PL SQL 101, Database Developer, Database, SQL and PL SQL, DBA

Controlling the flow of execution in PL/SQL

September 2, 2020 | 18 minute read

Steven Feuerstein
Developer Advocate for PL/SQL

Part 2 in a series of articles on understanding and using PL/SQL for accessing Oracle Database

PL/SQL is one of the core technologies at Oracle and is essential to leveraging the full potential of Oracle Database. PL/SQL combines the relational data access capabilities of the Structured Query Language with a flexible embedded procedural language, and it executes complex queries and programmatic logic run inside the database engine itself. This enhances the agility, efficiency, and performance of database-driven applications.

Steven Feuerstein, one of the industry's best-respected and most prolific experts in PL/SQL, wrote a 12-part tutorial series on the language. Those articles, first published in 2011, have been among the most popular ever published on the Oracle website and continue to find new readers and enthusiasts in the database community. Beginning with the first installment, the entire series is being updated and republished; please enjoy!

There is one way in which PL/SQL is just like every other programming language you will ever use: It (the PL/SQL runtime engine) does only exactly what you tell it to do. Each block of PL/SQL code you write contains one or more statements implementing complex business rules or processes. When you run a block of code, as either an anonymous block or a script or by calling a stored program unit that contains all the logic, Oracle Database follows the directions you specify in that block.

It is therefore critical to know how to specify which statements should be run, under what circumstances, and with what frequency. To do this, Oracle Database offers conditional and iterative constructs. This article introduces you to the IF statement, the CASE statement and expression, and the various types of loops PL/SQL supports.

Conditional branching in code

Almost every piece of code you write will require *conditional control*, that is, the ability to direct the flow of execution through your program, based on a condition. You do this with IF-THEN-ELSE and CASE statements.

There are also CASE *expressions*; although not the same as CASE *statements*, they can sometimes be used to eliminate the need for an IF or CASE statement altogether.

IF. The IF statement enables you to implement conditional branching logic in your programs. With it, you'll be able to implement requirements such as the following:

If the salary is between $10,000 and $20,000, apply a bonus of $1,500.

If the collection contains more than 100 elements, truncate it.

The IF statement comes in three flavors, as shown in **Table 1**. Let's take a look at some examples of IF statements.

IF Type	Characteristics

IF THEN
END IF;
	This is the simplest form of the IF statement. The condition between IF and THEN determines whether the set of statements between THEN and END IF should be executed. If the condition evaluates to FALSE or NULL, the code will not be executed.

IF THEN
ELSE
END IF;
	This combination implements either/or logic: based on the condition between the IF and THEN keywords, execute the code either between THEN and ELSE or between ELSE and END IF. One of these two sections of statements is executed.

F THEN
ELSIF
ELSE
END IF;
	This last and most complex form of the IF statement selects a condition that is TRUE from a series of mutually exclusive conditions and then executes the set of statements associated with that condition. If you're writing IF statements like this in any Oracle Database release from Oracle9*I* Database Release 1 onward, you should consider using searched CASE statements instead.

Table 1: IF statement flavors

IF-THEN. The following statement compares two numeric values. Note that if one of these two values is NULL, the entire expression will return NULL. In the following example, the bonus is not given when salary is NULL:

Copy code snippet

```
IF l_salary > 40000
THEN
   give_bonus (l_employee_id,500);
END IF;
```

There are exceptions to the rule that a NULL in a Boolean expression leads to a NULL result. Some operators and functions are specifically designed to deal with NULLs in a way that leads to TRUE and FALSE (and not NULL) results. For example, you can use IS NULL to test for the presence of a NULL:

Copy code snippet

```
IF l_salary > 40000 OR l_salary
IS NULL
```

```
   THEN
      give_bonus (l_employee_id,500);
   END IF;
```

In this example, “salary IS NULL” evaluates to TRUE in the event that salary has no value; otherwise, it evaluates to FALSE. Employees whose salaries are missing will now get bonuses too. (As indeed they probably should, considering that their employer was so inconsiderate as to lose track of their pay in the first place.)

IF-THEN-ELSE. Here is an example of the IF-THEN-ELSE construct (which builds upon the IF-THEN construct):

Copy code snippet

```
   IF l_salary <= 40000
   THEN
      give_bonus (l_employee_id, 0);
   ELSE
      give_bonus (l_employee_id, 500);
   END IF;
```

In this example, employees with a salary greater than $40,000 will get a bonus of $500, whereas all other employees will get no bonus at all—or will they? What happens if the salary, for whatever reason, happens to be NULL for a given employee? In that case, the statements following the ELSE will be executed and the employee in question will get the bonus, which is supposed to go only to highly paid employees. If the salary could be NULL, you can protect yourself against this problem by using the NVL function:

Copy code snippet

```
   IF NVL(l_salary,0) <= 40000
   THEN
      give_bonus (l_employee_id, 0);
   ELSE
      give_bonus (l_employee_id, 500);
   END IF;
```

The NVL function will return 0 whenever salary is NULL, ensuring that any employees with a NULL salary will also get no bonus (those poor employees).

The important thing to remember is that one of the two sequences of statements will *always* execute, because IF-THEN-ELSE is an either/or construct. Once the appropriate set of statements has been executed, control passes to the statement immediately following END IF.

IF-ELSIF. Now let’s take a look at the use of ELSIF. This last form of the IF statement comes in handy when you have to implement logic that has many alternatives; it is not an either/or situation. The IF-ELSIF formulation provides a way to handle multiple conditions within a single IF statement. In general, you execution of the IF statement).

Each ELSIF clause must have a THEN after its condition. (Only the ELSE keyword does not need the THEN keyword.) The ELSE clause in the IF-ELSIF is the “otherwise” of the statement. If none of the conditions evaluates to TRUE, the statements in the ELSE clause will be executed. But the ELSE clause is optional. You can code an IF-ELSIF that has only IF and ELSIF clauses. In such a case, if none of the conditions is TRUE, no statements inside the IF block will be executed.

Here’s an example of an IF-ELSIF statement that checks for three different conditions and contains an ELSE clause in case none of those conditions evaluates to TRUE.

Copy code snippet

```
IF l_salary BETWEEN 10000 AND 20000
THEN
   give_bonus(l_employee_id, 1000);
ELSIF l_salary > 20000
THEN
   give_bonus(l_employee_id, 500);
ELSE
   give_bonus(l_employee_id, 0);
END IF;
```

CASE: A useful alternative to IF

The CASE statement and expression offer another way to implement conditional branching. By using CASE instead of IF, you can often express conditions more clearly and even simplify your code. There are two forms of the CASE statement:

Simple CASE statement. This one associates each of one or more sequences of PL/SQL statements with a value. The simple CASE statement chooses which sequence of statements to execute, based on an expression that returns one of those values.

Searched CASE statement. This one chooses which of one or more sequences of PL/SQL statements to execute by evaluating a list of Boolean conditions. The sequence of statements associated with the first condition that evaluates to TRUE is executed.

In addition to CASE statements, PL/SQL also supports CASE expressions. A CASE expression is similar in form to a CASE statement and enables you to choose which of one or more expressions to evaluate. The result of a CASE expression is a single value, whereas the result of a CASE statement is the execution of a sequence of PL/SQL statements.

Simple CASE statements. A simple CASE statement enables you to choose which of several sequences of PL/SQL statements to execute, based on the results of a single expression. Simple CASE statements take the following form:

Copy code snippet

```
CASE expression
```

```
      -
   WHEN result1 THEN
      statements1
   WHEN result2 THEN
      statements2
   ...
   ELSE
      statements_else
   END CASE;
```

Here is an example of a simple CASE statement that uses the employee type as a basis for selecting the proper bonus algorithm:

Copy code snippet

```
CASE l_employee_type
   WHEN 'S'
   THEN
      award_bonus (l_employee_id);
   WHEN 'H'
   THEN
      award_bonus (l_employee_id);
   WHEN 'C'
   THEN
       award_commissioned_bonus (
          l_employee_id);
   ELSE
       RAISE invalid_employee_type;
END CASE;
```

This CASE statement has an explicit ELSE clause, but the ELSE is optional. When you do not explicitly specify an ELSE clause of your own, PL/SQL implicitly uses the following:

Copy code snippet

```
ELSE
   RAISE CASE_NOT_FOUND;
```

In other words, if you don't specify an ELSE clause and none of the results in the WHEN clauses matches the result of the CASE expression, PL/SQL will raise a CASE_NOT_FOUND error. This behavior is different from that of IF statements. When an IF statement lacks an ELSE clause, nothing happens when the condition is not met. With CASE, the analogous situation leads to an error.

Searched CASE statements. A searched CASE statement evaluates a list of Boolean expressions and, when it finds an expression that evaluates to TRUE, executes a sequence of statements associated with that expression. Searched CASE statements have the following form:

Copy code snippet

```
CASE
WHEN expression1 THEN
   statements1
WHEN expression2 THEN
   statements2
...
ELSE
   statements_else
END CASE;
```

A searched CASE statement is a perfect fit for the problem of implementing my bonus logic, as shown below:

Copy code snippet

```
CASE
  WHEN l_salary
  BETWEEN 10000 AND 20000
  THEN
    give_bonus(l_employee_id, 1500);
  WHEN salary > 20000
  THEN
    give_bonus(l_employee_id, 1000);
  ELSE
    give_bonus(l_employee_id, 0);
END CASE;
```

By the way, because WHEN clauses are evaluated in order, you may be able to squeeze some extra efficiency out of your code by listing the most likely WHEN clauses first. In addition, if you have WHEN clauses with "expensive" expressions (that is, ones requiring lots of CPU cycles and memory), you may want to list those last to minimize the chances that they will be evaluated.

Use searched CASE statements when you want to use Boolean expressions as a basis for identifying a set of statements to execute. Use simple CASE statements when you can base that decision on the result of a single expression.

Using CASE expressions. A CASE expression returns a single value, the result of whichever result_expression is chosen. Each WHEN clause must be associated with exactly one *expression* (not statement). Do not use semicolons or END CASE to mark the end of the CASE expression. CASE expressions are terminated by a simple END.

A searched CASE expression can be used to simplify the code for applying bonuses. Rather than writing an IF or CASE statement with three different calls to give_bonus, I can call give_bonus just once and use a CASE expression in place of the second argument:

Copy code snippet

```
give_bonus(l_employee_id,
  CASE
     WHEN l_salary
     BETWEEN 10000 AND 20000
     THEN 1500
     WHEN l_salary > 40000
     THEN 500
     ELSE 0
END);
```

Unlike with the CASE statement, no error is raised in the event that no WHEN clause is selected in a CASE expression. Instead, if no WHEN condition is met, a CASE expression will simply return NULL.

Iterative processing with loops

Loops in code give you a way to execute the same body of code more than once. Loops are a very common element of programming, because so much of the real-world activity modeled by our programs involves repetitive processing. We might need, for example, to perform an operation for each month of the previous year. To cite a common example from Oracle's famous employees table, we might want to update information for all the employees in a given department.

PL/SQL provides three kinds of loop constructs:

- The FOR loop (numeric and cursor)
- The simple (or infinite) loop
- The WHILE loop

Each type of loop is designed for a specific purpose with its own nuances, rules for use, and guidelines for high-quality construction.

To give you a feel for how the different loops solve problems in different ways, consider the following three loop examples. In each case, the procedure makes a call to display_total_sales for a particular year, for each year between the start and end argument values.

The FOR loop. Oracle Database offers both a numeric and a cursor FOR loop. With the numeric FOR loop, you specify the start and end integer values, and PL/SQL does the rest of the work for you, iterating through each integer value between the start and the end and then terminating the loop:

Copy code snippet

```
PROCEDURE display_multiple_years (
   start_year_in IN PLS_INTEGER
  ,end_year_in   IN PLS_INTEGER
)
IS
BEGIN
  FOR l_current_year
  IN start_year_in .. end_year_in
```

```
      display_total_sales
              (l_current_year);
    END LOOP;
  END display_multiple_years;
```

The cursor FOR loop has the same basic structure, but with it you supply an explicit cursor or SELECT statement in place of the low/high integer range:

Copy code snippet

```
  PROCEDURE display_multiple_years (
      start_year_in IN PLS_INTEGER
     ,end_year_in IN PLS_INTEGER
  )
  IS
  BEGIN
    FOR l_current_year IN (
       SELECT * FROM sales_data
         WHERE year
         BETWEEN start_year_in
         AND end_year_in)
    LOOP
       display_total_sales
               (l_current_year);
    END LOOP;
  END display_multiple_years;
```

In both the numeric and the cursor FOR loop, Oracle Database implicitly declares the iterator (in the examples above, it is l_current_year) for you, as either an integer or a record. You do not have to (and should not) declare a variable with that same name, as in

Copy code snippet

```
  PROCEDURE display_multiple_years (
     start_year_in IN PLS_INTEGER
    ,end_year_in IN PLS_INTEGER
  )
  IS
    l_current_year
            INTEGER; /* NOT NEEDED */
  BEGIN
    FOR l_current_year
            IN start_year_in
               .. end_year_in
```

In fact, if you *do* declare such a variable, it will not be used by the FOR loop. It is, more specifically, a *different* integer variable than that declared implicitly by Oracle Database and used within the body of the loop.

The simple loop. It's called *simple* for a reason: It starts simply with the LOOP keyword and ends with the END LOOP statement. The loop will terminate if you execute an EXIT, EXIT WHEN, or RETURN within the body of the loop (or if an exception is raised).

Listing 1 presents a simple loop. **Listing 2** presents a simple loop that iterates through the rows of a cursor (logically equivalent to the cursor FOR loop of the previous section).

Code listing 1: A simple loop

Copy code snippet

```
PROCEDURE display_multiple_years (
   start_year_in IN PLS_INTEGER
  ,end_year_in   IN PLS_INTEGER
)
IS
   l_current_year PLS_INTEGER := start_year_in;
BEGIN
   LOOP
      EXIT WHEN l_current_year > end_year_in;
      display_total_sales (l_current_year);
      l_current_year :=  l_current_year + 1;
   END LOOP;
END display_multiple_years;
```

Compare the cursor-based simple loop in **Listing 2** with the cursor FOR loop. Note that I must explicitly open the cursor, fetch the next record, determine by using the %NOTFOUND cursor attribute whether or not I am done fetching, and then close the cursor after the loop terminates.

Code listing 2: A simple loop that iterates through the rows of a cursor

Copy code snippet

```
PROCEDURE display_multiple_years (
   start_year_in   IN PLS_INTEGER
 , end_year_in     IN PLS_INTEGER)
IS
   CURSOR years_cur
   IS
      SELECT *
        FROM sales_data
       WHERE year BETWEEN start_year_in AND end_year_in;

   l_year   sales_data%ROWTYPE;
BEGIN
   OPEN years_cur;

   LOOP
```

```
      FETCH years_cur INTO l_year;

      EXIT WHEN years_cur%NOTFOUND;

      display_total_sales (l_year);
   END LOOP;

   CLOSE years_cur;
END display_multiple_years;
```

The cursor FOR loop requires none of these steps; Oracle Database performs all steps (open, fetch, terminate, close) implicitly.

Although a simple loop should not be used to fetch row by row through a cursor's dataset, it *should* be used when (1) you may need to conditionally exit from the loop (with an EXIT statement) and (2) you want the body of the loop to execute at least once.

The WHILE loop. The WHILE loop is very similar to the simple loop; but a critical difference is that it checks the termination condition up front. That is, a WHILE loop might not even execute its body a single time. **Listing 3** demonstrates the WHILE loop.

Code listing 3: A WHILE loop

Copy code snippet

```
PROCEDURE display_multiple_years (
   start_year_in IN PLS_INTEGER
  ,end_year_in   IN PLS_INTEGER
)
IS
   l_current_year PLS_INTEGER := start_year_in;
BEGIN
   WHILE (l_current_year <= end_year_in)
   LOOP
      display_total_sales (l_current_year);
      l_current_year :=  l_current_year + 1;
   END LOOP;
END display_multiple_years;
```

The WHILE loop consists of a condition (a Boolean expression) and a loop body. Before each iteration of the body, Oracle Database evaluates the condition. If it evaluates to TRUE, the loop body will execute. If it evaluates to FALSE or NULL, the loop will terminate.

You must then make sure to include code in the body of the loop that will affect the evaluation of the condition—and, at some point, cause the loop to stop. In the procedure in **Listing 3**, that code is

Copy code snippet

```
   l_current_year :=  l_current_year + 1
```

In other words, move to the next year until the total sales for all specified years are displayed.

If your WHILE loop does not somehow change the way the condition is evaluated, that loop will never terminate.

One way in, one way out

In all the examples in the previous section, the FOR loop clearly requires the smallest amount of code. Generally, you want to find the simplest, most readable implementation for your requirements. Does that mean that you should always use a FOR loop? Not at all.

Using the FOR loop is the best solution for the scenario described because I needed to run the body of the loop a fixed number of times. In many other situations, the number of times a loop must execute will vary, depending on the state of the data in the application. You may also need to terminate the loop when a certain condition has occurred; in this case, a FOR loop is *not* a good fit.

One important and fundamental principle in structured programming is "one way in, one way out"—that is, a program should have a single point of entry and a single point of exit. A single point of entry is not an issue with PL/SQL—no matter what kind of loop you are using, there is always only one entry point into the loop: the first executable statement following the LOOP keyword. It is quite possible, however, to construct loops that have multiple exit paths. *Avoid this practice*. Having multiple ways of terminating a loop results in code that is much harder to debug and maintain than it would be otherwise.

In particular, you should follow these two guidelines for loop termination:

1. Do not use EXIT or EXIT WHEN statements within FOR and WHILE loops. You should use a FOR loop only when you want to iterate through *all* the values (integer or record) specified in the range. An EXIT inside a FOR loop disrupts this process and subverts the intent of that structure. A WHILE loop, on the other hand, specifies its termination condition in the WHILE statement itself. **Listing 4** presents an example of a FOR loop with a conditional exit. I want to display the total sales for each year within the specified range. If I ever encounter a year with zero sales (calculated by the total_sales_for_year function), however, I should stop the loop. **Code listing 4:** A FOR loop with a conditional exit

Copy code snippet

```
PROCEDURE display_multiple_years (
   start_year_in    IN PLS_INTEGER
 , end_year_in      IN PLS_INTEGER)
IS
BEGIN
   FOR l_current_year IN start_year_in .. end_year_in
   LOOP
      display_total_sales_for_year (l_current_year);

      EXIT WHEN total_sales_for_year (l_current_year) = 0;
   END LOOP;
END display_multiple_years;
```

There are now two ways "out" of the loop. In this situation, I will rewrite the FOR loop as a WHILE loop, which means I must now also declare the iterator and add to it within the loop, as shown in **Listing 5**.

2. Do not use RETURN or GOTO statements within a loop—these cause the premature, unstructured termination of the loop.

Code listing 5: A WHILE loop with one exit

Copy code snippet

```
PROCEDURE display_multiple_years (
   start_year_in    IN PLS_INTEGER
 , end_year_in      IN PLS_INTEGER)
IS
   l_current_year PLS_INTEGER := start_year_in;
BEGIN
   WHILE ( l_current_year <= end_year_in
         AND total_sales_for_year (l_current_year) > 0)
   LOOP
      display_total_sales_for_year (l_current_year);
       l_current_year := l_current_year + 1;
   END LOOP;
END display_multiple_years;
```

Listing 6 presents an example of a FOR loop with a RETURN in it. The total_sales function returns the total sales across the specified years, but if any year has $0 in sales, the function should terminate the loop and return the current total sales.

Code listing 6: A FOR loop with two instances of RETURN

Copy code snippet

```
FUNCTION total_sales (
   start_year_in    IN PLS_INTEGER
 , end_year_in      IN PLS_INTEGER)
   RETURN PLS_INTEGER
IS
   l_return    PLS_INTEGER := 0;
BEGIN
   FOR l_current_year IN start_year_in .. end_year_in
   LOOP
      IF total_sales_for_year (l_current_year) = 0
      THEN
         RETURN l_return;
      ELSE
         l_return :=
            l_return + total_sales_for_year (l_current_year);
      END IF;
   END LOOP;
```

```
      RETURN l_return;
   END total_sales;
```

Note that the loop terminates in one of two ways: either by iterating through all integers between the start and end years or by executing the RETURN *inside* the loop. In addition and related to that, this function now has *two* instances of RETURN, which means that there are two ways "out" of the function. This is also not a recommended way to design your functions. You should have just a single RETURN in your executable section: the last line of the function.

I can restructure this function so that both the loop and the function have just "one way out," as shown in **Listing 7**.

Code listing 7: A loop revision with one way out

Copy code snippet

```
   FUNCTION total_sales (
      start_year_in    IN PLS_INTEGER
    , end_year_in      IN PLS_INTEGER)
      RETURN PLS_INTEGER
   IS
      l_current_year   PLS_INTEGER := start_year_in;
      l_return         PLS_INTEGER := 0;
   BEGIN
      WHILE (l_current_year <= end_year_in
         AND total_sales_for_year (l_current_year) > 0)
      LOOP
         l_return := l_return + total_sales_for_year (l_current_year);
         l_current_year := l_current_year + 1;
      END LOOP;

      RETURN l_return;
   END total_sales;
```

All the logic required to terminate the loop is now in the WHILE condition, and after the loop is finished, the function executes a single RETURN to send back the total sales value. This second implementation is now simpler and easier to understand—and that is for a program that itself is already quite simple. When you work with much more complex algorithms, following the "one way in, one way out" guidelines will have an even *greater* impact on the readability of your code.

In this article, you learned about how to tell the PL/SQL compiler to conditionally and iteratively execute the statements in your block. This control will enable you to write stored program units that mirror the business process flow defined by your users.

Spoiler alert

The next article in this series, Part 3, will discuss working with strings in PL/SQL programs.