

ORACLE

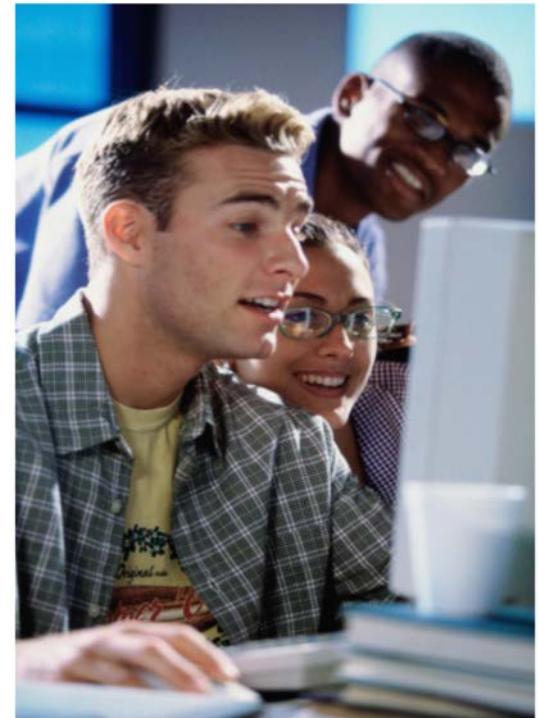
Academy

Database Programming with PL/SQL

4-2

Conditional Control: Case Statements

ORACLE
Academy



Objectives

- This lesson covers the following objectives:
 - 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

Purpose

- 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 work like functions to return one value from a number of values into a variable

Using a CASE Statement

- Look at this IF statement. What do you notice?
- All the conditions test the same variable v_numvar
- And the coding is very repetitive: v_numvar is coded many times

```
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;
```



Using a CASE Statement

- Here is the same logic, but using a CASE statement
- It is much easier to read. `v_numvar` is written only once

```
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;
```

CASE Statements: An Example

- A simple example to demonstrate the CASE logic

```
DECLARE
    v_num    NUMBER := 15;
    v_txt    VARCHAR2(50);
BEGIN
    CASE v_num
        WHEN 20 THEN v_txt := 'number equals 20';
        WHEN 17 THEN v_txt := 'number equals 17';
        WHEN 15 THEN v_txt := 'number equals 15';
        WHEN 13 THEN v_txt := 'number equals 13';
        WHEN 10 THEN v_txt := 'number equals 10';
        ELSE v_txt := 'some other number';
    END CASE;
    DBMS_OUTPUT.PUT_LINE(v_txt);
END;
```

Searched CASE Statements

- You can use CASE statements to test for non-equality conditions such as <, >, >=, etc.
- These are called searched CASE statements
- The syntax is virtually identical to an equivalent IF statement

```
DECLARE
  v_num    NUMBER := 15;
  v_txt     VARCHAR2(50);
BEGIN
  CASE
    WHEN v_num > 20 THEN v_txt := 'greater than 20';
    WHEN v_num > 15 THEN v_txt := 'greater than 15';
    ELSE v_txt := 'less than 16';
  END CASE;
  DBMS_OUTPUT.PUT_LINE(v_txt);
END;
```

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
- Again, the coding is very repetitive

```
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;
```

Using a CASE Expression

- 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;
```

CASE Expression Syntax

- A CASE expression is different from a CASE statement because it selects one of a number of results and assigns it to a variable
- A CASE expression ends with END not END CASE
- In the syntax, expressionN can be a literal value or an expression such as (v_other_var * 2)

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

CASE Expression Example

- What would be the result of this code if v_grade was initialized as "C" instead of "A"

```
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;
```

RESULT:
Grade: A
Appraisal: Excellent

Statement processed.

CASE Expression: A Second Example

- Determine what 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;
```

Searched CASE Expression Syntax

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

```
variable_name := 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

Searched CASE Expressions: An Example

- Searched CASE expressions allow non-equality conditions, compound conditions, and different variables to be used in different WHEN clauses

```
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;
```

How are CASE Expressions Different From CASE Statements?

- They are different because:
 - 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;
```

How are CASE Expressions Different From CASE Statements?

- 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;
```

Logic Tables

- When using IF and CASE statements you often need to combine conditions using AND, OR, and NOT
- The following Logic Table displays the results of all possible combinations of two conditions
- Example: TRUE and FALSE is FALSE

AND	TRUE	FALSE	NULL	OR	TRUE	FALSE	NULL	NOT	
TRUE	TRUE	Ex. 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

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

Terminology

- Key terms used in this lesson included:
 - CASE expression
 - CASE statement
 - Logic tables

Summary

- In this lesson, you should have learned how 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

ORACLE

Academy

ORACLE

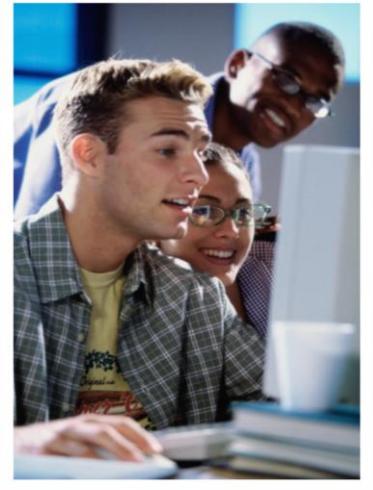
Academy

Database Programming with PL/SQL

4-2

Conditional Control: Case Statements

ORACLE
Academy



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Objectives

- This lesson covers the following objectives:
 - 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

Purpose

- 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 work like functions to return one value from a number of values into a variable

Using a CASE Statement

- Look at this IF statement. What do you notice?
- All the conditions test the same variable v_numvar
- And the coding is very repetitive: v_numvar is coded many times

```
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;
```



Academy

PLSQL 4-2
Conditional Control: Case Statements

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

5

Using a CASE Statement

- Here is the same logic, but using a CASE statement
- It is much easier to read. v_numvar is written only once

```
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;
```

ORACLE

Academy

PLSQL 4-2
Conditional Control: Case Statements

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

6

The WHEN clauses, “WHEN 5,” “WHEN 10,” “WHEN 12,” etc. mean: WHEN v_numvar is equal to 5, and WHEN v_numvar is equal to 10, etc.

CASE Statements: An Example

- A simple example to demonstrate the CASE logic

```
DECLARE
    v_num    NUMBER := 15;
    v_txt    VARCHAR2(50);
BEGIN
    CASE v_num
        WHEN 20 THEN v_txt := 'number equals 20';
        WHEN 17 THEN v_txt := 'number equals 17';
        WHEN 15 THEN v_txt := 'number equals 15';
        WHEN 13 THEN v_txt := 'number equals 13';
        WHEN 10 THEN v_txt := 'number equals 10';
        ELSE v_txt := 'some other number';
    END CASE;
    DBMS_OUTPUT.PUT_LINE(v_txt);
END;
```



Academy

PLSQL 4-2
Conditional Control: Case Statements

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

7

Searched CASE Statements

- You can use CASE statements to test for non-equality conditions such as <, >, >=, etc.
- These are called searched CASE statements
- The syntax is virtually identical to an equivalent IF statement

```
DECLARE
    v_num      NUMBER := 15;
    v_txt      VARCHAR2(50);
BEGIN
    CASE
        WHEN v_num > 20 THEN v_txt := 'greater than 20';
        WHEN v_num > 15 THEN v_txt := 'greater than 15';
        ELSE v_txt := 'less than 16';
    END CASE;
    DBMS_OUTPUT.PUT_LINE(v_txt);
END;
```

ORACLE

Academy

PLSQL 4-2
Conditional Control: Case Statements

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

8

The same logic from above, rewritten using an IF statement:

```
DECLARE
    v_num      NUMBER := 15;
    v_txt      VARCHAR2(50);
BEGIN
    IF v_num > 20 THEN v_txt := 'greater than 20';
    ELSIF v_num > 15 THEN v_txt := 'greater than 15';
    ELSE v_txt := 'less than 16';
    END IF;
    DBMS_OUTPUT.PUT_LINE(v_txt);
END;
```

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
- Again, the coding is very repetitive

```
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;
```

Using a CASE Expression

- 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;
```

CASE Expression Syntax

- A CASE expression is different from a CASE statement because it selects one of a number of results and assigns it to a variable
- A CASE expression ends with END not END CASE
- In the syntax, expressionN can be a literal value or an expression such as (v_other_var * 2)

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

ORACLE

Academy

PLSQL 4-2
Conditional Control: Case Statements

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

11

CASE Expression Example

- What would be the result of this code if v_grade was initialized as "C" instead of "A"

```
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;
```

RESULT:
Grade: A
Appraisal: Excellent

Statement processed.

ORACLE
Academy

PLSQL 4-2
Conditional Control: Case Statements

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

12

Answer:

Grade: C

Appraisal: Good

Statement processed.

CASE Expression: A Second Example

- Determine what 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;
```



PLSQL 4-2
Conditional Control: Case Statements

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

13

Answer: "Same value" will be displayed. Remember, read the highlighted WHEN clause as, "When v_in_var is equal to v_in_var, then assign "Same value" to v_out_var and skip to the line following the CASE expression end." Since v_in_var is equal to itself, the code will display the phrase, "Same value."

Searched CASE Expression Syntax

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

```
variable_name := 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



Searched CASE expressions are more flexible, allowing non-equality conditions (and compound conditions) to be tested and different variables to be used in different WHEN clauses.

Searched CASE Expressions: An Example

- Searched CASE expressions allow non-equality conditions, compound conditions, and different variables to be used in different WHEN clauses

```
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;
```

ORACLE

Academy

PLSQL 4-2
Conditional Control: Case Statements

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

15

We could have used a non-searched CASE expression. The example in the slide could be written as:

```
v_appraisal :=
CASE v_grade
    WHEN 'A' THEN 'Excellent'
    WHEN 'B' THEN 'Good'
    WHEN 'C' THEN 'Good'
    ELSE 'No such grade'
END;
```

How are CASE Expressions Different From CASE Statements?

- They are different because:
 - 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;
```

ORACLE

Academy

PLSQL 4-2
Conditional Control: Case Statements

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

16

CASE expressions are actually functions, which always return exactly one value.

A common, built-in function is SYSDATE. Just like a CASE expression, SYSDATE assigns one value to a variable.

```
v_date := SYSDATE;
```

How are CASE Expressions Different From CASE Statements?

- 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;
```

ORACLE

Academy

PLSQL 4-2
Conditional Control: Case Statements

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

17

A CASE expression has only one terminating semicolon at the END;, while a CASE statement has a semicolon after each executable statement in a WHEN clause.

Logic Tables

- When using IF and CASE statements you often need to combine conditions using AND, OR, and NOT
- The following Logic Table displays the results of all possible combinations of two conditions
- Example: TRUE and FALSE is FALSE

AND	TRUE	FALSE	NULL	OR	TRUE	FALSE	NULL	NOT	
TRUE	TRUE	Ex. 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

ORACLE

Academy

PLSQL 4-2
Conditional Control: Case Statements

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

18

You can build a simple Boolean condition by combining number, character, or 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.

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

Boolean Conditions

- What is the value of v_flag in each case?

```
v_flag := v_reordered_flag AND v_available_flag;
```

V_reordered_flag	V_available_flag	V_flag
TRUE	TRUE	1. _____
TRUE	FALSE	2. _____
NULL	TRUE	3. _____
NULL	FALSE	4. _____

The AND Logic Table on the previous slide can help you evaluate the possibilities for the Boolean condition in this slide.

Answers:

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

Terminology

- Key terms used in this lesson included:
 - CASE expression
 - CASE statement
 - Logic tables

- CASE expression – An expression that selects a result and returns it into a variable.
- CASE statement – A block of code that performs actions based on conditional tests.
- Logic Tables – Shows the results of all possible combinations of two conditions.

Summary

- In this lesson, you should have learned how 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

ORACLE

Academy

Database Programming with PL/SQL

4-2: Conditional Control: Case Statements

Practice Activities

Vocabulary

Identify the vocabulary word for each definition below:

	An expression that selects a result and returns it into a variable.
	Shows the results of all possible combinations of two conditions.
	A block of code that performs actions based on conditional tests.

Try It / Solve It

1. Write a PL/SQL block:

- A. To find the number of airports from the countries table for a supplied country_name. Based on this number, display a customized message as follows:

# Airports	Message
0–100	There are 100 or fewer airports.
101–1,000	There are between 101 and 1,000 airports.
1001–1,0000	There are between 1,001 and 10,000 airports.
> 10,000	There are more than 10,000 airports.
No value in database	The number of airports is not available for this country.

Use a CASE statement to process your comparisons.

You can use the following code to get started:

```
DECLARE
  v_country_name countries.country_name%TYPE := '<country_name>';
  v_airports      countries.airports%TYPE;
BEGIN
  SELECT airports INTO v_airports
    FROM countries
   WHERE country_name = v_country_name;
CASE
  WHEN ...
  ...
END CASE;
END;
```

B. Test your code for the following countries and confirm the results.

	No value	< 101	101-1,000	1,001-10,000	> 10,000
Canada				X	
Japan			X		
Malaysia			X		
Mongolia		X			
Navassa Island	X				
Romania		X			
United States of America					X

2. Write a PL/SQL block:

- A. To find the amount of coastline for a supplied country name. Use the countries table. Based on the amount of coastline for the country, display a customized message as follows:

Length of Coastline	Message
0	no coastline
< 1,000	a small coastline
< 10,000	a mid-range coastline
All other values	a large coastline

Use a CASE expression.

Use the following code to get started:

```

DECLARE
    v_country_name      countries.country_name%TYPE := '<country name>';
    v_coastline          countries.coastline %TYPE;
    v_coastline_description  VARCHAR2(50);
BEGIN
    SELECT coastline INTO v_coastline
    FROM countries
    WHERE country_name = v_country_name;
    v_coastline_description :=
        CASE
            ...
        END;
    DBMS_OUTPUT.PUT_LINE('Country ' || v_country_name || ' has '
        || v_coastline_description);
END;
```

- B. Test your code for the following countries and confirm the results.

	No coastline	Small coastline	Mid-range coastline	Large coastline
Canada				X
Grenada		X		
Jamaica			X	
Japan				X
Mongolia	X			
Ukraine			X	

3. Use a CASE statement:

A. Write a PL/SQL block to select the number of countries using a supplied currency name. If the number of countries is greater than 20, display “More than 20 countries”. If the number of countries is between 10 and 20, display “Between 10 and 20 countries”. If the number of countries is less than 10, display “Fewer than 10 countries”. Use a CASE statement.

B. Test your code using the following data:

	Fewer than 10 countries	Between 10 and 20 countries	More than 20 countries
US Dollar		X	
Swiss franc	X		
Euro			X

4. Examine the following code.

A. What do you think the output will be? Test your prediction by running the code.

```
DECLARE
  x BOOLEAN := FALSE;
  y BOOLEAN;
  v_color VARCHAR(20) := 'Red';
BEGIN
  IF (x OR y)
    THEN v_color := 'White';
  ELSE
    v_color := 'Black';
  END IF;
  DBMS_OUTPUT.PUT_LINE(v_color);
END;
```

B. Change the declarations to x and y as follows. What do you think the output will be? Test your prediction by running the code again.

```
x BOOLEAN ;
y BOOLEAN ;
```

C. Change the declarations to x and y as follows. What do you think the output will be? Test your prediction by running the code again.

```
x BOOLEAN := TRUE;
y BOOLEAN := TRUE;
```

D. Experiment with changing the OR condition to AND.

ORACLE

Academy

Database Programming with PL/SQL

4-1

Conditional Control: IF Statements

ORACLE
Academy



Objectives

- This lesson covers the following objectives:
 - 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 PL/SQL to handle the null condition in IF statements

Purpose

- 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

Purpose

- Think of a logic test as something you do every day
- If you get up in the morning and it is cold outside, you will choose to wear cold-weather clothing
- If you get up in the morning and it is warm outside, you will choose to wear warm-weather clothing
- And if there is a chance of rain, then you will bring a rain coat or an umbrella with you

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





IF Statement

- The IF statement shown below using "pseudocode" 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

```
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"
```

A diagram illustrating the flow of an IF statement. A red-bordered box labeled "Conditions" has three arrows pointing to the three separate "if" blocks in the pseudocode above.

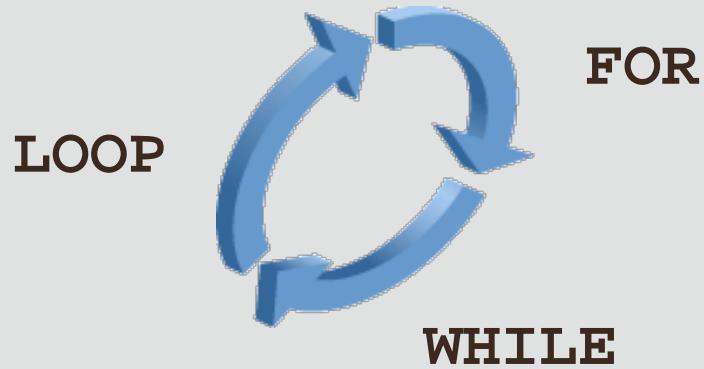
CASE Expressions

- 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 pseudocode 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" ...
```

LOOP Control Structures

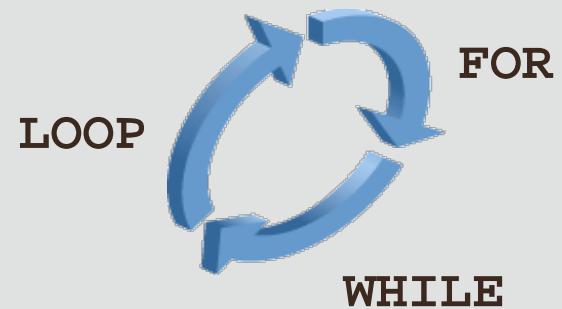
- Loop control structures are repetition statements that enable you to execute statements in a PLSQL block repeatedly
- Three types of loop control structures are supported by PL/SQL: BASIC, FOR, and WHILE



LOOP Control Structures

- Consider the following pseudocode example:
 - Print the 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.
```



IF Statements Structure

- 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;
```

IF Statements

- 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

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

IF Statements

- 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;
```

IF Statements

- ELSIF is a keyword that introduces an additional Boolean expression
- If the first condition yields FALSE or NULL, then the ELSIF keyword introduces additional conditions
- ELSIF is the correct spelling, not ELSEIF

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

IF Statements

- 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;
```

ORACLE

Academy

IF Statements 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;
```

Simple IF Statement

- This is an example of a simple IF statement with a THEN clause
- The v_myage variable is initialized to 31

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

Simple IF Statement

- 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;
```

IF THEN ELSE Statement

- The ELSE clause has been added to this example
- The condition has not changed, thus it still evaluates to FALSE

```
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;
```

IF THEN ELSE Statement

- Remember that the statements in the THEN clause are only executed if the condition returns TRUE
- In this case, the condition returns FALSE, so control passes 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;
```

IF ELSIF ELSE Clause

- The IF statement now contains multiple ELSIF clauses as well as an ELSE clause
- Notice that the ELSIF clauses add additional conditions

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 mature');
    END IF;
END;
```

IF ELSIF ELSE Clause

- As with the IF statement, each ELSIF condition is followed by a THEN clause
- This is executed only if the ELSIF condition returns TRUE

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 mature');
    END IF;
END;
```

IF ELSIF ELSE Clause

- When you have multiple clauses in the IF statement and a condition is FALSE or NULL, control then passes to the next clause
- Conditions are evaluated one by one
- If all conditions are FALSE or NULL, then the statements in the ELSE clause are executed

IF ELSIF ELSE Clause

```
...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;...
```

IF ELSIF ELSE Clause

- 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;...
```

IF Statement with Multiple Expressions

- An IF statement can have multiple conditional expressions related with logical operators, such as AND, OR, and NOT
- This example uses the AND operator
- Therefore, it evaluates to TRUE only if both BOTH the first name and age conditions are evaluated as TRUE

```
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;
```

IF Statement with Multiple Expressions

- There is no limitation on the number of conditional expressions that can be used
- However, these statements must be connected with the appropriate logical operators

```
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;
```

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, which is neither TRUE nor FALSE
- In such a case, the control goes to the ELSE statement because, just NULL is not TRUE

NULL Values in IF Statements

```
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;
```

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



Handling Nulls Example

- In this example, you might expect the sequence of statements to execute because a and b seem equal
- But, NULL is unknown, so we don't know if a and b are equal
- The IF condition yields NULL and the THEN clause is bypassed, with control going to the line following the THEN clause

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

Guidelines for Using IF Statements

- Follow these guidelines when 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



Guidelines for Using IF Statements

- 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



Terminology

- Key terms used in this lesson included:
 - CASE
 - Condition
 - IF
 - LOOP

Summary

- In this lesson, you should have learned how 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 PL/SQL to handle the null condition in IF statements

ORACLE

Academy

ORACLE

Academy

Database Programming with PL/SQL

4-1

Conditional Control: IF Statements

ORACLE
Academy



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Objectives

- This lesson covers the following objectives:
 - 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 PL/SQL to handle the null condition in IF statements

Purpose

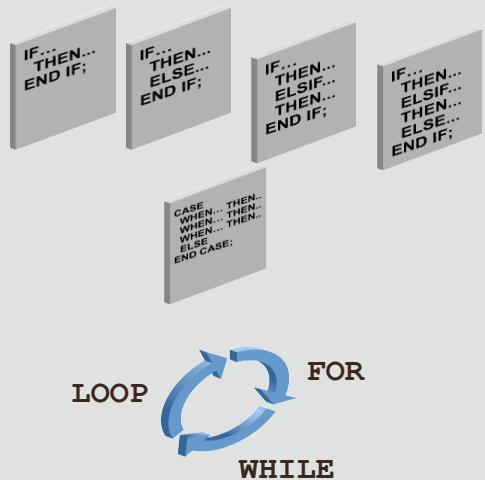
- In this section, you learn how to use the conditional logic in a PL/SQL block
- Conditional processing extends the usefulness of programs by allowing the use of simple logical tests to determine which statements are executed

Purpose

- Think of a logic test as something you do every day
- If you get up in the morning and it is cold outside, you will choose to wear cold-weather clothing
- If you get up in the morning and it is warm outside, you will choose to wear warm-weather clothing
- And if there is a chance of rain, then you will bring a rain coat or an umbrella with you

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 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.

The CASE statement contains alternative courses of action in a block based on one condition. They are different in that they can be used outside of a PLSQL block in a SQL statement.

LOOPS are control structures that allow iteration of statements. 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.

IF Statement

- The IF statement shown below using "pseudocode" 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

```
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

Writing in pseudocode combines natural language with code-like elements. It is sometimes used when introducing new structures and concepts. Pseudocode generally excludes syntax requirements and adds words that capture the logic of the structure/concept under consideration.

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 pseudocode 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" ...
```



Academy

PLSQL 4-1
Conditional Control: IF Statement

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

8

CASE expressions will be discussed more fully in a future lesson.

LOOP Control Structures

- Loop control structures are repetition statements that enable you to execute statements in a PLSQL block repeatedly
- Three types of loop control structures are supported by PL/SQL: BASIC, FOR, and WHILE



Loops will be discussed more fully in a future lesson.

LOOP Control Structures

- Consider the following pseudocode example:
 - Print the 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.
```



IF Statements Structure

- 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;
```



Academy

PLSQL 4-1
Conditional Control: IF Statement

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

11

Please note the ELSIF does not have an 'E' after the 'S,' and there is no space. There can be many ELSIF clauses within an IF statement. There can be only one ELSE clause within an IF statement (and it comes just before the END IF). Also, each THEN clause within the IF statement ends with a semicolon.

IF Statements

- 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

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



PLSQL 4-1
Conditional Control: IF Statement

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

12

IF Statements

- 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;
```

ORACLE

Academy

PLSQL 4-1
Conditional Control: IF Statement

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

13

IF Statements

- ELSIF is a keyword that introduces an additional Boolean expression
- If the first condition yields FALSE or NULL, then the ELSIF keyword introduces additional conditions
- ELSIF is the correct spelling, not ELSEIF

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



PLSQL 4-1
Conditional Control: IF Statement

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

14

IF Statements

- 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;
```

ORACLE

Academy

PLSQL 4-1
Conditional Control: IF Statement

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

15

IF Statements 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;
```

Simple IF Statement

- This is an example of a simple IF statement with a THEN clause
- The v_myage variable is initialized to 31

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



PLSQL 4-1
Conditional Control: IF Statement

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

17

Simple IF Statement

- 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;
```

IF THEN ELSE Statement

- The ELSE clause has been added to this example
- The condition has not changed, thus it still evaluates to FALSE

```
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;
```



PLSQL 4-1
Conditional Control: IF Statement

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

19

IF THEN ELSE Statement

- Remember that the statements in the THEN clause are only executed if the condition returns TRUE
- In this case, the condition returns FALSE, so control passes 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;
```

ORACLE
Academy

PLSQL 4-1
Conditional Control: IF Statement

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

20

IF ELSIF ELSE Clause

- The IF statement now contains multiple ELSIF clauses as well as an ELSE clause
- Notice that the ELSIF clauses add additional conditions

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 mature');
  END IF;
END;
```

ORACLE

Academy

PLSQL 4-1
Conditional Control: IF Statement

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

22

IF ELSIF ELSE Clause

- As with the IF statement, each ELSIF condition is followed by a THEN clause
- This is executed only if the ELSIF condition returns TRUE

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 mature');
    END IF;
END;
```



Academy

PLSQL 4-1
Conditional Control: IF Statement

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

24

IF ELSIF ELSE Clause

- When you have multiple clauses in the IF statement and a condition is FALSE or NULL, control then passes to the next clause
- Conditions are evaluated one by one
- If all conditions are FALSE or NULL, then the statements in the ELSE clause are executed

The IF clause now contains multiple ELSIF clauses and an ELSE clause. Observe that the ELSIF clauses have conditions, but the ELSE clause has no condition. The condition for ELSIF should be followed by the THEN clause that is executed if the condition of the ELSIF clause returns TRUE. When you have multiple ELSIF clauses, if the first condition is FALSE or NULL, the control shifts to the next ELSIF clause.

IF ELSIF ELSE Clause

```
...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;...
```

IF ELSIF ELSE Clause

- 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;...
```

IF Statement with Multiple Expressions

- An IF statement can have multiple conditional expressions related with logical operators, such as AND, OR, and NOT
- This example uses the AND operator
- Therefore, it evaluates to TRUE only if both BOTH the first name and age conditions are evaluated as TRUE

```
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;
```

ORACLE

Academy

PLSQL 4-1
Conditional Control: IF Statement

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

28

IF Statement with Multiple Expressions

- There is no limitation on the number of conditional expressions that can be used
- However, these statements must be connected with the appropriate logical operators

```
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;
```

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, which is neither TRUE nor FALSE
- In such a case, the control goes to the ELSE statement because, just NULL is not TRUE

NULL Values in IF Statements

```
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;
```



PLSQL 4-1
Conditional Control: IF Statement

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

31

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



Handling Nulls Example

- In this example, you might expect the sequence of statements to execute because a and b seem equal
- But, NULL is unknown, so we don't know if a and b are equal
- The IF condition yields NULL and the THEN clause is bypassed, with control going to the line following the THEN clause

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

ORACLE

Academy

PLSQL 4-1
Conditional Control: IF Statement

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

33

Guidelines for Using IF Statements

- Follow these guidelines when 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



Guidelines for Using IF Statements

- 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



Terminology

- Key terms used in this lesson included:
 - CASE
 - Condition
 - IF
 - LOOP

- CASE – An expression that determines a course of action based on conditions and can be used outside of a PL/SQL block in a SQL statement.
- Condition – An expression with a TRUE or FALSE value that is used to make a decision.
- IF – Statement that enables PL/SQL to perform actions selectively based on conditions.
- LOOP – Control structures- Repetition statements that enable you to execute statements in a PL/SQL block repeatedly.

Summary

- In this lesson, you should have learned how 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 PL/SQL to handle the null condition in IF statements

ORACLE

Academy

Database Programming with PL/SQL

4-1: Conditional Control: IF Statements

Practice Activities

Vocabulary

Identify the vocabulary word for each definition below:

	Statement that enables PL/SQL to perform actions selectively based on conditions.
	Control structures – Repetition statements that enable you to execute statements in a PL/SQL block repeatedly.
	An expression with a TRUE or FALSE value that is used to make a decision.
	An expression that determines a course of action based on conditions and can be used outside a PL/SQL block in a SQL statement.

Try It / Solve It

1. What is the purpose of a conditional control structure in PL/SQL?
2. List the three categories of control structures in PL/SQL.
3. List the keywords that can be part of an IF statement.
4. List the keywords that are a required part of an IF statement.
5. Write a PL/SQL block to find the population of a given country in the countries table. Display a message indicating whether the population is greater than or less than 1 billion (1,000,000,000). Test your block twice using India (country_id = 91) and United Kingdom (country_id = 44). India's population should be greater than 1 billion, while United Kingdom's should be less than 1 billion.

6. Modify the code from the previous exercise so that it handles all the following cases:

- A. Population is greater than 1 billion.
- B. Population is greater than 0.
- C. Population is 0.
- D. Population is null. (Display: No data for this country.)

Run your code using the following country ids. Confirm the indicated results.

- China (country_id = 86): Population is greater than 1 billion.
- United Kingdom (country_id = 44): Population is greater than 0.
- Antarctica (country_id = 672): Population is 0.
- Europa Island (country_id = 15): No data for this country.

7. Examine the following code:

```
DECLARE
  v_country_id  countries.country_name%TYPE := <a value>;
  v_ind_date    countries.date_of_independence%TYPE;
  v_natl_holiday countries.national_holiday_date%TYPE;
BEGIN
  SELECT date_of_independence, national_holiday_date
    INTO v_ind_date, v_natl_holiday
   FROM countries
  WHERE country_id = v_country_id;
  IF v_ind_date IS NOT NULL THEN
    DBMS_OUTPUT.PUT_LINE('A');
  ELSIF v_natl_holiday IS NOT NULL THEN
    DBMS_OUTPUT.PUT_LINE('B');
  ELSIF v_natl_holiday IS NULL AND v_ind_date IS NULL THEN
    DBMS_OUTPUT.PUT_LINE('C');
  END IF;
END;
```

- A. What would print if the country has an independence date equaling NULL and a national holiday date equaling NULL?
- B. What would print if the country has an independence date equaling NULL and a national holiday date containing a value?
- C. What would print if the country has an independence date equaling a value and a national holiday date equaling NULL?
- D. Run a SELECT statement against the COUNTRIES table to determine whether the following countries have independence dates or national holiday dates, or both. Predict the output of running the anonymous block found at the beginning of this question.

Country	Country_ID	Independence Date	National Holiday Date	Output should be
Antarctica	672			
Iraq	964			
Spain	34			
United States	1			

- E. Finally, run the anonymous block found at the beginning of this question using each of the above country ids as input. Check whether your output answers are correct.
8. Examine the following code. What output do you think it will produce?

```

DECLARE
    v_num1  NUMBER(3) := 123;
    v_num2  NUMBER;
BEGIN
    IF v_num1 <> v_num2 THEN
        DBMS_OUTPUT.PUT_LINE('The two numbers are not equal');
    ELSE
        DBMS_OUTPUT.PUT_LINE('The two numbers are equal');
    END IF;
END;

```

Run the code to check if your prediction was correct. What was the result and why? Modify the code to use various comparison operators to see different results.

9. Write a PL/SQL block to accept a year and check whether it is a leap year. For example, if the year entered is 1990, the output should be “1990 is not a leap year.”

Hint: A leap year should be exactly divisible by 4, but not exactly divisible by 100. However, any year exactly divisible by 400 is a leap year.

Test your solution with the following years:

Year	Result Should Be
1990	Not a leap year
2000	Leap year
1996	Leap year
1900	Not a leap year
2016	Leap year
1884	Leap year

ORACLE

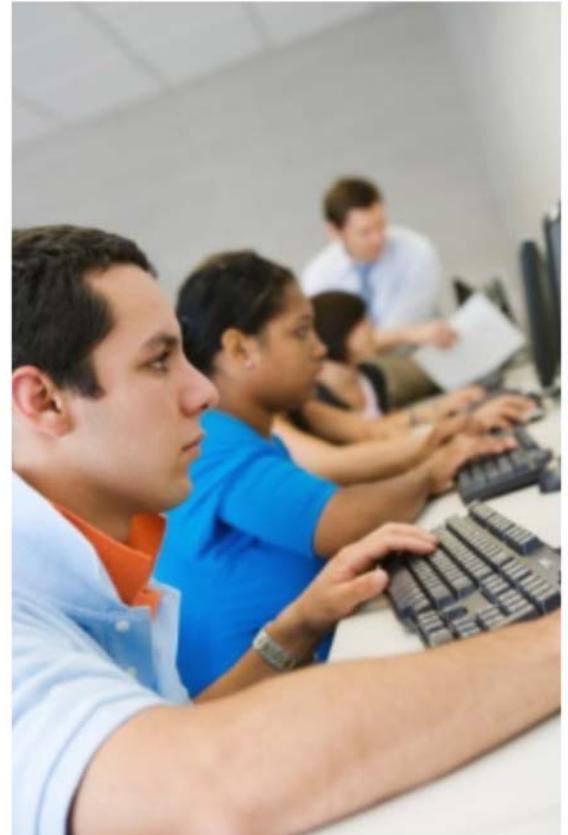
Academy

Database Programming with PL/SQL

1-2

Benefits of PL/SQL

ORACLE
Academy



Objectives

- This lesson covers the following objectives:
 - List and explain the benefits of PL/SQL
 - List the differences between PL/SQL and other programming languages
 - Give examples of how to use PL/SQL in other Oracle products

Purpose

- PL/SQL is a programming language suitable for several tasks involving an Oracle database
- In this lesson, you learn about the benefits of the PL/SQL programming language and how it compares to other programming languages
- You also learn how PL/SQL relates to other Oracle products

Benefits of PL/SQL

- There are many benefits to using the PL/SQL programming language with an Oracle database
- Integration of procedural constructs with SQL
- Modularized program development
- Improved performance
- Integration with Oracle tools
- Portability
- Exception handling

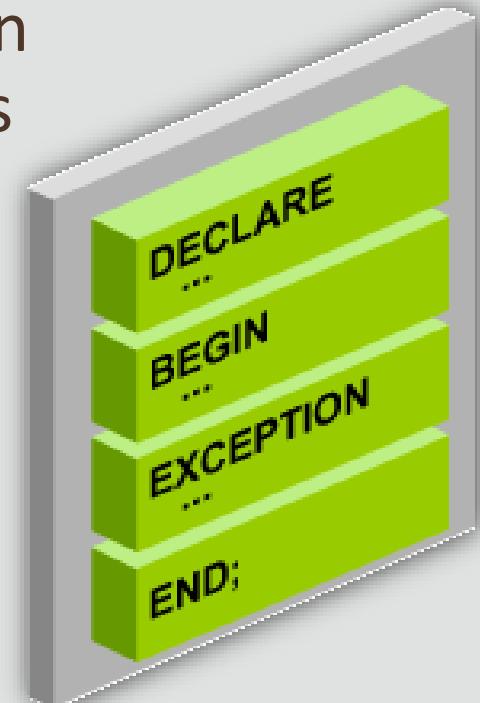


Benefit 1: Integration of Procedural Constructs With SQL

- The primary advantage of PL/SQL is the integration of procedural constructs with SQL
- SQL is a nonprocedural language
- When you issue an SQL command, your command tells the database server what to do
- However, you cannot specify how to do it or how often to do it
- PL/SQL integrates control statements and conditional statements with SQL
- This gives you better control of your SQL statements and their execution

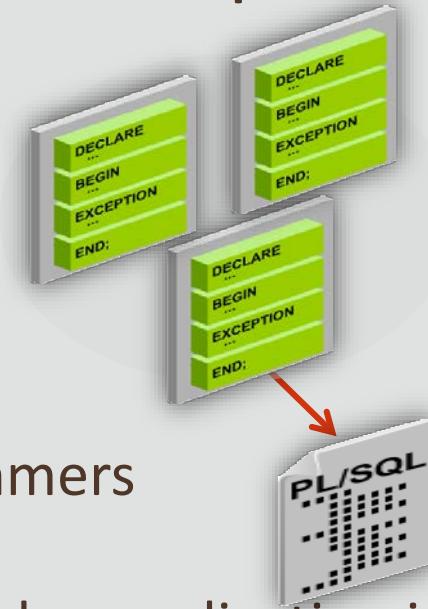
Benefit 2: Modularized Program Development

- The basic unit in a PL/SQL program is a block. All PL/SQL programs consist of blocks
- You can think of these blocks as modules and you can “modularize” these blocks in a sequence, or nest them in other blocks
- Good programming practice uses modular programs to break program control into sections that may be easier to understand and maintain



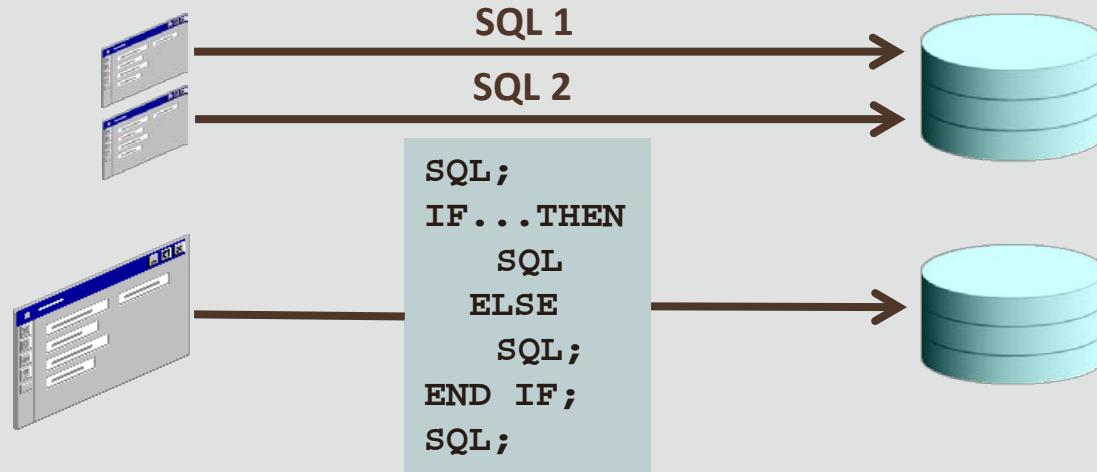
Benefit 2: Modularized Program Development

- Modularized program advantages:
 - You can group logically related statements within blocks
 - You can nest blocks inside other blocks to build powerful programs
 - You can share blocks with other programmers to speed up development time
 - PL/SQL allows you to break down a complex application into smaller, more manageable, and logically related modules increasing ease of maintenance and debugging



Benefit 3: Improved Performance

- PL/SQL allows you to logically combine multiple SQL statements as one unit or block
- The application can send the entire block to the database instead of sending the SQL statements one at a time, making the program process faster
- This significantly reduces the number of database calls

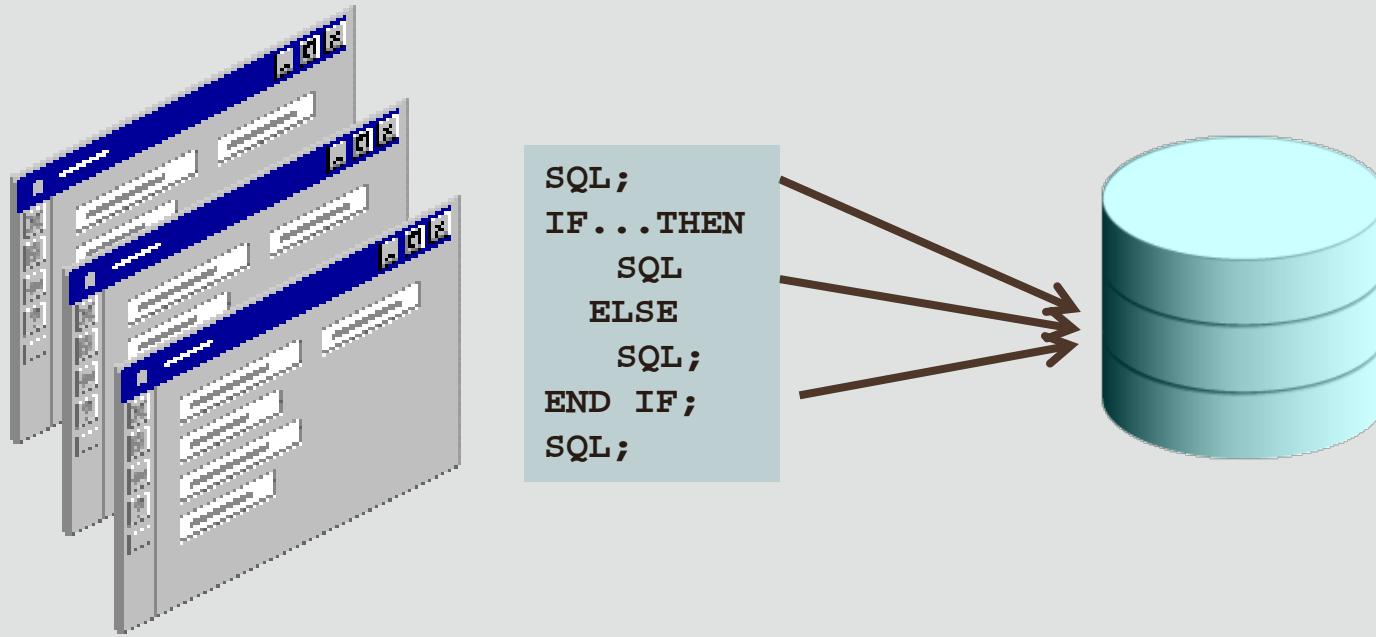


Benefit 3: Improved Performance

- The following features also result in improved performance:
 - PL/SQL variables store data in the same internal binary format as the database does, so no data conversion is needed
 - PL/SQL is executed in the same memory space as the Oracle server and therefore there is no communications overhead between the two programs
 - PL/SQL functions can be called directly from SQL
 - A special kind of PL/SQL procedure, called a trigger, can execute automatically whenever something important happens in the database

Benefit 4: Integration With Oracle Tools

- PL/SQL is integrated in Oracle tools, such as Oracle Forms Developer, Oracle Report Builder, and Application Express



Benefit 5: Portability

- PL/SQL programs can run anywhere an Oracle server runs, regardless of the operating system and the platform
- PL/SQL programs do not need to be tailored for different operating systems and platforms



Benefit 5: Portability

- You can write portable program packages and create libraries that can be reused on Oracle databases in different environments
- You can even anticipate those differences and establish instructions to run a specific way given a specific environment



Linux

HP Tru64



IBM z/OS

Solaris

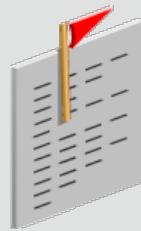


Benefit 6: Exception Handling

- An exception is an error that occurs when accessing a database
- Examples of exceptions include:
 - hardware or network failures
 - application logic errors
 - data integrity errors, and so on
- You can prepare for these errors by writing exception handling code
- Exception handling code tells your application what to do in the event of an exception

Benefit 6: Exception Handling

- PL/SQL allows you to handle database and program exceptions efficiently
- You can define separate blocks for dealing with exceptions



If no data is found then...

If too many rows are found then...

If an invalid number is calculated then...

- In this way, your application can handle the error, communicating the problem to the user, without causing a system crash

PL/SQL Compared to Other Languages

	PL/SQL	C	Java
Requires Oracle database or tool	Yes	No	No
Object-oriented	Some features	No	Yes
Performance against an Oracle database	Very efficient	Less efficient	Less efficient
Portable to different operating systems	Yes	Somewhat	Yes
Ease of learning	Relatively easy	More difficult	More difficult

PL/SQL in Oracle Products

Oracle Product	PL/SQL
	You can write PL/SQL code to manage application data or to manage the Oracle database itself. For example, you can write code for updating data (DML), creating data (DDL), generating reports, managing security, and so on.
	Using the Web Application Toolkit, you can create database-centric web applications written entirely or partially in PL/SQL.
	Using Forms Builder and Reports Developer, Oracle's client-side developer tools, you can build database-centric web applications and reports that include PL/SQL.
	Using a Web browser you can develop web applications that include PL/SQL.

Terminology

- Key terms used in this lesson included:
 - Blocks
 - Portability
 - Exceptions

Summary

- In this lesson, you should have learned how to:
 - List and explain the benefits of PL/SQL
 - List differences between PL/SQL and other programming languages
 - Give examples of how to use PL/SQL in other Oracle products

ORACLE

Academy

ORACLE

Academy

Database Programming with PL/SQL

1-2

Benefits of PL/SQL

ORACLE
Academy



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Objectives

- This lesson covers the following objectives:
 - List and explain the benefits of PL/SQL
 - List the differences between PL/SQL and other programming languages
 - Give examples of how to use PL/SQL in other Oracle products

Purpose

- PL/SQL is a programming language suitable for several tasks involving an Oracle database
- In this lesson, you learn about the benefits of the PL/SQL programming language and how it compares to other programming languages
- You also learn how PL/SQL relates to other Oracle products

Benefits of PL/SQL

- There are many benefits to using the PL/SQL programming language with an Oracle database
- Integration of procedural constructs with SQL
- Modularized program development
- Improved performance
- Integration with Oracle tools
- Portability
- Exception handling



Benefit 1: Integration of Procedural Constructs With SQL

- The primary advantage of PL/SQL is the integration of procedural constructs with SQL
- SQL is a nonprocedural language
- When you issue an SQL command, your command tells the database server what to do
- However, you cannot specify how to do it or how often to do it
- PL/SQL integrates control statements and conditional statements with SQL
- This gives you better control of your SQL statements and their execution

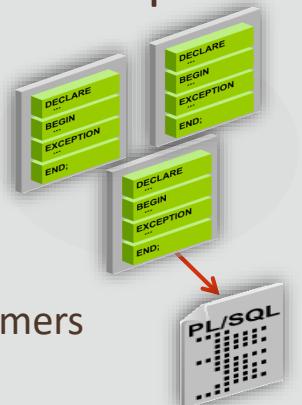
Benefit 2: Modularized Program Development

- The basic unit in a PL/SQL program is a block. All PL/SQL programs consist of blocks
- You can think of these blocks as modules and you can “modularize” these blocks in a sequence, or nest them in other blocks
- Good programming practice uses modular programs to break program control into sections that may be easier to understand and maintain



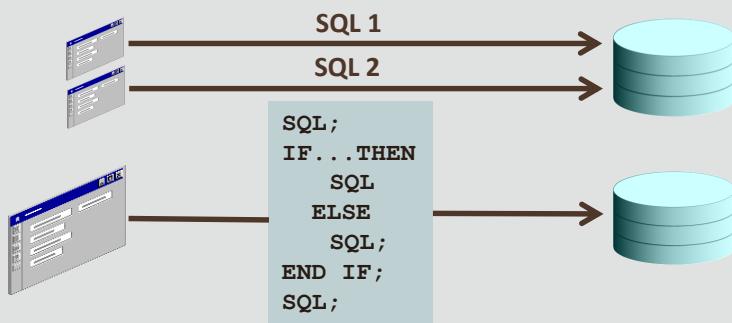
Benefit 2: Modularized Program Development

- Modularized program advantages:
 - You can group logically related statements within blocks
 - You can nest blocks inside other blocks to build powerful programs
 - You can share blocks with other programmers to speed up development time
 - PL/SQL allows you to break down a complex application into smaller, more manageable, and logically related modules increasing ease of maintenance and debugging



Benefit 3: Improved Performance

- PL/SQL allows you to logically combine multiple SQL statements as one unit or block
- The application can send the entire block to the database instead of sending the SQL statements one at a time, making the program process faster
- This significantly reduces the number of database calls

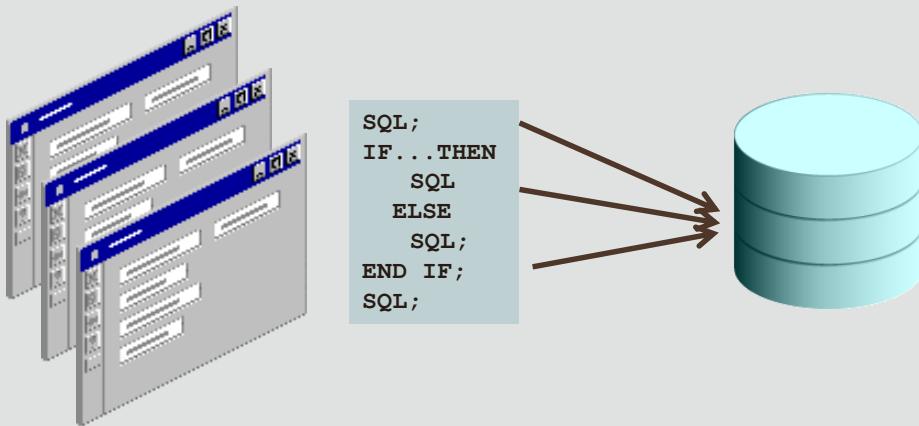


Benefit 3: Improved Performance

- The following features also result in improved performance:
 - PL/SQL variables store data in the same internal binary format as the database does, so no data conversion is needed
 - PL/SQL is executed in the same memory space as the Oracle server and therefore there is no communications overhead between the two programs
 - PL/SQL functions can be called directly from SQL
 - A special kind of PL/SQL procedure, called a trigger, can execute automatically whenever something important happens in the database

Benefit 4: Integration With Oracle Tools

- PL/SQL is integrated in Oracle tools, such as Oracle Forms Developer, Oracle Report Builder, and Application Express



Benefit 5: Portability

- PL/SQL programs can run anywhere an Oracle server runs, regardless of the operating system and the platform
- PL/SQL programs do not need to be tailored for different operating systems and platforms



Benefit 5: Portability

- You can write portable program packages and create libraries that can be reused on Oracle databases in different environments
- You can even anticipate those differences and establish instructions to run a specific way given a specific environment



Benefit 6: Exception Handling

- An exception is an error that occurs when accessing a database
- Examples of exceptions include:
 - hardware or network failures
 - application logic errors
 - data integrity errors, and so on
- You can prepare for these errors by writing exception handling code
- Exception handling code tells your application what to do in the event of an exception

Benefit 6: Exception Handling

- PL/SQL allows you to handle database and program exceptions efficiently
- You can define separate blocks for dealing with exceptions



If no data is found then...
If too many rows are found then...
If an invalid number is calculated then...

- In this way, your application can handle the error, communicating the problem to the user, without causing a system crash

PL/SQL Compared to Other Languages

	PL/SQL	C	Java
Requires Oracle database or tool	Yes	No	No
Object-oriented	Some features	No	Yes
Performance against an Oracle database	Very efficient	Less efficient	Less efficient
Portable to different operating systems	Yes	Somewhat	Yes
Ease of learning	Relatively easy	More difficult	More difficult

- PL/SQL requires an Oracle database or tool. You cannot create a PL/SQL program that runs all by itself. C and Java programs do not require an Oracle database to run or compile. You can develop standalone programs using Java and C.
- PL/SQL has included some object-oriented features such as abstract data types, multi-level collections, encapsulation, function overloading, and inheritance. Java is an object-oriented programming language and C is not.
- PL/SQL is tightly integrated with an Oracle database and is therefore highly efficient when accessing data. Java and C are less efficient because they are not as integrated with the database.
- PL/SQL is compatible with Oracle databases on different operating systems. Java also is highly portable. Different C compilers and libraries are not 100% compatible on different operating systems.
- PL/SQL is relatively easy to learn in relation to Java and C.

PL/SQL in Oracle Products

Oracle Product	PL/SQL
	You can write PL/SQL code to manage application data or to manage the Oracle database itself. For example, you can write code for updating data (DML), creating data (DDL), generating reports, managing security, and so on.
	Using the Web Application Toolkit, you can create database-centric web applications written entirely or partially in PL/SQL.
	Using Forms Builder and Reports Developer, Oracle's client-side developer tools, you can build database-centric web applications and reports that include PL/SQL.
ORACLE Application Express	Using a Web browser you can develop web applications that include PL/SQL.

Terminology

- Key terms used in this lesson included:
 - Blocks
 - Portability
 - Exceptions

- Blocks – The basic unit of PL/SQL programs; also known as modules.
- Portability – The ability for PL/SQL programs to run anywhere an Oracle server runs.
- Exceptions – An error that occurs in the database or in a user's program during runtime.

Summary

- In this lesson, you should have learned how to:
 - List and explain the benefits of PL/SQL
 - List differences between PL/SQL and other programming languages
 - Give examples of how to use PL/SQL in other Oracle products

ORACLE

Academy

Database Programming with PL/SQL

1-2: Benefits of PL/SQL

Practice Activities

Vocabulary

Identify the vocabulary word for each definition below:

	The ability for PL/SQL programs to run anywhere an Oracle server runs.
	The basic unit of PL/SQL programs- also known as modules.
	An error that occurs in the database or in a user's program during runtime.

Try It / Solve It

1. Why is it more efficient to combine SQL statements into PL/SQL blocks
2. Why is it beneficial to use PL/SQL with an Oracle database? List at least three reasons.
3. How is PL/SQL different from C and Java? List three differences.
4. List three examples of what you can build with PL/SQL code.

ORACLE

Academy

Database Programming with PL/SQL

1-3

Creating PL/SQL Blocks

ORACLE
Academy



Objectives

- This lesson covers the following objectives:
 - Describe the structure of a PL/SQL block
 - Identify the different types of PL/SQL blocks
 - Identify PL/SQL programming environments
 - Create and execute an anonymous PL/SQL block
 - Output messages in PL/SQL

Purpose

- Here you will learn the structure of a PL/SQL block and create one kind of block: an anonymous block
- Later in the course, you will learn to create Procedures, Functions, and Packages using the basic structure found in anonymous blocks
- After learning about the different environments into which you can develop your PL/SQL programs, you will also begin coding PL/SQL in the Application Express development environment

PL/SQL Block Structure

- A PL/SQL block consists of three sections

Section	Description
Declarative (optional)	The declarative section begins with the keyword DECLARE and ends when your executable section starts.
Executable (mandatory)	The executable section begins with the keyword BEGIN and ends with END. Observe that END is terminated with a semicolon. The executable section of a PL/SQL block can include any number of nested PL/SQL blocks.
Exception handling (optional)	The exception section is nested within the executable section. This section begins with the keyword EXCEPTION.

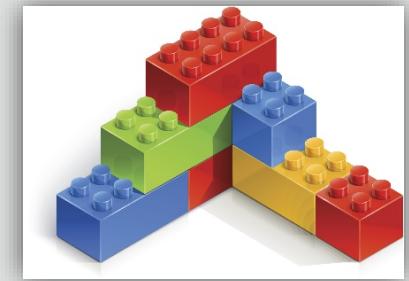


PL/SQL Block Structure Sections

Section	Description	Inclusion
Declarative (DECLARE)	Contains declarations of all variables, constants, cursors, and user-defined exceptions that are referenced in the executable and exception sections.	Optional
Executable (BEGIN ... END;)	Contains SQL statements to retrieve data from the database and PL/SQL statements to manipulate data in the block. Must contain at least one statement.	Mandatory
Exception (EXCEPTION)	Specifies the actions to perform when errors and abnormal conditions arise in the executable section.	Optional

Anonymous Blocks

- Characteristics of anonymous blocks:
 - Unnamed block
 - Not stored in the database
 - Declared inline at the point in an application where it is executed
 - Compiled each time the application is executed
 - Passed to the PL/SQL engine for execution at run time
 - Cannot be invoked or called because it does not have a name and does not exist after it is executed



Anonymous Blocks – Basic Structure

- Basic structure of an anonymous block:

```
[ DECLARE ]  
  
BEGIN  
  --statements  
  
[ EXCEPTION ]  
  
END;
```

- The DECLARE and EXCEPTION keywords/sections are optional

Examples of Anonymous Blocks

- Executable section only (minimum required)

```
BEGIN
    DBMS_OUTPUT.PUT_LINE('PL/SQL is easy!');
END;
```

- Declarative and Executable sections

```
DECLARE
    v_date      DATE := SYSDATE;
BEGIN
    DBMS_OUTPUT.PUT_LINE(v_date);
END;
```

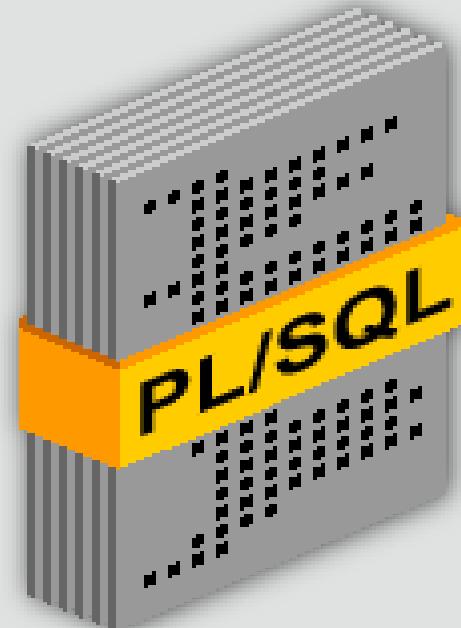
Examples of Anonymous Blocks

- Declarative, Executable, and Exception sections

```
DECLARE
    v_first_name      VARCHAR2(25);
    v_last_name       VARCHAR2(25);
BEGIN
    SELECT first_name, last_name
        INTO v_first_name, v_last_name
        FROM employees
        WHERE last_name = 'Oswald';
    DBMS_OUTPUT.PUT_LINE ('The employee of the month is: '
                          || v_first_name || ' ' || v_last_name || '.');
EXCEPTION
    WHEN TOO_MANY_ROWS THEN
        DBMS_OUTPUT.PUT_LINE ('Your select statement retrieved
                           multiple rows. Consider using a cursor or changing
                           the search criteria.');
END;
```

The PL/SQL Compiler

- The anonymous block found on the previous slide is automatically compiled when it is executed
- If the code has errors that prevent it from compiling, it will not execute, but will return the first compile error it detects



The PL/SQL Compiler

- Every program written in a high-level programming language (C, Java, PL/SQL, and so on) must be checked and translated into binary code (ones and zeros) before it can execute
- The software that does this checking and translation is called a compiler
- The PL/SQL compiler executes automatically when needed
- It checks not only that every command is spelled correctly, but also that any referenced database objects (such as tables) exist, and that the user has the necessary privileges to access them

Subprograms

- Are named PL/SQL blocks
- Are stored in the database
- Can be invoked whenever you want depending on your application

```
PROCEDURE name
IS
    -- variable declarations
BEGIN
    -- statements
[EXCEPTION]
END;
```

Subprograms

- Can be declared as procedures or as functions
 - Procedure: Performs an action
 - Function: Computes and returns a value

```
FUNCTION name
RETURN datatype
-- variable declaration(s)
IS
BEGIN
-- statements
RETURN value;

[EXCEPTION]

END;
```

Examples of Subprogram Code Blocks

- Code block to create a procedure called PRINT_DATE:

```
CREATE OR REPLACE PROCEDURE print_date IS
    v_date VARCHAR2(30);
BEGIN
    SELECT TO_CHAR(SYSDATE,'Mon DD, YYYY')
        INTO v_date
        FROM DUAL;
    DBMS_OUTPUT.PUT_LINE(v_date);
END;
```

- You could call this procedure in an executable section :

```
BEGIN
    PRINT_DATE;
END;
```

Examples of Subprogram Code Blocks

- Code block to create a function called TOMORROW:

```
CREATE OR REPLACE FUNCTION tomorrow (p_today IN DATE)
  RETURN DATE IS
  v_tomorrow DATE;
BEGIN
  SELECT p_today + 1 INTO v_tomorrow
    FROM DUAL;
  RETURN v_tomorrow;
END;
```

- You could call the function using either a SQL statement or a PL/SQL block as shown below:

```
SELECT TOMORROW(SYSDATE) AS "Tomorrow's Date"
FROM DUAL;
or
BEGIN
DBMS_OUTPUT.PUT_LINE(TOMORROW(SYSDATE));
END;
```

PL/SQL Programming Environments

- There are many tools available from Oracle that provide an environment for developing database-driven applications using PL/SQL

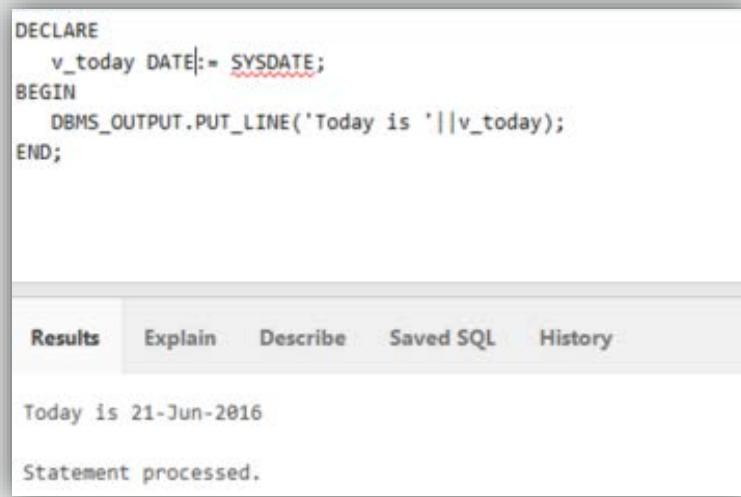
ORACLE

Application Express	Browser-based, database-driven, application development environment.
SQL Workshop	A component of Application Express.
Application Builder	A component of Application Express.
SQL Developer	An IDE for database development and management.
JDeveloper	An IDE for Java-based development.
NetBeans	An IDE for Java, HTML5, PHP, and C++.

ORACLE
Academy

SQL Commands

- As you did in the SQL course, you can use SQL Commands to enter and run a SINGLE SQL statement
- You also use SQL Commands to enter and run a SINGLE PL/SQL block



The screenshot shows a code editor window with the following PL/SQL block:

```
DECLARE
    v_today DATE := SYSDATE;
BEGIN
    DBMS_OUTPUT.PUT_LINE('Today is ' || v_today);
END;
```

Below the code editor is a toolbar with tabs: Results, Explain, Describe, Saved SQL, and History. The Results tab is selected. The results pane displays the output of the executed block:

Today is 21-Jun-2016
Statement processed.



SQL Scripts

- SQL Scripts can contain one or more SQL statements and/or PL/SQL blocks
- Use SQL Scripts to enter and run multi-statement scripts
- In SQL Scripts, anonymous PL/SQL blocks must be followed by a forward slash (/)

```
1 SELECT COUNT(*) FROM employees;
2 /
3 DECLARE
4     v_count NUMBER(6,0);
5 BEGIN
6     SELECT COUNT(*) INTO v_count FROM employees;
7     DBMS_OUTPUT.PUT_LINE(v_count|| ' employees');
8 END;
9 /
10 SELECT SYSDATE FROM dual;
11 /
```

Using DBMS_OUTPUT.PUT_LINE Example

- Look at this simple PL/SQL block and its output
- How can you display the result?

The screenshot shows a PL/SQL editor window with the following code:

```
DECLARE
    v_result NUMBER;
BEGIN
    v_result := 10*25;
END;|
```

Below the code, there is a toolbar with tabs: Results, Explain, Describe, Saved SQL, and History. The "Results" tab is selected. The results pane displays the message "Statement processed." and the execution time "0.00 seconds".

Using DBMS_OUTPUT.PUT_LINE Example

- Let's add a call to the PUT_LINE function in the DBMS_OUTPUT package.
- Now you can see the result!

The screenshot shows a PL/SQL editor window with the following code:

```
Rows 10
DECLARE
    v_result NUMBER;
BEGIN
    v_result := 10*25;
    DBMS_OUTPUT.PUT_LINE(v_result);
END;
|
```

Below the code, there is a toolbar with tabs: Results, Explain, Describe, and Saved S. The Results tab is selected, showing the output:

250
Statement processed.
0.00 seconds

Using DBMS_OUTPUT.PUT_LINE

- The DBMS_OUTPUT.PUT_LINE allows you to display results so that you can check that your block is working correctly
- It allows you to display one character string at a time, although this can be concatenated

```
DECLARE
    v_emp_count NUMBER;
BEGIN
    DBMS_OUTPUT.PUT_LINE('PL/SQL is easy so far!');
    SELECT COUNT(*) INTO v_emp_count FROM employees;
    DBMS_OUTPUT.PUT_LINE('There are '||v_emp_count|||
                           rows in the employees table');
END;
```

Terminology

- Key terms used in this lesson included:
 - Anonymous PL/SQL block
 - Compiler
 - Subprograms
 - Procedures
 - Functions

Summary

- In this lesson, you should have learned how to:
 - Describe the structure of a PL/SQL block
 - Identify the different types of PL/SQL blocks
 - Identify PL/SQL programming environments
 - Create and execute an anonymous PL/SQL block
 - Output messages in PL/SQL

ORACLE

Academy

ORACLE

Academy

Database Programming with PL/SQL

1-3

Creating PL/SQL Blocks

ORACLE
Academy



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Objectives

- This lesson covers the following objectives:
 - Describe the structure of a PL/SQL block
 - Identify the different types of PL/SQL blocks
 - Identify PL/SQL programming environments
 - Create and execute an anonymous PL/SQL block
 - Output messages in PL/SQL

Purpose

- Here you will learn the structure of a PL/SQL block and create one kind of block: an anonymous block
- Later in the course, you will learn to create Procedures, Functions, and Packages using the basic structure found in anonymous blocks
- After learning about the different environments into which you can develop your PL/SQL programs, you will also begin coding PL/SQL in the Application Express development environment

PL/SQL Block Structure

- A PL/SQL block consists of three sections

Section	Description
Declarative (optional)	The declarative section begins with the keyword DECLARE and ends when your executable section starts.
Executable (mandatory)	The executable section begins with the keyword BEGIN and ends with END. Observe that END is terminated with a semicolon. The executable section of a PL/SQL block can include any number of nested PL/SQL blocks.
Exception handling (optional)	The exception section is nested within the executable section. This section begins with the keyword EXCEPTION.



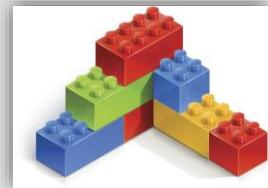
PL/SQL Block Structure Sections

Section	Description	Inclusion
Declarative (DECLARE)	Contains declarations of all variables, constants, cursors, and user-defined exceptions that are referenced in the executable and exception sections.	Optional
Executable (BEGIN ... END;)	Contains SQL statements to retrieve data from the database and PL/SQL statements to manipulate data in the block. Must contain at least one statement.	Mandatory
Exception (EXCEPTION)	Specifies the actions to perform when errors and abnormal conditions arise in the executable section.	Optional

DECLARE is not needed if no variables, constants, cursors or user-defined exceptions are required. But nearly all real-life blocks will need variables and/or cursors, therefore nearly all real-life blocks will need a DECLARE section.

Anonymous Blocks

- Characteristics of anonymous blocks:
 - Unnamed block
 - Not stored in the database
 - Declared inline at the point in an application where it is executed
 - Compiled each time the application is executed
 - Passed to the PL/SQL engine for execution at run time
 - Cannot be invoked or called because it does not have a name and does not exist after it is executed



Anonymous Blocks – Basic Structure

- Basic structure of an anonymous block:

```
[DECLARE]
```

```
BEGIN  
  --statements
```

```
[EXCEPTION]
```

```
END ;
```

- The DECLARE and EXCEPTION keywords/sections are optional

Examples of Anonymous Blocks

- Executable section only (minimum required)

```
BEGIN  
    DBMS_OUTPUT.PUT_LINE('PL/SQL is easy!');  
END ;
```

- Declarative and Executable sections

```
DECLARE  
    v_date      DATE := SYSDATE;  
BEGIN  
    DBMS_OUTPUT.PUT_LINE(v_date);  
END ;
```

PUT_LINE is a function in the DBMS_OUTPUT package that displays its argument (in this slide, the value stored in v_date) on the screen for the user to see.

Examples of Anonymous Blocks

- Declarative, Executable, and Exception sections

```
DECLARE
    v_first_name    VARCHAR2(25);
    v_last_name     VARCHAR2(25);
BEGIN
    SELECT first_name, last_name
        INTO v_first_name, v_last_name
        FROM employees
        WHERE last_name = 'Oswald';
    DBMS_OUTPUT.PUT_LINE ('The employee of the month is: '
        || v_first_name || ' ' || v_last_name || '.');
EXCEPTION
    WHEN TOO_MANY_ROWS THEN
        DBMS_OUTPUT.PUT_LINE ('Your select statement retrieved
            multiple rows. Consider using a cursor or changing
            the search criteria.');
END;
```



Academy

PLSQL 1-3
Creating PL/SQL Blocks

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

10

What happens if the SELECT statement finds more than one employee with that last name? There is only room for one first name and one last name. Without the EXCEPTION section, the DECLARATIVE section would abort and return an Oracle error message. With the EXCEPTION section, the abort is handled by the code in the EXCEPTION section and the user (or later, the calling application) sees a friendlier error message.

The PL/SQL Compiler

- The anonymous block found on the previous slide is automatically compiled when it is executed
- If the code has errors that prevent it from compiling, it will not execute, but will return the first compile error it detects



The PL/SQL Compiler

- Every program written in a high-level programming language (C, Java, PL/SQL, and so on) must be checked and translated into binary code (ones and zeros) before it can execute
- The software that does this checking and translation is called a compiler
- The PL/SQL compiler executes automatically when needed
- It checks not only that every command is spelled correctly, but also that any referenced database objects (such as tables) exist, and that the user has the necessary privileges to access them

Subprograms

- Are named PL/SQL blocks
- Are stored in the database
- Can be invoked whenever you want depending on your application

```
PROCEDURE name
IS
    -- variable declarations
BEGIN
    -- statements
[EXCEPTION]
END;
```

Subprograms

- Can be declared as procedures or as functions
 - Procedure: Performs an action
 - Function: Computes and returns a value

```
FUNCTION name
RETURN datatype
  -- variable declaration(s)
IS
BEGIN
  -- statements
  RETURN value;
[EXCEPTION]
END;
```

ORACLE

Academy

PLSQL 1-3
Creating PL/SQL Blocks

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

14

Examples of Subprogram Code Blocks

- Code block to create a procedure called PRINT_DATE:

```
CREATE OR REPLACE PROCEDURE print_date IS
    v_date VARCHAR2(30);
BEGIN
    SELECT TO_CHAR(SYSDATE, 'Mon DD, YYYY')
        INTO v_date
        FROM DUAL;
    DBMS_OUTPUT.PUT_LINE(v_date);
END;
```

- You could call this procedure in an executable section :

```
BEGIN
    PRINT_DATE;
END;
```



This is an example of a PL/SQL code block that is NOT an anonymous block.

This block creates a PROCEDURE that when called will display today's date.

Examples of Subprogram Code Blocks

- Code block to create a function called TOMORROW:

```
CREATE OR REPLACE FUNCTION tomorrow (p_today IN DATE)
  RETURN DATE IS
  v_tomorrow DATE;
BEGIN
  SELECT p_today + 1 INTO v_tomorrow
    FROM DUAL;
  RETURN v_tomorrow;
END;
```

- You could call the function using either a SQL statement or a PL/SQL block as shown below:

```
SELECT TOMORROW(SYSDATE) AS "Tomorrow's Date"
FROM DUAL;
or
BEGIN
DBMS_OUTPUT.PUT_LINE(TOMORROW(SYSDATE));
END;
```

ORACLE

Academy

PLSQL 1-3
Creating PL/SQL Blocks

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

16

This is an example of a PL/SQL code block that is NOT an anonymous block.

This block creates a FUNCTION that will return tomorrow's date (much like SYSDATE is a function that returns today's date).

The FUNCTION could also be used to initialize a variable:

```
DECLARE  v_tomorrow  DATE := TOMORROW(SYSDATE);
BEGIN
DBMS_OUTPUT.PUT_LINE('Tomorrow is ' || v_tomorrow);
END;
```

PL/SQL Programming Environments

- There are many tools available from Oracle that provide an environment for developing database-driven applications using PL/SQL



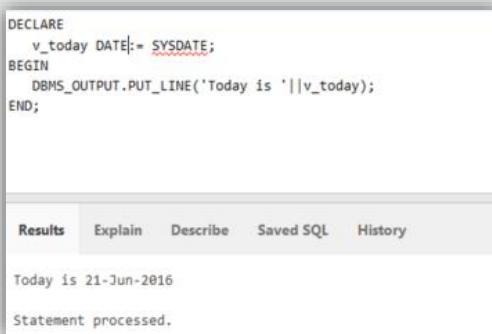
Application Express	Browser-based, database-driven, application development environment.
SQL Workshop	A component of Application Express.
Application Builder	A component of Application Express.
SQL Developer	An IDE for database development and management.
JDeveloper	An IDE for Java-based development.
NetBeans	An IDE for Java, HTML5, PHP, and C++.



This course will focus on Application Express and its SQL Workshop. You also will use Application Builder in a project. The other tools are all free downloads with excellent documentation available from www.oracle.com. All of these tools are widely-used in the business world.

SQL Commands

- As you did in the SQL course, you can use SQL Commands to enter and run a SINGLE SQL statement
- You also use SQL Commands to enter and run a SINGLE PL/SQL block



The screenshot shows a PL/SQL block being run in an Oracle SQL developer interface. The code is:

```
DECLARE
    v_today DATE := SYSDATE;
BEGIN
    DBMS_OUTPUT.PUT_LINE('Today is '||v_today);
END;
```

The results tab is selected, showing the output:

Today is 21-Jun-2016
Statement processed.

SQL Scripts

- SQL Scripts can contain one or more SQL statements and/or PL/SQL blocks
- Use SQL Scripts to enter and run multi-statement scripts
- In SQL Scripts, anonymous PL/SQL blocks must be followed by a forward slash (/)

```
1 SELECT COUNT(*) FROM employees;
2 /
3 DECLARE
4   v_count NUMBER(6,0);
5 BEGIN
6   SELECT COUNT(*) INTO v_count FROM employees;
7   DBMS_OUTPUT.PUT_LINE(v_count|| ' employees');
8 END;
9 /
10 SELECT SYSDATE FROM dual;
11 /
```



Although not required, the convention is to follow all SQL statements with a forward slash (/), as well as the anonymous blocks, in SQL script files.

Using DBMS_OUTPUT.PUT_LINE Example

- Look at this simple PL/SQL block and its output
- How can you display the result?

```
DECLARE
    v_result NUMBER;
BEGIN
    v_result := 10*25;
END;|
```

Results Explain Describe Saved SQL History

Statement processed.

0.00 seconds

Using DBMS_OUTPUT.PUT_LINE Example

- Let's add a call to the PUT_LINE function in the DBMS_OUTPUT package.
- Now you can see the result!

The screenshot shows a SQL developer interface. At the top, there is a row count input field set to 10. Below it is a code editor containing the following PL/SQL block:

```
DECLARE
    v_result NUMBER;
BEGIN
    v_result := 10*25;
    DBMS_OUTPUT.PUT_LINE(v_result);
END;
```

Below the code editor is a results panel with tabs: Results (selected), Explain, Describe, and Saved. The results tab displays the output of the query:

```
250
Statement processed.
```

At the bottom of the results panel, it shows "0.00 seconds" for execution time.

ORACLE
Academy

PLSQL 1-3
Creating PL/SQL Blocks

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

21

Using DBMS_OUTPUT.PUT_LINE

- The DBMS_OUTPUT.PUT_LINE allows you to display results so that you can check that your block is working correctly
- It allows you to display one character string at a time, although this can be concatenated

```
DECLARE
  v_emp_count NUMBER;
BEGIN
  DBMS_OUTPUT.PUT_LINE('PL/SQL is easy so far!');
  SELECT COUNT(*) INTO v_emp_count FROM employees;
  DBMS_OUTPUT.PUT_LINE('There are'||v_emp_count|||
                        'rows in the employees table');
END;
```

The second DBMS_OUTPUT.PUT_LINE call in the slide shows that number values (v_emp_count) can be displayed by PUT_LINE in combination with string values.

In this example, the Oracle server has performed an implicit datatype conversion (TO_CHAR(v_emp_count)) to convert the number to a character string for concatenation.

Terminology

- Key terms used in this lesson included:
 - Anonymous PL/SQL block
 - Compiler
 - Subprograms
 - Procedures
 - Functions

- Anonymous PL/SQL block – unnamed blocks of code not stored in the database and do not exist after they are executed
- Compiler – software that checks and translates programs written in high-level programming languages into binary code to execute
- Subprograms – named PL/SQL blocks that are stored in the database and can be declared as PROCEDURES or FUNCTIONS
- Procedures – subprograms that perform an action and may return one or more values
- Functions – subprograms that return a single value

Summary

- In this lesson, you should have learned how to:
 - Describe the structure of a PL/SQL block
 - Identify the different types of PL/SQL blocks
 - Identify PL/SQL programming environments
 - Create and execute an anonymous PL/SQL block
 - Output messages in PL/SQL

ORACLE

Academy

Database Programming with PL/SQL

1-3: Creating PL/SQL Blocks

Practice Activities

Vocabulary

Identify the vocabulary word for each definition below:

	Unnamed blocks of code not stored in the database and do not exist after they are executed
	A program that computes and returns a single value
	Named PL/SQL blocks that are stored in the database and can be declared as procedures or functions
	Software that checks and translates programs written in high-level programming languages into binary code to execute
	A program that performs an action, but does not have to return a value

Try It / Solve It

1. Complete the following chart defining the syntactical requirements for a PL/SQL block:

	Optional or Mandatory?	Describe what is included in this section
DECLARE		
BEGIN		
EXCEPTION		
END;		

2. Which of the following PL/SQL blocks executes successfully? For the blocks that fail, explain why they fail

- A. BEGIN
END;

- B. DECLARE
 amount INTEGER(10);
END;

- C. DECLARE
BEGIN
END;

- D. DECLARE
 amount NUMBER(10);
BEGIN
 DBMS_OUTPUT.PUT_LINE(amount);
END;

3. Fill in the blanks:

- A. PL/SQL blocks that have no names are called _____.

 - B. _____ and _____ are named blocks and are stored in the database.
4. In Application Express, create and execute a simple anonymous block that outputs “Hello World.”
5. Create and execute a simple anonymous block that does the following:
- Declares a variable of datatype DATE and populates it with the date that is six months from today
 - Outputs “In six months, the date will be: <insert date>.”

ORACLE

Academy

Database Programming with PL/SQL

1-1

Introduction to PL/SQL

ORACLE
Academy



Objectives

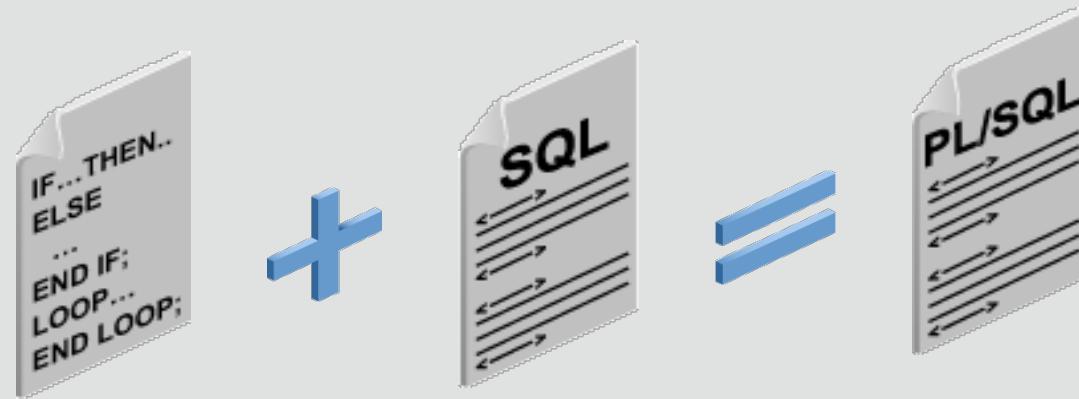
- This lesson covers the following objectives:
 - Describe PL/SQL
 - Differentiate between SQL and PL/SQL
 - Explain the need for PL/SQL

Purpose

- PL/SQL is Oracle Corporation's procedural programming language for relational databases
- To describe PL/SQL, you learn its characteristics and identify the differences between PL/SQL and SQL
- Identifying limitations of SQL and how PL/SQL addresses those limitations will help you to understand why PL/SQL is needed

PL/SQL Description

- Procedural Language extension to SQL
- A third-generation programming language (3GL)
- An Oracle proprietary programming language
- Combines program logic and control flow with SQL



Evolution/Generations of Programming Languages:

- 1GL: First-generation programming languages; these are machine level languages specific to a particular CPU
- 2GL: Second-generation programming languages; assembly languages specific to a particular CPU; converted by an assembler into a machine language; commonly used for performance-oriented and processing-intensive applications such as firmware interfaces and hardware drivers
- 3GL: Third-generation programming languages; converted into machine language by a compiler; less cryptic and thus more programmer-friendly than 2GLs (ex., Visual Basic, C, C++, COBOL, FORTRAN, Java, Pascal, PL/SQL)

Evolution/Generations of Programming Languages:

- 4GL: Fourth-generation programming languages; less cryptic and thus more programmer-friendly than 2GLs; unlike the broad applicability of 3GLs, most 4GLs are used with databases, for queries, report generation, data manipulation, etc. (ex., SQL, MySQL)
- 5GL: Fifth-generation programming languages; generally involve a visual or graphical development environment that exports source language to a 3GL or 4GL compiler; 5GL may also refer to languages that define a problem and a set of constraints, then let the computer find a solution; (ex., AI systems such as PROLOG used with IBM's Watson)
- As an Oracle proprietary programming language, PL/SQL is only used with an Oracle database

SQL Description

- A nonprocedural language
- Also known as a "declarative language," allows programmer to focus on input and output rather than the program steps
- A fourth-generation programming language (4GL)
- Primary language used to access and modify data in relational databases
- Standardized by the American National Standards Institute (ANSI)
- Vendors such as Oracle typically include some proprietary SQL features in their database environments

SQL Statement

- The SQL statement shown is simple and straightforward
- However, if you need to modify a data item in a conditional manner, you come across a limitation of SQL

```
SELECT employee_id, job_id, hire_date  
      FROM employees;
```

- For example, how would you write an SQL statement to update the job_id data with a new value determined by the current job_id and the hire_date?

Limitations of SQL

- Assume the company decides to promote all sales representatives, marketing representatives, and stock clerks employed for at least ten years to senior representatives and clerks
- If the current date is 05-Feb-2015, sales reps 174, 176, and 178 qualify for the promotion

EMPLOYEE_ID	JOB_ID	HIRE_DATE	“NEW” JOB_ID
174	SA_REP	11-May-1996	SR_SA REP
176	SA_REP	24-Mar-1998	SR_SA REP
178	SA_REP	24-May-1999	SR_SA REP
240	SA_REP	02-Oct-2005	
242	SA_REP	09-Dec-2007	

Limitations of SQL

- If the current date is 05-FEB-2015, stock clerks 141, 142, 143, and 144 also qualify for the promotion

EMPLOYEE_ID	JOB_ID	HIRE_DATE	“NEW” JOB_ID
141	ST_CLERK	17-Oct-1995	SR_ST_CLERK
142	ST_CLERK	29-Jan-1997	SR_ST_CLERK
143	ST_CLERK	15-Mar-1998	SR_ST_CLERK
144	ST_CLERK	09-Jul-1998	SR_ST_CLERK
244	ST_CLERK	07-Sep-2009	

Limitations of SQL

- One solution to updating the job_id data is shown
- How many SQL statements do you need to write for sales representatives, marketing representatives, and stock clerks?
- What if there are other job_ids to update?

```
UPDATE employees
      SET job_id = 'SR_SA_REP'
    WHERE job_id = 'SA_REP' AND
          hire_date <= '05-Feb-2005'
```

```
UPDATE employees
      SET job_id = 'SR_ST_CLERK'
    WHERE job_id = 'ST_CLERK' AND
          hire_date <= '05-Feb-2005'
```

Limitations of SQL

- You would need to write a separate SQL statement for each job_id that needs to be updated
- Depending on the number of job_ids, this could be a tedious task
- It would be easier to write a single statement to accomplish this task
- The statement would require logic, otherwise known as procedural logic
- PL/SQL extends SQL with procedural logic and makes it possible to write one statement to accomplish this task

PL/SQL Extends SQL with Procedural Logic

- Using PL/SQL, you can write one statement to promote the sales representatives, marketing representatives, and stock clerks.

```
DECLARE
    CURSOR c_employees IS SELECT * FROM employees;
BEGIN
    FOR c_emp in c_employees
    LOOP
        IF c_emp.job_id = 'SA_REP' AND c_emp.hire_date <= '05-Feb-2005' THEN
            UPDATE employees
            SET job_id = 'SR_SA REP' WHERE employee_id = c_emp.employee_id;
        ELSIF c_emp.job_id = 'MK_REP' AND c_emp.hire_date <= '05-Feb-2005' THEN
            UPDATE employees
            SET job_id = 'SR_MK_REP' WHERE employee_id = c_emp.employee_id;
        ELSIF c_emp.job_id = 'ST_CLERK' AND c_emp.hire_date <= '05-Feb-2005' THEN
            UPDATE employees
            SET job_id = 'SR_ST_CLRK' WHERE employee_id = c_emp.employee_id;
        END IF;
    END LOOP;
END;
```

Procedural Constructs

- You use PL/SQL to write the procedural code and embed SQL statements within the PL/SQL code
- The procedural code includes variables, constants, cursors, conditional logic, and iteration
- PL/SQL code blocks can be saved and named, then executed whenever needed



Procedural Constructs Highlighted

- Several PL/SQL constructs are highlighted below

```
DECLARE          Cursor
    CURSOR c_employees IS SELECT * FROM employees;
BEGIN
    FOR c_emp IN c_employees
    LOOP
        IF c_emp.job_id = 'SA_REP' AND c_emp.hire_date <= '05-Feb-2005' THEN
            UPDATE employees
            SET job_id = 'SR_SA REP'
            WHERE employee_id = c_emp.employee_id;
        ELSIF c_emp.job_id = 'MK_REP' AND c_emp.hire_date <= '05-Feb-2005' THEN
            UPDATE employees
            SET job_id = 'SR_MK REP'
            WHERE employee_id = c_emp.employee_id;
        ELSIF c_emp.job_id = 'ST_CLERK' AND c_emp.hire_date <= '05-Feb-2005' THEN
            UPDATE employees
            SET job_id = 'SR_ST CLRK'
            WHERE employee_id = c_emp.employee_id;
        END IF;
    END LOOP;
END;
```

Iterative Control (highlighted by a red box and arrows)

Conditional Control (highlighted by a red box and arrows)

SQL (highlighted by a red box and arrows)

Cursor (highlighted by a red box)

Characteristics of PL/SQL

- **PL/SQL:**

- Is a highly structured, readable, and accessible language
- Is a standard and portable language for Oracle development
- Is an embedded language and it works with SQL
- Is a high-performance, highly integrated database language
- Is based on the Ada Programming Language and has many similarities in syntax

Terminology

- Key terms used in this lesson included:
 - PL/SQL
 - Procedural Constructs

Summary

- In this lesson, you should have learned how to:
 - Describe PL/SQL
 - Differentiate between SQL and PL/SQL
 - Explain the need for PL/SQL

ORACLE

Academy

ORACLE

Academy

Database Programming with PL/SQL

1-1

Introduction to PL/SQL

ORACLE
Academy



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Objectives

- This lesson covers the following objectives:
 - Describe PL/SQL
 - Differentiate between SQL and PL/SQL
 - Explain the need for PL/SQL

Purpose

- PL/SQL is Oracle Corporation's procedural programming language for relational databases
- To describe PL/SQL, you learn its characteristics and identify the differences between PL/SQL and SQL
- Identifying limitations of SQL and how PL/SQL addresses those limitations will help you to understand why PL/SQL is needed

PL/SQL Description

- Procedural Language extension to SQL
- A third-generation programming language (3GL)
- An Oracle proprietary programming language
- Combines program logic and control flow with SQL



Evolution/Generations of Programming Languages:

- 1GL: First-generation programming languages; these are machine level languages specific to a particular CPU
- 2GL: Second-generation programming languages; assembly languages specific to a particular CPU; converted by an assembler into a machine language; commonly used for performance-oriented and processing-intensive applications such as firmware interfaces and hardware drivers
- 3GL: Third-generation programming languages; converted into machine language by a compiler; less cryptic and thus more programmer-friendly than 2GLs (ex., Visual Basic, C, C++, COBOL, FORTRAN, Java, Pascal, PL/SQL)

Evolution/Generations of Programming Languages:

- 4GL: Fourth-generation programming languages; less cryptic and thus more programmer-friendly than 2GLs; unlike the broad applicability of 3GLs, most 4GLs are used with databases, for queries, report generation, data manipulation, etc. (ex., SQL, MySQL)
- 5GL: Fifth-generation programming languages; generally involve a visual or graphical development environment that exports source language to a 3GL or 4GL compiler; 5GL may also refer to languages that define a problem and a set of constraints, then let the computer find a solution; (ex., AI systems such as PROLOG used with IBM's Watson)
- As an Oracle proprietary programming language, PL/SQL is only used with an Oracle database

SQL Description

- A nonprocedural language
- Also known as a "declarative language," allows programmer to focus on input and output rather than the program steps
- A fourth-generation programming language (4GL)
- Primary language used to access and modify data in relational databases
- Standardized by the American National Standards Institute (ANSI)
- Vendors such as Oracle typically include some proprietary SQL features in their database environments



PLSQL 1-1
Introduction to PL/SQL

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

8

Although PL/SQL was developed after SQL, SQL is a 4GL while PL/SQL is a 3GL.

SQL and PL/SQL are both needed. They are not alternatives to each other. Only SQL can be used to access database tables and only PL/SQL can be used to write the procedural logic.

SQL Statement

- The SQL statement shown is simple and straightforward
- However, if you need to modify a data item in a conditional manner, you come across a limitation of SQL

```
SELECT employee_id, job_id, hire_date  
      FROM employees;
```

- For example, how would you write an SQL statement to update the job_id data with a new value determined by the current job_id and the hire_date?

Limitations of SQL

- Assume the company decides to promote all sales representatives, marketing representatives, and stock clerks employed for at least ten years to senior representatives and clerks
- If the current date is 05-Feb-2015, sales reps 174, 176, and 178 qualify for the promotion

EMPLOYEE_ID	JOB_ID	HIRE_DATE	"NEW" JOB_ID
174	SA REP	11-May-1996	SR_SA REP
176	SA REP	24-Mar-1998	SR_SA REP
178	SA REP	24-May-1999	SR_SA REP
240	SA REP	02-Oct-2005	
242	SA REP	09-Dec-2007	

Limitations of SQL

- If the current date is 05-FEB-2015, stock clerks 141, 142, 143, and 144 also qualify for the promotion

EMPLOYEE_ID	JOB_ID	HIRE_DATE	"NEW" JOB_ID
141	ST_CLERK	17-Oct-1995	SR_ST_CLERK
142	ST_CLERK	29-Jan-1997	SR_ST_CLERK
143	ST_CLERK	15-Mar-1998	SR_ST_CLERK
144	ST_CLERK	09-Jul-1998	SR_ST_CLERK
244	ST_CLERK	07-Sep-2009	

Limitations of SQL

- One solution to updating the job_id data is shown
- How many SQL statements do you need to write for sales representatives, marketing representatives, and stock clerks?
- What if there are other job_ids to update?

```
UPDATE employees
    SET job_id = 'SR_SA REP'
    WHERE job_id = 'SA REP' AND
        hire_date <= '05-Feb-2005'
```

```
UPDATE employees
    SET job_id = 'SR_ST_CLERK'
    WHERE job_id = 'ST_CLERK' AND
        hire_date <= '05-Feb-2005'
```

Limitations of SQL

- You would need to write a separate SQL statement for each job_id that needs to be updated
- Depending on the number of job_ids, this could be a tedious task
- It would be easier to write a single statement to accomplish this task
- The statement would require logic, otherwise known as procedural logic
- PL/SQL extends SQL with procedural logic and makes it possible to write one statement to accomplish this task

PL/SQL Extends SQL with Procedural Logic

- Using PL/SQL, you can write one statement to promote the sales representatives, marketing representatives, and stock clerks.

```
DECLARE
    CURSOR c_employees IS SELECT * FROM employees;
BEGIN
    FOR c_emp IN c_employees
    LOOP
        IF c_emp.job_id = 'SA REP' AND c_emp.hire_date <= '05-Feb-2005' THEN
            UPDATE employees
            SET job_id = 'SR_SA REP' WHERE employee_id = c_emp.employee_id;
        ELSIF c_emp.job_id = 'MK REP' AND c_emp.hire_date <= '05-Feb-2005' THEN
            UPDATE employees
            SET job_id = 'SR_MK REP' WHERE employee_id = c_emp.employee_id;
        ELSIF c_emp.job_id = 'ST CLERK' AND c_emp.hire_date <= '05-Feb-2005' THEN
            UPDATE employees
            SET job_id = 'SR_ST CLRK' WHERE employee_id = c_emp.employee_id;
        END IF;
    END LOOP;
END;
```

Procedural Constructs

- You use PL/SQL to write the procedural code and embed SQL statements within the PL/SQL code
- The procedural code includes variables, constants, cursors, conditional logic, and iteration
- PL/SQL code blocks can be saved and named, then executed whenever needed



ORACLE
Academy

PLSQL 1-1
Introduction to PL/SQL

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

15

As powerful as SQL is, it simply does not offer the flexibility and power developers need to create full-blown applications. Oracle's PL/SQL language ensures that we can stay entirely within the operating system-independent Oracle environment and still write highly efficient applications that meet our users' requirements.

Procedural Constructs Highlighted

- Several PL/SQL constructs are highlighted below

```
DECLARE
    CURSOR c_employees IS SELECT * FROM employees;
BEGIN
    FOR c_emp IN c_employees
    LOOP
        IF c_emp.job_id = 'SA_REP' AND c_emp.hire_date <='05-Feb-2005' THEN
            UPDATE employees
            SET job_id = 'SR_SA REP'
            WHERE employee_id = c_emp.employee_id;
        ELSIF c_emp.job_id = 'MK_REP' AND c_emp.hire_date <= '05-Feb-2005' THEN
            UPDATE employees
            SET job_id = 'SR_MK REP'
            WHERE employee_id = c_emp.employee_id;
        ELSIF c_emp.job_id = 'ST_CLERK' AND c_emp.hire_date <='05-Feb-2005' THEN
            UPDATE employees
            SET job_id = 'SR_ST_CLRK'
            WHERE employee_id = c_emp.employee_id;
        END IF;
    END LOOP;
END;
```

Iterative Control (points to the `FOR` loop and the `LOOP` keyword)

Conditional Control (points to the `IF`, `ELSIF`, and `END IF` keywords)

Cursor (points to the `CURSOR` declaration)

SQL (points to the `UPDATE`, `SET`, and `WHERE` clauses)

Characteristics of PL/SQL

- **PL/SQL:**

- Is a highly structured, readable, and accessible language
- Is a standard and portable language for Oracle development
- Is an embedded language and it works with SQL
- Is a high-performance, highly integrated database language
- Is based on the Ada Programming Language and has many similarities in syntax

Terminology

- Key terms used in this lesson included:
 - PL/SQL
 - Procedural Constructs

- PL/SQL is Oracle Corporation's procedural language for relational databases which allows program logic and control flow to be combined with SQL statements.
- The procedural constructs include variables, constants, cursors, conditional logic, and iteration.

Summary

- In this lesson, you should have learned how to:
 - Describe PL/SQL
 - Differentiate between SQL and PL/SQL
 - Explain the need for PL/SQL

ORACLE

Academy

Database Programming with PL/SQL

1-1: Introduction to PL/SQL

Practice Activities

Vocabulary

Identify the vocabulary word for each definition below:

	Programming language features such as reusable/callable program units, modular blocks, cursors, constants, variables, assignment statements, conditional control statements, and loops
	Oracle Corporations standard procedural language for relational databases which allows basic program logic and control flow to be combined with SQL statements

Try It / Solve It

1. Circle the programming language meeting the criteria

Criteria	Language	
3GL	PL/SQL	SQL
4GL	PL/SQL	SQL
Is proprietary to Oracle Corporation	PL/SQL	SQL
Nonprocedural	PL/SQL	SQL
Procedural	PL/SQL	SQL
Is ANSI-compliant	PL/SQL	SQL

2. In your own words, describe why a procedural language like PL/SQL is needed.
3. List some examples of procedural constructs in PL/SQL.

4. In the following code, identify (circle or highlight) examples of these procedural constructs: variable, conditional control statement, reusable/callable program unit, and an assignment statement.

```
DECLARE
    v_first_name varchar2(40);
    v_last_name varchar2(40);
    v_first_letter varchar2(1);
BEGIN
    SELECT first_name, last_name INTO v_first_name, v_last_name
        FROM students
        WHERE student_id = 105;
    v_first_letter := get_first_letter(v_last_name);
    IF 'N' > v_first_letter THEN
        DBMS_OUTPUT.PUT_LINE('The last name for ' || v_first_name || ' ' || v_last_name || ' is
                            between A and M');
    ELSE
        DBMS_OUTPUT.PUT_LINE('The last name for ' || v_first_name || ' ' || v_last_name || ' is
                            between N and Z');
    END IF;
END;
```

ORACLE

Academy

Database Programming with PL/SQL

2-1

Using Variables in PL/SQL

ORACLE
Academy



Objectives

- This lesson covers the following objectives:
 - List the uses of variables in PL/SQL
 - Identify the syntax for variables in PL/SQL
 - Declare and initialize variables in PL/SQL
 - Assign new values to variables in PL/SQL

Purpose

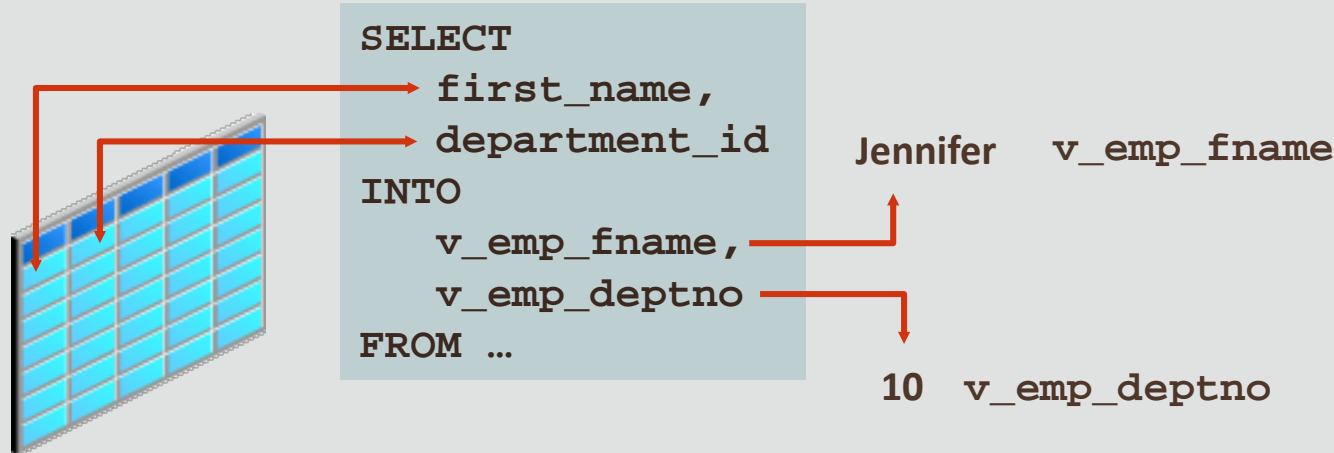
- You use variables to store and manipulate data
- In this lesson, you learn how to declare and initialize variables in the declarative section of a PL/SQL block
- With PL/SQL, you can declare variables and then use them in both SQL and procedural statements
- Variables can be thought of as storage containers that hold something until it is needed

Use of Variables

- Variables are expressions that stand for something of value (in the equation $x + y = 45$, x and y are variables that stand for two numbers that add up to 45)
- When defining a variable in a PL/SQL declaration section, you label a memory location, assign a datatype, and, if needed, assign a starting value for the variable
- A variable can represent a number, character string, boolean (true/false value), or other datatypes
- Throughout the PL/SQL code, variable values can be changed by the assignment operator (`:=`)

Use of Variables

- Use variables for:
 - Temporary storage of data
 - Manipulation of stored values
 - Reusability



Handling Variables in PL/SQL

- Variables are:
 - Declared and initialized in the declarative section
 - Used and assigned new values in the executable section
- Variables can be:
 - Passed as parameters to PL/SQL subprograms
 - Assigned to hold the output of a PL/SQL subprogram



Declaring Variables

- All PL/SQL variables must be declared in the declaration section before referencing them in the PL/SQL block
- The purpose of a declaration is to allocate storage space for a value, specify its data type, and name the storage location so that you can reference it
- You can declare variables in the declarative part of any PL/SQL block, subprogram, or package



Declaring Variables: Syntax

- The identifier is the name of the variable
- It and the datatype are the minimum elements required

```
identifier [CONSTANT] datatype [NOT NULL]  
[ := expr | DEFAULT expr];
```



Initializing Variables

- Variables are assigned a memory location inside the DECLARE section
- Variables can be assigned a value at the same time
- This process is called initialization
- The value in a variable also can be modified by reinitializing the variable in the executable section

```
DECLARE
    v_counter INTEGER := 0;
BEGIN
    v_counter := v_counter + 1;
    DBMS_OUTPUT.PUT_LINE(v_counter);
END;
```

Declaring and Initializing Variables Example 1

- This example shows the declaration of several variables of various datatypes using syntax that sets constraints, defaults, and initial values
- You will learn more about the different syntax as the course progresses

```
DECLARE
    fam_birthdate DATE;
    fam_size          NUMBER(2) NOT NULL := 10;
    fam_location      VARCHAR2(13) := 'Florida';
    fam_bank          CONSTANT NUMBER := 50000;
    fam_population    INTEGER;
    fam_name          VARCHAR2(20) DEFAULT 'Roberts';
    fam_party_size    CONSTANT PLS_INTEGER := 20;
```

Declaring and Initializing Variables Example 2

- This example shows the convention of beginning variables with v_ and variables that are configured as constants with c_

```
DECLARE
    v_emp_hiredate      DATE;
    v_emp_deptno        NUMBER(2) NOT NULL := 10;
    v_location          VARCHAR2(13) := 'Atlanta';
    c_comm               CONSTANT NUMBER := 1400;
    v_population         INTEGER;
    v_book_type          VARCHAR2(20) DEFAULT 'fiction';
    v_artist_name        VARCHAR2(50);
    v_firstname          VARCHAR2(20) := 'Rajiv';
    v_lastname           VARCHAR2(20) DEFAULT 'Kumar';
    c_display_no         CONSTANT PLS_INTEGER := 20;
```

- The defining of data types and data structures using a standard in a programming language is a significant aid to readability

Assigning Values in the Executable Section Example 1

- After a variable is declared, you can use it in the executable section of a PL/SQL block
- For example, in the following block, the variable `v_myname` is declared in the declarative section of the block

```
DECLARE
    v_myname  VARCHAR2(20);
BEGIN
    DBMS_OUTPUT.PUT_LINE('My name is: ' || v_myname);
    v_myname := 'John';
    DBMS_OUTPUT.PUT_LINE('My name is: ' || v_myname);
END;
```

Assigning Values in the Executable Section Example 1

- You can access this variable in the executable section of the same block
- What do you think the block will print?

```
DECLARE
    v_myname  VARCHAR2(20);
BEGIN
    DBMS_OUTPUT.PUT_LINE('My name is: ' || v_myname);
    v_myname := 'John';
    DBMS_OUTPUT.PUT_LINE('My name is: ' || v_myname);
END;
```

Assigning Values in the Executable Section Example 1

- In this example, the variable has no value when the first PUT_LINE is executed, but then the value John is assigned to the variable before the second PUT_LINE
- The value of the variable is then concatenated with the string My name is:
- The output is:

```
My name is:  
My name is: John  
  
Statement process.
```

- A non-initialized variable contains a NULL value until a non-null value is explicitly assigned to it

Assigning Values in the Executable Section Example 2

- In this block, the variable `v_myname` is declared and initialized
- It begins with the value John, but the value is then manipulated in the executable section of the block

```
DECLARE
    v_myname  VARCHAR2(20) := 'John';
BEGIN
    v_myname  := 'Steven';
    DBMS_OUTPUT.PUT_LINE('My name is: ' || v_myname);
END;
```

- The output is:

```
My name is: Steven
Statement processed.
```

Passing Variables as Parameters to PL/SQL Subprograms

- Parameters are values passed to a subprogram by the user or by another program
- The subprogram uses the value in the parameter when it runs
- The subprogram may also return a parameter to the calling environment. In PL/SQL, subprograms are generally known as procedures or functions
- You will learn more about procedures and functions as the course progresses

Passing Variables as Parameters to PL/SQL Subprograms

- In the following example, the parameter v_date is being passed to the procedure PUT_LINE, which is part of the package DBMS_OUTPUT

```
DECLARE
    v_date      VARCHAR2( 30 );
BEGIN
    SELECT TO_CHAR(SYSDATE) INTO v_date FROM DUAL;
    DBMS_OUTPUT.PUT_LINE(v_date);
END;
```

Assigning Variables to PL/SQL Subprogram Output

- You can use variables to hold values that are returned by a function (see function definition below and a call to this function on the following slide)

```
CREATE FUNCTION num_characters (p_string IN VARCHAR2)
RETURN INTEGER IS
    v_num_characters INTEGER;
BEGIN
    SELECT LENGTH(p_string) INTO v_num_characters
        FROM DUAL;
    RETURN v_num_characters;
END;
```

- The concept, creation, and use of functions will be covered later in this course

Assigning Variables to PL/SQL Subprogram Output

- In the call to the function num_characters, the value returned by the function will be stored in the variable v_length_of_string

```
DECLARE
    v_length_of_string    INTEGER;
BEGIN
    v_length_of_string := num_characters('Oracle
Corporation');
    DBMS_OUTPUT.PUT_LINE(v_length_of_string);
END;
```

Terminology

- Key terms used in this lesson included:
 - Parameters
 - Variables

Summary

- In this lesson, you should have learned how to:
 - List the uses of variables in PL/SQL
 - Identify the syntax for variables in PL/SQL
 - Declare and initialize variables in PL/SQL
 - Assign new values to variables in PL/SQL

ORACLE

Academy

ORACLE

Academy

Database Programming with PL/SQL

2-7

Good Programming Practices

ORACLE
Academy



Objectives

- This lesson covers the following objectives:
 - List examples of good programming practices
 - Accurately insert comments into PL/SQL code
 - Create PL/SQL code that follows formatting guidelines to produce readable code

Purpose

- Good programming practices are techniques that you can follow to create the best code possible
- Programming practices cover everything from making code more readable to creating code with faster performance
- Software engineering teams often follow a style guide so that everyone on the team uses the same techniques
- This makes it easier to read and modify code written by others
- Course work should follow the conventions demonstrated in this lesson

Good Programming Practices

- Several examples of good programming practices have already been demonstrated and/or discussed in this course:
 - Use explicit data type conversions because implicit data type conversions can be slower and the rules can change in later software releases
 - Use meaningful identifiers when declaring variables, constants, and parameters
 - Declare one variable or constant identifier per line for better readability and code maintenance

Good Programming Practices

- Other good programming practices demonstrated and/or discussed:
 - Avoid ambiguity when choosing identifiers
 - Use the %TYPE attribute to declare a variable according to another previously declared variable or database column
 - Use the NOT NULL constraint when declaring a variable that must hold a value

Programming Guidelines

- Other programming guidelines include:
 - Documenting code with comments
 - Developing a case convention for the code
 - Developing naming conventions for identifiers and other objects
 - Enhancing readability by indenting



Commenting Code

- Comments assist in future maintenance or modification by helping other programmers know what the original programmer intended by the code written
- Even the original programmer will benefit from commenting his/her code when returning for the first time to that code six months hence
- Comments are strictly informational and do not enforce any conditions or behavior on logic or data. They are ignored when code is compiled

Commenting Code Example

- Prefix single-line comments with two dashes (--)
- Place multiple-line comments between the symbols "/* " and " */ "

```
DECLARE
  -- Created by Clara Oswald
  ...
  v_annual_sal NUMBER (9,2);

BEGIN          -- Start of executable section

  /* Compute the annual salary based on the monthly
   salary input from the user */

  v_annual_sal := v_monthly_sal * 12;
  ...
END;          -- End of executable section
```

Variable Case Conventions

- Case Conventions are shown below
- The following table provides guidelines for writing code in uppercase and lowercase to help you distinguish keywords from named objects

Category	Case Convention	Examples
SQL keywords	Uppercase	SELECT, INSERT
PL/SQL keywords	Uppercase	DECLARE, BEGIN, IF
Data types	Uppercase	VARCHAR2, BOOLEAN
Identifiers (variables, etc.)	Lowercase	v_salary, emp_cursor, c_tax_rate, p_empno
Tables and columns	Lowercase	employees, dept_id, salary, hire_date

Naming Conventions

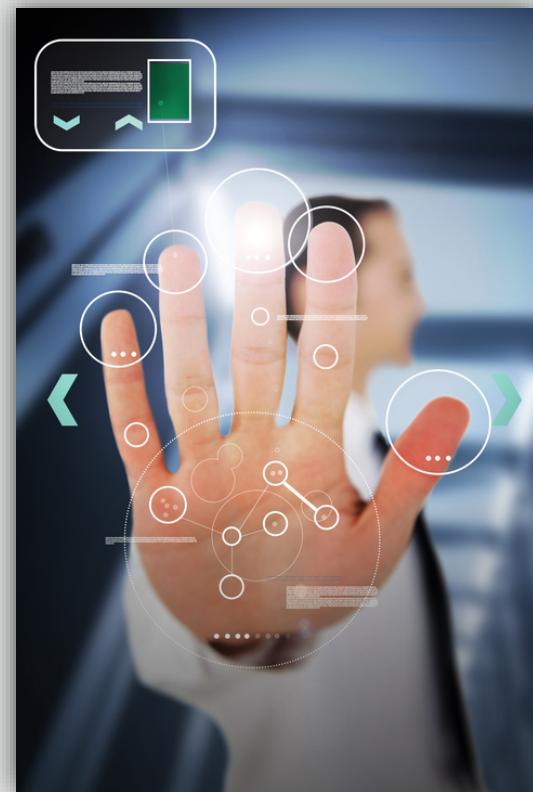
- The naming of identifiers should be clear, consistent, and unambiguous
- One commonly-used convention is to name:
 - Variables starting with v_
 - Constants starting with c_
 - Parameters starting with p_ (for passing to procedures and functions)



Naming Conventions

- Examples:

- v_date_of_birth
- v_last_name
- c_tax_rate
- c_commission_rate
- p_employee_id
- p_salary





Indenting Code

- For clarity, indent each level of code. Examples:

```
BEGIN
  IF x = 0 THEN
    y := 1;
  END IF;
END;
```

```
DECLARE
  v_deptno  NUMBER(4);
  v_location_id NUMBER(4);

BEGIN
  SELECT department_id, location_id
    INTO v_deptno, v_location_id
    FROM departments
   WHERE department_name = 'Sales';
  DBMS_OUTPUT.PUTLINE(...);

END;
```

Summary

- In this lesson, you should have learned how to:
 - List examples of good programming practices
 - Accurately insert comments into PL/SQL code
 - Create PL/SQL code that follows formatting guidelines to produce readable code

ORACLE

Academy

ORACLE

Academy

Database Programming with PL/SQL

2-7

Good Programming Practices

ORACLE
Academy



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Objectives

- This lesson covers the following objectives:
 - List examples of good programming practices
 - Accurately insert comments into PL/SQL code
 - Create PL/SQL code that follows formatting guidelines to produce readable code

Purpose

- Good programming practices are techniques that you can follow to create the best code possible
- Programming practices cover everything from making code more readable to creating code with faster performance
- Software engineering teams often follow a style guide so that everyone on the team uses the same techniques
- This makes it easier to read and modify code written by others
- Course work should follow the conventions demonstrated in this lesson

Good Programming Practices

- Several examples of good programming practices have already been demonstrated and/or discussed in this course:
 - Use explicit data type conversions because implicit data type conversions can be slower and the rules can change in later software releases
 - Use meaningful identifiers when declaring variables, constants, and parameters
 - Declare one variable or constant identifier per line for better readability and code maintenance

Good Programming Practices

- Other good programming practices demonstrated and/or discussed:
 - Avoid ambiguity when choosing identifiers
 - Use the %TYPE attribute to declare a variable according to another previously declared variable or database column
 - Use the NOT NULL constraint when declaring a variable that must hold a value

Programming Guidelines

- Other programming guidelines include:
 - Documenting code with comments
 - Developing a case convention for the code
 - Developing naming conventions for identifiers and other objects
 - Enhancing readability by indenting



Each organization will typically develop and require its own programming guidelines and conventions.

In a sense, it doesn't matter which conventions are adopted, so long as a meaningful convention exists and is used consistently.

Commenting Code

- Comments assist in future maintenance or modification by helping other programmers know what the original programmer intended by the code written
- Even the original programmer will benefit from commenting his/her code when returning for the first time to that code six months hence
- Comments are strictly informational and do not enforce any conditions or behavior on logic or data. They are ignored when code is compiled

Commenting Code Example

- Prefix single-line comments with two dashes (--)
- Place multiple-line comments between the symbols "/* " and " */"

```
DECLARE
  -- Created by Clara Oswald
  ...
  v_annual_sal NUMBER (9,2);

BEGIN      -- Start of executable section

  /* Compute the annual salary based on the monthly
   salary input from the user */

  v_annual_sal := v_monthly_sal * 12;
  ...
END;        -- End of executable section
```

Variable Case Conventions

- Case Conventions are shown below
- The following table provides guidelines for writing code in uppercase and lowercase to help you distinguish keywords from named objects

Category	Case Convention	Examples
SQL keywords	Uppercase	SELECT, INSERT
PL/SQL keywords	Uppercase	DECLARE, BEGIN, IF
Data types	Uppercase	VARCHAR2, BOOLEAN
Identifiers (variables, etc.)	Lowercase	v_salary, emp_cursor, c_tax_rate, p_empno
Tables and columns	Lowercase	employees, dept_id, salary, hire_date

The case convention described here is commonly used in SQL and PL/SQL, and is also the one used in the Oracle product documentation.

Naming Conventions

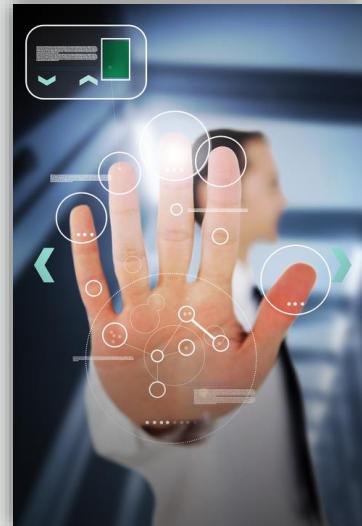
- The naming of identifiers should be clear, consistent, and unambiguous
- One commonly-used convention is to name:
 - Variables starting with v_
 - Constants starting with c_
 - Parameters starting with p_ (for passing to procedures and functions)



Naming Conventions

- Examples:

- v_date_of_birth
- v_last_name
- c_tax_rate
- c_commission_rate
- p_employee_id
- p_salary



Indenting Code

- For clarity, indent each level of code. Examples:

```
BEGIN
  IF x = 0 THEN
    y := 1;
  END IF;
END;
```

```
DECLARE
  v_deptno  NUMBER(4);
  v_location_id NUMBER(4);
BEGIN
  SELECT department_id, location_id
    INTO v_deptno, v_location_id
    FROM departments
   WHERE department_name = 'Sales';
  DBMS_OUTPUT.PUTLINE(...);
END;
```



Academy

PLSQL 2-7
Good Programming Practices

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

13

For clarity, and to enhance readability, indent each level of code using spaces or tabs, put a space before and after operators, and begin new clauses on a new line by using carriage returns. Compare the following IF statements for readability:

OPTION #1:

```
BEGIN IF x>y THEN v_max:=x;ELSE v_max:=y;END IF;END;
```

OPTION #2:

```
BEGIN
  IF x > y THEN
    v_max := x;
  ELSE
    v_max := y;
  END IF;
END;
```

With Option #2, it is much easier to see what is happening in the code. Depending on the code editor you are using, each new level of code should be indented two spaces or one tab stop.

Summary

- In this lesson, you should have learned how to:
 - List examples of good programming practices
 - Accurately insert comments into PL/SQL code
 - Create PL/SQL code that follows formatting guidelines to produce readable code

ORACLE

Academy

ORACLE

Academy

Database Programming with PL/SQL

2-6

Nested Blocks and Variable Scope

ORACLE
Academy



Objectives

- This lesson covers the following objectives:
 - Understand the scope and visibility of variables
 - Write nested blocks and qualify variables with labels
 - Describe the rules for variable scope when a variable is nested in a block
 - Recognize a variable scope issue when a variable is used in nested blocks
 - Qualify a variable nested in a block with a label

Purpose

- A large, complex block can be hard to understand
- You can break it down into smaller blocks that are nested one inside the other, making the code easier to read and correct
- When you nest blocks, declared variables might not be available depending on their scope and visibility
- You can make invisible variables available by using block labels



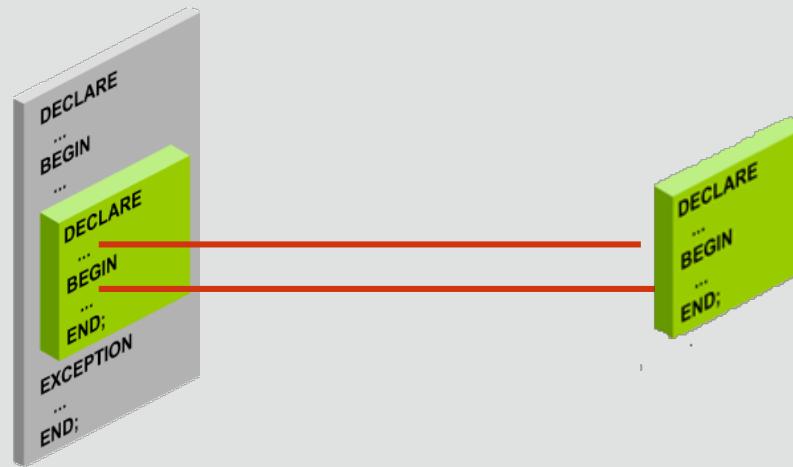
Nested Blocks

- PL/SQL is a block-structured language
- The basic units (procedures, functions, and anonymous blocks) are logical blocks, which can contain any number of nested sub-blocks
- Each logical block corresponds to a problem to be solved



Nested Blocks Illustrated

- Nested blocks are blocks of code placed within other blocks of code
- There is an outer block and an inner block
- You can nest blocks within blocks as many times as you need to; there is no practical limit to the depth of nesting Oracle allows



Nested Block Example

- The example shown in the slide has an outer (parent) block (illustrated in blue text) and a nested (child) block (illustrated in red text)
- The variable `v_outer_variable` is declared in the outer block and the variable `v_inner_variable` is declared in the inner block

Nested Block Example

```
DECLARE
  v_outer_variable VARCHAR2(20):='GLOBAL VARIABLE';
BEGIN
  DECLARE
    v_inner_variable VARCHAR2(20):='LOCAL VARIABLE';
  BEGIN
    DBMS_OUTPUT.PUT_LINE(v_inner_variable);
    DBMS_OUTPUT.PUT_LINE(v_outer_variable);
  END;
  DBMS_OUTPUT.PUT_LINE(v_outer_variable);
END;
```

Variable Scope

- The scope of a variable is the block or blocks in which the variable is accessible, that is, where it can be used
- In PL/SQL, a variable's scope is the block in which it is declared plus all blocks nested within the declaring block

Variable Scope

- What are the scopes of the two variables declared in this example?

```
DECLARE
  v_outer_variable VARCHAR2(20):='GLOBAL VARIABLE';
BEGIN
  DECLARE
    v_inner_variable VARCHAR2(20):='LOCAL VARIABLE';
  BEGIN
    DBMS_OUTPUT.PUT_LINE(v_inner_variable);
    DBMS_OUTPUT.PUT_LINE(v_outer_variable);
  END;
  DBMS_OUTPUT.PUT_LINE(v_outer_variable);
END;
```

Variable Scope Example

- Examine the following code
- What is the scope of each of the variables?

```
DECLARE
  v_father_name  VARCHAR2(20) := 'Patrick';
  v_date_of_birth DATE := '20-Apr-1972';
BEGIN
  DECLARE
    v_child_name VARCHAR2(20) := 'Mike';
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Father''s Name: ' || v_father_name);
    DBMS_OUTPUT.PUT_LINE('Date of Birth: ' || v_date_of_birth);
    DBMS_OUTPUT.PUT_LINE('Child''s Name: ' || v_child_name);
  END;
  DBMS_OUTPUT.PUT_LINE('Date of Birth: ' || v_date_of_birth);
END;
```

Local and Global Variables

- Variables declared in a PL/SQL block are considered local to that block and global to all blocks nested within it
- `v_outer_variable` is local to the outer block but global to the inner block

```
DECLARE
  v_outer_variable  VARCHAR2(20):='GLOBAL VARIABLE';
BEGIN
  DECLARE
    v_inner_variable VARCHAR2(20):='LOCAL VARIABLE';
  BEGIN
    DBMS_OUTPUT.PUT_LINE(v_inner_variable);
    DBMS_OUTPUT.PUT_LINE(v_outer_variable);
  END;
  DBMS_OUTPUT.PUT_LINE(v_outer_variable);
END;
```

Local and Global Variables

- When you access this variable in the inner block, PL/SQL first looks for a local variable in the inner block with that name
- If there are no similarly named variables, PL/SQL looks for the variable in the outer block

```
DECLARE
  v_outer_variable  VARCHAR2(20) := 'GLOBAL VARIABLE';
BEGIN
  DECLARE
    v_inner_variable VARCHAR2(20) := 'LOCAL VARIABLE';
  BEGIN
    DBMS_OUTPUT.PUT_LINE(v_inner_variable);
    DBMS_OUTPUT.PUT_LINE(v_outer_variable);
  END;
  DBMS_OUTPUT.PUT_LINE(v_outer_variable);
END;
```

Local and Global Variables

- The v_inner_variable variable is local to the inner block and is not global because the inner block does not have any nested blocks
- This variable can be accessed only within the inner block

```
DECLARE
  v_outer_variable  VARCHAR2(20):='GLOBAL VARIABLE';
BEGIN
  DECLARE
    v_inner_variable  VARCHAR2(20):='LOCAL VARIABLE';
  BEGIN
    DBMS_OUTPUT.PUT_LINE(v_inner_variable);
    DBMS_OUTPUT.PUT_LINE(v_outer_variable);
  END;
  DBMS_OUTPUT.PUT_LINE(v_outer_variable);
END;
```

Local and Global Variables

- If PL/SQL does not find the variable declared locally, it looks upward in the declarative section of the parent blocks
- PL/SQL does not look downward into the child blocks

```
DECLARE
  v_outer_variable  VARCHAR2(20):='GLOBAL VARIABLE';
BEGIN
  DECLARE
    v_inner_variable VARCHAR2(20):='LOCAL VARIABLE';
  BEGIN
    DBMS_OUTPUT.PUT_LINE(v_inner_variable);
    DBMS_OUTPUT.PUT_LINE(v_outer_variable);
  END;
  DBMS_OUTPUT.PUT_LINE(v_outer_variable);
END;
```

Variable Scope Accessible to Outer Block

- The variables `v_father_name` and `v_date_of_birth` are declared in the outer block
- They are local to the outer block and global to the inner block
- Their scope includes both blocks

```
DECLARE
  v_father_name  VARCHAR2(20):='Patrick';
  v_date_of_birth DATE:='20-Apr-1972';
BEGIN
  DECLARE
    v_child_name VARCHAR2(20):='Mike';
  ...

```

Variable Scope Accessible to Outer Block

- The variable `v_child_name` is declared in the inner (nested) block
- This variable is accessible only within the inner block and is not accessible in the outer block

```
DECLARE
  v_father_name  VARCHAR2(20):='Patrick';
  v_date_of_birth DATE:='20-Apr-1972';
BEGIN
  DECLARE
    v_child_name VARCHAR2(20):='Mike';
  ...

```

A Scoping Example

- Why will this code not work correctly?

```
DECLARE
    v_first_name      VARCHAR2( 20 );
BEGIN
    DECLARE
        v_last_name      VARCHAR2( 20 );
    BEGIN
        v_first_name := 'Carmen';
        v_last_name := 'Miranda';
        DBMS_OUTPUT.PUT_LINE
            (v_first_name || ' ' || v_last_name);
    END;
    DBMS_OUTPUT.PUT_LINE
        (v_first_name || ' ' || v_last_name);
END;
```

A Second Scoping Example

- Will this code work correctly? Why or why not?

```
DECLARE
    v_first_name    VARCHAR2( 20 );
    v_last_name     VARCHAR2( 20 );
BEGIN
    BEGIN
        v_first_name := 'Carmen';
        v_last_name  := 'Miranda';
        DBMS_OUTPUT.PUT_LINE
            (v_first_name || ' ' || v_last_name);
    END;
    DBMS_OUTPUT.PUT_LINE
        (v_first_name || ' ' || v_last_name);
END;
```

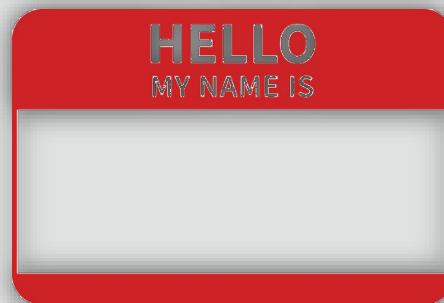
Three Levels of Nested Block

- What is the scope of each of these variables?

```
DECLARE          -- outer block
    v_outervar VARCHAR2(20);
BEGIN
    DECLARE          -- middle block
        v_middlevar VARCHAR2(20);
    BEGIN
        BEGIN          -- inner block
            v_outervar := 'Joachim';
            v_middlevar := 'Chang';
        END;
    END;
END;
```

Variable Naming

- You cannot declare two variables with the same name in the same block
- However, you can declare variables with the same name in two different blocks when one block is nested within the other block
- The two items represented by the same name are distinct, and any change in one does not affect the other



Example of Variable Naming

- Are the following declarations valid?

```
DECLARE          -- outer block
    v_myvar      VARCHAR2(20);
BEGIN
DECLARE          -- inner block
    v_myvar      VARCHAR2(15);
BEGIN
    ...
END;
END;
```

Variable Visibility

- What if the same name is used for two variables, one in each of the blocks?
- In this example, the variable `v_date_of_birth` is declared twice

```
DECLARE
  v_father_name  VARCHAR2(20):='Patrick';
  v_date_of_birth DATE:='20-Apr-1972';
BEGIN
  DECLARE
    v_child_name  VARCHAR2(20):='Mike';
    v_date_of_birth DATE:='12-Dec-2002';
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Date of Birth:' ||
      v_date_of_birth);
  ...

```



Variable Visibility

- Which v_date_of_birth is referenced in the DBMS_OUTPUT.PUT_LINE statement?

```
DECLARE
  v_father_name  VARCHAR2(20):='Patrick';
  v_date_of_birth DATE:='20-Apr-1972';
BEGIN
  DECLARE
    v_child_name  VARCHAR2(20):='Mike';
    v_date_of_birth DATE:='12-Dec-2002';
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Date of Birth:' ||
      v_date_of_birth);
  ...

```

Variable Visibility

- The visibility of a variable is the portion of the program where the variable can be accessed without using a qualifier
- What is the visibility of each of the variables?

```
DECLARE
  v_father_name  VARCHAR2(20):='Patrick';
  v_date_of_birth DATE:='20-Apr-1972';
BEGIN
  DECLARE
    v_child_name  VARCHAR2(20):='Mike';
    v_date_of_birth DATE:='12-Dec-2002';
    BEGIN
      1  DBMS_OUTPUT.PUT_LINE('Father''s Name: ' || v_father_name);
      DBMS_OUTPUT.PUT_LINE('Date of Birth: ' || v_date_of_birth);
      DBMS_OUTPUT.PUT_LINE('Child''s Name: ' || v_child_name);
    END;
      2  DBMS_OUTPUT.PUT_LINE('Date of Birth: ' || v_date_of_birth);
    END;
```

Variable Visibility

- The `v_date_of_birth` variable declared in the outer block has scope even in the inner block
- This variable is visible in the outer block
- However, it is not visible in the inner block because the inner block has a local variable with the same name

```
DECLARE
  v_father_name      VARCHAR2(20) := 'Patrick';
  v_date_of_birth    DATE := '20-Apr-1972';

BEGIN
  DECLARE
    v_child_name      VARCHAR2(20) := 'Mike';
    v_date_of_birth   DATE := '12-Dec-2002';
  ...

```

Variable Visibility

- The `v_father_name` variable is visible in the inner and outer blocks
- The `v_child_name` variable is visible only in the inner block
- What if you want to reference the outer block's `v_date_of_birth` within the inner block?

```
DECLARE
  v_father_name      VARCHAR2(20) := 'Patrick';
  v_date_of_birth    DATE := '20-Apr-1972';

BEGIN
  DECLARE
    v_child_name      VARCHAR2(20) := 'Mike';
    v_date_of_birth   DATE := '12-Dec-2002';
```

Qualifying an Identifier

- A qualifier is a label given to a block
- You can use this qualifier to access the variables that have scope but are not visible
- The outer block below is labeled <>outer>>

```
<>outer>>
DECLARE
  v_father_name  VARCHAR2(20):='Patrick';
  v_date_of_birth DATE:='20-Apr-1972';
BEGIN
  DECLARE
    v_child_name  VARCHAR2(20):='Mike';
    v_date_of_birth DATE:='12-Dec-2002';
  ...

```

- Each nested inner block also can be labeled

Qualifying an Identifier

- Using the outer label to qualify the v_date_of_birth identifier, you can now print the father's date of birth using code in the inner block

```
<<outer>>
DECLARE
    v_father_name      VARCHAR2(20):='Patrick';
    v_date_of_birth    DATE:='20-Apr-1972';
BEGIN
    DECLARE
        v_child_name      VARCHAR2(20):='Mike';
        v_date_of_birth   DATE:='12-Dec-2002';
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Father''s Name: ' || v_father_name);
        DBMS_OUTPUT.PUT_LINE('Date of Birth: ' || outer.v_date_of_birth);
        DBMS_OUTPUT.PUT_LINE('Child''s Name: ' || v_child_name);
        DBMS_OUTPUT.PUT_LINE('Date of Birth: ' || v_date_of_birth);
    END;
END;
```

Father's Name: Patrick
Date of Birth: 20-Apr-1972
Child's Name: Mike
Date of Birth: 12-Dec-2002

Statement processed.

Terminology

- Key terms used in this lesson included:
 - Block label
 - Variable scope
 - Variable visibility

Summary

- In this lesson, you should have learned how to:
 - Understand the scope and visibility of variables
 - Write nested blocks and qualify variables with labels
 - Describe the rules for variable scope when a variable is nested in a block
 - Recognize a variable scope issue when a variable is used in nested blocks
 - Qualify a variable nested in a block with a label

ORACLE

Academy

ORACLE

Academy

Database Programming with PL/SQL

2-6

Nested Blocks and Variable Scope

ORACLE
Academy



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Objectives

- This lesson covers the following objectives:
 - Understand the scope and visibility of variables
 - Write nested blocks and qualify variables with labels
 - Describe the rules for variable scope when a variable is nested in a block
 - Recognize a variable scope issue when a variable is used in nested blocks
 - Qualify a variable nested in a block with a label

Purpose

- A large, complex block can be hard to understand
- You can break it down into smaller blocks that are nested one inside the other, making the code easier to read and correct
- When you nest blocks, declared variables might not be available depending on their scope and visibility
- You can make invisible variables available by using block labels



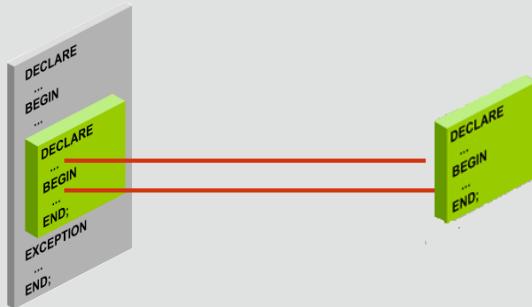
Nested Blocks

- PL/SQL is a block-structured language
- The basic units (procedures, functions, and anonymous blocks) are logical blocks, which can contain any number of nested sub-blocks
- Each logical block corresponds to a problem to be solved



Nested Blocks Illustrated

- Nested blocks are blocks of code placed within other blocks of code
- There is an outer block and an inner block
- You can nest blocks within blocks as many times as you need to; there is no practical limit to the depth of nesting Oracle allows



Nested Block Example

- The example shown in the slide has an outer (parent) block (illustrated in blue text) and a nested (child) block (illustrated in red text)
- The variable `v_outer_variable` is declared in the outer block and the variable `v_inner_variable` is declared in the inner block

Nested Block Example

```
DECLARE
  v_outer_variable VARCHAR2(20) :='GLOBAL VARIABLE';
BEGIN
  DECLARE
    v_inner_variable VARCHAR2(20) :='LOCAL VARIABLE';
  BEGIN
    DBMS_OUTPUT.PUT_LINE(v_inner_variable);
    DBMS_OUTPUT.PUT_LINE(v_outer_variable);
  END;
  DBMS_OUTPUT.PUT_LINE(v_outer_variable);
END;
```



Variable Scope

- The scope of a variable is the block or blocks in which the variable is accessible, that is, where it can be used
- In PL/SQL, a variable's scope is the block in which it is declared plus all blocks nested within the declaring block

Answer: The scope of `v_outer_variable` includes both the outer and inner blocks. The scope of `v_inner_variable` includes only the inner block. It is valid to refer to `v_outer_variable` within the inner block, but referencing `v_inner_variable` within the outer block would return an error.

Each block allows the grouping of logically related declarations and statements. This makes structured programming easy to use due to placing declarations close to where they are used (in each block).

Variable Scope

- What are the scopes of the two variables declared in this example?

```
DECLARE
  v_outer_variable VARCHAR2(20) := 'GLOBAL VARIABLE';
BEGIN
  DECLARE
    v_inner_variable VARCHAR2(20) := 'LOCAL VARIABLE';
  BEGIN
    DBMS_OUTPUT.PUT_LINE(v_inner_variable);
    DBMS_OUTPUT.PUT_LINE(v_outer_variable);
  END;
  DBMS_OUTPUT.PUT_LINE(v_outer_variable);
END;
```

Variable Scope Example

- Examine the following code
- What is the scope of each of the variables?

```
DECLARE
  v_father_name  VARCHAR2(20) :='Patrick';
  v_date_of_birth DATE:='20-Apr-1972';
BEGIN
  DECLARE
    v_child_name VARCHAR2(20) :='Mike';
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Father''s Name: '||v_father_name);
    DBMS_OUTPUT.PUT_LINE('Date of Birth: '||v_date_of_birth);
    DBMS_OUTPUT.PUT_LINE('Child''s Name: '||v_child_name);
  END;
  DBMS_OUTPUT.PUT_LINE('Date of Birth: '||v_date_of_birth);
END;
```



Academy

PLSQL 2-6
Nested Blocks and Variable Scope

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

11

Answer: The scope of v_father_name and v_date_of_birth is both blocks (inner and outer). The scope of v_child_name is the inner block only.

The scope of a variable is the block in which it is declared plus all blocks nested within the declaring block.

Local and Global Variables

- Variables declared in a PL/SQL block are considered local to that block and global to all blocks nested within it
- V_outer_variable is local to the outer block but global to the inner block

```
DECLARE
  v_outer_variable  VARCHAR2(20) :='GLOBAL VARIABLE';
BEGIN
  DECLARE
    v_inner_variable VARCHAR2(20) :='LOCAL VARIABLE';
  BEGIN
    DBMS_OUTPUT.PUT_LINE(v_inner_variable);
    DBMS_OUTPUT.PUT_LINE(v_outer_variable);
  END;
  DBMS_OUTPUT.PUT_LINE(v_outer_variable);
END;
```



Academy

PLSQL 2-6
Nested Blocks and Variable Scope

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

12

Local and Global Variables

- When you access this variable in the inner block, PL/SQL first looks for a local variable in the inner block with that name
- If there are no similarly named variables, PL/SQL looks for the variable in the outer block

```
DECLARE
  v_outer_variable  VARCHAR2(20) :='GLOBAL VARIABLE';
BEGIN
  DECLARE
    v_inner_variable VARCHAR2(20) :='LOCAL VARIABLE';
  BEGIN
    DBMS_OUTPUT.PUT_LINE(v_inner_variable);
    DBMS_OUTPUT.PUT_LINE(v_outer_variable);
  END;
  DBMS_OUTPUT.PUT_LINE(v_outer_variable);
END;
```



Academy

PLSQL 2-6
Nested Blocks and Variable Scope

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

13

Local and Global Variables

- The v_inner_variable variable is local to the inner block and is not global because the inner block does not have any nested blocks
- This variable can be accessed only within the inner block

```
DECLARE
  v_outer_variable  VARCHAR2(20) :='GLOBAL VARIABLE';
BEGIN
  DECLARE
    v_inner_variable VARCHAR2(20) :='LOCAL VARIABLE';
  BEGIN
    DBMS_OUTPUT.PUT_LINE(v_inner_variable);
    DBMS_OUTPUT.PUT_LINE(v_outer_variable);
  END;
  DBMS_OUTPUT.PUT_LINE(v_outer_variable);
END;
```



Academy

PLSQL 2-6
Nested Blocks and Variable Scope

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

14

Local and Global Variables

- If PL/SQL does not find the variable declared locally, it looks upward in the declarative section of the parent blocks
- PL/SQL does not look downward into the child blocks

```
DECLARE
  v_outer_variable  VARCHAR2(20) :='GLOBAL VARIABLE';
BEGIN
  DECLARE
    v_inner_variable VARCHAR2(20) :='LOCAL VARIABLE';
  BEGIN
    DBMS_OUTPUT.PUT_LINE(v_inner_variable);
    DBMS_OUTPUT.PUT_LINE(v_outer_variable);
  END;
  DBMS_OUTPUT.PUT_LINE(v_outer_variable);
END;
```

Variable Scope Accessible to Outer Block

- The variables `v_father_name` and `v_date_of_birth` are declared in the outer block
- They are local to the outer block and global to the inner block
- Their scope includes both blocks

```
DECLARE
  v_father_name  VARCHAR2(20) :='Patrick';
  v_date_of_birth DATE:='20-Apr-1972';
BEGIN
  DECLARE
    v_child_name VARCHAR2(20) :='Mike';
    ...
  
```

Variable Scope Accessible to Outer Block

- The variable `v_child_name` is declared in the inner (nested) block
- This variable is accessible only within the inner block and is not accessible in the outer block

```
DECLARE
  v_father_name  VARCHAR2(20) :='Patrick';
  v_date_of_birth DATE :='20-Apr-1972';
BEGIN
  DECLARE
    v_child_name VARCHAR2(20) :='Mike';
  ...

```

A Scoping Example

- Why will this code not work correctly?

```
DECLARE
    v_first_name      VARCHAR2(20);
BEGIN
    DECLARE
        v_last_name      VARCHAR2(20);
    BEGIN
        v_first_name := 'Carmen';
        v_last_name := 'Miranda';
        DBMS_OUTPUT.PUT_LINE
            (v_first_name || ' ' || v_last_name);
    END;
    DBMS_OUTPUT.PUT_LINE
        (v_first_name || ' ' || v_last_name);
END;
```

Answer: v_last_name is defined in the inner block and is not accessible in the outer block.

A Second Scoping Example

- Will this code work correctly? Why or why not?

```
DECLARE
    v_first_name  VARCHAR2(20);
    v_last_name   VARCHAR2(20);
BEGIN
    BEGIN
        v_first_name := 'Carmen';
        v_last_name  := 'Miranda';
        DBMS_OUTPUT.PUT_LINE
            (v_first_name || ' ' || v_last_name);
    END;
    DBMS_OUTPUT.PUT_LINE
        (v_first_name || ' ' || v_last_name);
END;
```



Academy

PLSQL 2-6
Nested Blocks and Variable Scope

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

19

Answer: Yes. Both variables are defined in the outer block, so they are accessible in the inner block and the outer block.

Three Levels of Nested Block

- What is the scope of each of these variables?

```
DECLARE          -- outer block
    v_outervar      VARCHAR2(20);
BEGIN
    DECLARE          -- middle block
        v_middlevar  VARCHAR2(20);
    BEGIN
        BEGIN          -- inner block
            v_outervar := 'Joachim';
            v_middlevar := 'Chang';
        END;
    END;
END;
```



PLSQL 2-6
Nested Blocks and Variable Scope

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

20

CCCC

Variable Naming

- You cannot declare two variables with the same name in the same block
- However, you can declare variables with the same name in two different blocks when one block is nested within the other block
- The two items represented by the same name are distinct, and any change in one does not affect the other



Example of Variable Naming

- Are the following declarations valid?

```
DECLARE          -- outer block
    v_myvar      VARCHAR2(20);
BEGIN
    DECLARE      -- inner block
        v_myvar  VARCHAR2(15);
    BEGIN
        ...
    END;
END;
```

Answer: Yes, they are valid, but the code could be confusing if it needs to be modified later. This is not recommended.

Variable Visibility

- What if the same name is used for two variables, one in each of the blocks?
- In this example, the variable `v_date_of_birth` is declared twice

```
DECLARE
  v_father_name  VARCHAR2(20) :='Patrick';
  v_date_of_birth DATE:='20-Apr-1972';
BEGIN
  DECLARE
    v_child_name  VARCHAR2(20) :='Mike';
    v_date_of_birth DATE:='12-Dec-2002';
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Date of Birth:' || 
      v_date_of_birth);
  ...

```



Academy

PLSQL 2-6
Nested Blocks and Variable Scope

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

23

Variable Visibility

- Which v_date_of_birth is referenced in the DBMS_OUTPUT.PUT_LINE statement?

```
DECLARE
  v_father_name  VARCHAR2(20) :='Patrick';
  v_date_of_birth DATE:='20-Apr-1972';
BEGIN
  DECLARE
    v_child_name  VARCHAR2(20) :='Mike';
    v_date_of_birth DATE:='12-Dec-2002';
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Date of Birth:' || 
      v_date_of_birth);
  ...

```

Answer: The PUT_LINE will reference the v_date_of_birth declared in the inner block.

Variable Visibility

- The visibility of a variable is the portion of the program where the variable can be accessed without using a qualifier
- What is the visibility of each of the variables?

```
DECLARE
  v_father_name  VARCHAR2(20):='Patrick';
  v_date_of_birth DATE:='20-Apr-1972';
BEGIN
  DECLARE
    v_child_name  VARCHAR2(20):='Mike';
    v_date_of_birth DATE:='12-Dec-2002';
  BEGIN
    1 DBMS_OUTPUT.PUT_LINE('Father''s Name: ' || v_father_name);
    DBMS_OUTPUT.PUT_LINE('Date of Birth: ' || v_date_of_birth);
    DBMS_OUTPUT.PUT_LINE('Child''s Name: ' || v_child_name);
  END;
  2 DBMS_OUTPUT.PUT_LINE('Date of Birth: ' || v_date_of_birth);
END;
```



1 Observe the code in the executable section of the inner PL/SQL block. You can print the father's name, the child's name, and the child's date of birth.

Only the child's date of birth can be printed within the inner block because the father's date of birth is not visible here.

2 The father's date of birth is visible here (having now returned to the outer block) and can now be printed.

Variable Visibility

- The `v_date_of_birth` variable declared in the outer block has scope even in the inner block
- This variable is visible in the outer block
- However, it is not visible in the inner block because the inner block has a local variable with the same name

```
DECLARE
  v_father_name      VARCHAR2(20) := 'Patrick';
  v_date_of_birth    DATE := '20-Apr-1972';
BEGIN
  DECLARE
    v_child_name      VARCHAR2(20) := 'Mike';
    v_date_of_birth   DATE := '12-Dec-2002';
  ...

```



Academy

PLSQL 2-6
Nested Blocks and Variable Scope

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

26

Variable Visibility

- The `v_father_name` variable is visible in the inner and outer blocks
- The `v_child_name` variable is visible only in the inner block
- What if you want to reference the outer block's `v_date_of_birth` within the inner block?

```
DECLARE
  v_father_name      VARCHAR2(20) := 'Patrick';
  v_date_of_birth    DATE := '20-Apr-1972';
BEGIN
  DECLARE
    v_child_name      VARCHAR2(20) := 'Mike';
    v_date_of_birth   DATE := '12-Dec-2002';
```



Academy

PLSQL 2-6
Nested Blocks and Variable Scope

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

27

Qualifying an Identifier

- A qualifier is a label given to a block
- You can use this qualifier to access the variables that have scope but are not visible
- The outer block below is labeled <>outer>>

```
<<outer>>
DECLARE
  v_father_name  VARCHAR2(20) := 'Patrick';
  v_date_of_birth DATE := '20-Apr-1972';
BEGIN
  DECLARE
    v_child_name  VARCHAR2(20) := 'Mike';
    v_date_of_birth DATE := '12-Dec-2002';
  ...

```

- Each nested inner block also can be labeled

Qualifying an Identifier

- Using the outer label to qualify the `v_date_of_birth` identifier, you can now print the father's date of birth using code in the inner block

```
<<outer>>
DECLARE
  v_father_name      VARCHAR2(20) := 'Patrick';
  v_date_of_birth    DATE := '20-Apr-1972';
BEGIN
  DECLARE
    v_child_name       VARCHAR2(20) := 'Mike';
    v_date_of_birth   DATE := '12-Dec-2002';
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Father''s Name: ' || v_father_name);
    DBMS_OUTPUT.PUT_LINE('Date of Birth: ' || outer.v_date_of_birth);
    DBMS_OUTPUT.PUT_LINE('Child''s Name: ' || v_child_name);
    DBMS_OUTPUT.PUT_LINE('Date of Birth: ' || v_date_of_birth);
  END;
END;
```

Father's Name: Patrick
Date of Birth: 20-Apr-1972
Child's Name: Mike
Date of Birth: 12-Dec-2002

Statement processed.



Terminology

- Key terms used in this lesson included:
 - Block label
 - Variable scope
 - Variable visibility

- Block label – A name given to a block of code which allows references to be made back to the variables and their values within that block of code from other blocks of code.
- Variable scope – Consists of all the blocks in which the variable is either local (the declaring block) or global (nested blocks within the declaring block).
- Variable visibility – The portion of the program where the variable can be accessed without using a qualifier.

Summary

- In this lesson, you should have learned how to:
 - Understand the scope and visibility of variables
 - Write nested blocks and qualify variables with labels
 - Describe the rules for variable scope when a variable is nested in a block
 - Recognize a variable scope issue when a variable is used in nested blocks
 - Qualify a variable nested in a block with a label

ORACLE

Academy

ORACLE

Academy

Database Programming with PL/SQL

2-3

Recognizing Data Types

ORACLE
Academy



Objectives

- This lesson covers the following objectives:
 - Define data type and explain why it is needed
 - List and describe categories of data types
 - Give examples of scalar and composite data types

Purpose

- As a result of the Database Programming with SQL course, you should be familiar with several data types that were used when defining the type of data stored in the different columns of a table (NUMBER, VARCHAR2, DATE, etc.)
- PL/SQL includes a variety of data types for use when defining variables, constants, and parameters
- As with table columns, these data types specify what type and size of data can be stored in a particular location

PL/SQL Data Types

- PL/SQL supports five categories of data type
- A data type specifies a storage format, constraints, and a valid range of values

Data Type	Description
Scalar	Holds a single value with no internal elements.
Composite	Contains multiple internal elements that can be manipulated individually.
Large Object (LOB)	Holds values called locators that specify the location of large objects (such as graphic images) that are stored out of line.
Reference	Holds values called pointers that point to a storage location.
Object	Is a schema object with a name, attributes, and methods. An object data type is similar to the class mechanism supported by C++ and Java.

Scalar Data Types

- Scalar data types:
 - Hold a single value
 - Have no internal components
- Can be classified into four categories:
 - Character
 - Number
 - Date
 - Boolean
- Note : The BOOLEAN data type may be new to you, as it is not a column data type used in table definitions



Scalar Data Types

- The different data types specify:
 - What type and size of data can be stored in a particular location
 - The range of values the variable can have
 - The set of operations that can be applied to values of that type

Scalar Data Types

- The Character, Number, and Date categories include several data types
- SCALAR CHARACTER data types:
 - CHAR
 - VARCHAR2
 - LONG
- SCALAR NUMERIC data types:
 - NUMBER
 - PLS_INTEGER
- SCALAR DATE data types:
 - DATE
 - TIMESTAMP
 - TIMESTAMP WITH TIME ZONE

Note : For more information and the complete list of scalar data types, refer to the PL/SQL User's Guide and Reference.

Scalar Data Types: Character (or String)

- Character data types also are known as strings and allow storage of alphanumeric data (letters, numbers, and symbols)

Data Type	Description
CHAR [(maximum_length)]	Base type for fixed-length character data up to 32,767 characters. If you do not specify a maximum_length, the default length is set to 1.
VARCHAR2 (maximum_length)	Base type for variable-length character data up to 32,767 characters. VARCHAR2 is optimized for performance or efficiency, depending on the size.
LONG	Character data of variable length up to 2 gigabytes size.

- Ex. `v_first_name VARCHAR2(20) := 'Neena';`

Scalar Data Types: Number

- Number data types allow storage of integers, decimals, and a positive or negative indicator

Data Type	Description
NUMBER	Floating-point number from 1E-130 to 10E125.
NUMBER(p,s)	Fixed-point number with precision p. Precision includes scale s and can range from 1 to 38. Scale can range from -84 to 127 and determines where rounding occurs as well as the fixed number of decimal places to store.
NUMBER(p)	Integers with maximum number of digits p (range 1-38).
PLS_INTEGER	Requires less storage and is faster than NUMBER.

- Ex. `v_salary NUMBER(8,2) := 9999.99;`

Scalar Data Types: Date

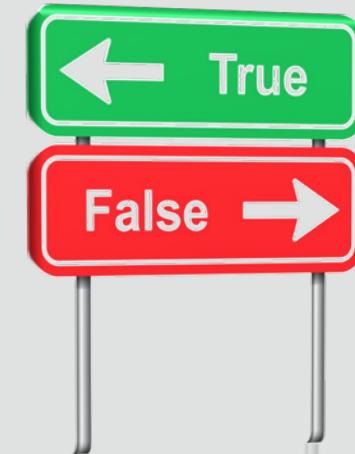
- Date data types provide storage of dates and times

Data Type	Description
DATE	Base type for dates and times. DATE values include the time of day in seconds since midnight. The range for dates is between 1-Jan-4712 BCE and 31-Dec-9999 CE.
TIMESTAMP	TIMESTAMP extends the DATE data type to store the year, month, day, hour, minute, second, and fraction of seconds.
TIMESTAMP WITH TIME ZONE	TIMESTAMP WITH TIME ZONE extends the TIMESTAMP data type to include a time-zone displacement—that is, the difference (in hours and minutes) between local time and Coordinated Universal Time (UTC).

- Ex. `v_hire_date DATE := '15-Apr-2015' ;`

Scalar Data Types: Boolean

- Use the BOOLEAN data type to store the logical values TRUE, FALSE, and NULL
- Only logic operations are allowed on BOOLEAN variables
- Column values cannot be fetched into a BOOLEAN variable and a table column cannot be defined with a BOOLEAN data type
- Ex. `v_control BOOLEAN := TRUE;`
- The BOOLEAN data type is available in PL/SQL, but is not valid in SQL



Composite Data Types

- Composite data types have internal components, sometimes called elements, that can be manipulated individually
- It may be helpful to think of a scalar type as being like a single column value in a table, while a record data type is like a whole row of a table
- Composite data types include the following:
 - RECORD
 - TABLE
 - VARRAY

Record Composite Data Type

- A composite variable that contains internal components that match the data structure of the EMPLOYEES table can be created using:

```
v_emp_record    employees%ROWTYPE;
```

- The internal elements can be referenced by prefixing the column-name with the record-name:

```
v_emp_record.first_name
```

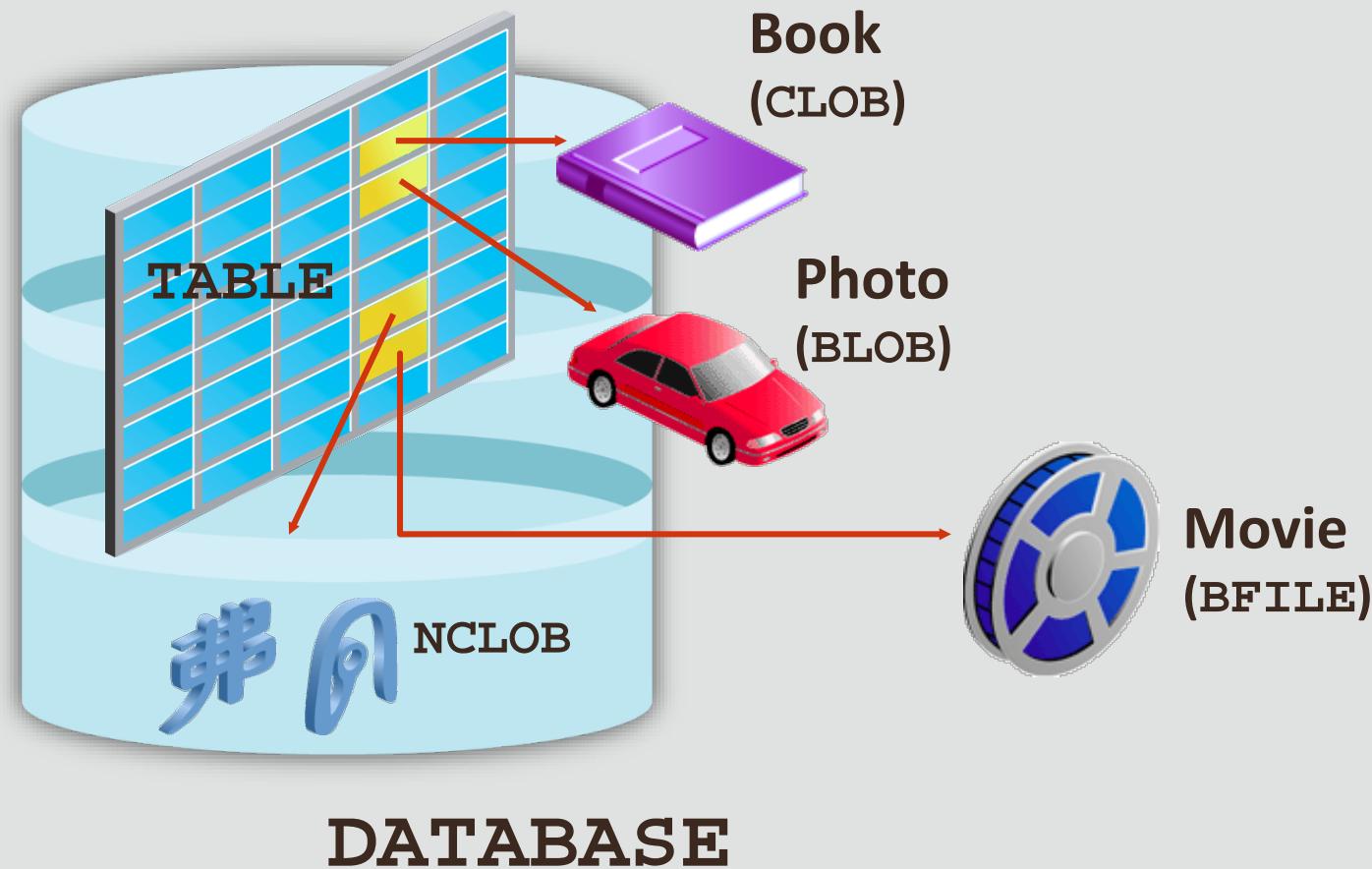
LOB (Large Object) Data Type

- LOB data types allow you to store blocks of unstructured data (such as text, graphic images, video, or audio) up to 4 gigabytes in size
- A database column can be a LOB data type
- There are four LOB data types:
 - Character large object (CLOB)
 - Binary large object (BLOB)
 - Binary file (BFILE)
 - National language character large object (NCLOB)

LOB Data Type

- LOB data types store locators, which point to large objects stored in an external file
- LOB data types allow efficient, random, piece-wise access to the data
- CLOB, BLOB, and NCLOB data is stored in the database, either inside or outside of the row
- BFILE data is stored in operating system files outside the database.

LOB Data Type



Terminology

- Key terms used in this lesson included:
 - BFILE
 - BLOB
 - CLOB
 - Composite
 - LOB

Terminology

- Key terms used in this lesson included:
 - NCLOB
 - Object
 - Reference
 - Scalar

Summary

- In this lesson, you should have learned how to:
 - Define data type and explain why it is needed
 - List and describe categories of data types
 - Give examples of scalar and composite data types

ORACLE

Academy

ORACLE

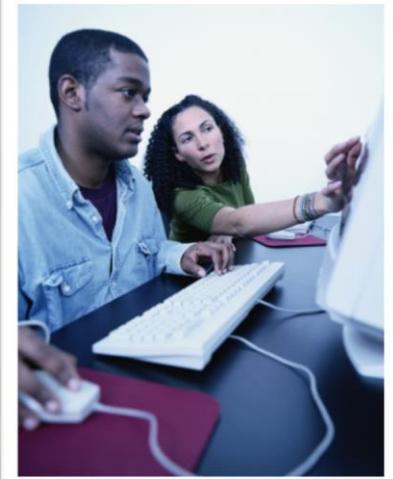
Academy

Database Programming with PL/SQL

2-3

Recognizing Data Types

ORACLE
Academy



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Objectives

- This lesson covers the following objectives:
 - Define data type and explain why it is needed
 - List and describe categories of data types
 - Give examples of scalar and composite data types

Purpose

- As a result of the Database Programming with SQL course, you should be familiar with several data types that were used when defining the type of data stored in the different columns of a table (NUMBER, VARCHAR2, DATE, etc.)
- PL/SQL includes a variety of data types for use when defining variables, constants, and parameters
- As with table columns, these data types specify what type and size of data can be stored in a particular location

PL/SQL Data Types

- PL/SQL supports five categories of data type
- A data type specifies a storage format, constraints, and a valid range of values

Data Type	Description
Scalar	Holds a single value with no internal elements.
Composite	Contains multiple internal elements that can be manipulated individually.
Large Object (LOB)	Holds values called locators that specify the location of large objects (such as graphic images) that are stored out of line.
Reference	Holds values called pointers that point to a storage location.
Object	Is a schema object with a name, attributes, and methods. An object data type is similar to the class mechanism supported by C++ and Java.

Scalar, Composite, and LOB data types are covered in this course.

Before a variable, constant, or parameter can be referenced in a PL/SQL program, it must be defined using a data type in the declarative section of a PL/SQL program.

Scalar Data Types

- Scalar data types:
 - Hold a single value
 - Have no internal components
- Can be classified into four categories:
 - Character
 - Number
 - Date
 - Boolean
- Note : The BOOLEAN data type may be new to you, as it is not a column data type used in table definitions



Scalar Data Types

- The different data types specify:
 - What type and size of data can be stored in a particular location
 - The range of values the variable can have
 - The set of operations that can be applied to values of that type

Scalar Data Types

- The Character, Number, and Date categories include several data types
- SCALAR CHARACTER data types:
 - CHAR
 - VARCHAR2
 - LONG
- SCALAR NUMERIC data types:
 - NUMBER
 - PLS_INTEGER
- SCALAR DATE data types:
 - DATE
 - TIMESTAMP
 - TIMESTAMP WITH TIME ZONE

Note : For more information and the complete list of scalar data types, refer to the PL/SQL User's Guide and Reference.

Scalar Data Types: Character (or String)

- Character data types also are known as strings and allow storage of alphanumeric data (letters, numbers, and symbols)

Data Type	Description
CHAR [(maximum_length)]	Base type for fixed-length character data up to 32,767 characters. If you do not specify a maximum_length, the default length is set to 1.
VARCHAR2 (maximum_length)	Base type for variable-length character data up to 32,767 characters. VARCHAR2 is optimized for performance or efficiency, depending on the size.
LONG	Character data of variable length up to 2 gigabytes size.

- Ex. `v_first_name VARCHAR2(20) := 'Neena' ;`

Scalar Data Types: Number

- Number data types allow storage of integers, decimals, and a positive or negative indicator

Data Type	Description
NUMBER	Floating-point number from 1E-130 to 10E125.
NUMBER(p,s)	Fixed-point number with precision p. Precision includes scale s and can range from 1 to 38. Scale can range from -84 to 127 and determines where rounding occurs as well as the fixed number of decimal places to store.
NUMBER(p)	Integers with maximum number of digits p (range 1-38).
PLS_INTEGER	Requires less storage and is faster than NUMBER.

- Ex. `v_salary NUMBER(8,2) := 9999.99;`

Scalar Data Types: Date

- Date data types provide storage of dates and times

Data Type	Description
DATE	Base type for dates and times. DATE values include the time of day in seconds since midnight. The range for dates is between 1-Jan-4712 BCE and 31-Dec-9999 CE.
TIMESTAMP	TIMESTAMP extends the DATE data type to store the year, month, day, hour, minute, second, and fraction of seconds.
TIMESTAMP WITH TIME ZONE	TIMESTAMP WITH TIME ZONE extends the TIMESTAMP data type to include a time-zone displacement—that is, the difference (in hours and minutes) between local time and Coordinated Universal Time (UTC).

- Ex. `v_hire_date DATE := '15-Apr-2015' ;`



TIMESTAMP WITH LOCAL TIME ZONE - This data type differs from TIMESTAMP WITH TIME ZONE in that when you insert a value into a database column, the value is normalized to the database time zone, and the time-zone displacement is not stored in the column. When you retrieve the value, the Oracle server returns the value in your local session time zone.

INTERVAL YEAR TO MONTH - Use the INTERVAL YEAR TO MONTH data type to store and manipulate intervals of years and months.

INTERVAL DAY TO SECOND - Use the INTERVAL DAY TO SECOND data type to store and manipulate intervals of days, hours, minutes, and seconds.

Scalar Data Types: Boolean

- Use the BOOLEAN data type to store the logical values TRUE, FALSE, and NULL
- Only logic operations are allowed on BOOLEAN variables
- Column values cannot be fetched into a BOOLEAN variable and a table column cannot be defined with a BOOLEAN data type
- Ex. `v_control BOOLEAN := TRUE;`
- The BOOLEAN data type is available in PL/SQL, but is not valid in SQL



Composite Data Types

- Composite data types have internal components, sometimes called elements, that can be manipulated individually
- It may be helpful to think of a scalar type as being like a single column value in a table, while a record data type is like a whole row of a table
- Composite data types include the following:
 - RECORD
 - TABLE
 - VARRAY

RECORD and TABLE data types are covered later in this course.

Record Composite Data Type

- A composite variable that contains internal components that match the data structure of the EMPLOYEES table can be created using:

```
v_emp_record    employees%ROWTYPE;
```

- The internal elements can be referenced by prefixing the column-name with the record-name:

```
v_emp_record.first_name
```

LOB (Large Object) Data Type

- LOB data types allow you to store blocks of unstructured data (such as text, graphic images, video, or audio) up to 4 gigabytes in size
- A database column can be a LOB data type
- There are four LOB data types:
 - Character large object (CLOB)
 - Binary large object (BLOB)
 - Binary file (BFILE)
 - National language character large object (NCLOB)

LOB Data Type

- LOB data types store locators, which point to large objects stored in an external file
- LOB data types allow efficient, random, piece-wise access to the data
- CLOB, BLOB, and NCLOB data is stored in the database, either inside or outside of the row
- BFILE data is stored in operating system files outside the database.

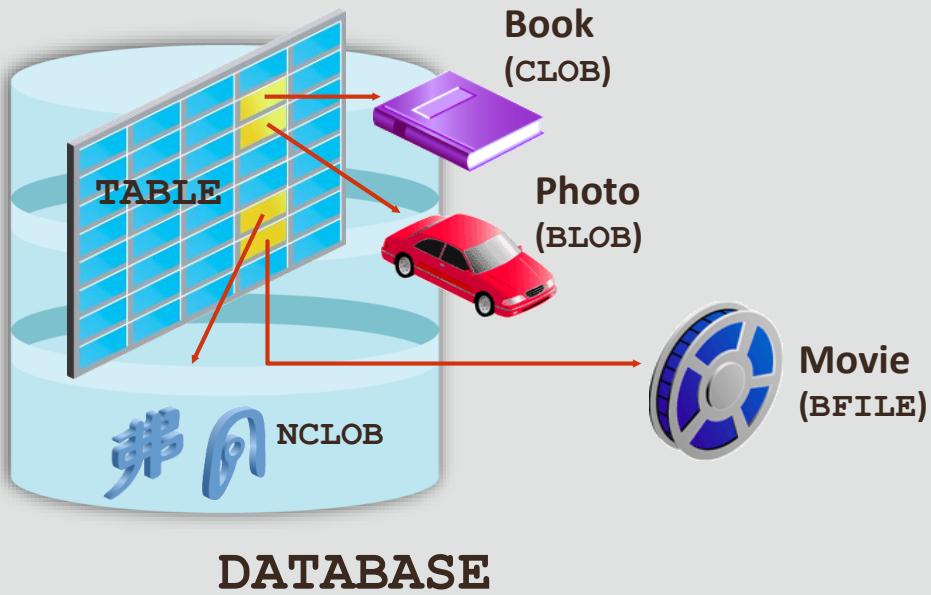
The character large object (CLOB) data type is used to store large blocks of text data in the database.

The binary large object (BLOB) data type is used to store large binary objects (ex. graphic image) in the database. When you insert or retrieve such data to and from the database, the database does not interpret the data. External applications that use this data must interpret the data.

The national language character large object (NCLOB) data type is used to store large blocks of NCHAR data in the database.

The binary file (BFILE) data type is used to store large binary objects (ex. a video or audio file) in operating system files outside the database.

LOB Data Type



ORACLE
Academy

PLSQL 2-3
Recognizing Data Types

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

17

BLOB, CLOB, and NCLOB objects can be stored in-line (inside the row) or out-of-line (outside the row, but still in the database, as indicated in the image).

BFILE objects are stored outside the database in operating system files.

Terminology

- Key terms used in this lesson included:
 - BFILE
 - BLOB
 - CLOB
 - Composite
 - LOB

- BFILE – Store large binary objects outside of the database.
- BLOB – Store large binary objects in the database.
- CLOB – Store large blocks of character data in the database.
- Composite – Contain internal elements that can be manipulated individually.
- LOB – Data types that hold values called locators, which specify the location of large objects (such as graphic images).

Terminology

- Key terms used in this lesson included:
 - NCLOB
 - Object
 - Reference
 - Scalar

- NCLOB (National Language Character Large Object) – Store large blocks of NCHAR data in the database.
- Object – A schema object with a name, attributes, and methods.
- Reference – Hold values, called pointers, which point to a storage location.
- Scalar – Hold a single value with no internal components.

Summary

- In this lesson, you should have learned how to:
 - Define data type and explain why it is needed
 - List and describe categories of data types
 - Give examples of scalar and composite data types

ORACLE

Academy

ORACLE

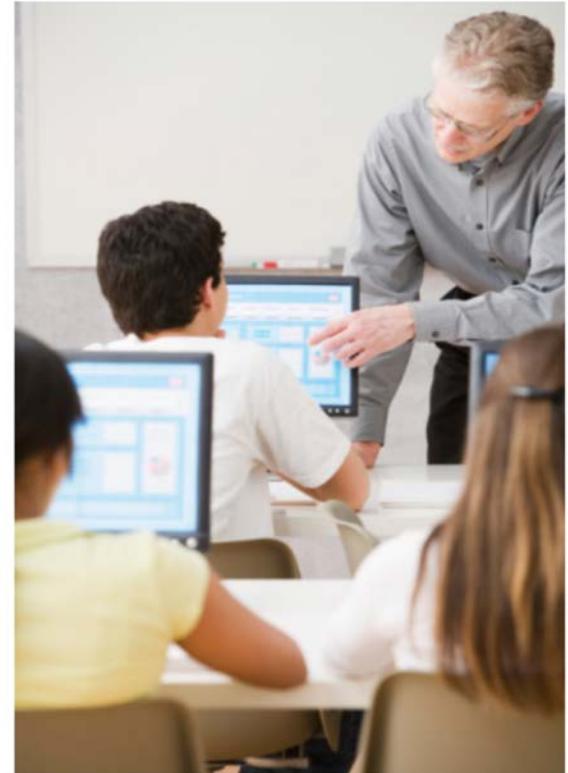
Academy

Database Programming with PL/SQL

2-2

Recognizing PL/SQL Lexical Units

ORACLE
Academy



Objectives

- This lesson covers the following objectives:
 - List and define the different types of lexical units available in PL/SQL
 - Describe identifiers and identify valid and invalid identifiers in PL/SQL
 - Describe and identify reserved words, delimiters, literals, and comments in PL/SQL

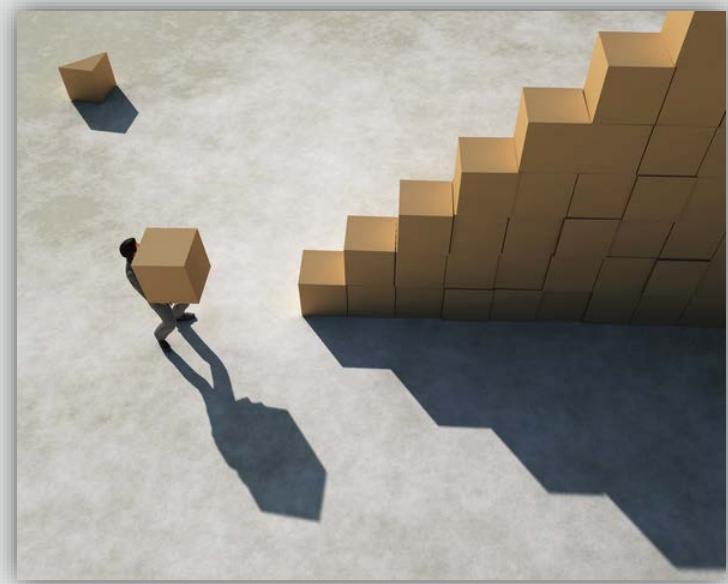
Purpose

- A spoken language has different parts of speech
- Each part of speech (such as an adjective, noun, and verb) is used differently and must follow rules
- Similarly, a programming language has different parts of speech that are used differently and must follow rules
- These parts of speech are called lexical units



Lexical Units in a PL/SQL Block

- Lexical units:
 - Are the building blocks of any PL/SQL block
 - Are sequences of characters including letters, digits, tabs, returns, and symbols
- Can be classified as:
 - Identifiers
 - Reserved words
 - Delimiters
 - Literals
 - Comments



Identifiers

- An identifier is the name given to a PL/SQL object, including any of the following:

Procedure	Function	Variable
Exception	Constant	Package
Record	PL/SQL table	Cursor

- Do not be concerned if you do not know what all of the above objects are
- You will learn about PL/SQL objects throughout this course



Identifiers Highlighted

- Several identifiers are highlighted in the PL/SQL code shown below

```
CREATE PROCEDURE print_date IS  
  
    v_date VARCHAR2(30);  
  
BEGIN  
  
    SELECT TO_CHAR(SYSDATE,'Mon DD, YYYY')  
        INTO v_date  
        FROM DUAL;  
    DBMS_OUTPUT.PUT_LINE(v_date);  
  
END;
```

- Key: Variables Packages Procedures Functions

Identifier Properties

- Identifiers:
 - Maximum 30 characters in length
 - Must begin with a letter
 - May include \$ (dollar sign), _ (underscore), or # (hashtag)
 - May not contain spaces
 - Identifiers are NOT case sensitive
- Be sure to name your objects carefully
- Ideally, the identifier name should describe the object and its purpose
- Avoid using identifier names such as A, X, Y1, temp, etc., because they make your code more difficult to read



Valid and Invalid Identifiers

- Examples of valid identifiers:

<code>First_Name</code>	<code>Lastname</code>	<code>address_1</code>
<code>ID#</code>	<code>Total_ \$</code>	<code>primary_department_contact</code>

- Examples of invalid identifiers:

<code>First Name</code>	Contains a space
<code>Last-Name</code>	Contains invalid symbol "-"
<code>1st_address_line</code>	Begins with a number
<code>Total_%</code>	Contains invalid symbol "%"
<code>primary_building_department_contact</code>	More than 30 characters

Reserved Words

- Reserved words are words that have special meaning to the Oracle database
- Reserved words cannot be used as identifiers in a PL/SQL program



Partial List of Reserved Words

- The following is a partial list of reserved words

ALL	CREATE	FROM	MODIFY	SELECT
ALTER	DATE	GROUP	NOT	SYNONYM
AND	DEFAULT	HAVING	NULL	SYSDATE
ANY	DELETE	IN	NUMBER	TABLE
AS	DESC	INDEX	OR	THEN
ASC	DISTINCT	INSERT	ORDER	UPDATE
BETWEEN	DROP	INTEGER	RENAME	VALUES
CHAR	ELSE	INTO	ROW	VARCHAR2
COLUMN	EXISTS	IS	ROWID	VIEW
COMMENT	FOR	LIKE	ROWNUM	WHERE

- Note: For more information, refer to the “PL/SQL User’s Guide and Reference”

Using Reserved Words

- What happens when you try to use a reserved word as an identifier in a PL/SQL program?

```
DECLARE
    date DATE;
BEGIN
    SELECT ADD_MONTHS(SYSDATE,3) INTO date
    FROM dual;
END;
```

```
ORA-06550: line 4, column 37:
PL/SQL: ORA-00936: missing expression
ORA-06550: line 4, column 3:
PL/SQL: SQL Statement ignored
2.      date DATE;
3. BEGIN
4.         SELECT ADD_MONTHS(SYSDATE,3) INTO date
5.         FROM DUAL;
6. END;
```

Delimiters

- Delimiters are symbols that have special meaning
- Simple delimiters consist of one character

Symbol	Meaning
+	addition operator
-	subtraction/negation operator
*	multiplication operator
/	division operator
=	equality operator
'	character string delimiter
;	statement terminator

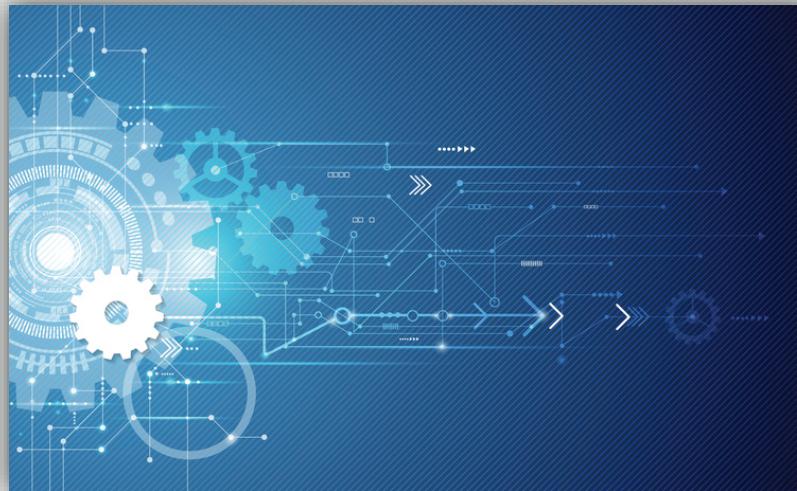
Delimiters

- Compound delimiters consist of two characters

Symbol	Meaning
<>	inequality operator
!=	inequality operator
	concatenation operator
--	single-line comment indicator
/*	beginning comment delimiter
*/	ending comment delimiter
**	exponent
:=	assignment operator

Literals

- A literal is an explicit numeric, character string, date, or Boolean value that might be stored in a variable
- Literals are classified as:
 - Character (also known as string literals)
 - Numeric
 - Boolean



Character Literals

- May include any printable character in the PL/SQL character set: letters, numerals, spaces, and symbols
- Typically defined using the VARCHAR2 data type
- Must be enclosed by character string delimiters ('')
- Can be composed of zero or more characters
- Are case sensitive; therefore, PL/SQL is NOT equivalent to pl/sql

Character Literals

- The following are examples of character literals being assigned to variables
- The literals are the characters between the single quotes (the character string delimiters) and are shown here in red text for emphasis

```
DECLARE
    v_firstname    VARCHAR2(30) := 'John';
    v_classroom    VARCHAR2(4)  := '12C';
    v_course_id    VARCHAR2(8)  := 'CS 101';
BEGIN
    ...

```

Numeric Literals

- Literals that represent numbers are numeric literals
- Numeric literals can be a simple value
(ex. 5, -32.5, 127634, 3.141592)
- Scientific notation also may be used
(ex. 2E5, meaning $2 * (10 \text{ to the power of } 5)$)
- Typically defined using the NUMBER data type



0 1 2 3 4 5 6 7 8 9 0

Numeric Literals

- The following are examples of numeric literals being assigned to variables (and one constant)
- The literals are shown here in red text for emphasis

```
DECLARE
    v_classroom    NUMBER(3) := 327;
    v_grade        NUMBER(3) := 95;
    v_price        NUMBER(5) := 150;
    v_salary       NUMBER(8) := 2E5;
    c_pi           CONSTANT NUMBER(7,6) := 3.141592;
BEGIN
    ...
END;
```



Boolean Literals

- Values that are assigned to Boolean variables are Boolean literals
- TRUE, FALSE, and NULL are the Boolean literals

```
DECLARE
    v_new_customer    BOOLEAN := FALSE;
    v_fee_paid        BOOLEAN := TRUE;
    v_diploma         BOOLEAN := NULL;
BEGIN
    ...
END;
```

- Note that character string delimiters are not required

Comments

- Comments explain what a piece of code is trying to achieve
- Well-placed comments are extremely valuable for code readability and future code maintenance
- Comments should describe the purpose and use of each block of code
- It is good programming practice to comment code
- Comments are ignored by PL/SQL
- They make no difference to how a PL/SQL block executes or the results it displays

Syntax for Commenting Code

- Two ways to indicate comments in PL/SQL
- When commenting a single line, use two dashes (--)
- When commenting multiple lines, begin the comment with /* and end the comment with */

```
DECLARE
  -- converts monthly salary to annual salary
  v_monthly_sal NUMBER(9,2);
  v_annual_sal NUMBER(9,2);
BEGIN      -- begin executable section
  ...
/* Compute the annual salary based on the
   monthly salary input from the user */
  v_annual_sal := v_monthly_sal * 12;
END;        -- end block
```

Terminology

- Key terms used in this lesson included:

- Lexical units
 - Identifiers
 - Reserved words
 - Delimiters
 - Literals
 - Comments

Summary

- In this lesson, you should have learned how to:
 - List and define the different types of lexical units available in PL/SQL
 - Describe identifiers and identify valid and invalid identifiers in PL/SQL
 - Describe and identify reserved words, delimiters, literals, and comments in PL/SQL

ORACLE

Academy

ORACLE

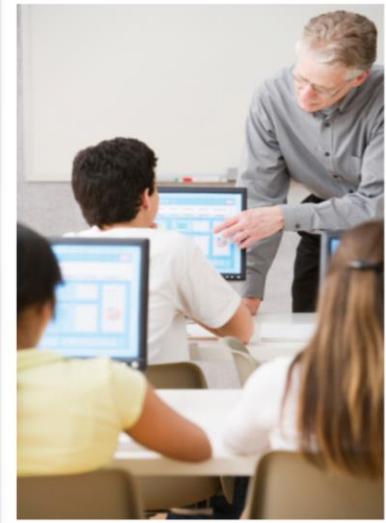
Academy

Database Programming with PL/SQL

2-2

Recognizing PL/SQL Lexical Units

ORACLE
Academy



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Objectives

- This lesson covers the following objectives:
 - List and define the different types of lexical units available in PL/SQL
 - Describe identifiers and identify valid and invalid identifiers in PL/SQL
 - Describe and identify reserved words, delimiters, literals, and comments in PL/SQL

Purpose

- A spoken language has different parts of speech
- Each part of speech (such as an adjective, noun, and verb) is used differently and must follow rules
- Similarly, a programming language has different parts of speech that are used differently and must follow rules
- These parts of speech are called lexical units

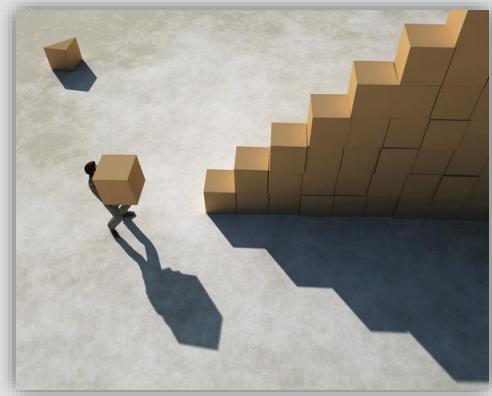
Lexical Units in a PL/SQL Block

- Lexical units:

- Are the building blocks of any PL/SQL block
- Are sequences of characters including letters, digits, tabs, returns, and symbols

- Can be classified as:

- Identifiers
- Reserved words
- Delimiters
- Literals
- Comments



- Identifiers are names for objects such as variables, functions, and procedures.
- Reserved words are words already used by PL/SQL, such as BEGIN, END, and DECLARE, and words already used by SQL, such as SELECT, INTO, FROM, etc.
- Delimiters are symbols that have special meaning (+, ;, :=, ", etc.)
- A literal is an explicit numeric, character string, date, or Boolean value (ex. a person's first name could be a literal that is stored in a variable).
- Comments explain what a piece of code is trying to achieve and are ignored when the code is processed.

Identifiers

- An identifier is the name given to a PL/SQL object, including any of the following:

Procedure	Function	Variable
Exception	Constant	Package
Record	PL/SQL table	Cursor

- Do not be concerned if you do not know what all of the above objects are
- You will learn about PL/SQL objects throughout this course

Identifiers Highlighted

- Several identifiers are highlighted in the PL/SQL code shown below

```
CREATE PROCEDURE print_date IS  
  
    v_date VARCHAR2(30);  
  
BEGIN  
  
    SELECT TO_CHAR(SYSDATE, 'Mon DD, YYYY')  
        INTO v_date  
        FROM DUAL;  
    DBMS_OUTPUT.PUT_LINE(v_date);  
  
END;
```

- Key: Variables Packages Procedures Functions



Identifiers are names for objects such as variables, functions, procedures, and packages.

The words PROCEDURE, IS, VARCHAR2, BEGIN, SELECT, INTO, FROM, DUAL, and END are reserved words.

As the course progresses, you will learn more about each of these elements.

Identifier Properties

- Identifiers:
 - Maximum 30 characters in length
 - Must begin with a letter
 - May include \$ (dollar sign), _ (underscore), or # (hashtag)
 - May not contain spaces
 - Identifiers are NOT case sensitive
- Be sure to name your objects carefully
- Ideally, the identifier name should describe the object and its purpose
- Avoid using identifier names such as A, X, Y1, temp, etc., because they make your code more difficult to read



The convention is to begin variables with v_ and constants with c_.

Identifiers are NOT case sensitive, so v_num, V_NUM, and V_Num are all the same identifier.

Valid and Invalid Identifiers

- Examples of valid identifiers:

First_Name	LastName	address_1
ID#	Total_\$	primary_department_contact

- Examples of invalid identifiers:

First Name	Contains a space
Last-Name	Contains invalid symbol "-"
1st_address_line	Begins with a number
Total_%	Contains invalid symbol "%"
primary_building_department_contact	More than 30 characters

Reserved Words

- Reserved words are words that have special meaning to the Oracle database
- Reserved words cannot be used as identifiers in a PL/SQL program



Partial List of Reserved Words

- The following is a partial list of reserved words

ALL	CREATE	FROM	MODIFY	SELECT
ALTER	DATE	GROUP	NOT	SYNONYM
AND	DEFAULT	HAVING	NULL	SYSDATE
ANY	DELETE	IN	NUMBER	TABLE
AS	DESC	INDEX	OR	THEN
ASC	DISTINCT	INSERT	ORDER	UPDATE
BETWEEN	DROP	INTEGER	RENAME	VALUES
CHAR	ELSE	INTO	ROW	VARCHAR2
COLUMN	EXISTS	IS	ROWID	VIEW
COMMENT	FOR	LIKE	ROWNUM	WHERE

- Note: For more information, refer to the “PL/SQL User’s Guide and Reference”

Using Reserved Words

- What happens when you try to use a reserved word as an identifier in a PL/SQL program?

```
DECLARE
    date DATE;
BEGIN
    SELECT ADD_MONTHS(SYSDATE, 3) INTO date
    FROM dual;
END ;
```

```
ORA-06550: line 4, column 37:
PL/SQL: ORA-00936: missing expression
ORA-06550: line 4, column 3:
PL/SQL: SQL Statement ignored
2.      date DATE;
3. BEGIN
4.         SELECT ADD_MONTHS(SYSDATE, 3) INTO date
5.         FROM DUAL;-
6. END ;
```

Delimiters

- Delimiters are symbols that have special meaning
- Simple delimiters consist of one character

Symbol	Meaning
+	addition operator
-	subtraction/negation operator
*	multiplication operator
/	division operator
=	equality operator
'	character string delimiter
;	statement terminator

You have already learned that the symbol ";" is used to terminate a SQL or PL/SQL statement. It tells the compiler that it is the end of the statement. Lines of code are not terminated at the physical end of the line. They are terminated by the semi-colon. Often a single statement is spread over several lines to make the code more readable.

Delimiters

- Compound delimiters consist of two characters

Symbol	Meaning
<>	inequality operator
!=	inequality operator
	concatenation operator
--	single-line comment indicator
/*	beginning comment delimiter
*/	ending comment delimiter
**	exponent
:=	assignment operator

The single-line comment indicator is shown with a space between the dashes for visual clarity. When used, there should not be a space between any of the characters in a compound delimiter.

Literals

- A literal is an explicit numeric, character string, date, or Boolean value that might be stored in a variable
- Literals are classified as:
 - Character (also known as string literals)
 - Numeric
 - Boolean



Character Literals

- May include any printable character in the PL/SQL character set: letters, numerals, spaces, and symbols
- Typically defined using the VARCHAR2 data type
- Must be enclosed by character string delimiters ('')
- Can be composed of zero or more characters
- Are case sensitive; therefore, PL/SQL is NOT equivalent to pl/sql

Be careful when copying code into the APEX SQL Commands window. Some word processors may default to displaying "smart quotes." Smart quotes are curly or sloped. PL/SQL code requires the use of "straight quotes."

Character Literals

- The following are examples of character literals being assigned to variables
- The literals are the characters between the single quotes (the character string delimiters) and are shown here in red text for emphasis

```
DECLARE
    v_firstname    VARCHAR2(30) := 'John';
    v_classroom    VARCHAR2(4)  := '12C';
    v_course_id    VARCHAR2(8)  := 'CS 101';
BEGIN
    ...

```

Numeric Literals

- Literals that represent numbers are numeric literals
- Numeric literals can be a simple value
(ex. 5, -32.5, 127634, 3.141592)
- Scientific notation also may be used
(ex. 2E5, meaning 2×10^5)
- Typically defined using the NUMBER data type

01234567890

Numeric Literals

- The following are examples of numeric literals being assigned to variables (and one constant)
- The literals are shown here in red text for emphasis

```
DECLARE
    v_classroom  NUMBER(3)  := 327;
    v_grade      NUMBER(3)  := 95;
    v_price      NUMBER(5)  := 150;
    v_salary     NUMBER(8)  := 2E5;
    c_pi         CONSTANT NUMBER(7, 6)  := 3.141592;
BEGIN
    ...

```

Boolean Literals

- Values that are assigned to Boolean variables are Boolean literals
- TRUE, FALSE, and NULL are the Boolean literals

```
DECLARE
    v_new_customer    BOOLEAN := FALSE;
    v_fee_paid        BOOLEAN := TRUE;
    v_diploma         BOOLEAN := NULL;
BEGIN
    ...

```

- Note that character string delimiters are not required

The idea of Boolean variables and literals may be new to students because an Oracle database table cannot contain columns with a Boolean data type. More information regarding the use of Boolean variables will be covered later in the course.

Boolean literals are NOT case sensitive, so FALSE is the same as false, TRUE is the same as true, and NULL is the same as null.

Comments

- Comments explain what a piece of code is trying to achieve
- Well-placed comments are extremely valuable for code readability and future code maintenance
- Comments should describe the purpose and use of each block of code
- It is good programming practice to comment code
- Comments are ignored by PL/SQL
- They make no difference to how a PL/SQL block executes or the results it displays

Syntax for Commenting Code

- Two ways to indicate comments in PL/SQL
- When commenting a single line, use two dashes (--)
- When commenting multiple lines, begin the comment with /* and end the comment with */

```
DECLARE
  -- converts monthly salary to annual salary
  v_monthly_sal NUMBER(9,2);
  v_annual_sal NUMBER(9,2);
BEGIN      -- begin executable section
  ...
/* Compute the annual salary based on the
   monthly salary input from the user */
  v_annual_sal := v_monthly_sal * 12;
END;        -- end block
```

This example includes thorough commenting, especially for such a simple block of code, but is shown to demonstrate various ways to add comments to your code.

Terminology

- Key terms used in this lesson included:
 - Lexical units
 - Identifiers
 - Reserved words
 - Delimiters
 - Literals
 - Comments



- Lexical Units – Building blocks of any PL/SQL block and are sequences of characters including letters, digits, tabs, returns, and symbols.
- Identifiers – A name, up to 30 characters in length, given to a PL/SQL object.
- Reserved words – Words that have special meaning to an Oracle database and cannot be used as identifiers.
- Delimiters – Symbols that have special meaning to an Oracle database.
- Literals – An explicit numeric, character string, date, or Boolean value that is not represented by an identifier.

- Comments – Describe the purpose and use of each code segment and are ignored by PL/SQL.

Summary

- In this lesson, you should have learned how to:
 - List and define the different types of lexical units available in PL/SQL
 - Describe identifiers and identify valid and invalid identifiers in PL/SQL
 - Describe and identify reserved words, delimiters, literals, and comments in PL/SQL

ORACLE

Academy

ORACLE

Academy

Database Programming with PL/SQL

2-4

Using Scalar Data Types

ORACLE
Academy



Objectives

- This lesson covers the following objectives:
 - Declare and use scalar data types in PL/SQL
 - Define guidelines for declaring and initializing PL/SQL variables
 - Identify the benefits of anchoring data types with the %TYPE attribute

Purpose

- Most of the variables you define and use in PL/SQL have scalar data types
- A variable can have an explicit data type, such as VARCHAR2, or it can automatically have the same data type as a table column in the database
- You will learn the benefits of basing some variables on table columns

Declaring Character Variables

- All variables must be declared
- The data itself will determine what data type you assign to each variable
- Commonly used character data types include CHAR and VARCHAR2
- Columns that may exceed the 32,767 character limit of a VARCHAR2 could be defined using LONG, but should be defined using CLOB

```
DECLARE
  v_country_id      CHAR(2);
  v_country_name    VARCHAR2(70);
  v_country_rpt     CLOB;
```

Declaring Number Variables

- Number data types include NUMBER, INTEGER, PLS_INTEGER, BINARY_FLOAT and several others
- Adding the keyword CONSTANT constrains the variable so that its value cannot change
- Constants must be initialized

```
DECLARE
  v_employee_id      NUMBER(6,0);
  v_loop_count        INTEGER := 0;
  c_tax_rate          CONSTANT NUMBER(3,2) := 8.25;
...

```

Declaring Date Variables

- Date data types include DATE, TIMESTAMP, and TIMESTAMP WITH TIMEZONE

```
DECLARE
    v_date1      DATE := '05-Apr-2015';
    v_date2      DATE := v_date1 + 7;
    v_date3      TIMESTAMP := SYSDATE;
    v_date4      TIMESTAMP WITH TIME ZONE := SYSDATE;
BEGIN
    DBMS_OUTPUT.PUT_LINE(v_date1);
    DBMS_OUTPUT.PUT_LINE(v_date2);
    DBMS_OUTPUT.PUT_LINE(v_date3);
    DBMS_OUTPUT.PUT_LINE(v_date4);
END;
```

- Choosing between DATE, TIMESTAMP, etc., is determined by what data you need to know in the future

Declaring BOOLEAN Variables

- BOOLEAN is a data type that stores one of the three possible values used for logical calculations: TRUE, FALSE, or NULL

```
DECLARE
    v_valid1      BOOLEAN := TRUE;
    v_valid2      BOOLEAN;
    v_valid3      BOOLEAN NOT NULL := FALSE;
BEGIN
    IF v_valid1 THEN
        DBMS_OUTPUT.PUT_LINE('Test is TRUE');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Test is FALSE');
    END IF;
END;
```

Using BOOLEAN Variables

- When using BOOLEAN variables:
 - Only the values TRUE, FALSE, and NULL can be assigned to a BOOLEAN variable
 - Conditional expressions use the logical operators AND and OR, and the operator NOT to check the variable values
 - The variables always yield TRUE, FALSE, or NULL
 - You can use arithmetic, character, and date expressions to return a BOOLEAN value



Guidelines for Declaring PL/SQL Variables

- Use meaningful and appropriate variable names
- Follow naming conventions
- Use `v_name` to represent a variable and `c_name` to represent a constant
- Declare one identifier per line for better readability, code maintenance, and easier commenting
- Use the NOT NULL constraint when the variable must hold a value
- Use the CONSTANT constraint when the variable value should not change within the block

Guidelines for Declaring PL/SQL Variables

- Set initial values for BOOLEANS and NUMBERS
- Avoid using column names as identifiers

```
DECLARE
    first_name VARCHAR2( 20 );
BEGIN
    SELECT first_name
        INTO first_name
        FROM employees
       WHERE last_name = 'Vargas';
    DBMS_OUTPUT.PUT_LINE(first_name);
END;
```

Defining Variables with the %TYPE Attribute

- Variables derived from database fields should be defined using the %TYPE attribute, which has several advantages
- For example, in the EMPLOYEES table, the column first_name is defined as VARCHAR2(20)
- In a PL/SQL block, you could define a matching variable with either:

```
v_first_name  VARCHAR2( 20 );
```

- Or

```
v_first_name  employees.last_name%TYPE;
```

Using the %TYPE Attribute

- Look at this partial table definition from the EMPLOYEES table
- Then look at the code in the next slide

Column Name	Data Type
EMPLOYEE_ID	NUMBER(6,0)
FIRST_NAME	VARCHAR2(20)
LAST_NAME	VARCHAR2(25)
EMAIL	VARCHAR2(25)

Using the %TYPE Attribute

- This PL/SQL block stores the correct first name in the `v_first_name` variable
- But what if the table column is later altered to be `VARCHAR2(25)` and a name longer than 20 characters is added?

```
DECLARE
  v_first_name      VARCHAR2(20);
BEGIN
  SELECT first_name
    INTO v_first_name
   FROM employees
  WHERE last_name = 'Vargas';
  DBMS_OUTPUT.PUT_LINE(v_first_name);
END;
```

Using the %TYPE Attribute

- Using the %TYPE attribute to define an "anchored data type" for v_first_name would solve the problem
- Otherwise, a programmer would have to find and modify every place in every program with a variable defined to hold an employee's first name

```
DECLARE
    v_first_name          employees.first_name%TYPE;
BEGIN
    SELECT first_name
        INTO v_first_name
        FROM employees
       WHERE last_name = 'Vargas';
    DBMS_OUTPUT.PUT_LINE(v_first_name);
END;
```

Using the %TYPE Attribute

- The %TYPE attribute:
 - Is used to automatically give a variable the same data type and size as:
 - A database column
 - Another declared variable
 - Is prefixed with either of the following:
 - The database table name and column name
 - The name of the other declared variable



Using the %TYPE Attribute

- Syntax:

```
identifier      table_name.column_name%TYPE;  
identifier      identifier%TYPE;
```

- Examples:

```
DECLARE  
    v_first_name      employees.first_name%TYPE;  
    v_salary          employess.salary%TYPE;  
    v_old_salary      v_salary%TYPE;  
    v_new_salary      v_salary%TYPE;  
    v_balance         NUMBER(10,2);  
    v_min_balance    v_balance%TYPE := 1000;  
    ...
```

Advantages of the %TYPE Attribute

- Advantages of the %TYPE attribute are:
 - You can avoid errors caused by data type mismatch or wrong precision
 - You need not change the variable declaration if the table column definition changes
 - Otherwise, if you have already declared some variables for a particular table column without using the %TYPE attribute, then the PL/SQL block can return errors if the table column is altered

Advantages of the %TYPE Attribute

- Advantages of the %TYPE attribute are:
 - When you use the %TYPE attribute, PL/SQL determines the data type and size of the variable when the block is compiled
 - This ensures that such a variable is always compatible with the column that is used to populate it



Terminology

- Key terms used in this lesson included:
 - %TYPE
 - BOOLEAN

Summary

- In this lesson, you should have learned how to:
 - Declare and use scalar data types in PL/SQL
 - Define guidelines for declaring and initializing PL/SQL variables
 - Identify the benefits of anchoring data types with the %TYPE attribute

ORACLE

Academy

ORACLE

Academy

Database Programming with PL/SQL

2-4

Using Scalar Data Types

ORACLE
Academy



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Objectives

- This lesson covers the following objectives:
 - Declare and use scalar data types in PL/SQL
 - Define guidelines for declaring and initializing PL/SQL variables
 - Identify the benefits of anchoring data types with the %TYPE attribute

Purpose

- Most of the variables you define and use in PL/SQL have scalar data types
- A variable can have an explicit data type, such as VARCHAR2, or it can automatically have the same data type as a table column in the database
- You will learn the benefits of basing some variables on table columns

Declaring Character Variables

- All variables must be declared
- The data itself will determine what data type you assign to each variable
- Commonly used character data types include CHAR and VARCHAR2
- Columns that may exceed the 32,767 character limit of a VARCHAR2 could be defined using LONG, but should be defined using CLOB

```
DECLARE
  v_country_id      CHAR(2);
  v_country_name    VARCHAR2(70);
  v_country_rpt     CLOB;
```



The internationally recognized two-character country abbreviations (ex. BR, CA, EG, IN, RO, UK, etc.) can be defined using CHAR(2) since they are all two-characters in length.

Since the length of country names may vary, that variable should be defined using VARCHAR2.

Although the v_country_report could be defined as a LONG data type, the CLOB data type was introduced in the Oracle8i database and should now be used for documents that use the database character set exclusively.

Declaring Number Variables

- Number data types include NUMBER, INTEGER, PLS_INTEGER, BINARY_FLOAT and several others
- Adding the keyword CONSTANT constrains the variable so that its value cannot change
- Constants must be initialized

```
DECLARE
  v_employee_id      NUMBER(6,0);
  v_loop_count        INTEGER := 0;
  c_tax_rate          CONSTANT NUMBER(3,2) := 8.25;
  ...
```



PLSQL 2-4
Using Scalar Data Types

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

6

Data types such as BINARY_FLOAT, BINARY_DOUBLE, PLS_INTEGER, and BINARY_INTEGER are used primarily for high-speed scientific computation, as they are faster in calculations.

PLS_INTEGER and BINARY_INTEGER are PL/SQL-only data types that are more efficient than NUMBER or INTEGER for integer calculations.

Declaring Date Variables

- Date data types include DATE, TIMESTAMP, and TIMESTAMP WITH TIMEZONE

```
DECLARE
    v_date1      DATE := '05-Apr-2015';
    v_date2      DATE := v_date1 + 7;
    v_date3      TIMESTAMP := SYSDATE;
    v_date4      TIMESTAMP WITH TIME ZONE := SYSDATE;
BEGIN
    DBMS_OUTPUT.PUT_LINE(v_date1);
    DBMS_OUTPUT.PUT_LINE(v_date2);
    DBMS_OUTPUT.PUT_LINE(v_date3);
    DBMS_OUTPUT.PUT_LINE(v_date4);
END;
```

- Choosing between DATE, TIMESTAMP, etc., is determined by what data you need to know in the future



Academy

PLSQL 2-4
Using Scalar Data Types

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

7

Declaring BOOLEAN Variables

- BOOLEAN is a data type that stores one of the three possible values used for logical calculations: TRUE, FALSE, or NULL

```
DECLARE
    v_valid1      BOOLEAN := TRUE;
    v_valid2      BOOLEAN;
    v_valid3      BOOLEAN NOT NULL := FALSE;
BEGIN
    IF v_valid1 THEN
        DBMS_OUTPUT.PUT_LINE('Test is TRUE');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Test is FALSE');
    END IF;
END;
```



Academy

PLSQL 2-4
Using Scalar Data Types

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

8

A BOOLEAN variable is NULL by default and may remain NULL (unless the NOT NULL constraint is included, in which case the variable must be initialized to either TRUE or FALSE).

Notice that when a BOOLEAN variable is initialized, the value is NOT a string and is not enclosed in single quotes.

Using BOOLEAN Variables

- When using BOOLEAN variables:
 - Only the values TRUE, FALSE, and NULL can be assigned to a BOOLEAN variable
 - Conditional expressions use the logical operators AND and OR, and the operator NOT to check the variable values
 - The variables always yield TRUE, FALSE, or NULL
 - You can use arithmetic, character, and date expressions to return a BOOLEAN value



Guidelines for Declaring PL/SQL Variables

- Use meaningful and appropriate variable names
- Follow naming conventions
- Use v_name to represent a variable and c_name to represent a constant
- Declare one identifier per line for better readability, code maintenance, and easier commenting
- Use the NOT NULL constraint when the variable must hold a value
- Use the CONSTANT constraint when the variable value should not change within the block

NUMBER variables are typically initialized to "0" so as to avoid calculations using a NULL value.

BOOLEAN variables are typically initialized to FALSE and tested for TRUE.

Guidelines for Declaring PL/SQL Variables

- Set initial values for BOOLEANS and NUMBERS
- Avoid using column names as identifiers

```
DECLARE
    first_name VARCHAR2(20);
BEGIN
    SELECT first_name
        INTO first_name
        FROM employees
       WHERE last_name = 'Vargas';
    DBMS_OUTPUT.PUT_LINE(first_name);
END;
```



PLSQL 2-4
Using Scalar Data Types

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

11

In this example, the code works because the Oracle server knows the identifier following the SELECT keyword must be a column name and the identifier following the INTO keyword must be a variable name. Likewise, the argument for the PUT_LINE function also must be a variable name.

Although this code works, and is simple enough that we know what is happening, more complex code could be confusing to maintain or modify if variable names are identical to column names.

Defining Variables with the %TYPE Attribute

- Variables derived from database fields should be defined using the %TYPE attribute, which has several advantages
- For example, in the EMPLOYEES table, the column first_name is defined as VARCHAR2(20)
- In a PL/SQL block, you could define a matching variable with either:

```
v_first_name  VARCHAR2(20) ;
```

- Or

```
v_first_name  employees.last_name%TYPE ;
```



Academy

PLSQL 2-4
Using Scalar Data Types

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

12

The %TYPE reference is resolved at the time the code is compiled. It also establishes a dependency between the code and the table column referenced. This means that if the table column is changed, the code that references it is marked invalid. No changes are required to the invalid code, but the invalid code must be recompiled to update the reference.

Defining a data type using the %TYPE attribute is referred to as an "anchored data type."

Using the %TYPE Attribute

- Look at this partial table definition from the EMPLOYEES table
- Then look at the code in the next slide

Column Name	Data Type
EMPLOYEE_ID	NUMBER(6,0)
FIRST_NAME	VARCHAR2(20)
LAST_NAME	VARCHAR2(25)
EMAIL	VARCHAR2(25)

Using the %TYPE Attribute

- This PL/SQL block stores the correct first name in the v_first_name variable
- But what if the table column is later altered to be VARCHAR2(25) and a name longer than 20 characters is added?

```
DECLARE
  v_first_name    VARCHAR2(20);
BEGIN
  SELECT first_name
    INTO v_first_name
   FROM employees
 WHERE last_name = 'Vargas';
 DBMS_OUTPUT.PUT_LINE(v_first_name);
END;
```

Using the %TYPE attribute to define an "anchored data type" for v_first_name would solve the problem. Otherwise, a programmer would have to find and modify every place in every program with a variable defined to hold an employee's first name.

With the %TYPE attribute, the code would look like this:

```
DECLARE
  v_first_name employees.first_name%TYPE;
BEGIN
  SELECT first_name
    INTO v_first_name
   FROM employees
 WHERE last_name = 'Vargas';
 DBMS_OUTPUT.PUT_LINE(v_first_name);
```

END;

Using the %TYPE Attribute

- Using the %TYPE attribute to define an "anchored data type" for v_first_name would solve the problem
- Otherwise, a programmer would have to find and modify every place in every program with a variable defined to hold an employee's first name

```
DECLARE
    v_first_name          employees.first_name%TYPE;
BEGIN
    SELECT first_name
        INTO v_first_name
        FROM employees
       WHERE last_name = 'Vargas';
    DBMS_OUTPUT.PUT_LINE(v_first_name);
END;
```

Using the %TYPE Attribute

- The %TYPE attribute:

- Is used to automatically give a variable the same data type and size as:
 - A database column
 - Another declared variable
- Is prefixed with either of the following:
 - The database table name and column name
 - The name of the other declared variable



Using the %TYPE Attribute

- Syntax:

```
identifier      table_name.column_name%TYPE;  
identifier      identifier%TYPE;
```

- Examples:

```
DECLARE  
    v_first_name      employees.first_name%TYPE;  
    v_salary          employess.salary%TYPE;  
    v_old_salary      v_salary%TYPE;  
    v_new_salary      v_salary%TYPE;  
    v_balance         NUMBER(10,2);  
    v_min_balance    v_balance%TYPE := 1000;  
    ...
```



Academy

PLSQL 2-4
Using Scalar Data Types

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

17

In the examples,

v_first_name will have the same data type as the column first_name in the employees table.

v_salary will have the same data type as the column salary in the employees table.

v_old_salary will have the same data type as the variable v_salary.

v_new_salary will have the same data type as the variable v_salary.

v_min_balance will have the same data type as the variable v_balance.

A NOT NULL database column constraint does not apply to variables that are declared using %TYPE. Therefore, if you declare a variable using the %TYPE attribute that uses a database column defined as NOT NULL, that database constraint does NOT pass to the variable.

Advantages of the %TYPE Attribute

- Advantages of the %TYPE attribute are:
 - You can avoid errors caused by data type mismatch or wrong precision
 - You need not change the variable declaration if the table column definition changes
 - Otherwise, if you have already declared some variables for a particular table column without using the %TYPE attribute, then the PL/SQL block can return errors if the table column is altered

Advantages of the %TYPE Attribute

- Advantages of the %TYPE attribute are:
 - When you use the %TYPE attribute, PL/SQL determines the data type and size of the variable when the block is compiled
 - This ensures that such a variable is always compatible with the column that is used to populate it



Terminology

- Key terms used in this lesson included:
 - %TYPE
 - BOOLEAN

- %TYPE – Attribute used to declare a variable according to another previously declared variable or table column.
- BOOLEAN – A datatype that stores one of the three possible values used for logical calculations: TRUE, FALSE, or NULL.

Summary

- In this lesson, you should have learned how to:
 - Declare and use scalar data types in PL/SQL
 - Define guidelines for declaring and initializing PL/SQL variables
 - Identify the benefits of anchoring data types with the %TYPE attribute

ORACLE

Academy

ORACLE

Academy

Database Programming with PL/SQL

2-1

Using Variables in PL/SQL

ORACLE
Academy



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Objectives

- This lesson covers the following objectives:
 - List the uses of variables in PL/SQL
 - Identify the syntax for variables in PL/SQL
 - Declare and initialize variables in PL/SQL
 - Assign new values to variables in PL/SQL

Purpose

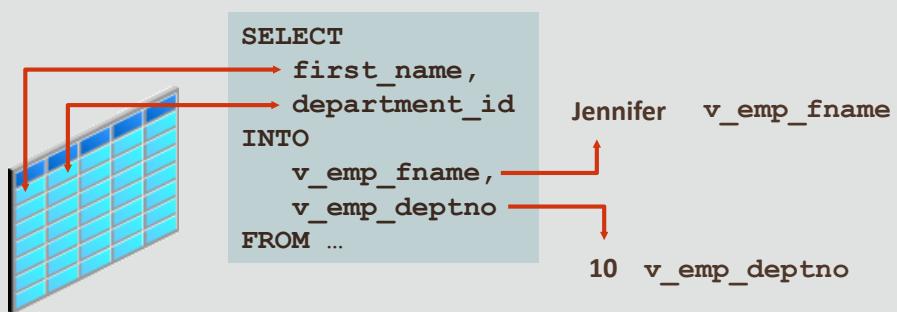
- You use variables to store and manipulate data
- In this lesson, you learn how to declare and initialize variables in the declarative section of a PL/SQL block
- With PL/SQL, you can declare variables and then use them in both SQL and procedural statements
- Variables can be thought of as storage containers that hold something until it is needed

Use of Variables

- Variables are expressions that stand for something of value (in the equation $x + y = 45$, x and y are variables that stand for two numbers that add up to 45)
- When defining a variable in a PL/SQL declaration section, you label a memory location, assign a datatype, and, if needed, assign a starting value for the variable
- A variable can represent a number, character string, boolean (true/false value), or other datatypes
- Throughout the PL/SQL code, variable values can be changed by the assignment operator (`:=`)

Use of Variables

- Use variables for:
 - Temporary storage of data
 - Manipulation of stored values
 - Reusability



ORACLE
Academy

PLSQL 2-1
Using Variables in PL/SQL

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

6

Variables are mainly used for the storage of data and manipulation of those stored values. Consider the SQL statement shown in the slide. The statement is retrieving the `first_name` and `department_id` from the table. If you have to manipulate the `first_name` or the `department_id`, then you have to store the retrieved value. Variables are used to temporarily store the values. You can use the value stored in these variables for processing or manipulating the data.

Handling Variables in PL/SQL

- Variables are:
 - Declared and initialized in the declarative section
 - Used and assigned new values in the executable section
- Variables can be:
 - Passed as parameters to PL/SQL subprograms
 - Assigned to hold the output of a PL/SQL subprogram



ORACLE
Academy

PLSQL 2-1
Using Variables in PL/SQL

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

7

Declaring Variables

- All PL/SQL variables must be declared in the declaration section before referencing them in the PL/SQL block
- The purpose of a declaration is to allocate storage space for a value, specify its data type, and name the storage location so that you can reference it
- You can declare variables in the declarative part of any PL/SQL block, subprogram, or package



A variable is a name or label that points to a value stored at a particular location in the computer's memory.

Declaring Variables: Syntax

- The identifier is the name of the variable
- It and the datatype are the minimum elements required

```
identifier [CONSTANT] datatype [NOT NULL]  
[:= expr | DEFAULT expr];
```



In addition to variables, you also can declare cursors and exceptions in the declarative section. You will learn how to declare cursors and exceptions later in the course.

Initializing Variables

- Variables are assigned a memory location inside the DECLARE section
- Variables can be assigned a value at the same time
- This process is called initialization
- The value in a variable also can be modified by reinitializing the variable in the executable section

```
DECLARE
    v_counter INTEGER := 0;
BEGIN
    v_counter := v_counter + 1;
    DBMS_OUTPUT.PUT_LINE(v_counter);
END;
```

Declaring and Initializing Variables Example 1

- This example shows the declaration of several variables of various datatypes using syntax that sets constraints, defaults, and initial values
- You will learn more about the different syntax as the course progresses

```
DECLARE
    fam_birthdate DATE;
    fam_size      NUMBER(2) NOT NULL := 10;
    fam_location  VARCHAR2(13) := 'Florida';
    fam_bank      CONSTANT NUMBER := 50000;
    fam_population INTEGER;
    fam_name      VARCHAR2(20) DEFAULT 'Roberts';
    fam_party_size CONSTANT PLS_INTEGER := 20;
```

Declaring and Initializing Variables Example 2

- This example shows the convention of beginning variables with v_ and variables that are configured as constants with c_

```
DECLARE
    v_emp_hiredate      DATE;
    v_emp_deptno        NUMBER(2) NOT NULL := 10;
    v_location          VARCHAR2(13) := 'Atlanta';
    c_comm              CONSTANT NUMBER := 1400;
    v_population        INTEGER;
    v_book_type         VARCHAR2(20) DEFAULT 'fiction';
    v_artist_name       VARCHAR2(50);
    v_firstname         VARCHAR2(20) := 'Rajiv';
    v_lastname          VARCHAR2(20) DEFAULT 'Kumar';
    c_display_no        CONSTANT PLS_INTEGER := 20;
```

- The defining of data types and data structures using a standard in a programming language is a significant aid to readability



Academy

PLSQL 2-1
Using Variables in PL/SQL

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

12

Assigning Values in the Executable Section Example 1

- After a variable is declared, you can use it in the executable section of a PL/SQL block
- For example, in the following block, the variable `v_myname` is declared in the declarative section of the block

```
DECLARE
    v_myname  VARCHAR2(20);
BEGIN
    DBMS_OUTPUT.PUT_LINE('My name is: '||v_myname);
    v_myname := 'John';
    DBMS_OUTPUT.PUT_LINE('My name is: '||v_myname);
END;
```



PLSQL 2-1
Using Variables in PL/SQL

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

13

Assigning Values in the Executable Section Example 1

- You can access this variable in the executable section of the same block
- What do you think the block will print?

```
DECLARE
    v_myname  VARCHAR2(20);
BEGIN
    DBMS_OUTPUT.PUT_LINE('My name is: '||v_myname);
    v_myname := 'John';
    DBMS_OUTPUT.PUT_LINE('My name is: '||v_myname);
END;
```



PLSQL 2-1
Using Variables in PL/SQL

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

14

The code results in the following output:

My name is:

My name is: John

Statement processed.

Assigning Values in the Executable Section Example 1

- In this example, the variable has no value when the first PUT_LINE is executed, but then the value John is assigned to the variable before the second PUT_LINE
- The value of the variable is then concatenated with the string My name is:
- The output is:

```
My name is:  
My name is: John  
  
Statement process.
```

- A non-initialized variable contains a NULL value until a non-null value is explicitly assigned to it

Assigning Values in the Executable Section Example 2

- In this block, the variable `v_myname` is declared and initialized
- It begins with the value John, but the value is then manipulated in the executable section of the block

```
DECLARE
    v_myname  VARCHAR2(20) := 'John';
BEGIN
    v_myname := 'Steven';
    DBMS_OUTPUT.PUT_LINE('My name is: '|| v_myname);
END;
```

- The output is:

```
My name is: Steven
Statement processed.
```

Passing Variables as Parameters to PL/SQL Subprograms

- Parameters are values passed to a subprogram by the user or by another program
- The subprogram uses the value in the parameter when it runs
- The subprogram may also return a parameter to the calling environment. In PL/SQL, subprograms are generally known as procedures or functions
- You will learn more about procedures and functions as the course progresses

Passing Variables as Parameters to PL/SQL Subprograms

- In the following example, the parameter `v_date` is being passed to the procedure `PUT_LINE`, which is part of the package `DBMS_OUTPUT`

```
DECLARE
    v_date      VARCHAR2(30);
BEGIN
    SELECT TO_CHAR(SYSDATE) INTO v_date FROM DUAL;
    DBMS_OUTPUT.PUT_LINE(v_date);
END;
```



The syntax in this slide will be explained in more detail in later lessons. For now, you simply need to understand that in PL/SQL, a variable can be passed from one program to another program, and we call these variables parameters.

Assigning Variables to PL/SQL Subprogram Output

- You can use variables to hold values that are returned by a function (see function definition below and a call to this function on the following slide)

```
CREATE FUNCTION num_characters (p_string IN VARCHAR2)
RETURN INTEGER IS
    v_num_characters INTEGER;
BEGIN
    SELECT LENGTH(p_string) INTO v_num_characters
        FROM DUAL;
    RETURN v_num_characters;
END ;
```

- The concept, creation, and use of functions will be covered later in this course

Assigning Variables to PL/SQL Subprogram Output

- In the call to the function num_characters, the value returned by the function will be stored in the variable v_length_of_string

```
DECLARE
    v_length_of_string  INTEGER;
BEGIN
    v_length_of_string := num_characters('Oracle
Corporation');
    DBMS_OUTPUT.PUT_LINE(v_length_of_string);
END;
```

Terminology

- Key terms used in this lesson included:
 - Parameters
 - Variables

- Parameters – values passed to a program by a user or by another program
- Variables – used for temporary storage and manipulation of data

Summary

- In this lesson, you should have learned how to:
 - List the uses of variables in PL/SQL
 - Identify the syntax for variables in PL/SQL
 - Declare and initialize variables in PL/SQL
 - Assign new values to variables in PL/SQL

ORACLE

Academy

ORACLE

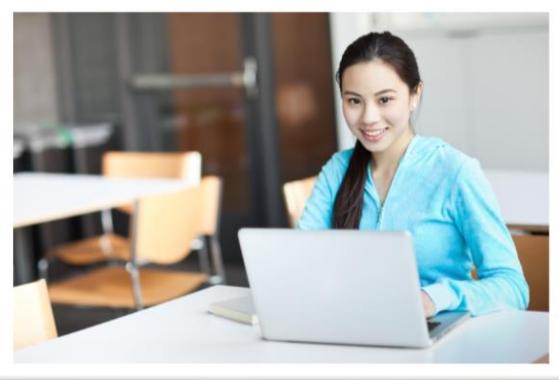
Academy

Database Programming with PL/SQL

2-5

Writing PL/SQL Executable Statements

ORACLE
Academy



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Objectives

- This lesson covers the following objectives:
 - Construct accurate variable assignment statements in PL/SQL
 - Construct accurate statements using built-in SQL functions in PL/SQL
 - Differentiate between implicit and explicit conversions of data types
 - Describe when implicit conversions of data types take place
 - List the drawbacks of implicit data type conversions
 - Construct accurate statements using functions to explicitly convert data types
 - Construct statements using operators in PL/SQL

Purpose

- We've introduced variables and identifiers
- In this lesson, you build your knowledge of the PL/SQL programming language by writing code to assign variable values
- These values can be literals or values returned by a function
- SQL provides a number of predefined functions that you can use in SQL statements
- Most of these functions are also valid in PL/SQL expressions

Assigning New Values to Variables

- Character and date literals must be enclosed in single quotation marks

```
v_name := 'Henderson';
v_start_date := '12-Dec-2005';
```

- Statements can continue over several lines

```
v_quote := 'The only thing that we can know is that we know
nothing and that is the highest flight of human reason.';
```

- Numbers can be simple values or scientific notation
(2E5 meaning 2×10^5 = 200,000)

```
v_my_integer := 100;
v_my_sci_not := 2E5;
```



Academy

PLSQL 2-S
Writing PL/SQL Executable Statements

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

5

Assigning a value to a variable: the variable must always be on the left side of the assignment symbol (:=); the value will always be on the right side of the assignment symbol.

```
v_num := v_count + c_people
```

v_num is the variable memory location that will be assigned the value in the variable memory location v_count plus the value of the memory location c_people.

SQL Functions in PL/SQL

- You are already familiar with functions in SQL statements, for example:

```
SELECT LAST_DAY(SYSDATE)
  FROM DUAL;
```

- You can also use these functions in PL/SQL procedural statements, for example:

```
DECLARE
  v_last_day DATE;
BEGIN
  v_last_day := LAST_DAY(SYSDATE);
  DBMS_OUTPUT.PUT_LINE(v_last_day);
END;
```



Functions are used as short cuts. Someone already programmed a block of code to accomplish a specific process. These blocks of code are called procedures and functions. Use them to make writing your program easier. Later in this course, you will learn how to write your own procedures and functions.

SQL Functions in PL/SQL

- Functions available in procedural statements:
 - Single-row character
 - Single-row number
 - Date
 - Data-type conversion
 - Miscellaneous functions
- Not available in procedural statements:
 - DECODE (CASE is used instead)
 - Group functions (AVG, MIN, MAX etc. may be used ONLY within a SQL statement)



SQL Functions in PL/SQL

- SQL functions help you to manipulate data; they fall into the following categories:
 - Character
 - Number
 - Date
 - Conversion
 - Miscellaneous

Character Functions

- Valid character functions in PL/SQL include:

ASCII	LENGTH	RPAD
CHR	LOWER	RTRIM
CONCAT	LPAD	SUBSTR
INITCAP	LTRIM	TRIM
INSTR	REPLACE	UPPER

- This is not an exhaustive list
- Refer to the Oracle documentation for the complete list

Examples of Character Functions

- Get the length of a string:

```
v_desc_size  INTEGER(5);
v_prod_description VARCHAR2(70) :='You can use this product
with your radios for higher frequency';

-- get the length of the string in prod_description
v_desc_size:= LENGTH(v_prod_description);
```

- Convert the name of the country capitol to upper case:

```
v_capitol_name:= UPPER(v_capitol_name);
```

- Concatenate the first and last names:

```
v_emp_name:= v_first_name||' '||v_last_name;
```



Academy

PLSQL 2-S
Writing PL/SQL Executable Statements

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

10

Number Functions

- Valid number functions in PL/SQL include:

ABS	EXP	ROUND
ACOS	LN	SIGN
ASIN	LOG	SIN
ATAN	MOD	TAN
COS	POWER	TRUNC

- This is not an exhaustive list
- Refer to the Oracle documentation for the complete list

Examples of Number Functions

- Get the sign of a number:

```
DECLARE
  v_my_num BINARY_INTEGER := -56664;
BEGIN
  DBMS_OUTPUT.PUT_LINE(SIGN(v_my_num));
END;
```

- Round a number to 0 decimal places:

```
DECLARE
  v_median_age NUMBER(6,2);
BEGIN
  SELECT meian_age INTO v_median_age
  FROM countries
  WHERE country_id = 27;
  DBMS_OUTPUT.PUT_LINE(ROUND(v_median_age,0));
END;
```



Academy

PLSQL 2-S
Writing PL/SQL Executable Statements

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

12

Date Functions

- Valid date functions in PL/SQL include:

ADD_MONTHS	MONTHS_BETWEEN
CURRENT_DATE	ROUND
CURRENT_TIMESTAMP	SYSDATE
LAST_DAY	TRUNC

- This is not an exhaustive list
- Refer to the Oracle documentation for the complete list



Examples of Date Functions

- Add months to a date:

```
DECLARE
  v_new_date DATE;
  v_num_months NUMBER := 6;
BEGIN
  v_new_date := ADD_MONTHS(SYSDATE,v_num_months);
  DBMS_OUTPUT.PUT_LINE(v_new_date);
END;
```

- Calculate the number of months between two dates:

```
DECLARE
  v_no_months PLS_INTEGER :=0;
BEGIN
  v_no_months := MONTHS_BETWEEN('31-Jan-2006','31-May-2005');
  DBMS_OUTPUT.PUT_LINE(v_no_months);
END;
```

Data-Type Conversion

- In any programming language, converting one data type to another is a common requirement
- PL/SQL can handle such conversions with scalar data types
- Data-type conversions can be of two types:
 - Implicit conversions
 - Explicit conversions



Implicit Conversions

- In implicit conversions, PL/SQL attempts to convert data types dynamically if they are mixed in a statement
- Implicit conversions can happen between many types in PL/SQL, as illustrated by the following chart

	DATE	LONG	NUMBER	PLS_INTEGER	VARCHAR2
DATE	N/A	X			X
LONG		N/A			X
NUMBER		X	N/A	X	X
PLS_INTEGER		X	X	N/A	X
VARCHAR2	X	X	X	X	N/A

Whenever PL/SQL detects that a conversion is necessary, it attempts to change the values as required to perform the operation.

In the chart, the cells marked 'X' show which implicit conversions can be done.

For this course, we will focus on implicit conversions between:

Characters and numbers

Characters and dates

For more information about the above chart, refer to “Converting PL/SQL Data Types” in the PL/SQL User’s Guide and Reference.

Example of Implicit Conversion

- In this example, the variable v_sal_increase is of type VARCHAR2
- While calculating the total salary, PL/SQL first converts v_sal_increase to NUMBER and then performs the operation
- The result of the operation is the NUMBER type

```
DECLARE
    v_salary    NUMBER(6) := 6000;
    v_sal_increase VARCHAR2(5) := '1000';
    v_total_salary v_salary%TYPE;
BEGIN
    v_total_salary := v_salary + v_sal_increase;
    DBMS_OUTPUT.PUT_LINE(v_total_salary);
END;
```



Academy

PLSQL 2-S
Writing PL/SQL Executable Statements

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

17

Drawbacks of Implicit Conversions

- At first glance, implicit conversions might seem useful; however, there are several drawbacks:
 - Implicit conversions can be slower
 - When you use implicit conversions, you lose control over your program because you are making an assumption about how Oracle handles the data
 - If Oracle changes the conversion rules, then your code can be affected
 - Code that uses implicit conversion is harder to read and understand

Drawbacks of Implicit Conversions

- Additional drawbacks:
 - Implicit conversion rules depend upon the environment in which you are running
 - For example, the date format varies depending on the language setting and installation type
 - Code that uses implicit conversion might not run on a different server or in a different language
 - It is strongly recommended that you AVOID allowing SQL or PL/SQL to perform implicit conversions on your behalf
 - You should use conversion functions to guarantee that the right kinds of conversions take place

Drawbacks of Implicit Conversions

- It is the programmer's responsibility to ensure that values can be converted
- For instance, PL/SQL can convert the CHAR value '02-Jun-1992' to a DATE value, but cannot convert the CHAR value 'Yesterday' to a DATE value
- Similarly, PL/SQL cannot convert a VARCHAR2 value containing alphabetic characters to a NUMBER value

Valid?	Statement
Yes	v_new_date DATE := '02-Jun-1992';
No	v_new_date DATE := 'Yesterday';
Yes	v_my_number NUMBER := '123';
No	v_my_number NUMBER := 'abc';

Explicit Conversions

- Explicit conversions convert values from one data type to another by using built-in functions
- Examples of conversion functions include:

TO_NUMBER()	ROWIDTONCHAR()
TO_CHAR()	HEXTORAW()
TO_CLOB()	RAWTOHEX()
CHARTOROWID()	RAWTONHEX()
ROWIDTOCHAR()	TO_DATE()

Examples of Explicit Conversions

- **TO_CHAR**

```
BEGIN  
  DBMS_OUTPUT.PUT_LINE(TO_CHAR(SYSDATE, 'Month YYYY'));  
END;
```

- **TO_DATE**

```
BEGIN  
  DBMS_OUTPUT.PUT_LINE(TO_DATE('April-1999', 'Month-YYYY'));  
END;
```

Examples of Explicit Conversions

- TO_NUMBER

```
DECLARE
  v_a VARCHAR2(10) := '-123456';
  v_b VARCHAR2(10) := '+987654';
  v_c PLS_INTEGER;
BEGIN
  v_c := TO_NUMBER(v_a) + TO_NUMBER(v_b);
  DBMS_OUTPUT.PUT_LINE(v_c);
END;
```

- Note that the DBMS_OUTPUT.PUT_LINE procedure expects an argument with a character type such as VARCHAR2
- Variable v_c is a number, therefore we should explicitly code: DBMS_OUTPUT.PUT_LINE(TO_CHAR(v_c));



Academy

PLSQL 2-S
Writing PL/SQL Executable Statements

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

23

Data Type Conversion Examples

- Example #1

```
v_date_of_joining DATE := '02-Feb-2014';
```

- Example #2

```
v_date_of_joining DATE := 'February 02, 2014';
```

- Example #3

```
v_date_of_joining DATE := TO_DATE('February 02, 2014',  
'Month DD, YYYY');
```

The examples in the slide show implicit and explicit conversions of the DATE data type.

Example #1 - Implicit conversion happens in this case and the date is assigned to v_date_of_joining.

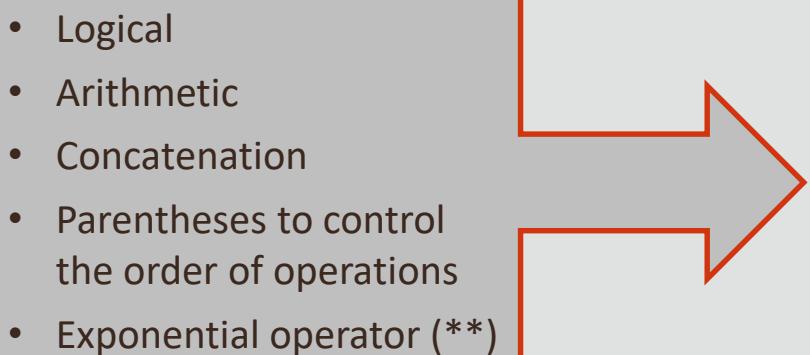
Example #2 - PL/SQL gives you an error because the date that is being assigned is not in the default format.

Example #3 - This is how it should be done. Use the TO_DATE function to explicitly convert the given date in a particular format and assign it to the DATE data type variable date_of_joining.

Operators in PL/SQL

- The operations within an expression are performed in a particular order depending on their precedence (priority)

- Logical
- Arithmetic
- Concatenation
- Parentheses to control the order of operations
- Exponential operator (**)



Same as
in SQL

Operators in PL/SQL

- The following table shows the default order of operations from high priority to low priority:

Operator	Operation
**	Exponentiation
+, -	Identity, negation
*, /	Multiplication, division
+, -,	Addition, subtraction, concatenation
=, <, >, <=, >=, <>, !=, ~~, ^~, IS NULL, LIKE, BETWEEN, IN	Comparison
NOT	Logical negation
AND	Conjunction
OR	Inclusion

Operators in PL/SQL Examples

- Increment the counter for a loop

```
v_loop_count := v_loop_count + 1;
```

- Set the value of a Boolean flag

```
v_good_sal := v_sal BETWEEN 50000 AND 150000;
```

- Validate whether an employee number contains a value

```
v_valid      := (v_empno IS NOT NULL);
```



ORACLE
Academy

PLSQL 2-S
Writing PL/SQL Executable Statements

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

27

The second example is particularly elegant when you consider the same result could be obtained by the following code:

```
IF v_sal BETWEEN 50000 AND 150000 THEN
  v_good_sal := TRUE;
ELSE
  v_good_sal := FALSE;
END IF;
```

Terminology

- Key terms used in this lesson included:
 - Explicit conversion
 - Implicit conversion

- Explicit conversion – Converts values from one data type to another by using built-in functions.
- Implicit conversion – Converts data types dynamically if they are mixed in a statement.

Summary

- In this lesson, you should have learned how to:
 - Construct accurate variable assignment statements in PL/SQL
 - Construct accurate statements using built-in SQL functions in PL/SQL
 - Differentiate between implicit and explicit conversions of data types
 - Describe when implicit conversions of data types take placeList the drawbacks of implicit data type conversions
 - Construct accurate statements using functions to explicitly convert data types
 - Construct statements using operators in PL/SQL

ORACLE

Academy

ORACLE

Academy

Database Programming with PL/SQL

2-5

Writing PL/SQL Executable Statements

ORACLE
Academy



Objectives

- This lesson covers the following objectives:
 - Construct accurate variable assignment statements in PL/SQL
 - Construct accurate statements using built-in SQL functions in PL/SQL
 - Differentiate between implicit and explicit conversions of data types
 - Describe when implicit conversions of data types take place
 - List the drawbacks of implicit data type conversions
 - Construct accurate statements using functions to explicitly convert data types
 - Construct statements using operators in PL/SQL

Purpose

- We've introduced variables and identifiers
- In this lesson, you build your knowledge of the PL/SQL programming language by writing code to assign variable values
- These values can be literals or values returned by a function
- SQL provides a number of predefined functions that you can use in SQL statements
- Most of these functions are also valid in PL/SQL expressions

Assigning New Values to Variables

- Character and date literals must be enclosed in single quotation marks

```
v_name := 'Henderson';  
v_start_date := '12-Dec-2005';
```

- Statements can continue over several lines

```
v_quote := 'The only thing that we can know is that we know  
nothing and that is the highest flight of human reason.';
```

- Numbers can be simple values or scientific notation
(2E5 meaning 2×10^5 = 200,000)

```
v_my_integer := 100;  
v_my_sci_not := 2E5;
```

SQL Functions in PL/SQL

- You are already familiar with functions in SQL statements, for example:

```
SELECT LAST_DAY(SYSDATE)
  FROM DUAL;
```

- You can also use these functions in PL/SQL procedural statements, for example:

```
DECLARE
  v_last_day DATE;
BEGIN
  v_last_day := LAST_DAY(SYSDATE);
  DBMS_OUTPUT.PUT_LINE(v_last_day);
END;
```

SQL Functions in PL/SQL

- Functions available in procedural statements:
 - Single-row character
 - Single-row number
 - Date
 - Data-type conversion
 - Miscellaneous functions
- Not available in procedural statements:
 - DECODE (CASE is used instead)
 - Group functions (AVG, MIN, MAX etc. may be used ONLY within a SQL statement)



SQL Functions in PL/SQL

- SQL functions help you to manipulate data; they fall into the following categories:
 - Character
 - Number
 - Date
 - Conversion
 - Miscellaneous

Character Functions

- Valid character functions in PL/SQL include:

ASCII	LENGTH	RPAD
CHR	LOWER	RTRIM
CONCAT	LPAD	SUBSTR
INITCAP	LTRIM	TRIM
INSTR	REPLACE	UPPER

- This is not an exhaustive list
- Refer to the Oracle documentation for the complete list

Examples of Character Functions

- Get the length of a string:

```
v_desc_size  INTEGER(5);
v_prod_description VARCHAR2(70):='You can use this product
with your radios for higher frequency';

-- get the length of the string in prod_description
v_desc_size:= LENGTH(v_prod_description);
```

- Convert the name of the country capitol to upper case:

```
v_capitol_name:= UPPER(v_capitol_name);
```

- Concatenate the first and last names:

```
v_emp_name:= v_first_name||' '||v_last_name;
```

Number Functions

- Valid number functions in PL/SQL include:

ABS	EXP	ROUND
ACOS	LN	SIGN
ASIN	LOG	SIN
ATAN	MOD	TAN
COS	POWER	TRUNC

- This is not an exhaustive list
- Refer to the Oracle documentation for the complete list

Examples of Number Functions

- Get the sign of a number:

```
DECLARE
  v_my_num BINARY_INTEGER := -56664;
BEGIN
  DBMS_OUTPUT.PUT_LINE(SIGN(v_my_num));
END;
```

- Round a number to 0 decimal places:

```
DECLARE
  v_median_age NUMBER(6,2);
BEGIN
  SELECT meian_age INTO v_median_age
  FROM countries
  WHERE country_id = 27;
  DBMS_OUTPUT.PUT_LINE(ROUND(v_median_age,0));
END;
```

Date Functions

- Valid date functions in PL/SQL include:

ADD_MONTHS	MONTHS_BETWEEN
CURRENT_DATE	ROUND
CURRENT_TIMESTAMP P	SYSDATE
LAST_DAY	TRUNC

- This is not an exhaustive list
- Refer to the Oracle documentation for the complete list



Examples of Date Functions

- Add months to a date:

```
DECLARE
  v_new_date DATE;
  v_num_months NUMBER := 6;
BEGIN
  v_new_date := ADD_MONTHS(SYSDATE,v_num_months);
  DBMS_OUTPUT.PUT_LINE(v_new_date);
END;
```

- Calculate the number of months between two dates:

```
DECLARE
  v_no_months PLS_INTEGER :=0;
BEGIN
  v_no_months := MONTHS_BETWEEN('31-Jan-2006','31-May-2005');
  DBMS_OUTPUT.PUT_LINE(v_no_months);
END;
```

Data-Type Conversion

- In any programming language, converting one data type to another is a common requirement
- PL/SQL can handle such conversions with scalar data types
- Data-type conversions can be of two types:
 - Implicit conversions
 - Explicit conversions



Implicit Conversions

- In implicit conversions, PL/SQL attempts to convert data types dynamically if they are mixed in a statement
- Implicit conversions can happen between many types in PL/SQL, as illustrated by the following chart

	DATE	LONG	NUMBER	PLS_INTEGER	VARCHAR2
DATE	N/A	X			X
LONG		N/A			X
NUMBER		X	N/A	X	X
PLS_INTEGER		X	X	N/A	X
VARCHAR2	X	X	X	X	N/A

Example of Implicit Conversion

- In this example, the variable `v_sal_increase` is of type `VARCHAR2`
- While calculating the total salary, PL/SQL first converts `v_sal_increase` to `NUMBER` and then performs the operation
- The result of the operation is the `NUMBER` type

```
DECLARE
    v_salary NUMBER(6) := 6000;
    v_sal_increase VARCHAR2(5) := '1000';
    v_total_salary v_salary%TYPE;
BEGIN
    v_total_salary := v_salary + v_sal_increase;
    DBMS_OUTPUT.PUT_LINE(v_total_salary);
END;
```

Drawbacks of Implicit Conversions

- At first glance, implicit conversions might seem useful; however, there are several drawbacks:
 - Implicit conversions can be slower
 - When you use implicit conversions, you lose control over your program because you are making an assumption about how Oracle handles the data
 - If Oracle changes the conversion rules, then your code can be affected
 - Code that uses implicit conversion is harder to read and understand

Drawbacks of Implicit Conversions

- Additional drawbacks:
 - Implicit conversion rules depend upon the environment in which you are running
 - For example, the date format varies depending on the language setting and installation type
 - Code that uses implicit conversion might not run on a different server or in a different language
 - It is strongly recommended that you AVOID allowing SQL or PL/SQL to perform implicit conversions on your behalf
 - You should use conversion functions to guarantee that the right kinds of conversions take place

Drawbacks of Implicit Conversions

- It is the programmer's responsibility to ensure that values can be converted
- For instance, PL/SQL can convert the CHAR value '02-Jun-1992' to a DATE value, but cannot convert the CHAR value 'Yesterday' to a DATE value
- Similarly, PL/SQL cannot convert a VARCHAR2 value containing alphabetic characters to a NUMBER value

Valid?	Statement
Yes	v_new_date DATE := '02-Jun-1992';
No	v_new_date DATE := 'Yesterday';
Yes	v_my_number NUMBER := '123';
No	v_my_number NUMBER := 'abc';

Explicit Conversions

- Explicit conversions convert values from one data type to another by using built-in functions
- Examples of conversion functions include:

TO_NUMBER()	ROWIDTONCHAR()
TO_CHAR()	HEXTORAW()
TO_CLOB()	RAWTOHEX()
CHARTOROWID()	RAWTONHEX()
ROWIDTOCHAR()	TO_DATE()

Examples of Explicit Conversions

- TO_CHAR

```
BEGIN  
  DBMS_OUTPUT.PUT_LINE(TO_CHAR(SYSDATE, 'Month YYYY'));  
END;
```

- TO_DATE

```
BEGIN  
  DBMS_OUTPUT.PUT_LINE(TO_DATE('April-1999', 'Month-YYYY'));  
END;
```

Examples of Explicit Conversions

- TO_NUMBER

```
DECLARE
  v_a VARCHAR2(10) := '-123456';
  v_b VARCHAR2(10) := '+987654';
  v_c PLS_INTEGER;
BEGIN
  v_c := TO_NUMBER(v_a) + TO_NUMBER(v_b);
  DBMS_OUTPUT.PUT_LINE(v_c);
END;
```

- Note that the DBMS_OUTPUT.PUT_LINE procedure expects an argument with a character type such as VARCHAR2
- Variable v_c is a number, therefore we should explicitly code: DBMS_OUTPUT.PUT_LINE(TO_CHAR(v_c));

Data Type Conversion Examples

- Example #1

```
v_date_of_joining DATE := '02-Feb-2014';
```

- Example #2

```
v_date_of_joining DATE := 'February 02, 2014';
```

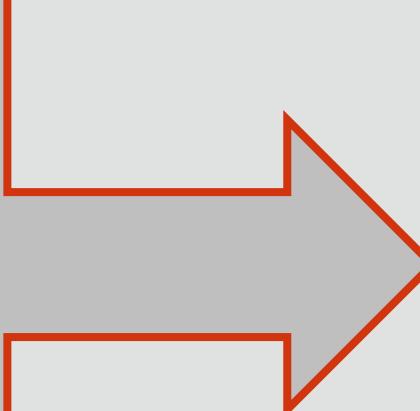
- Example #3

```
v_date_of_joining DATE := TO_DATE('February 02, 2014',  
'Month DD, YYYY');
```

Operators in PL/SQL

- The operations within an expression are performed in a particular order depending on their precedence (priority)

- Logical
- Arithmetic
- Concatenation
- Parentheses to control the order of operations
- Exponential operator (**)



Same as
in SQL

Operators in PL/SQL

- The following table shows the default order of operations from high priority to low priority:

Operator	Operation
**	Exponentiation
+, -	Identity, negation
*, /	Multiplication, division
+, -,	Addition, subtraction, concatenation
=, <, >, <=, >=, <>, !=, ~=, ^=, IS NULL, LIKE, BETWEEN, IN	Comparison
NOT	Logical negation
AND	Conjunction
OR	Inclusion

Operators in PL/SQL Examples

- Increment the counter for a loop

```
v_loop_count := v_loop_count + 1;
```

- Set the value of a Boolean flag

```
v_good_sal := v_sal BETWEEN 50000 AND 150000;
```

- Validate whether an employee number contains a value

```
v_valid := (v_empno IS NOT NULL);
```



EXAMPLE

Terminology

- Key terms used in this lesson included:
 - Explicit conversion
 - Implicit conversion

Summary

- In this lesson, you should have learned how to:
 - Construct accurate variable assignment statements in PL/SQL
 - Construct accurate statements using built-in SQL functions in PL/SQL
 - Differentiate between implicit and explicit conversions of data types
 - Describe when implicit conversions of data types take placeList the drawbacks of implicit data type conversions
 - Construct accurate statements using functions to explicitly convert data types
 - Construct statements using operators in PL/SQL

ORACLE

Academy

ORACLE

Academy

ORACLE

Academy

Database Programming with PL/SQL

3-3

Manipulating Data in PL/SQL

ORACLE
Academy



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Objectives

- This lesson covers the following objectives:
 - 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

Purpose

- You have 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
- DML statements will help you perform a task on more than a single row

Create Copy of Original Table

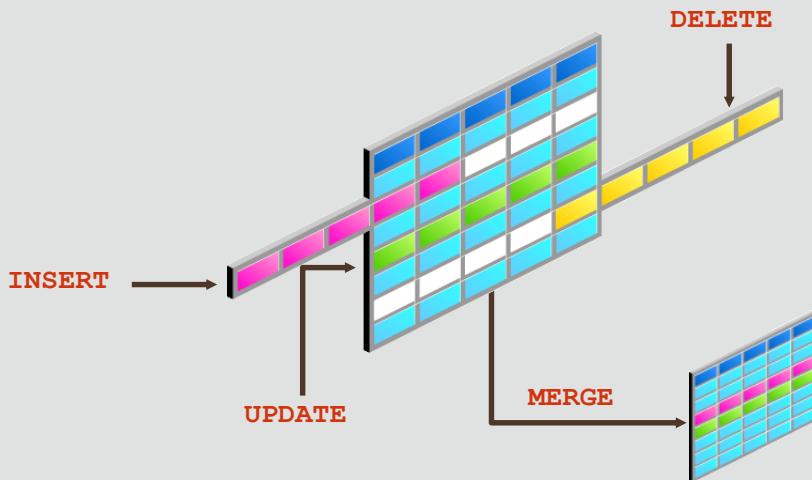
- It is very important that you do NOT modify the existing tables (such as EMPLOYEES and DEPARTMENTS), because they will be needed later in the course
- The examples in this lesson use the COPY_EMP table
- If you haven't already created the COPY_EMP table, do so now by executing this SQL statement:

```
CREATE TABLE copy_emp
  AS SELECT *
    FROM employees;
```

Manipulating Data Using PL/SQL

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

- INSERT
- UPDATE
- DELETE
- MERGE



A DML statement within a PL/SQL block can modify many rows (unlike a SELECT statement, which must read exactly one row).

Manipulating Data Using PL/SQL

- 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
 - 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

Manipulating Data Using PL/SQL

- 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

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

Important Note: The data in the EMPLOYEES table needs to remain unchanged for later in the course. Therefore we use the COPY_EMP table in all these DML examples.

If you haven't already created the COPY_EMP table, do so now by executing this SQL statement:

```
CREATE TABLE copy_emp
AS SELECT *
FROM employees;
```

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 :=
BEGIN
    UPDATE copy_emp ←
        SET      salary = salary + v_sal_increas
        WHERE job_id = 'ST_CLERK';
END;
```

It is recommended that the UPDATE statement be on a line of its own.



There may be ambiguity in the SET clause of the UPDATE statement because although the identifier on the left of the assignment operator is always a database column, the identifier on the right can be either a database column or a PL/SQL variable. Recall that if column names and identifier names are identical in the WHERE clause, then the Oracle server looks to the database first for the name.

PL/SQL variable assignments always use :=, and SQL column assignments always use =.

Although we are using the copy_emp table, it is ok to use the original table for the definition of the variable datatype employees.salary%TYPE.

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;
```



ORACLE
Academy

PLSQL 3-3
Manipulating Data in PL/SQL

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

11

The DELETE statement removes unwanted rows from a table. If the WHERE clause is not used, then all the rows in a table will be removed, provided that no integrity constraints are violated.

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;
```



The example shown matches the employee_id in the COPY_EMP table to the employee_id in the EMPLOYEES table. If a match is found, then the row is updated to match the row in the EMPLOYEES table. If the row is not found, then it is inserted into the copy_emp table.

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

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

The word “cursor” has several meanings in Oracle. It is sometimes used to mean a pointer to the private memory area, rather than the memory area itself. It is also used to refer to an area of shared memory. In this course, we focus only on its meaning in the PL/SQL environment.

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



Implicit cursors: Implicit cursors are created and managed by the Oracle server. Oracle uses an implicit cursor for each SELECT, UPDATE, DELETE, or INSERT statement you execute in a program. The Oracle server creates such a cursor when it has to execute a SQL statement. The remaining pages in this lesson discuss implicit cursors.

Explicit cursors: Explicit cursors are declared by the programmer. Explicit cursors are discussed in a later lesson.

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



Cursor Attributes for Implicit Cursors

Attribute	Description
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.

You can test the attributes — SQL%ROWCOUNT, SQL%FOUND, and SQL%NOTFOUND — in the executable section of a block to gather information after the appropriate command.

The SQL%NOTFOUND attribute is opposite to SQL%FOUND. This attribute may be used as the exit condition in a loop. It is useful in UPDATE or DELETE statements when no rows are changed because exceptions are not returned in these cases.

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;
```

The example in the slide deletes all rows with department_id 50 from the copy_emp table. Using the SQL%ROWCOUNT attribute, you can display the number of rows deleted.

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;
```

Using Implicit Cursor Attributes: Good Practice Guideline

- Look at this code which creates a table and then executes a PL/SQL block
- Determine 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;
```



PLSQL 3-3
Manipulating Data in PL/SQL

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

20

No value is inserted into RESULTS table.

The goal of the code in the slide is to store the number of rows updated by the UPDATE statement in the RESULTS table. What actually happens, though, is an error. Upon starting the INSERT statement, the previous cursor attributes are "erased." Until the INSERT statement completes, there are no new cursor attributes, so there is no value to be inserted into the RESULTS table.

Using Implicit Cursor Attributes: Good Practice Guideline

- To INSERT the value in the RESULTS table - we must save the SQL%ROWCOUNT value in a declared variable
- The value is displayed using PUT_LINE

```
DECLARE
  v_rowcount      INTEGER;
BEGIN
UPDATE copy_emp
  SET salary = salary + 100
 WHERE job_id = 'ST_CLERK';
DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT || ' rows in COPY_EMPupdated.');
v_rowcount := SQL%ROWCOUNT;
INSERT INTO results (num_rows)
VALUES (v_rowcount);
DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT || ' rows in RESULTS updated.');
END;
```

Terminology

- Key terms used in this lesson included:
 - INSERT
 - UPDATE
 - DELETE
 - MERGE
 - Explicit cursors
 - Implicit cursors

- INSERT - Statement adds new rows to the table.
- UPDATE - Statement modifies existing rows in the table.
- DELETE - Statement removes rows from the table.
- 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.
- Explicit cursors: Defined by the programmer for queries that return more than one row.
- Implicit cursors: Defined automatically by Oracle for all SQL data manipulation statements, and for queries that return only one row.

Summary

- In this lesson, you should have learned how 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

ORACLE

Academy

Database Programming with PL/SQL

3-3: Manipulating Data in PL/SQL

Practice Activities

Vocabulary

Identify the vocabulary word for each definition below:

	Defined automatically by Oracle for all SQL data manipulation statements, and for queries that return only one row.
	Defined by the programmer for queries that return more than one row.
	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.
	Statement adds new rows to the table.
	Statement removes rows from the table.
	Statement modifies existing rows in the table.

Try It / Solve It

- True or False: When you use DML in a PL/SQL block, Oracle uses explicit cursors to track the data changes.
- SQL%FOUND, SQL%NOTFOUND, and SQL%ROWCOUNT are _____ and are available when you use _____ cursors.

The following questions use a copy of the departments table. Execute the following SQL statement to create the copy table.

```
CREATE TABLE new_depts AS SELECT * FROM departments;
```

3. Examine and run the following PL/SQL code, which obtains and displays the maximum department_id from new_depts. What is the maximum department id?

```
DECLARE
    v_max_deptno new_depts.department_id%TYPE;
BEGIN
    SELECT MAX(department_id) INTO v_max_deptno
        FROM new_depts;
    DBMS_OUTPUT.PUT_LINE('The maximum department id is: ' || v_max_deptno);
END;
```

4. Modify the code to declare two additional variables (assigning a new department name to one of them), by adding the following two lines to your Declaration section:

```
v_dept_name    new_depts.department_name%TYPE := 'A New Department';
v_dept_id       new_depts.department_id%TYPE;
```

5. Modify the code to add 10 to the current maximum department number and assign the result to v_dept_id.
6. Modify the code to include an INSERT statement to insert a new row into the new_depts table, using v_dept_id and v_dept_name to populate the department_id and department_name columns. Insert NULL into the location_id and manager_id columns. Execute your code and confirm that the new row has been inserted.
7. Now modify the code to use SQL%ROWCOUNT to display the number of rows inserted, and execute the block again.
8. Now modify the block, removing the INSERT statement and adding a statement that will UPDATE all rows with location_id = 1700 to location_id = 1400. Execute the block again to see how many rows were updated.

ORACLE

Academy

Database Programming with PL/SQL

3-2

Retrieving Data in PL/SQL

ORACLE
Academy



Objectives

- This lesson covers the following objectives:
 - Recognize the SQL statements that can be directly included in a PL/SQL executable block
 - Construct and execute an INTO clause to hold the values returned by a single-row SQL SELECT statement
 - Construct statements to retrieve data that follow good practice guidelines
 - Construct statements that apply good practice guidelines for naming variables

Purpose

- In this lesson, you learn to embed standard SQL SELECT statements in PL/SQL blocks
- You also learn the importance of following usage guidelines and naming convention guidelines when retrieving data
- Blocks can be a good method for organizing your code
- When you review code written by someone else, it is easier to read chunks of a program than it is to read one long continuous program

SQL Statements in PL/SQL

- You can use the following kinds of SQL statements in PL/SQL:
 - SELECT statements to retrieve data from a database
 - DML statements, such as INSERT, UPDATE, and DELETE, to make changes to the database
 - Transaction control statements, such as COMMIT, ROLLBACK, or SAVEPOINT, to make changes to the database permanent or to discard them
 - Transaction control statements will be covered later and are not available in the iAcademy-hosted APEX environment

DDL/DCL Limitations in PL/SQL

- You cannot use DDL and DCL directly in PL/SQL

Handle Style	Description
DDL	CREATE TABLE, ALTER TABLE, DROP TABLE
DCL	GRANT, REVOKE

- PL/SQL does not directly support DDL statements, such as CREATE TABLE, ALTER TABLE, and DROP TABLE, or DCL statements such as GRANT and REVOKE

DDL/DCL Limitations in PL/SQL

- You cannot directly execute DDL and DCL statements because they are constructed and executed at run time—that is, they are dynamic
- There are times when you may need to run DDL or DCL within PL/SQL
- The recommended way of working with DDL and DCL within PL/SQL is to use Dynamic SQL with the EXECUTE IMMEDIATE statement
- This will be discussed later in the course

SELECT Statements in PL/SQL

- Retrieve data from a database into a PL/SQL variable with a SELECT statement so you can work with the data within PL/SQL

```
SELECT  select_list
INTO    {variable_name[, variable_name]...
        | record_name}
FROM    table
[WHERE condition];
```



Using the INTO Clause

- The INTO clause is mandatory and occurs between the SELECT and FROM clauses
- It is used to specify the names of PL/SQL variables that hold the values that SQL returns from the SELECT clause

```
DECLARE
    v_emp_lname employees.last_name%TYPE;
BEGIN
    SELECT last_name
        INTO v_emp_lname
        FROM employees
       WHERE employee_id = 100;
    DBMS_OUTPUT.PUT_LINE('His last name is ' || v_emp_lname);
END;
```

Retrieving Data in PL/SQL Example

- You must specify one variable for each item selected, and the order of the variables must correspond with the order of the items selected

```
DECLARE
    v_emp_hiredate    employees.hire_date%TYPE;
    v_emp_salary      employees.salary%TYPE;
BEGIN
    SELECT    hire_date, salary
        INTO    v_emp_hiredate, v_emp_salary
        FROM    employees
        WHERE   employee_id = 100;
    DBMS_OUTPUT.PUT_LINE('Hiredate: ' || v_emp_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary: ' || v_emp_salary);
END;
```

Retrieving Data in PL/SQL Embedded Rule

- SELECT statements within a PL/SQL block fall into the ANSI classification of embedded SQL for which the following rule applies: embedded queries must return exactly one row
- A query that returns more than one row or no rows generates an error

```
DECLARE
    v_salary employees.salary%TYPE;
BEGIN
    SELECT salary INTO v_salary
    FROM employees;
    DBMS_OUTPUT.PUT_LINE(' Salary is : ' || v_salary);
END;
```

```
ORA-01422: exact fetch returns more than requested number of rows
```

Retrieving Data in PL/SQL Example

- Return the sum of the salaries for all the employees in the specified department

```
DECLARE
  v_sum_sal NUMBER(10,2);
  v_deptno  NUMBER NOT NULL := 60;
BEGIN
  SELECT SUM(salary) -- group function
    INTO v_sum_sal
   FROM employees
  WHERE department_id = v_deptno;
  DBMS_OUTPUT.PUT_LINE('Dep #60 Salary Total: ' || v_sum_sal);
END;
```

Guidelines for Retrieving Data in PL/SQL

- The guidelines for retrieving data in PL/SQL are:
 - Terminate each SQL statement with a semicolon (;
 - Every value retrieved must be stored in a variable using the INTO clause
 - The WHERE clause is optional and can contain input variables, constants, literals, or PL/SQL expressions
- However, you should fetch only one row and the usage of the WHERE clause is therefore needed in nearly all cases
- Can you think of a case where it isn't needed?

Guidelines for Retrieving Data in PL/SQL

- Specify the same number of variables in the INTO clause as database columns in the SELECT clause
- Be sure the columns and variables are in the same positional order and their data types are compatible
- To insure data type compatibility between columns and variables, declare the receiving variables using %TYPE



Guidelines for Naming Conventions

- In potentially ambiguous SQL statements, the names of database columns take precedence over the names of local variables

```
DECLARE
    v_hire_date      employees.hire_date%TYPE;
    employee_id      employees.employee_id%TYPE := 176;
BEGIN
    SELECT          hire_date
    INTO            v_hire_date
    FROM           employees
    WHERE          employee_id = employee_id;
END;
```

ORA-01422: exact fetch returns more than requested number of rows

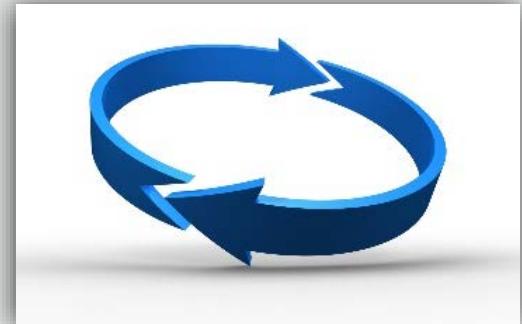
This example raises an unhandled run-time exception because in the WHERE clause, the PL/SQL variable name is the same as that of the database column name in the employees table.

Guidelines for Naming Conventions Example

- What is deleted by the following PL/SQL block?

```
DECLARE
    last_name employees.last_name%TYPE := 'King';
BEGIN
    DELETE FROM emp_dup WHERE last_name = last_name;
END;
```

- Does it remove the row where the employee's last name is King?



Guidelines for Naming Conventions Details

- Guidelines for naming conventions:
 - Use a naming convention to avoid ambiguity in the WHERE clause (for example, ensure all variable names begin with v_)
 - Avoid using database column names as identifiers
 - Errors can occur during execution because PL/SQL checks the database first for a column in the table
 - The names of local variables and formal parameters take precedence over the names of database tables (in a PL/SQL statement)
 - The names of database table columns take precedence over the names of local variables

Summary

- In this lesson, you should have learned how to:
 - Recognize the SQL statements that can be directly included in a PL/SQL executable block
 - Construct and execute an INTO clause to hold the values returned by a single-row SQL SELECT statement
 - Construct statements to retrieve data that follow good practice guidelines
 - Construct statements that apply good practice guidelines for naming variables

ORACLE

Academy

ORACLE

Academy

Database Programming with PL/SQL

3-2

Retrieving Data in PL/SQL

ORACLE
Academy



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Objectives

- This lesson covers the following objectives:
 - Recognize the SQL statements that can be directly included in a PL/SQL executable block
 - Construct and execute an INTO clause to hold the values returned by a single-row SQL SELECT statement
 - Construct statements to retrieve data that follow good practice guidelines
 - Construct statements that apply good practice guidelines for naming variables

Purpose

- In this lesson, you learn to embed standard SQL SELECT statements in PL/SQL blocks
- You also learn the importance of following usage guidelines and naming convention guidelines when retrieving data
- Blocks can be a good method for organizing your code
- When you review code written by someone else, it is easier to read chunks of a program than it is to read one long continuous program

SQL Statements in PL/SQL

- You can use the following kinds of SQL statements in PL/SQL:
 - SELECT statements to retrieve data from a database
 - DML statements, such as INSERT, UPDATE, and DELETE, to make changes to the database
 - Transaction control statements, such as COMMIT, ROLLBACK, or SAVEPOINT, to make changes to the database permanent or to discard them
 - Transaction control statements will be covered later and are not available in the iAcademy-hosted APEX environment

PL/SQL is tightly integrated with SQL and therefore the database.

If using the online APEX environment through iAcademy, all SQL statements are automatically committed. The transaction control statements, COMMIT, ROLLBACK, and SAVEPOINT, are NOT available through iAcademy.

DDL/DCL Limitations in PL/SQL

- You cannot use DDL and DCL directly in PL/SQL

Handle Style	Description
DDL	CREATE TABLE, ALTER TABLE, DROP TABLE
DCL	GRANT, REVOKE

- PL/SQL does not directly support DDL statements, such as CREATE TABLE, ALTER TABLE, and DROP TABLE, or DCL statements such as GRANT and REVOKE

DDL/DCL Limitations in PL/SQL

- You cannot directly execute DDL and DCL statements because they are constructed and executed at run time—that is, they are dynamic
- There are times when you may need to run DDL or DCL within PL/SQL
- The recommended way of working with DDL and DCL within PL/SQL is to use Dynamic SQL with the EXECUTE IMMEDIATE statement
- This will be discussed later in the course

DDL and DCL statements cannot simply run within the executable section of a PL/SQL block of code.

SELECT Statements in PL/SQL

- Retrieve data from a database into a PL/SQL variable with a SELECT statement so you can work with the data within PL/SQL

```
SELECT  select_list
  INTO   {variable_name[, variable_name] ...
          | record_name}
  FROM   table
  [WHERE condition] ;
```



Use the SELECT statement to retrieve data from the database. In the syntax:

select_list is a list of at least one column and can include SQL expressions, row functions, or group functions

variable_name is a scalar variable that holds a retrieved value

record_name is the PL/SQL record that holds the retrieved values

table specifies the database table name

condition is composed of column names, expressions, constants, and comparison operators, including PL/SQL variables and constants

Using the INTO Clause

- The INTO clause is mandatory and occurs between the SELECT and FROM clauses
- It is used to specify the names of PL/SQL variables that hold the values that SQL returns from the SELECT clause

```
DECLARE
    v_emp_lname employees.last_name%TYPE;
BEGIN
    SELECT last_name
        INTO v_emp_lname
        FROM employees
       WHERE employee_id = 100;
    DBMS_OUTPUT.PUT_LINE('His last name is ' || v_emp_lname);
END;
```

ORACLE

Academy

PLSQL 3-2
Retrieving Data in PL/SQL

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

9

Retrieving Data in PL/SQL Example

- You must specify one variable for each item selected, and the order of the variables must correspond with the order of the items selected

```
DECLARE
  v_emp_hiredate  employees.hire_date%TYPE;
  v_emp_salary    employees.salary%TYPE;
BEGIN
  SELECT  hire_date, salary
  INTO    v_emp_hiredate, v_emp_salary
  FROM    employees
  WHERE   employee_id = 100;
  DBMS_OUTPUT.PUT_LINE('Hiredate: ' || v_emp_hiredate);
  DBMS_OUTPUT.PUT_LINE('Salary: ' || v_emp_salary);
END;
```



In the example in the slide, the v_emp_hiredate and v_emp_salary variables are declared in the declarative section of the PL/SQL block. In the executable section, the values of the columns hire_date and salary for the employee with the employee_id 100 is retrieved from the employees table and stored in the v_emp_hiredate and v_emp_salary variables, respectively. Observe how the INTO clause, along with the SELECT clause, retrieves the database column values into the PL/SQL variables.

Note: The SELECT statement is retrieving hire_date and then salary, That means the variables in the INTO clause must be in the same order.

Retrieving Data in PL/SQL Embedded Rule

- SELECT statements within a PL/SQL block fall into the ANSI classification of embedded SQL for which the following rule applies: embedded queries must return exactly one row
- A query that returns more than one row or no rows generates an error

```
DECLARE
    v_salary employees.salary%TYPE;
BEGIN
    SELECT salary INTO v_salary
    FROM employees;
    DBMS_OUTPUT.PUT_LINE(' Salary is : ' || v_salary);
END;
ORA-01422: exact fetch returns more than requested number of rows
```



Academy

PLSQL 3-2
Retrieving Data in PL/SQL

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

11

A SELECT statement with the INTO clause can retrieve only one row at a time. If your requirement is to retrieve multiple rows and operate on the data, then you can make use of explicit cursors. PL/SQL explicit cursors will be discussed in upcoming lessons.

Retrieving Data in PL/SQL Example

- Return the sum of the salaries for all the employees in the specified department

```
DECLARE
  v_sum_sal NUMBER(10,2);
  v_deptno  NUMBER NOT NULL := 60;
BEGIN
  SELECT SUM(salary) -- group function
    INTO v_sum_sal
   FROM employees
  WHERE department_id = v_deptno;
  DBMS_OUTPUT.PUT_LINE('Dep #60 Salary Total: ' || v_sum_sal);
END;
```



In this example, the `v_sum_sal` and `v_deptno` variables are declared in the declarative section of the PL/SQL block. In the executable section, the total salary for the employees in the department with the `department_id` 60 is computed using the SQL group function `SUM`. The calculated total salary is assigned to the `v_sum_sal` variable.

Group functions cannot be used directly in PL/SQL syntax.

For example, the following code does not work:

```
v_sum_sal := SUM(employees.salary);
```

Group functions must be part of a SQL statement within a PL/SQL block.

Guidelines for Retrieving Data in PL/SQL

- The guidelines for retrieving data in PL/SQL are:
 - Terminate each SQL statement with a semicolon (;
 - Every value retrieved must be stored in a variable using the INTO clause
 - The WHERE clause is optional and can contain input variables, constants, literals, or PL/SQL expressions
- However, you should fetch only one row and the usage of the WHERE clause is therefore needed in nearly all cases
- Can you think of a case where it isn't needed?

Answer: when "retrieving" data from the DUAL table as in:

```
DECLARE
  v_date DATE;
BEGIN
  SELECT SYSDATE
    INTO v_date
   FROM DUAL;
  DBMS_OUTPUT.PUT_LINE('The date is ' || v_date);
END;
```

Guidelines for Retrieving Data in PL/SQL

- Specify the same number of variables in the INTO clause as database columns in the SELECT clause
- Be sure the columns and variables are in the same positional order and their data types are compatible
- To insure data type compatibility between columns and variables, declare the receiving variables using %TYPE



Guidelines for Naming Conventions

- In potentially ambiguous SQL statements, the names of database columns take precedence over the names of local variables

```
DECLARE
    v_hire_date    employees.hire_date%TYPE;
    employee_id    employees.employee_id%TYPE := 176;
BEGIN
    SELECT      hire_date
    INTO        v_hire_date
    FROM        employees
    WHERE       employee_id = employee_id;
END;
```

ORA-01422: exact fetch returns more than requested number of rows

This example raises an unhandled run-time exception because in the WHERE clause, the PL/SQL variable name is the same as that of the database column name in the employees table.



PLSQL 3-2
Retrieving Data in PL/SQL

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

15

Try running the SELECT statement from the slide (without the INTO clause) outside of a PL/SQL block and you will see it returns more than one row, which we know violates the embedded query rule.

```
SELECT hire_date FROM employees
WHERE employee_id = employee_id;
```

Guidelines for Naming Conventions Example

- What is deleted by the following PL/SQL block?

```
DECLARE
    last_name employees.last_name%TYPE := 'King';
BEGIN
    DELETE FROM emp_dup WHERE last_name = last_name;
END;
```

- Does it remove the row where the employee's last name is King?



Answer:

The DELETE statement will remove ALL employees from the emp_dup table as opposed to just the row(s) that contain the last name "King." This is because the Oracle server first resolves WHERE clause data items to the table in the FROM clause. Since last_name is in the table, and the last_name column in each row is "equal" to itself, the DELETE statement deletes ALL rows in the emp_dup table. Had the variable in the declarative section been named v_last_name as our naming convention suggests, this mistake would not have happened.

Guidelines for Naming Conventions Details

- Guidelines for naming conventions:

- Use a naming convention to avoid ambiguity in the WHERE clause (for example, ensure all variable names begin with v_)
- Avoid using database column names as identifiers
- Errors can occur during execution because PL/SQL checks the database first for a column in the table
- The names of local variables and formal parameters take precedence over the names of database tables (in a PL/SQL statement)
- The names of database table columns take precedence over the names of local variables

There is no possibility for ambiguity in the SELECT clause because any identifier in the SELECT clause must be a database column name. There is no possibility for ambiguity in the INTO clause because identifiers in the INTO clause must be PL/SQL variables. There is the possibility of ambiguity or confusion only in the WHERE clause.

Summary

- In this lesson, you should have learned how to:
 - Recognize the SQL statements that can be directly included in a PL/SQL executable block
 - Construct and execute an INTO clause to hold the values returned by a single-row SQL SELECT statement
 - Construct statements to retrieve data that follow good practice guidelines
 - Construct statements that apply good practice guidelines for naming variables

ORACLE

Academy

Database Programming with PL/SQL

3-2: Retrieving Data in PL/SQL

Practice Activities

Vocabulary

No new vocabulary for this lesson

Try It / Solve It

1. State whether each of the following SQL statements can be included directly in a PL/SQL block.

Statement	Valid in PL/SQL	Not Valid in PL/SQL
ALTER USER SET password = 'oracle';		
CREATE TABLE test (a NUMBER);		
DROP TABLE test;		
SELECT emp_id INTO v_id FROM employees;		
GRANT SELECT ON employees TO PUBLIC;		
INSERT INTO grocery_items (product_id, brand, description) VALUES (199, 'Coke', 'Soda');		
REVOKE UPDATE ON employees FROM PUBLIC;		
ALTER TABLE employees RENAME COLUMN employee_id TO emp_id;		
DELETE FROM grocery_items WHERE description = 'Soap';		

2. Create a PL/SQL block that selects the maximum department_id in the departments table and stores it in the v_max_deptno variable. Display the maximum department_id. Declare v_max_deptno to be the same datatype as the department_id column. Include a SELECT statement to retrieve the highest department_id from the departments table. Display the variable v_max_deptno.

3. The following code is supposed to display the lowest and highest elevations for a country name entered by the user. However, the code does not work. Fix the code by following the guidelines for retrieving data that you learned in this lesson.

```
DECLARE
    v_country_name    countries.country_name%TYPE := Federative Republic of Brazil;
    v_lowest_elevation countries.lowest_elevation%TYPE;
    v_highest_elevation countries.highest_elevation%TYPE;
BEGIN
    SELECT lowest_elevation, highest_elevation
        FROM countries;
    DBMS_OUTPUT.PUT_LINE('The lowest elevation in '
        || v_country_name || ' is ' || v_lowest_elevation
        || ' and the highest elevation is ' || v_highest_elevation || '.');
END;
```

4. Run the following anonymous block. It should execute successfully.

```
DECLARE
    v_emp_lname    employees.last_name%TYPE;
    v_emp_salary   employees.salary%TYPE;
BEGIN
    SELECT last_name, salary INTO v_emp_lname, v_emp_salary
        FROM employees
        WHERE job_id = 'AD_PRES';
    DBMS_OUTPUT.PUT_LINE(v_emp_lname || ' ' || v_emp_salary);
END;
```

- A. Now modify the block to use 'IT_PROG' instead of 'AD_PRES' and re-run it. Why does it fail this time?
- B. Now modify the block to use 'IT_PRAG' instead of 'IT_PROG' and re-run it. Why does it still fail?

5. Use (but don't execute) the following code to answer this question:

```
DECLARE
    last_name VARCHAR2(25) := 'Fay';
BEGIN
    UPDATE emp_dup
        SET first_name = 'Jennifer'
        WHERE last_name = last_name;
END;
```

What do you think would happen if you ran the above code? Write your answer here and then follow the steps below to test your theory.

- A. Create a table called emp_dup that is a duplicate of employees.
 - B. Select the first_name and last_name values for all rows in emp_dup.
 - C. Run the anonymous PLSQL block shown at the beginning of this question.
 - D. Select the first_name and last_name columns from emp_dup again to confirm your theory.
 - E. Now we are going to correct the code so that it changes only the first name for the employee whose last name is "Fay". Drop emp_dup and re-create it.
 - F. Modify the code shown at the beginning of this question so that for the employee whose last_name = "Fay", the first_name is updated to Jennifer. Run your modified block.
 - G. Confirm that your update statement worked correctly.
6. Is it possible to name a column in a table the same name as the table? Create a table to test this question. Don't forget to populate the table with data.
7. Is it possible to have a column, table, and variable, all with the same name? Using the table you created in the question above, write a PL/SQL block to test your theory.

ORACLE

Academy

Database Programming with PL/SQL

3-1

Review of SQL DML

ORACLE
Academy



Objectives

- This lesson covers the following objectives:
 - Insert data into a database table
 - Update data in a database table
 - Delete data from a database table
 - Merge data into a database table

Purpose

- When you create, change, or delete an object in a database, the language you use is referred to as data definition language (DDL)
- When you change data within an object (inserting rows, deleting rows, changing column values within a row), it is called data manipulation language (DML)
- This lesson reviews basic SQL DML statements
- Later, you will use DML statements in your PL/SQL code to modify data

Data Manipulation Language (DML)

- You can use DML commands to add rows, delete rows and modify the data within a row
- The DML commands are INSERT, UPDATE, DELETE, and MERGE



Data Manipulation Language (DML)

- In online/hosted versions of Application Express at iAcademy, the default setting for the APEX SQL command processor is that AUTOCOMMIT is turned on
- This means each statement executed will be automatically committed as it is executed
- There is no user control to disable AUTOCOMMIT

Data Manipulation Language (DML)

- If you have a locally installed version of APEX, there is an AUTOCOMMIT checkbox that can be unchecked to disable AUTOCOMMIT
- When AUTOCOMMIT is disabled, DML commands produce permanent changes only when the COMMIT command is issued either within a PL/SQL block or by itself
- If the user logs off normally or closes the browser window before executing a COMMIT command, the changes will be rolled back

INSERT

- You use the INSERT statement to add new rows to a table
- It requires at least two items:
 - The name of the table
 - Values for each of the columns in the table
 - (Optional, but recommended) The names of the columns that will receive a value when the row is inserted
 - If this option is used, then the list of values must match the list of columns
- At the very least, you must list and initialize with a value each column that can not be NULL

INSERT Explicit Syntax

- The syntax shown explicitly lists each column in the table that can not be NULL plus the column for the employee's first name
- The values for each column must be listed in the same order as the columns are listed

```
INSERT INTO employees (employee_id, first_name,
    last_name, email, hire_date, job_id)
VALUES (305, 'Kareem', 'Naser',
    'naserk@oracle.com', SYSDATE, 'SR_SA_REP');
```

INSERT Explicit Syntax

- When inserting a row into a table, you must provide a value for each column that can not be NULL
- You also could insert the new employee by listing all the columns in the table and including NULL in the list of values that are not known at the moment and can be updated later

```
INSERT INTO employees (employee_id, first_name, last_name,
    email, phone_number, hire_date, job_id, salary,
    commission_pct, manager_id, department_head)
VALUES (305, 'Kareem', 'Naser', 'naserk@oracle.com',
    '111-222-3333', SYSDATE, 'SR_SA_REP', 7000, NULL, NULL, NULL);
```

INSERT Implicit Syntax

- Another way to insert values in a table is to implicitly add them without listing the column names
- The values must match the order in which the columns appear in the table and a value must be provided for each column

```
INSERT INTO employees
VALUES (305, 'Kareem', 'Naser',
'naserk@oracle.com', '111-222-3333', SYSDATE,
'SR_SA_REP', 7000, NULL, NULL, NULL, NULL);
```

UPDATE

- The UPDATE statement is used to modify existing rows in a table
- It requires at least three items:
 - The name of the table
 - The name of at least one column to modify
 - A value for each column being modified
 - (Optional) a WHERE clause that identifies the row or rows to be modified
 - If the WHERE clause is omitted, ALL rows will be modified
 - Be very careful when running an UPDATE statement without a WHERE clause!

UPDATE Syntax

- A single column can be modified
- The WHERE clause identifies the row to be modified

It is recommended that the UPDATE statement be on a line of its own

```
UPDATE employees  
    SET salary = 11000  
 WHERE employee_id = 176;
```

- An UPDATE statement can modify multiple columns

```
UPDATE employees  
    SET salary = 11000, commission_pct = .3  
 WHERE employee_id = 176;
```

DELETE

- You use the DELETE statement to remove existing rows in a table
- The statement requires at least one item:
 - The name of the table
 - (Optional) a WHERE clause that identifies the row or rows to be deleted
- Please note, if the WHERE clause is omitted, ALL rows will be deleted
- Be very careful when running a DELETE statement without a WHERE clause
- Situations requiring that will be rare

DELETE Syntax

- The WHERE clause identifies the row or rows to be deleted

```
DELETE FROM employees  
WHERE employee_id = 149;
```

```
DELETE FROM employees  
WHERE department_id = 80;
```

- Be careful with the DELETE statement
- If the WHERE clause is omitted, ALL rows will be deleted
- Very few situations will require a DELETE statement without a WHERE clause

MERGE

- The MERGE statement will INSERT a new row into a target table or UPDATE existing data in a target table, based on a comparison of the data in the two tables
- The WHEN clause determines the action to be taken
- For example, if a specific row exists in the source table, but there is no matching row in the target table, the row from the source table will be inserted into the target table
- If the matching row does exist in the target table, but some data in the source table is different for that row, the target table may be updated with the different data

MERGE Usage and Syntax

- To set up our MERGE example, consider a situation where we need to calculate annual bonuses for employees earning less than \$10,000 USD
- First, we create a table called bonuses

```
CREATE TABLE bonuses
(employee_id NUMBER(6,0) NOT NULL,
bonus NUMBER(8,2) DEFAULT 0);
```



MERGE Usage and Syntax

- We then populate the table with the employee ids of all employees with a salary less than \$10,000 USD

```
INSERT INTO bonuses(employee_id)
  (SELECT employee_id FROM employees
 WHERE salary < 10000);
```



MERGE Usage and Syntax

- Each employee with a salary less than \$10,000 USD is to receive a bonus of 5% of their salary
- To use the salary column from the employees table to calculate the amount of the bonus and update the bonus column in the bonuses table, we use the following MERGE statement

```
MERGE INTO bonuses b
  USING employees e
  ON (b.employee_id = e.employee_id)
WHEN MATCHED THEN
  UPDATE SET b.bonus = e.salary * .05;
```

MERGE Usage and Syntax

- The resulting bonus table will look something like this:

EMPLOYEE_ID	BONUS
200	220
206	415
176	430
178	350
124	290

- If you attempt to duplicate this example, your results may vary depending on the data in your employees table

Terminology

- Key terms used in this lesson included:

- Data Definition Language (DDL)
 - Data Manipulation Language (DML)
 - DELETE
 - INSERT
 - MERGE
 - UPDATE

Summary

- In this lesson, you should have learned how to:
 - Insert data into a database table
 - Update data in a database table
 - Delete data from a database table
 - Merge data into a database table

ORACLE

Academy

ORACLE

Academy

Database Programming with PL/SQL

3-1

Review of SQL DML

ORACLE
Academy



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Objectives

- This lesson covers the following objectives:
 - Insert data into a database table
 - Update data in a database table
 - Delete data from a database table
 - Merge data into a database table

Purpose

- When you create, change, or delete an object in a database, the language you use is referred to as data definition language (DDL)
- When you change data within an object (inserting rows, deleting rows, changing column values within a row), it is called data manipulation language (DML)
- This lesson reviews basic SQL DML statements
- Later, you will use DML statements in your PL/SQL code to modify data

Data Manipulation Language (DML)

- You can use DML commands to add rows, delete rows and modify the data within a row
- The DML commands are INSERT, UPDATE, DELETE, and MERGE



Data Manipulation Language (DML)

- In online/hosted versions of Application Express at iAcademy, the default setting for the APEX SQL command processor is that AUTOCOMMIT is turned on
- This means each statement executed will be automatically committed as it is executed
- There is no user control to disable AUTOCOMMIT

Data Manipulation Language (DML)

- If you have a locally installed version of APEX, there is an AUTOCOMMIT checkbox that can be unchecked to disable AUTOCOMMIT
- When AUTOCOMMIT is disabled, DML commands produce permanent changes only when the COMMIT command is issued either within a PL/SQL block or by itself
- If the user logs off normally or closes the browser window before executing a COMMIT command, the changes will be rolled back

INSERT

- You use the INSERT statement to add new rows to a table
- It requires at least two items:
 - The name of the table
 - Values for each of the columns in the table
 - (Optional, but recommended) The names of the columns that will receive a value when the row is inserted
 - If this option is used, then the list of values must match the list of columns
- At the very least, you must list and initialize with a value each column that can not be NULL

INSERT Explicit Syntax

- The syntax shown explicitly lists each column in the table that can not be NULL plus the column for the employee's first name
- The values for each column must be listed in the same order as the columns are listed

```
INSERT INTO employees (employee_id, first_name,
    last_name, email, hire_date, job_id)
VALUES (305, 'Kareem', 'Naser',
    'naserk@oracle.com', SYSDATE, 'SR_SA_REP');
```

When inserting a row into a table, you must provide a value for each column that can not be NULL. You also could insert the new employee by listing all the columns in the table and including NULL in the list of values that are not known at the moment and can be updated later. For example:

```
INSERT INTO employees (employee_id, first_name, last_name, email, phone_number,
    hire_date, job_id, salary, commission_pct, manager_id, department_head)
VALUES (305, 'Kareem', 'Naser', 'naserk@oracle.com', '111-222-3333', SYSDATE, 'SR_SA_REP',
7000, NULL, NULL, NULL);
```

Notice the hire_date column is receiving its value from the SYSDATE function. A column's value can be an explicit value or the result of a function. It also can be the result of a subquery or the result of an expression.

INSERT Explicit Syntax

- When inserting a row into a table, you must provide a value for each column that can not be NULL
- You also could insert the new employee by listing all the columns in the table and including NULL in the list of values that are not known at the moment and can be updated later

```
INSERT INTO employees (employee_id, first_name, last_name,
email, phone_number, hire_date, job_id, salary,
commission_pct, manager_id, department_head)
VALUES (305, 'Kareem', 'Naser', 'naserk@oracle.com',
'111-222-3333', SYSDATE, 'SR_SA_REP', 7000, NULL, NULL, NULL);
```

Notice the hire_date column is receiving its value from the SYSDATE function. A column's value can be an explicit value or the result of a function. It also can be the result of a subquery or the result of an expression.

INSERT Implicit Syntax

- Another way to insert values in a table is to implicitly add them without listing the column names
- The values must match the order in which the columns appear in the table and a value must be provided for each column

```
INSERT INTO employees
VALUES (305, 'Kareem', 'Naser',
'naserk@oracle.com', '111-222-3333', SYSDATE,
'SR_SA REP', 7000, NULL, NULL, NULL, NULL);
```



Although this syntax works, it is not recommended. In this example, the EMPLOYEES table has the same eleven columns as listed in the explicit example. But what if the DBA later adds a twelfth column using a DDL ALTER TABLE statement? The INSERT statement above would no longer work. It is good programming practice to explicitly identify the columns when inserting a new row as done on the previous slides.

UPDATE

- The UPDATE statement is used to modify existing rows in a table
- It requires at least three items:
 - The name of the table
 - The name of at least one column to modify
 - A value for each column being modified
 - (Optional) a WHERE clause that identifies the row or rows to be modified
 - If the WHERE clause is omitted, ALL rows will be modified
 - Be very careful when running an UPDATE statement without a WHERE clause!

UPDATE Syntax

It is recommended that the UPDATE statement be on a line of its own

- A single column can be modified
- The WHERE clause identifies the row to be modified

```
UPDATE employees  
    SET salary = 11000  
 WHERE employee_id = 176;
```

- An UPDATE statement can modify multiple columns

```
UPDATE employees  
    SET salary = 11000, commission_pct = .3  
 WHERE employee_id = 176;
```

DELETE

- You use the DELETE statement to remove existing rows in a table
- The statement requires at least one item:
 - The name of the table
 - (Optional) a WHERE clause that identifies the row or rows to be deleted
- Please note, if the WHERE clause is omitted, ALL rows will be deleted
- Be very careful when running a DELETE statement without a WHERE clause
- Situations requiring that will be rare

DELETE Syntax

- The WHERE clause identifies the row or rows to be deleted

```
DELETE FROM employees  
WHERE employee_id = 149;
```

```
DELETE FROM employees  
WHERE department_id = 80;
```

- Be careful with the DELETE statement
- If the WHERE clause is omitted, ALL rows will be deleted
- Very few situations will require a DELETE statement without a WHERE clause

The first example deletes one row because employee_id is the primary key.

The second example may delete multiple rows. It will delete all rows that have the value 80 in the department_id column.

MERGE

- The MERGE statement will INSERT a new row into a target table or UPDATE existing data in a target table, based on a comparison of the data in the two tables
- The WHEN clause determines the action to be taken
- For example, if a specific row exists in the source table, but there is no matching row in the target table, the row from the source table will be inserted into the target table
- If the matching row does exist in the target table, but some data in the source table is different for that row, the target table may be updated with the different data

MERGE Usage and Syntax

- To set up our MERGE example, consider a situation where we need to calculate annual bonuses for employees earning less than \$10,000 USD
- First, we create a table called bonuses

```
CREATE TABLE bonuses
  (employee_id NUMBER(6,0) NOT NULL,
   bonus NUMBER(8,2) DEFAULT 0);
```



MERGE Usage and Syntax

- We then populate the table with the employee ids of all employees with a salary less than \$10,000 USD

```
INSERT INTO bonuses(employee_id)
  (SELECT employee_id FROM employees
 WHERE salary < 10000);
```



ORACLE
Academy

PLSQL 3-1
Review of SQL DML

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

18

MERGE Usage and Syntax

- Each employee with a salary less than \$10,000 USD is to receive a bonus of 5% of their salary
- To use the salary column from the employees table to calculate the amount of the bonus and update the bonus column in the bonuses table, we use the following MERGE statement

```
MERGE INTO bonuses b
  USING employees e
  ON (b.employee_id = e.employee_id)
WHEN MATCHED THEN
  UPDATE SET b.bonus = e.salary * .05;
```

MERGE Usage and Syntax

- The resulting bonus table will look something like this:

EMPLOYEE_ID	BONUS
200	220
206	415
176	430
178	350
124	290

- If you attempt to duplicate this example, your results may vary depending on the data in your employees table

Terminology

- Key terms used in this lesson included:
 - Data Definition Language (DDL)
 - Data Manipulation Language (DML)
 - DELETE
 - INSERT
 - MERGE
 - UPDATE

- Data Definition Language (DDL) -- When you create, change, or delete an object in a database.
- Data Manipulation Language (DML) -- When you change data in an object (for example, by inserting or deleting rows).
- DELETE -- Statement used to remove existing rows in a table.
- INSERT -- Statement used to add new rows to a table.
- MERGE -- Statement used to INSERT and/or UPDATE a target table, based on matching values in a source table.
- UPDATE -- Statement used to modify existing rows in a table.

Summary

- In this lesson, you should have learned how to:
 - Insert data into a database table
 - Update data in a database table
 - Delete data from a database table
 - Merge data into a database table

ORACLE

Academy

Database Programming with PL/SQL

3-1: Review of SQL DML

Practice Activities

Vocabulary

Identify the vocabulary word for each definition below:

	Statement used to remove existing rows in a table.
	Statement used to add new rows to a table.
	Statement used to INSERT and/or UPDATE a target table, based on matching values in a source table.
	Statement used to modify existing rows in a table.
	When you create, change, or delete an object in a database.
	When you change data in an object (for example, by inserting or deleting rows).

Try It / Solve It

- Evaluate the following SQL statement.

DELETE FROM students;

This SQL statement will:

- A. Not execute due to wrong syntax
- B. Delete the first row from STUDENTS
- C. Delete all rows from STUDENTS
- D. None of the above

2. Evaluate the following SQL statement.

```
INSERT INTO STUDENTS (id, last_name, first_name)
VALUES (29,'Perez','Jessica');
```

This SQL statement:

- A. Does an explicit insert
- B. Does an implicit insert

Use the following table for questions 3 through 8.

grocery_items		
product_id	brand	description
110	Colgate	Toothpaste
111	Ivory	Soap
112	Heinz	Ketchup

3. Write a SQL statement to create the above table.
4. Write and execute three SQL statements to explicitly add the above data to the table.
5. Write and execute a SQL statement that will explicitly add your favorite beverage to the table.
6. Write and execute a SQL statement that modifies the description for Heinz ketchup to “tomato catsup”.
7. Write and execute a SQL statement that will implicitly add your favorite candy to the table.
8. Write and execute a SQL statement that changes the soap brand from “Ivory” to “Dove.”

Use the following table for questions 9 through 14.

new_items		
product_id	brand	description
110	Colgate	Dental paste
175	Dew	Soda
275	Palmolive	Dish detergent

9. Write and execute SQL statements to create the new_items table and populate it with the data in the table.
10. Write a SQL statement that will update the grocery_items table with the brand and description from the new_items table when the product ID values match. If they don't match, add a new row to the grocery_items table. DO NOT EXECUTE YOUR STATEMENT YET.
11. How many rows will be updated by the SQL statement in question 10?
12. How many rows will be inserted by the SQL statement in question 10?
13. Which of the following is true about the SQL statement in question 10?
 - A. new_items is the source table and grocery_items is the target table.
 - B. grocery_items is the source table and new_items is the target table.
14. Execute the SQL statement you wrote in question 10, and then SELECT all data from the grocery_items table to verify your answers to questions 11 and 12.

ORACLE

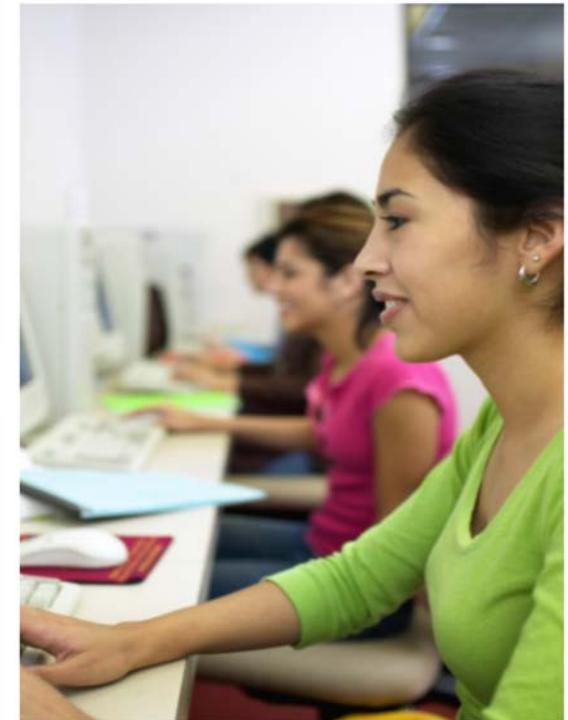
Academy

Database Programming with PL/SQL

3-4

Using Transaction Control Statements

ORACLE
Academy



Objectives

- This lesson covers the following objectives:
 - Define a transaction and provide an example
 - Construct and execute a transaction control statement in PL/SQL
 - Since Oracle Application Express automatically commits changes, the following information will be presented as if you were issuing the commands in an installed/local environment with the ability to use COMMIT and ROLLBACK

Purpose

- In this lesson, you learn how to include transaction control statements such as COMMIT, ROLLBACK, and SAVEPOINT in PL/SQL
- Transactions often have multiple parts or steps
- If something happens to prevent, say, the third step from completing, database integrity is maintained because steps one and two didn't actually change the database because a COMMIT statement was never issued
- ROLLBACK and SAVEPOINT statements give you additional control in processing transactions

Database Transaction

- A transaction is an inseparable list of database operations that must be executed either in its entirety or not at all
- Transactions maintain data integrity and guarantee that the database is always in a consistent state



Example of a Transaction

- To illustrate the concept of a transaction, consider a banking database
- When a bank customer transfers money from a savings account to a checking account, the transaction can consist of three separate operations:

Decrease savings
account
balance.

Increase checking
account
balance.

Record the
transaction in the
transaction
journal.

Transaction

Example of a Transaction

- What would happen if there were insufficient funds in the savings account?
- Would the funds still be added to the checking account?
- Would an entry be logged in the transaction journal?

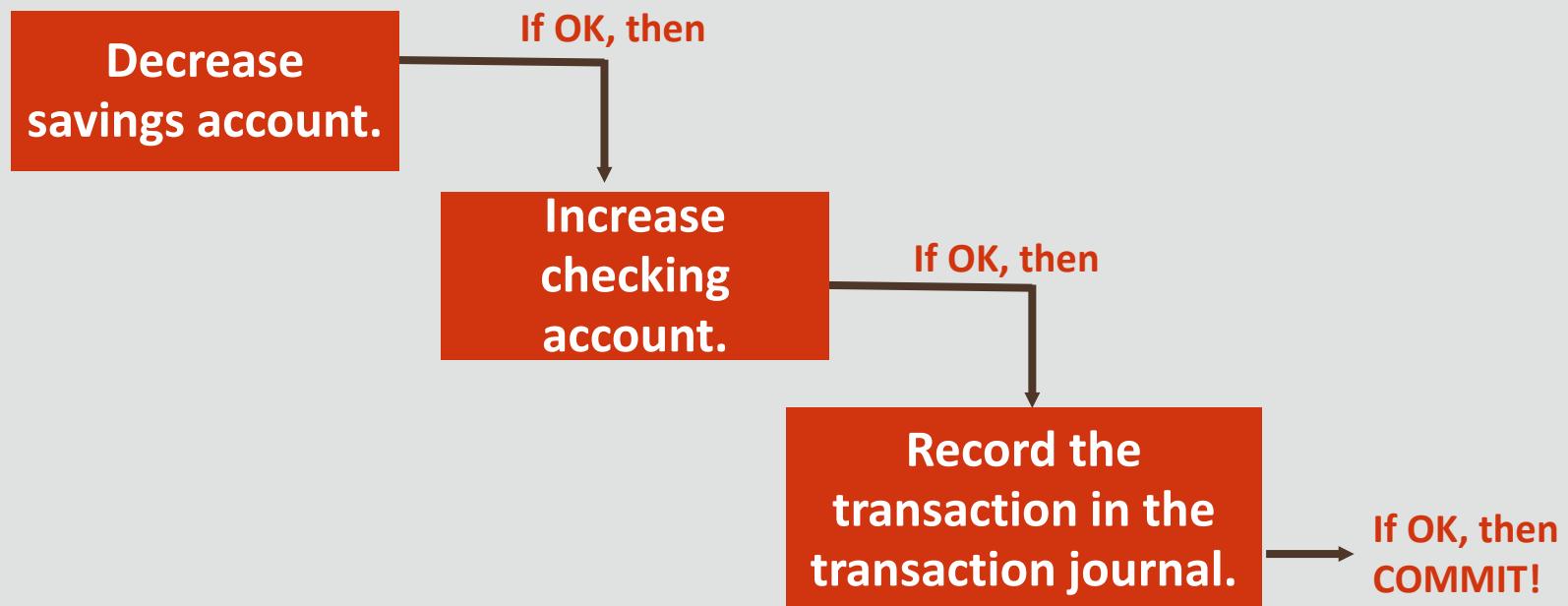
Example of a Transaction

- What do you think should happen?

Decrease savings account balance.	<pre>UPDATE savings_accounts SET balance = balance - 500 WHERE account = 3209;</pre>
Increase checking account balance.	<pre>UPDATE checking_accounts SET balance = balance + 500 WHERE account = 3208;</pre>
Record the transaction in the transaction journal.	<pre>INSERT INTO journal VALUES (journal_seq.NEXTVAL, '1B' 3209, 3208, 500);</pre>

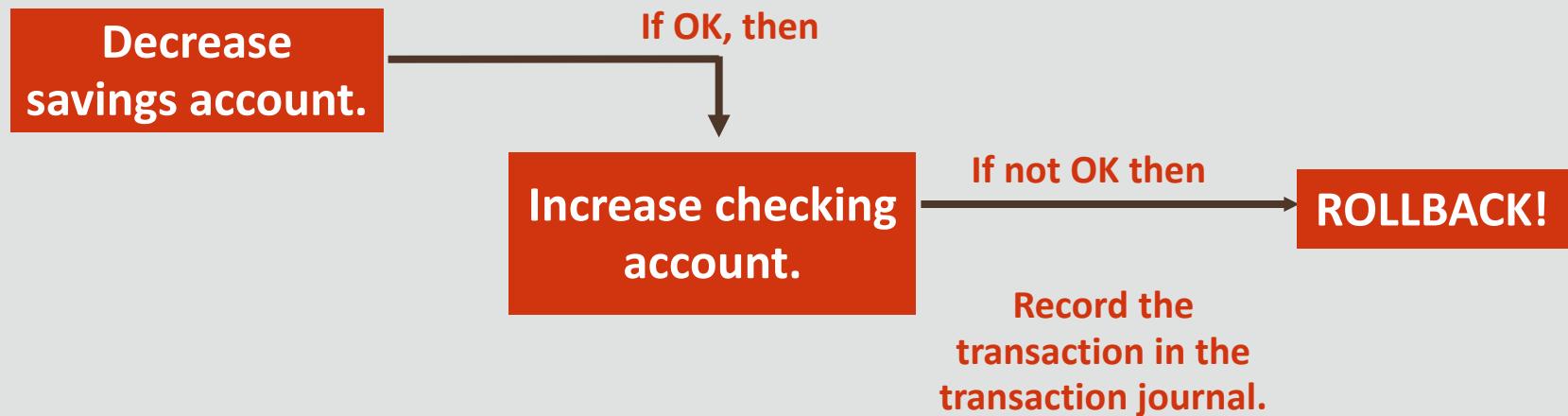
Example of a Transaction

- If all three SQL statements can be performed to maintain the accounts in proper balance, the effects of the transaction can be committed, or applied to the database tables



Example of a Transaction

- However, if a problem, such as insufficient funds, invalid account number, or a hardware failure prevents one or two of the statements in the transaction from completing, the entire transaction must be rolled back (reversed out) so that the balance of all accounts is correct.



Transaction Control Statements

- You use transaction control statements to make the changes to the database permanent or to discard them
- The three main transaction control statements are:
 - COMMIT
 - ROLLBACK
 - SAVEPOINT
- The transaction control commands are valid in PL/SQL and therefore can be used directly in the executable or exception section of a PL/SQL block

COMMIT

- COMMIT is used to make the database changes permanent
- If a transaction ends with a COMMIT statement, all the changes made to the database during that transaction are made permanent

```
BEGIN  
    INSERT INTO pairstable VALUES (1, 2);  
    COMMIT;  
END;
```

- The keyword END signals the end of a PL/SQL block, not the end of a transaction
- A transaction may include multiple PL/SQL blocks and nested PL/SQL blocks

ROLLBACK

- ROLLBACK is for discarding any changes that were made to the database after the last COMMIT
- If the transaction fails, or ends with a ROLLBACK, then none of the statements take effect
- In the example, only the second INSERT statement adds a row of data

```
BEGIN
    INSERT INTO pairstable VALUES (3, 4);
    ROLLBACK;
    INSERT INTO pairstable VALUES (5, 6);
    COMMIT;
END;
```

SAVEPOINT

- SAVEPOINT is used to mark an intermediate point in transaction processing
- ROLLBACK is used to return the data values to the point of a SAVEPOINT

```
BEGIN
    INSERT INTO pahtable VALUES (7, 8);
    SAVEPOINT my_sp_1;
    INSERT INTO pahtable VALUES (9, 10);
    SAVEPOINT my_sp_2;
    INSERT INTO pahtable VALUES (11, 12);
    ROLLBACK to my_sp_1;
    INSERT INTO pahtable VALUES (13, 14);
    COMMIT;
END;
```

Terminology

- Key terms used in this lesson included:
 - COMMIT
 - END
 - ROLLBACK
 - SAVEPOINT
 - Transaction

Summary

- In this lesson, you should have learned how to:
 - Define a transaction and provide an example
 - Construct and execute a transaction control statement in PL/SQL

ORACLE

Academy

ORACLE

Academy

Database Programming with PL/SQL

3-4

Using Transaction Control Statements

ORACLE
Academy



Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

Objectives

- This lesson covers the following objectives:
 - Define a transaction and provide an example
 - Construct and execute a transaction control statement in PL/SQL
 - Since Oracle Application Express automatically commits changes, the following information will be presented as if you were issuing the commands in an installed/local environment with the ability to use COMMIT and ROLLBACK

Purpose

- In this lesson, you learn how to include transaction control statements such as COMMIT, ROLLBACK, and SAVEPOINT in PL/SQL
- Transactions often have multiple parts or steps
- If something happens to prevent, say, the third step from completing, database integrity is maintained because steps one and two didn't actually change the database because a COMMIT statement was never issued
- ROLLBACK and SAVEPOINT statements give you additional control in processing transactions

Database Transaction

- A transaction is an inseparable list of database operations that must be executed either in its entirety or not at all
- Transactions maintain data integrity and guarantee that the database is always in a consistent state



Example of a Transaction

- To illustrate the concept of a transaction, consider a banking database
- When a bank customer transfers money from a savings account to a checking account, the transaction can consist of three separate operations:

Decrease savings
account
balance.

Increase checking
account
balance.

Record the
transaction in the
transaction
journal.

Transaction

Example of a Transaction

- What would happen if there were insufficient funds in the savings account?
- Would the funds still be added to the checking account?
- Would an entry be logged in the transaction journal?

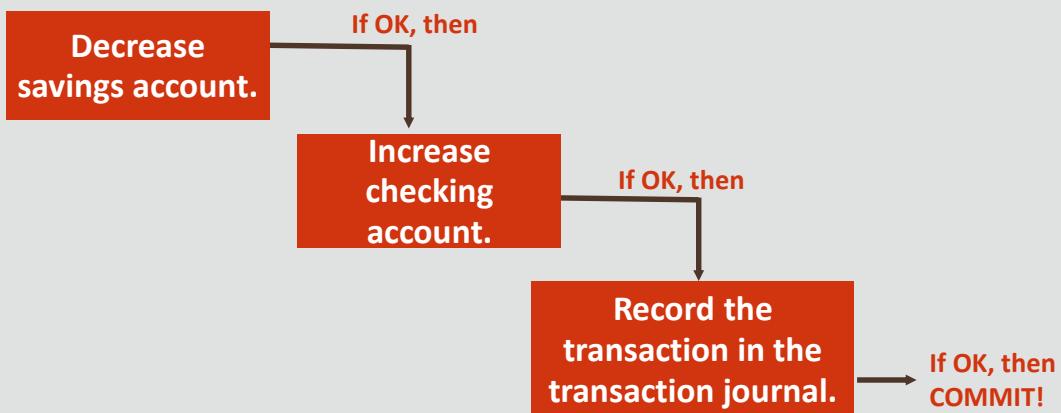
Example of a Transaction

- What do you think should happen?

Decrease savings account balance.	<pre>UPDATE savings_accounts SET balance = balance - 500 WHERE account = 3209;</pre>
Increase checking account balance.	<pre>UPDATE checking_accounts SET balance = balance + 500 WHERE account = 3208;</pre>
Record the transaction in the transaction journal.	<pre>INSERT INTO journal VALUES (journal_seq.NEXTVAL, '1B' 3209, 3208, 500);</pre>

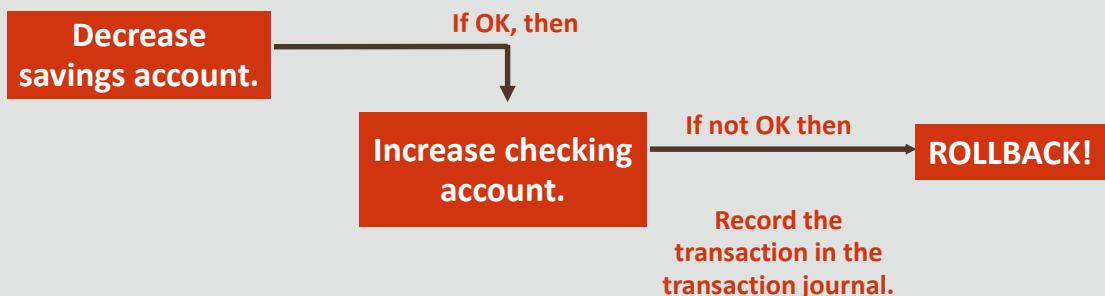
Example of a Transaction

- If all three SQL statements can be performed to maintain the accounts in proper balance, the effects of the transaction can be committed, or applied to the database tables



Example of a Transaction

- However, if a problem, such as insufficient funds, invalid account number, or a hardware failure prevents one or two of the statements in the transaction from completing, the entire transaction must be rolled back (reversed out) so that the balance of all accounts is correct.



Transaction Control Statements

- You use transaction control statements to make the changes to the database permanent or to discard them
- The three main transaction control statements are:
 - COMMIT
 - ROLLBACK
 - SAVEPOINT
- The transaction control commands are valid in PL/SQL and therefore can be used directly in the executable or exception section of a PL/SQL block

Remember, in the online, iAcademy Application Express environment, all statements are automatically committed. This will not be the case with databases in a production environment.

COMMIT

- COMMIT is used to make the database changes permanent
- If a transaction ends with a COMMIT statement, all the changes made to the database during that transaction are made permanent

```
BEGIN  
    INSERT INTO pahtable VALUES (1, 2);  
    COMMIT;  
END;
```

- The keyword END signals the end of a PL/SQL block, not the end of a transaction
- A transaction may include multiple PL/SQL blocks and nested PL/SQL blocks

ROLLBACK

- ROLLBACK is for discarding any changes that were made to the database after the last COMMIT
- If the transaction fails, or ends with a ROLLBACK, then none of the statements take effect
- In the example, only the second INSERT statement adds a row of data

```
BEGIN
    INSERT INTO pairtable VALUES (3, 4);
    ROLLBACK;
    INSERT INTO pairtable VALUES (5, 6);
    COMMIT;
END;
```

ROLLBACK and SAVEPOINT are useful commands, and a major component of real-world database programming. They are used to restore the data to its original values, when and if, the transaction cannot complete all of its steps.

SAVEPOINT

- SAVEPOINT is used to mark an intermediate point in transaction processing
- ROLLBACK is used to return the data values to the point of a SAVEPOINT

```
BEGIN
    INSERT INTO pairtable VALUES (7, 8);
    SAVEPOINT my_sp_1;
    INSERT INTO pairtable VALUES (9, 10);
    SAVEPOINT my_sp_2;
    INSERT INTO pairtable VALUES (11, 12);
    ROLLBACK to my_sp_1;
    INSERT INTO pairtable VALUES (13, 14);
    COMMIT;
END;
```



PLSQL 3-4
Using Transaction Control Statements

Copyright © 2020, Oracle and/or its affiliates. All rights reserved.

14

SAVEPOINTS are useful in application programs. If a procedure contains several functions, then you can create a SAVEPOINT before each function begins. Then, if a function fails, it is easy to return the data to its state before the function began and re-run the function with revised parameters or perform a recovery action.

Terminology

- Key terms used in this lesson included:
 - COMMIT
 - END
 - ROLLBACK
 - SAVEPOINT
 - Transaction

- COMMIT – Statement used to make database changes permanent.
- END – Keyword used to signal the end of a PL/SQL block, not the end of a transaction.
- ROLLBACK – Used for discarding any changes that were made to the database after the last COMMIT.
- SAVEPOINT – Used to mark an intermediate point in transaction processing.
- Transaction – An inseparable list of database operations, which must be executed either in its entirety or not at all.

Summary

- In this lesson, you should have learned how to:
 - Define a transaction and provide an example
 - Construct and execute a transaction control statement in PL/SQL

ORACLE

Academy

Database Programming with PL/SQL

3-4: Using Transaction Control Statements

Practice Activities

Vocabulary

Identify the vocabulary word for each definition below:

	An inseparable list of database operations, which must be executed either in its entirety or not at all.
	Used for discarding any changes that were made to the database after the last COMMIT.
	Used to mark an intermediate point in transaction processing.
	Keyword used to signal the end of a PL/SQL block, not the end of a transaction.
	Statement used to make database changes permanent.

Try It / Solve It

Because our online version of Oracle Application Express (APEX) automatically commits changes as the code runs, the following activities will NOT work as intended unless you are using an installed/local APEX environment.

1. How many transactions are shown in the following code? Explain your reasoning.

```
BEGIN
    INSERT INTO my_savings (account_id, amount)
        VALUES (10377, 200);
    INSERT INTO my_checking (account_id, amount)
        VALUES (10378, 100);
END;
```

Because our online version of Oracle Application Express (APEX) automatically commits changes as the code runs, the following activities will NOT work as intended unless you are using an installed/local APEX environment.

2. Create the endangered species table by running the following statement in Application Express:

Unless you are using an installed/local APEX environment, there is no reason to run this code. If you are using our online version of Oracle Application Express (APEX), you should pretend this code runs successfully before you try to answer the next question.

```
CREATE TABLE endangered_species
(species_id      NUMBER(4) CONSTRAINT es_spec_pk PRIMARY KEY,
 common_name    VARCHAR2(30) CONSTRAINT es_com_name_nn NOT NULL,
 scientific_name VARCHAR2(30) CONSTRAINT es_sci_name_nn NOT NULL);
```

3. Examine the following block of code. If you were to run this block, what data do you think would be saved in the database?

Unless you are using an installed/local APEX environment, there is no reason to run this code. If you are using our online version of Oracle Application Express (APEX), you should pretend this code runs successfully to answer the remaining questions.

```
BEGIN
  INSERT INTO endangered_species
    VALUES (100, 'Polar Bear', 'Ursus maritimus');
  SAVEPOINT sp_100;
  INSERT INTO endangered_species
    VALUES (200, 'Spotted Owl', 'Strix occidentalis');
  SAVEPOINT sp_200;
  INSERT INTO endangered_species
    VALUES (300, 'Asiatic Black Bear', 'Ursus thibetanus');
  ROLLBACK TO sp_100;
  COMMIT;
END;
```

4. Run the block above to test your theory. Confirm your projected data was added.

Unless you are using an installed/local APEX environment, you should skip this question. The block above will NOT run as intended in our online version of Oracle Application Express (APEX) because it automatically commits changes as the code runs.

5. Examine the following block. If you were to run this block, what data do you think would be saved in the database? Run the block to test your theory.

Because our online version of Oracle Application Express (APEX) automatically commits changes as the code runs, the following block will NOT work as intended unless you are using an installed/local APEX environment.

```
BEGIN
    INSERT INTO endangered_species
        VALUES (400, 'Blue Gound Beetle', 'Carabus intricatus');
    SAVEPOINT sp_400;
    INSERT INTO endangered_species
        VALUES (500, 'Little Spotted Cat', 'Leopardus tigrinus');
    ROLLBACK;
    INSERT INTO endangered_species
        VALUES (600, 'Veined Tongue-Fern', 'Elaphoglossum nervosum');
    ROLLBACK TO sp_400;
END;
```

Database Programming with PL/SQL

3-3

Manipulating Data in PL/SQL

ORACLE
Academy



Objectives

- This lesson covers the following objectives:
 - 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

Purpose

- You have 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
- DML statements will help you perform a task on more than a single row

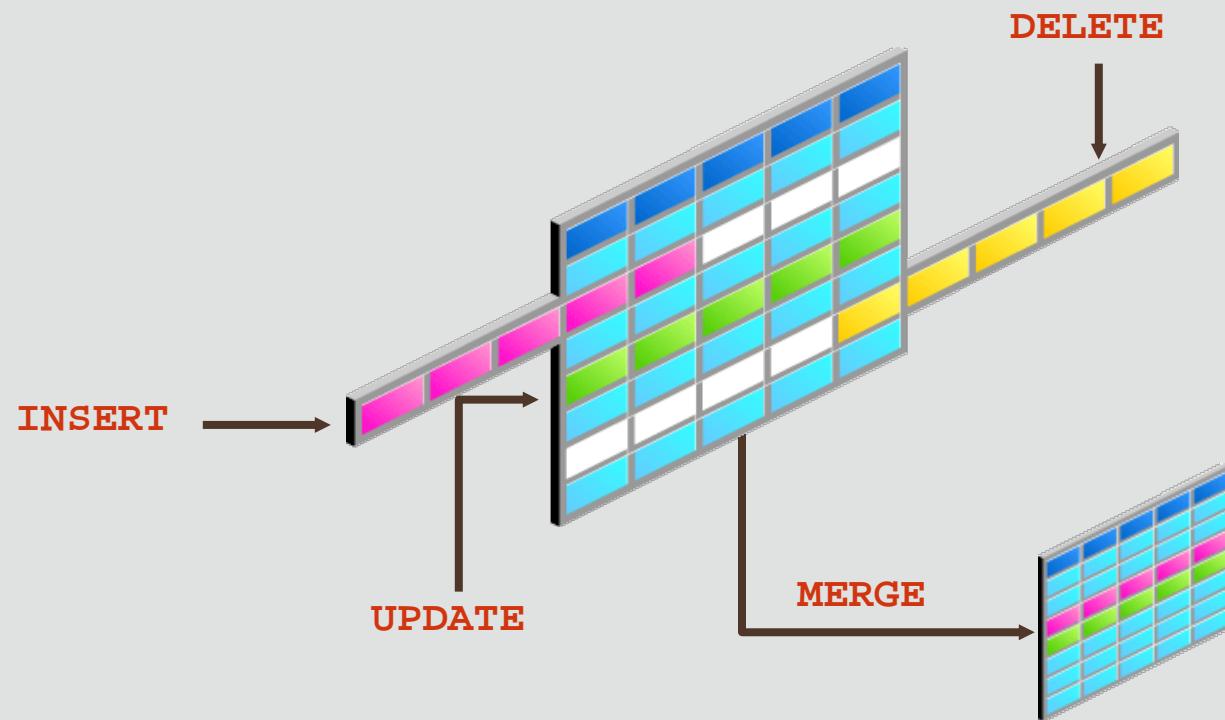
Create Copy of Original Table

- It is very important that you do NOT modify the existing tables (such as EMPLOYEES and DEPARTMENTS), because they will be needed later in the course
- The examples in this lesson use the COPY_EMP table
- If you haven't already created the COPY_EMP table, do so now by executing this SQL statement:

```
CREATE TABLE copy_emp
  AS SELECT *
  FROM employees;
```

Manipulating Data Using PL/SQL

- Make changes to data by using DML commands within your PLSQL block:
 - INSERT
 - UPDATE
 - DELETE
 - MERGE



Manipulating Data Using PL/SQL

- 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
 - 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

Manipulating Data Using PL/SQL

- 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



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

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 := 10;
BEGIN
    UPDATE copy_emp ←
        SET salary = salary + v_sal_increse
        WHERE job_id = 'ST_CLERK';
END;
```

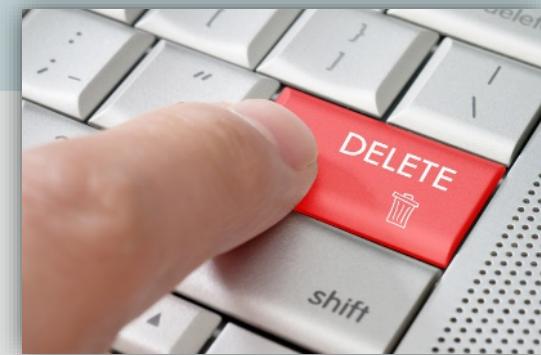
It is recommended that the UPDATE statement be on a line of its own.



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;
```



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;
```

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

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



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



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



Cursor Attributes for Implicit Cursors

Attribute	Description
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.

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;
```

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;
```

Using Implicit Cursor Attributes: Good Practice Guideline

- Look at this code which creates a table and then executes a PL/SQL block
- Determine 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;
```

Using Implicit Cursor Attributes: Good Practice Guideline

- To INSERT the value in the RESULTS table - we must save the SQL%ROWCOUNT value in a declared variable
- The value is displayed using PUT_LINE

```
DECLARE
  v_rowcount      INTEGER;
BEGIN
  UPDATE copy_emp
    SET salary = salary + 100
   WHERE job_id = 'ST_CLERK';
  DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT || ' rows in COPY_EMPupdated.');
  v_rowcount := SQL%ROWCOUNT;
  INSERT INTO results (num_rows)
    VALUES (v_rowcount);
  DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT || ' rows in RESULTS updated.');
END;
```

Terminology

- Key terms used in this lesson included:
 - INSERT
 - UPDATE
 - DELETE
 - MERGE
 - Explicit cursors
 - Implicit cursors

Summary

- In this lesson, you should have learned how 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

ORACLE

Academy