WEEK 7

Your document in a PowerPoint format is almost the same

First class after the break

PL/SQL

Source:

Lecture 07

<https://docs.oracle.com/cd/B28359_01/appdev.111/b28843/tdddg_procedures.htm>

[https://docs.oracle.com/database/121/LNPLS/controlstatements.htm#LNPLS411](https://docs.oracle.com/database/121/LNPLS/controlstatements.htm)

Go to Les07-PL/SQL-extra notes

(Slide 2)

Stored Procedures

Functions

PL/SQL

Side notes – not for students

CREATE OR REPLACE PROCEDURE nums (pInput IN VARCHAR2)

AS

x\_num VARCHAR(20);

BEGIN

x\_num := pInput ;

dbms\_output.put\_line('You entered: ' || x\_num);

END;

/

begin

nums(pinput => &input); -- put entry in quotes ‘F’

end;

Agenda(2) - what we cover today will be

* PL/SQL Overview
* Creating Standalone Procedures and Functions
* Variable and Constraints
* General Comparison Functions

# Overview notes

(3)

You already know how to interact with the database using SQL, it is not enough.

Need more to build entire enterprise applications.

Remember SQL basically is to get information from tables, change it and insert it.

PL/SQL is a third-generation language that has the procedures etc like other languages, but integrates well with SQL

PL/SQL makes it possible to build complex and powerful applications.

- Because PL/SQL is executed in the database, you can include SQL statements in your code without having to establish a separate connection.

The main types of program units

The three are known as

**STORED PROCEDURES**

- standalone procedures

- functions, and

- packages.

Once stored in the database, -- can be used as building blocks for several different applications.

You can have standalone procedures, but it is recommended to place your code into a package. More about that later

Basic Procedure BLOCKS

Declarative (optional)

- Variables and constants are identified by keyword DECLARE.

Executable (mandatory)

- Contains the application logic.

These are **KEYWORDS**

Starts with **BEGIN**

Ends with **END**

Exception handling (optional)

- Starts with keyword EXCEPTION and

- handles error conditions that may occur in the executable part.

JUMP TO 4 on slide

CREATE PROCEDURES/FUNCTIONS

(6)

SYNTAX

**CREATE OR REPLACE PROCEDURE** schema.procedure\_name(arg1 datatype, ...) AS

**BEGIN**

....

**END** procedure\_name;

**CREATE OR REPLACE FUNCTION** schema.function\_name(arg1 datatype, ...) AS

**BEGIN**

....

**RETURN**

**END** function\_name;

# MORE PROCEDURES

**Arguments in Procedures**

(7)

A procedure/function may receive arguments.

Argument has the following elements:

* + Datatype
    1. Can be any datatype supported by PL/SQL.
  + IN / OUT / IN OUT
    1. IN indicates that the procedure must receive a value for the argument.
    2. OUT indicate that the procedure/function passes a value for the argument back to the calling program.
    3. IN OUT indicates that procedure must receive a value for the argument and passes a value back to the calling program.
  + Default
    1. Using DEFAULT keyword, you can define a value for an argument.

# Sample – simple PL/SQL Block

(8)

**PURPOSE: To output a simple line of text**

Try this – watch quotes might be a problem – have students try it

SET SERVEROUTPUT ON

BEGIN -- no ending like a semi colon

DBMS\_OUTPUT. PUT\_LINE (' Welcome to 2nd half of DBS311'); -- watch single quotes still

END; -- semi-colon

## ****What is dbms\_output.put\_line?****

The Oracle dbms\_output is a **package** (more later) that allows us to write data to direct our PL/SQL output to a screen. It has a procedure called put\_line that display the information in a line. The package is particularly useful for displaying debugging information.

## ****What is a package****

A package is a schema object that groups logically related PL/SQL types, variables, constants, subprograms, cursors, and exceptions. A package is compiled and stored in the database, where many applications can share its contents. A package always has a specification, which declares the public items that can be referenced from outside the package.

Have students type it in

BEGIN

DBMS\_OUTPUT. PUT\_LINE (' Welcome to 2nd half of DBS311'); -- watch single quotes still

END;

**OUTPUT:**

Welcome to 2nd half of DBS311

PL/SQL procedure successfully completed.

NOTE: This is often called an ANONYMOUS BLOCK because it was not named.

A block without a name is an anonymous block. An anonymous block is not saved in the Oracle Database server, so it is just for one-time use. However, PL/SQL anonymous blocks can be useful for testing purposes.

Sample Procedure to try

(9)

Run this then jump to next page to show meaning

# DECLARE Section

**To define variables and**

**constants**

DECLARE

value\_1 NUMBER := 20; -- declaring the variable and assigning a value

value\_2 NUMBER := 5;

addition NUMBER;

subtraction NUMBER; -- defining a variable with no initial value

multiplication NUMBER;

division NUMBER;

BEGIN

addition := value\_1 + value\_2;

subtraction := value\_1 - value\_2;

multiplication := value\_1 \* value\_2;

division := value\_1 / value\_2;

DBMS\_OUTPUT.PUT\_LINE ('addition: ' || addition);

DBMS\_OUTPUT.PUT\_LINE ('subtraction: ' || subtraction);

DBMS\_OUTPUT.PUT\_LINE ('multiplication: ' || multiplication);

DBMS\_OUTPUT.PUT\_LINE ('division: ' || division);

END;

Next screen demo with STRING

# Another using a string output

(no slide)

When using character literals in PL/SQL, remember:

1 Character literals are case-sensitive. For example, 'Z' and 'z' are different.

2 Whitespace characters are significant.

Significance example of whitespace

Show tis example …

BEGIN

DBMS\_OUTPUT.PUT\_LINE('This string breaks

here.');

END;

Notice the output is on 2 lines

Significance of whitespace

This string breaks

here.

How to fix it (assuming you did not want 2 lines)

??????????

# EXCEPTION (10)

# This section handles errors that occur when a PL/SQL block executes

Example:

DECLARE

value\_1 NUMBER := 20;

value\_2 NUMBER := 0;

division NUMBER;

BEGIN

division := value\_1 / value\_2; -- died 20 by zero generates an error

DBMS\_OUTPUT.PUT\_LINE ('division: ' || division);

END;

=======================================

RUN IT you get this error message

Error report -

ORA-06512: at line 8

01476. 00000 - "divisor is equal to zero"

FIXING NEXT 2 PAGES

# Here is a fix

(11)

**DECLARE**

value\_1 **NUMBER** := 20;

value\_2 **NUMBER** := 0;

division **NUMBER**;

**BEGIN**

division := value\_1 / value\_2;

DBMS\_OUTPUT.PUT\_LINE ('division: ' || division);

**EXCEPTION**

**WHEN OTHERS**

Not a specific fix

**THEN**

DBMS\_OUTPUT.PUT\_LINE ('Error!');

**END**;

OUTPUT… must look closely to see it

Error!

Fixing it more

DECLARE

value\_1 NUMBER := 20;

value\_2 NUMBER := 0;

division NUMBER;

BEGIN

division := value\_1 / value\_2;

DBMS\_OUTPUT.PUT\_LINE ('division: ' || division);

EXCEPTION

WHEN ZERO\_DIVIDE -- caught by this error handling

THEN

DBMS\_OUTPUT.PUT\_LINE ('Divider is zero!');

WHEN OTHERS -- WHEN OTHERS must be last

THEN

DBMS\_OUTPUT.PUT\_LINE ('Error!');

END;

Divider is zero!

PL/SQL procedure successfully completed.

.

# SELECT INTO -- one row retrieved from select

(14)

**Using a procedure, but getting the data from a table**

The following PL/SQL code searches for a specific product by its product ID and displays the product ID and the product name for that product

🡺 Using a SELECT from a table to load the defined variables.

Put up this code and explain

**DECLARE -- define variables**

**productId NUMBER := 2;**

**productName VARCHAR2(255 BYTE);**

**price NUMBER(9,2);**

**BEGIN**

**SELECT product\_name, List\_price** -- select data from these columns

**INTO productName, price** -- insert them into the above declared variables

**FROM products**

**WHERE product\_id = productID; -- will get 1 row or none**

-- now ouput the findings – assumed example worked

**DBMS\_OUTPUT.PUT\_LINE ('Product Name: ' || productName);**

**DBMS\_OUTPUT.PUT\_LINE ('Product Price: ' || price);**

**END;**

**===========================**

**OUTPUT:**

Product Name: Intel Xeon E5-2697 V4

Product Price: 2554.99

PL/SQL procedure successfully completed.

# SELECT INTO with more than one row retrieved

(14)

Creates an error.

Need to handle it

Change the problem

We change the condition to search for products with category 2.

Since, we have many products in this category, the SELECT INTO statement generates an error

START WITH

**DECLARE**

categoryID **NUMBER** := 2;

productName **VARCHAR2**(255 BYTE);

price **NUMBER**(9,2);

**BEGIN**

**SELECT** product\_name, List\_price **INTO** productName, price

**FROM** products

**WHERE** category\_id = categoryId;

DBMS\_OUTPUT.PUT\_LINE ('Product Name: ' || productName);

DBMS\_OUTPUT.PUT\_LINE ('Product Price: ' || price);

END;

OUTPUT:

ORA-01422**: exact fetch returns more than requested number of rows**

Fixing it(15)

**DECLARE**

categoryId **NUMBER** := 2;

productName **VARCHAR2**(255 BYTE);

price **NUMBER**(9,2);

**BEGIN**

**SELECT** product\_name, List\_price **INTO** productName, price

**FROM** products

**WHERE** category\_id = categoryId;

DBMS\_OUTPUT.PUT\_LINE ('Product Name: ' || productName);

DBMS\_OUTPUT.PUT\_LINE ('Product Price: ' || price);

**EXCEPTION**

**WHEN** TOO\_MANY\_ROWS

**THEN**

DBMS\_OUTPUT.PUT\_LINE ('Too Many Rows Returned!');

**END**;

Too Many Rows Returned!

# Change problem to NO DATA FOUND

And using an exception handler

**DECLARE**

productId **NUMBER** := 300;

productName **VARCHAR2**(255 BYTE);

price **NUMBER**(9,2);

**BEGIN**

**SELECT** product\_name, List\_price **INTO** productName, price

**FROM** products

**WHERE** product\_id = productId;

DBMS\_OUTPUT.PUT\_LINE ('Product Name: ' || productName);

DBMS\_OUTPUT.PUT\_LINE ('Product Price: ' || price);

**EXCEPTION**

**WHEN** NO\_DATA\_FOUND

**THEN**

DBMS\_OUTPUT.PUT\_LINE ('No Data Found!');

**END**;

No Data Found!

PL/SQL procedure successfully completed.

# Anonymous Blocks 🡪 your first procedure

(17,18)

If a code is used multiple times or by different applications, then you need to store the block in the database.

Storing it is known as a **stored procedure or stored function**

**#1 create a table called NEW\_EMPLOYEE from employees table**

**Create table NEW\_EMPLOYEES AS  
(select \* from employees);**

**SIMPLE SAMPLE**

**CREATE OR REPLACE PROCEDURE remove\_employee AS** -- gave procedure a name

**employeeId NUMBER;**

**BEGIN**

**employeeId := 1;**

**DELETE FROM new\_employees**

**WHERE employee\_id = employeeId;**

**EXCEPTION**

**WHEN OTHERS**

**THEN**

**DBMS\_OUTPUT.PUT\_LINE ('Error!');**

**END;**

Look for employee 1

select \* from new\_employees where employee\_id = 1;

Run the procedure to remove the employee

BEGIN

remove\_employee();

END;

Test it again …

select \* from new\_employees where employee\_id = 1;

# CONTROL STATEMENTS (19)

Normal output of a procedure to update, delete etc 🡺 indicates it was successful.

🡺 But we want to know the number of rows.

Leads to CONDITIONAL STATEMENTS

Here are 3 types

1 - conditional selection statements

Same logic is in programming languages

2 - Loop statements

3 - Sequential Control statements

Will discuss what it means

AND

Look at examples

# IF THEN Statement

(20)

CREATE the procedure with the new info

Recreate table first

DROP table new\_employees;

**Create table NEW\_EMPLOYEES AS  
(select \* from employees);**

Check that 2 exists

**CREATE OR REPLACE PROCEDURE** remove\_employee AS

employeeId **NUMBER**;

**BEGIN**

employeeId := 2; -- using employee 2

**DELETE FROM** new\_employees

**WHERE** employee\_id = employeeId;

**IF** SQL%ROWCOUNT = 0

**THEN**

DBMS\_OUTPUT.PUT\_LINE ('Employee with ID ' || employeeId || ' does not exists');

**END** **IF**;

**EXCEPTION**

**WHEN OTHERS**

**THEN**

DBMS\_OUTPUT.PUT\_LINE ('Error!');

**END**;

Then run it

BEGIN

remove\_employee(); -- removes employee 2

END;

Run it again now that the employee is deleted

OUTPUT:🡺 Employee with ID 2 does not exists

# IF THEN ELSE

(22)

Put the employees back together

DROP table new\_employees;

**Create table NEW\_EMPLOYEES AS  
(select \* from employees);**

**Remove employee 2, but state what was deleted**

CREATE OR REPLACE PROCEDURE remove\_employee AS

employeeId NUMBER;

BEGIN

employeeId := 2;

DELETE FROM new\_employees

WHERE employee\_id = employeeId;

**IF SQL%ROWCOUNT = 0**

**THEN**

**DBMS\_OUTPUT.PUT\_LINE ('Employee with ID ' || employeeId || ' does not exists');**

**ELSE**

**DBMS\_OUTPUT.PUT\_LINE ('Employee with ID ' || employeeId || ' DELETED!');**

**END IF;**

EXCEPTION

WHEN OTHERS

THEN

DBMS\_OUTPUT.PUT\_LINE ('Error!');

END;

Procedure REMOVE\_EMPLOYEE compiled

Run it

BEGIN

remove\_employee();

END;

Employee with ID 2 DELETED!

# IF THEN --- ELSE

(22)

But testing the other IF or else

Do NOT ROLLBACK -- leave employee\_id 2 as deleted.

**CREATE OR REPLACE PROCEDURE** remove\_employee AS

employeeId **NUMBER**;

**BEGIN**

employeeId := 2;

**DELETE FROM** new\_employees

**WHERE** employee\_id = employeeId;

**IF SQL%ROWCOUNT = 0 THEN**

**DBMS\_OUTPUT.PUT\_LINE ('Employee with ID ' || employeeId || ' does not exists');**

**ELSE**

**DBMS\_OUTPUT.PUT\_LINE ('Employee with ID ' || employeeId || ' DELETED!');**

**END IF;**

**EXCEPTION**

**WHEN OTHERS**

**THEN**

DBMS\_OUTPUT.PUT\_LINE ('Error!');

**END**;

Now run it again

BEGIN

remove\_employee();

END;

Employee with ID 2 does not exists

# IF THEN ELSIF

(23)

Run this code notice manager id of 2 … there are several -- 5 of them

**CREATE** **OR REPLACE PROCEDURE** remove\_employee AS

managerId **NUMBER**;

**BEGIN**

managerId := 2;

**DELETE** FROM new\_employees

**WHERE** manager\_id = managerId;

**IF** SQL%ROWCOUNT = 0

THEN

DBMS\_OUTPUT.PUT\_LINE ('No employee is deleted');

**ELSIF**

SQL%ROWCOUNT = 1

THEN

DBMS\_OUTPUT.PUT\_LINE ('One employee is deleted.');

**ELSE**

DBMS\_OUTPUT.PUT\_LINE ('More than one employee is deleted!');

**END IF**;

**EXCEPTION**

**WHEN OTHERS**

**THEN**

DBMS\_OUTPUT.PUT\_LINE ('Error!');

**END**;

Run it

OUTPUT:

More than one employee is deleted!

# NESTING – IF THEN ELSE

(24)

Sample syntax

**IF** condition **THEN**

Again, the logic is like any other language

**IF** condition **THEN**

statements

**ELSE** condition

statements

**END IF**;

**ELSIF**

**IF condition THEN**

statements

**END IF**;

**ELSE**

statements

**END IF;**

# CASE

# (25)

**CASE** selector

**WHEN** value\_1 **THEN** statements

**WHEN** value\_2 **THEN** statements

...

**WHEN** value\_n **THEN** statements

**ELSE**

statements ]

**END CASE**;

As soon as a value matches the statement is executed

If no match occurs, then the ELSE executes …. If there is an ELSE

Example next page

# Example of CASE

**26**

**DECLARE**

semester **CHAR**(1);

**BEGIN**

semester := 'S';

**CASE** semester

**WHEN** 'F' **THEN** DBMS\_OUTPUT.PUT\_LINE('Fall Term');

**WHEN** 'W' **THEN** DBMS\_OUTPUT.PUT\_LINE('Winter Term');

**WHEN** 'S' **THEN** DBMS\_OUTPUT.PUT\_LINE('Summer Term');

**ELSE** DBMS\_OUTPUT.PUT\_LINE('Wrong Value');

**END CASE**;

**END**;

**OUTPUT:**

Summer Term 🡸

PL/SQL procedure successfully completed.

# Same using IF ELSEIF

# (27)

DECLARE

semester CHAR(1);

BEGIN

semester := 'f';

CASE semester

WHEN 'F' THEN DBMS\_OUTPUT.PUT\_LINE('Fall Term');

WHEN 'W' THEN DBMS\_OUTPUT.PUT\_LINE('Winter Term');

WHEN 'S' THEN DBMS\_OUTPUT.PUT\_LINE('Summer Term');

ELSE DBMS\_OUTPUT.PUT\_LINE('Wrong Value');

END CASE;

END;

The END