Oracle PL/SQL by Example

SIXTH EDITION

ORACLE

Elena Rakhimov

Oracle PL/SQL by Example

Sixth Edition

Using the CONTINUE WHEN Statement

The CONTINUE WHEN statement causes a loop to terminate its current iteration and pass control to the next iteration of the loop only when the continue condition evaluates to TRUE. Control then passes to the first executable statement inside the body of the loop. The structure of a loop using a CONTINUE WHEN clause is shown in Listing 7.2.

Listing 7.2 *Simple Loop Structure with a CONTINUE WHEN Statement*

```
LOOP
   STATEMENT 1;
   STATEMENT 2;
   CONTINUE WHEN CONTINUE CONDITION;

   EXIT WHEN EXIT CONDITION;
END LOOP;
STATEMENT 3;
```

Note that the flow of the logic illustrated in Figure 7.1 applies to the CONTINUE WHEN statement as well. In other words,

```
IF CONDITION
THEN
   CONTINUE;
END IF;
```

is equivalent to

```
CONTINUE WHEN CONDITION;
```

This similarity is illustrated further by the modified version of the previous example. When executed, this version produces output much like that of the previous version (affected statements are shown in bold).

For Example *ch07_1b.sql*

```
DECLARE
   v_counter BINARY_INTEGER := 0;
BEGIN
   LOOP
      -- increment loop counter by one
      v_counter := v_counter + 1;
      DBMS_OUTPUT.PUT_LINE
         ('Before continue condition, v_counter = '||v_counter);

      -- if continue condition yields TRUE pass control to the
      -- first executable statement of the loop
      CONTINUE WHEN v_counter < 3;

      DBMS_OUTPUT.PUT_LINE
         ('After continue condition, v_counter = '||v_counter);
```

```
      -- if exit condition yields TRUE exit the loop
      IF v_counter = 5
      THEN
         EXIT;
      END IF;

   END LOOP;
   -- control resumes here
   DBMS_OUTPUT.PUT_LINE ('Done…');
END;
```

Watch Out!

The CONTINUE and CONTINUE WHEN statements are valid only when placed inside a loop. When placed outside a loop, they will cause a syntax error.

When you are working with the exit and continue conditions, the execution of a loop and the number of iterations are affected *by the placement of those conditions inside the body of the loop*. This is illustrated further by the following example (changes are highlighted in bold).

For Example *ch07_1c.sql*

```
DECLARE
   v_counter BINARY_INTEGER := 0;
BEGIN
   LOOP
      -- increment loop counter by one
      v_counter := v_counter + 1;
      DBMS_OUTPUT.PUT_LINE
         ('Before continue condition, v_counter = '||v_counter);

      -- if continue condition yields TRUE pass control to the
      -- first executable statement of the loop
      CONTINUE WHEN v_counter > 3;

      DBMS_OUTPUT.PUT_LINE
         ('After continue condition, v_counter = '||v_counter);

      -- if exit condition yields TRUE exit the loop
      IF v_counter = 5
      THEN
         EXIT;
      END IF;

   END LOOP;
   -- control resumes here
   DBMS_OUTPUT.PUT_LINE ('Done…');
END;
```

In this version of the script, the continue condition has been changed to

```
CONTINUE WHEN v_counter > 3;
```

This change leads to an infinite loop. As long as the value of v_counter is less than or equal to 3, the continue condition evaluates to FALSE. Therefore, for

the first three iterations of the loop, all statements inside the body of the loop are executed along with the exit condition, which evaluates to FALSE.

Starting with the fourth iteration of the loop, the continue condition evaluates to TRUE, causing partial execution of the loop. Due to this partial execution, the exit condition cannot be reached, causing this loop to become infinite. To mitigate this situation, you should change the placement of the continue and exit conditions as shown in the following example (changes are shown in bold).

For Example *ch07_1d.sql*

```
DECLARE
   v_counter BINARY_INTEGER := 0;
BEGIN
   LOOP
      -- increment loop counter by one
      v_counter := v_counter + 1;
      -- if exit condition yields TRUE exit the loop
      IF v_counter = 5
      THEN
         EXIT;
      END IF;

      DBMS_OUTPUT.PUT_LINE
      ('Before continue condition, v_counter = '||v_counter);

      -- if continue condition yields TRUE pass control to the
      -- first executable statement of the loop
      CONTINUE WHEN v_counter > 3;

      DBMS_OUTPUT.PUT_LINE
         ('After continue condition, v_counter = '||v_counter);
   END LOOP;
   -- control resumes here
   DBMS_OUTPUT.PUT_LINE ('Done…');
END;
```

In this version of the script, the exit condition appears before the continue condition. Such placement of the exit condition guarantees eventual termination of the loop, as illustrated by the following output:

```
Before continue condition, v_counter = 1
After continue condition, v_counter = 1
Before continue condition, v_counter = 2
After continue condition, v_counter = 2
Before continue condition, v_counter = 3
After continue condition, v_counter = 3
Before continue condition, v_counter = 4
Done…
```

Here, on the fifth iteration of the loop, the value of v_counter is incremented by 1, and the exit condition evaluates to TRUE. As a result, none of the DBMS_OUTPUT.PUT_LINE statements inside the body of the loop are executed; instead, control of the execution passes to the first executable statement after the END LOOP, and Done. . . is displayed on the screen.

Lab 7.2: Nested Loops

After this lab, you will be able to

- Use Nested Loops
- Use Loop Labels

Using Nested Loops

You have explored three types of loops: simple loops, WHILE loops, and numeric FOR loops. Any of these three types of loops can be nested inside one another. For example, a simple loop can be nested inside a WHILE loop, and vice versa. Consider the following example.

For Example *ch07_2a.sql*

```
DECLARE
   v_counter1 BINARY_INTEGER := 0;
   v_counter2 BINARY_INTEGER;
BEGIN
   WHILE v_counter1 < 3
   LOOP
      DBMS_OUTPUT.PUT_LINE ('v_counter1: '||v_counter1);
      v_counter2 := 0;
      LOOP
         DBMS_OUTPUT.PUT_LINE (' v_counter2: '||v_counter2);
         v_counter2 := v_counter2 + 1;
         EXIT WHEN v_counter2 >= 2;
      END LOOP;
      v_counter1 := v_counter1 + 1;
   END LOOP;
END;
```

In this example, the WHILE loop is called an *outer loop* because it encompasses the simple loop. The simple loop (highlighted in bold) is called an *inner loop* because it is enclosed by the body of the WHILE loop.

The outer loop is controlled by the loop counter, v_counter1, and it will execute, providing the value of v_counter1 is less than 3. With each iteration of the loop, the value of v_counter1 is displayed on the screen. Next, the value of v_counter2 is initialized to 0. Note that v_counter2 is not initialized at the time of the declaration. The simple loop is placed inside the body of the WHILE loop, so the value of v_counter2 must be initialized every time before control passes to the simple loop.

After control passes to the inner loop, the value of v_counter2 is displayed on the screen and incremented by 1. Next, the exit when condition is evaluated. If this condition evaluates to FALSE, control passes back to the top of the simple loop. If it evaluates to TRUE, control passes to the first executable statement outside the

loop. In this case, control passes back to the outer loop, the value of v_counter1 is incremented by 1, and the test condition of the WHILE loop is evaluated again.

This logic is demonstrated by the output produced by the example:

```
v_counter1: 0
 v_counter2: 0
 v_counter2: 1
v_counter1: 1
 v_counter2: 0
 v_counter2: 1
v_counter1: 2
 v_counter2: 0
 v_counter2: 1
```

Notice that for each value of v_counter1, two values of v_counter2 are displayed. For the first iteration of the outer loop, the value of v_counter1 is equal to 0. After control passes to the inner loop, the value of v_counter2 is displayed on the screen twice, and so forth.

Using Loop Labels

In Chapter 2, "PL/SQL Language Fundamentals," you learned about labeling of PL/SQL blocks. Loops can be labeled in a similar manner, as illustrated in Listing 7.3.

Listing 7.3 *Loop Labels*

```
<<label_name>>
FOR loop_counter IN lower_limit..upper_limit
LOOP
   STATEMENT 1;
   …
   STATEMENT N;
END LOOP label_name;
```

The label must appear immediately before the beginning of the loop. The preceding syntax shows that the label can be optionally used at the end of the loop statement. It is helpful to label nested loops as labels to improve the script's readability. Consider the following example.

For Example *ch07_3a.sql*

```
BEGIN
   <<outer _loop>>
   FOR i IN 1..3
   LOOP
      DBMS_OUTPUT.PUT_LINE ('i = '||i);
      <<inner_loop>>
      FOR j IN 1..2
      LOOP
         DBMS_OUTPUT.PUT_LINE ('j = '||j);
      END LOOP inner_loop;
   END LOOP outer_loop;
END;
```

For both outer and inner loops, the statement `END LOOP` must be used. If the loop label is added to each `END LOOP` statement, it becomes easier to understand which loop is being terminated.

Loop labels can also be used when referencing loop counters, as shown in the following example.

For Example *ch07_4a.sql*

```
BEGIN
   <<outer>>
   FOR i IN 1..3
   LOOP
      <<inner>>
      FOR i IN 1..2
      LOOP
         DBMS_OUTPUT.PUT_LINE ('outer.i '||outer.i);
         DBMS_OUTPUT.PUT_LINE ('inner.i '||inner.i);
      END LOOP inner;
   END LOOP outer;
END;
```

In this example, both the inner and outer loops use the same loop counter, `i`. To reference both the outer and inner values of `i`, loop labels are used. This example produces the following output:

```
outer.i 1
inner.i 1
outer.i 1
inner.i 2
outer.i 2
inner.i 1
outer.i 2
inner.i 2
outer.i 3
inner.i 1
outer.i 3
inner.i 2
```

Note that the script is able to differentiate between two variables having the same name because loop labels are used when the variables are referenced. If no loop labels are used when `i` is referenced, the output produced by this script will change significantly. Basically, after control passes to the inner loop, the value of `i` from the outer loop is unavailable. When control passes back to the outer loop, the value of `i` becomes available again, as shown in the following example (affected statements are shown in bold). Recall that you have seen similar behavior in Chapter 6 where the same variable was used in the PL/SQL block and as an index variable of the loop.