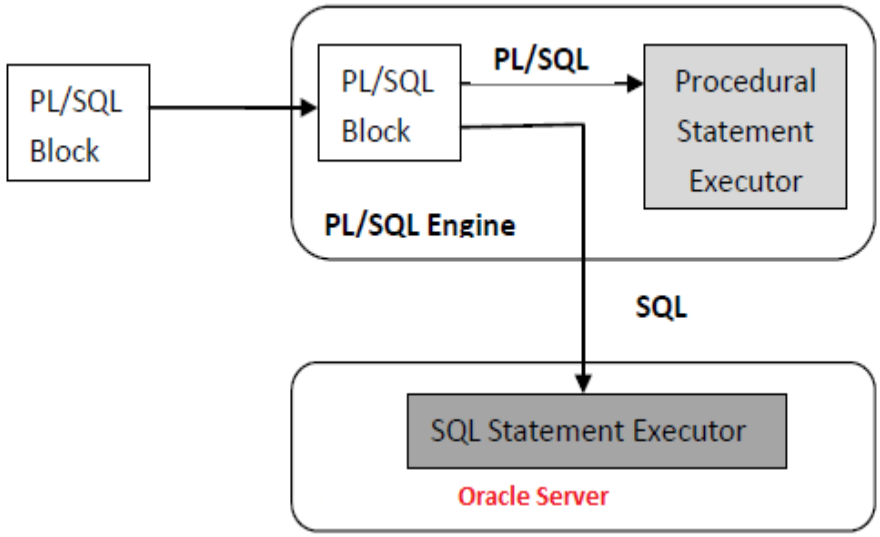
What is PL/SQL

- PL/SQL stands for Procedural Language extension of SQL.
- PL/SQL is a combination of SQL along with the procedural features of programming languages. It was developed by Oracle Corporation in the early 90's to enhance the capabilities of SQL.
- It offers features such as data encapsulation, exception handling, information hiding, object orientation and brings state-of-art programming to the Oracle Server and toolset

The PL/SQL Environment



When you submit PL/SQL blocks from Pro*C or Pro*Cobol , iSQL*Plus, the PL/SQL engine in the Oracle Server processes them. It separates the SQL statements and sends them individually to the SQL statements executor.

Many oracle tools , including Oracle Developer, have their own PL/SQL engine, which is independent of the engine in the Oracle Server.

- Blocks of PL/SQL are passed to and processed by PL/SQL engine, which may reside within the tool or within Oracle server.
- The engine that is used depends on where the PL/SQL block is being invoked from.
- A PL/SQL code block can be stored in the client system (client-side) or in the database (server-side).

Advantages of PL/SQL

These are the advantages of PL/SQL.

- **Block Structures**: PL SQL consists of blocks of code, which can be nested within each other. Each block forms a unit of a task or a logical module. PL/SQL Blocks can be stored in the database and reused.
- **Procedural Language Capability**: PL SQL consists of procedural language constructs such as conditional statements (if else statements) and loops like (FOR loops).
- **Better Performance**: PL/SQL can be used to group SQL statements together with a single block and

to send the entire block to the server in a single call, thereby reducing networking traffic. Without PL/SQL, the SQL statements are sent to the Oracle server one at a time. Each SQL statement results in another call to the Oracle Server and higher performance overhead.

- **Error Handling**: PL/SQL handles errors or exceptions effectively during the execution of a PL/SQL program. Once an exception is caught, specific actions can be taken depending upon the type of the exception or it can be displayed to the user with a message.
- **Portability** : As PL/SQL is native to the Oracle server, the PL/SQL program can run anywhere the Oracle server can run; you do not need to tailor them to each new environment.

A Simple PL/SQL Block:

PL/SQL is a block structured language, meaning that programs can be divided into logical blocks contains SQL and PL/SQL statements.

A PL/SQL Block consists of three sections:
 	
• The Declaration section (optional).
• The Execution section (mandatory).
• The Exception (or Error) Handling section (optional).
	DECLARE
	  Variable declaration
	BEGIN
	  Program Execution
	EXCEPTION
	  Exception handling
	END;

- Every statement in these sections must end with a semicolon.
- PL/SQL blocks can be nested within other PL/SQL blocks.
- Comments can be used to document code.

Declaration Section:

- Starts with keyword DECLARE and this section is optional
- It is used to declare any placeholders like variables, constants, records and cursors, which are used to manipulate data in the execution section. Placeholders may be any of Variables, Constants and Records, which stores data temporarily.

Execution Section:

- Starts with the reserved keyword BEGIN and ends with END and it is a mandatory section
- The program logic is written to perform any task. The programmatic constructs like loops, conditional statement and SQL statements are the part of execution section.

Exception Section:

- Starts with the reserved keyword EXCEPTION. This section is optional.
- Any errors in the program can be handled in this section, so that the PL/SQL Blocks terminates gracefully. If the PL/SQL Block contains exceptions that cannot be handled, the Block terminates abruptly with errors.

DBMS_OUTPUT.PUT_LINE

This Oracle supplied package provides some function to display the result from the PL/SQL block.

```
SET SERVEROUTPUT ON
DEFINE annual_sal = 60000
DECLARE
   V_sal   NUBMER(9,2) := &annual_sal;
BEGIN
   V_sal := v_sal /12;
   DBMS_OUTPUT.PUT_LINE ( 'The monthly salary is ' || To_Char(v_sal));
END;
```

Must be enabled in iSQL*Plus with SET SERVEROUTPUT ON

Block Types

Anonymous

```
[DECLARE]

BEGIN
..... Statements

[EXCEPTION]

END;
```

Procedure

```
PROCEDURE name IS

BEGIN
..... Statements

[EXCEPTION]

END;
```

Function

```
FUNCTION name
RETURN datatype IS

BEGIN
..... Statements
   RETURN value;
[EXCEPTION]
```

Anonymous Block (unnamed blocks) : They are declared at the point in an application where they are to be executed and are passed to the PL/SQL engine for execution at run time.

Subprograms : They are named PL/SQL blocks that can accept parameters and can be invoked. You can declare them either as procedures or as functions.

Program Constructs

Program Construct	Description	Availability
Anonymous Blocks	Unnamed PL/SQL blocks that are embedded within an application or are issued interactively	ALL PL/SQL environment
Application Procedures or Function	Name PL/SQL blocks stored in an Oracle Forms Developer application or shared library; can accept parameters and can be invoked repeatedly by name.	Oracle Developer tools components for example, Oracle Forms Developer, Oracle Reports

Stored Procedures or Functions	Name PL/SQL blocks stored in an Oracle server; can accept parameters and can be invoked repeatedly by name.	Oracle Server
Packages (Application or Stored)	Name PL/SQL modules that group related procedures, functions and identifiers	Oracle Server and Oracle Developer tools components for example, Oracle Forms Developer
Application	PL/SQL blocks that are associated with an application	Oracle Developer
Triggers	Event and fired automatically.	tools components for example, Oracle Forms Developer
Database triggers	PL/SQL blocks that are associated with a database table and fired automatically when triggered by DML statements.	Oracle Server
Object Types	User defined composite data types that encapsulates a data structure along with the functions and procedures needed to manipulate the data	Oracle Server and Oracle Developer tools

PL/SQL Placeholders

- Placeholders are temporary storage area and can be any of Variables, Constants and Records.
- Oracle defines placeholders to store data temporarily, which are used to manipulate data during the execution of a PL SQL block.
- Placeholders are defined with a name and a datatype. Few of the datatypes used to define placeholders are as given below. Number (n,m) , Char (n) , Varchar2 (n) , Date , Long , Long raw, Raw, Blob, Clob, Nclob, Bfile

PL/SQL Variables

- The General Syntax to declare a variable is: **variable_name datatype [NOT NULL := value];**
  variable_name is the name of the variable.
  datatype is a valid PL/SQL datatype.
  NOT NULL is an optional specification on the variable.
  value or DEFAULT value is also an optional specification, where you can initialize a variable.

- Each variable declaration is a separate statement and must be terminated by a semicolon.

For example, if you want to store the current salary of an employee, you can use a variable.

```
DECLARE
   salary number (6); -- "salary" is a variable of datatype number and of length 6.

    dept varchar2(10) NOT NULL := "HR Dept";

    v_mgr NUMBER(6) DEFAULT 100;
```

When a variable is specified as NOT NULL, you must initialize the variable when it is declared.

Default keyword instead of assignment operation is used to initialize the variables.

The value of a variable can change in the execution or exception section of the PL/SQL Block. We can assign values to variables in the two ways given below.

Assign values to variables : **variable_name:= value;**

Assign values to variables directly from the database columns by using a SELECT.. INTO statement.

SELECT column_name INTO variable_name FROM table_name [WHERE condition];

Note : = is the equality comparison operator, both in PL/SQL and SQL. := is the PL/SQL value assignment operator. These are analogous to == and = in C-derived languages.

Example: The below program will get the salary of an employee with id '1116' and display it on the screen.

```
DECLARE
    var_salary number(6);
    var_emp_id number(6) := 1116;
BEGIN
    SELECT salary INTO var_salary  FROM employee  WHERE emp_id = var_emp_id;
    dbms_output.put_line(var_salary);
    dbms_output.put_line('The employee ' || var_emp_id || ' has salary ' || var_salary);
END;
/
```

NOTE: The backward slash '/' indicates to execute the above PL/SQL Block.

Examples :

```
DECLARE
   v_job varchar2(9);
   v_total_sale NUMBER(9,2) := 0;
   c_tax_rate CONSTANT NUMBER(3,2) := 8.25;
   v_count BINARY_INTEGER := 0;
   v_orderdate DATE := SYSDATE + 7 ;
   v_valid BOOLEAN NOT NULL :=TRUE;
```

Scope of Variables

- PL/SQL allows the nesting of Blocks within Blocks i.e, the Execution section of an outer block can contain inner blocks.
- A variable which is accessible to an outer Block is also accessible to all nested inner Blocks.

- The variables declared in the inner blocks are not accessible to outer blocks.

Based on their declaration we can classify variables into two types.

Local variables - declared in an inner block and cannot be referenced by outside Blocks.
Global variables - declared in an outer block and can be referenced by itself and by its inner blocks.

Example :

Two variables declared in the outer block and values are assigned.

```
DECLARE
    var_num1 number;
    var_num2 number;
BEGIN
    var_num1 := 100;
    var_num2 := 200;
     DECLARE
          var_mult number;
     BEGIN
          var_mult := var_num1 * var_num2;
     END;
END;
/
```

The variable 'var_mult' is declared in the inner block, so cannot be accessed in the outer block . The variables 'var_num1' and 'var_num2' can be accessed anywhere in the block.

Qualify an identifier (variable name)

```
<<Outer>>
DECLARE
  Birthdate  DATE;
BEGIN
    DECLARE
       Birthdate  DATE;
    BEGIN
        ..............
       Outer.birthdate  := TO_DATE ( '03-AUG-1975', 'DD-MON-YYYY');
    END;
```

Block label prefix.

Qualify an identifier by using block label prefix.

PL/SQL Constants

- A constant is a value used in a PL/SQL Block that remains unchanged throughout the program.
- A constant is a user-defined literal value. You can declare a constant and use it instead of actual value.

For example: If you want to write a program which will increase the salary of the employees by 25%, you can declare a constant and use it throughout the program. Next time when you want to increase

the salary again you can change the value of the constant which will be easier than changing the actual value throughout the program.

The General Syntax to declare a constant is: **constant_name CONSTANT datatype := VALUE;**

For example, to declare salary_increase, you can write code as follows:

```
DECLARE
salary_increase CONSTANT number (3) := 10;
```

VALUE - It is a value which must be assigned to a constant when it is declared. You cannot assign a value later.

You must assign a value to a constant at the time you declare it. If you do not assign a value to a constant while declaring it and try to assign a value in the execution section, you will get an error.

Declaring variable with %Type attribute

When you declare PL/SQL variable to hold column values, you must ensure the variable is of the correct data type and precision. Rather than hard coding the data type and precision of the variable, you can use the %Type attribute to declare a variable according to another previously declared variable or database column.

Syntax :	**Example :**
Variable_name Table.Column_name%Type;	**V_name employees.last_name%Type;** **V_balance NUMBER(7,2);** **V_min_balance v_balance%Type :=10;**

Composite Data Types

- A scalar type has no internal components.
- A composite type has internal components that can be manipulated individually.
- Composite data type (also known as collections) are of **TABLE**, **RECORD**, **NESTED TABLE**, and **VARRAY** types.
- RECORD data type is used to treat related but dissimilar data as a logical unit
- TABLE data type is used to refer and manipulate collections of data as a whole object

PL/SQL Record Type

What are records?

- Records are another type of datatypes which oracle allows to be defined as a placeholder.
- It is composite datatypes - a combination of different scalar datatypes like char, varchar etc.
- Each scalar data types in the record holds a value. A record can be visualized as a row of data. It can contain all the contents of a row.

Declaring a record:

First define a composite datatype; then declare a record for that type. The General Syntax to define a composite datatype is:

```
TYPE record_type_name IS RECORD (first_col_name column_datatype ,
                                 second_col_name column_datatype, ...);
```

1. *record_type_name* – it is the name of the composite type you want to define.
2. *first_col_name, second_col_name, etc.,- it is the* names the fields/columns within the record.
3. *column_datatype* defines the scalar datatype of the fields.

There are different ways you can declare the datatype of the fields.

1. You can declare the field in the same way as you declares the field while creating the table.
2. If a field is based on a column from database table, you can define the field_type as follows:

col_name table_name.column_name%type;

The following code shows how to declare a record called **employee_rec** based on a user-defined type. DECLARE TYPE **employee_type** IS RECORD (employee_id number(5), employee_first_name varchar2(25), employee_last_name, employee.last_name%type, employee_dept employee.dept%type) ; **employee_rec employee_type**;	If all the fields of a record are based on the columns of a table, we can declare the record as follows: **record_name table_name%ROWTYPE;** For example, the above declaration of employee_rec can as follows: **DECLARE** employee_rec employee%ROWTYPE;

The advantages of declaring the record as a ROWTYPE are:

1. You do not need to explicitly declare variables for all the columns in a table.
2. If you alter the column specification in the database table, you do not need to update the code.

The disadvantage of declaring the record as a ROWTYPE is:
When you create a record as a ROWTYPE, fields will be created for all the columns in the table and memory will be used to create the datatype for all the fields. So use ROWTYPE only when you are using all the columns of the table in the program.

NOTE: When you are creating a record, you are just creating a datatype, similar to creating a variable. You need to assign values to the record to use them.

The following table consolidates the different ways in which you can define and declare a pl/sql record.

Syntax	**Usage**
TYPE record_type_name IS RECORD (column_name1 datatype, column_name2 datatype, ...);	Define a composite datatype, where each field is scalar.
col_name table_name.column_name%type;	Dynamically define the datatype of a column based on a database column.
record_name record_type_name;	Declare a record based on a user-defined

NOTE: You can use also *%type* to declare variables and constants. The General Syntax to declare a record of a user-defined datatype is: record_name record_type_name;

Passing Values To and From a Record

When you assign values to a record, you actually assign values to the fields within it.

- The General Syntax to assign a value to a column within a record directly is: record_name.col_name := value;
- The General Syntax to retrieve a value from a specific field into another variable is var_name := record_name.col_name;

If you used %ROWTYPE to declare a record, you can assign values as shown:	record_name.column_name := value;
We can assign values to records using SELECT Statements	SELECT col1, col2 INTO record_name.col_name1, record_name.col_name2 FROM table_name [WHERE clause];
If %ROWTYPE is used to declare a record then you can directly assign values to the whole record instead of each column separately. In this case, you must SELECT all the columns from the table into the record.	SELECT * INTO record_name FROM table_name [WHERE clause];

Bind Variables

- Variable declared in host environment is known as bind variable(variable in precompiled programs, screen fields in Oracle Forms application, and iSQL*Plus)
- Used to pass run-time values into or out of one or more PL/SQL programs. PL/SQL programs use bind variables like other PL/SQL variable.

Example of declaring variable in iSQL*Plus environment :

VARIABLE return_code NUMBER **VARIABLE** Result NUMBER

Both iSQL*Plus and SQL can refer the bind variable, and iSQL*Plus can display its value using PRINT command. **PRINT result** will print the variable result.

Using the host variable result in PL/SQL block :	BEGIN
	SELECT (Salary *12) + NVL(Commission,0) INTO : Result
	FROM Employees WHERE emp_id = 123;
	END;
	/
	Print Result

Store a variable value defined in PL/SQL into bind variable	**:Result** := v_sal /12; Prefix the bind variable with a colon **(:) distinguish it from other variables**

Compute the monthly salary based on annual salary.
 	
iSQL*PLUS command is for defining the host variables. DEFINE command specifies a user variable and assigns it a CHAR value.
 	
Even though you enter the number 50000, iSQL*Plus assign a CHAR value to monthly_sal.
	SET VERIFY OFF
	VARIABLE monthly_sal NUMBER
	DEFINE annula_sal = 50000
	DECLARE
	V_sal NUMBER(9,2) := &annual_sal;
	BEGIN
	:monthly_sal := v_sal/12;
	END;
	/
	Print monthly_sal
	Here we are defining a host variable, reference it in PL/SQL and then display its content in iSQL*Plus (using Print command)

Some Rules and Guidelines for writing PL/SQL

- Identifier (variable name) can contain up to 30 character, must start with alphabet.
- Identifier name cannot contain hyphens, slashes and spaces
- Name must not be the same as a database table column name, should not be reserved words.
- Character and date literal must be enclosed in single quotation marks (v_name := 'P Das');
- A slash (/) runs the PL/SQL block in a script file or in some tools such as iSQL*Plus.
- Prefix single line comments with two dashes **(--)**
- Place multiline comments between **/* and */**
- **Code Conventions: Following programming guidelines should be used to produce clear code and reduce maintenance cost when developing PL/SQL block**

Category	Case Convention	Example
SQL Statement	Uppercase	SELECT, INSERT
PL/SQL Keywords	Uppercase	DECLARE, BEGIN, IF …..
Datatypes	Uppercase	VARCHAR2, BOOLEAN
Identifiers and parameters	Lowercase	V_sal, p_empno
Database tables, and columns	Lowercase	Employees, employee_id

- Indent codes appropriately.
- Most of the functions available in SQL are also valid in PL/SQL expression (except DECODE and Group functions such as AVG, MIN, MAX etc.)
    - Example :
      V_mail_address := v_name || CHR(10) || v_adress || CHR(10);
      V_ename := LOWER(v_ename);
- Convert data to comparable data types using TO_CHAR, TO_DATE, TO_NUMBER
  o Example : v_date DATE := TO_DATE('12-JAN-2001', 'DD-MON-YYYY');

Conditional Statements in PL/SQL

As the name implies, PL/SQL supports programming language features like conditional statements, iterative statements. The programming constructs are similar to how you use in programming languages like Java and C++.

IF THEN ELSE STATEMENT
IF condition
THEN
  statement 1;
ELSE
  statement 2;
END IF;
	IF condition 1
	THEN
	  statement 1;
	  statement 2;
	ELSIF condtion2 THEN
	  statement 3;
	ELSE
	  statement 4;
	END IF
		IF condition 1
		THEN
		  statement 1;
		  statement 2;
		ELSIF condtion2 THEN
		  statement 3;
		ELSE
		  statement 4;
		END IF;

Iterative Statements in PL/SQL

There are three types of loops in PL/SQL: Simple Loop, While Loop, For Loop

Simple Loop :
The General Syntax to write a Simple Loop is:
LOOP
  statements;
  EXIT;
  {or EXIT WHEN condition;}
END LOOP;
	These are the important steps to be followed while using Simple Loop.
	• Initialize a variable before the loop body.
	• Increment the variable in the loop.
	• Use EXIT WHEN statement to exit from the Loop. If you use EXIT statement without WHEN condition, the statements in the loop is executed only once.

While Loop : The General Syntax to write a WHILE LOOP is: WHILE <condition> LOOP statements;	Important steps to follow when executing a while loop: • Initialize a variable before the loop body. • Increment the variable in the loop. • EXIT WHEN statement and EXIT statements can

END LOOP;	be used in while loops but it's not done often.
For Loop : The General Syntax FOR counter IN val1..val2 LOOP statements; END LOOP;	val1 - Start integer value. val2 - End integer value. .. – used for rang specification

Exercise :

v_id NUMBER	Legal
V_x, v_y, v_z VARCHAR2(30);	Illegal – only one identifier per declaration allowed
v_birthdate DATE NOT NULL;	Illegal – not null variable must be initialized
v_in_stock BOOLEAN := 1;	Illegal – 1 is not Boolean exp.
v_days_togo := v_duedate – SYSDATE;	Valid – resultant data type Number
v_sum := $100,000 + $25,000;	Illegal – cannot convert special symbol
v_flag := v_n2 > (2 * v_n3); where v_flag is boolean	Valid – Resultant type is boolean
v_value := NULL;	Valid – any scalar data type

Create and execute PL/SQL block that accepts two numbers through iSQL*Plus substitution variables, add them in PL/SQL print the result in screen.

```
SET ECHO OFF
SET VERIFY OFF
SET SERVEROUTPUT ON
DEFINE p_num1 = 2
DEFINE p_num2 = 4
DECLARE
   v_num1 NUMBER(9,2) := &p_num1;
   v_num2 NUMBER(9,2) := &p_num2;
   v_result NUMBER(9,2) ;
BEGIN
   v_result := v_num1 + v_num2;
   DBMS_OUTPUT.PUT_LINE (v_result);
END;
/
SET SERVEROUTPUT OFF
SET VERIFY ON
SET ECHO ON
```

Create a PL/SQL block that selects the maximum department number in the department table and store it in iSQL*Plus variable.

```
VARIABLE g_maxdept NUMBER
DECLARE
    v_maxdept NUMBER;
BEGIN
    SELECT max(dept_no)
    INTO v_maxdept
    FROM departments;
    :g_maxdept := v_maxdept;
END;
/
PRINT g_maxdept
```

Variable in DECLARE section is used inside BEGIN... END part. (executed on Oracle server). DEFINE is a way to substitute values (in SQL*Plus, SQL devevelop) - substitution done by the client side tool (sqlplus) before sending to server.

Alternately you can use following statement to print the PL/SQL variable directly.

```
dbms_output.put_line(v_maxdept);
```

The **SERVEROUTPUT** setting controls whether SQL*Plus prints the output generated by the DBMS_OUTPUT package from PL/SQL procedures.

```
SET ECHO OFF
SET VERIFY OFF
DEFINE p_dname = 'Education'
DECLARE
    v_naxsal employees.salary%Type ;

BEGIN
    SELECT max(salary) + 5000 INTO v_maxsal FROM employees;
    INSERT INTO employees (emp_id, salary, dept_name)  values ( 550, v_maxsal, '&p_dname');
    COMMIT;
END;
/
SET VERIFY ON
ET ECHO ON
```

The ECHO setting tells SQL*Plus whether you want the contents of script files to be echoed to the screen as they are executed.
The VERIFY setting controls whether or not SQL*Plus displays before and after images of each line that contains a substitution variable.

Find the maximum salary from employees table and add 5000 to it and insert a new record in employees table with employee id = 550 and departname = Education

---

```
SET ECHO OFF
SET VERIFY OFF
DEFINE p_salary = 20000
DEFINE p_empid = 550
BEGIN
   UPDATE employees SET salary = &p_salary WHERE emp_id = &p_empid;
   COMMIT;
END;
/
SET VERIFY ON
SET ECHO ON
```

Now update the salary of that employee by 20000

```
SET ECHO OFF
SET VERIFY OFF
DEFINE p_empid = 550
DECLARE
     v_result NUMBER(2) ;
BEGIN
     DELETE FROM employees WHERE emp_id = &p_empid;
     v_result := SQL%ROWCOUNT;
     dbms_output.put_line( TO_CHAR(v_result) || ' row(s) deleted');
COMMIT;
END;
/
SET VERIFY ON
SET ECHO ON
```

Now delete that employee and print the no of record affected

```
SET ECHO OFF
SET VERIFY OFF
SET SERVEROUTPUT ON
DEFINE p_empid = 550
DECLARE
    v_empid employees.emp_id%Type := &p_empid;
    v_salary employees.salary%Type;
    v_bonusper NUMBER(7,2);
    v_bonus NUMBER(7,2) ;
BEGIN
    SELECT salary INTO v_salary FROM employees WHERE emp_id = &p_empid;
    IF v_salary < 5000 THEN v_bonusper := .10;
       ELSEIF    v_salary BETWEEN 5000 and 10000 THEN v_bonusper := .15
       ELSEIF    v_salary > 10000 THEN v_bonusper := .20
       ELSE v_bonusper := 0;
    END IF;
    v_bonus := v_salary * v_bonusper;
    dbms_output.put_line( 'The bonus for the employee with id ' || TO_CHAR(v_empid)
                                  || ' is ' || v_bonus) ;
     COMMIT;
END;
/
SET VERIFY ON
SET ECHO ON
```

Calculate and print the bonus of an employee as per business rule

---

```
SET ECHO OFF
SET VERIFY OFF
SET SERVEROUTPUT ON
DEFINE p_empid = 550
DECLARE
    v_emprecord  employees%ROWTYPE;
BEGIN
    SELECT * INTO v_emprecord FROM employees WHERE emp_id = &p_empid;
    dbms_output.put_line( 'The employee information : ' || ' Employee id – ' ||
                                    v_emprecord.emp_id || ' Salary - ' || v_emprecord.salary ) ;
END;
/
SET VERIFY ON
SET ECHO ON
```

Print information of an employee

What are Cursors?

- A cursor is a temporary work area created in the **system memory** when a SQL statement is executed.
- It contains information on a select statement and the rows of data accessed by it. This temporary work area is used to store the data retrieved from the database, and manipulate this data.
- It can hold more than one row, but can process only one row at a time. The set of rows the cursor holds is called the ***active* set**. There are two types of cursors in PL/SQL:

Implicit cursors:	***Explicit cursors:***
These are created by default when DML statements like, INSERT, UPDATE, and DELETE statements are executed. They are also created when a SELECT statement that returns just one row is executed. Oracle Server uses implicit cursors to parse and execute your SQL statement.	They must be created when you are executing a SELECT statement that returns more than one row. Even though the cursor stores multiple records, only one record can be processed at a time, which is called as current row. When you fetch a row the current row position moves to next row.
Both implicit and explicit cursors have the same functionality, but they differ in the way they are accessed.	

Implicit Cursors:

- When DML statements like DELETE, INSERT, UPDATE and SELECT statements are executed implicit cursors are created to process these statements.
- Oracle provides few attributes called as implicit **cursor attributes** to check the **status of DML operations.** The cursor attributes available are **%FOUND, %NOTFOUND, %ROWCOUNT, and %ISOPEN.**

For example, when you execute INSERT, UPDATE, or DELETE statements the cursor attributes tell us whether any rows are affected and how many have been affected. When a SELECT... INTO statement is executed in a PL/SQL Block, implicit cursor attributes can be used to find out whether any row has been returned by the SELECT statement. PL/SQL returns an error when no data is selected.

You can use these attributes in the exception section of a block to gather information about the execution of a DML statement. PL/SQL does not return an error if a DML statement does not affect any rows in the underlying table. However, if a select statement does not retrieve any rows PL/SQL returns exception.

The status of the cursor for each of these attributes are defined in the below table.

Attributes	Return Value	Example
%FOUND And %NOTFOUND	The return value is TRUE, if the DML statements like INSERT, DELETE and UPDATE affect at least one row and if SELECT ….INTO statement return at least one row. The return value is FALSE, if DML statements like INSERT, DELETE and UPDATE do not affect row and if SELECT….INTO statement do not return a row. **Return value of %NOTFOUND is just reverse of %FOUND**	SQL%FOUND SQL%NOTFOUND
%ROWCOUNT	Return the number of rows affected by the DML operations INSERT, DELETE, UPDATE, SELECT	SQL%ROWCOUNT
%ISOPEN	Always evaluates to FALSE because PL/SQL closes implicit cursors immediately after they are executed.	SQL%ISOPEN

For Example: Consider the PL/SQL Block that uses implicit cursor attributes as shown below:

```
DECLARE var_rows number(5);
BEGIN
    UPDATE employee SET salary = salary + 1000;
    IF SQL%NOTFOUND THEN
        dbms_output.put_line('None of the salaries where updated');
    ELSIF SQL%FOUND THEN
        var_rows := SQL%ROWCOUNT;
        dbms_output.put_line('Salaries for ' || var_rows || 'employees are updated');
    END IF;
END;
```

Explicit Cursors

An explicit cursor is defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row. We can provide a suitable name for the cursor. The General Syntax for creating a cursor is as given below:

CURSOR cursor_name IS select_statement;

- *cursor_name – A suitable name for the cursor.*
- *select_statement – A select query which returns multiple rows.*

How to use Explicit Cursor? There are four steps in using an Explicit Cursor.

- DECLARE the cursor in the declaration section.
- OPEN the cursor in the Execution Section.
- FETCH the data from cursor into PL/SQL variables or records in the Execution Section.

- CLOSE the cursor in the Execution Section before you end the PL/SQL Block.

Declaring a Cursor in the Declaration Section:	**DECLARE** CURSOR emp_cur IS SELECT * FROM emp_tbl WHERE salary > 5000; Once the cursor is created in the declaration section we can access the cursor in the execution section of the PL/SQL program.
Open the cursor.	**BEGIN** OPEN cursor_name;
Fetch the data from cursor	**BEGIN** FETCH cursor_name INTO record_name; OR FETCH cursor_name INTO variable_list;
Close the cursor	**BEGIN** CLOSE cursor_name;

When a cursor is opened, the first row becomes the current row.

- When the data is fetched it is copied to the record or variables and the logical pointer moves to the next row and it becomes the current row.
- On every fetch statement, the pointer moves to the next row.
- Fetching after the last row, throw an error.

Points to remember while fetching a row:

We can fetch the rows in a cursor to a PL/SQL Record or a list of variables created in the PL/SQL Block.

If you are fetching a cursor to a PL/SQL Record, the record should have the same structure as cursor.
If you are fetching a cursor to a list of variables, the variables should be listed in the same order in the fetch statement as the columns are present in the cursor.

Example :

```
DECLARE
emp_rec emp_tbl%rowtype;
CURSOR emp_cur IS
   SELECT * FROM emp_tbl WHERE salary > 10;
BEGIN
   OPEN emp_cur;
   FETCH emp_cur INTO emp_rec;
   dbms_output.put_line (emp_rec.first_name ||
       ' ' || emp_rec.last_name);
   CLOSE emp_cur;
END;
```

- First we are creating a record ‘emp_rec’ of the same structure as of table ‘emp_tbl’.
- Declaring a cursor ‘emp_cur’ from a select query.
- Opening the cursor and then fetching the cursor to the record.
- Then we are displaying the first_name and last_name of the employee in the record emp_rec.
- At last we are closing the cursor.

What are Explicit Cursor Attributes?

Oracle provides some attributes known as Explicit Cursor Attributes to control the data processing while using cursors. We use these attributes to avoid errors while accessing cursors through OPEN,

FETCH and CLOSE Statements.

These are the attributes available to check the status of an explicit cursor.

Attributes	**Return values**	**Example**
%FOUND	TRUE, if fetch statement returns at least one row. FALSE, it doesn't return a row.	Cursor_name%FOUND
%NOTFOUND	TRUE, if fetch statement doesn't return a row. FALSE, if it returns at least one row.	Cursor_name%NOTFOUND
%ROWCOUNT	The number of rows fetched by the fetch statement. If no row is returned, the PL/SQL statement returns an error.	Cursor_name%ROWCOUNT
%ISOPEN	TRUE, if the cursor is already open in the program. FALSE, if the cursor is not opened in the program.	Cursor_name%ISNAME

Using Loops with Explicit Cursors:

Oracle provides three types of loops namely SIMPLE LOOP, WHILE LOOP and FOR LOOP. These loops can be used to process multiple rows in the cursor. Here I will modify the same example for each loop to explain how to use loops with cursors.

Cursor with a Simple Loop:

```
DECLARE
   CURSOR emp_cur IS
     SELECT first_name, last_name, salary FROM emp_tbl;
   emp_rec emp_cur%rowtype;
BEGIN
   IF NOT sales_cur%ISOPEN THEN
       OPEN sales_cur;
   END IF;
   LOOP
       FETCH emp_cur INTO emp_rec;
       EXIT WHEN emp_cur%NOTFOUND;
       dbms_output.put_line (emp_cur.first_name || ' '
           ||emp_cur.last_name || ' ' ||emp_cur.salary);
   END LOOP;
END;
/
```

Cursor with a While Loop:

```
DECLARE
   CURSOR emp_cur IS
      SELECT first_name, last_name,
          salary FROM emp_tbl;
      emp_rec emp_cur%rowtype;
BEGIN
   IF NOT sales_cur%ISOPEN THEN
       OPEN sales_cur;
   END IF;
   FETCH sales_cur INTO sales_rec;
   WHILE sales_cur%FOUND THEN
       LOOP
          dbms_output.put_line
            (emp_cur.first_name || ' '
            ||emp_cur.last_name
            || ' ' ||emp_cur.salary);
          FETCH sales_cur INTO sales_rec;
       END LOOP;
END;
/
```

Stored Procedures

- A **stored procedure** or in simple a proc is a **named PL/SQL block** which performs one or more specific task. This is similar to a procedure in other programming languages.
- A procedure has a **header** and a **body**.
    - The header consists of the name of the procedure and the parameters or variables passed to the procedure.
    - The body consists or declaration section, execution section and exception section similar to a general PL/SQL Block. A procedure is similar to an anonymous PL/SQL Block but it is named for repeated usage.
- We can pass parameters to procedures in three ways : IN-parameters, OUT-parameters, IN OUT-parameters
- A procedure may or may not return any value.

General Syntax to create a procedure is:

```
CREATE [OR REPLACE] PROCEDURE proc_name
        [(list of parameters) ]
{IS | AS}
Declaration section
BEGIN
   Execution section
EXCEPTION
   Exception section
END;
```

IS | AS - marks the beginning of the body of the procedure and is similar to **DECLARE** in anonymous PL/SQL Blocks. The code between IS and BEGIN forms the Declaration section.

By using CREATE OR REPLACE together the procedure is created if no other procedure with the same name exists or the existing procedure is replaced with the current code.

Creates a procedure 'employer_details' which gives the details of the employee.

```
CREATE OR REPLACE PROCEDURE employer_details
IS
CURSOR emp_cur IS
   SELECT first_name, last_name, salary FROM emp_tbl;
   emp_rec emp_cur%rowtype;
BEGIN
  FOR emp_rec in sales_cur
    LOOP
      dbms_output.put_line (emp_cur.first_name || ' '
        ||emp_cur.last_name || ' ' ||emp_cur.salary);
    END LOOP;
END;
/
```

How to execute a Stored Procedure?

There are two ways to execute a procedure :

1) From the SQL prompt.

```
EXECUTE [or EXEC] procedure_name;
```

2) Within another procedure – simply use the procedure name.

```
procedure_name;
```

NOTE: backward slash '/' at the end of the program indicates the oracle engine that the PL/SQL program has ended and it can begin processing the statements.

PL/SQL Functions

A function is a named PL/SQL Block which is similar to a procedure. The major difference between a procedure and a function is, a function must always return a value, but a procedure may or may not return a value. The General Syntax to create a function is:

```
CREATE [OR REPLACE] FUNCTION function_name [parameters]
RETURN return_datatype;
IS | AS
Declaration_section
BEGIN
    Execution_section
    Return return_variable;
EXCEPTION
    exception section
    Return return_variable;
END;
```

- Return Type: The header section defines the return type of the function. The return datatype can be any of the oracle datatype like varchar, number etc.
- The execution and exception section both should return a value which is of the datatype defined in the header section.

For example, let's create a frunction called ''employer_details_func' similar to the one created in stored proc

```
CREATE OR REPLACE FUNCTION employer_details_func
RETURN VARCHAR(20);
IS
   emp_name VARCHAR(20);
BEGIN
  SELECT first_name INTO emp_name FROM emp_tbl
            WHERE empID = '100';
  RETURN emp_name;
END;
/
```

How to execute a PL/SQL Function?
A function can be executed in the following ways.

- Since a function returns a value we can assign it to a variable.

```
employee_name := employer_details_func;
```

- As a part of a SELECT statement

```
SELECT employer_details_func FROM dual;
```

- In a PL/SQL Statements like,

dbms_output.put_line(employer_details_func); - it displays the value returned by the function.

Parameters in Procedure and Functions

In PL/SQL, we can pass parameters to procedures and functions in three ways.

- **IN type parameter:** These types of parameters are used to send values to stored procedures. This is similar to passing parameters in programming languages. We can pass values to the stored procedure through these parameters or variables. This type of parameter is a read only parameter.

We can assign the value of IN type parameter to a variable or use it in a query, but we cannot change its value inside the procedure.

- **OUT type parameter:** These types of parameters are used to get values from stored procedures. This is similar to a return type in functions.
- **IN OUT parameter:** These types of parameters are used to send values and get values from stored procedures. By using IN OUT parameter we can pass values into a parameter and return a value to the calling program using the same parameter. But this is possible only if the value passed to the procedure and output value have a same datatype.

NOTE: If a parameter is not explicitly defined a parameter type, then by default it is an IN type parameter.

Example1: Using IN and OUT parameter:

Let's create a procedure which gets the name of the employee when the employee id is passed.

```
CREATE OR REPLACE PROCEDURE emp_name (id IN NUMBER, emp_name OUT NUMBER)
IS
BEGIN
    SELECT first_name INTO emp_name FROM emp_tbl WHERE empID = id;
END;
/
```

We can call the procedure 'emp_name' in this way from a PL/SQL Block.

```
DECLARE
   empName varchar(20);
   CURSOR id_cur SELECT id FROM emp_ids;
 BEGIN
   FOR emp_rec in id_cur
      LOOP
        emp_name(emp_rec.id, empName);
        dbms_output.putline('The employee ' || empName || ' has id ' || emp-rec.id);
      END LOOP;
END;
/
```

Example 2: Using IN OUT parameter in procedures:

```
CREATE OR REPLACE PROCEDURE emp_salary_increase
     (emp_id IN emptbl.empID%type, salary_inc IN OUT emptbl.salary%type)
IS
    tmp_sal number;
BEGIN
   SELECT salary  INTO tmp_sal FROM emp_tbl  WHERE empID = emp_id;
   IF tmp_sal between 10000 and 20000 THEN
```

```
        salary_inc := tmp_sal * 1.2;
    ELSIF tmp_sal between 20000 and 30000 THEN
        salary_inc := tmp_sal * 1.3;
    ELSIF tmp_sal > 30000 THEN
        salary_inc := tmp_sal * 1.4;
    END IF;
END;
/
```

The below PL/SQL block shows how to execute the above 'emp_salary_increase' procedure.

```
DECLARE
   CURSOR updated_sal is SELECT empID,salary FROM emp_tbl;
   pre_sal number;
BEGIN
   FOR emp_rec IN updated_sal LOOP
        pre_sal := emp_rec.salary;
        emp_salary_increase(emp_rec.empID, emp_rec.salary);
        dbms_output.put_line('The salary of ' || emp_rec.empID ||
                        ' increased from '|| pre_sal || ' to '||emp_rec.salary);
   END LOOP;
END;
/
```

Exception Handling

PL/SQL provides a feature to handle the Exceptions which occur in a PL/SQL Block known as exception Handling. Using Exception Handling we can test the code and avoid it from exiting abruptly. When an exception occurs a messages which explains its cause is received.	PL/SQL Exception message consists of three parts. 1) Type of Exception 2) An Error Code 3) A message By Handling the exceptions we can ensure a PL/SQL block does not exit abruptly.

Structure of Exception Handling.

The General Syntax for coding the exception section DECLARE Declaration section BEGIN Execution section EXCEPTION WHEN ex_name1 THEN -Error handling statements	General PL/SQL statements can be used in the Exception Block. When an exception is raised, Oracle searches for an appropriate exception handler in the exception section. For example in the above example, if the error raised is 'ex_name1 ', then the error is handled according to the statements under it. Since, it is not possible to determine all the possible runtime errors during testing for the code, the 'WHEN Others' exception is used to manage the

WHEN ex_name2 THEN -Error handling statements WHEN Others THEN -Error handling statements END;	exceptions that are not explicitly handled. Only one exception can be raised in a Block and the control does not return to the Execution Section after the error is handled.

If there are nested PL/SQL blocks like this.
DELCARE
Declaration section
BEGIN
DECLARE
Declaration section
BEGIN
Execution section
EXCEPTION
Exception section
END;
EXCEPTION
Exception section
END;
	In this case, if the exception is raised in the inner block it should be handled in the exception block of the inner PL/SQL block else the control moves to the Exception block of the next upper PL/SQL Block.
	
	If none of the blocks handle the exception the program ends abruptly with an error.

Types of Exception

There are 3 types of Exceptions.

1) Named System Exceptions 2) Unnamed System Exceptions 3) User-defined Exceptions

Named System Exceptions

System exceptions are automatically raised by Oracle, when a program violates a RDBMS rule. There are some system exceptions which are raised frequently, so they are pre-defined and given a name in Oracle which are known as Named System Exceptions.

For example: NO_DATA_FOUND and ZERO_DIVIDE are called Named System exceptions.

Named system exceptions are:

1) Not Declared explicitly,
2) Raised implicitly when a predefined Oracle error occurs,
3) Caught by referencing the standard name within an exception-handling routine.

Exception Name	Reason	Error Number
CURSOR_ALREADY_OPEN	When you open a cursor that is already open.	ORA-06511

INVALID_CURSOR	When you perform an invalid operation on a cursor like closing a cursor, fetch data from a cursor that is not opened.	ORA-01001
NO_DATA_FOUND	When a SELECT...INTO clause does not return any row from a table.	ORA-01403
TOO_MANY_ROWS	When you SELECT or fetch more than one row into a record or variable.	ORA-01422
ZERO_DIVIDE	When you attempt to divide a number by zero.	ORA-01476

For Example: Suppose a NO_DATA_FOUND exception is raised in a proc, we can write a code to handle the exception as given below.

```
BEGIN
   Execution section
EXCEPTION
   WHEN NO_DATA_FOUND THEN
       dbms_output.put_line ('A SELECT...INTO did not return any row.');
END;
```

Unnamed System Exceptions

Those system exception for which oracle does not provide a name is known as unamed system exception. These exceptions do not occur frequently. These Exceptions have a code and an associated message.

There are two ways to handle unnamed system exceptions:

1) By using the WHEN OTHERS exception handler, or
2) By associating the exception code to a name and using it as a named exception.

PRAGMA refers to a compiler directive or "hint" it is used to provide an instruction to the compiler. ... **PRAGMA**

We can assign a name to unnamed system exceptions using a **Pragma** called **EXCEPTION_INIT. EXCEPTION_INIT** will associate a predefined Oracle **error number** to a programmer_defined **exception name.**

Steps to be followed to use unnamed system exceptions are

- They are raised implicitly.
- If they are not handled in WHEN Others they must be handled explicitly.
- To handle the exception explicitly, they must be declared using Pragma EXCEPTION_INIT as given above and handled referencing the user-defined exception name in the exception section.

The general syntax to declare unnamed system exception using EXCEPTION_INIT is:

```
DECLARE
    exception_name EXCEPTION;
PRAGMA
    EXCEPTION_INIT (exception_name, Err_code);
BEGIN
    Execution section
EXCEPTION
    WHEN exception_name THEN
            handle the exception
END;
```

EXCEPTION_INIT is a PRAGMA (compiler directive) to associate exception name to error code of unnamed system exception.

The pragma must appear in the same declarative part as its associated exception, somewhere after the exception declaration.

For Example: Consider the product table and order_items table from sql joins.

Here product_id is a primary key in product table and a foreign key in order_items table. If we try to delete a product_id from the product table when it has child records in order_id table an exception will be thrown with oracle code number -2292. We can provide a name to this exception and handle it in the exception section as given below.

```
DECLARE
     Child_rec_exception EXCEPTION;
PRAGMA
     EXCEPTION_INIT (Child_rec_exception, -2292);
BEGIN
     Delete FROM product where product_id= 104;
EXCEPTION
     WHEN Child_rec_exception  THEN
          Dbms_output.put_line('Child records are present for this product_id.');
END;
/
```

User-defined Exceptions

Apart from system exceptions we can explicitly define exceptions based on business rules. These are known as user-defined exceptions. Steps to be followed to use user-defined exceptions:

- They should be explicitly declared in the declaration section.
- They should be explicitly raised in the Execution Section.
- They should be handled by referencing the user-defined exception name in the exception section.

For Example: Let's consider the product table and order_items table from sql joins to explain user-defined exception.
Let's create a business rule that if the total no of units of any particular product sold is more than 20, then it is a huge quantity and a special discount should be provided.

```
DECLARE
    huge_quantity EXCEPTION;
    CURSOR product_quantity is
        SELECT p.product_name as name, sum(o.total_units) as units FROM order_tems o, product p
        WHERE o.product_id = p.product_id;
    quantity order_tems.total_units%type;
    up_limit CONSTANT order_tems.total_units%type := 20;
    message VARCHAR2(50);
BEGIN
    FOR product_rec in product_quantity LOOP
      quantity := product_rec.units;
      IF quantity > up_limit THEN
          message := 'The number of units of product ' || product_rec.name ||
              ' is more than 20. Special discounts should be provided. Rest of the records are skipped. ' ;
          RAISE huge_quantity;
       ELSIF quantity < up_limit THEN
           v_message:= 'The number of unit is below the discount limit.';
       END IF;
       dbms_output.put_line (message);
    END LOOP;
EXCEPTION
     WHEN huge_quantity THEN  dbms_output.put_line (message);
END;
/
```

RAISE_APPLICATION_ERROR ( )

- **RAISE_APPLICATION_ERROR** is a built-in procedure in oracle which is used to display the user-defined error messages along with the error number whose range is in between -20000 and -20999.
- Whenever a message is displayed using RAISE_APPLICATION_ERROR, all previous transactions which are not committed within the PL/SQL Block are rolled back automatically (i.e. change due to INSERT, UPDATE, or DELETE statements).
- RAISE_APPLICATION_ERROR raises an exception but does not handle it.
- RAISE_APPLICATION_ERROR is used for the following reasons,
  a) to create a unique id for an user-defined exception.
  b) to make the user-defined exception look like an Oracle error.

The General Syntax to use this procedure is:

RAISE_APPLICATION_ERROR (error_number, error_message);

- The Error number must be between -20000 and -20999
- The Error_message is the message you want to display when the error occurs.

Steps to be followed to use RAISE_APPLICATION_ERROR procedure:

1) Declare a user-defined exception in the declaration section.
2) Raise the user-defined exception based on a specific business rule in the execution section.
3) Finally, catch the exception and link the exception to a user-defined error number in RAISE_APPLICATION_ERROR.

Using the above example we can display an error message using RAISE_APPLICATION_ERROR.

```
DECLARE
    huge_quantity EXCEPTION;
    CURSOR product_quantity is
        SELECT p.product_name as name, sum(o.total_units) as units
        FROM order_tems o, product p WHERE o.product_id = p.product_id;
    quantity order_tems.total_units%type;
    up_limit CONSTANT order_tems.total_units%type := 20;
    message VARCHAR2(50);
BEGIN
    FOR product_rec in product_quantity LOOP
        quantity := product_rec.units;
        IF quantity > up_limit THEN
            RAISE huge_quantity;
        ELSIF quantity < up_limit THEN
             v_message:= 'The number of unit is below the discount limit.';
        END IF;
        Dbms_output.put_line (message);
    END LOOP;
EXCEPTION
WHEN huge_quantity THEN
        raise_application_error(-2100, 'The number of unit is above the discount limit.');
END;
/
```

PL/SQL cursor variables

- A cursor variable is a variable that references to a cursor. Different from implicit and explicit cursors, a cursor variable is not tied to any specific query. Meaning that a cursor variable can be opened for any query.
- The most important benefit of a cursor variable is that it enables passing the result of a query between PL/SQL programs. Without a cursor variable, you have to fetch all data from a cursor, store it in a variable e.g., a collection, and pass this variable as an argument. With a cursor variable, you simply pass the reference to that cursor.

- To declare a cursor variable, you use the **REF CURSOR** is the data type. PL/SQL has two forms of REF CURSORs: strong and weak.

The following shows an example of a strong REF CURSOR.

It is called strong because the cursor variable associated with a specific record structure.

```
DECLARE
   TYPE customer_data_t IS REF CURSOR
                           RETURN customers%ROWTYPE;
   customer_cur   customer_data_t;
```

And here is an example of a weak REF CURSOR declaration that is not associated with any particular structure:

```
DECLARE
   TYPE customer_data_t IS REF CURSOR;
   customer_cur   customer_data_t;
```

Starting from Oracle 9i, you can use **SYS_REFCURSOR**, which is a predefined weak REF CURSOR type, to declare a weak REF CURSOR as follows:

```
DECLARE
    customer_cur SYS_REFCURSOR;
```

PL/SQL cursor variables example

Get all employees report of a manager based on the manager id from the employees table. The function returns a weak REF CURSOR variable:

```
-- function
CREATE OR REPLACE FUNCTION get_direct_reports(
     p_manager_id   IN   employees.manager_id%TYPE
)
RETURN SYS_REFCURSOR
AS
  c_direct_reports  SYS_REFCURSOR;
BEGIN
    OPEN c_direct_reports FOR
         SELECT  employee_id,  first_name,  last_name, email   FROM  employees
         WHERE  manager_id = p_manager_id    ORDER BY  first_name,   last_name;
    RETURN c_direct_reports;
END;
```

Declare c_direct_reports cursor variable of type SYS_REFCURSOR

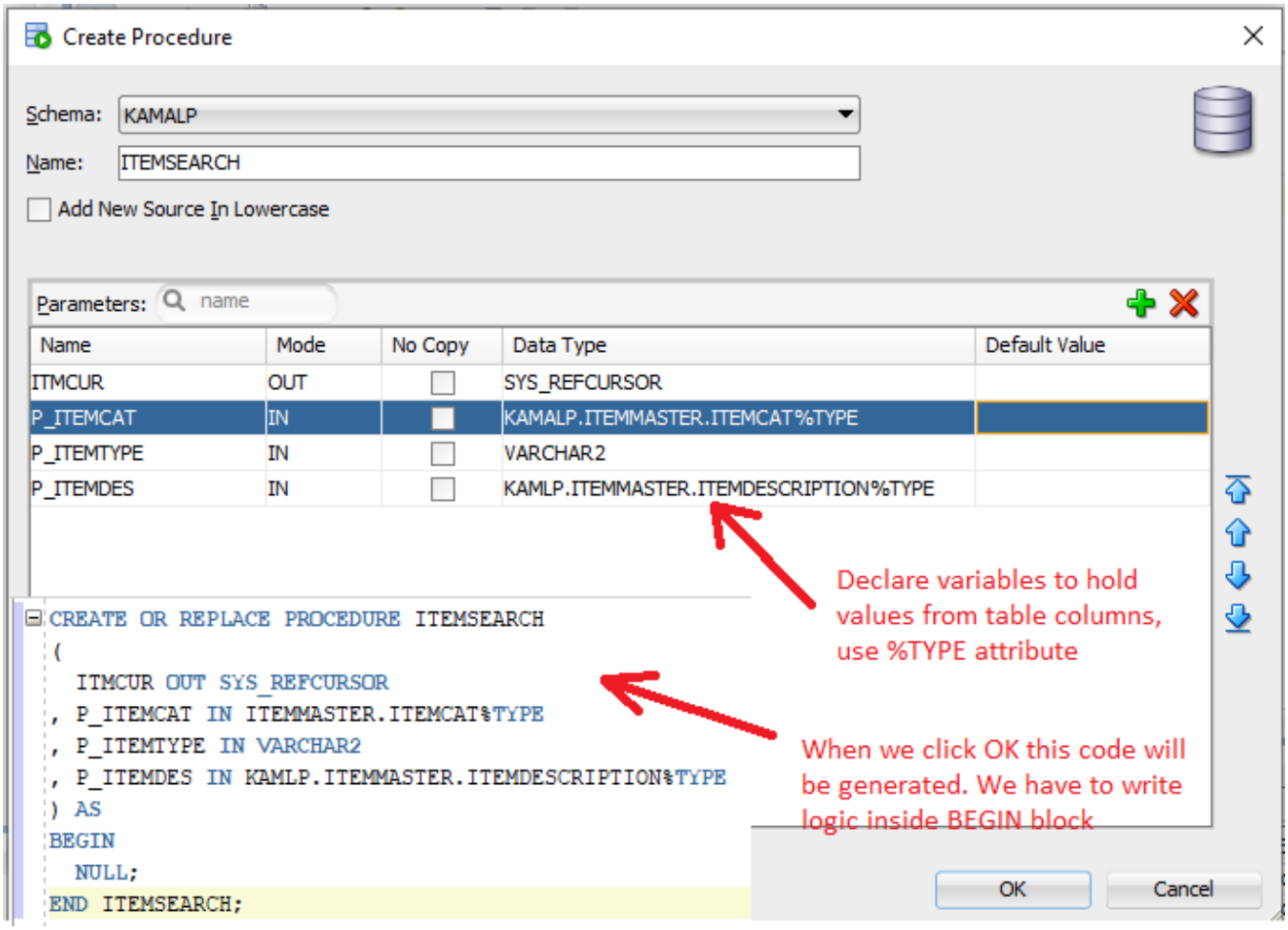
The following anonymous block calls the get_direct_reports() function and processes the cursor variable to display the direct reports of the manager whose id is 46.

```
SET serveroutput ON
DECLARE
  c_direct_reports  SYS_REFCURSOR;
  v_employee_id employees.employee_id%type;     v_first_name employees.first_name%type;
  v_last_name employees.last_name%type;          v_email employees.email%type;
BEGIN
  c_direct_reports := get_direct_reports(46);   -- get the ref cursor from function
  LOOP
    FETCH  c_direct_reports  INTO
```

```
        v_employee_id,  v_first_name,  v_last_name,  v_email;
    EXIT    WHEN c_direct_reports%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(v_first_name || ' ' || v_last_name || ' - ' ||    v_email );
  END LOOP;
END;
```

Creating SP in SQL Developer

Write a procedure to list items based on some selection criteria.

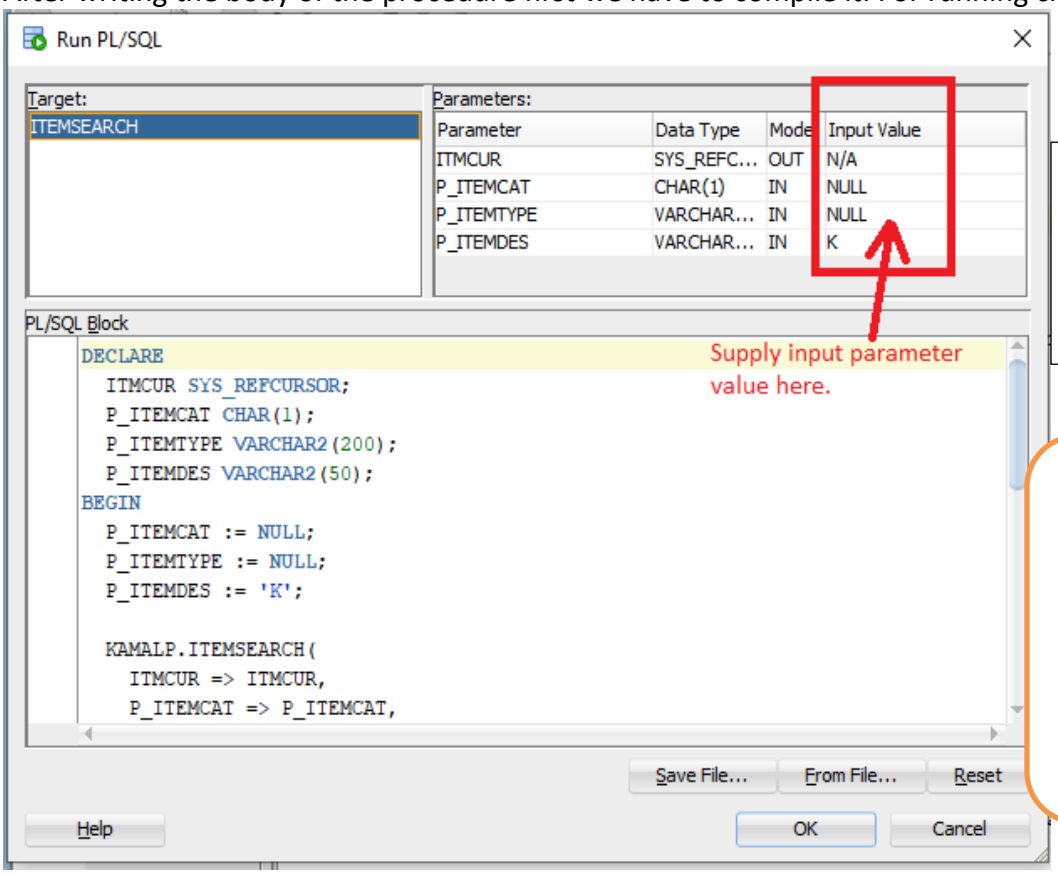



```
CREATE OR REPLACE PROCEDURE ITEMSEARCH
(
  ITMCUR OUT SYS_REFCURSOR
, P_ITEMCAT IN ITEMMASTER.ITEMCAT%TYPE
, P_ITEMTYPE IN VARCHAR2
, P_ITEMDES IN ITEMMASTER.ITEMDESCRIPTION%TYPE
) AS
BEGIN
```

While entering parameter in the tool we can either select Data Type from drop-down or enter the type directly. Here **itemmaster.itemcat%type** is entered directly.

```
  OPEN ITMCUR FOR
    SELECT  a.Itemid, a.Itemcode, a.Articlegr, a.Itemcat, a.Itemtype, a.Itemdescription,
            b.ArticlegrDescription
    FROM ITEMMASTER a
    INNER JOIN ARTICLEGROUP b   ON a.Articlegr = b.Articlegr
    WHERE (P_ITEMCAT is null OR a.Itemcat = P_ITEMCAT )
        AND  (P_ITEMTYPE is null OR a.ITEMTYPE = P_ITEMTYPE)
        AND  (P_ITEMDES is null OR a.Itemdescription LIKE concat(P_ITEMDES,'%'))
    ORDER BY a.Itemdescription;
END ITEMSEARCH;
```

After writing the body of the procedure first we have to compile it. For running choose Run button.



After supplying appropriate input parameters click on OK then SP will execute.

When we supply O as input value of P_ITEMCAT we will get following result when click on Output Variables Tab as shown below.