Manipulating Data in PL/SQL

What Will I Learn?

In this lesson, you will learn to:

- Construct and execute PL/SQL statements that manipulate data with DML statements
- Describe when to use implicit or explicit cursors in PL/SQL
- Create PL/SQL code to use SQL implicit cursor attributes to evaluate cursor activity

Why Learn It?

In the previous lesson, you learned that you can include SELECT statements that return a single row in a PL/SQL block. The data retrieved by the SELECT statement must be held in variables using the `INTO` clause.

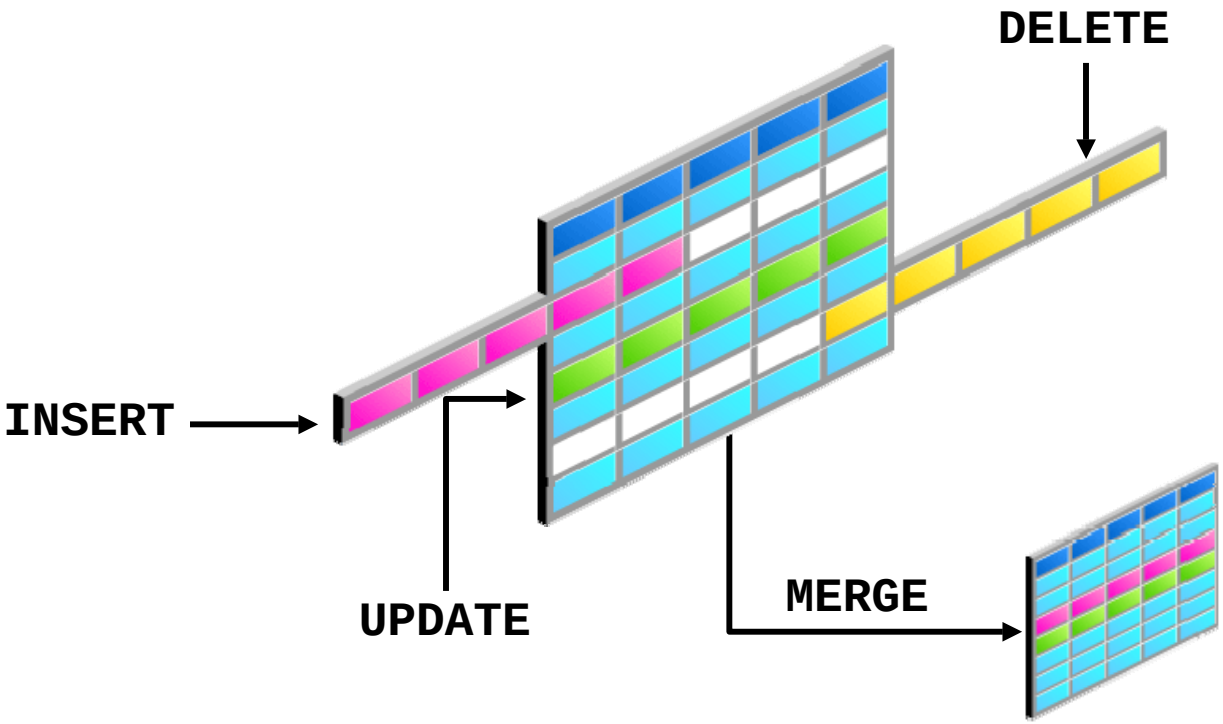
In this lesson, you learn how to include data manipulation language (DML) statements, such as `INSERT`, `UPDATE`, `DELETE`, and `MERGE` in PL/SQL blocks.

Tell Me/Show Me

Manipulating Data Using PL/SQL

Make changes to data by using DML commands within your PLSQL block:

- INSERT
- UPDATE
- DELETE
- MERGE



Tell Me/Show Me

Manipulating Data Using PL/SQL (continued)

- You manipulate data in the database by using the DML commands.
- You can issue the DML commands—`INSERT`, `UPDATE`, `DELETE`, and `MERGE`—without restriction in PL/SQL. Row locks (and table locks) are released by including `COMMIT` or `ROLLBACK` statements in the PL/SQL code.
  - The `INSERT` statement adds new rows to the table.
  - The `UPDATE` statement modifies existing rows in the table.
  - The `DELETE` statement removes rows from the table.
  - The `MERGE` statement selects rows from one table to update and/or insert into another table. The decision whether to update or insert into the target table is based on a condition in the `ON` clause.
- **Note:** `MERGE` is a deterministic statement—that is, you cannot update the same row of the target table multiple times in the same `MERGE` statement. You must have `INSERT` and `UPDATE` object privileges in the target table and the `SELECT` privilege in the source table.

Tell Me/Show Me

Inserting Data

The INSERT statement adds new row(s) to a table.

Example: Add new employee information to the COPY_EMP table.

```
BEGIN
  INSERT INTO copy_emp
     (employee_id, first_name, last_name, email,
      hire_date, job_id, salary)
   VALUES (99, 'Ruth', 'Cores',
           'RCORES', SYSDATE, 'AD_ASST', 4000);
END;
```

One new row is added to the COPY_EMP table.

Tell Me/Show Me

Updating Data

The UPDATE statement modifies existing row(s) in a table.

Example: Increase the salary of all employees who are stock clerks.

```
DECLARE
  v_sal_increase   employees.salary%TYPE := 800;
BEGIN
  UPDATE      copy_emp
    SET       salary = salary + v_sal_increase
    WHERE     job_id = 'ST_CLERK';
END;
```

Tell Me/Show Me

Deleting Data

The DELETE statement removes row(s) from a table.

Example: Delete rows that belong to department 10 from the COPY_EMP table.

```
DECLARE
  v_deptno   employees.department_id%TYPE := 10;
BEGIN
  DELETE FROM   copy_emp
    WHERE  department_id = v_deptno;
END;
```

Tell Me/Show Me

Merging Rows

The `MERGE` statement selects rows from one table to update and/or insert into another table. Insert or update rows in the `copy_emp` table to match the `employees` table.

```
BEGIN
  MERGE INTO copy_emp c
     USING employees e
     ON (e.employee_id = c.employee_id)
   WHEN MATCHED THEN
     UPDATE SET
       c.first_name     = e.first_name,
       c.last_name      = e.last_name,
       c.email          = e.email,
       . . .
   WHEN NOT MATCHED THEN
     INSERT VALUES(e.employee_id, e.first_name, …e.department_id);
END;
```

Tell Me/Show Me

Getting Information From a Cursor

Look again at the DELETE statement in this PL/SQL block.

```
DECLARE
  v_deptno   employees.department_id%TYPE := 10;
BEGIN
  DELETE FROM   copy_emp
    WHERE  department_id = v_deptno;
END;
```

It would be useful to know how many COPY_EMP rows were deleted by this statement.

To obtain this information, we need to understand cursors.

Tell Me/Show Me

What is a Cursor?

Every time an SQL statement is about to be executed, the Oracle server allocates a private memory area to store the SQL statement and the data that it uses. This memory area is called an implicit cursor.

Because this memory area is automatically managed by the Oracle server, you have no direct control over it. However, you can use predefined PL/SQL variables, called implicit cursor attributes, to find out how many rows were processed by the SQL statement.

Tell Me/Show Me

Implicit and Explicit Cursors

There are two types of cursors:

- Implicit cursors: Defined automatically by Oracle for all SQL data manipulation statements, and for queries that return only one row. An implicit cursor is always automatically named “SQL.”
- Explicit cursors: Defined by the PL/SQL programmer for queries that return more than one row. (Covered in a later lesson.)

Tell Me/Show Me

Cursor Attributes for Implicit Cursors

Cursor attributes are automatically declared variables that allow you to evaluate what happened when a cursor was last used. Attributes for implicit cursors are prefaced with "SQL." Use these attributes in PL/SQL statements, but not in SQL statements. Using cursor attributes, you can test the outcome of your SQL statements.

`SQL%FOUND`	Boolean attribute that evaluates to `TRUE` if the most recent SQL statement returned at least one row
`SQL%NOTFOUND`	Boolean attribute that evaluates to `TRUE` if the most recent SQL statement did not return even one row
`SQL%ROWCOUNT`	An integer value that represents the number of rows affected by the most recent SQL statement

Tell Me/Show Me

Using Implicit Cursor Attributes: Example 1

Delete rows that have the specified employee ID from the `copy_emp` table. Print the number of rows deleted.

```
DECLARE
  v_deptno copy_emp.department_id%TYPE := 50;
BEGIN
  DELETE FROM copy_emp
    WHERE department_id = v_deptno;
  DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT ||
                       ' rows deleted.');
END;
```

Tell Me/Show Me

Using Implicit Cursor Attributes: Example 2

Update several rows in the `COPY_EMP` table. Print the number of rows updated.

```
DECLARE
  v_sal_increase   employees.salary%TYPE := 800;
BEGIN
  UPDATE      copy_emp
    SET       salary = salary + v_sal_increase
    WHERE     job_id = 'ST_CLERK';
  DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT ||
                       ' rows updated.');
END;
```

Tell Me/Show Me

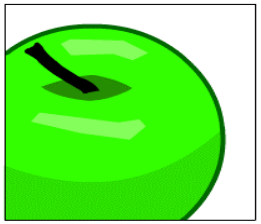
Using Implicit Cursor Attributes: Good Practice Guideline

Look at this code, which creates a table and then executes a PL/SQL block. What value is inserted into RESULTS?

```
CREATE TABLE results (num_rows NUMBER(4));

BEGIN
  UPDATE      copy_emp
    SET       salary = salary + 100
    WHERE     job_id = 'ST_CLERK';
  INSERT INTO results (num_rows)
    VALUES (SQL%ROWCOUNT);
END;
```

Tell Me/Show Me



Terminology

Key terms used in this lesson include:

INSERT

UPDATE

DELETE

MERGE

Implicit cursors

Explicit cursors

Summary

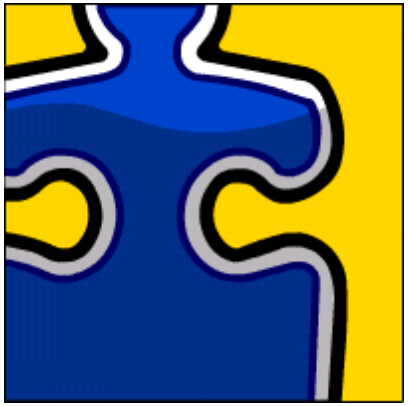
In this lesson, you learned to:

- Construct and execute PL/SQL statements that manipulate data with DML statements
- Describe when to use implicit or explicit cursors in PL/SQL
- Create PL/SQL code to use SQL implicit cursor attributes to evaluate cursor activity

Try It/Solve It

The exercises in this lesson cover the following topics:

- Executing PL/SQL statements that manipulate data with DML statements
- Describing when to use implicit or explicit cursors in PL/SQL
- Using SQL implicit cursor attributes in PL/SQL



Conditional Control: `IF` Statements

What Will I Learn?

In this lesson, you will learn to:

- Describe a use for conditional control structures
- List the types of conditional control structures
- Construct and use an `IF` statement
- Construct and use an `IF-THEN-ELSE` statement
- Create a PL/SQL to handle the null condition in `IF` statements

Why Learn It?

In this section, you learn how to use the conditional logic in a PL/SQL block. Conditional processing extends the usefulness of programs by allowing the use of simple logical tests to determine which statements are executed.

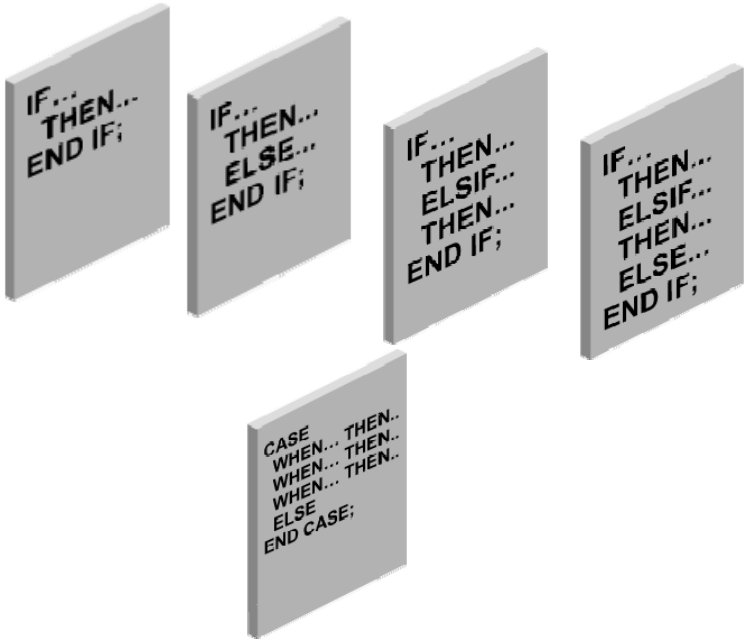
Tell Me/Show Me

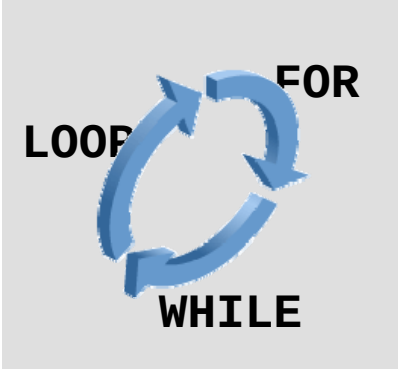
Controlling the Flow of Execution

You can change the logical flow of statements within the PL/SQL block with a number of control structures.

This lesson introduces three types of PL/SQL control structures:

- Conditional constructs with the `IF` statement
- `CASE` expressions
- `LOOP` control structures





The `IF` statement is discussed in detail in this lesson. Later lessons discuss `CASE` and `LOOP` in detail.

Tell Me/Show Me

IF statement

The IF statement contains alternative courses of action in a block based on conditions. A condition is an expression with a TRUE or FALSE value that is used to make a decision.

Consider the following example:

```
if the region_id is in (5, 13, 21)
    then print "AMERICAS"
otherwise, if the region_id is in (11, 14, 15)
    then print “AFRICA"
otherwise, if the region_id is in (30, 34, 35)
    then print "ASIA"
```

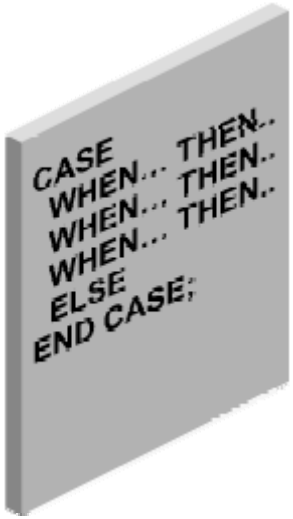
Conditions

Tell Me/Show Me

CASE Expressions

CASE expressions are similar to IF statements in that they also determine a course of action based on conditions. They are different in that they can be used outside of a PLSQL block in an SQL statement. Consider the following example:

```
If the region_id is
     5 then print  "AMERICAS"
     13 then print "AMERICAS"
     21 then print "AMERICAS"
     11 then print "AFRICA"
     14 then print "AFRICA"
     15 then print "AFRICA"
     30 then print "ASIA" …
```

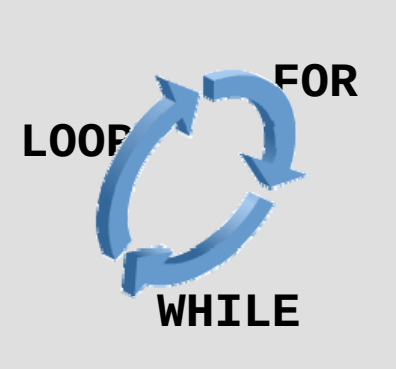


CASE expressions are discussed in the next lesson.

Tell Me/Show Me

LOOP control structures

Loop control structures are repetition statements that enable you to execute statements in a PLSQL block repeatedly. There are three types of loop control structures supported by PL/SQL, **BASIC**, **FOR**, and **WHILE**. Consider the following example:



Print a list of numbers 1–5 by using a loop and a counter.

```
Loop Counter equals: 1
Loop Counter equals: 2
Loop Counter equals: 3
Loop Counter equals: 4
Loop Counter equals: 5

Statement processed.
```

Loops are discussed in later lessons.

Tell Me/Show Me

IF Statements

The structure of the PL/SQL IF statement is similar to the structure of IF statements in other procedural languages. It enables PL/SQL to perform actions selectively based on conditions.

Syntax:

```
IF condition THEN
  statements;
[ELSIF condition THEN
  statements;]
[ELSE
  statements;]
END IF;
```

Tell Me/Show Me

`IF` Statements (continued)

- *condition* is a Boolean variable or expression that returns TRUE, FALSE, or NULL.
- THEN introduces a clause that associates the Boolean expression with the sequence of statements that follows it.
- *statements* can be one or more PL/SQL or SQL statements. (They can include further IF statements containing several nested IF, ELSE, and ELSIF statements.) The statements in the THEN clause are executed only if the condition in the associated IF clause evaluates to TRUE.

```
IF condition THEN
  statements;
[ELSIF condition THEN
  statements;]
[ELSE
  statements;]
END IF;
```

Tell Me/Show Me

IF Statements (continued)

- ELSIF is a keyword that introduces a Boolean expression. (If the first condition yields FALSE or NULL, then the ELSIF keyword introduces additional conditions.)
- ELSE introduces the default clause that is executed if and only if none of the earlier conditions (introduced by IF and ELSIF) are TRUE. The tests are executed in sequence so that a later condition that might be true is pre-empted by an earlier condition that is true.
- END IF; marks the end of an IF statement.

```
IF condition THEN
  statements;
[ELSIF condition THEN
  statements;]
[ELSE
  statements;]
END IF;
```

Tell Me/Show Me

IF Statements (continued)

Note: ELSIF and ELSE are optional in an IF statement. You can have any number of ELSIF keywords but only one ELSE keyword in your IF statement. END IF marks the end of an IF statement and must be terminated by a semicolon.

```
IF condition THEN
  statements;
[ELSIF condition THEN
  statements;]
[ELSE
  statements;]
END IF;
```

Tell Me/Show Me

Simple IF Statement

The following is an example of a simple IF statement with a THEN clause. The v_myage variable is initialized to 31. The condition for the IF statement returns FALSE because v_myage is not less than 11. Therefore, the control never reaches the THEN clause and nothing is printed to the screen.

```
DECLARE
  v_myage NUMBER:=31;
BEGIN
  IF v_myage < 11
  THEN
    DBMS_OUTPUT.PUT_LINE(' I am a child ');
  END IF;
END;
```

Tell Me/Show Me

IF THEN ELSE Statement

The ELSE clause is added to the code in the previous slide. The condition has not changed and, therefore, it still evaluates to FALSE. Remember that the statements in the THEN clause are only executed if the condition returns TRUE. In this case, the condition returns FALSE and, therefore, the control moves to the ELSE statement.

```
DECLARE
  v_myage NUMBER:=31;
BEGIN
  IF v_myage < 11
  THEN
    DBMS_OUTPUT.PUT_LINE(' I am a child ');
  ELSE
    DBMS_OUTPUT.PUT_LINE(' I am not a child ');
  END IF;
END;
```

Tell Me/Show Me

IF ELSIF ELSE Clause

The IF statement now contains multiple ELSIF clauses and an ELSE clause. Notice that the ELSIF clauses add additional conditions. As with the IF, each ELSIF condition is followed by a THEN clause, which is executed if the condition returns TRUE.

```
DECLARE
  v_myage NUMBER:=31;
BEGIN
  IF v_myage < 11
    THEN
      DBMS_OUTPUT.PUT_LINE('I am a child');
  ELSIF v_myage < 20
    THEN
      DBMS_OUTPUT.PUT_LINE('I am young');
  ELSIF v_myage < 30
    THEN
      DBMS_OUTPUT.PUT_LINE('I am in my twenties');
  ELSIF v_myage < 40
    THEN
      DBMS_OUTPUT.PUT_LINE('I am in my thirties');
  ELSE
    DBMS_OUTPUT.PUT_LINE('I am always young ');
  END IF;
END;
```

Tell Me/Show Me

IF ELSIF ELSE Clause (continued)

When you have multiple clauses in the IF statement and a condition is FALSE or NULL, control then shifts to the next clause. Conditions are evaluated one by one from the top. IF all conditions are FALSE or NULL, then the statements in the ELSE clause are executed. The final ELSE clause is optional.

```
…IF    v_myage < 11 THEN
     DBMS_OUTPUT.PUT_LINE(' I am a child ');
 ELSIF v_myage < 20 THEN
     DBMS_OUTPUT.PUT_LINE(' I am young ');
 ELSIF v_myage < 30 THEN
     DBMS_OUTPUT.PUT_LINE(' I am in my twenties ');
 ELSIF v_myage < 40 THEN
     DBMS_OUTPUT.PUT_LINE(' I am in my thirties ');
 ELSE
     DBMS_OUTPUT.PUT_LINE(' I am always young ');
END IF;…
```

Tell Me/Show Me

IF Statement With Multiple Expressions

An IF statement can have multiple conditional expressions related with logical operators, such as AND, OR, and NOT.
For example:

```
DECLARE
  v_myage       NUMBER       := 31;
  v_myfirstname VARCHAR2(11) := 'Christopher';
BEGIN
  IF v_myfirstname ='Christopher' AND v_myage < 11
  THEN
    DBMS_OUTPUT.PUT_LINE(' I am a child named Christopher');
  END IF;
END;
```

The condition uses the AND operator and, therefore, it evaluates to TRUE only if both the above conditions are evaluated as TRUE.
There is no limitation on the number of conditional expressions; however, these statements must be connected with the appropriate logical operators.

Tell Me/Show Me

NULL Values in IF Statements

In this example, the v_myage variable is declared but is not initialized. The condition in the IF statement returns NULL, and not TRUE or FALSE. In such a case, the control goes to the ELSE statement because just like FALSE, NULL is not TRUE.

```
DECLARE
  v_myage NUMBER;
BEGIN
  IF v_myage < 11
  THEN
    DBMS_OUTPUT.PUT_LINE(' I am a child ');
  ELSE
    DBMS_OUTPUT.PUT_LINE(' I am not a child ');
  END IF;
END;
```

Tell Me/Show Me

Handling Nulls

When working with nulls, you can avoid some common mistakes by keeping in mind the following rules:

- Simple comparisons involving nulls always yield `NULL`.
- Applying the logical operator `NOT` to a null yields `NULL`.
- In conditional control statements, if a condition yields `NULL`, it behaves just like a `FALSE`, and the associated sequence of statements is not executed.

Tell Me/Show Me

Handling Nulls (continued)

- Consider the following example:

```
x := 5;
y := NULL;
...
IF x != y THEN  … -- yields NULL, not TRUE and the
        sequence of statements are not executed
END IF;
```

- You can expect the sequence of statements to execute because x and y seem unequal. But, nulls are indeterminate. Whether x is equal to y is unknown. Therefore, the `IF` condition yields `NULL` and the sequence of statements is bypassed.

Tell Me/Show Me

Handling Nulls (continued)

- Consider the following example:

```
a := NULL;
b := NULL;
...
IF a = b THEN … -- yields NULL, not TRUE and the
                   sequence of statements are not executed
END IF;
```

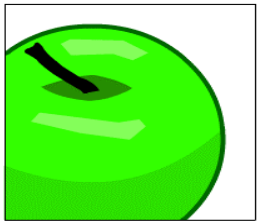
- In the example, you can expect the sequence of statements to execute because a and b seem equal. But, again, equality is unknown. So the `IF` condition yields `NULL` and the sequence of statements is bypassed.

Tell Me/Show Me

Guidelines for Using `IF` Statements

- You can perform actions selectively when a specific condition is being met.
- When writing code, remember the spelling of the keywords:
  - `ELSIF` is one word.
  - `END IF` is two words.
- If the controlling Boolean condition is `TRUE`, then the associated sequence of statements is executed; if the controlling Boolean condition is `FALSE` or `NULL`, then the associated sequence of statements is passed over. Any number of `ELSIF` clauses is permitted.
- Indent the conditionally executed statements for clarity.

Tell Me/Show Me



Terminology

Key terms used in this lesson include:

IF

Condition

CASE

LOOP

Summary

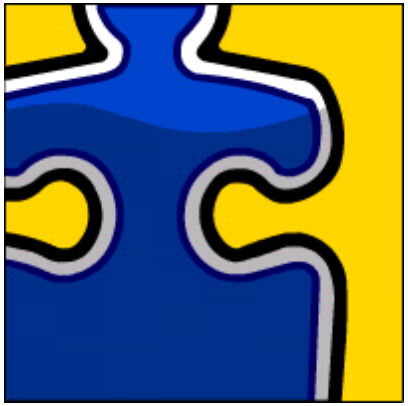
In this lesson, you learned to:

- Describe a use for conditional control structures
- List the types of conditional control structures
- Construct and use an `IF` statement
- Construct and use an `IF-THEN-ELSE` statement
- Create a PL/SQL to handle the null condition in `IF` statements

Try It/Solve It

The exercises in this lesson cover the following topics:

- Identifying the uses and types of conditional control structures
- Constructing and using an `IF` statement
- Constructing and using an `IF…THEN…ELSE` statement
- Handling the null condition in `IF` statements



Conditional Control: CASE Statements

What Will I Learn?

In this lesson, you will learn to:

- Construct and use `CASE` statements in PL/SQL
- Construct and use `CASE` expressions in PL/SQL
- Include the correct syntax to handle null conditions in PL/SQL `CASE` statements
- Include the correct syntax to handle Boolean conditions in PL/SQL `IF` and `CASE` statements

Why Learn It?

In this lesson, you learn how to use CASE statements and CASE expressions in a PL/SQL block.

CASE statements are similar to `IF` statements, but are often easier to write and easier to read.

CASE expressions are functions that return one of a number of values into a variable.

Tell Me/Show Me

Using a CASE Statement

Look at this IF statement What do you notice?

```
DECLARE
   v_numvar    NUMBER;
BEGIN
  ...
  IF    v_numvar = 5  THEN statement_1; statement_2;
  ELSIF v_numvar = 10 THEN statement_3;
  ELSIF v_numvar = 12 THEN statement_4; statement_5;
  ELSIF v_numvar = 27 THEN statement_6;
  ELSIF v_numvar ... – and so on
  ELSE  statement_15;
  END IF;
  ...
END;
```

All the conditions test the same variable v_numvar. And the coding is very repetitive: v_numvar is coded many times.

Tell Me/Show Me

Using a CASE Statement (continued)

Here is the same logic, but using a CASE statement:

```
DECLARE
   v_numvar    NUMBER;
BEGIN
  ...
  CASE v_numvar
    WHEN 5  THEN statement_1; statement_2;
    WHEN 10 THEN statement_3;
    WHEN 12 THEN statement_4; statement_5;
    WHEN 27 THEN statement_6;
    WHEN  ... – and so on
    ELSE statement_15;
  END CASE;
  ...
END;
```

It's much neater, isn't it? `v_numvar` is referenced only once. Easier to write, and easier to read.

Tell Me/Show Me

CASE Statements: A Second Example

```
DECLARE
   v_deptid   departments.department_id%TYPE;
   v_deptname departments.department_name%TYPE;
   v_emps     NUMBER;
   v_mngid    departments.manager_id%TYPE := 108;
BEGIN
  CASE  v_mngid
    WHEN  108 THEN
      SELECT department_id, department_name
        INTO v_deptid, v_deptname FROM departments
        WHERE manager_id=108;
      SELECT count(*) INTO v_emps FROM employees
        WHERE department_id=v_deptid;
    WHEN  200 THEN
    ...
  END CASE;
  DBMS_OUTPUT.PUT_LINE ('You are working in the '|| v_deptname||
  ' department. There are '||v_emps ||' employees in this
  department');
END;
```

Tell Me/Show Me

Using a CASE Expression

You want to assign a value to one variable that depends on the value in another variable. Look at this IF statement:

```
DECLARE
  v_out_var   VARCHAR2(15);
  v_in_var    NUMBER;
BEGIN
  ...
  IF    v_in_var = 1  THEN v_out_var := 'Low value';
  ELSIF v_in_var = 50 THEN v_out_var := 'Middle value';
  ELSIF v_in_var = 99 THEN v_out_var := 'High value';
  ELSE                     v_out_var := 'Other value';
  END IF;
  ...
END;
```

Again, the coding is very repetitive.

Tell Me/Show Me

Using a CASE Expression (continued)

Here is the same logic, but using a CASE expression:

```
DECLARE
  v_out_var   VARCHAR2(15);
  v_in_var    NUMBER;
BEGIN
  ...
  v_out_var :=
    CASE v_in_var
      WHEN 1  THEN 'Low value'
      WHEN 50 THEN 'Middle value'
      WHEN 99 THEN 'High value'
      ELSE         'Other value'
    END;
  ...
END;
```

Again, it is much neater than the equivalent IF statement.

Tell Me/Show Me

CASE Expressions

A CASE expression selects one of a number of results and returns it into a variable.

In the syntax, *expressionN* can be a literal value, such as 50, or an expression, such as (27+23) or (v_other_var*2).

```
variable_name :=
  CASE selector
    WHEN expression1 THEN result1
    WHEN expression2 THEN result2
    ...
    WHEN expressionN THEN resultN
   [ELSE resultN+1]
  END;
```

Tell Me/Show Me

CASE Expressions: A Second Example

```
DECLARE
   v_grade     CHAR(1) := 'A';
   v_appraisal VARCHAR2(20);
BEGIN
   v_appraisal :=
      CASE v_grade
         WHEN 'A' THEN 'Excellent'
         WHEN 'B' THEN 'Very Good'
         WHEN 'C' THEN 'Good'
         ELSE 'No such grade'
      END;
   DBMS_OUTPUT.PUT_LINE ('Grade: '|| v_grade ||
                         ' Appraisal ' || v_appraisal);
END;
```

```
Grade: A
Appraisal Excellent

Statement processed.
```

Tell Me/Show Me

CASE Expressions: A Third Example

What do you think will be displayed when this block is executed?

```
DECLARE
  v_out_var   VARCHAR2(15);
  v_in_var    NUMBER := 20;
BEGIN
  v_out_var :=
    CASE v_in_var
      WHEN 1        THEN 'Low value'
      WHEN v_in_var THEN 'Same value'
      WHEN 20       THEN 'Middle value'
      ELSE               'Other value'
    END;
  DBMS_OUTPUT.PUT_LINE(v_out_var);
END;
```

Tell Me/Show Me

Searched CASE Expressions

PL/SQL also provides a searched CASE expression, which has the following form:

```
CASE
   WHEN search_condition1 THEN result1
   WHEN search_condition2 THEN result2
   ...
   WHEN search_conditionN THEN resultN
  [ELSE resultN+1]
END;
```

A searched CASE expression 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.

Tell Me/Show Me

Searched CASE Expressions: An Example

```
DECLARE
  v_grade     CHAR(1) := 'A';
  v_appraisal VARCHAR2(20);
BEGIN
  v_appraisal :=
     CASE                        -- no selector here
       WHEN v_grade = 'A' THEN 'Excellent'
       WHEN v_grade IN ('B','C') THEN 'Good'
       ELSE 'No such grade'
     END;
   DBMS_OUTPUT.PUT_LINE ('Grade: '|| v_grade ||
                         ' Appraisal ' || v_appraisal);
END;
```

Tell Me/Show Me

How are CASE Expressions Different From CASE Statements?

- **CASE expressions** return a value into a variable.
- CASE expressions end with END;
- A CASE expression is a single PL/SQL statement.

```
DECLARE
  v_grade     CHAR(1) := 'A';
  v_appraisal VARCHAR2(20);
BEGIN
  v_appraisal :=
    CASE
      WHEN v_grade = 'A' THEN 'Excellent'
      WHEN v_grade IN ('B','C') THEN 'Good'
      ELSE 'No such grade'
    END;
  DBMS_OUTPUT.PUT_LINE ('Grade: '|| v_grade ||
                         ' Appraisal ' || v_appraisal);
END;
```

Tell Me/Show Me

How are CASE Expressions Different From CASE Statements? (continued)

- **CASE statements** evaluate conditions and perform actions
- A CASE statement can contain many PL/SQL statements.
- CASE statements end with END CASE;.

```
DECLARE
   v_grade CHAR(1) := 'A';
BEGIN
  CASE
    WHEN v_grade = 'A' THEN
          DBMS_OUTPUT.PUT_LINE ('Excellent');
    WHEN v_grade IN ('B','C') THEN
          DBMS_OUTPUT.PUT_LINE ('Good');
    ELSE
          DBMS_OUTPUT.PUT_LINE('No such grade');
  END CASE;
END;
```

Tell Me/Show Me

Logic Tables

When using IF and CASE statements you often need to combine conditions using AND, OR, and NOT. The following Logic Tables show the results of all possible combinations of two conditions.

AND	TRUE	FALSE	NULL	OR	TRUE	FALSE	NULL	NOT	
TRUE	TRUE	(1) FALSE	NULL	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE
FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	NULL	FALSE	TRUE
NULL	NULL	FALSE	NULL	NULL	TRUE	NULL	NULL	NULL	NULL

Example: (1) TRUE AND FALSE is FALSE

Tell Me/Show Me

Boolean Conditions

What is the value of `v_flag` in each case?

```
v_flag := v_reorder_flag AND v_available_flag;
```

V_REORDER_FLAG	V_AVAILABLE_FLAG	V_FLAG
TRUE	TRUE	?
TRUE	FALSE	?
NULL	TRUE	?
NULL	FALSE	?

Tell Me/Show Me

Terminology

Key terms used in this lesson include:

CASE expression

CASE statement

Logic Tables

Summary

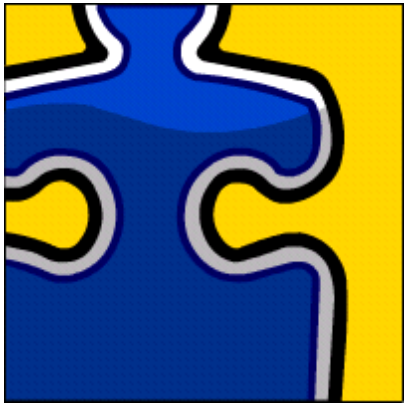
In this lesson, you learned to:

- Construct and use `CASE` statements in PL/SQL
- Construct and use `CASE` expressions in PL/SQL
- Include the correct syntax to handle null conditions in PL/SQL `CASE` statements
- Include the correct syntax to handle Boolean conditions in PL/SQL `IF` and `CASE` statements

Try It/Solve It

The exercises in this lesson cover the following topics:

- Constructing and using `CASE` statements
- Constructing and using `CASE` expressions
- Handling null conditions in `CASE` statements
- Handling Boolean conditions in `IF` and `CASE` statements



Iterative Control: Basic Loops

What Will I Learn?

In this lesson, you will learn to:

- Describe the need for `LOOP` statements in PL/SQL
- Recognize different types of `LOOP` statements
- Create PL/SQL containing a basic loop and an `EXIT` statement
- Create PL/SQL containing a basic loop and an `EXIT` statement with conditional termination

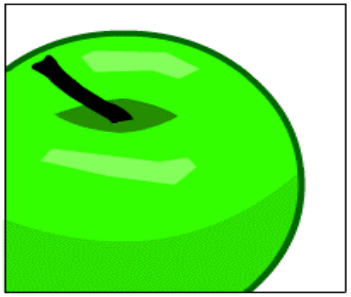
Why Learn It?

Looping constructs are the second type of control structure. Loops are mainly used to execute statements repeatedly until an `EXIT` condition is reached.

PL/SQL provides three ways to structure loops to repeat a statement or a sequence of statements multiple times. These are basic loops, `FOR` loops, and `WHILE` loops.

This lesson introduces the three loop types and discusses basic loops in greater detail.

Tell Me/Show Me



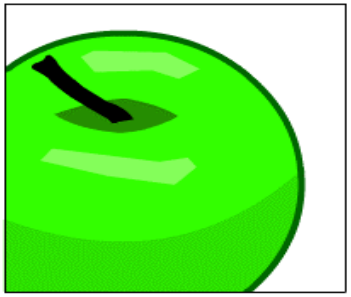
Iterative Control: `LOOP` Statements

Loops repeat a statement or a sequence of statements multiple times.

PL/SQL provides the following types of loops:

- Basic loops that perform repetitive actions without overall conditions
- `FOR` loops that perform iterative actions based on a counter
- `WHILE` loops that perform repetitive actions based on a condition

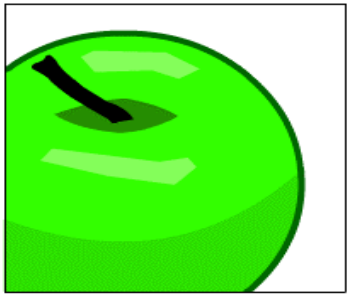
Tell Me/Show Me



Basic Loops

The simplest form of a LOOP statement is the basic (or infinite) loop, which encloses a sequence of statements between the keywords LOOP and END LOOP. Use the basic loop when the statements inside the loop must execute at least once.

Tell Me/Show Me



Basic Loops

Each time the flow of execution reaches the `END LOOP` statement, control is returned to the corresponding `LOOP` statement above it. A basic loop allows the execution of its statements at least once, even if the `EXIT` condition is already met upon entering the loop. Without the `EXIT` statement, the loop would be infinite.

Syntax:

```
LOOP
  statement1;
  . . .
  EXIT [WHEN condition];
END LOOP;
```

Tell Me/Show Me

Basic Loops

In this example, three new location IDs for the country code of CA and the city of Montreal are inserted.

```
DECLARE
  v_countryid    locations.country_id%TYPE := 'CA';
  v_loc_id       locations.location_id%TYPE;
  v_counter      NUMBER(2) := 1;
  v_new_city     locations.city%TYPE := 'Montreal';
BEGIN
  SELECT MAX(location_id) INTO v_loc_id FROM locations
    WHERE country_id = v_countryid;
  LOOP
    INSERT INTO locations(location_id, city, country_id)
    VALUES((v_loc_id + v_counter), v_new_city, v_countryid);
    v_counter := v_counter + 1;
    EXIT WHEN v_counter > 3;
  END LOOP;
END;
```

Tell Me/Show Me

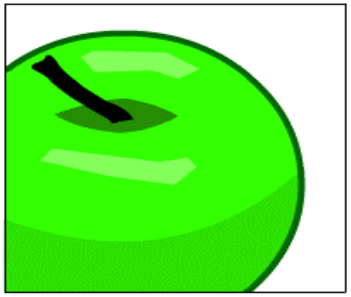
Basic Loops

The EXIT Statement

You can use the EXIT statement to terminate a loop. The control passes to the next statement after the END LOOP statement. You can issue EXIT either as an action within an IF statement or as a stand-alone statement within the loop.

```
DECLARE
  v_counter NUMBER := 1;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE('The square of '
              ||v_counter||' is: '|| POWER(v_counter,2));
    v_counter := v_counter + 1;
    IF v_counter > 10 THEN
      EXIT;
    END IF;
  END LOOP;
END;
```

Tell Me/Show Me

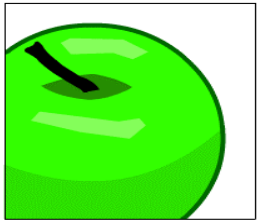


Basic Loops

The `EXIT` Statement (continued)

- The `EXIT` statement must be placed inside a loop.
- If the `EXIT` condition is placed at the top of the loop (before any of the other executable statements) and that condition is initially true, then the loop exits and the other statements in the loop never execute.
- A basic loop can contain multiple `EXIT` statements, but you should have only one `EXIT` point.

Tell Me/Show Me



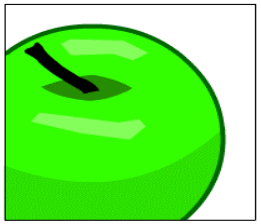
Basic Loops

The EXIT WHEN Statement

Use the WHEN clause to allow conditional termination of the loop. When the EXIT statement is encountered, the condition in the WHEN clause is evaluated. If the condition yields TRUE, then the loop ends and control passes to the next statement after the loop.

```
DECLARE
  v_counter NUMBER := 1;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE('The square of '
              ||v_counter||' is: '|| POWER(v_counter,2));
    v_counter :=v_counter + 1;
    EXIT WHEN v_counter > 10;
  END LOOP;
END;
```

Tell Me/Show Me



Terminology

Key terms used in this lesson include:

Basic (Infinite) loop

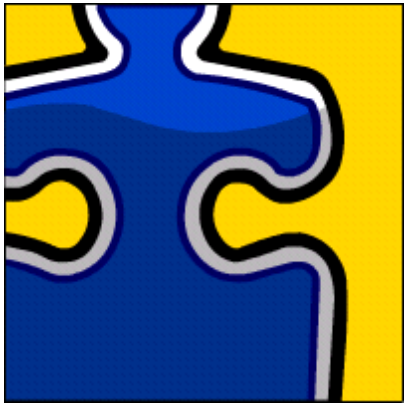
EXIT

Summary

In this lesson, you learned to:

- Describe the need for `LOOP` statements in PL/SQL
- Recognize different types of `LOOP` statements
- Create PL/SQL containing a basic loop and an `EXIT` statement
- Create PL/SQL containing a basic loop and an `EXIT` statement with conditional termination

Try It/Solve It



The exercises in this lesson cover the following topics:

- Describing the need for `LOOP` statements in PL/SQL
- Identifying different types of `LOOP` statements
- Using basic loops with `EXIT` conditions
- Using basic loops with `EXIT WHEN` conditions

Iterative Control: `WHILE` and `FOR` Loops

What Will I Learn?

In this lesson, you will learn to:

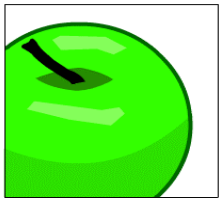
- Construct and use the `WHILE` looping construct in PL/SQL
- Construct and use the `FOR` looping construct in PL/SQL
- Describe when a `WHILE` loop is used in PL/SQL
- Describe when a `FOR` loop is used in PL/SQL

Why Learn It?

The previous lesson discussed the basic loop, which required that the statements inside the loop execute at least once.

This lesson introduces the WHILE loop and FOR loop. The WHILE loop is a looping construct, which requires that the EXIT condition be evaluated at the start of each iteration. The FOR loop should be used if the number of iterations is known.

Tell Me/Show Me



WHILE Loops:

You can use the WHILE loop to repeat a sequence of statements until the controlling condition is no longer TRUE. The condition is evaluated at the start of each iteration. The loop terminates when the condition is FALSE or NULL. If the condition is FALSE or NULL at the start of the loop, then no further iterations are performed.

Syntax:

```
WHILE condition LOOP
  statement1;
  statement2;
  . . .
END LOOP;
```

Tell Me/Show Me

WHILE Loops (continued):

- In the syntax:
  - *condition* is a Boolean variable or expression (TRUE, FALSE, or NULL)
  - *statement* can be one or more PL/SQL or SQL statements
- If the variables involved in the conditions do not change during the body of the loop, then the condition remains TRUE and the loop does not terminate.
- **Note:** If the condition yields NULL, then the loop is bypassed and the control passes to the next statement.

```
WHILE condition LOOP
  statement1;
  statement2;
  . . .
END LOOP;
```

Tell Me/Show Me

WHILE Loops (continued):

In the example in the slide, three new location IDs for the country code CA and the city of Montreal are being added. The counter is explicitly declared in this example.

```
DECLARE
  v_countryid   locations.country_id%TYPE := 'CA';
  v_loc_id      locations.location_id%TYPE;
  v_new_city    locations.city%TYPE := 'Montreal';
  v_counter     NUMBER := 1;
BEGIN
  SELECT MAX(location_id) INTO v_loc_id FROM locations
    WHERE country_id = v_countryid;
  WHILE v_counter <= 3 LOOP
    INSERT INTO locations(location_id, city, country_id)
    VALUES((v_loc_id + v_counter), v_new_city, v_countryid);
    v_counter := v_counter + 1;
  END LOOP;
END;
```

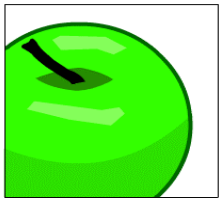
Tell Me/Show Me

WHILE Loops (continued):

With each iteration through the WHILE loop, a counter (v_counter) is incremented. If the number of iterations is less than or equal to the number 3, then the code within the loop is executed and a row is inserted into the locations table. After the counter exceeds the number of new locations for this city and country, the condition that controls the loop evaluates to FALSE and the loop is terminated.

```
DECLARE
  v_countryid   locations.country_id%TYPE := 'CA';
  v_loc_id      locations.location_id%TYPE;
  v_new_city    locations.city%TYPE := 'Montreal';
  v_counter     NUMBER := 1;
BEGIN
  SELECT MAX(location_id) INTO v_loc_id FROM locations
    WHERE country_id = v_countryid;
  WHILE v_counter <= 3 LOOP
    INSERT INTO locations(location_id, city, country_id)
    VALUES((v_loc_id + v_counter), v_new_city, v_countryid);
    v_counter := v_counter + 1;
  END LOOP;
END;
```

Tell Me/Show Me



FOR Loops:

FOR loops have the same general structure as the basic loop. In addition, they have a control statement before the LOOP keyword to set the number of iterations that PL/SQL performs.

```
FOR counter IN [REVERSE]
    lower_bound..upper_bound LOOP
  statement1;
  statement2;
  . . .
END LOOP;
```

- Use a FOR loop to shortcut the test for the number of iterations.
- Do not declare the counter; it is declared implicitly.
- *lower_bound .. upper_bound* is the required syntax.

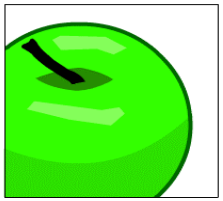
Tell Me/Show Me

```
FOR counter IN [REVERSE]
    lower_bound..upper_bound LOOP
  statement1;
  statement2;
  . . .
END LOOP;
```

FOR Loops (continued):

- In the syntax:
  - *Counter* is an implicitly declared integer whose value automatically increases or decreases (decreases if the REVERSE keyword is used) by 1 on each iteration of the loop until the upper or lower bound is reached.
  - REVERSE causes the counter to decrement with each iteration from the upper bound to the lower bound. (Note that the lower bound is still referenced first.)
  - *lower_bound* specifies the lower bound for the range of counter values.
  - *upper_bound* specifies the upper bound for the range of counter values.
- Do not declare the counter; it is declared implicitly as an integer.

Tell Me/Show Me



FOR Loops (continued):

- **Note:** The sequence of statements is executed each time the counter is incremented, as determined by the two bounds. The lower bound and upper bound of the loop range can be literals, variables, or expressions, but must evaluate to integers. The bounds are rounded to integers—that is, 11/3 or 8/5 are valid upper or lower bounds. The lower bound and upper bound are inclusive in the loop range. If the lower bound of the loop range evaluates to a larger integer than the upper bound, then the sequence of statements will not be executed.
  For example, the following statement is executed only once:

```
FOR i in 3..3
LOOP
  statement1;
END LOOP;
```

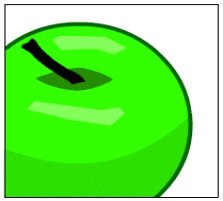
Tell Me/Show Me

FOR Loops (continued):

You have already learned how to insert three new locations for the country code CA and the city Montreal by using the simple LOOP and the WHILE loop. The slide shows you how to achieve the same by using the FOR loop.

```
DECLARE
  v_countryid   locations.country_id%TYPE := 'CA';
  v_loc_id      locations.location_id%TYPE;
  v_new_city    locations.city%TYPE := 'Montreal';
BEGIN
  SELECT MAX(location_id) INTO v_loc_id
    FROM locations
    WHERE country_id = v_countryid;
  FOR i IN 1..3 LOOP
    INSERT INTO locations(location_id, city, country_id)
    VALUES((v_loc_id + i), v_new_city, v_countryid );
  END LOOP;
END;
```

Tell Me/Show Me



FOR Loops:

Guidelines

- Reference the counter within the loop only; it is undefined outside the loop.
- Do not reference the counter as the target of an assignment.
- Neither loop bound should be NULL.

Tell Me/Show Me

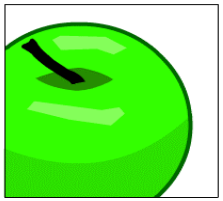
FOR Loops:

While writing a FOR loop, the lower and upper bounds of a LOOP statement do not need to be numeric literals. They can be expressions that convert to numeric values.

Example:

```
DECLARE
  v_lower  NUMBER := 1;
  v_upper  NUMBER := 100;
BEGIN
  FOR i IN v_lower..v_upper LOOP
  ...
  END LOOP;
END;
```

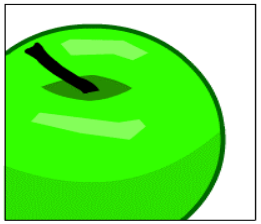
Tell Me/Show Me



Guidelines For Using Loops :

- Use the basic loop when the statements inside the loop must execute at least once.
- Use the `WHILE` loop if the condition has to be evaluated at the start of each iteration.
- Use a `FOR` loop if the number of iterations is known.

Tell Me/Show Me



Terminology

Key terms used in this lesson include:

WHILE loops

FOR loops

Summary

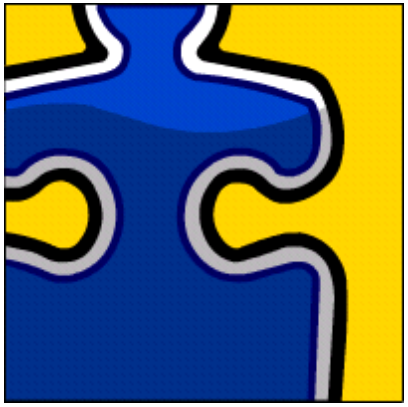
In this lesson, you learned to:

- Construct and use the `WHILE` looping construct in PL/SQL
- Construct and use the `FOR` looping construct in PL/SQL
- Describe when a `WHILE` loop is used in PL/SQL
- Describe when a `FOR` loop is used in PL/SQL

Try It/Solve It

The exercises in this lesson cover the following topics:

- Constructing and using `WHILE` loops in PL/SQL
- Constructing and using `FOR` loops in PL/SQL
- Describing when a `WHILE` loop is used in PL/SQL
- Describing when a `FOR` loop is used in PL/SQL



Iterative Control: Nested Loops

What Will I Learn?

In this lesson, you will learn to:

- Construct and execute PL/SQL using nested loops
- Label loops and use the labels in EXIT statements
- Evaluate a nested loop construct and identify the exit point

Why Learn It?

You’ve learned about looping constructs in PL/SQL. This lesson discusses how you can nest loops to multiple levels. You can nest `FOR`, `WHILE`, and basic loops within one another.

Tell Me/Show Me

Nested Loops

In PL/SQL, you can nest loops to multiple levels. You can nest `FOR`, `WHILE`, and basic loops within one another.

Consider the following example:

```
BEGIN
  FOR v_outerloop IN 1..3 LOOP
    FOR v_innerloop IN REVERSE 1..5 LOOP
     DBMS_OUTPUT.PUT_LINE('Outer loop is:'||v_outerloop||
                          ' and inner loop is: '||v_innerloop);
    END LOOP;
  END LOOP;
END;
```

Tell Me/Show Me

Nested Loops

This example contains EXIT conditions in nested basic loops.

```
DECLARE
  v_outer_done   CHAR(3) := 'NO';
  v_inner_done   CHAR(3) := 'NO';
BEGIN
  LOOP              -- outer loop
    ...
    LOOP            -- inner loop
      ...
      ...           -- step A
      EXIT WHEN v_inner_done = 'YES';
      ...

    END LOOP;
    ...
    EXIT WHEN v_outer_done = 'YES';
    ...
  END LOOP;
END;
```

What if you want to exit from the outer loop at step A?

Tell Me/Show Me

Use labels to distinguish between the loops:

```
DECLARE
  ...
BEGIN
 <<outer_loop>>
  LOOP                -- outer loop
    ...
    <<inner_loop>>
    LOOP              -- inner loop
      EXIT outer_loop WHEN ...  -- Exits both loops
      EXIT WHEN v_inner_done = 'YES';
      ...
    END LOOP;
    ...
    EXIT WHEN v_outer_done = 'YES';
    ...
  END LOOP;
END;
```

Tell Me/Show Me

Loop Labels

Loop label names follow the same rules as other identifiers. A label is placed before a statement, either on the same line or on a separate line. In `FOR` or `WHILE` loops, place the label before `FOR` or `WHILE` within label delimiters (<<*label*>>). If the loop is labeled, the label name can optionally be included after the `END LOOP` statement for clarity.

Tell Me/Show Me

Labels

Label basic loops by placing the label before the word LOOP within label delimiters (<<*label*>>).

```
DECLARE
  v_outerloop PLS_INTEGER :=0;
  v_innerloop PLS_INTEGER :=5;
BEGIN
 <<Outer_loop>>
  LOOP
    v_outerloop := v_outerloop + 1;
    v_innerloop := 5;
    EXIT WHEN v_outerloop > 3;
    <<Inner_loop>>
    LOOP
      DBMS_OUTPUT.PUT_LINE('Outer loop is:'||v_outerloop||
                           ' and inner loop is: '||v_innerloop);
      v_innerloop := v_innerloop - 1;
      EXIT WHEN v_innerloop =0;
    END LOOP Inner_loop;
  END LOOP Outer_loop;
END;
```

Tell Me/Show Me

Nested Loops and Labels

In this example, there are two loops. The outer loop is identified by the label `<<Outer_Loop>>`, and the inner loop is identified by the label `<<Inner_Loop>>`.

```
…BEGIN
  <<Outer_loop>>
  LOOP
    v_counter := v_counter+1;
  EXIT WHEN v_counter>10;
    <<Inner_loop>>
    LOOP
      ...
      EXIT Outer_loop WHEN v_total_done = 'YES';
      -- Leave both loops
      EXIT WHEN v_inner_done = 'YES';
      -- Leave inner loop only
      ...
    END LOOP Inner_loop;
    ...
  END LOOP Outer_loop;
END;
```

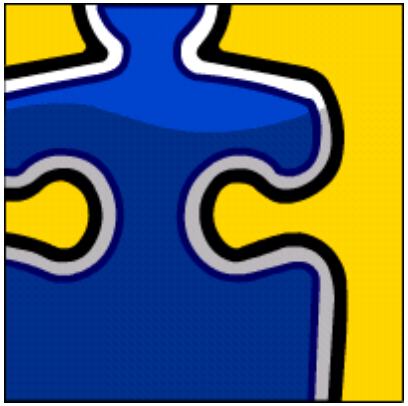
Summary

In this lesson, you learned to:

- Construct and execute PL/SQL using nested loops
- Label loops and use the labels in EXIT statements
- Evaluate a nested loop construct and identify the exit point

Try It/Solve It

The exercises in this lesson cover the following topics:



- Constructing and executing PL/SQL using nested loops
- Labeling loops and using the labels in EXIT statements
- Evaluating a nested loop construct and identifying the exit point