6

Writing Control Structures

ORACLE

Objectives

After completing this lesson, you should be able to do the following:

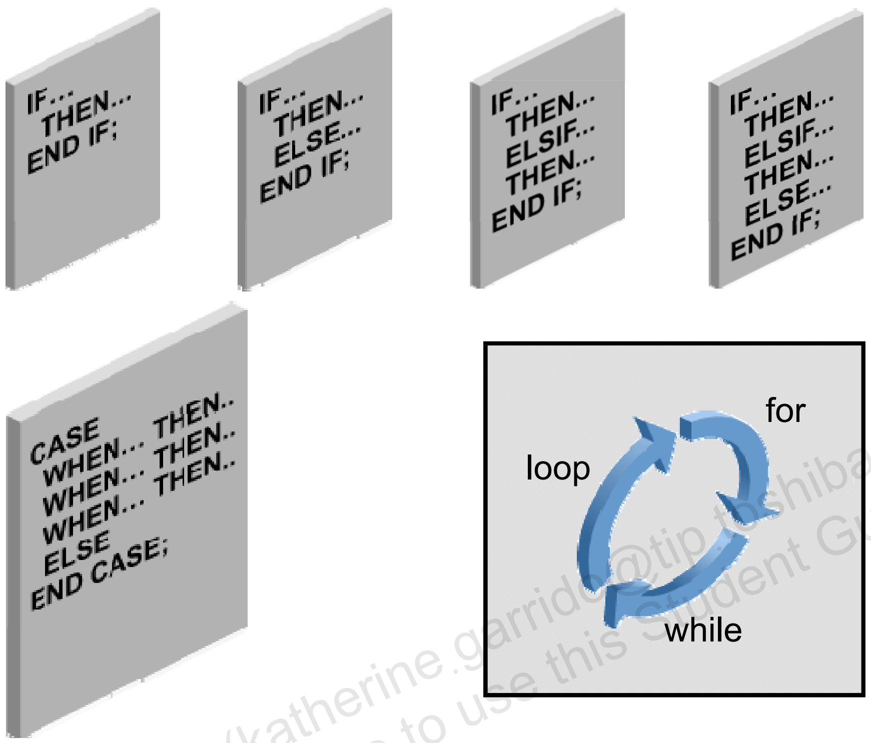
- Identify the uses and types of control structures
- Construct an `IF` statement
- Use `CASE` statements and `CASE` expressions
- Construct and identify loop statements
- Use guidelines when using conditional control structures

ORACLE

You have learned to write PL/SQL blocks containing declarative and executable sections. You have also learned to include expressions and SQL statements in the executable block.

In this lesson, you learn how to use control structures such as `IF` statements, `CASE` expressions, and `LOOP` structures in a PL/SQL block.

Controlling Flow of Execution


IF... THEN... END IF;
IF... THEN... ELSE... END IF;
IF... THEN... ELSIF... THEN... END IF;
IF... THEN... ELSIF... THEN... ELSE... END IF;
CASE WHEN... THEN.. WHEN... THEN.. WHEN... THEN.. ELSE END CASE;
for
loop
while

You can change the logical flow of statements within the PL/SQL block with a number of control structures. This lesson addresses four types of PL/SQL control structures: conditional constructs with the `IF` statement, `CASE` expressions, `LOOP` control structures, and the `CONTINUE` statement.

Agenda

- Using `IF` statements
- Using `CASE` statements and `CASE` expressions
- Constructing and identifying loop statements

IF Statement

Syntax:

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

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

In the syntax:

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 may include additional 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.

In the syntax:

`ELSIF`	Is a keyword that introduces a Boolean expression (If the first condition yields `FALSE` or `NULL`, the `ELSIF` keyword introduces additional conditions.)
`ELSE`	Introduces the default clause that is executed if and only if none of the earlier predicates (introduced by `IF` and `ELSIF`) are TRUE. The tests are executed in sequence so that a later predicate that might be true is preempted by an earlier predicate that is true.
`END IF`	Marks the end of an `IF` statement

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.

Simple `IF` Statement

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

```
anonymous block completed
```

Simple `IF` Example

The slide shows an example of a simple `IF` statement with the `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.

Adding Conditional Expressions

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

For example:

```
IF (myfirstname='Christopher' AND v_myage <11)
…
```

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

IF THEN 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;
/
```

```
anonymous block completed
 I am not a child
```

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

The output of the block is shown below the code.

IF ELSIF ELSE Clause

```
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;
/
```

```
anonymous block completed
 I am in my thirties
```

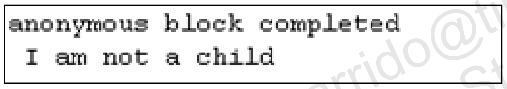
The IF clause may contain multiple ELSIF clauses and an ELSE clause. The example illustrates the following characteristics of these clauses:

- The ELSIF clauses can have conditions, unlike the ELSE clause.
- The condition for ELSIF should be followed by the THEN clause, which is executed if the condition for ELSIF returns TRUE.
- When you have multiple ELSIF clauses, if the first condition is FALSE or NULL, the control shifts to the next ELSIF clause.
- Conditions are evaluated one by one from the top.
- If all conditions are FALSE or NULL, the statements in the ELSE clause are executed.
- The final ELSE clause is optional.

In the example, the output of the block is shown below the code.

`NULL` Value in `IF` Statement

```
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;
/
```



```
anonymous block completed
 I am not a child
```

In the example shown in the slide, the variable `v_myage` is declared but not initialized. The condition in the `IF` statement returns `NULL` rather than `TRUE` or `FALSE`. In such a case, the control goes to the `ELSE` statement.

Guidelines

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

Agenda

- Using `IF` statements
- Using `CASE` statements and `CASE` expressions
- Constructing and identifying loop statements

ORACLE

`CASE` Expressions

- A `CASE` expression selects a result and returns it.
- To select the result, the `CASE` expression uses expressions. The value returned by these expressions is used to select one of several alternatives.

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

A `CASE` expression returns a result based on one or more alternatives. To return the result, the `CASE` expression uses a *selector*, which is an expression whose value is used to return one of several alternatives. The selector is followed by one or more `WHEN` clauses that are checked sequentially. The value of the selector determines which result is returned. If the value of the selector equals the value of a `WHEN` clause expression, that `WHEN` clause is executed and that result is returned.

PL/SQL also provides a searched `CASE` expression, which has the 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. Furthermore, the `WHEN` clauses in `CASE` expressions contain search conditions that yield a Boolean value rather than expressions that can yield a value of any type.

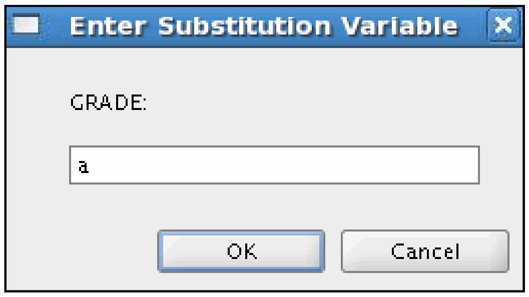
CASE Expressions: Example

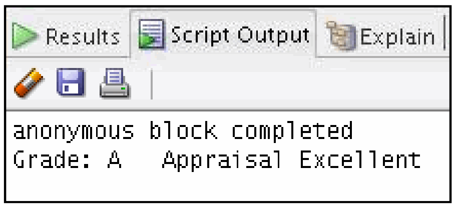
```
SET VERIFY OFF
DECLARE
   v_grade  CHAR(1) := UPPER('&grade');
   v_appraisal VARCHAR2(20);
BEGIN
   v_appraisal := CASE v_grade
         WHEN 'A' THEN 'Excellent'
         WHEN 'B' THEN 'Very Good'
         WHEN 'C' THEN 'Good'
         ELSE 'No such grade'
      END;
DBMS_OUTPUT.PUT_LINE ('Grade: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

In the example in the slide, the `CASE` expression uses the value in the `v_grade` variable as the expression. This value is accepted from the user by using a substitution variable. Based on the value entered by the user, the `CASE` expression returns the value of the `v_appraisal` variable based on the value of the `v_grade` value.

Result

When you enter `a` or `A` for `v_grade`, as shown in the Substitution Variable window, the output of the example is as follows:





```
anonymous block completed
Grade: A   Appraisal Excellent
```

Searched CASE Expressions

```
DECLARE
   v_grade  CHAR(1) := UPPER('&grade');
   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;
/
```

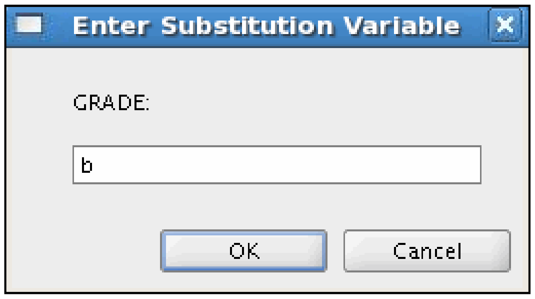
ORACLE

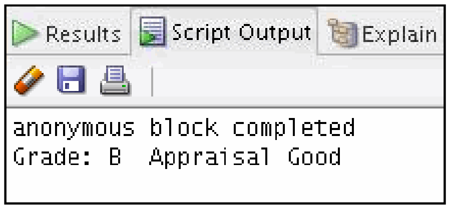
In the previous example, you saw a single test expression, the `v_grade` variable. The `WHEN` clause compared a value against this test expression.

In searched `CASE` statements, you do not have a test expression. Instead, the `WHEN` clause contains an expression that results in a Boolean value. The same example is rewritten in this slide to show searched `CASE` statements.

Result

The output of the example is as follows when you enter `b` or `B` for `v_grade`:





```
anonymous block completed
Grade: B  Appraisal Good
```

`CASE` Statement

```
DECLARE
   v_deptid NUMBER;
   v_deptname VARCHAR2(20);
   v_emps NUMBER;
   v_mngid NUMBER:= 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;
/
```

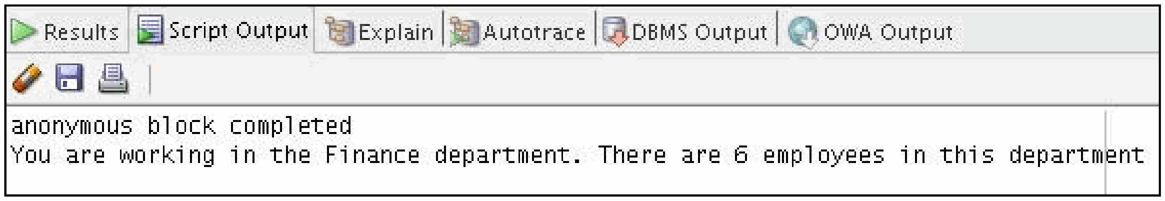
Recall the use of the `IF` statement. You may include *n* number of PL/SQL statements in the `THEN` clause and also in the `ELSE` clause. Similarly, you can include statements in the `CASE` statement, which is more readable compared to multiple `IF` and `ELSIF` statements.

How a `CASE` Expression Differs from a `CASE` Statement

A `CASE` expression evaluates the condition and returns a value, whereas a `CASE` statement evaluates the condition and performs an action. A `CASE` statement can be a complete PL/SQL block.

- `CASE` statements end with `END CASE;`
- `CASE` expressions end with `END;`

The output of the slide code example is as follows:



```
anonymous block completed
You are working in the Finance department. There are 6 employees in this department
```

Note: Whereas an `IF` statement is able to do nothing (the conditions could be all false and the `ELSE` clause is not mandatory), a `CASE` statement must execute some PL/SQL statement.

Handling Nulls

When you are 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`.
- If the condition yields `NULL` in conditional control statements, its associated sequence of statements is not executed.

ORACLE

Consider the following example:

```
x := 5;
y := NULL;
...
IF x != y THEN  -- yields NULL, not TRUE
  --  sequence_of_statements that are not executed
END IF;
```

You may expect the sequence of statements to execute because `x` and `y` seem unequal. But nulls are indeterminate. Whether or not `x` is equal to `y` is unknown. Therefore, the `IF` condition yields `NULL` and the sequence of statements is bypassed.

```
a := NULL;
b := NULL;
...
IF a = b THEN  -- yields NULL, not TRUE
  --  sequence_of_statements that are not executed
END IF;
```

In the second example, you may 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.

Logic Tables

Build a simple Boolean condition with a comparison operator.

AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

OR	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

NOT	
TRUE	FALSE
FALSE	TRUE
NULL	NULL

You can build a simple Boolean condition by combining number, character, and date expressions with comparison operators.

You can build a complex Boolean condition by combining simple Boolean conditions with the logical operators AND, OR, and NOT. The logical operators are used to check the Boolean variable values and return TRUE, FALSE, or NULL. In the logic tables shown in the slide:

- FALSE takes precedence in an AND condition, and TRUE takes precedence in an OR condition
- AND returns TRUE only if both of its operands are TRUE
- OR returns FALSE only if both of its operands are FALSE
- NULL AND TRUE always evaluates to NULL because it is not known whether the second operand evaluates to TRUE

Note: The negation of NULL (NOT NULL) results in a null value because null values are indeterminate.

Boolean Expressions or Logical Expression?

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

```
flag := reorder_flag AND available_flag;
```

REORDER_FLAG	AVAILABLE_FLAG	FLAG
TRUE	TRUE	? (1)
TRUE	FALSE	? (2)
NULL	TRUE	? (3)
NULL	FALSE	? (4)

The `AND` logic table can help you to evaluate the possibilities for the Boolean condition in the slide.

Answers

1. `TRUE`
2. `FALSE`
3. `NULL`
4. `FALSE`

Agenda

- Using `IF` statements
- Using `CASE` statements and `CASE` expressions
- Constructing and identifying loop statements

Iterative Control: `LOOP` Statements

- Loops repeat a statement (or a sequence of statements) multiple times.
- There are three loop types:
  - Basic loop
  - `FOR` loop
  - `WHILE` loop

PL/SQL provides several facilities to structure loops to repeat a statement or sequence of statements multiple times. Loops are mainly used to execute statements repeatedly until an exit condition is reached. It is mandatory to have an exit condition in a loop; otherwise, the loop is infinite.

Looping constructs are the third type of control structures. PL/SQL provides the following types of loops:

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

Note: An `EXIT` statement can be used to terminate loops. A basic loop must have an `EXIT`. The cursor `FOR` loop (which is another type of `FOR` loop) is discussed in the lesson titled “Using Explicit Cursors.”

Basic Loops

Syntax:

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

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

EXIT Statement

You can use the EXIT statement to terminate a loop. 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. The EXIT statement must be placed inside a loop. In the latter case, you can attach a WHEN clause to enable conditional termination of the loop. When the EXIT statement is encountered, the condition in the WHEN clause is evaluated. If the condition yields TRUE, the loop ends and control passes to the next statement after the loop.
A basic loop can contain multiple EXIT statements, but it is recommended that you have only one EXIT point.

Basic Loop: Example

```
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;
/
```

The basic loop example shown in the slide is defined as follows: “Insert three new location IDs for the CA country code and the city of Montreal.”

Note

- A basic loop allows execution of its statements until the EXIT WHEN condition is met.
- If the condition is placed in the loop such that it is not checked until after the loop statements execute, the loop executes at least once.
- However, if the exit condition is placed at the top of the loop (before any of the other executable statements) and if that condition is true, the loop exits and the statements never execute.

Results

To view the output, run the code example: `code_06_22_s.sql`.

WHILE Loops

Syntax:

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

Use the WHILE loop to repeat statements while a condition is TRUE.

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, no further iterations are performed. Thus, it is possible that none of the statements inside the loop are executed.

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, the condition remains TRUE and the loop does not terminate.

Note: If the condition yields NULL, the loop is bypassed and control passes to the next statement.

WHILE Loops: 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;
/
```

In the example in the slide, three new location IDs for the `CA` country code and the city of Montreal are added.

- 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`, the code within the loop is executed and a row is inserted into the `locations` table.
- After `v_counter` exceeds the number of new locations for this city and country, the condition that controls the loop evaluates to `FALSE` and the loop terminates.

Results

To view the output, run the code example: `code_06_24_s.sql`.

FOR Loops

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

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

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 the PL/SQL performs.

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:** 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.

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 they must evaluate to integers. The bounds are rounded to integers; that is, 11/3 and 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, the sequence of statements is not executed.
For example, the following statement is executed only once:

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

FOR Loops: Example

```
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;
/
```

You have already learned how to insert three new locations for the `CA` country code and the city of Montreal by using the basic loop and the `WHILE` loop. The example in this slide shows how to achieve the same by using the `FOR` loop.

Results

To view the output, run the code example `code_06_27_s.sql`.

`FOR` Loop Rules

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

The slide lists the guidelines to follow when writing a `FOR` loop.

Note: 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;
/
```

Suggested Use of Loops

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

A basic loop allows the execution of its statement at least once, even if the condition is already met upon entering the loop. Without the EXIT statement, the loop would be infinite.

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. If the condition is FALSE at the start of the loop, no further iterations are performed.

FOR loops have a control statement before the LOOP keyword to determine the number of iterations that the PL/SQL performs. Use a FOR loop if the number of iterations is predetermined.

Nested Loops and Labels

- You can nest loops to multiple levels.
- Use labels to distinguish between blocks and loops.
- Exit the outer loop with the `EXIT` statement that references the label.

You can nest the `FOR`, `WHILE`, and basic loops within one another. The termination of a nested loop does not terminate the enclosing loop unless an exception is raised. However, you can label loops and exit the outer loop with the `EXIT` statement.

Label names follow the same rules as the other identifiers. A label is placed before a statement, either on the same line or on a separate line. White space is insignificant in all PL/SQL parsing except inside literals. Label basic loops by placing the label before the word `LOOP` within label delimiters (<<*label*>>). In `FOR` and `WHILE` loops, place the label before `FOR` or `WHILE`.

If the loop is labeled, the label name can be included (optionally) after the `END LOOP` statement for clarity.

Nested Loops and Labels: Example

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

In the example in the slide, 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>>`. The identifiers are placed before the word `LOOP` within label delimiters (<<*label*>>). The inner loop is nested within the outer loop. The label names are included after the `END LOOP` statements for clarity.

PL/SQL `CONTINUE` Statement

- Definition
  - Adds the functionality to begin the next loop iteration
  - Provides programmers with the ability to transfer control to the next iteration of a loop
  - Uses parallel structure and semantics to the `EXIT` statement
- Benefits
  - Eases the programming process
  - May provide a small performance improvement over the previous programming workarounds to simulate the `CONTINUE` statement

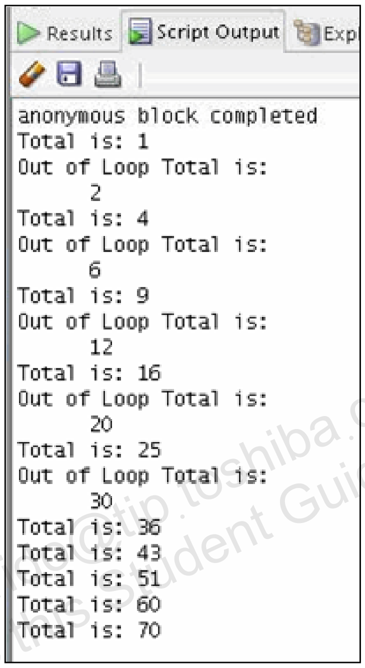
The `CONTINUE` statement enables you to transfer control within a loop back to a new iteration or to leave the loop. Many other programming languages have this functionality. With the Oracle Database 11*g* release, PL/SQL also offers this functionality. Before the Oracle Database 11*g* release, you could code a workaround by using Boolean variables and conditional statements to simulate the `CONTINUE` programmatic functionality. In some cases, the workarounds are less efficient.

The `CONTINUE` statement offers you a simplified means to control loop iterations. It may be more efficient than the previous coding workarounds.

The `CONTINUE` statement is commonly used to filter data within a loop body before the main processing begins.

PL/SQL `CONTINUE` Statement: Example 1

```
DECLARE
  v_total SIMPLE_INTEGER := 0;
BEGIN
  FOR i IN 1..10 LOOP
1   v_total := v_total + i;
    dbms_output.put_line
     ('Total is: '|| v_total);
    CONTINUE WHEN i > 5;
    v_total := v_total + i;
2   dbms_output.put_line
     ('Out of Loop Total is:
      '|| v_total);
  END LOOP;
END;
/
```



```
anonymous block completed
Total is: 1
Out of Loop Total is:
      2
Total is: 4
Out of Loop Total is:
      6
Total is: 9
Out of Loop Total is:
      12
Total is: 16
Out of Loop Total is:
      20
Total is: 25
Out of Loop Total is:
      30
Total is: 36
Total is: 43
Total is: 51
Total is: 60
Total is: 70
```

This is a 11*g* only code. Continue statement was introduced only in 11*g*.

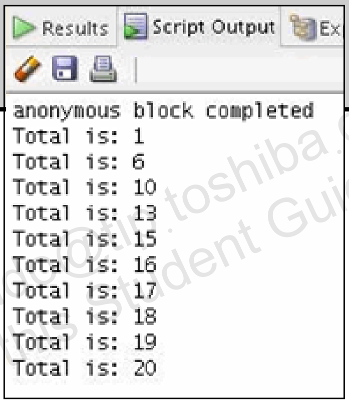
In the example, there are two assignments using the `v_total` variable:

1. The first assignment is executed for each of the 10 iterations of the loop.
2 The second assignment is executed for the first five iterations of the loop. The `CONTINUE` statement transfers control within a loop back to a new iteration, so for the last five iterations of the loop, the second `TOTAL` assignment is not executed.

The end result of the `TOTAL` variable is 70.

PL/SQL CONTINUE Statement: Example 2

```
DECLARE
 v_total NUMBER := 0;
BEGIN
 <<BeforeTopLoop>>
 FOR i IN 1..10 LOOP
   v_total := v_total + 1;
   dbms_output.put_line
     ('Total is: ' || v_total);
   FOR j IN 1..10 LOOP
     CONTINUE BeforeTopLoop WHEN i + j > 5;
     v_total := v_total + 1;
   END LOOP;
 END LOOP;
END two_loop;
```



```
anonymous block completed
Total is: 1
Total is: 6
Total is: 10
Total is: 13
Total is: 15
Total is: 16
Total is: 17
Total is: 18
Total is: 19
Total is: 20
```

You can use the CONTINUE statement to jump to the next iteration of an outer loop. This is a 11*g* only code.

To do this, provide the outer loop a label to identify where the CONTINUE statement should go.

The CONTINUE statement in the innermost loop terminates that loop whenever the WHEN condition is true (just like the EXIT keyword). After the innermost loop is terminated by the CONTINUE statement, control transfers to the next iteration of the outermost loop labeled BeforeTopLoop in this example.

When this pair of loops completes, the value of the TOTAL variable is 20.

You can also use the CONTINUE statement within an inner block of code, which does not contain a loop as long as the block is nested inside an appropriate outer loop.

Restrictions

- The CONTINUE statement cannot appear outside a loop at all—this generates a compiler error.
- You cannot use the CONTINUE statement to pass through a procedure, function, or method boundary—this generates a compiler error.

Quiz

There are three types of loops: basic, `FOR`, and `WHILE`.

a. True

b. False

Answer: a

Loop Types

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 count
- `WHILE` loops that perform iterative actions based on a condition

Summary

In this lesson, you should have learned to change the logical flow of statements by using the following control structures:

- Conditional (`IF` statement)
- `CASE` expressions and `CASE` statements
- Loops:
  - Basic loop
  - `FOR` loop
  - `WHILE` loop
- `EXIT` statement
- `CONTINUE` statement

A language can be called a programming language only if it provides control structures for the implementation of business logic. These control structures are also used to control the flow of the program. PL/SQL is a programming language that integrates programming constructs with SQL.

A conditional control construct checks for the validity of a condition and performs an action accordingly. You use the `IF` construct to perform a conditional execution of statements.

An iterative control construct executes a sequence of statements repeatedly, as long as a specified condition holds `TRUE`. You use the various loop constructs to perform iterative operations.

Practice 6: Overview

This practice covers the following topics:

- Performing conditional actions by using `IF` statements
- Performing iterative steps by using `LOOP` structures

In this practice, you create the PL/SQL blocks that incorporate loops and conditional control structures. The exercises test your understanding of writing various `IF` statements and `LOOP` constructs.