PL/SQL (Procedural Language Extension to SQL) (Part 2)

INFS602 Physical Database Design

Stored Functions

3. Named Blocks (stored functions)

- A function is a named PL/SQL block that returns a value.
- A function can be stored in the database, as a database object, for repeated execution.
- A function can be called as part of an expression.

- **The major difference between a procedure and a function is, a function must always return a value, but a procedure may or may not return a value.**

General Syntax to create a function

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

defines the return type of the function. The return datatype can be any of the oracle datatype like varchar, number etc.

NOTE: should return a value which is of the datatype defined in the header section.

Stored Function Example

```
CREATE OR REPLACE FUNCTION get_sal
  (v_id IN emp.empno%TYPE)
RETURN NUMBER
IS
 v_salary emp.sal%TYPE :=0;
BEGIN
 SELECT sal
 INTO v_salary
 FROM emp
 WHERE empno = v_id;
 RETURN (v_salary);
END get_sal;
/
```

Executing Functions

- We can use a host variable to quickly execute and test the function

```
SQL> VARIABLE g_salary NUMBER
SQL> EXECUTE :g_salary := get_sal(7934)

SQL> PRINT g_salary
```

- User-defined function can be called from any SQL expression wherever a built-in function can be called

- 1) Since a function returns a value we can assign it to a variable.
    - *employee_name := employer_details_func;*
- If ‘employee_name’ is of datatype varchar we can store the name of the employee by assigning the return type of the function to it.
- 2) As a part of a SELECT statement
    - *SELECT employer_details_func FROM dual;*
- 3) In a PL/SQL Statements like,
    - *dbms_output.put_line(employer_details_func);*
- This line displays the value returned by the function.

Exercise

- Create a function called Q_PROD to return a product description when passed a ProdID as a parameter.

```
SQL> desc prod
 Name              Null?    Type
 -------------     -------- ----
 PRODID            NOT NULL NUMBER(6)
 DESCRIP                    VARCHAR2(30)
```

- Create a function ANNUAL_COMP to return the annual salary when passed an employee’s monthly salary and annual commission.

Stored Function Restrictions

- A user-defined function must be a ROW function not a GROUP function.
- A user-defined function only takes IN parameters.
- When called from a SELECT statement the function cannot modify any database tables.
- When called from an INSERT, UPDATE, or DELETE statement, the function cannot query or modify any database tables modified by that statement.

Comparing Procedures and Functions

Procedure	Function
Execute as a PL/SQL statement	Invoke as part of an expression
No RETURN datatype	Must contain a RETURN datatype
Can return one or more values	Must return a value

Programming Guidelines

- Document code with comments
- Develop a case convention for the code
- Develop naming convention for identifiers and other objects
- Enhance readability by indenting

Cursors

- Pointer to a memory location that the DBMS uses to process a SQL query

- A *cursor* is a PL/SQL construct used to process a SQL statement one row at a time.

- Used to retrieve and manipulate database data

SQL Statements in PL/SQL

- Extract a row of data from the database by using the SELECT command. Only a single set of values can be returned (Implicit Cursor).
- The SELECT statement defines a virtual table called the *result set* that contains all the rows of the underlying SELECT statement.
- The cursor's attributes provide information about the cursor's structure and current status.
- Make changes to rows in the database by using DML (Data Manipulation Language) commands
- Control transactions with the COMMIT, ROLLBACK, or SAVEPOINT command.

SELECT Statements in PL/SQL

```
DECLARE
   V_deptno NUMBER(2);
   V_loc        VARCHAR2(15);
BEGIN
   SELECT deptno, loc INTO v_deptno, v_loc
   FROM dept
   WHERE dname = ‘SALES’
…
END;
```

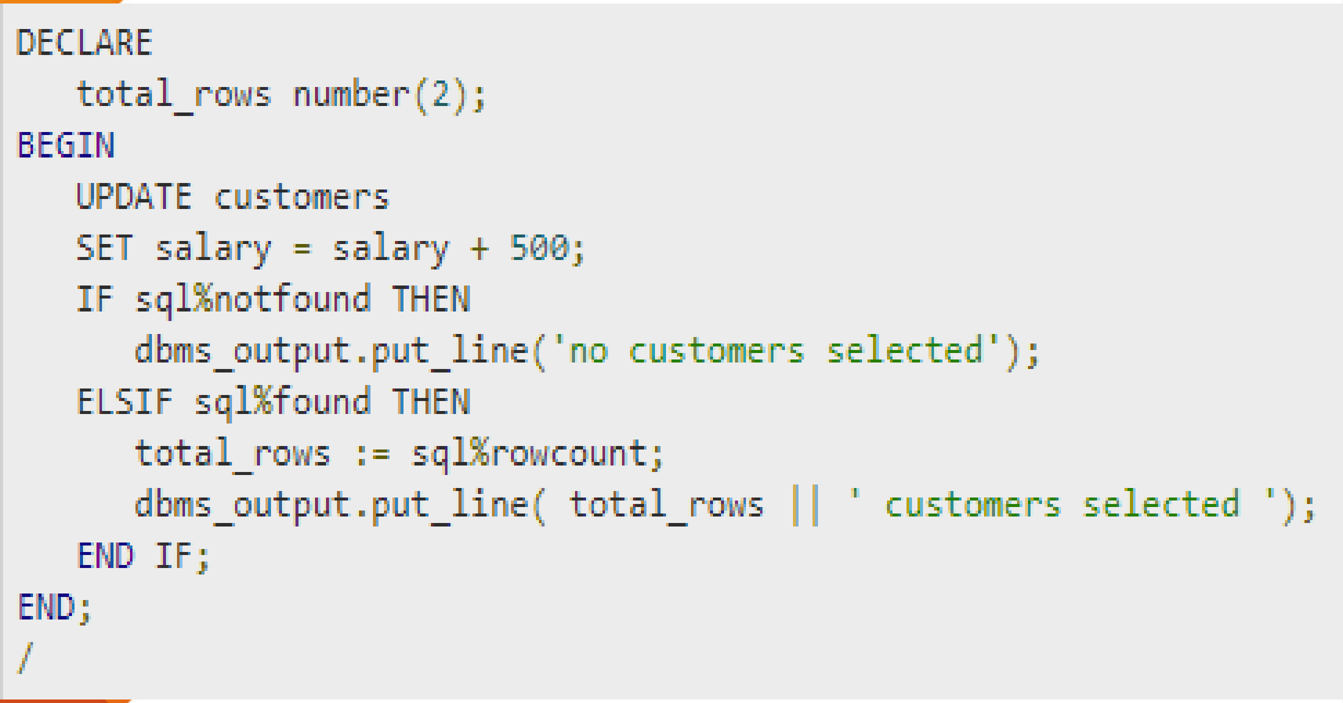
SQL Cursor

- A cursor is an SQL work area (temporary working area)
- Two type of cursors
  - Implicit cursors
  - Explicit cursors
- Both implicit and explicit cursors have the same functionality, but they differ in the way they are accessed.
- PL/SQL implicitly declares a cursor for all SQL data manipulation statements and queries that return only one row.
- For queries that return more than one row the programmer must explicitly declare a cursor! IMPORTANT!

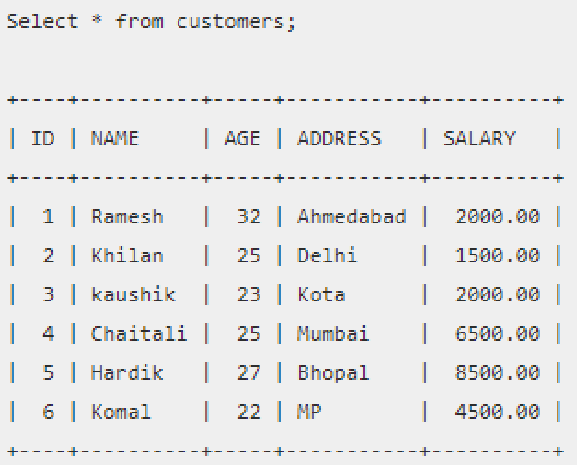
SQL Implicit Cursor Attributes

SQL%ROWCOUNT	Number of rows affected by the most recent SQL statement
SQL%FOUND	Boolean attribute that evaluates to TRUE if the most recent SQL statement affects one or more rows
SQL%NOTFOUND	Boolean attribute that evaluate to TRUE if the most recent SQL does not affect any rows
SQL%ISOPEN	Always evaluates to FALSE because PL/SQL closes implicit cursors immediately after they are executed

Implicite Cursor Example

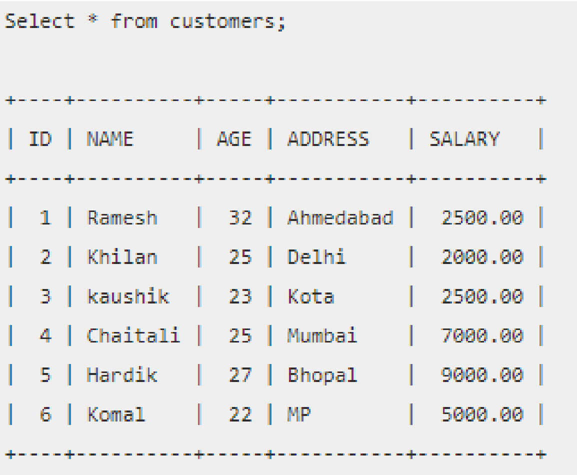


```
DECLARE
   total_rows number(2);
BEGIN
   UPDATE customers
   SET salary = salary + 500;
   IF sql%notfound THEN
      dbms_output.put_line('no customers selected');
   ELSIF sql%found THEN
      total_rows := sql%rowcount;
      dbms_output.put_line( total_rows || ' customers selected ');
   END IF;
END;
/
```



```
Select * from customers;

+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```



```
Select * from customers;

+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2500.00 |
|  2 | Khilan   |  25 | Delhi     |  2000.00 |
|  3 | kaushik  |  23 | Kota      |  2500.00 |
|  4 | Chaitali |  25 | Mumbai    |  7000.00 |
|  5 | Hardik   |  27 | Bhopal    |  9000.00 |
|  6 | Komal    |  22 | MP        |  5000.00 |
+----+----------+-----+-----------+----------+
```

PL/SQL Records

- Similar in structure to records in a 3GL
- Convenient for fetching a row of data from a table for processing.

```
…
TYPE emp_record_type IS RECORD
    (ename          VARCHAR2(10),
     Job            VARCHAR2(9),
     Sal            NUMBER(7,2));
emp_record          emp_record_type;
…
```

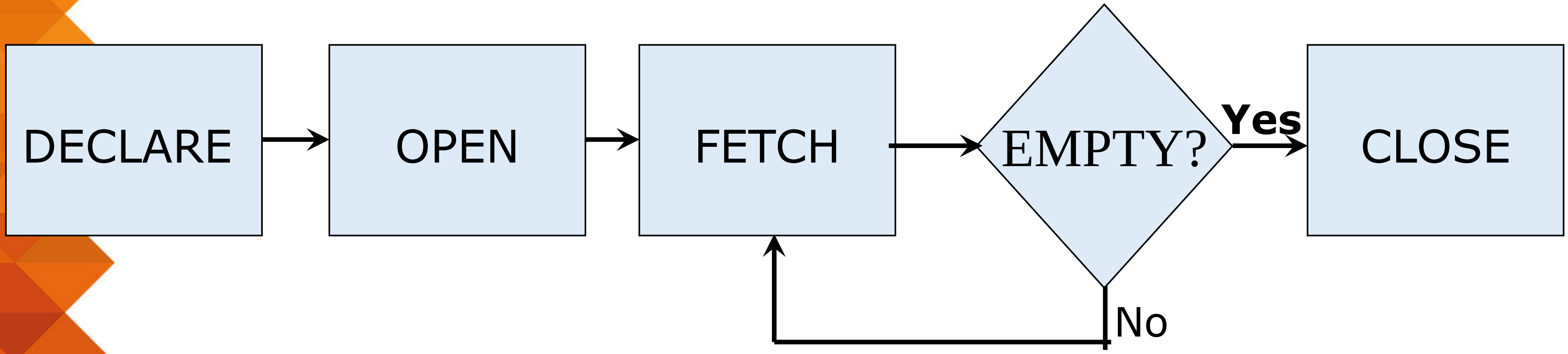
The %ROWTYPE Attribute

- Declare a variable according to a collection of columns in a database table or view.
- Prefix %ROWTYPE with the database table.
- Fields in the record take their name and datatypes from the columns of the table or view.

```
DECLARE
   Emp_record          emp%ROWTYPE;
```

Explicit Cursors

- Explicit cursors are programmer-defined cursors for gaining more control over the **context area**.
- Explicit cursors are named SQL work areas to manipulate queries returning more than one row.
- Use DECLARE, OPEN, FETCH and CLOSE to control explicit cursors.



- An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.

Working with Explicit Cursors

- Working with an explicit cursor includes the following steps;

- Declaring the cursor for initializing the memory
- Opening the cursor for allocating the memory
- Fetching the cursor for retrieving the data
- Closing the cursor to release the allocated memory

Declaring the Cursor

NOTE: declare a variable that will hold the information read from the cursor

This variable, declared as a **record**

- Syntax

```
CURSOR cursor_name IS
     SELECT_statement;
v_empno                emp.empno%Type
v_eName                emp.ename%Type
v_deptRec  dept%RowType
```

NOTE

- Examples

```
CURSOR emp_cursor IS
   SELECT empno, ename
   FROM emp;
CURSOR dept_cursor IS
   SELECT *
   FROM dept;
```

Opening the Cursor

- Syntax

```
OPEN cursor_name;
```

- Example

```
OPEN emp_cursor;
```

- Open the cursor to execute the query and identify the active set.
- The cursor now points to the first row in the active set

Fetching Data From the Cursor

- Syntax

```
FETCH cursor name INTO [variable1,
variable2, …]|record_name];
```

- Example

```
FETCH emp_cursor INTO v_empNo, v_eName;
FETCH dept_cursor INTO v_deptRec;
```

- Retrieve the current row values into variable(s) or record.
- Include the same number of variables.

Closing the Cursor

- Syntax

```
CLOSE cursor_name;
```

- Close the cursor after completing the processing of the rows

SQL Explicit Cursor Attributes

%ROWCOUNT	Evaluate to the total number of rows returned so far
%FOUND	Boolean attribute that evaluates to TRUE if the most recent fetch returns a row
%NOTFOUND	Boolean attribute that evaluate to TRUE if the most recent fetch does not return a row
%ISOPEN	Evaluates to TRUE if the cursor is open

Controlling Multiple Fetches

- Process several rows from an explicit cursor using a loop
- Fetch a row with each iteration
- Use the %NOTFOUND attribute to write a test for an unsuccessful fetch

Example Cursor

```
DECLARE
   V_empno       emp.empno%TYPE;
   V_ename       emp.ename%TYPE;
   CURSOR emp_cursor IS
      SELECT empno, ename FROM emp;
BEGIN
   OPEN emp_cursor;
   LOOP
      FETCH emp_cursor INTO v_empno, v_ename;
      EXIT WHEN emp_cursor%NOTFOUND;
      … do something with the cursor row
   END LOOP;
   CLOSE emp_cursor;
END;
```

Cursor FOR Loops

- Syntax

```
FOR record_name IN cursor_name
 LOOP
   Statement1;
   Statement2;
   …
END LOOP;
```

- *Implicit (automatic) open, fetch and close occur.*
- *The record is implicitly declared.*

Example Cursor For Loop

- DECLARE
  - CURSOR emp_cursor IS
    - SELECT empno, ename
    - FROM emp;
- BEGIN
  - FOR emp_record IN emp_cursor LOOP
    - … do required processing with emp_record
  - END LOOP;
- END;

Exceptions

- Errors are known as exceptions. An exception occurs when an unwanted situation arises during the execution of a program.
- Can result from a system error, a user error, or an application error.
- When an exception occurs, control of the current program block shifts to another section of the program, known as the exception handler.

Handling Exceptions

- Three types of exception
  - Predefined Oracle Server
  - Non-predefined Oracle Server
  - User Defined

Predefined Exceptions

- Sample predefined exception
    - NO_DATA_FOUND
    - TOO_MANY_ROWS
    - INVALID_CURSOR
    - ZERO_DIVIDE
    - DUP_VAL_ON_INDEX

    - Complete list is available in the PL/SQL User’s Guide and Reference, “Error Handling”

Handling Exceptions…

- Syntax

```
EXCEPTION
   WHEN exception1 [OR exception2 …] THEN
      Statement1;
      Statement2;
      …
   WHEN OTHERS THEN
      Statement1;
      Statement2;
      …
```

Some Examples

- You can write handlers for predefined exceptions using their names

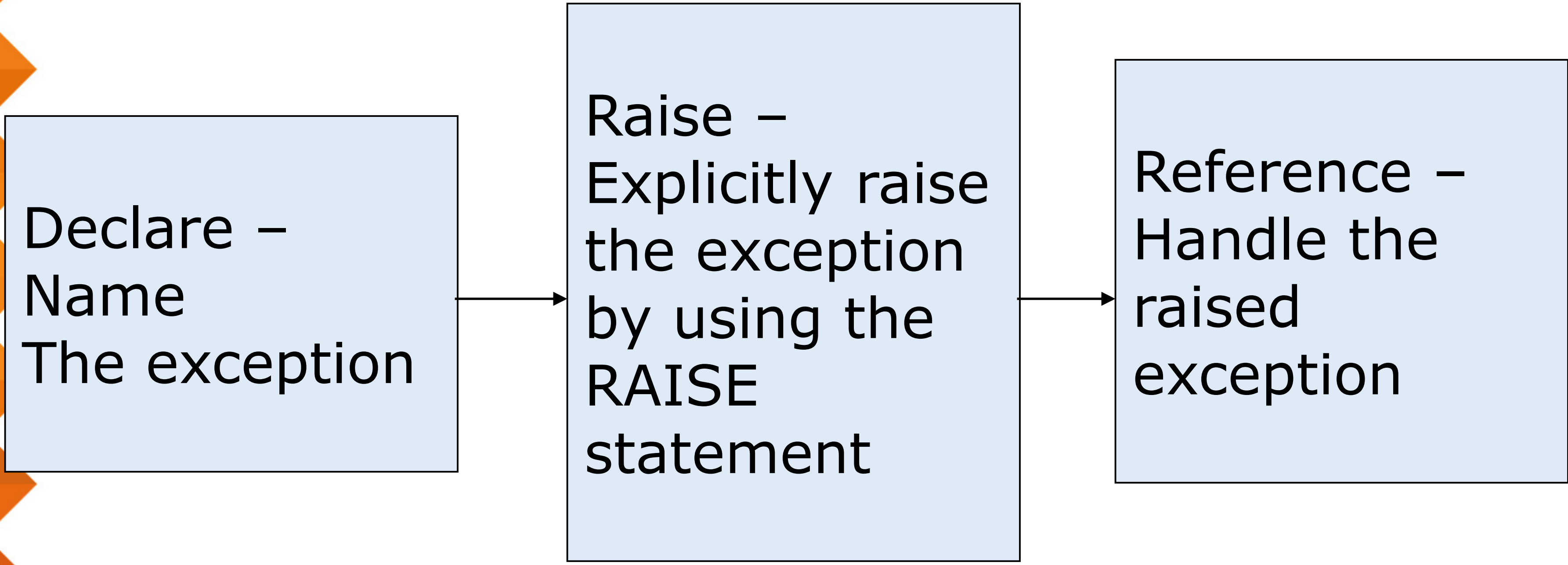
```
EXCEPTION
   WHEN NO_DATA_FOUND THEN
     dbms_output.put_line('No data!');
   WHEN TOO_MANY_ROWS THEN
     dbms_output.put_line('Too many!');
   WHEN OTHERS THEN
     dbms_output.put_line(‘Error,
                          closing program now’);
END;
```

Non-predefined Exceptions

- Non-predefined exception has an attached Oracle error code, but it is not named by Oracle.

- Such exceptions can be trapped with a WHEN OTHERS clause, or by **declaring them** with names.

User-Defined Exceptions



Declare –
Name
The exception

Raise –
Explicitly raise the exception by using the RAISE statement

Reference –
Handle the raised exception

User-defined Exception Example

```
DECLARE
    E_invalid_product EXCEPTION;
BEGIN
    UPDATE product
       SET descrip = ‘&product_description’
       WHERE prodid = &product_number;
    IF SQL%NOTFOUND THEN
          RAISE e_invalid_product;
    END IF;
    COMMIT;
EXCEPTION
    WHEN e_invalid_product THEN
          DBMS_OUTPUT.PUT_LINE (‘Invalid product number.’);
END;
```

Defining Your Own Error Messages

- Procedure RAISE_APPLICATION_ERROR
  - An application can call raise_application_error only from an executing stored subprogram
  - When called, raise_application_error ends the subprogram and returns a user-defined error number and message to the application
  - error_numbers should be a negative integer in the range -20000 .. -20999 and message is a character string up to 2048 bytes long
  - The error number and message can be trapped like any Oracle error

Procedure RAISE_APPLICATION_ERROR

- To call RAISE_APPLICATION_ERROR, use the syntax

  raise_application_error(error_number, message);

- For example:

```
…
        IF SaleQty > v_QOH THEN
                Raise_application_error(-20501,'Not enough

        stock on Hand');
        END IF;
```

Exercise

- Change the Debit_Account Procedure discussed earlier (slide 23) to include Exception Handling for an error of your choice.

Reference

- Oracle 11g PL/SQL User's Guide and Reference
- http://plsql-tutorial.com/plsql-variables.htm
- https://www.tutorialspoint.com/plsql/plsql_variable_types.htm