5

PL/SQL Control Statements

PL/SQL has three categories of control statements: conditional selection statements, loop statements and sequential control statements.

PL/SQL categories of control statements are:

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

  The conditional selection statements are `IF` 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.

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 `CASE` statement, which evaluates a single expression and compares it to several potential values.
- Searched `CASE` statement, 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.

IF THEN Statement

The `IF THEN` statement either runs or skips a sequence of one or more statements, depending on a condition.

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".

> **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;
> ```

Example 5-1 IF THEN Statement

In this example, the statements between `THEN` and `END IF` run if and only if the value of `sales` is greater than `quota`+200.

```
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.
```

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.

`IF` statements can be nested, as in Example 5-3.

For complete syntax, see "IF Statement".

Example 5-2 IF THEN ELSE Statement

In this example, 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.

```
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
```

Example 5-3 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.

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;
```

For complete syntax, see "IF Statement".

Example 5-4 IF THEN ELSIF Statement

In this example, 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.

```
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.
```

Example 5-5 IF THEN ELSIF Statement Simulates Simple CASE Statement

This example 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 5-6.

```
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_1a | dangling_predicate_1a }
    [, …, { selector_value_1n | dangling_predicate_1n }] THEN statements_1
WHEN { selector_value_2a | dangling_predicate_2a }
       [, …, { selector_value_2n | dangling_predicate_2n }] THEN statements_2
...
WHEN { selector_value_na | dangling_predicate_na }
       [, …, { selector_value_nn | dangling_predicate_nn }] 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. A *dangling_predicate* can also be used either instead of or in

combination with one or multiple *selector_values*. (For complete syntax, see "CASE Statement".)

A *dangling_predicate* is an ordinary expression with its left operand missing, for example, `< 2`. Using a *dangling_predicate* allows for more complicated comparisons that would otherwise require a searched `CASE` statement.

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

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

> **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 3-51). Instead, use a searched `CASE` statement with `WHEN condition IS NULL` (see Example 3-55).

Example 5-6 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
```

Example 5-7 Simple CASE Statement with Dangling Predicates

```
DECLARE
  grade NUMBER;
BEGIN
  grade := '85';

  CASE grade
    WHEN < 0, > 100 THEN DBMS_OUTPUT.PUT_LINE('No such grade');
    WHEN > 89 THEN DBMS_OUTPUT.PUT_LINE('A');
    WHEN > 79 THEN DBMS_OUTPUT.PUT_LINE('B');
    WHEN > 69 THEN DBMS_OUTPUT.PUT_LINE('C');
```

```
    WHEN > 59 THEN DBMS_OUTPUT.PUT_LINE('D');
    ELSE DBMS_OUTPUT.PUT_LINE('F');
  END CASE;
END;
/
```

Result:

```
B
```

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".)

The searched `CASE` statement in Example 5-8 is logically equivalent to the simple `CASE` statement in Example 5-6.

In both Example 5-8 and Example 5-6, the `ELSE` clause can be replaced by an `EXCEPTION` part. Example 5-9 is logically equivalent to Example 5-8.

Example 5-8 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
```

Example 5-9 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 iteratively with a series of different values.

A `LOOP` statement has three parts:

1. An iterand, also known as a loop variable, to pass values from the loop header to the loop body
2. Iteration controls to generate values for the loop
3. A loop body run once for each value

```
loop_statement ::= [ iteration_scheme ] LOOP
            loop_body
END LOOP [ label ];

iteration_scheme ::= WHILE expression
                   | FOR iterator
```

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` can appear anywhere inside a loop, but not outside a loop. Oracle recommends using these statements instead of the `GOTO` statement, 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.

`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).

> **See Also:**
>
> - GOTO Statement
> - CONTINUE Statement
> - "EXIT Statement"
> - "Overview of Exception Handling" for information about exceptions
> - "Processing Query Result Sets With Cursor FOR LOOP Statements" for information about the cursor `FOR LOOP`

Basic LOOP Statement

The basic `LOOP` statement has this structure.

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.

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

> **See Also:**
>
> "Basic LOOP Statement"

FOR LOOP Statement Overview

The `FOR LOOP` statement runs one or more statements for each value of the loop index.

A `FOR LOOP` header specifies the iterator. The iterator specifies an iterand and the iteration controls. The iteration control provides a sequence of values to the iterand for access in the loop body. The loop body has the statements that are processed once for each value of the iterand.

The iteration controls available are :

Stepped Range An iteration control that generates a sequence of stepped numeric values. When step is not specified, the counting control is a stepped range of type pls integer with a step of one.

Single Expression An iteration control that evaluates a single expression.

Repeated Expression An iteration control that repeatedly evaluates a single expression.

Values Of An iteration control that generates all the values from a collection in sequence. The collection can be a vector valued expression, cursor, cursor variable, or dynamic SQL.

Indices Of An iteration control that generates all the indices from a collection in sequence. While all the collection types listed for values of are allowed, indices of is most useful when the collection is a vector variable.

Pairs Of An iteration control that generates all the index and value pairs from a collection. All of the collection types allowed for values of are allowed for pairs of. Pairs of iteration controls require two iterands.

Cursor An iteration control that generates all the records from a cursor, cursor variable, or dynamic SQL.

The `FOR LOOP` statement has this structure:

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

for_loop_header ::= FOR iterator LOOP

iterator ::= iterand_decl [, iterand_decl] IN iteration_ctl_seq

iterand_decl  ::= pls_identifier [ MUTABLE | IMMUTABLE ] [ constrained_type ]

iteration_ctl_seq ::= qual_iteration_ctl [,]...

qual_iteration_ctl ::= [ REVERSE ] iteration_control  pred_clause_seq

iteration_control ::= stepped_control
                    | single_expression_control
                    | values_of_control
                    | indices_of_control
                    | pairs_of_control
                    | cursor_control

pred_clause_seq ::= [ stopping_pred ] [ skipping_pred ]

stopping_pred ::= WHILE boolean_expression

skipping_pred ::= WHEN boolean_expression

stepped_control ::= lower_bound .. upper_bound [ BY step ]

single_expression_control ::= [ REPEAT ] expr
```

> **See Also:**
>
> "FOR LOOP Statement" for more information about syntax and semantics

FOR LOOP Iterand

The index or iterand of a `FOR LOOP` statement is implicitly or explicitly declared as a variable that is local to the loop.

The statements in the loop can read the value of the iterand, but cannot change it. Statements outside the loop cannot reference the iterand. After the `FOR LOOP` statement runs, the iterand is undefined. A loop iterand is sometimes called a loop counter.

Example 5-10 FOR LOOP Statement Tries to Change Index Value

In this example, the `FOR LOOP` statement tries to change the value of its index, causing an error.

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

Example 5-11 Outside Statement References FOR LOOP Statement Index

In this example, a statement outside the `FOR LOOP` statement references the loop index, causing an error.

```
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));
                                                         *
PLS-00201: identifier 'I' must be declared
ORA-06550: line 6, column 3:
PL/SQL: Statement ignored
```

Example 5-12 FOR LOOP Statement Index with Same Name as Variable

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 this example shows.

```
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 5-13 FOR LOOP Statement References Variable with Same Name as Index

This example shows how to change Example 5-12 to allow the statement inside the loop to reference the variable declared in the enclosing block.

```
<<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
```

Example 5-14 Nested FOR LOOP Statements with Same Index Name

In this example, 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.

```
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
```

Iterand Mutability

The mutability property of an iterand determines whether or not it can be assigned in the loop body.

If all iteration controls specified in an iterator are cursor controls, the iterand is mutable by default. Otherwise, the iterand is immutable. The default mutability property of an iterand can be changed in the iterand declaration by specifying the `MUTABLE` or `IMMUTABLE` keyword after the iterand variable.

Considerations when declaring an iterand mutable:

- Any modification to the iterand for values of iteration control or the values iterand for a pairs of iteration control will not affect the sequence of values produced by that iteration control.
- Any modification to the iterand for stepped range iteration control or repeated single expression iteration control will likely affect the behaviour of that control and the sequence of values it produces.
- When the PL/SQL compiler can determine that making an iterand mutable may adversely affect runtime performance, it may report a warning.

Multiple Iteration Controls

Multiple iteration controls may be chained together by separating them with commas.

Each iteration control has a set of controlling expressions (some controls have none) that are evaluated once when the control starts. Evaluation of these expressions or conversion of the evaluated values to the iterand type may raise exceptions. In such cases, the loop is abandoned and normal exception handling occurs. The iterand is accessible in the list of iteration controls. It is initially set to the default value for its type. If that type has a not null constraint, any reference to the iterand in the controlling expressions for the first iteration control will produce a semantic error because the iterand cannot be implicitly initialized. When an iteration control is exhausted, the iterand contains the final value assigned to it while processing that iteration control and execution advances to the next iteration control. If no values are assigned to the iterand by an iteration control, it retains the value it had prior to the start of that iteration control. If the final value of a mutable iterand is modified in the loop body, that modified value will be visible when evaluating the control expressions from the following iteration control.

Expanding Multiple Iteration Controls Into PL/SQL

The first iteration control is initialized. The loop for the first iteration control is evaluated. The controlling expressions from the next iteration control is evaluated. The loop for the second iteration control is evaluated. Each iteration control and loop is evaluated in turn until there are no more iteration controls.

Example 5-15 Using Multiple Iteration Controls

This example shows the loop variable *i* taking the value three iteration controls in succession. The value of the iterator is printed for demonstration purpose. It shows that when a loop control is exhausted, the next iteration control begins. When the last iteration control is exhausted, the loop is complete.

```
DECLARE
   i PLS_INTEGER;
BEGIN
   FOR i IN 1..3, REVERSE i+1..i+10, 51..55 LOOP
      DBMS_OUTPUT.PUT_LINE(i);
   END LOOP;
END;
/


1
2
3
13
12
11
10
9
8
7
6
5
4
51
52
53
54
55
```

Stepped Range Iteration Controls

Stepped range iteration controls generate a sequence of numeric values.

Controlling expressions are the lower bound, upper bound, and step.

```
stepped_control ::= [ REVERSE ] lower_bound..upper_bound [ BY step ]
lower_bound ::= numeric_expression
upper_bound ::= numeric_expression
step ::= numeric_expression
```

Expanding Stepped Range Iteration Controls Into PL/SQL

When the iteration control is initialized, each controlling expression is evaluated and converted to the type of the iterand. `Step` must have a strictly positive numeric value. If any exception occurs while evaluating the controlling expressions, the loop is abandoned and normal exception handling occurs. When no step is specified, its value is one. The values generated by a stepped range iteration control go from lower bound to upper bound by step. When `REVERSE` is specified the values are decremented from the upper bound to lower bound by step. If the iterand has a floating point type, some combinations of loop control values may create an infinite loop because of rounding errors. No semantic or dynamic analysis will report this. When the iterand is mutable and is modified in the loop body, the modified value is used for the increment and loop exhaustion test in the next iterand update. This may change the sequence of values processed by the loop.

Example 5-16 FOR LOOP Statements Range Iteration Control

In this example, the `iterand i` has a *`lower_bound`* of 1 and an `upper_bound` of 3. The loop prints the numbers from 1 to 3.

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

Result:

```
1
2
3
```

Example 5-17 Reverse FOR LOOP Statements Range Iteration Control

The `FOR LOOP` statement in this example prints the numbers from 3 to 1. The loop variable *i* is implicitly declared as a `PLS_INTEGER` (the default for counting and indexing loops).

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

Result:

```
3
2
1
```

Example 5-18 Stepped Range Iteration Controls

This example shows a loop variable *n* declared explicitly as a NUMBER(5,1). The increment for the counter is 0.5.

```
BEGIN
   FOR n NUMBER(5,1) IN 1.0 .. 3.0 BY 0.5 LOOP
      DBMS_OUTPUT.PUT_LINE(n);
   END LOOP;
END;
/
```

Result:

```
1
1.5
2
2.5
3
```

Example 5-19 STEP Clause in FOR LOOP Statement

In this example, the `FOR LOOP` effectively increments the index by five.

```
BEGIN
  FOR i IN 5..15 BY 5 LOOP
    DBMS_OUTPUT.PUT_LINE (i);
  END LOOP;
END;
```

Result:

```
5
10
15
```

Example 5-20 Simple Step Filter Using FOR LOOP Stepped Range Iterator

This example illustrates a simple step filter. This filter is used in signal processing and other reduction applications. The predicate specifies that every Kth element of the original collection is passed to the collection being created.

```
FOR i IN start..finish LOOP
   IF (i - start) MOD k = 0 THEN
      newcol(i) := col(i)
   END IF;
END LOOP;
```

You can implement the step filter using a stepped range iterator.

```
FOR i IN start..finish BY k LOOP
   newcol(i) := col(i)
END LOOP;
```

You can implement the same filter by creating a new collection using a stepped iteration control embedded in a qualified expression.

```
newcol := col_t(FOR I IN start..finish BY k => col(i));
```

Single Expression Iteration Controls

A single expression iteration control generates a single value.

single_expression_control ::= [REPEAT] *expr*

A single expression iteration control has no controlling expressions.

When the iterand is mutable, changes made to it in the loop body will be seen when reevaluating the expression in the repeat form.

Expanding Single Expression Iteration Controls Into PL/SQL

The expression is evaluated, converted to the iterand type to create the next value. Any stopping predicate is evaluated. If it fails to evaluate to `TRUE`, the iteration control is exhausted. Any skipping predicate is evaluated. If it fails to evaluate to `TRUE`, skip the next step. Evaluate the loop body. If `REPEAT` is specified, evaluate the expression again. Otherwise, the iteration control is exhausted.

Example 5-21 Single Expression Iteration Control

This example shows the loop body being processed once.

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

Result:

```
1
```

This example shows the iterand starting with 1, then *i*2* is evaluated repeatedly until the stopping predicate evaluates to true.

```
BEGIN
   FOR i IN 1, REPEAT i*2 WHILE i < 100 LOOP
      DBMS_OUTPUT.PUT_LINE(i);
   END LOOP;
END;
/
```

Result:

```
1
2
4
8
16
32
64
```

Collection Iteration Controls

VALUES OF, INDICES OF, and PAIRS OF iteration controls generate sequences of values for an iterand derived from a collection.

```
collection_iteration_control ::= values_of_control
                                 | indices_of_control
                                 | pairs_of_control

values_of_control ::= VALUES OF expr
                      | VALUES OF (cursor_object)
                      | VALUES OF (sql_statement)
                      | VALUES OF cursor_variable
                      | VALUES OF (dynamic_sql)

indices_of_control ::= INDICES OF expr
                       | INDICES OF (cursor_object)
                       | INDICES OF (sql_statement)
                       | INDICES OF cursor_variable
                       | INDICES OF (dynamic_sql)

pairs_of_control ::= PAIRS OF expr
                     | PAIRS OF (cursor_object)
                     | PAIRS OF (sql_statement)
                     | PAIRS OF cursor_variable
                     | PAIRS OF (dynamic_sql)
```

The collection itself is the controlling expression. The collection can be a vector value expression, a cursor object, cursor variable, or dynamic SQL. If a collection is null, it is treated as if it were defined and empty.

A *cursor_object* is an explicit PL/SQL cursor object. A *sql_statement* is an implicit PL/SQL cursor object created for a SQL statement specified directly in the iteration control. A *cursor_variable* is a PL/SQL `REF CURSOR` object.

When the iterand for a values of iteration control or the value iterand for a `VALUES OF` iteration control is modified in the loop body, those changes have no effect on the next value generated by the iteration control.

If the collection is modified in the loop body, behavior is unspecified. If a cursor variable is accessed other than through the iterand during execution of the loop body, the behavior is unspecified. Most `INDICES OF` iteration controls produce a numeric sequence unless the collection is a vector variable.

Expanding VALUES OF Iteration Controls into PL/SQL

The collection is evaluated and assigned to a vector. If the collection is empty, the iteration control is exhausted. A temporary hidden index is initialized with the index of the first element (or last element if `REVERSE` is specified). A value is fetched from the collection based on the temporary index to create the next value for the iterand. Any stopping predicate is evaluated. If it fails to evaluate to `TRUE`, the iteration control is exhausted. Any skipping predicate is evaluated. If it fails to evaluate to `TRUE`, skip the next step. Evaluate the loop body. Advance the index temporary to the index of the next element in the vector (previous element for REVERSE). Determine the next value and reiterate with each iterand value until the iteration control is exhausted.

Example 5-22 `VALUES OF` Iteration Control

This example prints the values from the collection vec: [11, 10, 34]. The iterand values of the iteration control variable *i* is the value of the first element in the vector, then the next element, and the last one.

```
DECLARE
   TYPE intvec_t IS TABLE OF PLS_INTEGER INDEX BY PLS_INTEGER;
   vec intvec_t := intvec_t(3 => 10, 1 => 11, 100 => 34);
BEGIN
   FOR i IN VALUES OF vec LOOP
      DBMS_OUTPUT.PUT_LINE(i);
   END LOOP;
END;
/
```

Result:

```
11 10 34
```

Expanding INDICES OF Iteration Controls into PL/SQL

The collection is evaluated and assigned to a vector. If the collection is empty, the iteration control is exhausted. The next value for the iterand is determined (index of the first element or last element if `REVERSE` is specified). The next value is assigned to the iterand. Any stopping predicate is evaluated. If it fails to evaluate to `TRUE`, the iteration control is exhausted. Any skipping predicate is evaluated. If it fails to evaluate to `TRUE`, skip the next step. The loop body

is evaluated. Advance the iterand to the next value which is the index of the next element in the vector (previous element for `REVERSE`). Reiterate with each iterand value (assigned the index of the next or previous element) until the iteration control is exhausted.

Example 5-23 `INDICES OF` Iteration Control

This example prints the indices of the collection vec : [1, 3, 100]. The iterand values of the iteration control variable *i* is the index of the first element in the vector, then the next element, and the last one.

```
DECLARE
   TYPE intvec_t IS TABLE OF PLS_INTEGER INDEX BY PLS_INTEGER;
   vec intvec_t := intvec_t(3 => 10, 1 => 11, 100 => 34);
BEGIN
   FOR i IN INDICES OF vec LOOP
       DBMS_OUTPUT.PUT_LINE(i);
   END LOOP;
END;
/
```

Result:

```
1 3 100
```

Expanding PAIRS OF Iteration Controls into PL/SQL

The collection is evaluated and assigned to a vector. If the collection is empty, the iteration control is exhausted. The next index value for the iterand is determined (index of the first element or last element if `REVERSE` is specified). The next value of the element indexed by the next value is assigned to the iterand. Any stopping predicate is evaluated. If it fails to evaluate to `TRUE`, the iteration control is exhausted. Any skipping predicate is evaluated. If it fails to evaluate to `TRUE`, skip the next step. The loop body is evaluated. Advance the iterand to the next index value which is the index of the next element in the vector (previous element for `REVERSE`). Reiterate with each iterand value until the iteration control is exhausted.

Example 5-24 `PAIRS OF` Iteration Control

This example inverts a collection vec into a collection result and prints the resulting index value pairs (10 => 3, 11 => 1, 34 => 100).

```
DECLARE
   TYPE intvec_t IS TABLE OF PLS_INTEGER INDEX BY PLS_INTEGER;
   vec  intvec_t := intvec_t(3 => 10, 1 => 11, 100 => 34);
   result intvec_t;
BEGIN
   result := intvec_t(FOR i,j IN PAIRS OF vec INDEX j => i);
   FOR i,j IN PAIRS OF result LOOP
      DBMS_OUTPUT.PUT_LINE(i || '=>'|| j);
   END LOOP;
END;
/
```

Result:

```
10=>3 11=>1 34=>100
```

Cursor Iteration Controls

Cursor iteration controls generate the sequence of records returned by an explicit or implicit cursor.

The cursor definition is the controlling expression. You cannot use `REVERSE` with a cursor iteration control.

```
cursor_iteration__control ::=  { cursor _object
                    | sql_statement
                    | cursor_variable
                    | dynamic_sql }
```

A *cursor_object* is an explicit PL/SQL cursor object. A *sql_statement* is an implicit PL/SQL cursor object created for a SQL statement specified directly in the iteration control. A *cursor_variable* is a PL/SQL `REF CURSOR` object. A cursor iteration control is equivalent to a `VALUES OF` iteration control whose collection is a cursor. When the iterand is modified in the loop body, it has no effect on the next value generated by the iteration control. When the collection is a cursor variable, it must be open when the iteration control is encountered or an exception will be raised. It remains open when the iteration control is exhausted. If the cursor variable is accessed other than through the iterand during execution of the loop body, the behavior is unspecified.

Expanding Cursor Iteration Controls Into PL/SQL

The cursor is evaluated to create a vector of iterands. If the vector is empty, the iteration control is exhausted. A value is fetched in the vector to create the next value for the iterand. Any stopping predicate is evaluated. If it fails to evaluate to `TRUE`, the iteration control is exhausted. Any skipping predicate is evaluated. If it fails to evaluate to `TRUE`, skip the next step. Evaluate the loop body. Reiterate the same with each iterand value fetched until the iteration control is exhausted.

Example 5-25 Cursor Iteration Controls

This example creates an associative array mapping of id to data from table t.

```
OPEN c FOR SELECT id, data FROM T;
FOR r rec_t IN c LOOP
   result(r.id) := r.data;
END LOOP;
CLOSE c;
```

Using Dynamic SQL in Iteration Controls

```
...
dynamic_sql ::= EXECUTE IMMEDIATE dynamic_sql_stmt [ using_clause ]

using_clause ::= USING [ [ IN ] (bind_argument [,])+ ]
```

Dynamic SQL may be used in a cursor or collection iteration control. Such a construct cannot provide a default type; if it is used as the first iteration control, an explicit type must be specified for the iterand (or for the value iterand for a pairs of control). The *using_clause* is the only clause allowed. No `INTO` or dynamic returning clauses may be used. If the specified SQL statement is a kind that cannot return any rows, a runtime error will be reported similar to that

reported if a bulk collect into or into clause were specified on an ordinary `EXECUTE IMMEDIATE` statement.

Example 5-26 Using Dynamic SQL As An Iteration Control

This example shows the iteration control generates all the records from a dynamic SQL. It prints the last_name and employee_id of all employees having an employee_id less than 103. It executes the loop body when the stopping predicate is `TRUE`.

```
DECLARE
   cursor_str VARCHAR2(500) := 'SELECT last_name, employee_id FROM hr.employees ORDER BY
last_name';
   TYPE rec_t IS RECORD (last_name VARCHAR2(25),
                         employee_id NUMBER);
BEGIN
   FOR r rec_t IN VALUES OF (EXECUTE IMMEDIATE cursor_str) WHEN r.employee_id < 103 LOOP
      DBMS_OUTPUT.PUT_LINE(r.last_name || ', ' || r.employee_id);
   END LOOP;
END;
/
```

Result:

```
Garcia, 102
King, 100
Yang, 101
```

Example 5-27 Using Dynamic SQL As An Iteration Control In a Qualified Expression

```
v := vec_rec_t( FOR r rec_t IN (EXECUTE IMMEDIATE query_var) SEQUENCE => r);
```

Stopping and Skipping Predicate Clauses

A stopping predicate clause can cause the iteration control to be exhausted while a skipping predicate clause can cause the loop body to be skipped for some values.

The expressions in these predicate clauses are not controlling expressions.

A stopping predicate clause can cause the iteration control to be exhausted. The *boolean_expression* is evaluated at the beginning of each iteration of the loop. If it fails to evaluate to `TRUE`, the iteration control is exhausted.

A skipping predicate clause can cause the loop body to be skipped for some values. The *boolean_expression* is evaluated. If it fails to evaluate to `TRUE`, the iteration control skips to the next value.

pred_clause_seq ::= [*stopping_pred*] [*skipping_pred*]

stopping_pred ::= WHILE *boolean_expression*

skipping_pred ::= WHEN *boolean_expression*

Example 5-28 Using FOR LOOP Stopping Predicate Clause

This example shows an iteration control with a WHILE stopping predicate clause The iteration control is exhausted if the stopping predicate does not evaluate to `TRUE`.

```
BEGIN
   FOR power IN 1, REPEAT power*2 WHILE power <= 64 LOOP
      DBMS_OUTPUT.PUT_LINE(power);
   END LOOP;
END;
/
```

Result:

```
1
2
4
8
16
32
64
```

Example 5-29 Using FOR LOOP Skipping Predicate Clause

This example shows an iteration control with a WHEN skipping predicate clause. If the skipping predicate does not evaluate to `TRUE`, the iteration control skips to the next value.

```
BEGIN
   FOR power IN 2, REPEAT power*2 WHILE power <= 64 WHEN MOD(power, 32)= 0 LOOP
      DBMS_OUTPUT.PUT_LINE(power);
   END LOOP;
END;
/
```

Result:

```
2
32
64
```

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".

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.

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

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".

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 PL/SQL Error Handling.

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

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
- To improve readability by making the meaning and action of conditional statements clear
- To create placeholders and stub subprograms
- To show that you are aware of a possibility, but that no action is necessary

> **Note:**
>
> Using the `NULL` statement might raise an `unreachable code` warning if warnings are enabled. For information about warnings, see "Compile-Time Warnings".

Example 5-30 NULL Statement Showing No Action

The NULL statement emphasizes that only salespersons receive commissions.

```
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;
/
```

Example 5-31 NULL Statement as Placeholder During Subprogram Creation

The NULL statement lets you compile this subprogram and fill in the real body later.

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

Example 5-32 NULL Statement in ELSE Clause of Simple CASE Statement

The NULL statement shows that you have chosen to take no action for grades other than A, B, C, D, and F.

```
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;
/
BEGIN
  print_grade('A');
  print_grade('S');
END;
/
```

Result:

```
Excellent
```