**Oracle® Database PL/SQL User's Guide and Reference 10g Release 2 (10.2)**
B14261-01



# 4 Using PL/SQL Control Structures

This chapter shows you how to structure the flow of control through a PL/SQL program. PL/SQL provides conditional tests, loops, and branches that let you produce well-structured programs.

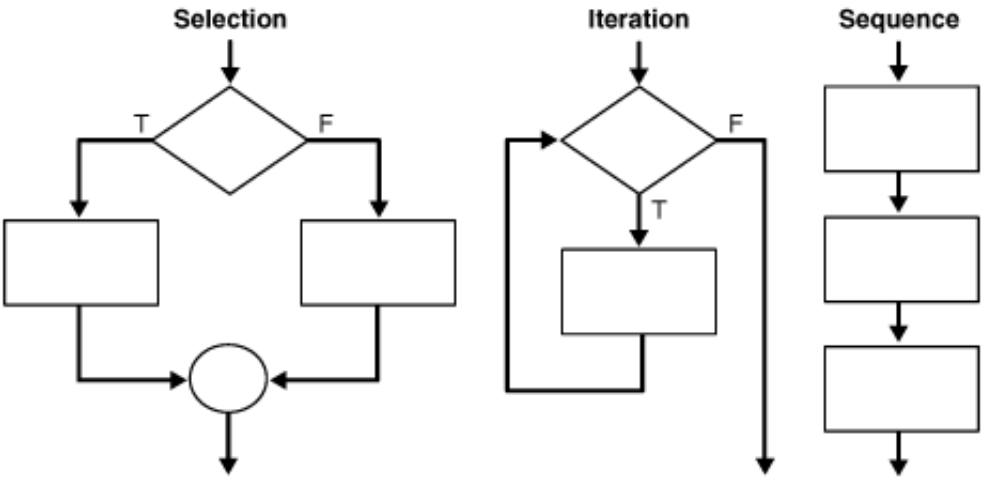
This chapter contains these topics:



## Overview of PL/SQL Control Structures

Procedural computer programs use the basic control structures shown in Figure 4-1.

**Figure 4-1 Control Structures**




Description of the illustration lnpls008.gif

The selection structure tests a condition, then executes one sequence of statements instead of another, depending on whether the condition is true or false. A condition is any variable or expression that returns a `BOOLEAN` value (`TRUE` or `FALSE`). The iteration structure executes a sequence of statements repeatedly as long as a condition holds true. The sequence structure simply executes a sequence of statements in the order in which they occur.

# Testing Conditions: IF and CASE Statements

The IF statement executes a sequence of statements depending on the value of a condition. There are three forms of IF statements: IF-THEN, IF-THEN-ELSE, and IF-THEN-ELSIF. For a description of the syntax of the IF statement, see "IF Statement".

The CASE statement is a compact way to evaluate a single condition and choose between many alternative actions. It makes sense to use CASE when there are three or more alternatives to choose from. For a description of the syntax of the CASE statement, see "CASE Statement".

## Using the IF-THEN Statement

The simplest form of IF statement associates a condition with a sequence of statements enclosed by the keywords THEN and END IF (not ENDIF) as illustrated in Example 4-1.

The sequence of statements is executed only if the condition is TRUE. If the condition is FALSE or NULL, the IF statement does nothing. In either case, control passes to the next statement.

**Example 4-1 Using a Simple IF-THEN Statement**

DECLARE sales NUMBER(8,2) := 10100; quota NUMBER(8,2) := 10000; bonus NUMBER(6,2); emp_id NUMBER(6) := 120; BEGIN IF sales > (quota + 200) THEN bonus := (sales - quota)/4; UPDATE employees SET salary = salary + bonus WHERE employee_id = emp_id; END IF; END; /

## Using the IF-THEN-ELSE Statement

The second form of IF statement adds the keyword ELSE followed by an alternative sequence of statements, as shown in Example 4-2.

The statements in the ELSE clause are executed only if the condition is FALSE or NULL. The IF-THEN-ELSE statement ensures that one or the other sequence of statements is executed. In the Example 4-2, the first UPDATE statement is executed when the condition is TRUE, and the second UPDATE statement is executed when the condition is FALSE or NULL.

**Example 4-2 Using a Simple IF-THEN-ELSE Statement**

```
DECLARE
  sales  NUMBER(8,2) := 12100;
  quota  NUMBER(8,2) := 10000;
  bonus  NUMBER(6,2);
  emp_id NUMBER(6) := 120;
BEGIN
  IF sales > (quota + 200) THEN
     bonus := (sales - quota)/4;
  ELSE
```

```
      bonus := 50;
  END IF;
  UPDATE employees SET salary = salary + bonus WHERE employee_id = emp_id;
END;
/
```

IF statements can be nested as shown in Example 4-3.

### Example 4-3 Nested IF Statements

```
DECLARE
  sales  NUMBER(8,2) := 12100;
  quota  NUMBER(8,2) := 10000;
  bonus  NUMBER(6,2);
  emp_id NUMBER(6) := 120;
BEGIN
  IF sales > (quota + 200) THEN
     bonus := (sales - quota)/4;
  ELSE
     IF sales > quota THEN
        bonus := 50;
     ELSE
        bonus := 0;
     END IF;
  END IF;
  UPDATE employees SET salary = salary + bonus WHERE employee_id = emp_id;
END;
/
```

## Using the IF-THEN-ELSIF Statement

Sometimes you want to choose between several alternatives. You can use the keyword ELSIF (not ELSEIF or ELSE IF) to introduce additional conditions, as shown in Example 4-4.

If the first condition is FALSE or NULL, the ELSIF clause tests another condition. An IF statement can have any number of ELSIF clauses; the final ELSE clause is optional. Conditions are evaluated one by one from top to bottom. If any condition is TRUE, its associated sequence of statements is executed and control passes to the next statement. If all conditions are false or NULL, the sequence in the ELSE clause is executed, as shown in Example 4-4.

### Example 4-4 Using the IF-THEN-ELSEIF Statement

```
DECLARE
  sales  NUMBER(8,2) := 20000;
```

```
  bonus  NUMBER(6,2);
  emp_id NUMBER(6) := 120;
BEGIN
   IF sales > 50000 THEN
      bonus := 1500;
   ELSIF sales > 35000 THEN
      bonus := 500;
   ELSE
      bonus := 100;
   END IF;
   UPDATE employees SET salary = salary + bonus WHERE employee_id = emp_id;
END;
/
```

If the value of `sales` is larger than 50000, the first and second conditions are `TRUE`. Nevertheless, `bonus` is assigned the proper value of 1500 because the second condition is never tested. When the first condition is `TRUE`, its associated statement is executed and control passes to the `INSERT` statement.

Another example of an `IF-THEN-ELSE` statement is Example 4-5.

**Example 4-5 Extended IF-THEN Statement**

```
DECLARE
  grade CHAR(1);
BEGIN
  grade := 'B';
  IF grade = 'A' THEN
    DBMS_OUTPUT.PUT_LINE('Excellent');
  ELSIF grade = 'B' THEN
    DBMS_OUTPUT.PUT_LINE('Very Good');
  ELSIF grade = 'C' THEN
    DBMS_OUTPUT.PUT_LINE('Good');
  ELSIF grade = 'D' THEN
    DBMS_OUTPUT. PUT_LINE('Fair');
  ELSIF grade = 'F' THEN
    DBMS_OUTPUT.PUT_LINE('Poor');
  ELSE
    DBMS_OUTPUT.PUT_LINE('No such grade');
  END IF;
ENd;
/
```

# Using CASE Statements

Like the `IF` statement, the `CASE` statement selects one sequence of statements to execute. However,

to select the sequence, the CASE statement uses a selector rather than multiple Boolean expressions. A selector is an expression whose value is used to select one of several alternatives.

To compare the IF and CASE statements, consider the code in Example 4-5 that outputs descriptions of school grades. Note the five Boolean expressions. In each instance, we test whether the same variable, grade, is equal to one of five values: 'A', 'B', 'C', 'D', or 'F'. You can rewrite the code inExample 4-5 using the CASE statement, as shown in Example 4-6.

**Example 4-6 Using the CASE-WHEN Statement**

```
DECLARE
  grade CHAR(1);
BEGIN
  grade := 'B';
  CASE grade
    WHEN 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
    WHEN 'B' THEN DBMS_OUTPUT.PUT_LINE('Very Good');
    WHEN 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
    WHEN 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
    WHEN 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
    ELSE DBMS_OUTPUT.PUT_LINE('No such grade');
  END CASE;
END;
/
```

The CASE statement is more readable and more efficient. When possible, rewrite lengthy IF-THEN-ELSIF statements as CASE statements.

The CASE statement begins with the keyword CASE. The keyword is followed by a selector, which is the variable grade in the last example. The selector expression can be arbitrarily complex. For example, it can contain function calls. Usually, however, it consists of a single variable. The selector expression is evaluated only once. The value it yields can have any PL/SQL datatype other than BLOB, BFILE, an object type, a PL/SQL record, an index-by-table, a varray, or a nested table.

The selector is followed by one or more WHEN clauses, which are checked sequentially. The value of the selector determines which clause is executed. If the value of the selector equals the value of a WHEN-clause expression, that WHEN clause is executed. For instance, in the last example, if grade equals 'C', the program outputs 'Good'. Execution never falls through; if any WHEN clause is executed, control passes to the next statement.

The ELSE clause works similarly to the ELSE clause in an IF statement. In the last example, if the grade is not one of the choices covered by a WHEN clause, the ELSE clause is selected, and the phrase 'No such grade' is output. The ELSE clause is optional. However, if you omit the ELSE clause, PL/SQL adds the following implicit ELSE clause:

```
ELSE RAISE CASE_NOT_FOUND;
```

There is always a default action, even when you omit the ELSE clause. If the CASE statement does not match any of the WHEN clauses and you omit the ELSE clause, PL/SQL raises the predefined exception CASE_NOT_FOUND.

The keywords END CASE terminate the CASE statement. These two keywords must be separated by a space. The CASE statement has the following form:

Like PL/SQL blocks, CASE statements can be labeled. The label, an undeclared identifier enclosed by double angle brackets, must appear at the beginning of the CASE statement. Optionally, the label name can also appear at the end of the CASE statement.

Exceptions raised during the execution of a CASE statement are handled in the usual way. That is, normal execution stops and control transfers to the exception-handling part of your PL/SQL block or subprogram.

An alternative to the CASE statement is the CASE expression, where each WHEN clause is an expression. For details, see "CASE Expressions".

## Searched CASE Statement

PL/SQL also provides a searched CASE statement, similar to the simple CASE statement, which has the form shown in Example 4-7.

The searched CASE statement has no selector. Also, its WHEN clauses contain search conditions that yield a Boolean value, not expressions that can yield a value of any type. as shown in Example 4-7.

**Example 4-7 Using the Searched CASE Statement**

```
DECLARE
  grade CHAR(1);
BEGIN
  grade := 'B';
  CASE
    WHEN grade = 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
    WHEN grade = 'B' THEN DBMS_OUTPUT.PUT_LINE('Very Good');
    WHEN grade = 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
    WHEN grade = 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
    WHEN grade = 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
    ELSE DBMS_OUTPUT.PUT_LINE('No such grade');
  END CASE;
END;
-- rather than using the ELSE in the CASE, could use the following
--  EXCEPTION
--    WHEN CASE_NOT_FOUND THEN
```

```
--      DBMS_OUTPUT.PUT_LINE('No such grade');
/
```

The search conditions are evaluated sequentially. The Boolean value of each search condition determines which WHEN clause is executed. If a search condition yields TRUE, its WHEN clause is executed. If any WHEN clause is executed, control passes to the next statement, so subsequent search conditions are not evaluated.

If none of the search conditions yields TRUE, the ELSE clause is executed. The ELSE clause is optional. However, if you omit the ELSE clause, PL/SQL adds the following implicit ELSE clause:

```
ELSE RAISE CASE_NOT_FOUND;
```

Exceptions raised during the execution of a searched CASE statement are handled in the usual way. That is, normal execution stops and control transfers to the exception-handling part of your PL/SQL block or subprogram.

## Guidelines for PL/SQL Conditional Statements

Avoid clumsy IF statements like those in the following example:

```
IF new_balance < minimum_balance THEN
  overdrawn := TRUE;
ELSE
  overdrawn := FALSE;
END IF;
  ...
IF overdrawn = TRUE THEN
  RAISE insufficient_funds;
END IF;
```

The value of a Boolean expression can be assigned directly to a Boolean variable. You can replace the first IF statement with a simple assignment:

```
overdrawn := new_balance < minimum_balance;
```

A Boolean variable is itself either true or false. You can simplify the condition in the second IF statement:

```
IF overdrawn THEN ...
```

When possible, use the ELSIF clause instead of nested IF statements. Your code will be easier to read and understand. Compare the following IF statements:

```
IF condition1 THEN statement1;
  ELSE IF condition2 THEN statement2;
    ELSE IF condition3 THEN statement3; END IF;
  END IF;
END IF;

IF condition1 THEN statement1;
  ELSEIF condition2 THEN statement2;
  ELSEIF condition3 THEN statement3;
END IF;
```

These statements are logically equivalent, but the second statement makes the logic clearer.

To compare a single expression to multiple values, you can simplify the logic by using a single `CASE` statement instead of an `IF` with several `ELSIF` clauses.

# Controlling Loop Iterations: LOOP and EXIT Statements

`LOOP` statements execute a sequence of statements multiple times. There are three forms of `LOOP` statements: `LOOP`, `WHILE-LOOP`, and `FOR-LOOP`. For a description of the syntax of the `LOOP` statement, see "LOOP Statements".

## Using the LOOP Statement

The simplest form of `LOOP` statement is the basic loop, which encloses a sequence of statements between the keywords `LOOP` and `END LOOP`, as follows:

```
LOOP
  sequence_of_statements
END LOOP;
```

With each iteration of the loop, the sequence of statements is executed, then control resumes at the top of the loop. You use an `EXIT` statement to stop looping and prevent an infinite loop. You can place one or more `EXIT` statements anywhere inside a loop, but not outside a loop. There are two forms of `EXIT` statements: `EXIT` and `EXIT-WHEN`.

## Using the EXIT Statement

The `EXIT` statement forces a loop to complete unconditionally. When an `EXIT` statement is encountered, the loop completes immediately and control passes to the next statement as shown in Example 4-8.

**Example 4-8 Using an EXIT Statement**

```
DECLARE
  credit_rating NUMBER := 0;
BEGIN
  LOOP
    credit_rating := credit_rating + 1;
    IF credit_rating > 3 THEN
       EXIT;  -- exit loop immediately
    END IF;
 END LOOP;
 -- control resumes here
 DBMS_OUTPUT.PUT_LINE ('Credit rating: ' || TO_CHAR(credit_rating));
 IF credit_rating > 3 THEN
       RETURN;  -- use RETURN not EXIT when outside a LOOP
 END IF;
 DBMS_OUTPUT.PUT_LINE ('Credit rating: ' || TO_CHAR(credit_rating));
END;
/
```

Remember, the EXIT statement must be placed inside a loop. To complete a PL/SQL block before its normal end is reached, you can use the RETURN statement. For more information, see "Using the RETURN Statement".

## Using the EXIT-WHEN Statement

The EXIT-WHEN statement lets a loop complete conditionally. When the EXIT statement is encountered, the condition in the WHEN clause is evaluated. If the condition is true, the loop completes and control passes to the next statement after the loop. See Example 1-10 for an example that uses the EXIT-WHEN statement.

Until the condition is true, the loop cannot complete. A statement inside the loop must change the value of the condition. In the previous example, if the FETCH statement returns a row, the condition is false. When the FETCH statement fails to return a row, the condition is true, the loop completes, and control passes to the CLOSE statement.

The EXIT-WHEN statement replaces a simple IF statement. For example, compare the following statements:

```
IF count > 100 THEN EXIT; ENDIF;
EXIT WHEN count > 100;
```

These statements are logically equivalent, but the EXIT-WHEN statement is easier to read and

understand.

## Labeling a PL/SQL Loop

Like PL/SQL blocks, loops can be labeled. The optional label, an undeclared identifier enclosed by double angle brackets, must appear at the beginning of the LOOP statement. The label name can also appear at the end of the LOOP statement. When you nest labeled loops, use ending label names to improve readability.

With either form of EXIT statement, you can complete not only the current loop, but any enclosing loop. Simply label the enclosing loop that you want to complete. Then, use the label in an EXIT statement, as shown in Example 4-9. Every enclosing loop up to and including the labeled loop is exited.

**Example 4-9 Using EXIT With Labeled Loops**

```
DECLARE
  s      PLS_INTEGER := 0;
  i      PLS_INTEGER := 0;
  j      PLS_INTEGER;
BEGIN
  <<outer_loop>>
  LOOP
    i := i + 1;
    j := 0;
    <<inner_loop>>
    LOOP
      j := j + 1;
      s := s + i * j; -- sum a bunch of products
      EXIT inner_loop WHEN (j > 5);
      EXIT outer_loop WHEN ((i * j) > 15);
    END LOOP inner_loop;
  END LOOP outer_loop;
  DBMS_OUTPUT.PUT_LINE('The sum of products equals: ' || TO_CHAR(s));
END;
/
```

## Using the WHILE-LOOP Statement

The WHILE-LOOP statement executes the statements in the loop body as long as a condition is true:

```
WHILE condition LOOP
  sequence_of_statements
END LOOP;
```

Before each iteration of the loop, the condition is evaluated. If it is TRUE, the sequence of statements is executed, then control resumes at the top of the loop. If it is FALSE or NULL, the loop is skipped and control passes to the next statement. See Example 1-9 for an example using the WHILE-LOOP statement.

The number of iterations depends on the condition and is unknown until the loop completes. The condition is tested at the top of the loop, so the sequence might execute zero times.

Some languages have a LOOP UNTIL or REPEAT UNTIL structure, which tests the condition at the bottom of the loop instead of at the top, so that the sequence of statements is executed at least once. The equivalent in PL/SQL would be:

```
LOOP
  sequence_of_statements
  EXIT WHEN boolean_expression;
END LOOP;
```

To ensure that a WHILE loop executes at least once, use an initialized Boolean variable in the condition, as follows:

```
done := FALSE;
WHILE NOT done LOOP
  sequence_of_statements
  done := boolean_expression;
END LOOP;
```

A statement inside the loop must assign a new value to the Boolean variable to avoid an infinite loop.

## Using the FOR-LOOP Statement

Simple FOR loops iterate over a specified range of integers. The number of iterations is known before the loop is entered. A double dot (..) serves as the range operator. The range is evaluated when the FOR loop is first entered and is never re-evaluated. If the lower bound equals the higher bound, the loop body is executed once.

As Example 4-10 shows, the sequence of statements is executed once for each integer in the range 1 to 500. After each iteration, the loop counter is incremented.

**Example 4-10 Using a Simple FOR..LOOP Statement**

```
DECLARE
  p      NUMBER := 0;
BEGIN
```

```
  FOR k IN 1..500 LOOP -- calculate pi with 500 terms
    p := p +  (  ( (-1) ** (k + 1) ) / ((2 * k) - 1) );
  END LOOP;
  p := 4 * p;
  DBMS_OUTPUT.PUT_LINE( 'pi is approximately : ' || p ); -- print result
END;
/
```

By default, iteration proceeds upward from the lower bound to the higher bound. If you use the keyword REVERSE, iteration proceeds downward from the higher bound to the lower bound. After each iteration, the loop counter is decremented. You still write the range bounds in ascending (not descending) order.

**Example 4-11 Using a Reverse FOR..LOOP Statement**

```
BEGIN
  FOR i IN REVERSE 1..3 LOOP  -- assign the values 1,2,3 to i
    DBMS_OUTPUT.PUT_LINE (TO_CHAR(i));
  END LOOP;
END;
/
```

Inside a FOR loop, the counter can be read but cannot be changed.

```
BEGIN
  FOR i IN 1..3 LOOP  -- assign the values 1,2,3 to i
    IF i < 3 THEN
       DBMS_OUTPUT.PUT_LINE (TO_CHAR(i));
    ELSE
       i := 2; -- not allowed, raises an error
    END IF;
  END LOOP;
END;
/
```

A useful variation of the FOR loop uses a SQL query instead of a range of integers. This technique lets you run a query and process all the rows of the result set with straightforward syntax. For details, see "Querying Data with PL/SQL: Implicit Cursor FOR Loop".

## How PL/SQL Loops Iterate

The bounds of a loop range can be literals, variables, or expressions but must evaluate to numbers. Otherwise, PL/SQL raises the predefined exception VALUE_ERROR. The lower bound need not be 1, but

the loop counter increment or decrement must be 1.

```
j IN -5..5
k IN REVERSE first..last
step IN 0..TRUNC(high/low) * 2
```

Internally, PL/SQL assigns the values of the bounds to temporary `PLS_INTEGER` variables, and, if necessary, rounds the values to the nearest integer. The magnitude range of a `PLS_INTEGER` is -2147483648 to 2147483647, represented in 32 bits. If a bound evaluates to a number outside that range, you get a numeric overflow error when PL/SQL attempts the assignment. See "PLS_INTEGER Datatype".

Some languages provide a `STEP` clause, which lets you specify a different increment (5 instead of 1 for example). PL/SQL has no such structure, but you can easily build one. Inside the `FOR` loop, simply multiply each reference to the loop counter by the new increment. In Example 4-12, you assign today's date to elements 5, 10, and 15 of an index-by table:

**Example 4-12 Changing the Increment of the Counter in a FOR..LOOP Statement**

```
DECLARE
   TYPE DateList IS TABLE OF DATE INDEX BY PLS_INTEGER;
   dates DateList;
   k CONSTANT INTEGER := 5;  -- set new increment
BEGIN
   FOR j IN 1..3 LOOP
      dates(j*k) := SYSDATE;  -- multiply loop counter by increment
   END LOOP;
END;
/
```

## Dynamic Ranges for Loop Bounds

PL/SQL lets you specify the loop range at run time by using variables for bounds as shown in Example 4-13.

**Example 4-13 Specifying a LOOP Range at Run Time**

```
CREATE TABLE temp (emp_no NUMBER, email_addr VARCHAR2(50));
DECLARE
  emp_count NUMBER;
BEGIN
  SELECT COUNT(employee_id) INTO emp_count FROM employees;
  FOR i IN 1..emp_count LOOP
   INSERT INTO temp VALUES(i, 'to be added later');
```

```
  END LOOP;
  COMMIT;
END;
/
```

If the lower bound of a loop range evaluates to a larger integer than the upper bound, the loop body is not executed and control passes to the next statement:

```
-- limit becomes 1
FOR i IN 2..limit LOOP
  sequence_of_statements -- executes zero times
END LOOP;
-- control passes here
```

## Scope of the Loop Counter Variable

The loop counter is defined only within the loop. You cannot reference that variable name outside the loop. After the loop exits, the loop counter is undefined:

**Example 4-14 Scope of the LOOP Counter Variable**

```
BEGIN
  FOR i IN 1..3 LOOP  -- assign the values 1,2,3 to i
    DBMS_OUTPUT.PUT_LINE (TO_CHAR(i));
  END LOOP;
  DBMS_OUTPUT.PUT_LINE (TO_CHAR(i)); -- raises an error
END;
/
```

You do not need to declare the loop counter because it is implicitly declared as a local variable of type `INTEGER`. It is safest not to use the name of an existing variable, because the local declaration hides any global declaration.

```
DECLARE
  i NUMBER := 5;
BEGIN
  FOR i IN 1..3 LOOP  -- assign the values 1,2,3 to i
    DBMS_OUTPUT.PUT_LINE (TO_CHAR(i));
  END LOOP;
  DBMS_OUTPUT.PUT_LINE (TO_CHAR(i)); -- refers to original variable value (5)
END;
/
```

To reference the global variable in this example, you must use a label and dot notation, as shown in Example 4-15.

**Example 4-15 Using a Label for Referencing Variables Outside a Loop**

```
<<main>>
DECLARE
  i NUMBER := 5;
BEGIN
  FOR i IN 1..3 LOOP  -- assign the values 1,2,3 to i
    DBMS_OUTPUT.PUT_LINE( 'local: ' || TO_CHAR(i)
                        || ' global: ' || TO_CHAR(main.i));
  END LOOP;
END main;
/
```

The same scope rules apply to nested FOR loops. In Example 4-16 both loop counters have the same name. To reference the outer loop counter from the inner loop, you use a label and dot notation.

**Example 4-16 Using Labels on Loops for Referencing**

```
BEGIN
<<outer_loop>>
  FOR i IN 1..3 LOOP  -- assign the values 1,2,3 to i
    <<inner_loop>>
    FOR i IN 1..3 LOOP
      IF outer_loop.i = 2 THEN
        DBMS_OUTPUT.PUT_LINE( 'outer: ' || TO_CHAR(outer_loop.i) || ' inner: '
                            || TO_CHAR(inner_loop.i));
      END IF;
    END LOOP inner_loop;
  END LOOP outer_loop;
END;
/
```

## Using the EXIT Statement in a FOR Loop

The EXIT statement lets a FOR loop complete early. In Example 4-17, the loop normally executes ten times, but as soon as the FETCH statement fails to return a row, the loop completes no matter how many times it has executed.

**Example 4-17 Using EXIT in a LOOP**

```
DECLARE
   v_employees employees%ROWTYPE;  -- declare record variable
   CURSOR c1 is SELECT * FROM employees;
BEGIN
  OPEN c1; -- open the cursor before fetching
-- An entire row is fetched into the v_employees record
  FOR i IN 1..10 LOOP
    FETCH c1 INTO v_employees;
    EXIT WHEN c1%NOTFOUND;
    -- process data here
  END LOOP;
  CLOSE c1;
END;
/
```

Suppose you must exit early from a nested FOR loop. To complete not only the current loop, but also any enclosing loop, label the enclosing loop and use the label in an EXIT statement as shown in Example 4-18.

**Example 4-18 Using EXIT With a Label in a LOOP**

```
DECLARE
   v_employees employees%ROWTYPE;  -- declare record variable
   CURSOR c1 is SELECT * FROM employees;
BEGIN
  OPEN c1; -- open the cursor before fetching
-- An entire row is fetched into the v_employees record
<<outer_loop>>
  FOR i IN 1..10 LOOP
    -- process data here
    FOR j IN 1..10 LOOP
      FETCH c1 INTO v_employees;
      EXIT WHEN c1%NOTFOUND;
      -- process data here
    END LOOP;
  END LOOP outer_loop;
  CLOSE c1;
END;
/
```

See also Example 6-10.

# Sequential Control: GOTO and NULL Statements

Unlike the IF and LOOP statements, the GOTO and NULL statements are not crucial to PL/SQL

programming. The GOTO statement is seldom needed. Occasionally, it can simplify logic enough to warrant its use. The NULL statement can improve readability by making the meaning and action of conditional statements clear.

Overuse of GOTO statements can result in code that is hard to understand and maintain. Use GOTO statements sparingly. For example, to branch from a deeply nested structure to an error-handling routine, raise an exception rather than use a GOTO statement. PL/SQL's exception-handling mechanism is discussed in Chapter 10, "Handling PL/SQL Errors".

## Using the GOTO Statement

The GOTO statement branches to a label unconditionally. The label must be unique within its scope and must precede an executable statement or a PL/SQL block. When executed, the GOTO statement transfers control to the labeled statement or block. The labeled statement or block can be down or up in the sequence of statements. In Example 4-19 you go to a PL/SQL block up in the sequence of statements.

### Example 4-19 Using a Simple GOTO Statement

```
DECLARE
  p        VARCHAR2(30);
  n        PLS_INTEGER := 37; -- test any integer > 2 for prime
BEGIN
  FOR j in 2..ROUND(SQRT(n)) LOOP
    IF n MOD j = 0 THEN -- test for prime
      p := ' is not a prime number'; -- not a prime number
      GOTO print_now;
    END IF;
  END LOOP;
  p := ' is a prime number';
<<print_now>>
  DBMS_OUTPUT.PUT_LINE(TO_CHAR(n) || p);
END;
/
```

The label end_loop in the Example 4-20 is not allowed unless it is preceded by an executable statement. To make the label legal, a NULL statement is added.

### Example 4-20 Using a NULL Statement to Allow a GOTO to a Label

```
DECLARE
   done  BOOLEAN;
BEGIN
   FOR i IN 1..50 LOOP
      IF done THEN
```

```
         GOTO end_loop;
      END IF;
   <<end_loop>>  -- not allowed unless an executable statement follows
   NULL; -- add NULL statement to avoid error
   END LOOP;  -- raises an error without the previous NULL
END;
/
```

Example 4-21 shows a GOTO statement can branch to an enclosing block from the current block.

**Example 4-21 Using a GOTO Statement to Branch an Enclosing Block**

```
-- example with GOTO statement
DECLARE
   v_last_name  VARCHAR2(25);
   v_emp_id     NUMBER(6) := 120;
BEGIN
   <<get_name>>
   SELECT last_name INTO v_last_name FROM employees
          WHERE employee_id = v_emp_id;
   BEGIN
      DBMS_OUTPUT.PUT_LINE (v_last_name);
      v_emp_id := v_emp_id + 5;
      IF v_emp_id < 120 THEN
        GOTO get_name;  -- branch to enclosing block
      END IF;
   END;
END;
/
```

The GOTO statement branches to the first enclosing block in which the referenced label appears.

## Restrictions on the GOTO Statement

Some possible destinations of a GOTO statement are not allowed. Specifically, a GOTO statement cannot branch into an IF statement, CASE statement, LOOP statement, or sub-block. For example, the following GOTO statement is not allowed:

```
BEGIN
  GOTO update_row; -- cannot branch into IF statement
  IF valid THEN
    <<update_row>>
    UPDATE emp SET ...
```

```
  END IF;
END;
```

A GOTO statement cannot branch from one IF statement clause to another, or from one CASE statement WHEN clause to another.

A GOTO statement cannot branch from an outer block into a sub-block (that is, an inner BEGIN-END block).

A GOTO statement cannot branch out of a subprogram. To end a subprogram early, you can use the RETURN statement or use GOTO to branch to a place right before the end of the subprogram.

A GOTO statement cannot branch from an exception handler back into the current BEGIN-END block. However, a GOTO statement can branch from an exception handler into an enclosing block.

## Using the NULL Statement

The NULL statement does nothing, and passes control to the next statement. Some languages refer to such an instruction as a no-op (no operation). See "NULL Statement".

In Example 4-22, the NULL statement emphasizes that only salespeople receive commissions.

**Example 4-22 Using the NULL Statement to Show No Action**

```
DECLARE
  v_job_id  VARCHAR2(10);
  v_emp_id  NUMBER(6) := 110;
BEGIN
  SELECT job_id INTO v_job_id FROM employees WHERE employee_id = v_emp_id;
  IF v_job_id = 'SA_REP' THEN
    UPDATE employees SET commission_pct = commission_pct * 1.2;
  ELSE
    NULL; -- do nothing if not a sales representative
  END IF;
END;
/
```

The NULL statement is a handy way to create placeholders and stub procedures. In Example 4-23, the NULL statement lets you compile this procedure, then fill in the real body later. Note that the use of the NULL statement might raise an `unreachable code` warning if warnings are enabled. See "Overview of PL/SQL Compile-Time Warnings".

**Example 4-23 Using NULL as a Placeholder When Creating a Subprogram**

```
CREATE OR REPLACE PROCEDURE award_bonus (emp_id NUMBER, bonus NUMBER) AS
```

```
BEGIN  -- executable part starts here
  NULL; -- use NULL as placeholder, raises "unreachable code" if warnings enabled
END award_bonus;
/
```

You can use the `NULL` statement to indicate that you are aware of a possibility, but no action is necessary. In the following exception block, the `NULL` statement shows that you have chosen not to take any action for unnamed exceptions:

```
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    ROLLBACK;
  WHEN OTHERS THEN
    NULL;
END;
```

See Example 1-12, "Creating a Stored Subprogram".