Les07 PL/SQL part 1 intro

Week changes depending on semester length
PL/SQL part 1

Document in a PowerPoint format is almost the same

Source:
Lecture 07
https://docs.oracle.com/cd/B28359_01/appdev.111/b28843/tddd g_procedures.htm
https://docs.oracle.com/database/121/LNPLS/controlstatements .htm#LNPLS411

Go to Les07-PL/SQL-extra notes has how to do it with screen shots for the SQL Developer GUI interface
(Slide 2)
Stored Procedures
Functions
PL/SQL

Agenda(2) - what we cover today will be

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

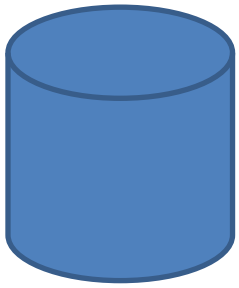
Overview notes

(3)

Programming Language

Like C or Java

drivers



PL/SQL

Overview

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.

PROBLEM:

Suppose you want to
1 Give some employees based on job title a 20% raise
2 Other employees a 10% raise
3 Other will be fired but store the data for 7 years

If you wanted to do this say every year then it would be a good idea to have aa saved program set up. No need tto call you every time.

NEEDED

```
IF
      UPDATE
ELSE
      UPDATE
ENDIF
```

Could use a CASE statement

INSERT into Fired employees table

COMMIT

Rather than executing a series of statements individually, PL/SQL allows for combining these into a block of code.

Aside: You can appreciate that a company cannot run if it depends on waiting for a programmer to execute every SQL statement in a normal business operation.

ADVANTAGE of PL/SQL:

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
1 - standalone procedures
2 - functions, and
3 - packages.

The three are known as
STORED PROCEDURES

Once stored in the database, -- they can be used again 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.
  - Starts with **BEGIN**
  - Ends with **END;**

These are **KEYWORDS**

Exception handling (optional)

- Starts with keyword EXCEPTION and
  - handles error conditions that may occur in the executable part.

/ ← do not forget this on the first blank line to execute the procedure ←

JUMP TO 4 on slide

CREATE PROCEDURES/FUNCTIONS

(6)

SYNTAX or general format

```
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

Can be any datatype supported by PL/SQL.

- IN / OUT / IN OUT

IN indicates that the procedure must receive a value for the argument.

OUT indicate that the procedure/function passes a value for the argument back to the calling program.

IN OUT indicates that procedure must receive a value for the argument and passes a value back to the calling program.

- Default

Using DEFAULT keyword, you can define a value for an argument.

Example1: Creating Procedure

Calling it using EXEC

In this example, we are going to create an Oracle procedure that takes the name as input and prints the welcome message as output. We are going to use EXEC command to call procedure.

```
CREATE OR REPLACE PROCEDURE welcome_msg (p_name IN VARCHAR2)
IS
BEGIN
dbms_output.put_line ('Welcome '|| p_name);
END;
/

EXEC welcome_msg ('ron');
```

You will not see anything happening.
Explanation coming

What does it all mean

Code Explanation:

-- Creating the procedure with name ➔ 'welcome_msg' and
➔with one parameter 'p_name' of 'IN' type.

-- Printing the welcome message by concatenating the input name to the word Welcome.

-- Procedure is compiled successfully.

-- Calling the procedure using EXEC command with the parameter 'ron'.

Procedure is executed, and the message is printed out as "Welcome ron".

You will not see anything happening yet
The reason is coming next page

Another 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 ← ← you are going to forget t

BEGIN -- no ending like a semi colon

DBMS_OUTPUT.PUT_LINE (' Welcome to PL/SQL section of DBS311'); -- watch quotes

END;

What is dbms_output.put_line?

The Oracle dbms_output is a **package** (more later) that allows us to write data and to direct our PL/SQL output to a screen. It has a procedure called put_line that displays 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.

TRY IT

```
BEGIN
        DBMS_OUTPUT.PUT_LINE (' Welcome to 2nd half of DBS311 --------------');

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

Copy this piece

-- 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;
```

Performing actions on the variables declared above

Display results

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 this example … what happens with strings

```
BEGIN
  DBMS_OUTPUT.PUT_LINE('This string breaks
here.');
END;
```

STOP HERE ----------

This string breaks
here.

Notice the output is on 2 lines
Significance of whitespace

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

Look at the calculation and ask yourself what will happen.

Example:

```
DECLARE
  value_1     NUMBER := 20;
  value_2     NUMBER := 0;
  division    NUMBER;

BEGIN

  division := value_1 / value_2;          -- divide 20 by zero generates an error

  DBMS_OUTPUT.PUT_LINE ('division: ' || division);
END;
```

Look at the calculation and ask yourself what will happen.

WHAT HAPPENS ???

========= stop here ==========

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)

Run this code and see the results

```
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
  THEN
     DBMS_OUTPUT.PUT_LINE ('Error  ------- has occurred!');

END;
```

Not a very specific fix.

NOTE: The error message before is replaced with your own message

OUTPUT… must look closely to see it. I say this because some cannot see the output easily.

Error ------- has occurred!

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;
```

Exception handling is very important in coding. These are just samples

ASIDE: Use indents. Many assignments, labs and tests not indented

ZERO_DIVIDE is a prewritten code that captures a specific error. When it does you can put your own error that makes sense to your application

===== stop here ==========

OUTPUT:

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

What is the requested activity?

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

This is 2020 summer code – next page for 2023-1 and later

```
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 output 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.

This is not the same data as in your tables for 2021-3 but you can run it to see what happens

```
DECLARE  -- define variables

  Product_id        NUMBER := 2;
  ProductName       VARCHAR2(255 BYTE);
  Price             NUMBER(9,2);

BEGIN
  SELECT prod_name, prod_sell              -- select data from these columns
                INTO productName, price    -- insert them into the above declared variables
  FROM products
  WHERE product_id = prod_no;              -- will get 1 row or none

                                   -- now output the findings – assumed example worked
  DBMS_OUTPUT.PUT_LINE ('Product Name: ' || productName);
  DBMS_OUTPUT.PUT_LINE ('Product Price: ' || price);
END;
```

STOP HERE ==========

What is the result????

```
Error report -
ORA-01403: no data found
ORA-06512: at line 8
01403. 00000 -  "no data found"
*Cause:    No data was found from the objects.
*Action:   There was no data from the objects which may be due to end of fetch.
```

Why?

FIX IT – it isn't wrong, but it does not show what we wanted to demonstrate

Fixing it to demo it working 2023-3 data

```
DECLARE  -- define variables

  Product_id        NUMBER := 40100;              -- specific value
  ProductName       VARCHAR2(255 BYTE);
  Price             NUMBER(9,2);

BEGIN
  SELECT prod_name, Prod_sell              -- select data from these columns
              INTO productName, price      -- insert them into the above declared variables
  FROM products
  WHERE product_id = prod_no;              -- will get 1 row or none

                                   -- now output the findings – assumed example worked
  DBMS_OUTPUT.PUT_LINE ('Product Name: ' || productName);
  DBMS_OUTPUT.PUT_LINE ('Product Price: ' || price);
END;
```

========== stop here ==========

Product Name: Star Lite
Product Price: 165

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 product type 'Tents"
Since, we have many products in this type, the SELECT INTO statement generates an error

START WITH (using 2023 data)

```
DECLARE
        ProductType         VARCHAR2(20):= 'Tents';
        productName         VARCHAR2(255 BYTE);
        price               NUMBER(9,2);
BEGIN
        SELECT      prod_name, prod_sell INTO productName, price
        FROM        products
        WHERE       Prod_type = ProductType;    -- meaning look for tents

        DBMS_OUTPUT.PUT_LINE ('Product Name: ' || productName);
        DBMS_OUTPUT.PUT_LINE ('Product Price: ' || price);
END;
```

STOP HERE ==========

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

Fixing it (15)

```
DECLARE
        ProductType         VARCHAR2(20):= 'Tents';
        productName         VARCHAR2(255 BYTE);
        price               NUMBER(9,2);
BEGIN
        SELECT      prod_name, prod_sell INTO productName, price
        FROM        products
        WHERE       Prod_type = ProductType;

        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;
```

OUTPUT:

Too Many Rows Returned!

What exceptions have we learned so far

TOO_MANY_ROWS

ZERO_DIVIDE

OTHERS ➔-- WHEN OTHERS must be last

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 prod_name, prod_sell INTO productName, price
  FROM products
  WHERE prod_no = 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.

What exceptions have we learned so far
TOO_MANY_ROWS
NO_DATA_FOUND
ZERO_DIVIDE
OTHERS ➔-- WHEN OTHERS must be last

Pre-defined Exceptions

PL/SQL provides many pre-defined exceptions, which are executed when any database rule is violated by a program. For example, the predefined exception NO_DATA_FOUND is raised when a SELECT INTO statement returns no rows. The following table lists few of the important pre-defined exceptions –

You will need more than the 3 for working environment.

Exception	Oracle Error	SQLCODE	Description
ACCESS_INTO_NULL	06530	-6530	It is raised when a null object is automatically assigned a value.
CASE_NOT_FOUND	06592	-6592	It is raised when none of the choices in the WHEN clause of a CASE statement is selected, and there is no ELSE clause.
COLLECTION_IS_NULL	06531	-6531	It is raised when a program attempts to apply collection methods other than EXISTS to an uninitialized nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray.
DUP_VAL_ON_INDEX	00001	-1	It is raised when duplicate values are attempted to be stored in a column with unique index.
INVALID_CURSOR	01001	-1001	It is raised when attempts are made to make a cursor operation that is not allowed, such as closing an unopened cursor.
INVALID_NUMBER	01722	-1722	It is raised when the conversion of a character string into a number fails because the string does not represent a valid number.
LOGIN_DENIED	01017	-1017	It is raised when a program attempts to log on to the database with an invalid username or password.
NO_DATA_FOUND	01403	+100	It is raised when a SELECT INTO statement returns no rows.

NOT_LOGGED_ON	01012	-1012	It is raised when a database call is issued without being connected to the database.
PROGRAM_ERROR	06501	-6501	It is raised when PL/SQL has an internal problem.
ROWTYPE_MISMATCH	06504	-6504	It is raised when a cursor fetches value in a variable having incompatible data type.
SELF_IS_NULL	30625	-30625	It is raised when a member method is invoked, but the instance of the object type was not initialized.
STORAGE_ERROR	06500	-6500	It is raised when PL/SQL ran out of memory or memory was corrupted.
TOO_MANY_ROWS	01422	-1422	It is raised when a SELECT INTO statement returns more than one row.
VALUE_ERROR	06502	-6502	It is raised when an arithmetic, conversion, truncation, or size constraint error occurs.
ZERO_DIVIDE	01476	1476	It is raised when an attempt is made to divide a number by zero.

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);
```

Check data loaded

SIMPLE SAMPLE of a stored procedure

Purpose: Will remove employee number 1

```
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;
```

ONLY Stored the procedure

OUTPUT: Procedure REMOVE_EMPLOYEE compiled

Check on left panel for procedure

Test it next page

Look for employee 1 to see what would get

```
SELECT    *
FROM      new_employees
WHERE     employee_id = 1;
```

NOTICE
It is not gone and why

Bjorn is there

Run the procedure to remove the employee

```
BEGIN
  remove_employee();    -- note the bracket
END;
```

Test it again …

```
SELECT    *
FROM      new_employees
WHERE     employee_id = 1;
```

RESULT:
no rows selected

CONTROL STATEMENTS (19)

Normal output of a procedure to update, delete etc

➔ indicates it was successful.
➔ But we want to know more like the number of rows.

Leads to CONDITIONAL STATEMENTS

Here are 3 types

1 - Conditional selection statements

2 - Loop statements (later)

3 - Sequential Control statements

Same logic is found in other programming languages

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 1 exists Bjorn
select *
from employees
where employee_id = 1;

```
CREATE OR REPLACE PROCEDURE remove_employee AS
  employeeId          NUMBER;

BEGIN
        employeeId := 2;                          -- using employee 2 – check and see if there is one

        DELETE FROM new_employees
                WHERE employee_id = employeeId;

         IF SQL%ROWCOUNT = 0
                THEN
                        DBMS_OUTPUT.PUT_LINE ('Employee with ID ' || employeeId || ' does not exist');
        END IF;

EXCEPTION
        WHEN OTHERS
                THEN
                DBMS_OUTPUT.PUT_LINE ('Error!');
END;
```

Then run it

```
BEGIN
  remove_employee();   -- supposed to remove employee 2
END;
```

OUTPUT:➔ Employee with ID 2 does not exist

IF THEN ELSE

(22)

Put the employees back together

```
DROP table new_employees;

Create table NEW_EMPLOYEES  AS
(select * from employees);

select * from new_employees
order by employee_id;                    -- note there is no employee 2
```

Create a procedure to remove employee 2
If it was removed, state employee 2 is deleted

Improving the code

```
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 exist');
        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 does not exist

IF THEN ELSIF

(23)

Run this code notice manager id of 124 … there are several of them

```
CREATE OR REPLACE PROCEDURE remove_employee AS
  managerId NUMBER;

BEGIN
  managerId := 124;
  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

```
BEGIN
   remove_employee();
END;
```

Common error. Forgetting you are calling a procedure and it needs the () and ; On the end

OUTPUT:

More than one employee is deleted!

Run it again

NESTING – IF THEN ELSE

(24)

Sample syntax

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

Again, the logic is like any other language

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

Aside; Just a reminder
This is an anonymous block and does not get saved. It will execute when you run it.
Named procedures first need to be compiled and stored. Then you must run the procedure.

```
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)

Look on slide 27

EXAMPLE of INPUT from user

```
/* begin comments with this will save the – every line
        Write a store procedure that gets an integer number and prints
        the number is even........
*/ end with this

set serveroutput ON;   -- DO NOT forget to do this at start of session



CREATE OR REPLACE PROCEDURE  evenodd  (instuff IN number)
as

BEGIN

if mod(instuff, 2) = 0
        then dbms_output.put_line('The number is even!');
else
        dbms_output.put_line('The number is odd!');
end if;

END evenodd;









--execution statement taking an input from user and passing it to the procedure
BEGIN
        evenodd(&input);    -- asks for input from user
END;
```

The END

Of course, the best way you can learn it is to type it in and not cut and paste it. Then you will get lots of errors to correct