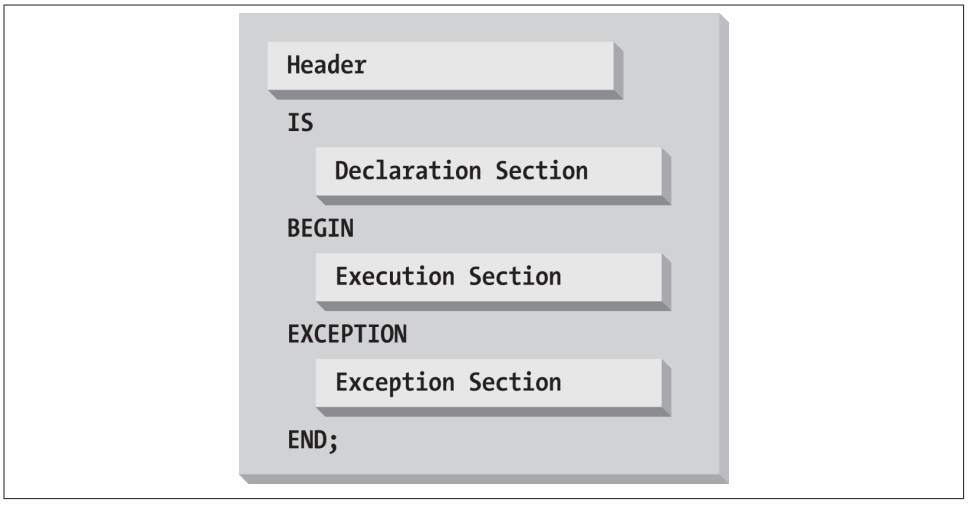

Figure 3-1. The PL/SQL block structure

Figure 3-2 shows a procedure containing all four sections of the elements of a block. This particular block begins with the keyword PROCEDURE, and, like all blocks, ends with the keyword END.

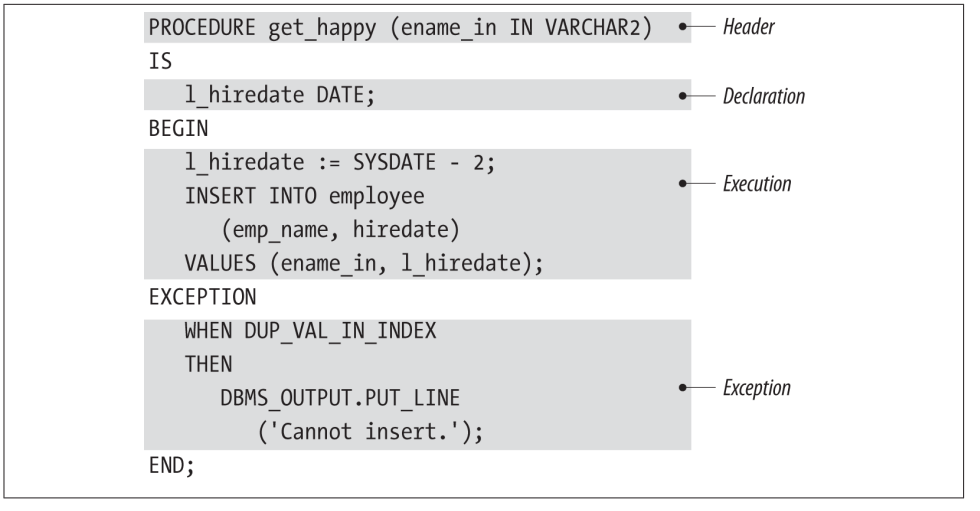

Figure 3-2. A procedure containing all four sections

Anonymous Blocks

When someone wishes to remain anonymous, that person goes unnamed. It's the same with the anonymous block in PL/SQL, which is shown in Figure 3-3: it lacks a header section altogether, beginning instead with either DECLARE or BEGIN. That means that it cannot be called by any other block—it doesn't have a handle for reference. Instead, anonymous blocks serve as containers that execute PL/SQL statements, usually including calls to procedures and functions. Because an anonymous block can have its own declaration and exception sections, developers often nest anonymous blocks to provide a scope for identifiers and exception handling within a larger program.

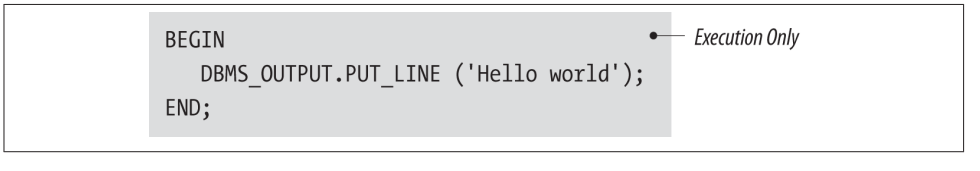

Figure 3-3. An anonymous block without declaration and exception sections

The general syntax of an anonymous PL/SQL block is as follows:

```
[ DECLARE   ... declaration statements ... ]
BEGIN   ... one or more executable statements ...
[ EXCEPTION
   ... exception handler statements ... ]
END;
```

The square brackets indicate an optional part of the syntax. You must have BEGIN and END statements, and you must have at least one executable statement. Here are a few examples:

- A bare minimum anonymous block:

```
BEGIN
   DBMS_OUTPUT.PUT_LINE(SYSDATE);
END;
```

- A functionally similar block, adding a declaration section:

```
DECLARE
   l_right_now VARCHAR2(9);
BEGIN
   l_right_now := SYSDATE;
   DBMS_OUTPUT.PUT_LINE (l_right_now);
END;
```

- The same block, but including an exception handler:

```
DECLARE
   l_right_now VARCHAR2(9);
BEGIN
   l_right_now := SYSDATE;
   DBMS_OUTPUT.PUT_LINE (l_right_now);
EXCEPTION
   WHEN VALUE_ERROR
   THEN
      DBMS_OUTPUT.PUT_LINE('I bet l_right_now is too small '
         || 'for the default date format!');
END;
```

Anonymous blocks execute a series of statements and then terminate, thus acting like procedures. In fact, all anonymous blocks are anonymous procedures. They are used in various environments where PL/SQL code is either executed directly or enclosed in some program in that environment. Common examples include:

Database triggers
: As discussed in Chapter 19, database triggers execute anonymous blocks when certain events occur.

Ad hoc commands or script files
: In SQL*Plus or similar execution environments, anonymous blocks run from hand-entered blocks or from scripts that call stored programs. Also, the SQL*Plus EXECUTE command translates its argument into an anonymous block by enclosing it between BEGIN and END statements.

Compiled 3GL (third generation language) program
: In Pro*C or OCI, anonymous blocks can be the means by which you can embed calls to stored programs.

In each case, the enclosing object—whether it's a trigger, a command-line environment, or a compiled program—provides the context and possibly a means of naming the program.

Named Blocks

While anonymous PL/SQL blocks are indispensable, the majority of code you write will be in named blocks. You've seen a few short examples of stored procedures in this book already (as in Figure 3-1), so you probably know that the difference is in the header. A procedure header looks like this:

```
PROCEDURE [schema.]name [ ( parameter [, parameter ... ] ) ]
   [AUTHID {DEFINER | CURRENT_USER}]
```

A function header has similar syntax, but includes the RETURN keyword:

```
FUNCTION [schema.]name [ ( parameter [, parameter ... ] ) ]
   RETURN return_datatype
   [AUTHID {DEFINER | CURRENT_USER}]
   [DETERMINISTIC]
   [PARALLEL ENABLE ...]
   [PIPELINED [USING...] | AGGREGATE USING...]
```

Because Oracle allows you to invoke some functions from within SQL statements, the function header includes more optional components than the procedure header, corresponding to the functionality and performance dimensions of the SQL runtime environment.

For a more complete discussion of procedures and functions, see Chapter 17.

Nested Blocks

PL/SQL shares with Ada and Pascal the additional definition of being a *block-structured language*—that is, blocks may "nest" within other blocks. In contrast, the C language has blocks, but standard C isn't strictly block-structured, because its subprograms cannot be nested.

Here's a PL/SQL example showing a procedure containing an anonymous, nested block:

```
PROCEDURE calc_totals
IS
   year_total NUMBER;
BEGIN
   year_total := 0;

   /* Beginning of nested block */
   DECLARE
      month_total NUMBER;
   BEGIN
      month_total := year_total / 12;
```

```
      END set_month_total;
      /* End of nested block */

END;
```

The /* and */ delimiters indicate comments (see "Comments" on page 75). You can nest anonymous blocks within anonymous blocks to more than one level, as shown in Figure 3-4.

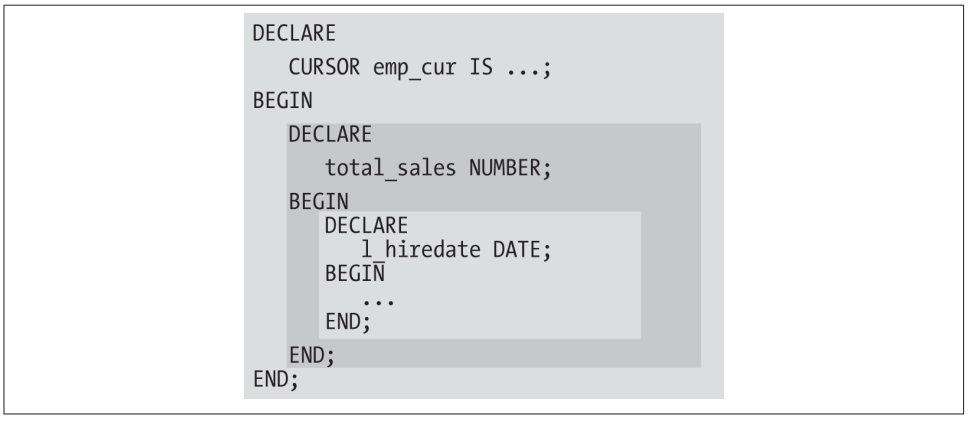

Figure 3-4. Anonymous blocks nested three levels deep

Other terms you may hear for nested block are *enclosed block*, *child block*, or *subblock*; the outer PL/SQL block may be called the *enclosing block* or the *parent block*.

In general, the advantage of nesting a block is that it gives you a way to control both scope and visibility in your code.

Scope

In any programming language, the term *scope* refers to the way of identifying which "thing" is referred to by a given identifier. If you have more than one occurrence of an identifier, the language's scoping rules define which one will be used. Carefully controlling identifier scope not only will increase your control over runtime behavior but also will reduce the likelihood of a programmer accidentally modifying the wrong variable.

In PL/SQL, variables, exceptions, modules, and a few other structures are local to the block that declares them. When the block stops executing, you can no longer reference any of these structures. For example, in the earlier calc_totals procedure, I can reference elements from the outer block, like the year_total variable, anywhere in the procedure; however, elements declared within an inner block are not available to the outer block.

Every PL/SQL variable has a scope: the region of a program unit (block, subprogram, or package) in which that variable can be referenced. Consider the following package definition:

```
PACKAGE scope_demo
IS
   g_global   NUMBER;

   PROCEDURE set_global (number_in IN NUMBER);
END scope_demo;

PACKAGE BODY scope_demo
IS
   PROCEDURE set_global (number_in IN NUMBER)
   IS
      l_salary  NUMBER := 10000;
      l_count   PLS_INTEGER;
   BEGIN

      <<local_block>>
      DECLARE
         l_inner   NUMBER;
      BEGIN
         SELECT COUNT (*)
           INTO l_count
           FROM employees
          WHERE department_id = l_inner AND salary > l_salary;
      END local_block;

      g_global := number_in;
   END set_global;
END scope_demo;
```

The scope_demo.g_global variable can be referenced from any block in any schema that has EXECUTE authority on scope_demo.

The l_salary variable can be referenced only inside the set_global procedure.

The l_inner variable can be referenced only inside the local or nested block; note that I have used the label "local_block" to give a name to that nested block.

Qualify All References to Variables and Columns in SQL Statements

None of the variables or column references in the last code example were *qualified* with the scope name. Here is another version of the same package body, but this time with qualified references (in bold):

```
PACKAGE BODY scope_demo
IS
   PROCEDURE set_global (number_in IN NUMBER)
   IS
      l_salary  NUMBER := 10000;
```

```
      l_count   PLS_INTEGER;
   BEGIN

      <<local_block>>
      DECLARE
         l_inner   PLS_INTEGER;
      BEGIN
         SELECT COUNT (*)
           INTO set_global.l_count
           FROM employees e
          WHERE e.department_id = local_block.l_inner
            AND e.salary > set_global.l_salary;
      END local_block;

      scope_demo.g_global := set_global.number_in;
   END set_global;
END scope_demo;
```

With these changes, every single reference to a column and variable is qualified by the table alias, the package name, the procedure name, or the nested block label name.

So now you know that you can do this—but why bother? There are several very good reasons:

- To improve readability of your code
- To avoid bugs that can arise when the names of variables are the same as the names of columns
- To take full advantage of the fine-grained dependency tracking made available in Oracle Database 11*g*

Let's take a closer look at the first two of these reasons. I'll describe the third in Chapter 20.

Improve readability

Just about every SQL statement embedded in PL/SQL programs contains references to both columns and variables. In small, simple SQL statements, it is relatively easy to distinguish between these different references. In most applications, however, you will find very long, extremely complex SQL statements that contain dozens or even hundreds of references to columns and variables.

If you do not qualify these references, it is much harder to distinguish at a glance between variables and columns. With these qualifiers, the code self-documents quite clearly the source of those references.

"Wait a minute," I can hear you say. "We use clearly defined naming conventions to distinguish between columns and variables. All our local variables start with 'l_' so we know immediately if the identifier is a local variable."

That is a *really* good idea; we should all have (and follow) established conventions so that the names of our identifiers reveal additional information about them (e.g., is it a parameter or a variable? What is its datatype?).

Yet while helpful, naming conventions are not sufficient to *guarantee* that over time the PL/SQL compiler will *always* interpret your identifiers as you intended.

Avoid bugs through qualifiers

If you do not qualify references to all PL/SQL variables in your embedded SQL statements, code that works correctly today might in the future suddenly *not* work anymore. And it could be very difficult to figure out what went wrong.

Consider again this embedded SQL statement that does not qualify its references:

```
SELECT COUNT (*)
  INTO l_count
  FROM employees
 WHERE department_id = l_inner AND salary > l_salary;
```

Today, l_salary unambiguously refers to the l_salary variable declared in the set_global procedure. I test my program—it works! And then it goes into production and everyone is happy.

Two years go by, and then the users ask our DBA to add a column to the employees table to record something described as "limited salary." The DBA decides to name this column "l_salary".

Can you see the problem?

Within an embedded SQL statement, the Oracle database always attempts to resolve unqualified identifier references first as columns in one of the specified tables. If it cannot find a match, it then tries to resolve the reference as an in-scope PL/SQL variable. With the column l_salary added to the employees table, my unqualified reference to l_salary in the SELECT statement is no longer resolved to the PL/SQL variable. Instead, the database resolves it as the column in the table. The consequence?

My scope_demo package still compiles without any errors, but the WHERE clause of that query is *not* going to behave as I expect. The database will not use the value of the l_salary variable, but will instead compare the salary column's value in a row of the employees table to the value of the l_salary column in that same row.

This could be a *very* tricky bug to track down and fix!

Rather than rely solely on naming conventions to avoid "collisions" between identifiers, you should also qualify references to all column names *and* variables in those embedded SQL statements. Then your code will be much less likely to behave erratically in the future as your underlying tables evolve.

Visibility

Once a variable is in scope, another important property is its *visibility*—that is, whether you can refer to it using only its name, or whether you need to attach a prefix in front of it.

Visible identifiers

First, I'd like to make an observation about the trivial case:

```
DECLARE
   first_day DATE;
   last_day DATE;
BEGIN
   first_day := SYSDATE;
   last_day := ADD_MONTHS (first_day, 6);
END;
```

Because both the first_day and last_day variables are declared in the same block where they are used, I can conveniently refer to them using only their "unqualified" identifiers, which are also known as *visible identifiers*. A visible identifier might actually reference any of the following:

- An identifier declared in the current block
- An identifier declared in a block that encloses the current block
- A standalone database object (table, view, sequence, etc.) or PL/SQL object (procedure, function, type) that you own
- A standalone database object or PL/SQL object on which you have the appropriate privilege and that is the target of an Oracle synonym that you can see
- A loop index variable (visible and in-scope only inside the loop body)

PL/SQL also allows the possibility of referring to in-scope items that are not directly visible, as the next section describes.

Qualified identifiers

A common example of an identifier that isn't visible is anything declared in a package specification, such as a variable, datatype, procedure, or function. To refer to one of these elements outside of that package, you merely need to prefix it with a dotted qualifier, similar to the way you would qualify a column name with the name of its table. For example:

price_util.compute_means
: A program named compute_means inside the price_util package

math.pi
: A constant named pi, declared and initialized in the math package

(Although the descriptions indicate what kinds of globals these are, you can't necessarily tell by looking—definitely an argument in favor of good naming conventions!)

You can use an additional qualifier to indicate the owner of the object. For example:

```
scott.price_util.compute_means
```

could refer to the compute_means procedure in the price_util package owned by the Oracle user account scott.

Qualifying identifier names with module names

When necessary, PL/SQL offers many ways to qualify an identifier so that a reference to the identifier can be resolved. Using packages, for example, you can create variables with global scope. Suppose that I create a package called company_pkg and declare a variable named last_company_id in that package's specification, as follows:

```
PACKAGE company_pkg
IS
   last_company_id NUMBER;
   ...
END company_pkg;
```

Then, I can reference that variable outside of the package, as long as I prefix the identifier name with the package name:

```
IF new_company_id = company_pkg.last_company_id THEN
```

By default, a value assigned to one of these package-level variables persists for the duration of the current database session; it doesn't go out of scope until the session disconnects.

I can also qualify the name of an identifier with the module in which it is defined:

```
PROCEDURE calc_totals
IS
   salary NUMBER;
BEGIN
   ...
   DECLARE
      salary NUMBER;
   BEGIN
      salary := calc_totals.salary;
   END;
   ...
END;
```

The first declaration of salary creates an identifier whose scope is the entire procedure. In the nested block, however, I declare another identifier with the same name. So when I reference the variable salary inside the inner block, it will always be resolved first against the declaration in the inner block, where that variable is visible without any qualification. If I wish to make reference to the procedure-wide salary variable inside

the inner block, I must qualify that variable name with the name of the procedure (cal_totals.salary).

This technique of qualifying an identifier also works in other contexts. Consider what will happen when you run a procedure such as this (order_id is the primary key of the orders table):

```
PROCEDURE remove_order (order_id IN NUMBER)
IS
BEGIN
   DELETE orders WHERE order_id = order_id; -- Oops!
END;
```

This code will delete everything in the orders table regardless of the order_id that you pass in. The reason: SQL's name resolution matches first on column names rather than on PL/SQL identifiers. The WHERE clause "order_id = order_id" is always true, so poof goes your data. One way to fix it would be:

```
PROCEDURE remove_order (order_id IN NUMBER)
IS
BEGIN
   DELETE orders WHERE order_id = remove_order.order_id;
END;
```

This forces the parser to do the right thing. (It will even work if you happen to have a packaged function called remove_order.order_id.)

PL/SQL goes to a lot of trouble and has established many rules for determining how to resolve such naming conflicts. While it is good to be aware of such issues, you are usually much better off never having to rely on these guidelines. Code defensively! If you don't want to qualify every variable to keep it unique, you will need to use careful naming conventions to avoid these kinds of name collisions.

Nested programs

To conclude the discussion of nesting, scope, and visibility, PL/SQL also offers a particularly important feature known as a *nested program*. A nested program is a procedure or function that appears *completely inside* the declaration section of the enclosing block. Significantly, the nested program can reference any variables and parameters previously declared in the outer block, as demonstrated in this example:

```
PROCEDURE calc_totals (fudge_factor_in IN NUMBER)
IS
   subtotal NUMBER := 0;

   /* Beginning of nested block (in this case a procedure). Notice
   |  we're completely inside the declaration section of calc_totals.
   */
   PROCEDURE compute_running_total (increment_in IN PLS_INTEGER)
   IS
```

```
   BEGIN
      /* subtotal, declared above, is both in scope and visible */
      subtotal := subtotal + increment_in * fudge_factor_in;
   END;
   /* End of nested block */
BEGIN
   FOR month_idx IN 1..12
   LOOP
      compute_running_total (month_idx);
   END LOOP;
   DBMS_OUTPUT.PUT_LINE('Fudged total for year: ' || subtotal);
END;
```

Nested programs can make your program more readable and maintainable, and also allow you to reuse logic that appears in multiple places in the block. For more information about this topic, see Chapter 17.

The PL/SQL Character Set

A PL/SQL program consists of a sequence of statements, each made up of one or more lines of text. The precise characters available to you will depend on what database character set you're using. For example, Table 3-1 illustrates the available characters in the US7ASCII character set.

Table 3-1. Characters available to PL/SQL in the US7ASCII character set

Type	Characters
Letters	A–Z, a–z
Digits	0–9
Symbols	~ ! @ # $ % * ( ) _ – + = | : ; " ' < > , . ? / ^
Whitespace	Tab, space, newline, carriage return

Every keyword, operator, and token in PL/SQL is made from various combinations of characters in this character set. Now you just have to figure out how to put them all together!

And now for some real PL/SQL trivia. Oracle's documentation—as well as earlier editions of this book—lists the ampersand, curly braces, and square brackets as part of the default character set:

```
& { } [ ]
```

While all characters are allowed in literal strings, Oracle does not seem to use these particular five characters anywhere in the visible portions of PL/SQL. Moreover, there is no direct way to use these characters in programmer-defined identifiers.

Regardless of your memory for such trivia, you'll definitely want to remember that *PL/SQL is a case-insensitive language*. That is, it doesn't matter how you type keywords and

identifiers; uppercase letters are treated the same way as lowercase letters unless surrounded by delimiters that make them a literal string. By convention, the authors of this book prefer uppercase for built-in language keywords (and certain identifiers used by Oracle as built-in function and package names), and lowercase for programmer-defined identifiers.

A number of these characters—both singly and in combination with other characters—have a special significance in PL/SQL. Table 3-2 lists these special symbols.

Table 3-2. Simple and compound symbols in PL/SQL

Symbol	Description
;	Semicolon: terminates declarations and statements
%	Percent sign: attribute indicator (cursor attributes like %ISOPEN and indirect declaration attributes like %ROWTYPE); also used as a wildcard symbol with the LIKE condition
_	Single underscore: single-character wildcard symbol in LIKE condition
@	At sign: remote location indicator
:	Colon: host variable indicator, such as :block.item in Oracle Forms
**	Double asterisk: exponentiation operator
< > or != or ^= or ~=	Ways to denote the "not equal" relational operator
||	Double vertical bar: concatenation operator
<< and >>	Label delimiters
<= and >=	Less than or equal to and greater than or equal to relational operators
:=	Assignment operator
=>	Association operator for positional notation
..	Double dot: range operator
--	Double dash: single-line comment indicator
/* and */	Beginning and ending multiline comment block delimiters

Characters are grouped together into *lexical units*, also called *atomics* of the language because they are the smallest individual components. A lexical unit in PL/SQL is any of the following:

- Identifier
- Literal
- Delimiter
- Comment

These are described in the following sections.

Identifiers

An identifier is a name for a PL/SQL object, including any of the following:

- Constant or variable
- Exception
- Cursor
- Program name: procedure, function, package, object type, trigger, etc.
- Reserved word
- Label

Default properties of PL/SQL identifiers are summarized as follows:

- Up to 30 characters in length
- Must start with a letter
- Can include $ (dollar sign), _ (underscore), and # (hash sign)
- Cannot contain any "whitespace" characters

If the only difference between two identifiers is the case of one or more letters, PL/SQL normally treats those two identifiers as the same.[1] For example, the following identifiers are all considered by PL/SQL to be the same:

```
lots_of_$MONEY$
LOTS_of_$MONEY$
Lots_of_$Money$
```

The following strings are valid names of identifiers:

```
company_id#
primary_acct_responsibility
First_Name
FirstName
address_line1
S123456
```

The following identifiers are all illegal in PL/SQL:

```
1st_year                          -- Doesn't start with a letter
procedure-name                    -- Contains invalid character -
minimum_%_due                     -- Contains invalid character %
maximum_value_exploded_for_detail  -- Too long
company ID                        -- Has embedded whitespace
```

1. The compiler accomplishes this internally by converting program text into uppercase during an early phase of compilation.

Identifiers are the handles for objects in your program and one of your chief means of communicating with other programmers. For this reason, many organizations adopt naming conventions. If your project doesn't require naming conventions, you will still want to choose variable names carefully... even if you are the only person who will ever see the code!

Although rarely done in practice, you can actually break some of these rules by surrounding identifiers with double quotation marks. I don't recommend programming like this, but you may one day have to deal with some "clever" code such as:

```
SQL> DECLARE
  2     "pi" CONSTANT NUMBER := 3.141592654;
  3     "PI" CONSTANT NUMBER := 3.14159265358979323846;
  4     "2 pi" CONSTANT NUMBER := 2 * "pi";
  5  BEGIN
  6     DBMS_OUTPUT.PUT_LINE('pi: ' || "pi");
  7     DBMS_OUTPUT.PUT_LINE('PI: ' || pi);
  8     DBMS_OUTPUT.PUT_LINE('2 pi: ' || "2 pi");
  9  END;
 10  /

pi: 3.141592654
PI: 3.14159265358979323846
2 pi: 6.283185308
```

Notice that line 7 refers to pi without quotation marks. Because the compiler accomplishes its case-independence by defaulting identifiers and keywords to uppercase, the variable that line 7 refers to is the one declared on line 3 as "PI".

You may need to use the double-quote trick in SQL statements to refer to database objects that exist with mixed-case names. I've seen this happen when a programmer used Microsoft Access to create Oracle tables.

Reserved Words

Of course, you don't get to (or have to) define all the identifiers in your programs. The PL/SQL language recognizes certain identifiers (such as BEGIN, IF, and THEN) as having special meaning.

PL/SQL provides two kinds of built-in identifiers:

- Reserved words
- Identifiers from the STANDARD package

In both cases you should not—and, in many cases, *cannot*—redefine the identifier for your program's own use.

Reserved words

The PL/SQL compiler reserves certain identifiers for its use only. In other words, you cannot declare a variable with the name of that identifier. These are called *reserved words*. For example, one very important reserved word is END, which terminates blocks, IF statements, and loops. If you try to declare a variable named end:

```
DECLARE
   end VARCHAR2(10) := 'blip';  /* Will not work; "end" is reserved. */
BEGIN
   DBMS_OUTPUT.PUT_LINE (end);
END;
/
```

you will receive this error message from the compiler:

```
PLS-00103: Encountered the symbol "END" when expecting one of the following:…
```

Identifiers from STANDARD package

In addition to avoiding identifiers that duplicate keywords, you should also avoid using identifiers that, in effect, *override* names that Oracle Corporation has defined in a special built-in package named STANDARD. STANDARD is one of two default packages in PL/SQL; Oracle defines in this package many of the basic building blocks of the PL/SQL language, including datatypes like PLS_INTEGER, exceptions like DUP_VAL_ON_INDEX, and functions like UPPER, REPLACE, and TO_DATE.

It may come as a surprise to many developers, but the identifiers defined in STANDARD (and DBMS_STANDARD, the other default package) are *not* reserved words. You *can* declare your own variables with the same names, and your code will compile. You will, however, create lots of confusion if you do this.

The STANDARD package is explored in detail in Chapter 24.

How to avoid using reserved words

Finding a valid name for your identifier should be the least of your problems, as there are thousands and thousands of permutations of the legal characters. The question is: how will you know if you inadvertently use a reserved word in your own program? First of all, the compiler will let you know if you try to use a name for an identifier that is actually reserved. If your curiosity compels you to investigate further, you could build a query against the V$RESERVED_WORDS view, and then try to compile a dynamically constructed PL/SQL block that uses the reserved word as an identifier. I did precisely that; you will find the script in the *reserved_words.sql* file on the book's website. The output from running this script is in *reserved.txt*.

The results are very interesting. Here's the overall summary:

```
Reserved Word Analysis Summary
Total count in V$RESERVED_WORDS = 1733
```

```
Total number of reserved words = 118
Total number of non-reserved words = 1615
```

In other words, the vast majority of words that Oracle includes in this view are not truly reserved; that is, you can use them as the names of your own identifiers.

Generally, I recommend that you avoid using any words that Oracle Corporation uses as part of its own technology. Better yet, use naming conventions that employ consistent prefixes and suffixes, virtually guaranteeing that you will not encounter a true PL/SQL reserved word.

Whitespace and Keywords

Identifiers must be separated by at least one space or by a delimiter, but you can format your text by adding additional spaces, line breaks (newlines and/or carriage returns), and tabs wherever you can put a space, without changing the meaning of your code.

The two statements shown here are therefore equivalent:

```
IF too_many_orders
THEN
   warn_user;
ELSIF no_orders_entered
THEN
   prompt_for_orders;
END IF;

IF too_many_orders THEN warn_user;
ELSIF no_orders_entered THEN prompt_for_orders;
END IF;
```

You may not, however, place a space or carriage return or tab within a lexical unit, such as the "not equals" symbol (!=). This statement results in a compile error:

```
IF max_salary ! = min_salary THEN   -- yields PLS-00103 compile error
```

because the code contains a space between the ! and the =.

Literals

A literal is a value that is not represented by an identifier; it is simply a value. Here is a smattering of literals you *could* see in a PL/SQL program:

Number
: 415, 21.6, 3.141592654f, 7D, NULL

String
: 'This is my sentence', '01-OCT-1986', q'hello!', NULL

Time interval
: INTERVAL '25-6' YEAR TO MONTH, INTERVAL '-18' MONTH, NULL

Boolean
: TRUE, FALSE, NULL

The trailing *f* in number literal 3.141592654f designates a 32-bit floating-point number as defined by the IEEE 754 standard, which Oracle partially supports beginning with Oracle Database 10*g* Release 1. Similarly, 7D is the number 7 as represented in a 64-bit float.

The string q'hello!' bears some explanation. The ! is a user-defined delimiter, also introduced in Oracle Database 10*g*; the leading q and the surrounding single quotes tell the compiler that the ! is the delimiter, and the string represented is simply the word hello.

The INTERVAL datatype allows you to manage amounts of time between dates or timestamps. The first example ('25-6') represents "25 years and 6 months after"; the second ('-18') represents "18 months before."

Even though the database allows you to specify intervals using a literal format, you cannot do so with DATE datatypes; notice that '01-OCT-1986' is listed as a string rather than as an Oracle DATE. Yes, PL/SQL or SQL *can* implicitly convert '01-OCT-198' to and from Oracle's internal date format,[2] but you will normally use built-in functions to perform explicit conversions. For example:

```
TO_DATE('01-OCT-1986', 'DD-MON-YYYY')
TO_TIMESTAMP_TZ('01-OCT-1986 00:00:00 -6','DD-MON-YYYY HH24:MI:SS TZH')
```

Both expressions return October 1, 1986, with zero hours, zero minutes, and zero seconds; the first in the DATE datatype, and the second in the TIMESTAMP WITH TIME ZONE datatype. The second expression also includes time zone information; the –6 represents the number of hours' difference from GMT (UCT).

Unlike identifiers, string literals in PL/SQL are case sensitive. As you would probably expect, the following two literals are different:

```
'Steven'
'steven'
```

So the following condition evaluates to FALSE:

```
IF 'Steven' = 'steven'
```

NULLs

The absence of a value is represented in the Oracle database by the keyword NULL. As shown in the previous section, variables of almost all PL/SQL datatypes can exist in a null state (the exception to this rule is any associative array type, instances of which are

2. As long as the database or session has its NLS_DATE_FORMAT parameter set to DD-MON-YYYY.

never null). Although it can be challenging for a programmer to handle NULL variables properly regardless of their datatype, strings that are null require special consideration.

In Oracle SQL and PL/SQL, a null string is *usually* indistinguishable from a literal of zero characters, represented literally as '' (two consecutive single quotes with no characters between them). For example, the following expression will evaluate to TRUE in both SQL and PL/SQL:

```
'' IS NULL
```

Assigning a zero-length string to a VARCHAR2(*n*) variable in PL/SQL also yields a NULL result:

```
DECLARE
   str VARCHAR2(1) := '';
BEGIN
   IF str IS NULL   -- will be TRUE
```

This behavior is consistent with the database's treatment of VARCHAR2 table columns.

Let's look at CHAR data, though—it's a little quirky. If you create a CHAR(*n*) variable in PL/SQL and assign a zero-length string to it, the database *blank-pads the empty variable with space characters*, making it not null:

```
DECLARE
   flag CHAR(2) := '';  -- try to assign zero-length string to CHAR(2)
BEGIN
   IF flag = '  '   ...   -- will be TRUE
   IF flag IS NULL  ...   -- will be FALSE
```

Strangely, PL/SQL is the only place you will see such behavior. In the database, when you insert a zero-length string into a CHAR(*n*) table column, the database does *not* blank-pad the contents of the column, but leaves it NULL instead!

These examples illustrate Oracle's partial adherence to the 92 and 99 versions of the ANSI SQL standard, which mandate a difference between a zero-length string and a NULL string. Oracle admits this difference, and says it may fully adopt the standard in the future. It's been issuing that warning for about 15 years, though, and it hasn't happened yet.

While NULL tends to behave as if its default datatype is VARCHAR2, the database will try to implicitly cast NULL to whatever type is needed for the current operation. Occasionally, you may need to make the cast explicit, using syntax such as TO_NUMBER(NULL) or CAST(NULL AS NUMBER).

Embedding Single Quotes Inside a Literal String

An unavoidably ugly aspect of working with string literals occurs when you need to put the delimiter itself inside the string. Until Oracle Database 10*g* was released, you would

write two single quotes next to each other if you wanted the string to contain a single quote in that position. The following table offers some examples.

Literal (default delimiter)	Actual value
`'There''s no business like show business.'`	`There's no business like show business.`
`'"Hound of the Baskervilles"'`	`"Hound of the Baskervilles"`
`''''`	`'`
`'''hello'''`	`'hello'`
`''''''`	`''`

The examples show, for instance, that it takes six single quotes to designate a literal containing two consecutive single quotes. In an attempt to simplify this type of construct, Oracle Database 10*g* introduced user-defined delimiters. Start the literal with "q" to mark your delimiter, and surround your delimited expression with single quotes. The following table shows this feature in action.

Literal (delimiters highlighted)	Actual value
`q' ( There's no business like show business.) '`	`There's no business like show busi ness.`
`q' { "Hound of the Baskervilles" } '`	`"Hound of the Baskervilles"`
`q' [ ' ] '`	`'`
`q' !'hello' ! '`	`'hello'`
`q' |'' | '`	`''`

As the examples show, you can use plain delimiters such as ! or |, or you can use "mated" delimiters such as left and right parentheses, curly braces, and square brackets.

One final note: as you would expect, a double-quote character does not have any special significance inside a string literal. It is treated the same as a letter or number.

Numeric Literals

Numeric literals can be integers or real numbers (a number that contains a fractional component). Note that PL/SQL considers the number 154.00 to be a real number of type NUMBER, even though the fractional component is zero and the number is actually an integer. Internally, integers and reals have a different representation, and there is some small overhead involved in converting between the two.

You can also use scientific notation to specify a numeric literal. Use the letter *E* (upper- or lowercase) to multiply a number by 10 to the *n*th power (e.g., 3.05E19, 12e–5).

Beginning with Oracle Database 10*g*, a real can be either an Oracle NUMBER type or an IEEE 754 standard floating-point type. Floating-point literals are either BINARY (32-bit; designated with a trailing *F*) or BINARY DOUBLE (64-bit; designated with a *D*).

In certain expressions you may use the named constants in the following table, as prescribed by the IEEE standard.

Description	Binary float (32-bit)	Binary double (64-bit)
"Not a number" (NaN); result of divide by 0 or invalid operation	BINARY_FLOAT_NAN	BINARY_DOUBLE_NAN
Positive infinity	BINARY_FLOAT_INFINITY	BINARY_DOUBLE_INFINITY
Absolute maximum number that can be represented	BINARY_FLOAT_MAX_NORMAL	BINARY_DOUBLE_MAX_NORMAL
Smallest normal number; underflow threshold	BINARY_FLOAT_MIN_NORMAL	BINARY_DOUBLE_MIN_NORMAL
Maximum positive number that is less than the underflow threshold	BINARY_FLOAT_MAX_SUBNORMAL	BINARY_DOUBLE_MAX_SUBNORMAL
Absolute minimum positive number that can be represented	BINARY_FLOAT_MIN_SUBNORMAL	BINARY_DOUBLE_MIN_SUBNORMAL

Boolean Literals

PL/SQL provides two literals to represent Boolean values: TRUE and FALSE. These values are not strings; you should not put quotes around them. Use Boolean literals to assign values to Boolean variables, as in:

```
DECLARE
   enough_money BOOLEAN; -- Declare a Boolean variable
BEGIN
   enough_money := FALSE; -- Assign it a value
END;
```

You do not, on the other hand, need to refer to the literal value when checking the value of a Boolean expression. Instead, just let that expression speak for itself, as shown in the conditional clause of the following IF statement:

```
DECLARE
   enough_money BOOLEAN;
BEGIN
   IF enough_money
   THEN
      ...
```

A Boolean expression, variable, or constant may also evaluate to NULL, which is neither TRUE nor FALSE. For more information, see Chapter 4, particularly the sidebar "Three-Valued Logic" on page 84.

The Semicolon Delimiter

A PL/SQL program is made up of a series of declarations and statements. These are defined logically, as opposed to physically. In other words, they are not terminated with

the physical end of a line of code; instead, they are terminated with a semicolon (;). In fact, a single statement is often spread over several lines to make it more readable. The following IF statement takes up four lines and is indented to reinforce the logic behind the statement:

```
IF salary < min_salary (2003)
THEN
   salary := salary + salary * .25;
END IF;
```

There are two semicolons in this IF statement. The first semicolon indicates the end of the single executable statement within the IF-END IF construct. The second semicolon terminates the IF statement itself. This same statement could also be placed on a single physical line and have exactly the same result:

```
IF salary < min_salary (2003) THEN salary := salary + salary*.25; END IF;
```

The semicolons are still needed to terminate each logical, executable statement, even if they are nested inside one another. Unless you're *trying* to create unreadable code, I suggest that you not combine the different components of the IF statement on a single line. I also recommend that you place no more than one statement or declaration on each line.

Comments

Inline documentation, otherwise known as *comments*, is an important element of a good program. While this book offers many suggestions on how to make your program self-documenting through good naming practices and modularization, such techniques are seldom enough by themselves to communicate a thorough understanding of a complex program.

PL/SQL offers two different styles for comments: single-line and multiline block comments.

Single-Line Comment Syntax

The single-line comment is initiated with two hyphens (--), which cannot be separated by a space or any other characters. All text after the double hyphen to the end of the physical line is considered commentary and is ignored by the compiler. If the double hyphen appears at the beginning of the line, the whole line is a comment.

Remember: the double hyphen comments out the remainder of a physical line, not a logical PL/SQL statement. In the following IF statement, I use a single-line comment to clarify the logic of the Boolean expression:

```
IF salary < min_salary (2003) -- Function returns min salary for year.
THEN
```

```
   salary := salary + salary*.25;
END IF;
```

Multiline Comment Syntax

While single-line comments are useful for documenting brief bits of code or ignoring a line that you do not want executed at the moment, the multiline comment is superior for including longer blocks of commentary.

Multiline comments start with a slash-asterisk (/*) and end with an asterisk-slash (*/). PL/SQL considers all characters found between these two sequences of symbols to be part of the comment, and the compiler ignores them.

The following example of a multiline comment shows a header section for a procedure. I use the vertical bars in the left margin so that, as the eye moves down the left edge of the program, it can easily pick out the chunks of comments:

```
PROCEDURE calc_revenue (company_id IN NUMBER) IS
/*
| Program: calc_revenue
| Author: Steven Feuerstein
| Change history:
|   10-JUN-2009 Incorporate new formulas
|   23-SEP-2008 - Program created
|*/
BEGIN
   ...
END;
```

You can also use multiline comments to block out lines of code for testing purposes. In the following example, the additional clauses in the EXIT statement are ignored so that testing can concentrate on the a_delimiter function:

```
EXIT WHEN a_delimiter (next_char)
/*
            OR
         (was_a_delimiter AND NOT a_delimiter (next_char))
*/
;
```

The PRAGMA Keyword

A programming notion that is truly derived from Greek is *pragma*, which means "deed" or, by implication, an "action." In various programming languages, a pragma is generally a line of source code prescribing an action you want the compiler to take. It's like an option that you give the compiler; it can result in different runtime behavior for the program, but it doesn't get translated directly into bytecode.

PL/SQL has a PRAGMA keyword with the following syntax:

```
PRAGMA instruction_to_compiler;
```

The PL/SQL compiler will accept such directives anywhere in the declaration section, but most of them have certain additional requirements regarding placement.

PL/SQL offers several pragmas:

AUTONOMOUS_TRANSACTION
: Tells the PL/SQL runtime engine to commit or roll back any changes made to the database inside the current block without affecting the main or outer transaction. See Chapter 14 for more information.

EXCEPTION_INIT
: Tells the compiler to associate a particular error number with an identifier you have declared as an exception in your program. Must follow the declaration of the exception. See Chapter 6 for more information.

RESTRICT_REFERENCES
: Tells the compiler the purity level (freedom from side effects) of a packaged program. See Chapter 17 for more information.

SERIALLY_REUSABLE
: Tells the PL/SQL runtime engine that package-level data should not persist between references to that data. See Chapter 18 for more information.

The following block demonstrates the use of the EXCEPTION_INIT pragma to name a built-in exception that would otherwise have only a number:

```
DECLARE
   no_such_sequence EXCEPTION;
   PRAGMA EXCEPTION_INIT (no_such_sequence, -2289);
BEGIN
   ...
EXCEPTION
   WHEN no_such_sequence
   THEN
      q$error_manager.raise_error ('Sequence not defined');
END;
```

Labels

A PL/SQL label is a way to name a particular part of your program. Syntactically, a label has the format:

```
<<identifier>>
```

where *identifier* is a valid PL/SQL identifier (up to 30 characters in length and starting with a letter, as discussed in the section "Identifiers" on page 67). There is no terminator. Labels appear directly in front of the thing they're labeling, which must be an executable statement—even if it is merely the NULL statement:

```
BEGIN
   ...
   <<the_spot>>
   NULL;
```

Because anonymous blocks are themselves executable statements, a label can "name" an anonymous block for the duration of its execution. For example:

```
<<insert_but_ignore_dups>>
BEGIN
   INSERT INTO catalog
   VALUES (...);
EXCEPTION
   WHEN DUP_VAL_ON_INDEX
   THEN
      NULL;
END insert_but_ignore_dups;
```

One reason you might label a block is to improve the readability of your code. When you give something a name, you self-document that code. You also clarify your own thinking about what that code is supposed to do, sometimes ferreting out errors in the process.

Another reason to use a block label is to allow you to qualify references to elements from an enclosing block that have duplicate names in the current, nested block. Here's a schematic example:

```
<<outerblock>>
DECLARE
   counter INTEGER := 0;
BEGIN
   ...
   DECLARE
      counter INTEGER := 1;
   BEGIN
      IF counter = outerblock.counter
      THEN
         ...
      END IF;
   END;
END;
```

Without the block label, there would be no way to distinguish between the two counter variables. Again, though, a better solution would probably have been to use distinct variable names.

A third function of labels is to serve as the target of a GOTO statement. See the discussion of GOTO in Chapter 4.

Although few programs I've seen or worked on require the use of labels, there is one final use of this feature that is more significant than the previous three combined: a label can serve as a target for the EXIT statement in a nested loop. Here's some example code:

```
BEGIN
   <<outer_loop>>
   LOOP
      LOOP
         EXIT outer_loop;
      END LOOP;
      some_statement;
   END LOOP;
END;
```

Without the <<outer_loop>> label, the EXIT statement would have exited only the inner loop and would have executed *some_statement*. But I didn't want it to do that. So, in this case, the label provides functionality that PL/SQL does not offer in any other straightforward way.