PL/SQL NOTES

DSO23BT/SFW20BT

Mr. M.C. Phiri
phirimc@tut.ac.za

Mr. S.K. Mogapi
mogapis@tut.ac.za

CHAPTER 1
Declaring **Variables**

The aim of the lesson is to make a student be able:

- Recognize the basic **PL/SQL** block and its sections.
- Describe the significance of **variables** in **PL/SQL**.
- Declare **PL/SQL variables**.
- Execute a PL/SQL block.

The PL/SQL block structure is divided in four **sections.**

DECLARE **- Optional**
Variables, cursors, user-defined exceptions } **Declarative environment**

BEGIN **- Mandatory/Required**
--SQL statements
--PL/SQL statements } **Executable environment**

EXCEPTION **- Optional**
Actions to perform when errors occur. } **Exceptional area**

END; **- Mandatory/Required**
/

- **Executing statements and PL/SQL blocks**
  All declared variables must be terminated by a **semi-colon(;)** **except** variables declared within a record as they are separated by a **comma(,)**

  A successful executed block is the one without **unhandled errors or compile errors,** message output be as follows:

  PL/SQL procedure successfully completed.

- **Block Types**

  A **PL/SQL** program comprises one or more blocks. The blocks can be entirely separate or nested one within another.
  One block can represent a small part of another block, which in turn can be part of the whole unit of code.

  The following are some of the blocks that a PL/SQL program:

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

  b) Subprograms
  Subprograms are named PL/SQL blocks that can accept **parameters** and can be invoked. You can declare them either as **PROCEDURES or FUNCTIONS.**

- **Use of Variables**

  Variables can be used for:

  a) Temporary storage of data – *data can be temporarily stored in more than one variable for use when validating data input and for processing later.*
  b) Manipulation of stored values – *variables can be used for calculations and other data manipulations without accessing the database.*
  c) Reusability – *after they are declared, variables can be used repeatedly in an application simply by referencing them in other statements.*
  d) Ease of maintenance – *when using **%TYPE** and **%ROWTYPE,** you declare variables, basing the declarations on the definitions of database columns.*

- **Handling Variables in PL/SQL**

  Declare and initialize variables in the declaration section.

  - Declare variables in the declarative part of any PL/SQL block.
  - Declarations allocate storage space for a value, specify its data type, and name the storage location so that you can reference it.
  - Variables must be declared first before referencing it in the block's statement.

  Assign new values to variables in the executable section.

  - In the executable section, the existing value of the variable is replaced with a new value that is assigned to the variable.

  Pass values into PL/SQL blocks through parameters.

  - There three parameter modes **IN**(by default), **OUT**, and **IN OUT**. Use **IN** parameter to pass values to either the **PROCEDURE** or **FUNCTION** being called.
  - Use the **OUT** parameters to return values to the caller of the subprogram.
  - Use the **IN OUT** parameters to pass initial values to the subprogram being called and to return updated values to the caller.

- **Types of Variables**

  a) **Scalar –** this data types hold a single value. (data types that corresponds with column types.
  b) **Composite –** they allow groups of fields to be defined and manipulated in blocks.
  c) **References –** they hold values, called pointers, but designate other program items.

```
DECLARE
  v_name    VARCHAR2(20);  VARCHAR2 is a variable-length character data. No default size.
  v_initials    CHAR(2);  CHAR is a fixed-length  character data. The length is up to 9.
  v_hiredate  DATE; it accepts a value in the format of DD/MM/YY
  v_custno    NUMBER(5); this number data type has only precision.
  v_salary     NUMBER(7,2); this number data type has a precision and scale.
  v_answer   BOOLEAN;  this data type accepts one of the two values YES/NO
   All variables declare here have a v_ as a prefix except constant variables which has a
    prefix c_
BEGIN
    SELECT columns
    INTO variables
    FROM tables
    WHERE condition using substitution
    AND another condition;
END;
/
```

```
                                    VARCHAR2(30)
VARIABLE g_binding                  NUMBER          prefix of binding variable is g.
DEFINE p_height = 23
VARIABLE g_area                     NUMBER
VARIABLE g_length                   NUMBER
VARIABLE g_width                    NUMBER
BEGIN
   :g_length := &length;
   :g_width := &width;
   :g_area := :g_length * :g_width * &p_height;
END;
/
PRINT g_area – the print cannot be used inside the PL/SQL block.
```

```
Enter value for width: 5
Enter value for width: 8

PL/SQL procedure successfully completed.

SQL> print g_area

G_AREA
----------
       920
```

- **Declaring and initializing PL/SQL Variables**

```
DECLARE                                          → The value to be used which can change.
  v_hiredate        DATE          DEFAULT SYSDATE; (today's date accepted)
  v_count           NUMBER(2)     NOT NULL :=1;
  v_emp_name        VARCHAR2(14) := '&employee_name';
  c_tax_rate        CONSTANT NUMBER(2,3) := 0.15; (initializing tax)
  v_valid           BOOLEAN       NOT NULL := FALSE;
  v_salary          NUMBER(8,2) :=0;
BEGIN

END;
```

- **The %TYPE Attribute**

Rather than hardcoding the data type e.g., **empname VARCHAR2(14)**, you can use the **%TYPE** attribute to declare a variable according to another database columns. The attribute gives the variables the data type and length of the column specified in the declaration.

```
DECLARE
  v_hiredate        emp.hiredate%TYPE := sysdate + 7; (today's plus or minus 7 days)
  v_count           NUMBER(2) NOT NULL :=1;
  v_emp_name        emp.ename%TYPE;
  v_emp_no          emp.empno%TYPE := &employeeno;
  c_tax_rate        CONSTANT NUMBER(2,3) := 0.15; (initializing tax)
  v_valid           BOOLEAN  NOT NULL := FALSE;
  v_salary          emp.sal%TYPE :=0;
BEGIN
  SELECT  ename,hiredate,sal
  INTO  v_emp_name,v_hiredate, v_salary – your SELECT statement must read INTO
  FROM emp
  WHERE empno = v_emp_no;

                                                  Works as a tab
  DBMS_OUTPUT.PUT_LINE(v_emp_name||CHR(9)||v_hiredate||CHR(9)||
   TO_CHAR(v_salary,'L99,999.99'));
  The database's DBMS is used to display individual variables with headings within
   the executable area,  because it can be used within the PL/SQL block
END;
```

Blocks cannot retrieve more than one row or records but only a single row or record. It retrieves an error if it encountered this error, the same when it cannot retrieve any row or records, in this instance your linking of primary and foreign keys from different tables might not corresponds or field being tested might be incorrectly tested.

Using the DBMS_OUTPUT.PUT_LINE, you have to declare a host variable that will then enable to display the content of the block. The declaration must be done outside the block in this format SET SERVEROUT ON

CHAPTER 2

Writing Executable Statements

The aim of the lesson is to enable students to:

- Describe the significance of the executable section.
- Use identifiers correctly.
- Write statements in executable section.
- Describe the rules of nested blocks.
- Execute and test a PL/SQL block.

PL/SQL block syntax and guidelines

The PL/SQL is an extension of SQL, the general syntax rules that apply to SQL also apply to the PL/SQL language.

A line of PL/SQL text contains groups of characters known as lexical units, which can be classified as follows:

- **Delimiters** (simple and compound symbols**). It refers to arithmetic symbols and logical operators.**

Simple Symbols	
Symbol	**Meaning**
+	Addition operator
-	Subtraction/ negation operator
*****	Multiplication operator
/	Division operator
=	Relational operator
@	Remote access indicator
;	Statement terminator

Compound Symbols	
Symbol	**Meaning**
<>	Relational operator
!=	Relational operator
II	Concatenation operator
--	Single line comment
/*	Beginning of comment delimiter
***/**	Ending of comment delimiter
:=	Assignment operator

- **Identifiers** (this include reserved words)
  - i. Can contain up to 30 characters but must begin with an alphabetic character.
  - ii. Can contain numerals, dollar signs, underscores, number signs, characters such as hyphens, slashes, and spaces.
  - iii. The following examples are incorrectly declared:
    …&--
    debit-amount → illegal hyphen
    on/off → illegal slash
    user id → illegal space.
    and these are allowed:
    money$$$tree
    SN##
    try_again_
  - iv. Do not use the **same name** for the identifier as the name of the table column.

v. Should not be reserved words such as **SELECT, FROM, WHERE**, etc using in the executions of block statements.

– **Literals**

i. Character and date literals must be enclosed in single quotation marks.

```
DECLARE
    v_name          emp.ename%TYPE;
    v_birthdate     DATE;
BEGIN
    v_name:= 'Mosimane';
    v_birthdate:= '15-AUG-89';
END;
```

ii. Numbers can be simple values or scientific notation.

– **Comments**

```
VARIABLE g_monthly_sal NUMBER
DECLARE
   v_salary         NUMBER(9,2);
BEGIN
   :g_monthly_sal := &month_salary; --prompted to enter monthly salary
   /* Compute the annual salary based on the monthly
         salary input from the user */
   v_salary := :g_monthly_sal * 12;
END;
/
/*  and  */ it is a comment that spans several lines.
-- the comment is on a single line.
```

- **Data Type Conversion**

Convert data to comparable data types.
Conversion functions:

- TO_CHAR
- TO_NUMBER
- TO_DATE

CHAPTER 3

Interacting with The Oracle Server

In this lesson the aim is enable the students to do the following:

- Write a successful **SELECT** statement in **PL/SQL.**
- Write **DML**(Data Manipulation Language) statements in PL/SQL.
- Control transactions in PL/SQL.
- Determine the outcome of **SQL Data Manipulation Language (DML)** statements.

a) SQL statements in PFL/SQL

- Extract a row of data from the database by using the **SELECT** command.
- Make changes to rows in the database by using **DML** commands.
- Control a transaction with the ***COMMIT, ROLLBACK or SAVEPOINT*** command.
- Determine DML outcome with implicit **cursor** attributes.

SELECT statement in **PL/SQL** is different from the one in **SQL** in that it uses **INTO** clause, to retrieve values into **corresponding** variables declared in the declaration area or into binding variables.

Queries must return a single row or record otherwise it will trigger errors if it returns more than one row or if it does retrieve anything.

- **Retrieving Data Using PL/SQL**

The block below retrieves the details of an employee when the user is prompted to enter the employee number, the substitution is done directly within the execution area and the value will only be used in the executable area;

```
DECLARE
  v_empname          emp.ename%TYPE;
  v_job_title        emp.job%TYPE;
  v_salary           emp.sal%TYPE;
  v_deptname         dept.dname%TYPE;
BEGIN
  SELECT ename,job,sal,dname
  INTO v_empname,v_job_title,v_salary,v_deptname
  FROM emp e, dept d
  WHERE e.empno=&employeeno – the prompted value must be linked the correct column
  AND e.deptno = d.deptno;/*querying from one table means you must be able to relate
  them according to their relationship link if you do it incorrectly it will return no rows.*/

  DBMS_OUTPUT.PUT_LINE(INITCAP(v_empname)||CHR(9)||
  INITCAP(v_job_title)||CHR(9)||TO_CHAR(v_salary,'L99,999.99')||CHR(9)||
  INITCAP(v_deptname) );
END;
/
```

The block below retrieves the details of an employee when the user is prompted to enter the employee name, the substitution is done in the declaration area so that the value received could be used throughout the block. **The substitution value is converted into UPPER case so that it corresponds with the value in the database table**;

```
DECLARE
  v_empname          emp.ename%TYPE:=UPPER('&employee_name');
  v_job_title        emp.job%TYPE;
  v_salary           emp.sal%TYPE;
  v_deptname         dept.dname%TYPE;
BEGIN
  SELECT ename,job,sal,dname
  INTO v_empname,v_job_title,v_salary,v_deptname
  FROM emp e, dept d
  WHERE e.empno=&v_empname – the prompted value must be linked the correct column
  AND e.deptno = d.deptno; /*querying from one table means you must be able to relate
  them according to their relationship link if you do it incorrectly it will return no rows.*/

  DBMS_OUTPUT.PUT_LINE(INITCAP(v_empname)||CHR(9)||
  INITCAP(v_job_title)||CHR(9)||TO_CHAR(v_salary,'L99,999.99')||CHR(9)||
  INITCAP(v_deptname) );
END;
/
```

```
Enter value for employeeno: 7788
Scott   Analyst         R3,000.00    Research
```

PL/SQL procedure successfully completed.

The block below retrieves the details of an employee when the user is prompted to enter the **job_title**. The block must calculate how many employees are in the job:

```
DECLARE
  v_deptname         dept.dname%TYPE;
  v_job_title        emp.job%TYPE:=UPPER('&Job_title');
  v_tot_employ       NUMBER(3);
BEGIN
  SELECT dname, COUNT(job) job
  INTO v_deptname, v_tot_employ
  FROM emp e, dept d
  WHERE e.empno=v_job_title– the prompted value must be linked the correct column
  AND e.deptno = d.deptno;/*querying from one table means you must be able to relate
  them according to their relationship link if you do it incorrectly it will return no rows.*/
  DBMS_OUTPUT.PUT_LINE()
END;
```

- **Manipulating Data Using PL/SQL**

The block below retrieves manipulate data in the database by either ADDING or UPDATING or DELETING a row or rows using a substitute:

```
DECLARE

BEGIN
  INSERT INTO emp VALUES(9000,'JOHN','MANGER',7788,TO_DATE(sysdate,'DD-MON-
  RR'),2500,NULL,40);
                                                    OR
  UPDATE emp
  SET sal = sal * .10
  WHERE job IN('CLERK','SALESMAN');
                                                    OR
  DELETE emp
  WHERE job = 'CLERK';
END;
/
```

CHAPTER 4
Writing **Control Structures**

The aim of the chapter is to enable students to do the following:

- Identify the uses and types of control structures.
- Construct an **IF statement** also how to use a **CASE expression**.
- Construct and identify different **LOOP** statements.
- Use logic tables.
- Control block using **nested loops** and **labels**.

a) There are three forms of **IF** statements:

- **IF THEN END IF**
- **IF THEN ELSE END IF**
- **IF THEN ELSEIF END IF**

The block below retrieves the details of an employee when the user is prompted to enter the employee name, the record retrieved is tested as to whether the **departno is 20** and **job is CLERK**. **This IF statement has only one option to choose from.**

```
DECLARE
  v_empname          emp.ename%TYPE:=UPPER('&employee_name');
  v_job_title        emp.job%TYPE;
  v_salary           emp.sal%TYPE;
  v_deptno           emp.deptno%TYPE;
  v_deptname         dept.dname%TYPE;
  v_new_salary       emp.sal%TYPE;
BEGIN
  SELECT ename,job,sal,e.deptno,dname
  INTO v_empname,v_job_title,v_salary,v_deptno,v_deptname
  FROM emp e, dept d
  WHERE e.ename=v_empname
  AND e.deptno = d.deptno;

  IF v_deptno = 20 AND v_job_title = 'CLERK' THEN
     v_new_salary := v_salary * 1.1; -- or v_new_salary:= v_salary + (v_salary * 0.1)

    DBMS_OUTPUT.PUT_LINE('Employee : '||v_empname||' employed as '||
     v_job_title||' earns '||TO_CHAR(v_salary,'fmL99,990.99')||' before it has been
      increased by 10% to '||TO_CHAR(v_salary,'fmL99,990.99')||' and is employed in the
      department '||v_deptname);
  END IF;
END;
/
```

Enter value for employee_name: **allen**

PL/SQL procedure successfully completed.

Enter value for employee_name: **adams**
Employee : **ADAMS** employed as **CLERK** earns **R1,100**. before it has been increased by **10%** to **R1,210**. and is employed in the department **RESEARCH**.

The block below retrieves the details of an employee when the user is prompted to enter the employee name, the record retrieved is tested as to whether the **job is an Analyst.** For an **Analyst** salary has to be increased by **7.5%** and the rest it must be increased by **5%**.
This IF statement has two options to choose from.

```
DECLARE
  v_empname          emp.ename%TYPE:=UPPER('&Employ_name');
  v_job_title        emp.job%TYPE;
  v_salary           emp.sal%TYPE;
  v_new_salary       emp.sal%TYPE;
BEGIN
  SELECT ename,job,sal
  INTO v_empname,v_job_title,v_salary
  FROM emp e
  WHERE e.ename=&v_empname ;

  IF v_job_title = UPPER('Analyst') THEN
     v_new_salary := v_salary * 1.075; -- or v_new_salary:= v_salary + (v_salary * 0.75)

    DBMS_OUTPUT.PUT_LINE('Employee : '||v_empname||' employed as '||
     v_job_title||' earns '||TO_CHAR(v_salary,'fmL99,990.99')||' before it has been
      increased by 10% to '||TO_CHAR(v_salary,'fmL99,990.99'));
  ELSE
     v_new_salary := v_salary * 1.05; -- or v_new_salary:= v_salary + (v_salary * 0.75)

    DBMS_OUTPUT.PUT_LINE('Employee : '||v_empname||' employed as '||
     v_job_title||' earns '||TO_CHAR(v_salary,'fmL99,990.99')||' before it has been
      increased by 10% to '||TO_CHAR(v_salary,'fmL99,990.99'));
  END IF;
END;
/
```

Enter value for employee_name: **adams**
Employee : **ADAMS** employed as **CLERK** earns **R1,100**. before it has been increased by **5%** to **R1,155**.

Enter value for employee_name: **ford**
Employee : **FORD** employed as **ANALYST** earns **R3,000**. before it has been increased by **7.5%** to **R3,225**.

PL/SQL procedure successfully completed.

The block below retrieves the details of an employee when the user is prompted to enter the **job_title** as is, the row retrieved must calculate total salary and total employees per job. The code must test for the **job** to either the **is an Analyst or Manager or Clerk.** For an **Analyst,** the salary has been increased by **5.5%,** a **Manager** will have a salary increased by **5%**, whereas the salary of the Clerk will increase by **7.5%** and the rest of the employees will have salary increase of **8%**. **This IF statement has more than one options to choose from.**

```
DECLARE
  v_empname          emp.ename%TYPE:=UPPER('&Employ_name');
  v_job_title        emp.job%TYPE;
  v_salary           emp.sal%TYPE;
  v_new_salary       emp.sal%TYPE;
BEGIN
  SELECT ename,job,sal
  INTO v_empname,v_job_title,v_salary
  FROM emp e
  WHERE e.ename=&v_empname;

  IF v_job_title = UPPER('Analyst') THEN
     v_new_salary := v_salary * 1.055; -- or v_new_salary:= v_salary + (v_salary * 0.75);
    DBMS_OUTPUT.PUT_LINE('Employee : '||v_empname||' employed as '||
     v_job_title||' earns '||TO_CHAR(v_salary,'fmL99,990.99')||' before it has been
      increased by 5.5% to '||TO_CHAR(v_salary,'fmL99,990.99'));
  ELSIF INITCAP(v_job_title) = 'Manager' THEN
      v_new_salary := v_salary * 1.05; -- or v_new_salary:= v_salary + (v_salary * 0.05);
      DBMS_OUTPUT.PUT_LINE('Employee : '||v_empname||' employed as '||v_job_title
      ||' earns '||TO_CHAR(v_salary,'fmL99,990.99')||' before it has been  increased by 5%
      to '||TO_CHAR(v_salary,'fmL99,990.99'));
  ELSIF INITCAP(v_job_title) = 'Clerk' THEN
      v_new_salary := v_salary * 1.075; -- or v_new_salary:= v_salary + (v_salary * 0.075);
      DBMS_OUTPUT.PUT_LINE('Employee : '||v_empname||' employed as '||
      v_job_title||' earns '||TO_CHAR(v_salary,'fmL99,990.99')||' before it has been
       increased by 7.5% to '||TO_CHAR(v_salary,'fmL99,990.99'));
  ELSE
     v_new_salary := v_salary * 1.08; -- or v_new_salary:= v_salary + (v_salary * 0.08);
```

```
    DBMS_OUTPUT.PUT_LINE('Employee : '||v_empname||' employed as '||v_job_title
   ||' earns '||TO_CHAR(v_salary,'fmL99,990.99')||' before it has been  increased by 8% to
     '||TO_CHAR(v_salary,'fmL99,990.99'));
  END IF;
 END;
  /
```

- **CASE expression**

  The **CASE** expression selects a result and return it. The PL/SQL CASE is little different to the SQL one. If the value of the selector is equals to the value of the WHEN clause expression, the **WHEN** clause will be executed. Few examples are shown below:

The block below retrieves the salary using the value the user is prompted to enter. The user is prompted to enter **job_title**. Then block test as to whether the **job retrieved is an Analyst or Manager or Clerk.** For an **Analyst,** the salary has been increased by **5.5%,** a **Manager** will have a salary increased by **5%**, whereas the salary of the Clerk will increase by **7.5%** and the rest of the employees will have salary increase of **8%**.

The CASE expression evaluates the value of the v_new_salary variable based on the value of the v_job_title value.

Example 1:

```
DECLARE
  v_job_title          emp.job%TYPE:=INITCAP ('&job_title');
  v_tot_salary         emp.sal%TYPE;
  v_new_salary         emp.sal%TYPE;
  v_count              NUMBER(3);
BEGIN
  SELECT COUNT(*) total,SUM(sal) tot_salary
  INTO  v_count,v_tot_salary
  FROM emp
  WHERE INITCAP(e.job)=v_job_title ;
  v_new_salary:=
    CASE (v_job_title)
        WHEN 'Analyst' THEN  v_new_salary * 1.055
        WHEN 'Manager' THEN v_new_salary * 1.05
        WHEN 'Clerk' THEN v_new_salary * 1.075
        ELSE v_new_salary * 1.08
    END;
  DBMS_OUTPUT.PUT_LINE('The job title  '||v_job_title||' has '||v_count||' pays a total
  salary of  '|| TO_CHAR(v_tot_salary,'fmL99,990.99')||' before it was increased to ' ||
 TO_CHAR(v_salary,'fmL99,990.99'));
 END;
 /
```

Example 2:

```
DECLARE
   v_job_title          emp.job%TYPE:=INITCAP ('&job_title');
   v_tot_salary         emp.sal%TYPE;
   v_new_salary         emp.sal%TYPE;
   v_count              NUMBER(3);
BEGIN
   SELECT COUNT(*) total,SUM(sal) tot_salary
   INTO  v_count,v_tot_salary
   FROM emp e
   WHERE INITCAP(e.job)=v_job_title;

   CASE INITCAP(v_job_title)
      WHEN 'Analyst' THEN v_new_salary:=  v_tot_salary * 1.055
      WHEN 'Manager' THEN v_new_salary:=  v_tot_salary * 1.05
      WHEN 'Clerk' THEN v_new_salary:=  v_tot_salary * 1.075
      ELSE v_new_salary:=  v_tot_salary * 1.08
   END;

   DBMS_OUTPUT.PUT_LINE('The job title  '||v_job_title||' has '||v_count||' pays a total
   salary of  '||TO_CHAR(v_tot_salary,'fmL99,990.99')||' before it was increased to '||
   TO_CHAR(v_salary,'fmL99,990.99'));
END;
 /
```

Enter value for job_title: **analyst**
The job title **Analyst** has **2** pays a total salary of **R6,000**. before it was increased to **R6,330**.

Enter value for job_title: **manager**
The job title **Manager** has **3** pays a total salary of **R8,275**. before it was increased to **R8,688.75.**

Example 3:

```
DECLARE
  v_job_title          emp.job%TYPE:= INITCAP('&job_title');
  v_tot_salary         emp.sal%TYPE;
  v_new_salary         emp.sal%TYPE;
  v_count              NUMBER(3);
BEGIN
  SELECT COUNT(*) total,SUM(sal) tot_salary
  INTO  v_count,v_tot_salary
  FROM emp e
  WHERE INITCAP(e.job)=v_job_title ;

  CASE
    WHEN INITCAP(v_job_title) :='Analyst' THEN v_new_salary:=  v_tot_salary * 1.055
    WHEN INITCAP(v_job_title) :='Manager' THEN v_new_salary:=  v_tot_salary * 1.05
    WHEN INITCAP(v_job_title) :='Clerk' THEN v_new_salary:=  v_tot_salary * 1.075
    ELSE v_new_salary:=  v_tot_salary * 1.08
  END;

  DBMS_OUTPUT.PUT_LINE('The job title  '||v_job_title||' has '||v_count||' pays a
  total salary of  '|| TO_CHAR(v_tot_salary,'L99,990.99')||' before it was increased to '||
  TO_CHAR(v_salary,'L99,990.99'));
END;
 /
```

b) **Iterative Control: LOOP** statements

Loops repeat a statement or sequence of statements multiple times.

Looping constructs are second type of control structure. PL/SQL provides the following types of loops:

- **Basic loop**
- **FOR loop**
- **WHILE loop**

i. **Basic Loops**

It allows execution of its statements at least one, even if the condition has been met upon entering the loop. It initials counter to 1 before execution, then the counter will increase within the loop and tested the end of the loop if it has met the required condition, if so the loop with stop execution. **Use this loop when the statements inside the loop must execute at least once. Do not forget to include the EXIT statement, because if omitted your loop will continue ENDLESSLY.**

E.g., the expression below **Adds** a record twice into a database using a BASIC loop. The loop test if the condition is met or not at its EXIT not at its beginning. It uses the counter to test if the loop has reached the number iteration it is supposed to meet.

```
DECLARE
   v_deptno          dept.deptno%TYPE;
   v_deptname        dept.dename%TYPE;
   v_location        dept.loc%TYPE :='BOSTON';
   v_counter         NUMBER(2) := 1;
BEGIN
   SELECT deptno,dname,loc
   INTO  v_deptno,v_deptname,v_location
   FROM dept
   WHERE loc=v_location;
   LOOP
     INSERT INTO dept VALUES(v_deptno + (v_counter * 10), v_deptname,
      v_location);
     v_counter:= v_counter + 1;
      EXIT WHEN v_counter > 3; --the loop will execute until counter is 4
    END LOOP;
  END;
   /
```

ii. **WHILE Loops**

With each iteration through the WHILE loop, counter which was initialized to the value of one before the loop was executed is incremented by a specific value as long as its value is within the condition in the WHILE clause but if the value is greater than the condition the loop will stop executing. In this loop the test is done at the beginning of the loop. **Use this loop when you want to repeat a sequence of statements until the controlling conditions is no longer TRUE.**

E.g., the expression below **Adds** a record twice into a database. The loop test if the condition is met or not at the beginning of the WHILE loop not at its end. It uses the counter to test if the loop has reached the number iteration it is supposed to meet.

```
DECLARE
    v_deptno            dept.deptno%TYPE;
    v_deptname          dept.dename%TYPE;
    v_location          dept.loc%TYPE :='BOSTON';
    v_counter           NUMBER(2) := 1;
BEGIN
    SELECT deptno,dname,loc
    INTO  v_deptno,v_deptname,v_location
    FROM dept
    WHERE loc=v_location;

    WHILE v_counter<= 3 LOOP
       INSERT INTO dept VALUES(v_deptno + (v_counter * 10), v_deptname,
       v_location);
       v_counter:= v_counter + 1;
    END LOOP; --the loop will execute until counter is 4
END;
 /
```

iii. **FOR Loops**

With the FOR loop counter is not declared just like we do in the other two iteration structures, it is declared in the PL/SQL server. Its value increases or decreases(only if the REVERSE word is used). The FOR loop has a lower counter value and upper counter value for the range to be successful. **Use the FOR loop if the number iterations are known.**

E.g., the expression below **Adds** a record twice into a database. The loop test if the condition is met or not at the beginning of the FOR loop not at its end. It implicitly declares the counter and uses the starting and ending values provided by the designer.

```
DECLARE
    v_deptno            dept.deptno%TYPE;
    v_deptname          dept.dename%TYPE;
    v_location          dept.loc%TYPE :='BOSTON';
    v_counter           NUMBER(2) := 1;
BEGIN
    SELECT deptno,dname,loc
    INTO  v_deptno,v_deptname,v_location
    FROM dept
    WHERE loc=v_location ;

    FOR  v_counter IN 1..3 LOOP --the loop will execute until counter is 3
        INSERT INTO dept VALUES(v_deptno + (v_counter * 10), v_deptname,
        v_location);
    END LOOP;
END;
 /
```

CHAPTER 5
Working with **Composite Data Types**

The aim of this chapter is to enable students to do the following:

- Create user-defined PL/SQL **records**.
- Create a record with the **%ROWTYPE** attribute.

Composite Data Types are of two types:

- PL/SQL records.
- PL/SQL collections
  - **INDEX BY** table.
  - **Nested** Table
  - **VARRAY**
- Contains internal components.
- Are reusable.

This semester we will deal with the first type of composite data types, which is the records. Like scalar variables, composite variables have a data type. **RECORDS** are used to treat related but not similar data as a unit. They are groups of related data items stored as fields, each with its own name and data type, separated by a comma and a semi-colon is used to terminate the record. Records have similar features as an SQL table, the difference is the heading of each.

Creating a table, we use this form:

```
CREATE TABLE table_name
(
        All the required columns or fields and their data type are written within the
        brackets. Each column is separated by a comma from the next one.
);
```

Creating a PL/SQL record will follow this declaration. Let us say we are required to create a record type as it is well-known, of all employees who earns salary more than R1700 by prompting the user to enter employee#, then a block will test if the employee earns that salary before displaying the details. The record must store the employee's number, name, job_title, salary and annul salary.

```
DECLARE
  TYPE employee_salary_type IS RECORD      cannot be used to retrieve values.
    (staffno          NUMBER(4),
     emp_name         VARCHAR(13),
     job_title        VARCHAR(15),
     salary           NUMBER(8,2),
     ann_salary       NUMBER(10,2));

     v_empno          emp.empno%TYPE:=&employee_no;
  --- declare a record below to take the structure of the record_type above
     employ_sal_rec   employee_salary_type;
BEGIN
  SELECT empno,ename,job,sal, sal * 12
  INTO employ_sal_rec
  FROM emp
  WHERE empno = v_empno;

  IF employ_sal_rec.salary > 1700 THEN
     DBMS_OUTPUT.PUT_LINE('Employee #       : '||employ_sal_rec.staffno);
     DBMS_OUTPUT.PUT_LINE('Employee Name: '||INITCAP(employ_sal_rec.emp_name);
     DBMS_OUTPUT.PUT_LINE('Job Title         : '||INITCAP(employ_sal_rec.job_title);
     DBMS_OUTPUT.PUT_LINE('Monthly Salary  : '||
     TO_CHAR(employ_sal_rec.salary,'L999,999.99');
     DBMS_OUTPUT.PUT_LINE('Annual Salary    : '||
     employ_sal_rec.ann_salary,'L999,999.99'));
  END IF;
END;
 /
```

The **%ROWTYPE** attribute declare a variable according to the collection of columns in the database table or view. Fields in the record take the names and data types from the columns of the table or view.

```
DECLARE
    v_empno          emp.empno%TYPE:=&employee_no;
    v_ann_salary     NUMBER(10,2);
    --- declare a record below to take the structure using a specific table
     employ_sal_rec   emp%ROWTYPE;
BEGIN
   SELECT empno,ename,job,sal, sal * 12
   INTO employ_sal_rec
   FROM emp
   WHERE empno = v_empno;
```

```
    IF employ_sal_rec.sal > 1700 THEN
      v_ann_salary := employ_sal_rec.sal * 12;

     DBMS_OUTPUT.PUT_LINE('Employee #       : '||employ_sal_rec.empno);
     DBMS_OUTPUT.PUT_LINE('Employee Name: '|| INITCAP(employ_sal_rec.ename);
     DBMS_OUTPUT.PUT_LINE('Job Title          : '||INITCAP(employ_sal_rec.job);
     DBMS_OUTPUT.PUT_LINE('Monthly Salary  : '||
     TO_CHAR(employ_sal_rec.sal,'L999,999.99');
     DBMS_OUTPUT.PUT_LINE('Annual Salary    : '|| v_ann_salary,'L999,999.99'));
   END IF;
END;
 /
```

```
Enter value for employee_no: 7788
Employee #          : 7788
Employee Name       : Scott
Job Title           : Analyst
Monthly Salary      : R3,000.
Annual Salary       : R36,000.
```

CHAPTER 6
Writing **Explicit Cursors**

The aim of this chapter is to enable students to do the following:

- Distinguish between an **implicit** and an **explicit cursor**.
- Discuss when and why to use an explicit cursor.
- Use a PL/SQL **record variable**.
- Write a cursor **FOR loop**.

An explicit cursor is declared and named by the designer whereas the implicit cursor is declared by PL/SQL internally for all **DML(Data Manipulation Language)**.
A cursor works the same as the pointers in C++ as the point to the selected rows or records that must be retrieved by a programming code.

a) **Explicit cursors**

Explicit cursor is used to individually process each row returned by a multiple-row SELECT statement.
The set of rows returned by the multi-row query is called the active set.
The procedure of write a cursor are as follow:

```
DECLARE
    Declare the cursor here using the SELECT statement without using INTO just like
    when creat a view.
    Declare either variables or record variables to accept the values from the cursor.
BEGIN
    Open the cursor or test if the cursor is implicitly open.
    Display all the headings here.
    Loop
        Fetch the values from the cursor into individual variables or record variable.
        Exit the loop code;
   End Loop;
   Close the cursor;
END;
```

The functions of an explicit cursor:

- Can process beyond the first row returned by the query, row by row.
- Keep track of which row is currently being processed.
- Allow the programmer to manually control explicit cursors.

The programmer is allowed to declare more than one cursor.

An example below is the cursor named **emp_hiredate_cur** that prompt the user to enter the employee name and then retrieve employee no, employee name, hire date, job title, salary,

experience in years for all employees hired after the entered employee name. Calculate and display the total employees retrieved without using **%ROWCOUNT** attribute.

- **Cursor and individual variables**

```
DECLARE
    v_emp_no        emp.empno%TYPE;
    v_emp_name      emp.ename%TYPE:=UPPER('&employeename');
    v_hiredate      DATE;
    v_ jobtitle     emp.job%TYPE;
    v_salary        emp.sal%TYPE;
    v_experience    NUMBER(3):=0;
    v_count         NUMBER(3):=0;

    CURSOR emp_hiredate_cur IS
    SELECT empno,ename,hiredate,job,sal, TO_CHAR(sysdate,'yyyy') –
    TO_CHAR(hiredate,'yyyy') experience
    FROM emp
    WHERE hiredate > ALL (SELECT hiredate
                          FROM emp
                          WHERE ename = v_emp_name);
BEGIN
  OPEN emp_hiredate_cur;
  --- headings must be displayed outside the Loop to avoid displaying them more than
      once.
  DBMS_OUTPUT.PUT_LINE('Emp No'||CHR(9)||'Emp Name'||CHR(9)||'Hire Date'||
   CHR(9)||'Job Title'||CHR(9)||'Salary'||CHR(9)||'Experience');

   DBMS_OUTPUT.PUT_LINE('------'||CHR(9)||'---------'||CHR(9)||'----------------------'||
   CHR(9)||'---------'||CHR(9)||'---------------'||CHR(9)||'------------');

  LOOP
      FETCH emp_hiredate_cur INTO v_emp_no,v_emp_name,v_hiredate,v_jobtitle,
          v_salary,v_experience;
      DBMS_OUTPUT.PUT_LINE(v_emp_no ||CHR(9)|| v_emp_name ||CHR(9)||
       hiredate||CHR(9)||v_jobtitle||CHR(9)||v_salary||CHR(9)|| v_experience);
          v_count := v_count + 1;
      EXIT WHEN emp_hiredate_cur%NOTFOUND; --stop FETCH when no records exist
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('Total employees retrieved : '||v_count); display outside
      the loop
  CLOSE emp_hiredate_cur;
END;
/
```

- **Cursor and Records**

  i. Declared Record

```
DECLARE
    TYPE staff_hiredate_type IS RECORD      cannot be used to retrieve values.
    (staffno          NUMBER(4),
     emp_name         VARCHAR(13),
     hiredate         DATE,
     job_title        VARCHAR(15),
     salary           NUMBER(8,2),
     experience       NUMBER(2));

     staff_rec        staff_hiredate_type;

     v_emp_name       emp.ename%TYPE :=UPPER('&employee_name');
     v_count          NUMBER(3):=0;

    CURSOR emp_hiredate_cur IS
    SELECT empno,ename,hiredate,job,sal, TO_CHAR(sysdate,'yyyy') –
         TO_CHAR(hiredate,'yyyy') experience
    FROM emp
    WHERE hiredate > ALL (SELECT hiredate
                          FROM emp
                          WHERE ename = v_emp_name);
BEGIN
   OPEN emp_hiredate_cur;
   --- headings must be displayed outside the Loop to avoid repetition
   DBMS_OUTPUT.PUT_LINE('Emp No'||CHR(9)||'Emp Name'||CHR(9)||'Hire Date'||
    CHR(9)||'Job Title'||CHR(9)||'Salary'||CHR(9)||'Experience');
   DBMS_OUTPUT.PUT_LINE('------'||CHR(9)||'---------'||CHR(9)||'----------------------'||
    CHR(9)||'---------'||CHR(9)||'---------------'||CHR(9)||'------------');
   LOOP
       FETCH emp_hiredate_cur INTO staff_rec;
        DBMS_OUTPUT.PUT_LINE(staff_rec.staffno ||CHR(9)||staff_rec. emp_name
         ||CHR(9)|| staff_rec.hiredate||CHR(9)||staff_rec. job_title||CHR(9)||
         staff_rec. salary||CHR(9)||staff_rec. experience);
         v_count := v_count + 1;
       EXIT WHEN emp_hiredate_cur%NOTFOUND; --stop FETCH when no records exist!
   END LOOP;
   DBMS_OUTPUT.PUT_LINE('Total employees retrieved : '||v_count);
   CLOSE emp_hiredate_cur;
END;
/
```

ii. Record derived from Cursor

```
DECLARE
    v_emp_name      emp.ename%TYPE :=UPPER('&employee_name');
    v_count         NUMBER(3):=0;

    CURSOR emp_hiredate_cur IS
    SELECT empno,ename,hiredate,job,sal, TO_CHAR(sysdate,'yyyy') –
        TO_CHAR(hiredate,'yyyy') experience
    FROM emp
    WHERE hiredate > ALL (SELECT hiredate
                          FROM emp
                          WHERE ename = v_emp_name);

    staff_rec       emp_hiredate_cur%ROWTYPE;
BEGIN
  IF NOT(emp_hiredate_cur%ISOPEN) THEN
          OPEN emp_hiredate_cur; --- test if cursor is opened if not open it
  END IF;
   --- headings must be displayed outside the Loop to avoid repetition
   DBMS_OUTPUT.PUT_LINE('Emp No'||CHR(9)||'Emp Name'||CHR(9)||'Hire Date'||
    CHR(9)||'Job Title'||CHR(9)||'Salary'||CHR(9)||'Experience');
   DBMS_OUTPUT.PUT_LINE('------'||CHR(9)||'---------'||CHR(9)||'----------------------'||
    CHR(9)||'---------'||CHR(9)||'---------------'||CHR(9)||'------------');
   LOOP
       FETCH emp_hiredate_cur INTO staff_rec;
        DBMS_OUTPUT.PUT_LINE(staff_rec.empno ||CHR(9)||staff_rec.ename
         ||CHR(9)|| staff_rec.hiredate||CHR(9)||staff_rec.job||CHR(9)||
         staff_rec.sal||CHR(9)||staff_rec.experience);

         v_count := v_count + 1; or leave it out
       EXIT WHEN emp_hiredate_cur%NOTFOUND; --stop FETCH when no records exist
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('Total employees retrieved : '|| emp_hire_cur%ROWCOUNT);
   CLOSE emp_hiredate_cur;
END;
/
```

```
Enter value for employeename: king
Emp No      Emp Name      Hire Date       Job Title       Salary          Experience
----------  ------------  --------------  --------------  --------------  ----------
7788        SCOTT         09/DEC/82       ANALYST               3000          39
7876        ADAMS         12/JAN/83       CLERK                 1100          38
7900        JAMES         03/DEC/81       CLERK                  950          40
7902        FORD          03/DEC/81       ANALYST               3000          40
7934        MILLER        23/JAN/82       CLERK                 1300          39

Total employees retrieved : 5Total employees retrieved is : 5

PL/SQL procedure successfully completed.
```

b) **Implicit cursors**

```
DECLARE
    v_emp_name      emp.ename%TYPE :=UPPER('&employee_name');
    v_count         NUMBER(3):=0;

    CURSOR emp_hiredate_cur IS
    SELECT empno,ename,hiredate,job,sal, TO_CHAR(sysdate,'yyyy') –
        TO_CHAR(hiredate,'yyyy') experience
    FROM emp
    WHERE hiredate > ALL (SELECT hiredate
                          FROM emp
                          WHERE ename = v_emp_name);
BEGIN
   --- No explicit Open
   DBMS_OUTPUT.PUT_LINE('Emp No'||CHR(9)||'Emp Name'||CHR(9)||'Hire Date'||
    CHR(9)||'Job Title'||CHR(9)||'Salary'||CHR(9)||'Experience');
   DBMS_OUTPUT.PUT_LINE('------'||CHR(9)||'---------'||CHR(9)||'----------------------'||
    CHR(9)||'---------'||CHR(9)||'---------------'||CHR(9)||'------------');
   FOR staff_rec IN emp_hiredate_cur LOOP
        --- implicit Open and Fetch
      DBMS_OUTPUT.PUT_LINE(staff_rec.empno||CHR(9)||staff_rec.ename||CHR(9)||
      staff_rec.hiredate||CHR(9)||staff_rec.job||CHR(9)||staff_rec.sal||CHR(9)||
       staff_rec.experience);
   END LOOP;
   v_count:=emp_hiredate_cur%ROWCOUNT;
   DBMS_OUTPUT.PUT_LINE('Total employees retrieved : '||v_count);
END;
/
```

CHAPTER 7
Advanced **Explicit Cursors Concepts**

The aim of the chapter is to enable the student to do the following:

- Write a cursor that uses parameters.
- Determine when a **FOR UPDATE** clause in a cursor is required.
- Determine when to use the **WHERE CURRENT OF** clause.
- Write a cursor that uses a subquery.

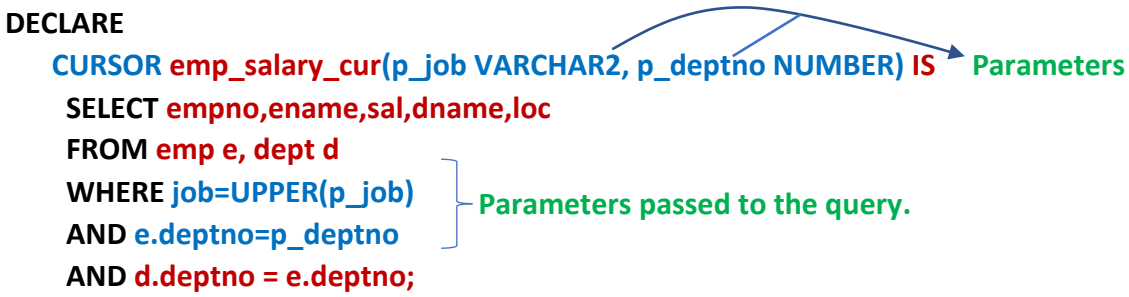
a) **Cursors with Parameters.**

Each parameter in the cursor declaration must have a corresponding actual parameter in the **OPEN** statement. Parameter data types are the same as those for scalar variables, but you do not mention their data type size. The parameter names are used for references purposes in the SELECT statement of the cursor.

A parameter cursor can accommodate one or more parameters in its declaration. As explained above the parameters are passed to the WHERE clause of the query. Parameters use the letter **p** as a prefix to their variable declaration just as we use **v** as a prefix to all declared variables.

The code or expression below passes two parameters, **job** and **deptno** to a cursor named **emp_salary_cur** to retrieve the employee's number, name, salary, department name and location. Using **%ROWCOUNT** get the number of records the cursor has retrieved.

The OPEN clause of the cursor might use entered values or prompted values to execute the cursor.




```
DECLARE
    CURSOR emp_salary_cur(p_job VARCHAR2, p_deptno NUMBER) IS
     SELECT empno,ename,sal,dname,loc
     FROM emp e, dept d
     WHERE job=UPPER(p_job)
     AND e.deptno=p_deptno
     AND d.deptno = e.deptno;

    emp_sal_rec          emp_salary_cur%ROWTYPE;
    v_count              NUMBER(3);
BEGIN
    OPEN emp_salary_cur('&job_title',&deptno); -- OR OPEN emp_salary_cur('Salesman',30);
    --- Values accepted by each parameter and passed to the SELECT statement
    DBMS_OUTPUT.PUT_LINE('Emp No'||CHR(9)||'Emp Name'||CHR(9)||'Salary'||CHR(9)
   ||'Dept Name'||CHR(9)||'Location');
     DBMS_OUTPUT.PUT_LINE('------'||CHR(9)||'---------'||CHR(9)||'---------------'||CHR(9)||
      '---------------'||CHR(9)||'---------------');

     LOOP
      FETCH emp_salary_cur INTO emp_sal_rec;
      EXIT WHEN emp_salary_cur%NOTFOUND;

     DBMS_OUTPUT.PUT_LINE(emp_sal_rec.empno||CHR(9)||INITCAP(emp_sal_rec.ename)
      ||CHR(9)||TO_CHAR(emp_sal_rec.sal,'l99,999.99')||CHR(9)||
       INITCAP(emp_sal_rec.dname) ||CHR(9)||INITCAP(emp_sal_rec.loc));
     END LOOP;
      v_count:= emp_salary_cur%ROWCOUNT;
     DBMS_OUTPUT.PUT_LINE('Total employees retrieved is : '||v_count);
     CLOSE emp_salary_cur;
    END;
     /
```

```
Enter value for job_title: clerk
Enter value for deptno: 20
Emp No     Emp Name     Salary                 Dept Name      Location
---------- ---------- ---------------------- -------------- --------------
7876       Adams              R1,100.00      Research       Dallas
7369       Smith                R800.00      Research       Dallas
Total employees retrieved is : 2

Enter value for job_title: salesman
Enter value for deptno: 30
Emp No     Emp Name     Salary                 Dept Name      Location
---------- ---------- ---------------------- -------------- --------------
7844       Turner             R1,500.00      Sales          Chicago
7654       Martin             R1,250.00      Sales          Chicago
7521       Ward               R1,250.00      Sales          Chicago
7499       Allen              R1,600.00      Sales          Chicago
Total employees retrieved is : 4
```

b) **The FOR UPDATE Clause**

This clause is used to lock rows that are supposed to be **UPDATED** or **DELETED**. It is added in the cursor query to lock the affected rows when the cursor is opened.
It must be the last clause in the SELECT statement.
The optional **NOWAIT** keyword tells Oracle not to wait if requested rows have been locked by another user for **UPDATE**.

c) **The WHERE CURRENT OF clause.**

The **WHERE CURRENT OF** clause is used to reference the current row from an explicit cursor. This allows the code to apply Updates or Deletes to the row currently being addressed.

In the expression below the code goes through the employee table to check as to whether an employee(s) who were hired after the employee's whose name was entered by the user earns a salary less than the R1800, if their salary is less than it must be increased by **6.5%.**

```
DECLARE
    v_emp_name      emp.ename%TYPE :=UPPER('&employee_name');
    v_count         NUMBER(3):=0;

    CURSOR emp_salary_cur IS
    SELECT empno,ename,hiredate,job,sal, TO_CHAR(sysdate,'yyyy') –
        TO_CHAR(hiredate,'yyyy') experience
    FROM emp
    WHERE hiredate > ALL (SELECT hiredate
                          FROM emp
                          WHERE ename = v_emp_name)
    FOR UPDATE OF sal NOWAIT;
    staff_rec       emp_hiredate_cur%ROWTYPE;
BEGIN
   OPEN emp_salary_cur;
   LOOP
      FETCH emp_salary_cur INTO staff_rec;
      EXIT WHEN emp_salary_cur%NOTFOUND;

      IF staff_rec.sal< 1800 THEN
         UPDATE emp
         SET sal = emp_salary_cur.sal * 1.065 or emp_salary_cur.sal  +
                                              ( emp_salary_cur.sal   * 0.065)
         WHERE CURRENT OF  emp_salary_cur;
      END IF;
   END LOOP;
END;
 /
```

d) **Cursors with Subqueries.**

A subquery is a query that appears within another query and is enclosed by parentheses. The subquery provides a value or set of values to the outer query.

The expression below is a cursor that uses a subquery to calculate the to number of employees per department and list all departments with more than 3 employees.

```
DECLARE
  CURSOR tot_employees_cur IS
   SELECT d.dname,d.deptno,e.total_staff
   FROM dept d,(SELECT deptno, COUNT(*) total_staff
                FROM emp e
                GROUP BY deptno) e
   WHERE d.deptno=e.deptno
   AND e.total_staff >3;
 BEGIN
  DBMS_OUTPUT.PUT_LINE (RPAD('Department Name',15)||' No. of Employees');
  DBMS_OUTPUT.PUT_LINE (RPAD('----------',15)||' ----------');
  FOR dept_rec IN tot_employees_cur LOOP
   DBMS_OUTPUT.PUT_LINE(RPAD(dept_rec.dname,15) || dept_rec.total_staff);
  END LOOP;
 END;
  /
```

```
Department Name        No. of Employees
-----------------------  ----------------------
RESEARCH                          5
SALES                             6

PL/SQL procedure successfully completed.
```

CHAPTER 8
Handling **Exceptions**

The aim of the chapter is to enable the student to do the following:

- Define PL/SQL exceptions.
- Recognize unhandled exceptions.
- List and use different types of PL/SQL **exception handlers**.
- Trap unanticipated errors.
- Describe the effect of exception propagation in Nested blocks.
- Customize PL/SQL exception messages.

- An **Exception** is an identifier in **PL/SQL** that is raised during execution.
- It is when an Oracle error occurs or can be raised by the programmer explicitly.
- Exception can be handled by:
  - Trapping it with the handler.
  - Propagating it to the calling environment.

Methods for raising an Exception:

- An Oracle error occurs, and the associated exception is raised automatically. For example, if the error **ORA-01403** occurs when no rows are retrieved from the database in a SELECT statement, then PL/SQL raise the exception **NO_DATA_FOUND**.
- You raise an exception explicitly by issuing the **RAISE** statement within the block.

Propagating an Exception

The moment the exception is raised in the executable section of the block and there is no corresponding exception handler, the PL/SQL block terminates with the failure and the exception is propagated to the calling environment.

Example:

```
DECLARE
 v_emp_id      employees.employee_id%TYPE:=&employee_no;
 v_fname       employees.first_name%TYPE;
 v_lname       employees.last_name%TYPe;
 v_job         employees.job_id%TYPE;
 v_salary      employees.salary%TYPE;
BEGIN
 SELECT first_name,last_name,job_id,salary
 INTO v_fname,v_lname,v_job,v_salary
 FROM employees
 WHERE employee_id=v_empid;
 DBMS_OUTPUT.PUT_LINE(UPPER(v_fname||' '||v_lname)||' is employed as '||v_job||'
and earns '||TO_CHAR(v_salary,'l999,999.99'));
END;
/
```

The moment it's executed the following happens;

Enter value for employee_no: 300

DECLARE

*****

ERROR at line 1:

ORA-01403: no data found

ORA-06512: at line 8

Due to the fact that exception handler is not used the PL/SQL display this message to display the error the block has encountered when it was executed.

Trapping an Exception

The moment the exception is raised in the executable section of the block, processing branches to the to the corresponding exception handler in the exception section of the block. If PL/SQL handles the exception, then the exception does not propagate to the enclosed block.

PL/SQL has three types of Exceptions , of which two are implicitly raised and the other one is explicitly raised.

- Predefined Oracle server
- Nonpredefined Oracle server

} **Implicitly raised**

- User-defined → **Explicitly raised**

Predefined error is one of approximately 20 errors that occur most often in PL/SQL code like the example I showed above in the propagation of exception. Do not declare and allow the Oracle Server to raise them implicitly. Predefined errors are listed in page **9 to 10** of this chapter 8.

Nonpredefined error is any other standard Oracle Server error. Declare within the declarative section and allow the Oracle server to raise them implicitly.

User-defined error is a condition that the developer determine as abnormal. Declare within the declarative section and raise explicitly.

Let us start with example of a predefine exception. **Exception trapping is the last section of your anonymous block. The Block consists of the declaration section, the executable section (an environment which we write the code to be executed) and lastly the exception section that is used to trap the errors.**

a) **Predefined Exception**

It has predefined names that can be used to specific error trapping, no declaration is needed.

```
Enter value for job_title: ad_assit
AD_ASSIT does not exists.

Enter value for job_title: It_prog
IT_PROG has more than one employee.
PL/SQL procedure successfully completed.

Enter value for job_title: ad_pres
STEVEN KING(100) is employed as AD_PRES and earns R24,000.
```

The code for the above output is posted below.

```
DECLARE
 v_emp_id      employees.employee_id%TYPE;
 v_fname       employees.first_name%TYPE;
 v_lname       employees.last_name%TYPe;
 v_job         employees.job_id%TYPE:=UPPER('&job_title');
 v_salary      employees.salary%TYPE;
BEGIN
 SELECT employee_id,first_name,last_name,job_id,salary
 INTO v_emp_id,v_fname,v_lname,v_job,v_salary
 FROM employees
 WHERE job_id=v_job;

 DBMS_OUTPUT.PUT_LINE(UPPER(v_fname||' '||v_lname)||'('||v_emp_id||') is
employed as '||v_job||' and earns '||TO_CHAR(v_salary,'fml999,999.99'));

EXCEPTION
 WHEN NO_DATA_FOUND THEN --if no row is found by the query then this error will be raised
        DBMS_OUTPUT.PUT_LINE(v_job||' does not exists.');
 WHEN TOO_MANY_ROWS THEN --if more than one row is found by the query then this error
will be raised
       DBMS_OUTPUT.PUT_LINE(v_job||' has more than one employee.');
 WHEN OTHERS THEN --this only become raised if the other errors are met
   DBMS_OUTPUT.PUT_LINE('No other errors');
END;
/
```

b) **Nonpredefined Exception**

It is like predefined exception but do not have predefined names but uses Oracle error number (**ORA,'####'**) and error message. It uses **EXCEPTION_INIT** function. You need to declare.

```
Enter value for job_title: sh_clerk
SH_CLERK has more than one employee.

Enter value for job_title: ac_pres
AC_PRES does not exists.
```

```
DECLARE
 v_emp_id      employees.employee_id%TYPE;
 v_fname       employees.first_name%TYPE;
 v_lname       employees.last_name%TYPE;
 v_job         employees.job_id%TYPE:=UPPER('&job_title');
 v_salary      employees.salary%TYPE;

--Declare the non-predefine exception her
 e_no_records          EXCEPTION;
 PRAGMA EXCEPTION_INIT(e_no_records,+100);

 e_more_records        EXCEPTION;
 PRAGMA EXCEPTION_INIT(e_more_records,-01422);
BEGIN
 SELECT employee_id,first_name,last_name,job_id,salary
 INTO  v_emp_id,v_fname,v_lname,v_job,v_salary
 FROM employees
 WHERE job_id=v_job;

 DBMS_OUTPUT.PUT_LINE(UPPER(v_fname||' '||v_lname)||'('||v_emp_id||') is
employed as '||v_job||' and earns '||TO_CHAR(v_salary,'fml999,999.99'));

EXCEPTION
 WHEN e_no_records THEN --if no row is found by the query then this error will be
raised
        DBMS_OUTPUT.PUT_LINE(v_job||' does not exists.');
 WHEN e_more_records THEN --if more than one row is found by the query then this
error will be raised
        DBMS_OUTPUT.PUT_LINE(v_job||' has more than one employee.');
END;
/
```

c) **User-defined Exception**

```
Enter value for job_title: ac_pres
AC_PRES does not exists.

Enter value for job_title: it_prog
IT_PROG has more than one employee.

Enter value for job_title: ad_pres
STEVEN KING(100) is employed as AD_PRES and earns R24,000.
```

```
DECLARE
 v_emp_id      employees.employee_id%TYPE;
 v_lname       employees.last_name%TYPE;
 v_job         employees.job_id%TYPE:=UPPER('&job_title');
 v_salary      employees.salary%TYPE;

 v_count       NUMBER(2):=0;
 e_no_records          EXCEPTION;     --Declare the non-predefine exception her
 e_more_records        EXCEPTION;
BEGIN
 SELECT COUNT(*) count –to count the number of rows retrieved.
 INTO v_count
 FROM employees
 WHERE job_id=v_job;

 IF v_count = 0 THE
   RAISE e_no_records;
 ELSIF v_count > 1 THEN
      RAISE e_more_records;
 ELSE
    SELECT employee_id, last_name,job_id,salary
    INTO v_emp_id,v_fname, v_job,v_salary
    FROM employees
    WHERE job_id=v_job;
    DBMS_OUTPUT.PUT_LINE(UPPER(v_lname)||'('||v_emp_id||')
    is employed as '||v_job||' and earns '||TO_CHAR(v_salary,'fml999,999.99'));
  END IF;
EXCEPTION
 WHEN e_no_records THEN--if no row is found by the query then this error will be
raised
         DBMS_OUTPUT.PUT_LINE(v_job||' does not exists.');
 WHEN e_more_records THEN --if more than one record is found by the query then this
         error will be raised
         DBMS_OUTPUT.PUT_LINE(v_job||' has more than one employee.');
END;
/
```

CHAPTER 9
Creating **Procedures**

The aim of this chapter is to enable students to do the following:

- Describe PL/SQL blocks and subprograms.
- Describe the uses of procedures.
- Create procedures.
- Differentiate between formal and actual parameters.
- List the features of different parameter modes.
- Create procedures with parameters.
- Invoke a procedure.
- Handle exceptions in procedure.
- Remove a procedure.

a) **Subprogram:**

- Is a named PL/SQL block that can accept **parameters** and be invoked from a calling environment.
- They are of two types:
  - A **procedure** that performs an action.
  - A **function** that computes a value and return answer.
- Is based on standard PL/SQL block structure.
- Provides modularity, reusability, extensibility, and maintainability.
- Provides easy maintenance, improved data security and integrity, improved performance, and improved code clarity.
- Are named PL/SQL blocks that can accept parameters and be invoked from a calling environment.
- **Subprogram specification**:
  - The header is relevant for named blocks only and determines the way that the program unit is called or invoke.
    The header determines:
    - → The PL/SQL subprogram type, i.e., either a procedure or function.
    - → The name of the subprogram.
    - → The parameter list if one exists.
    - → The RETURN clause, which applies only to functions.
  - The IS or AS keyword is mandatory.
- **Subprogram body:**
  - The declaration section of the block is between **IS|AS** and **BEGIN**. The **DECLARE** word used with anonymous block is not used here.
  - The executable section between **BEGIN** and **END** is compulsory, encloses the body of actions to be performed.
  - The exception section between **EXCEPTION** and **END** is optional.

b) **Procedures**

it is a named PL/SQL block that can accept parameters and be invoked. Procedures are used to perform actions. It has a **header**, a **declaration** section, an **executable** section, and an optional **exception-handling** section.

They can be compiled and stored in the database as a schema object.

They promote reusability and maintainability.

Procedures uses this syntax in its creation:

Indicates that if the procedure exists, it will be dropped and replaced with a new version created by the statement.

```
CREATE OR REPLACE PROCEDURE procedure_name( parameter1 IN or OUT
                                    datatype, parameter2 IN or OUT/IN OUT datatype)
IS|AS
        Declaration section
BEGIN

END procedure_name;
```

There three types of parameters:

IN (default) → **passes a constant value from the calling environment into the procedure.**
You might not specify it.
Can be assigned a default value.

OUT → **passes a value from the procedure to the calling environment.**
It must be specified.
Cannot be a variable.

IN OUT → **passes a value from the calling environment into the procedure and a different value from the procedure back to calling environment.**
It must be specified.
Cannot be assigned a default value.

Parameters apply the prefix **p** with their names.
Procedures can be executed as a standalone block or can use a cursor. It can be called by another procedure or function and it can also call another procedure or function.
Procedures and Functions can have syntax errors. The message below displays an error message after execution of the procedure.

Warning: Procedure created with compilation errors.

To retrieve these errors, you type: **show errors.**

A procedure without errors shows this message: **Procedure created** .

An expression below creates a procedure **emp_infor** that accept **employeeno** as a parameter, retrieve and display the firstname, lastname, hiredate, job_id and salary, for all employees earning a salary less that **R8000** and have an **o** as the second letter in the **lastname** without using a **LIKE** function.

The **IN** parameter only.

```
CREATE OR REPLACE PROCEDURE emp_infor (p_employeeno IN
                                        employees.employee_id%TYPE)
  IS
    v_firstname         employees.first_name%TYPE;
    v_lastname          employees.last_name%TYPE;
    v_hiredate          employees.hire_date%TYPE;
    v_job_title         employees.job_id%TYPE;
    v_salary            employees.salary%TYPE;
 BEGIN
    SELECT first_name,last_name,hire_date,job_id,salary
    INTO v_firstname,v_lastname,v_hiredate,v_job_title,v_salary
    FROM employees
    WHERE employee_id = p_employeeno –passing the parameter to the query
    AND salary < 8000
    AND SUBSTR(last_name,2,1)='o';

    DBMS_OUTPUT.PUT_LINE(UPPER(v_firstname||' '||v_lastname)||' employed as '||
    UPPER(v_job_title)||' from the '||TO_CHAR(v_hiredate,'ddth "of" fmMONTH yyyy')||
    ' with the salary of '||TO_CHAR(v_salary,'fmL999,999.99'));
END emp_infor;
  /
```

```
SQL>exec emp_infor(117) –executing the procedure
SIGAL TOBIAS employed as PU_CLERK from the 24th of JULY 1997 with the salary of R2,800.

PL/SQL procedure successfully completed.

SQL> exec emp_infor(122) –executing the procedure
*
ERROR at line 1:
ORA-01403: no data found.
ORA-06512: at "SYSTEM.EMP_INFOR", line 9
ORA-06512: at line 1
```

The **IN** and **OUT** parameters.
The example below shows how a procedure with few **IN** parameters and few **OUT** parameters works. The output can be displayed by either using an anonymous block or binding variables block that must call the procedure.

```
CREATE OR REPLACE PROCEDURE emp_infor (p_employeeno IN
                                       employees.employee_id%TYPE,
                                       p_firstname OUT
                                            employees.first_name%TYPE,
                                       p_lastname OUT
                                            employees.last_name%TYPE,
                                       p_hiredate OUT
                                            employees.hire_date%TYPE,
                                       p_job_title OUT employees.job_id%TYPE,
                                       p_salary OUT employees.salary%TYPE)
IS
BEGIN
   SELECT first_name,last_name,hire_date,job_id,salary
   INTO p_firstname,p_lastname,p_hiredate,p_job_title,p_salary
   FROM employees
   WHERE employee_id=p_employeeno
   AND salary < 8000
   AND SUBSTR(last_name,2,1)='o';
END emp_infor;
   /
```

Procedure being called by an **anonymous** PL/SQL block

```
DECLARE
  v_emp_no          employees.first_name%TYPE:=&employeesno;
  v_firstname       employees.first_name%TYPE;
  v_lastname        employees.last_name%TYPE;
  v_hiredate        employees.hire_date%TYPE;
  v_job_title       employees.job_id%TYPE;
  v_salary          employees.salary%TYPE;
BEGIN
   emp_infor(&v_emp_no,v_firstname,v_lastname,v_hiredate,v_job_title,v_salary);
         --procedure being called with corresponding parameters
   DBMS_OUTPUT.PUT_LINE(UPPER(v_firstname||' '||v_lastname)||' employed as '||
   UPPER(v_job_title)||' from the '||TO_CHAR(v_hiredate,'ddth "of" fmMONTH yyyy')||
   ' with the salary of '||TO_CHAR(v_salary,'fmL999,999.99'));
END;
   /
```

```
Enter value for employee_no: 117
SIGAL TOBIAS employed as PU_CLERK from the 24th of JULY 1997 with the salary of
R2,800.

PL/SQL procedure successfully completed.
```

A procedure being called using **binding block**.

```
VARIABLE g_firstname     VARCHAR2(15)
VARIABLE g_lastname      VARCHAR2(15)
VARIABLE g_hiredate      VARCHAR2(20)
VARIABLE g_job_title     VARCHAR2(20)
VARIABLE g_salary        NUMBER
VARIABLE g_message       VARCHAR2(100)

EXECUTE mp_infor(117,:g_firstname,:g_lastname,:g_hiredate,:g_job_title,:g_salary);

PRINT g_firstname
PRINT g_lastname
PRINT g_hiredate
PRINT g_job_title
PRINT g_salary
```

```
SQL> PRINT g_firstname
G_FIRSTNAME
--------------------------------
Sigal

SQL>     PRINT g_lastname
G_LASTNAME
--------------------------------
Tobias

SQL>     PRINT g_hiredate
G_HIREDATE
--------------------------------
24/JUL/97

SQL>     PRINT g_job_title
G_JOB_TITLE
--------------------------------
PU_CLERK

SQL>     PRINT g_salary
G_SALARY
------------
    2800
```

Procedure with a cursor

An expression below creates a procedure **emp_infor** that accept two parameters, an **employeeno** and **job_title**, then pass them to **explicit** cursor named **emp_list** which uses both parameters in a record named **emp_rec**(that uses the cursor's structures) to retrieve the firstname, lastname, hiredate, job_id and salary, of all employees earning a salary less that **R11000** and have an **o** as the second letter in the lastname without using a **LIKE** command. Create an anonymous block that will call the procedure to prompt the entry of **job_title** parameters.

```
CREATE OR REPLACE PROCEDURE emp_infor(p_job_title IN employees.job_id%TYPE)
 IS
  CURSOR emp_list_cur IS
  SELECT first_name,last_name,hire_date,job_id,salary
  FROM employees
  WHERE job_id =p_job_title  --- parameter passed by a procedure
  AND salary < 11000
  AND SUBSTR(first_name,2,1)='a';

  emp_rec              emp_list_cur%ROWTYPE;
BEGIN
  OPEN emp_list_cur;
  DBMS_OUTPUT.PUT_LINE('Employee Name'||CHR(9)||'Hiredate'||CHR(9)
  ||'Job Title'||CHR(9)||'Salary');
  DBMS_OUTPUT.PUT_LINE('-------------'||CHR(9)||'---------------'||CHR(9)
  ||'----------------'|| CHR(9)||'---------------');
  LOOP
    FETCH emp_list_cur INTO emp_rec;
    EXIT WHEN emp_list_cur%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(INITCAP(emp_rec.first_name)||', '||
    INITCAP(emp_rec.last_name)||CHR(9)||TO_CHAR(emp_rec.hire_date,'dd Month
      yyyy')||CHR(9)||INITCAP(emp_rec.job_id)||CHR(9)||
     TO_CHAR(emp_rec.salary,'l999,999.99'));
  END LOOP;
  CLOSE emp_list_cur;
 END emp_infor;
  /

DECLARE
    v_job_title          employees.job_id%TYPE:=UPPER('&job_title');
BEGIN
    emp_infor(v_job_title);  -- procedure being called or being invoked.
END;
 /
```

```
Enter value for job_title: sh_sales
Employee Name            Hiredate                  Job Title          Salary
-------------------------  -------------------------  ----------------  ----------------------

PL/SQL procedure successfully completed.

Enter value for job_title: sh_clerk
Employee Name            Hiredate                  Job Title          Salary
-------------------------  -------------------------  ----------------  ----------------------
Martha, Sullivan         21 June      1999         Sh_Clerk                    R2,500.00
Nandita, Sarchand        27 January   1996         Sh_Clerk                    R4,200.00
Randall, Perkins         19 December  1999         Sh_Clerk                    R2,500.00
Sarah, Bell              04 February  1996         Sh_Clerk                    R4,000.00
Samuel, Mccain           01 July      1998         Sh_Clerk                    R3,200.00
Vance, Jones             17 March     1999         Sh_Clerk                    R2,800.00

PL/SQL procedure successfully completed.
```

c) **Methods of passing parameters:**

Method	**Description**
Positional	Specify the same parameters in the same order as they are declared in the procedure. e.g., **emp_infor(empno,job)**
Named association	Specify the name of each parameter along with its value. The arrow => serves as the association operator. e.g., emp_infor(**empno => v_emplono**,job => v_job_title)
Combination	Specify the first parameter with positional notation, the switch to the named association for the last parameter. e.g., emp_infor(**empno, jpb =>v_job_title**)

Removing Procedures

To remove a procedure just like any other object we code **DROP PROCEDURE** procedure_name.

Benefits of Subprograms:

- Easy maintenance.
- Improved data security and integrity
- Improved performance
- Improved code.

CHAPTER 10
Creating **Functions**

The aim of the chapter is to enable the student to do the following:

- Describe the uses of functions.
- Create stored functions.
- Invoke a function.
- Remove a function.
- Differentiate between a procedure and a function.

a) Overview of stored functions

- A function is a named PL/SQL block that returns a value.
- A function can be stored in the database as a schema object for repeated execution. It can also be created at the client side application.
- A function promotes reusability and maintainability.
- A function is called as part of an expression.

b) Creating a function

As mentioned above, a function is a PL/SQL block that returns a value. You create new functions with the CREATE FUNCTION statement which may declare a list of parameters, must return one value, and must define the actions to be performed. The REPLACE statement as it is also used in most SQL and PL/SQL objects indicates that if an object such as a function exists, it will be updated with the new code added to it.

The function syntax:

CREATE OR REPLACE FUNCTION **function_name**(**parameters IN/OUT datatype, parameters IN/OUT datatype**)
RETURN datatype **– the datatype must not include the size.**
IS/AS
--- optional variables can be added here
BEGIN

END function_name;

Run the block to store the source code and compile the function.
If it returns errors the use SHOW ERRORS to see these compilation errors and rectify them.

E.g., the expression below creates a function named **total_staff** that accept **job_id** as an input parameter and calculated the total number of employees employed within the **job_id**.

```
CREATE OR REPLACE FUNCTION total_staff(p_job_title IN employees.job_id%TYPE)
   RETURN NUMBER – the data type of the value to be returned by function
   IS
    v_tot_staff   NUMBER(3):=0;
BEGIN
    SELECT COUNT(*) tot_emp
    INTO v_tot_staff
    FROM employees
    WHERE job_id=p_job_title -- parameter passed to the query
    GROUP BY job_id;

    RETURN v_tot_staff;
END total_staff;
   /
```

c) Executing Functions

A function as created above may accept one or many parameters but can return a single value. Just like in the PROCEDURE, the FUNCTION parameter may only use IN modes, because the purpose of the function is to accept no or more actual parameters and return a single value.

There are few ways that can be used to invoke a function as part of a PL/SQL expression.
The calling code must have a declared variable to hold the returned value.
Execute the function. The variable will be populated by the value returned through a RETURN statement.

Invoke the function using **binding** block.

```
VARIABLE g_total_emps          NUMBER
EXECUTE :g_total_emps:= total_staff('SH_CLERK')

 PRINT g_total_emps
```

```
G_TOTAL_EMPS
----------
        20
```

Invoke the function using an **anonymous** block.

```
DECLARE
   v_job_title          employees.job_id%TYPE:=UPPER('&job_title');
   v_total_emps         NUMBER(3):=0;
BEGIN
   v_total_emps:=total_staff(v_job_title); -- calling a function by a block

   DBMS_OUTPUT.PUT_LINE('The number of employees employed as '||v_job_title||' is
'||v_total_emps);
END;
 /
```

```
Enter value for job_title: fi_account
The number of employees employed as FI_ACCOUNT is 5

Enter value for job_title: sh_clerk
The number of employees employed as SH_CLERK is 20

Enter value for job_title: it_prog
The number of employees employed as IT_PROG is 5

Enter value for job_title: analyst
The number of employees employed as ANALYST is 0

PL/SQL procedure successfully completed.
```

The function that is invoked within a **procedure**.

E.g., the expression creates a procedure names **total_employees** that accept a job_title as a parameter and pass it to a cursor named **job_title_total_cur** to retrieve firstname, lastname, salary of each employee under the job_title accepted by the procedure and store them in a record named **staff_list_rec** that have the structure of the cursor.

Create an anonymous block that call the procedure and display a list of the employees retrieved by the cursor, calculate the **total_salary** of that job_title.

Call the function to display the number of the employees the function has returned. Display the **total_employess** and **total_salary** per job_title.

.

```
CREATE OR REPLACE PROCEDURE total_employees(p_job_title IN employees.job_id%TYPE)
    IS
     CURSOR job_title_total_cur IS
       SELECT first_name,last_name,salary
       FROM employees
       WHERE job_id = p_job_title; --passing parameter to the cursor

    staff_list_rec          job_title_total_cur%ROWTYPE; -- declare a record from cursor.
BEGIN
 OPEN job_title_total_cur;
 DBMS_OUTPUT.PUT_LINE('Employee Name'||CHR(9)||'Salary');
 DBMS_OUTPUT.PUT_LINE('-------------'||CHR(9)||'---------------');
 LOOP
  FETCH job_title_total_cur INTO staff_list_rec;
  EXIT WHEN job_title_total_cur%NOTFOUND;
  DBMS_OUTPUT.PUT_LINE(INITCAP(staff_list_rec.first_name)||', '||
  INITCAP(staff_list_rec.last_name)||CHR(9)||TO_CHAR(staff_list_rec.salary,'l999,999.99' ));
 END LOOP;
 CLOSE job_title_total_cur;
END total_employees;
 /

DECLARE
    v_total_staff           NUMBER(3);
    v_total_salary          NUMBER(12,2):=0;
    v_job_title             employees.job_id%TYPE:=UPPER('&job_id');
 BEGIN
    DBMS_OUTPUT.PUT_LINE('A list and totals of the job title: '||v_job_title);
    DBMS_OUTPUT.PUT_LINE('-----------------------------------------------------');

    total_employees(v_job_title); → procedure calling.

    SELECT SUM(salary) tot_salary
    INTO v_total_salary
    FROM employees
    WHERE job_id=v_job_title;

    v_total_staff:=total_staff(v_job_title); → function calling

    DBMS_OUTPUT.PUT_LINE(CHR(13)); → next line
    DBMS_OUTPUT.PUT_LINE(v_job_title||' has '||v_total_staff||' employees with a salary
    bill of '||TO_CHAR(v_total_salary,'fmL999,999.99'));
 END;
 /
```

Enter value for job_id: **it_prog**
A list and totals of the job title: **IT_PROG**

```
-------------------------------------------------------
Employee Name     Salary
----------------------          ----------------------
Alexander, Hunold                    R9,000.00
Bruce, Ernst                          R6,000.00
David, Austin                         R4,800.00
Valli, Pataballa                      R4,800.00
Diana, Lorentz                        R4,200.00
```

IT_PROG has **5** employees with a salary bill of **R28,800**.

Enter value for job_id: **fi_account**
A list and totals of the job title: **FI_ACCOUNT**

```
-------------------------------------------------------
Employee Name     Salary
----------------------          ----------------------
Daniel, Faviet                       R9,000.00
John, Chen                           R8,200.00
Ismael, Sciarra                      R7,700.00
Jose Manuel, Urman                   R7,800.00
Luis, Popp                           R6,900.00
```

FI_ACCOUNT has **5** employees with a salary bill of **R39,600**.

PL/SQL procedure successfully completed.