4 PL/SQL Control Statements

PL/SQL has three categories of control statements:

- **Conditional selection statements**, which run different statements for different data values.

  The conditional selection statements are `IF` and and `CASE`.

- **Loop statements**, which run the same statements with a series of different data values.

  The loop statements are the basic `LOOP`, `FOR LOOP`, and `WHILE LOOP`.

  The `EXIT` statement transfers control to the end of a loop. The `CONTINUE` statement exits the current iteration of a loop and transfers control to the next iteration. Both `EXIT` and `CONTINUE` have an optional `WHEN` clause, where you can specify a condition.

- **Sequential control statements**, which are not crucial to PL/SQL programming.

  The sequential control statements are `GOTO`, which goes to a specified statement, and `NULL`, which does nothing.

Topics

Conditional Selection Statements

The **conditional selection statements**, `IF` and `CASE`, run different statements for different data values.

The `IF` statement either runs or skips a sequence of one or more statements, depending on a condition. The `IF` statement has these forms:

- `IF THEN`
- `IF THEN ELSE`
- `IF THEN ELSIF`

The `CASE` statement chooses from a sequence of conditions, and runs the corresponding statement. The `CASE` statement has these forms:

- Simple, which evaluates a single expression and compares it to several potential values.

- Searched, which evaluates multiple conditions and chooses the first one that is true.

The `CASE` statement is appropriate when a different action is to be taken for each alternative.

Topics

- IF THEN Statement
- IF THEN ELSE Statement
- IF THEN ELSIF Statement
- Simple CASE Statement
- Searched CASE Statement

IF THEN Statement

The `IF THEN` statement has this structure:

```
IF condition THEN
  statements
END IF;
```

If the *`condition`* is true, the *`statements`* run; otherwise, the `IF` statement does nothing. (For complete syntax, see "IF Statement" on page 13-90.)

In Example 4–1, the statements between `THEN` and `END IF` run if and only if the value of `sales` is greater than `quota`+200.

Example 4–1 IF THEN Statement

```
DECLARE
  PROCEDURE p (
    sales  NUMBER,
    quota  NUMBER,
    emp_id NUMBER
  )
  IS
    bonus    NUMBER := 0;
    updated  VARCHAR2(3) := 'No';
  BEGIN
    IF sales > (quota + 200) THEN
      bonus := (sales - quota)/4;

      UPDATE employees
      SET salary = salary + bonus
      WHERE employee_id = emp_id;

      updated := 'Yes';
    END IF;

    DBMS_OUTPUT.PUT_LINE (
      'Table updated?  ' || updated || ', ' ||
      'bonus = ' || bonus || '.'
    );
  END p;
BEGIN
  p(10100, 10000, 120);
  p(10500, 10000, 121);
END;
```

```
/
```

Result:

```
Table updated?  No, bonus = 0.
Table updated?  Yes, bonus = 125.
```

Tip: Avoid clumsy `IF` statements such as:

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

Instead, assign the value of the `BOOLEAN` expression directly to a `BOOLEAN` variable:

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

A `BOOLEAN` variable is either `TRUE`, `FALSE`, or `NULL`. Do not write:

```
IF overdrawn = TRUE THEN
  RAISE insufficient_funds;
END IF;
```

Instead, write:

```
IF overdrawn THEN
  RAISE insufficient_funds;
END IF;
```

IF THEN ELSE Statement

The `IF THEN ELSE` statement has this structure:

```
IF condition THEN
  statements
ELSE
  else_statements
END IF;
```

If the value of *`condition`* is true, the *`statements`* run; otherwise, the *`else_statements`* run. (For complete syntax, see "IF Statement" on page 13-90.)

In Example 4–2, the statement between `THEN` and `ELSE` runs if and only if the value of `sales` is greater than `quota`+200; otherwise, the statement between `ELSE` and `END IF` runs.

Example 4–2 IF THEN ELSE Statement

```
DECLARE
  PROCEDURE p (
    sales  NUMBER,
    quota  NUMBER,
    emp_id NUMBER
  )
  IS
    bonus  NUMBER := 0;
  BEGIN
    IF sales > (quota + 200) THEN
      bonus := (sales - quota)/4;
    ELSE
      bonus := 50;
    END IF;

    DBMS_OUTPUT.PUT_LINE('bonus = ' || bonus);

    UPDATE employees
    SET salary = salary + bonus
    WHERE employee_id = emp_id;
  END p;
BEGIN
  p(10100, 10000, 120);
  p(10500, 10000, 121);
END;
/
```

Result:

```
bonus = 50
bonus = 125
```

`IF` statements can be nested, as in Example 4–3.

Example 4–3 Nested IF THEN ELSE Statements

```
DECLARE
  PROCEDURE p (
    sales  NUMBER,
    quota  NUMBER,
    emp_id NUMBER
  )
  IS
    bonus  NUMBER := 0;
  BEGIN
    IF sales > (quota + 200) THEN
      bonus := (sales - quota)/4;
    ELSE
      IF sales > quota THEN
        bonus := 50;
      ELSE
        bonus := 0;
      END IF;
    END IF;

    DBMS_OUTPUT.PUT_LINE('bonus = ' || bonus);

    UPDATE employees
    SET salary = salary + bonus
    WHERE employee_id = emp_id;
  END p;
BEGIN
  p(10100, 10000, 120);
  p(10500, 10000, 121);
  p(9500, 10000, 122);
END;
/
```

Result:

```
bonus = 50
bonus = 125
bonus = 0
```

IF THEN ELSIF Statement

The `IF THEN ELSIF` statement has this structure:

```
IF condition_1 THEN
  statements_1
ELSIF condition_2 THEN
  statements_2
[ ELSIF condition_3 THEN
    statements_3
]...
[ ELSE
    else_statements
]
END IF;
```

The `IF THEN ELSIF` statement runs the first *`statements`* for which *`condition`* is true. Remaining conditions are not evaluated. If no *`condition`* is true, the *`else_statements`* run, if they exist; otherwise, the `IF THEN ELSIF` statement does nothing. (For complete syntax, see "IF Statement" on page 13-90.)

In Example 4–4, when the value of `sales` is larger than 50000, both the first and second conditions are true. However, because the first condition is true, `bonus` is assigned the value 1500, and the second condition is never tested. After `bonus` is assigned the value 1500, control passes to the `DBMS_OUTPUT.PUT_LINE` invocation.

Example 4–4 IF THEN ELSIF Statement

```
DECLARE
  PROCEDURE p (sales NUMBER)
  IS
    bonus  NUMBER := 0;
  BEGIN
    IF sales > 50000 THEN
      bonus := 1500;
    ELSIF sales > 35000 THEN
      bonus := 500;
    ELSE
      bonus := 100;
    END IF;

    DBMS_OUTPUT.PUT_LINE (
      'Sales = ' || sales || ', bonus = ' || bonus || '.'
    );
  END p;
BEGIN
  p(55000);
  p(40000);
  p(30000);
END;
/
```

Result:

```
Sales = 55000, bonus = 1500.
Sales = 40000, bonus = 500.
Sales = 30000, bonus = 100.
```

A single IF THEN ELSIF statement is easier to understand than a logically equivalent nested IF THEN ELSE statement:

```
-- IF THEN ELSIF statement

IF condition_1 THEN statements_1;
  ELSIF condition_2 THEN statements_2;
  ELSIF condition_3 THEN statement_3;
END IF;

-- Logically equivalent nested IF THEN ELSE statements

IF condition_1 THEN
  statements_1;
ELSE
  IF condition_2 THEN
    statements_2;
  ELSE
    IF condition_3 THEN
      statements_3;
    END IF;
  END IF;
END IF;
```

Example 4–5 uses an IF THEN ELSIF statement with many ELSIF clauses to compare a single value to many possible values. For this purpose, a simple CASE statement is clearer—see Example 4–6.

Example 4–5 IF THEN ELSIF Statement Simulates Simple CASE 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;
/
```

Result:

```
Very Good
```

Simple CASE Statement

The simple CASE statement has this structure:

```
CASE selector
WHEN selector_value_1 THEN statements_1
WHEN selector_value_2 THEN statements_2
...
WHEN selector_value_n THEN statements_n
[ ELSE
  else_statements ]
END CASE;]
```

The `selector` is an expression (typically a single variable). Each `selector_value` can be either a literal or an expression. (For complete syntax, see "CASE Statement" on page 13-20.)

The simple `CASE` statement runs the first `statements` for which `selector_value` equals `selector`. Remaining conditions are not evaluated. If no `selector_value` equals `selector`, the `CASE` statement runs `else_statements` if they exist and raises the predefined exception `CASE_NOT_FOUND` otherwise.

Example 4–6 uses a simple `CASE` statement to compare a single value to many possible values. The `CASE` statement in Example 4–6 is logically equivalent to the `IF THEN ELSIF` statement in Example 4–5.

Example 4–6 Simple CASE 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;
/
```

Result:

```
Very Good
```

Note: As in a simple `CASE` expression, if the selector in a simple `CASE` statement has the value `NULL`, it cannot be matched by `WHEN NULL` (see Example 2–51, "Simple CASE Expression with WHEN NULL"). Instead, use a searched `CASE` statement with `WHEN` *condition* `IS NULL` (see Example 2–53, "Searched CASE Expression with WHEN ... IS NULL").

Searched CASE Statement

The searched `CASE` statement has this structure:

```
CASE
WHEN condition_1 THEN statements_1
WHEN condition_2 THEN statements_2
...
WHEN condition_n THEN statements_n
[ ELSE
```

```
  else_statements ]
END CASE;]
```

The searched CASE statement runs the first *statements* for which *condition* is true. Remaining conditions are not evaluated. If no *condition* is true, the CASE statement runs *else_statements* if they exist and raises the predefined exception CASE_NOT_FOUND otherwise. (For complete syntax, see "CASE Statement" on page 13-20.)

The searched CASE statement in Example 4–7 is logically equivalent to the simple CASE statement in Example 4–6.

Example 4–7 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;
/
```

Result:

```
Very Good
```

In both Example 4–7 and Example 4–6, the ELSE clause can be replaced by an EXCEPTION part. Example 4–8 is logically equivalent to Example 4–7.

Example 4–8 EXCEPTION Instead of ELSE Clause in 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');
  END CASE;
EXCEPTION
  WHEN CASE_NOT_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('No such grade');
END;
/
```

Result:

```
Very Good
```

LOOP Statements

Loop statements run the same statements with a series of different values. The loop statements are:

- Basic `LOOP`
- `FOR LOOP`
- Cursor `FOR LOOP`
- `WHILE LOOP`

The statements that exit a loop are:

- `EXIT`
- `EXIT WHEN`

The statements that exit the current iteration of a loop are:

- `CONTINUE`
- `CONTINUE WHEN`

`EXIT`, `EXIT WHEN`, `CONTINUE`, and `CONTINUE WHEN` and can appear anywhere inside a loop, but not outside a loop. Oracle recommends using these statements instead of the "GOTO Statement" on page 4-21, which can exit a loop or the current iteration of a loop by transferring control to a statement outside the loop. (A raised exception also exits a loop. For information about exceptions, see "Overview of Exception Handling" on page 11-4.)

`LOOP` statements can be labeled, and `LOOP` statements can be nested. Labels are recommended for nested loops to improve readability. You must ensure that the label in the `END LOOP` statement matches the label at the beginning of the same loop statement (the compiler does not check).

Topics

- Basic LOOP Statement
- EXIT Statement
- EXIT WHEN Statement
- CONTINUE Statement
- CONTINUE WHEN Statement
- FOR LOOP Statement
- WHILE LOOP Statement

For information about the cursor `FOR LOOP`, see "Query Result Set Processing With Cursor FOR LOOP Statements" on page 6-24.

Basic LOOP Statement

The basic `LOOP` statement has this structure:

```
[ label ] LOOP
  statements
END LOOP [ label ];
```

With each iteration of the loop, the *statements* run and control returns to the top of the loop. To prevent an infinite loop, a statement or raised exception must exit the loop.

See Also: "Basic LOOP Statement" on page 13-7

EXIT Statement

The EXIT statement exits the current iteration of a loop unconditionally and transfers control to the end of either the current loop or an enclosing labeled loop.

In Example 4–9, the EXIT statement inside the basic LOOP statement transfers control unconditionally to the end of the current loop.

Example 4–9 Basic LOOP Statement with EXIT Statement

```
DECLARE
  x NUMBER := 0;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE ('Inside loop:  x = ' || TO_CHAR(x));
    x := x + 1;
    IF x > 3 THEN
      EXIT;
    END IF;
  END LOOP;
  -- After EXIT, control resumes here
  DBMS_OUTPUT.PUT_LINE(' After loop:  x = ' || TO_CHAR(x));
END;
/
```

Result:

```
Inside loop:  x = 0
Inside loop:  x = 1
Inside loop:  x = 2
Inside loop:  x = 3
After loop:  x = 4
```

See Also: "EXIT Statement" on page 13-55

EXIT WHEN Statement

The EXIT WHEN statement exits the current iteration of a loop when the condition in its WHEN clause is true, and transfers control to the end of either the current loop or an enclosing labeled loop.

Each time control reaches the EXIT WHEN statement, the condition in its WHEN clause is evaluated. If the condition is not true, the EXIT WHEN statement does nothing. To prevent an infinite loop, a statement inside the loop must make the condition true, as in Example 4–10.

In Example 4–10, the EXIT WHEN statement inside the basic LOOP statement transfers control to the end of the current loop when x is greater than 3. Example 4–10 is logically equivalent to Example 4–9.

Example 4–10 Basic LOOP Statement with EXIT WHEN Statement

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

```
  LOOP
    DBMS_OUTPUT.PUT_LINE('Inside loop:  x = ' || TO_CHAR(x));
    x := x + 1;  -- prevents infinite loop
    EXIT WHEN x > 3;
  END LOOP;
  -- After EXIT statement, control resumes here
  DBMS_OUTPUT.PUT_LINE('After loop:  x = ' || TO_CHAR(x));
END;
/
```

Result:

```
Inside loop:  x = 0
Inside loop:  x = 1
Inside loop:  x = 2
Inside loop:  x = 3
After loop:  x = 4
```

See Also: "EXIT Statement" on page 13-55

In Example 4–11, one basic LOOP statement is nested inside the other, and both have labels. The inner loop has two EXIT WHEN statements; one that exits the inner loop and one that exits the outer loop.

Example 4–11 Nested, Labeled Basic LOOP Statements with EXIT WHEN Statements

```
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 several 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;
/
```

Result:

```
The sum of products equals: 166
```

An EXIT WHEN statement in an inner loop can transfer control to an outer loop only if the outer loop is labeled.

In Example 4–12, the outer loop is not labeled; therefore, the inner loop cannot transfer control to it.

Example 4–12 Nested, Unabeled Basic LOOP Statements with EXIT WHEN Statements

```
DECLARE
```

```
  i PLS_INTEGER := 0;
  j PLS_INTEGER := 0;

BEGIN
  LOOP
    i := i + 1;
    DBMS_OUTPUT.PUT_LINE ('i = ' || i);

    LOOP
      j := j + 1;
      DBMS_OUTPUT.PUT_LINE ('j = ' || j);
      EXIT WHEN (j > 3);
    END LOOP;

    DBMS_OUTPUT.PUT_LINE ('Exited inner loop');

    EXIT WHEN (i > 2);
  END LOOP;

  DBMS_OUTPUT.PUT_LINE ('Exited outer loop');
END;
/
```

Result:

```
i = 1
j = 1
j = 2
j = 3
j = 4
Exited inner loop
i = 2
j = 5
Exited inner loop
i = 3
j = 6
Exited inner loop
Exited outer loop

PL/SQL procedure successfully completed.
```

CONTINUE Statement

The `CONTINUE` statement exits the current iteration of a loop unconditionally and transfers control to the next iteration of either the current loop or an enclosing labeled loop.

In Example 4–13, the `CONTINUE` statement inside the basic `LOOP` statement transfers control unconditionally to the next iteration of the current loop.

Example 4–13 CONTINUE Statement in Basic LOOP Statement

```
DECLARE
  x NUMBER := 0;
BEGIN
  LOOP -- After CONTINUE statement, control resumes here
    DBMS_OUTPUT.PUT_LINE ('Inside loop:  x = ' || TO_CHAR(x));
    x := x + 1;
    IF x < 3 THEN
      CONTINUE;
    END IF;
    DBMS_OUTPUT.PUT_LINE
      ('Inside loop, after CONTINUE:  x = ' || TO_CHAR(x));
    EXIT WHEN x = 5;
  END LOOP;

  DBMS_OUTPUT.PUT_LINE (' After loop:  x = ' || TO_CHAR(x));
END;
/
```

Result:

```
Inside loop:  x = 0
Inside loop:  x = 1
Inside loop:  x = 2
Inside loop, after CONTINUE:  x = 3
Inside loop:  x = 3
Inside loop, after CONTINUE:  x = 4
Inside loop:  x = 4
Inside loop, after CONTINUE:  x = 5
After loop:  x = 5
```

See Also: "CONTINUE Statement" on page 13-38

CONTINUE WHEN Statement

The CONTINUE WHEN statement exits the current iteration of a loop when the condition in its WHEN clause is true, and transfers control to the next iteration of either the current loop or an enclosing labeled loop.

Each time control reaches the CONTINUE WHEN statement, the condition in its WHEN clause is evaluated. If the condition is not true, the CONTINUE WHEN statement does nothing.

In Example 4–14, the CONTINUE WHEN statement inside the basic LOOP statement transfers control to the next iteration of the current loop when x is less than 3. Example 4–14 is logically equivalent to Example 4–13.

Example 4–14 CONTINUE WHEN Statement in Basic LOOP Statement

```
DECLARE
  x NUMBER := 0;
BEGIN
  LOOP -- After CONTINUE statement, control resumes here
    DBMS_OUTPUT.PUT_LINE ('Inside loop:  x = ' || TO_CHAR(x));
    x := x + 1;
    CONTINUE WHEN x < 3;
    DBMS_OUTPUT.PUT_LINE
      ('Inside loop, after CONTINUE:  x = ' || TO_CHAR(x));
    EXIT WHEN x = 5;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE (' After loop:  x = ' || TO_CHAR(x));
END;
/
```

Result:

```
Inside loop:  x = 0
Inside loop:  x = 1
Inside loop:  x = 2
Inside loop, after CONTINUE:  x = 3
Inside loop:  x = 3
Inside loop, after CONTINUE:  x = 4
Inside loop:  x = 4
Inside loop, after CONTINUE:  x = 5
After loop:  x = 5
```

> **See Also:** "CONTINUE Statement" on page 13-38

FOR LOOP Statement

The FOR LOOP statement runs one or more statements while the loop index is in a specified range. The statement has this structure:

```
[ label ] FOR index IN [ REVERSE ] lower_bound..upper_bound LOOP
  statements
END LOOP [ label ];
```

Without REVERSE, the value of *index* starts at *lower_bound* and increases by one with each iteration of the loop until it reaches *upper_bound*. If *lower_bound* is greater than *upper_bound*, then the *statements* never run.

With REVERSE, the value of *index* starts at *upper_bound* and decreases by one with each iteration of the loop until it reaches *lower_bound*. If *upper_bound* is less than *lower_bound*, then the *statements* never run.

An EXIT, EXIT WHEN, CONTINUE, or CONTINUE WHEN in the *statements* can cause the loop or the current iteration of the loop to end early.

> **Tip:** To process the rows of a query result set, use a cursor FOR LOOP, which has a query instead of a range of integers. For details, see "Query Result Set Processing With Cursor FOR LOOP Statements" on page 6-24.

> **See Also:** "FOR LOOP Statement" on page 13-74

In Example 4–15, *index* is i, *lower_bound* is 1, and *upper_bound* is 3. The loop prints the numbers from 1 to 3.

Example 4–15 FOR LOOP Statements

```
BEGIN
  DBMS_OUTPUT.PUT_LINE ('lower_bound < upper_bound');

  FOR i IN 1..3 LOOP
    DBMS_OUTPUT.PUT_LINE (i);
  END LOOP;

  DBMS_OUTPUT.PUT_LINE ('lower_bound = upper_bound');

  FOR i IN 2..2 LOOP
    DBMS_OUTPUT.PUT_LINE (i);
  END LOOP;

  DBMS_OUTPUT.PUT_LINE ('lower_bound > upper_bound');

  FOR i IN 3..1 LOOP
    DBMS_OUTPUT.PUT_LINE (i);
  END LOOP;
END;
/
```

Result:

```
lower_bound < upper_bound
1
2
3
lower_bound = upper_bound
2
lower_bound > upper_bound
```

The FOR LOOP statement in Example 4–16 is the reverse of the one in Example 4–15: It prints the numbers from 3 to 1.

Example 4–16 Reverse FOR LOOP Statements

```
BEGIN
  DBMS_OUTPUT.PUT_LINE ('upper_bound > lower_bound');

  FOR i IN REVERSE 1..3 LOOP
    DBMS_OUTPUT.PUT_LINE (i);
  END LOOP;

  DBMS_OUTPUT.PUT_LINE ('upper_bound = lower_bound');

  FOR i IN REVERSE 2..2 LOOP
    DBMS_OUTPUT.PUT_LINE (i);
  END LOOP;

  DBMS_OUTPUT.PUT_LINE ('upper_bound < lower_bound');

  FOR i IN REVERSE 3..1 LOOP
    DBMS_OUTPUT.PUT_LINE (i);
  END LOOP;
END;
/
```

Result:

```
upper_bound > lower_bound
3
2
1
upper_bound = lower_bound
2
upper_bound < lower_bound
```

In some languages, the FOR LOOP has a STEP clause that lets you specify a loop index increment other than 1. To simulate the STEP clause in PL/SQL, multiply each reference to the loop index by the desired increment.

In Example 4–17, the FOR LOOP effectively increments the index by five.

Example 4–17 Simulating STEP Clause in FOR LOOP Statement

```
DECLARE
  step  PLS_INTEGER := 5;
BEGIN
  FOR i IN 1..3 LOOP
    DBMS_OUTPUT.PUT_LINE (i*step);
  END LOOP;
END;
/
```

Result:

```
5
10
15
```

Topics

- FOR LOOP Index
- Lower Bound and Upper Bound
- EXIT WHEN or CONTINUE WHEN Statement in FOR LOOP Statement

FOR LOOP Index

The index of a `FOR LOOP` statement is implicitly declared as a variable of type `PLS_INTEGER` that is local to the loop. The statements in the loop can read the value of the index, but cannot change it. Statements outside the loop cannot reference the index. After the `FOR LOOP` statement runs, the index is undefined. (A loop index is sometimes called a loop counter.)

In Example 4–18, the `FOR LOOP` statement tries to change the value of its index, causing an error.

Example 4–18 FOR LOOP Statement Tries to Change Index Value

```
BEGIN
  FOR i IN 1..3 LOOP
    IF i < 3 THEN
      DBMS_OUTPUT.PUT_LINE (TO_CHAR(i));
    ELSE
      i := 2;
    END IF;
  END LOOP;
END;
/
```

Result:

```
       i := 2;
       *
ERROR at line 6:
ORA-06550: line 6, column 8:
PLS-00363: expression 'I' cannot be used as an assignment target
ORA-06550: line 6, column 8:
PL/SQL: Statement ignored
```

In Example 4–19, a statement outside the `FOR LOOP` statement references the loop index, causing an error.

Example 4–19 Outside Statement References FOR LOOP Statement Index

```
BEGIN
  FOR i IN 1..3 LOOP
    DBMS_OUTPUT.PUT_LINE ('Inside loop, i is ' || TO_CHAR(i));
  END LOOP;

  DBMS_OUTPUT.PUT_LINE ('Outside loop, i is ' || TO_CHAR(i));
END;
/
```

Result:

```
  DBMS_OUTPUT.PUT_LINE ('Outside loop, i is ' || TO_CHAR(i));
                                                         *
ERROR at line 6:
ORA-06550: line 6, column 58:
PLS-00201: identifier 'I' must be declared
ORA-06550: line 6, column 3:
PL/SQL: Statement ignored
```

If the index of a `FOR LOOP` statement has the same name as a variable declared in an enclosing block, the local implicit declaration hides the other declaration, as Example 4–20 shows.

Example 4–20 FOR LOOP Statement Index with Same Name as Variable

```
DECLARE
  i NUMBER := 5;
BEGIN
  FOR i IN 1..3 LOOP
    DBMS_OUTPUT.PUT_LINE ('Inside loop, i is ' || TO_CHAR(i));
  END LOOP;

  DBMS_OUTPUT.PUT_LINE ('Outside loop, i is ' || TO_CHAR(i));
END;
/
```

Result:

```
Inside loop, i is 1
Inside loop, i is 2
Inside loop, i is 3
Outside loop, i is 5
```

Example 4–21 shows how to change Example 4–20 to allow the statement inside the loop to reference the variable declared in the enclosing block.

Example 4–21 FOR LOOP Statement References Variable with Same Name as Index

```
<<main>>  -- Label block.
DECLARE
  i NUMBER := 5;
BEGIN
  FOR i IN 1..3 LOOP
    DBMS_OUTPUT.PUT_LINE (
      'local: ' || TO_CHAR(i) || ', global: ' ||
      TO_CHAR(main.i)  -- Qualify reference with block label.
    );
  END LOOP;
END main;
/
```

Result:

```
local: 1, global: 5
local: 2, global: 5
local: 3, global: 5
```

In Example 4–22, the indexes of the nested `FOR LOOP` statements have the same name. The inner loop references the index of the outer loop by qualifying the reference with the label of the outer loop. For clarity only, the inner loop also qualifies the reference to its own index with its own label.

Example 4–22 Nested FOR LOOP Statements with Same Index Name

```
BEGIN
  <<outer_loop>>
  FOR i IN 1..3 LOOP
    <<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;
/
```

Result:

```
outer: 2 inner: 1
outer: 2 inner: 2
outer: 2 inner: 3
```

Lower Bound and Upper Bound

The lower and upper bounds of a `FOR LOOP` statement can be either numeric literals, numeric variables, or numeric expressions. If a bound does not have a numeric value, then PL/SQL raises the predefined exception `VALUE_ERROR`.

Example 4–23 FOR LOOP Statement Bounds

```
DECLARE
  first  INTEGER := 1;
  last   INTEGER := 10;
  high   INTEGER := 100;
  low    INTEGER := 12;
BEGIN
  -- Bounds are numeric literals:
  FOR j IN -5..5 LOOP
    NULL;
  END LOOP;

  -- Bounds are numeric variables:
  FOR k IN REVERSE first..last LOOP
    NULL;
  END LOOP;

 -- Lower bound is numeric literal,
 -- Upper bound is numeric expression:
  FOR step IN 0..(TRUNC(high/low) * 2) LOOP
    NULL;
  END LOOP;
END;
/
```

In Example 4–24, the upper bound of the `FOR LOOP` statement is a variable whose value is determined at run time.

Example 4–24 Specifying FOR LOOP Statement Bounds at Run Time

```
DROP TABLE temp;
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 (emp_no, email_addr)
    VALUES(i, 'to be added later');
  END LOOP;
END;
/
```

EXIT WHEN or CONTINUE WHEN Statement in FOR LOOP Statement

Suppose that you must exit a FOR LOOP statement immediately if a certain condition arises. You can put the condition in an EXIT WHEN statement inside the FOR LOOP statement.

In Example 4–25, the FOR LOOP statement executes 10 times unless the FETCH statement inside it fails to return a row, in which case it ends immediately.

Example 4–25 EXIT WHEN Statement in FOR LOOP Statement

```
DECLARE
  v_employees employees%ROWTYPE;
  CURSOR c1 is SELECT * FROM employees;
BEGIN
  OPEN c1;
  -- Fetch entire row into 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;
/
```

Now suppose that the FOR LOOP statement that you must exit early is nested inside another FOR LOOP statement. If, when you exit the inner loop early, you also want to exit the outer loop, then label the outer loop and specify its name in the EXIT WHEN statement, as in Example 4–26.

If you want to exit the inner loop early but complete the current iteration of the outer loop, then label the outer loop and specify its name in the CONTINUE WHEN statement, as in Example 4–27.

Example 4–26 EXIT WHEN Statement in Inner FOR LOOP Statement

```
DECLARE
  v_employees employees%ROWTYPE;
  CURSOR c1 is SELECT * FROM employees;
```

```
BEGIN
  OPEN c1;

  -- Fetch entire row into 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 outer_loop WHEN c1%NOTFOUND;
      -- Process data here
    END LOOP;
  END LOOP outer_loop;

  CLOSE c1;
END;
/
```

Example 4–27 CONTINUE WHEN Statement in Inner FOR LOOP Statement

```
DECLARE
  v_employees employees%ROWTYPE;
  CURSOR c1 is SELECT * FROM employees;
BEGIN
  OPEN c1;

  -- Fetch entire row into 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;
      CONTINUE outer_loop WHEN c1%NOTFOUND;
      -- Process data here
    END LOOP;
  END LOOP outer_loop;

  CLOSE c1;
END;
/
```

See Also: "Overview of Exception Handling" on page 11-4 for information about exceptions, which can also cause a loop to end immediately if a certain condition arises

WHILE LOOP Statement

The `WHILE LOOP` statement runs one or more statements while a condition is true. It has this structure:

```
[ label ] WHILE condition LOOP
  statements
END LOOP [ label ];
```

If the *condition* is true, the *statements* run and control returns to the top of the loop, where *condition* is evaluated again. If the *condition* is not true, control transfers to the statement after the `WHILE LOOP` statement. To prevent an infinite loop, a statement inside the loop must make the condition false or null. For complete syntax, see "WHILE LOOP Statement" on page 13-138.

An EXIT, EXIT WHEN, CONTINUE, or CONTINUE WHEN in the *statements* can cause the loop or the current iteration of the loop to end early.

In Example 4–28, the statements in the first WHILE LOOP statement never run, and the statements in the second WHILE LOOP statement run once.

Example 4–28 WHILE LOOP Statements

```
DECLARE
  done  BOOLEAN := FALSE;
BEGIN
  WHILE done LOOP
    DBMS_OUTPUT.PUT_LINE ('This line does not print.');
    done := TRUE;  -- This assignment is not made.
  END LOOP;

  WHILE NOT done LOOP
    DBMS_OUTPUT.PUT_LINE ('Hello, world!');
    done := TRUE;
  END LOOP;
END;
/
```

Result:

```
Hello, world!
```

Some languages have a LOOP UNTIL or REPEAT UNTIL structure, which tests a condition at the bottom of the loop instead of at the top, so that the statements run at least once. To simulate this structure in PL/SQL, use a basic LOOP statement with an EXIT WHEN statement:

```
LOOP
  statements
  EXIT WHEN condition;
END LOOP;
```

Sequential Control Statements

Unlike the IF and LOOP statements, the **sequential control statements** GOTO and NULL are not crucial to PL/SQL programming.

The GOTO statement, which goes to a specified statement, is seldom needed. Occasionally, it simplifies logic enough to warrant its use.

The NULL statement, which does nothing, can improve readability by making the meaning and action of conditional statements clear.

Topics

- GOTO Statement
- NULL Statement

GOTO Statement

The GOTO statement transfers control to a label unconditionally. The label must be unique in its scope and must precede an executable statement or a PL/SQL block. When run, the GOTO statement transfers control to the labeled statement or block. For GOTO statement restrictions, see "GOTO Statement" on page 13-88.

Use GOTO statements sparingly—overusing them results in code that is hard to understand and maintain. Do not use a GOTO statement to transfer control from a deeply nested structure to an exception handler. Instead, raise an exception. For information about the PL/SQL exception-handling mechanism, see Chapter 11, "PL/SQL Error Handling."

Example 4–29 GOTO Statement

```
DECLARE
  p  VARCHAR2(30);
  n  PLS_INTEGER := 37;
BEGIN
  FOR j in 2..ROUND(SQRT(n)) LOOP
    IF n MOD j = 0 THEN
      p := ' is 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;
/
```

Result:

```
37 is a prime number
```

A label can appear only before a block (as in Example 4–21) or before a statement (as in Example 4–29), not in a statement, as in Example 4–30.

Example 4–30 Incorrect Label Placement

```
DECLARE
  done  BOOLEAN;
BEGIN
  FOR i IN 1..50 LOOP
    IF done THEN
       GOTO end_loop;
    END IF;
    <<end_loop>>
  END LOOP;
END;
/
```

Result:

```
  END LOOP;
  *
ERROR at line 9:
ORA-06550: line 9, column 3:
PLS-00103: Encountered the symbol "END" when expecting one of the following:
( begin case declare exit for goto if loop mod null raise
return select update while with <an identifier>
<a double-quoted delimited-identifier> <a bind variable> <<
continue close current delete fetch lock insert open rollback
savepoint set sql run commit forall merge pipe purge
```

To correct Example 4–30, add a NULL statement, as in Example 4–31.

Example 4–31 GOTO Statement Goes to Labeled NULL Statement

```
DECLARE
  done  BOOLEAN;
BEGIN
  FOR i IN 1..50 LOOP
    IF done THEN
      GOTO end_loop;
    END IF;
    <<end_loop>>
    NULL;
  END LOOP;
END;
/
```

A GOTO statement can transfer control to an enclosing block from the current block, as in Example 4–32.

Example 4–32 GOTO Statement Transfers Control to Enclosing Block

```
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;
    END IF;
  END;
END;
/
```

Result:

```
Weiss
```

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

The GOTO statement in Example 4–33 transfers control into an IF statement, causing an error.

Example 4–33 GOTO Statement Cannot Transfer Control into IF Statement

```
DECLARE
  valid BOOLEAN := TRUE;
BEGIN
  GOTO update_row;

  IF valid THEN
  <<update_row>>
    NULL;
  END IF;
END;
```

```
/
```

Result:

```
  GOTO update_row;
  *
ERROR at line 4:
ORA-06550: line 4, column 3:
PLS-00375: illegal GOTO statement; this GOTO cannot transfer control to label
'UPDATE_ROW'
ORA-06550: line 6, column 12:
PL/SQL: Statement ignored
```

NULL Statement

The NULL statement only passes control to the next statement. Some languages refer to such an instruction as a no-op (no operation).

Some uses for the NULL statement are:

- To provide a target for a GOTO statement, as in Example 4–31.
- To improve readability by making the meaning and action of conditional statements clear, as in Example 4–34
- To create placeholders and stub subprograms, as in Example 4–35
- To show that you are aware of a possibility, but that no action is necessary, as in Example 4–36

In Example 4–34, the NULL statement emphasizes that only salespersons receive commissions.

Example 4–34 NULL Statement Showing 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;  -- Employee is not a sales rep
  END IF;
END;
/
```

In Example 4–35, the NULL statement lets you compile this subprogram and fill in the real body later.

Note: Using the NULL statement might raise an unreachable code warning if warnings are enabled. For information about warnings, see "Compile-Time Warnings" on page 11-2.

Example 4–35 NULL Statement as Placeholder During Subprogram Creation

```
CREATE OR REPLACE PROCEDURE award_bonus (
  emp_id NUMBER,
  bonus NUMBER
) AS
BEGIN    -- Executable part starts here
  NULL;  -- Placeholder
  -- (raises "unreachable code" if warnings enabled)
END award_bonus;
/
```

In Example 4–36, the `NULL` statement shows that you have chosen to take no action for grades other than A, B, C, D, and F.

Example 4–36 NULL Statement in ELSE Clause of Simple CASE Statement

```
CREATE OR REPLACE PROCEDURE print_grade (
  grade CHAR
) AUTHID DEFINER AS
BEGIN
  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 NULL;
  END CASE;
END;
/
```