EXCEPTIONS,PROCEDURES, FUNCTIONS, PACKAGES

Errors

- Two types of errors can be found in a program: compilation errors and runtime errors.
- There is a special section in a PL/SQL block that handles the runtime errors.
- This section is called the *exception-handling section*, and in it, runtime errors are referred to as *exceptions*.
- The exception-handling section allows programmers to specify what actions should be taken when a specific exception occurs.

Exception Handling

- In order to handle run time errors in the program, an exception handler must be added.
- The exception-handling section has the following structure:

```
EXCEPTION
    WHEN EXCEPTION_NAME
    THEN
        ERROR-PROCESSING STATEMENTS;
```

- The exception-handling section is placed after the executable section of the block.

Predetermined Internal PL/SQL Exceptions(Built –in Exceptions)

1. **DUP_VAL_ON_INDEX**: Raised when an insert or update attempts to create two rows with duplicate values in columns constrained by a unique index.

2. **LOGIN_DENIED**: Raised when an invalid username/password was used to log onto Oracle.

3. **NO_DATA_FOUND**: Raised when a select statement returns zero rows.

4. **NOT_LOGGED_ON**:Raised when PL/SQL issues an oracle call without being logged onto Oracle.

5. **PROGRAM_ERROR**: Raised when PL/SQL has an internal problem.

Predetermined Internal PL/SQL Exceptions(Built –in Exceptions)

6. **TIMEOUT_ON_RESOURCE**: Raised when Oracle has been waiting to access a resource beyond the user-defined timeout limit.

7. **TOO_MANY_ROWS**: Raised when a select statement returns more than one row.

8. **VALUE_ERROR**: Raised when the data type or data size is invalid.

9. **OTHERS**: stands for all other exceptions not explicitly named

10.ZERO DIVIDE:Raised when number is divided by zero

Example:1 Named Exception

Write a PL/SQL block to handle NO_DATA_FOUND exception raised when select .. Into statement failed to fetch record.

```
DECLARE
  v_empno  emp.empno%TYPE;
  v_ename  emp.ename%TYPE;
  v_salary  emp.salary%TYPE;
BEGIN
v_empno:=&v_empno;
  SELECT ename,salary INTO v_ename,v_salary FROM emp
WHERE empno=v_empno;
  DBMS_OUTPUT.PUT_LINE(v_ename||' draws'
||v_salary||' as salary');
EXCEPTION
 WHEN NO_DATA_FOUND THEN
 DBMS_OUTPUT.PUT_LINE('NO such employee found');
END;
/
```

Example:2 Named Exception

Write a PL/SQL block to divide a number by another number, handle the exception raised ZERO_DIVIDE when denominator is zero.

```
SET SERVEROUTPUT ON
DECLARE
  v_num1 number:=&v_num1;
  v_num2 number:=&v_num2;
  v_result number:=0;
BEGIN
  v_result:=v_num1/v_num2;
  DBMS_OUTPUT.PUT_LINE('result is'||v_result);
EXCEPTION
  WHEN ZERO_DIVIDE  THEN
    DBMS_OUTPUT.PUT_LINE('A number cannot be
divided by zero');
END;
/
```

User Defined Exceptions

- For example, your program asks a user to enter a value for emp_id. This value is then assigned to the variable v_empid that is used later in the program.
- Generally, you want a positive number for an id. By mistake, the user enters a negative number.
- However, no error has occurred because emp_id has been defined as a number, and the user has supplied a legitimate numeric value.
- Therefore, you may want to implement your own exception to handle this situation.

User Defined Exceptions

- This type of an exception is called a *user-defined exceptio*n because it is defined by the programmer.
- Before the exception can be used, it must be declared.
- A user-defined exception is declared in the declarative part of a PL/SQL block as shown below:

```
DECLARE
  exception_name EXCEPTION;
```

- Once an exception has been declared, the executable statements associated with this exception are specified in the exception-handling section of the block.
- The format of the exception-handling section is the same as for built-in exceptions.

User-Defined Exception usage

```
DECLARE
   exception_name EXCEPTION;
BEGIN
  IF CONDITION   THEN
        RAISE exception_name;
   ELSE
        …
    END IF;
EXCEPTION
     WHEN exception_name THEN
          ERROR-PROCESSING STATEMENTS;
END;
/
```

Example-3 User Defined Exception

```
DECLARE
  v_empno  emp.empno%TYPE;
  v_ename  emp.ename%TYPE;
  v_salary  emp.salary%TYPE;
  ex_invalid_id EXCEPTION;
BEGIN
v_empno:=&v_empno;
 IF v_empno <= 0 THEN
      RAISE ex_invalid_id;
else
  SELECT ename,salary INTO v_ename,v_salary FROM emp WHERE
  empno=v_empno;
  DBMS_OUTPUT.PUT_LINE(v_ename||' draws '||v_salary||' as
  salary');
 end if;
```

Example-3

Write a PL/SQL block to display employee name and salary drawn by the employee with employee number entered by the user. Raise an exception when user enters an negative employee number and handler to handle it.

…Example-3

```
EXCEPTION
 WHEN ex_invalid_id THEN
    DBMS_OUTPUT.PUT_LINE('Emp no must be greater than zero');
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('NO such employee found');
  END;
/
```

Example

```
--outer block
DECLARE
      e_exception1 EXCEPTION;
      e_exception2 EXCEPTION;
BEGIN
      -- inner block
      BEGIN
            RAISE e_exception1;
      EXCEPTION
      WHEN e_exception1     THEN
            RAISE e_exception2;
```

Example contd.

```
    WHEN e_exception2     THEN
        DBMS_OUTPUT.PUT_LINE ('An error has occurred
    in the     inner' || 'block');
    END;
EXCEPTION
WHEN e_exception2 THEN
    DBMS_OUTPUT.PUT_LINE ('An error has occurred in the
    program');
END;
```

Example: Write a PL/SQL block to accept account number and withdrawal amount. First check the existence of entered account number, if account number exists, deduct withdrawal amount from Balance in the Account table. If New Balance is less than 1000/- then raise an exception with an error message –Insufficient Fund. If entered account number do not exist, system raises NO_DATA_FOUND and handle the exception with error message – Account Number do not exist.

Using OTHERS Exception

Others exception may be used in cases when some other exception is raised apart from whatever exceptions and handlers are defined in the PL/SQL Block.

Example:

Write a PL/SQL block to accept employee number of user and display employees information such as Name and Salary. Handle any exceptions raised using OTHERS

Example: OTHERS exception

```
DECLARE
        ENO EMP.EMPNO%TYPE;
        --salary emp.sal%type;
        salary number(2);
BEGIN
        ENO:=&ENO;
        SELECT SAL INTO SALARY FROM EMP WHERE EMPNO=ENO;
EXCEPTION
/*When no_data_found then
        DBMS_OUTPUT.PUT_LINE (' employee not existing');*/
WHEN OTHERS THEN
                DBMS_OUTPUT.PUT_LINE ('Some Error occurred ...');
END;
/
```

Using System Defined Numbered Exception

Assume that following table is created with check constraint on supplier_id.

```
CREATE TABLE suppliers
( supplier_id number(4),
supplier_name varchar2(50),
CONSTRAINT check_supplier_id
CHECK (supplier_id BETWEEN 100 and 9999)
);
```

Whenever **check** constraint on supplier_id is violated **ORA -02290 exception number is raised**, we can associate an Exception name to this error number ORA -02290 and use in a PL/SQL block to handle this exception.

Using System Defined Numbered Exception

- First declare the exception name –

  Supplier_ID_Range EXCEPTION;

- Associate This exception name with the error number

  Pragma EXCEPTION_INIT(Supplier_ID_Range,-02290);

- Raise the exception when condition is met
- Write the exception handler to handle the exception

Example:

```
DECLARE
   Supplier_ID_Range1 EXCEPTION;
   pragma EXCEPTION_INIT(Supplier_ID_Range1,-02290);
BEGIN
  INSERT INTO suppliers ( supplier_id, supplier_name )
   VALUES ( 1, 'IBM' );
EXCEPTION
   WHEN Supplier_ID_Range1 THEN
   DBMS_OUTPUT.PUT_LINE(' Supplier ID entered must
   be in the range 100 t0 9999');
END;
/
```

Procedures and Functions

- Oracle subprograms – includes both procedures and functions.
- Both procedures and functions:
  - Can be programmed to perform a data processing task.
  - Are named PL/SQL blocks, and both can be coded to take parameters to generalize the code.
  - Can be written with declarative, executable, and exception sections.
- Functions are typically coded to perform some type of calculation.
- Primary difference – procedures are called with PL/SQL statements while functions are called as part of an expression.

Procedures and Functions

- Procedures and functions:
  - Normally stored in the database within package specifications – a package is a sort of wrapper for a group of named blocks.
  - Can be stored as individual database objects.
  - Are parsed and compiled at the time they are stored.
  - Compiled objects execute faster than nonprocedural SQL scripts because nonprocedural scripts require extra time for compilation.
  - Can be invoked from most Oracle tools like SQL*Plus, and from other programming languages like C++ and JAVA.

Procedures

- Procedures are named PL/SQL blocks.
- Created/owned by a particular schema
- Privilege to execute a specific procedure can be granted to or revoked from application users in order to control data access.
- Requires CREATE PROCEDURE (to create in your schema) or CREATE ANY PROCEDURE privilege (to create in other schemas).

CREATE PROCEDURE Syntax

```
CREATE [OR REPLACE] PROCEDURE <procedure_name>
  (<parameter1_name> <mode> <data type>,
  <parameter2_name> <mode> <data type>, ...) {AS|IS}
    <Variable declarations>;
BEGIN
    Executable statements
[EXCEPTION
    Exception handlers]
END <optional procedure name>;
```

- Unique procedure name is required.
- OR REPLACE clause replaces if existing.
- Parameters are optional – enclosed in parentheses when used.
- AS or IS keyword is used – both work identically.
- Procedure variables are declared prior to the BEGIN keyword.
- DECLARE keyword is NOT used in named procedure.

Compiling Procedure

- To Compile/Load a procedure use either the "@" symbol or the START SQL command to compile the file. The *<SQL filename>* parameter is the .sql file that contains the procedure to be compiled.

```
SQL>@ <SQL filename>
SQL>start <SQL filename>
```

- Filename does not need to be the same as the procedure name. The **.sql** file only contains the procedure code.
- Compiled procedure is stored in the database, not the .sql file.

Showing Compilation Errors & Execution

- Compiled procedure is stored in the database, not the .sql file.
- If the .sql file is compiled, we get message –

Procedure created

Otherwise,

Warning: Procedure created with compilation errors.

Use SHOW ERRORS command if the procedure does not compile without errors.

```
SQL> show errors;
```

OR

```
SQL> SHOW ERRORS PROCEDURE Procedure_Name;
```

Use EXECUTE to run procedure.

```
SQL> EXECUTE Procedure_Name
```

Parameters

- Both procedures and functions can take parameters.
- Values passed as parameters to a procedure as arguments in a calling statement are termed *actual parameters*.
- The parameters in a procedure declaration are called *formal parameters*.
- The values stored in actual parameters are values passed to the formal parameters – the formal parameters are like placeholders to store the incoming values.
- When a procedure completes, the actual parameters are assigned the values of the formal parameters.
- Assignment of values from Formal to Actual & vice versa depends on MODE of Parameters
- A formal parameter can have one of three possible modes:

  (1) IN, (2), OUT, or (3) IN OUT.

Defining the IN, OUT, and IN OUT Parameter Modes

- **IN** – This parameter type is passed to a procedure as a read-only value that cannot be changed within the procedure – this is the default mode.
- **OUT** – this parameter type is write-only, and can only appear on the left side of an assignment statement in the procedure – it is assigned an initial value of NULL.
- **IN OUT** – this parameter type combines both IN and OUT; a parameter of this mode is passed to a procedure, and its value can be changed within the procedure.

If a procedure raises an exception, the formal parameter values are not copied back to their corresponding actual parameters.

```
--Procedure Example
SET SERVEROUTPUT ON
CREATE or REPLACE PROCEDURE squareNum(x IN number) IS
A number;
BEGIN
 A:= x * x;
 dbms_output.put_line(A);
END;
/

DECLARE
  a number;
BEGIN
  a:=&a;
  squareNum(a);
  dbms_output.put_line(' Square is:'||a);
```

Save the file as say- **proce1.sql**
Compile proce1.sql
Show errors; to know compilation errors

Main- PL/SQL block name say- **main_proc1.sql** which calls the procedure **squareNum()**

Parameter Constraint Restrictions

- Procedures do not allow specifying a constraint on the parameter data type.

```
/* Invalid constraint on parameter. */
CREATE OR REPLACE PROCEDURE proSample
  (v_Variable NUMBER(2), ...)

/* Valid parameter. */
CREATE OR REPLACE PROCEDURE proSample
  (v_Variable NUMBER, ...)
```

Procedure with No Parameters

Write a procedure to display the message- Salary is > 25000 or not for the salary of employee corresponding to the employee number passed as parameter.

```
CREATE OR REPLACE PROCEDURE DisplaySalary IS
 temp_Salary NUMBER(10,2);
BEGIN
     SELECT Salary INTO temp_Salary FROM emp
 WHERE empno=102;
     IF temp_Salary > 25000 THEN
         DBMS_OUTPUT.PUT_LINE ('Salary is >
 than 25,000.');
     ELSE
         DBMS_OUTPUT.PUT_LINE ('Salary is <=
 than 25,000.');
     END IF;
END;
/
```

Executing *DisplaySalary* Procedure

```
SQL> @ch13-4.sql                    Compilation
Procedure created.
SQL> exec DisplaySalary             execution
Salary > 25,000.
PL/SQL procedure successfully completed.
```

Passing IN and OUT Parameters..

Write a procedure to display the message- Salary is > 25000 or not for the salary of employee corresponding to the employee number passed as parameter and return the salary of the employee to calling environment.

```
CREATE OR REPLACE PROCEDURE
 DisplaySalary2(p_EmployeeID
    IN CHAR, p_Salary OUT NUMBER) IS
    v_Salary NUMBER(10,2);
BEGIN
    SELECT Salary INTO v_Salary FROM Emp
    WHERE EmpNO = p_EmployeeID;
    IF v_Salary > 25000 THEN
        DBMS_OUTPUT.PUT_LINE ('Salary >
 25,000.');
    ELSE
```

..Passing IN and OUT Parameters

```
 DBMS_OUTPUT.PUT_LINE ('Salary <= 25,000.');
    END IF;
    p_Salary := v_Salary;
EXCEPTION
     WHEN NO_DATA_FOUND THEN
         DBMS_OUTPUT.PUT_LINE ('Employee not
 found.');
END DisplaySalary2;
/
```

Example 13.6 – Calling *DisplaySalary2*

```
/* PL SQL Example */
DECLARE
    v_SalaryOutput NUMBER := 0;
BEGIN
    -- call the procedure
    DisplaySalary2(100, v_SalaryOutput);
   -- display value of salary after the call
    DBMS_OUTPUT.PUT_LINE ('Actual salary: '
        ||TO_CHAR(v_SalaryOutput));
END;
/
```

Example– Using Bind Variables

- Another approach to test a procedure. This approach uses a *bind variable* in Oracle.
- A bind variable is a variable created at the SQL*Plus prompt that is used to reference variables in PL/SQL subprograms.
- A bind variable used in this fashion must be prefixed with a colon ":" – this syntax is required.

```
/* PL SQL Example */
SQL> var v_SalaryOutput NUMBER;
SQL> EXEC DisplaySalary2('01885', :v_SalaryOutput);
Salary > 15,000.
PL/SQL procedure successfully completed.
SQL> PRINT v_SalaryOutput;

V_SALARYOUTPUT
--------------
         16250
```

Dropping a Procedure

- The SQL statement to drop a procedure is -
- DROP PROCEDURE <procedureName> ;

```
SQL> DROP PROCEDURE DisplaySalary2;
Procedure dropped.
```

Create Function Syntax

- Like a procedure, a function can accept multiple parameters, and the data type of the return value must be declared in the header of the function.

```
CREATE [OR REPLACE] FUNCTION
<function_name> (<parameter1_name> <mode>
<data type>,
   <parameter2_name> <mode> <data
type>, ...)  RETURN <return value data
type> {AS|IS}
    <Variable declarations>
BEGIN
    Executable Commands
    RETURN (return_value);
    . . .
[EXCEPTION
    Exception handlers]
END;
```

Syntax of the RETURN statement is: **RETURN <expression>;**

Example– No Parameters in Function

```
CREATE OR REPLACE FUNCTION RetrieveSalary
      RETURN NUMBER
IS
      v_Salary NUMBER(10,2);
BEGIN
      SELECT Sal INTO v_Salary
      FROM Emp
      WHERE Empno = 7369;
      RETURN v_Salary;
END RetrieveSalary;
/
```

Example – Testing *RetrieveSalary* Function

```
SQL> @ RetrieveSalary
Function created.
SQL> SELECT RetrieveSalary FROM DUAL;
RETRIEVESALARY
--------------
         25000
```

Using Bind Variable

```
SQL> var v_SalaryOutput NUMBER;
SQL> EXEC :v_SalaryOutput := RetrieveSalary;

PL/SQL procedure successfully completed.

SQL> print v_SalaryOutput;
V_SALARYOUTPUT
--------------
         25000
```

Function with Parameter

(Assume the table Employee(Empno, Firstname, MiddleName, LastName);

```
CREATE OR REPLACE FUNCTION FullName
 (p_EmployeeID IN
        employee.EmployeeID%TYPE)
    RETURN VARCHAR2 IS
    v_FullName VARCHAR2(100);
    v_FirstName employee.FirstName%TYPE;
    v_MiddleName employee.MiddleName%TYPE;
    v_LastName employee.LastName%TYPE;
BEGIN
   SELECT FirstName, MiddleName, LastName INTO
        v_FirstName, v_MiddleName, v_LastName
        FROM Employee
        WHERE EmployeeID = p_EmployeeID;
```

Function with Parameter

```
v_FullName := v_LastName||','||v_FirstName;


IF LENGTH(v_MiddleName) > 0 THEN
        v_FullName := v_FullName|| ' '
            ||SUBSTR(v_MiddleName,1,1)||'.';
    END IF;
    RETURN v_FullName;
END FullName;
/
```

Testing *FullName* Function

- A simple SELECT statement executed within SQL*Plus can return the full name for any employee identifier value as shown in PL/SQL Example 13.10.

```
/* PL SQL Example 13.10 */
SQL> SELECT FullName(101)
     FROM Employee
     WHERE EmployeeID = 101;

FULLNAME(101)
-----------------------------------
Rao, Rajesh A.
```

Testing *FullName* Function

```
/* PL SQL Example 13.11 */
SQL> SELECT FullName(EmployeeID)
     FROM Employee
     ORDER BY FullName(EmployeeID);

FULLNAME(EMPLOYEEID)
--------------------------------
Kumar, Ravi M.
Rao, Rajesh A.
```

Dropping a Function

- The SQL statement to drop a function is -

- DROP FUNCTION <functionName>;

```
SQL> DROP FUNCTION FullName;
Function dropped.
```

PACKAGE

PACKAGES

- A *package* is a collection of PL/SQL objects grouped together under one package name.
- Packages provide a means to collect related procedures, functions, cursors, declarations, types, and variables into a single, named database object.
- Package variables – can be referenced in any procedure, function, (other object) defined within a package.

Package Specification and Scope

- A package consists of a **package specification** and a **package body**.
  - The *package specification*, also called the **package header**.
    - Declares global variables, cursors, exceptions, procedures, and functions that can be called or accessed by other program units.
    - A package specification must be a **uniquely named database object**.
    - Elements of a package can declared in any order. If element "A" is referenced by another element, then element "A" must be declared before it is referenced by another element. For example, a variable referenced by a cursor must be declared before it is used by the cursor.
- Declarations of subprograms must be forward declarations.
  - This means the declaration only includes the subprogram name and arguments, but does not include the actual program code.

Create Package Specification Syntax

- Basically, a package is a named declaration section.
  - Any object that can be declared in a PL/SQL block can be declared in a package.
  - Use the CREATE OR REPLACE PACKAGE clause.
  - Include the specification of each named PL/SQL block header that will be public within the package.
  - Procedures, functions, cursors, and variables that are declared in the package specification are *global*.

The basic syntax for a package specification is:

```
CREATE [OR REPLACE PACKAGE[ <package name>
{AS|IS}
    <variable declarations>;
    <cursor declarations>;
    <procedure and function declarations>;
END <package name>;
```

Declaring Procedures and Functions within a Package

- To declare a procedure in a package – specify the procedure name, followed by the parameters and variable types:

```
PROCEDURE <procedure_name> (param1 MODE param1datatype,
 param2 MODE param2datatype, ...);
```

- To declare a function in a package, you must specify the function name, parameters and return variable type:

```
FUNCTION <function_name> (param1 MODE param1datatype,
    param2 MODE param2datatype, ...)
  RETURN <return data type>;
```

Package Body

- Contains the code for the subprograms and other constructs, such as exceptions, declared in the package specification.
- Is optional – a package that contains only variable declarations, cursors, and the like, but no procedure or function declarations does not require a package body.
- Any subprograms declared in a package specification must be coded completely in the package body. The procedure and function specifications of the package body must match the package declarations including subprogram names, parameter names, and parameter modes.

Create Package Body Syntax

- Use the CREATE OR REPLACE PACKAGE BODY clause to create a package body. The basic syntax is:

```
CREATE [OR REPLACE] PACKAGE BODY
<package name> AS
    <cursor decleration>
    <subprogram specifications
and code>
END <package name>;
```

Package Specification Example:

Example:
Create a package containing one function which returns square of a number and a procedure that returns cube of a number.

```
CREATE OR REPLACE PACKAGE Calculate1 IS

   FUNCTION squrs1(Num1 IN NUMBER) return Number;

   PROCEDURE Cubes1(Num1 IN NUMBER,Num2 OUT
      NUMBER);

END Calculate1;
/
```

Save as sql file say pack_calc_spec.sql

Package Body Example:

```
CREATE OR REPLACE PACKAGE BODY Calculate1 IS
FUNCTION squrs1(Num1 IN NUMBER) return Number IS
 RESULT NUMBER(3);
BEGIN
        RESULT:= NUM1*NUM1;
        RETURN( RESULT);
END squrs1;
PROCEDURE Cubes1(Num1 IN NUMBER,Num2 OUT
NUMBER) IS
BEGIN
        NUM2:= NUM1*NUM1*NUM1;
END Cubes1;
END Calculate1;
/
```

Save as sql file say pack_calc_body.sql

Compiling Package

Compile package specification first

SQL> @ pack_calc_spec.sql

Package created.

Compile package body

SQL>@ pack_calc_body.sql

Package Body created.

Using Package in PL/SQL Block

```
Set Serveroutput on;
DECLARE
        M1 NUMBER(3);
        RESULT NUMBER(4,1);
BEGIN
        M1:=&M1;
        Result:=Calculate1.squrs1 (M1);  -- Function CALL
        DBMS_OUTPUT.PUT_LINE ('SQuare is '|| Result);
        Calculate1.Cubes1(M1,Result);
    DBMS_OUTPUT.PUT_LINE ('Cube is '|| Result);
END;
/
```

Example Package

Create a package containing a procedure to display full name of employee corresponding to employee number entered by the user and a function to check the existence of employee with employee number.

```
CREATE OR REPLACE PACKAGE EmpFullName AS

PROCEDURE FindEmployee(
   emp_ID IN employee.EmployeeID%TYPE,
   emp_FirstName OUT employee.FirstName%TYPE,
   emp_LastName  OUT employee.LastName%TYPE );
   e_EmployeeIDNotFound EXCEPTION;

FUNCTION GoodIdentifier(
      emp_ID  IN employee.EmployeeID%TYPE )
         RETURN BOOLEAN;
END EmpFullName;
/
```

Package Body

```
CREATE OR REPLACE PACKAGE BODY EmpFullName AS

   -- Procedure to find employees

   PROCEDURE FindEmployee(

      emp_ID IN employee.EmployeeID%TYPE,

      emp_FirstName OUT employee.FirstName%TYPE,

      emp_LastName  OUT employee.LastName%TYPE ) AS

   BEGIN

      SELECT FirstName, LastName

         INTO emp_FirstName, emp_LastName

         FROM Employee WHERE EmployeeID = emp_ID;

               -- Check for existence of employee

         IF SQL%ROWCOUNT = 0 THEN

             RAISE e_EmployeeIDNotFound;

         END IF;

   END FindEmployee;
```

Example– Package Body

```
FUNCTION GoodIdentifier(
        emp_ID  IN employee.EmployeeID%TYPE)
        RETURN BOOLEAN   IS
    v_ID_Count NUMBER;
  BEGIN
        SELECT COUNT(*) INTO v_ID_Count
        FROM Employee
        WHERE EmployeeID = emp_ID;
        -- return TRUE if v_ID_COUNT is 1
        RETURN (1 = v_ID_Count);
    EXCEPTION
        WHEN OTHERS THEN
            RETURN FALSE;
  END GoodIdentifier;
END EmpFullName;
/
```

Calling Package Procedure/Function

```
/* PL SQL */
DECLARE
    v_FirstName    employee.FirstName%TYPE;
    v_LastName     employee.LastName%TYPE;
 search_emp_id      employee.EmployeeID%TYPE;
BEGIN
 EmpFullName.FindEmployee (&search_emp_id,
 v_FirstName, v_LastName);
    DBMS_OUTPUT.PUT_LINE ('The employee name is:'
     || v_LastName || ', ' || v_FirstName);
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE ('Cannot find an
 employee with that ID.');
END;
/
```

Results of Calling Package Procedure

- When the employee identifier is valid, the code displays the employee name as shown here.

```
Enter value for search_emp_id: 100
The employee name is: Kumar, Ravi
PL/SQL procedure successfully completed.
```

- When the identifier is not valid, the exception raised within the called procedure is propagated back to the calling procedure and is trapped by the EXCEPTION section's WHEN OTHERS clause and an appropriate message is displayed as shown here.

```
Enter value for search_emp_id: 99
Cannot find an employee with that ID.
PL/SQL procedure successfully completed.
```

Cursors in Packages

```
create or replace package packg_cursor is
    cursor c_emp is select
     empno,ename,sal,hiredate from emp where
     deptno=10;

     r_emp c_emp%ROWTYPE;

     procedure p_printEmps;
end;
/
```

Cursors in Packages

```
create or replace package body packg_cursor is
    procedure p_printEmps is
       r_emp c_emp%ROWTYPE;
    begin
            open c_emp;
            loop
                fetch c_emp into r_emp;
                exit when c_emp%NOTFOUND;
      DBMS_OUTPUT.put_line(r_emp.empno);
      DBMS_OUTPUT.put_line(r_emp.ename);
      DBMS_OUTPUT.put_line(r_emp.sal||'
'||r_emp.hiredate);
            end loop;
            close c_emp;
        end;
   end;
   /
```

Executing Cursors in Packages

Either we can use a PL/SQL block to call a procedure in a package.

OR

Use EXEC command

```
SQL> EXEC packg_cursor. p_printEmps

7782
CLARK
2450  09-JUN-81
7839
KING
5000  17-NOV-81
7934
MILLER
1300  23-JAN-82
```

Cursors in Packages

- A *cursor variable* can make a cursor dynamic so that it is reusable and sharable among different procedures and functions such as those created as part of a package.
- A cursor variable has data type REF CURSOR. It is like a pointer in the C language, and it points to a query work area where a result set is stored.
- First you must define a REF CURSOR type.
- Next, you define a cursor variable of that type. In this general syntactic example, the <return_type> object represents a row in a database table.

```
TYPE ref_type_name IS REF CURSOR
    [RETURN <return_type>];
```

- This provides an example of declaring a cursor variable that can be used to process data rows for the *equipment* table of the Madison Hospital database.

```
DECLARE
    TYPE equipment_Type IS REF CURSOR
        RETURN equipment%ROWTYPE;
    cv_Equipment IN OUT equipment_Type;
```

Example 13.15 – REF CURSOR Type

- The Package Specification declares a REF CURSOR type named *equipment_Type* and two procedures named *OpenItem* and *FetchItem*.
- The cursor cv_Equipment in the *OpenItem* procedure is declared as an IN OUT parameter – it will store an equipment item after the procedure is executed—it is this stored value that is input to the *FetchItem* procedure.

```
/* PL SQL Example 13.15 File:  ch13-15.sql */
CREATE OR REPLACE PACKAGE ManageEquipment AS
    -- Create REF CURSOR type
    TYPE equipment_Type IS REF CURSOR
        RETURN equipment%ROWTYPE;
    -- Declare procedure
    PROCEDURE OpenItem (cv_Equipment IN OUT equipment_Type,
        p_EquipmentNumber IN CHAR);
    -- Declare procedure to fetch an equipment item
    PROCEDURE FetchItem (cv_Equipment IN equipment_Type,
        equipment_Row OUT equipment%ROWTYPE);
END ManageEquipment;
```

Example 13.16 – Package Body

```
/* PL SQL Example 13.16 File: ch13-16.sql */
CREATE OR REPLACE PACKAGE BODY ManageEquipment AS
    -- Procedure to get a specific item of equipment
    PROCEDURE OpenItem (cv_Equipment IN OUT equipment_Type,
        p_EquipmentNumber IN CHAR) AS
    BEGIN
        -- Populate the cursor
        OPEN cv_Equipment FOR
        SELECT * FROM Equipment
        WHERE EquipmentNumber = p_EquipmentNumber;
    END OpenItem;

    PROCEDURE FetchItem (cv_Equipment IN equipment_Type,
        equipment_Row OUT equipment%ROWTYPE) AS
    BEGIN
        FETCH cv_Equipment INTO equipment_Row;
    END FetchItem;
END ManageEquipment;
```

Example 13.16 – Use Cursor Variable

```
/* PL SQL Example 13.16 File: ch13-16.sql */
DECLARE
    -- Declare a cursor variable of the REF CURSOR type
    item_Cursor ManageEquipment.equipment_Type;
    v_EquipmentNumber equipment.EquipmentNumber%TYPE;
    equipment_Row equipment%ROWTYPE;
BEGIN
    -- Assign a equipment number to the variable
    v_EquipmentNumber := '5001';
    -- Open the cursor using a variable
    ManageEquipment.OpenItem (item_Cursor, v_EquipmentNumber);
    -- Fetch the equipment data and display it
    LOOP
          ManageEquipment.FetchItem( item_Cursor, equipment_Row);
          EXIT WHEN item_cursor%NOTFOUND;
          DBMS_OUTPUT.PUT (equipment_Row.EquipmentNumber || '   ');
          DBMS_OUTPUT.PUT_LINE (equipment_Row.Description);
     END LOOP;
 END;

5001  Computer, Desktop
PL/SQL procedure successfully completed.
```