Database Management System Lab (CS-2094)

KALINGA INSTITUTE OF INDUSTRIAL TECHNOLOGY

School of Computer Engineering

Strictly for internal circulation (within KIIT) and reference only. Not for outside circulation without permission

1 Credit

Course Committee

Lab Contents

Sr #	Major and Detailed Coverage Area	Lab#
11	PL/SQL Programming Language ❑ Accessing a table ❑ Procedures ❑ Functions ❑ Exception	11

Assigning SQL Query Results to PL/SQL Variables

SELECT INTO statement of SQL can be used to assign values to PL/SQL variables. For each item in the SELECT list, there must be a corresponding, type-compatible variable in the INTO list. The following example illustrates the concept.

Table Creation

```
CREATE TABLE CUSTOMER
(
  ID   INT NOT NULL,
  NAME VARCHAR (20) NOT NULL,
  AGE INT NOT NULL,
  ADDRESS CHAR (25),
  SALARY   DECIMAL (18, 2),
  PRIMARY KEY (ID)
);
```

Data Insertion

```
INSERT INTO CUSTOMER (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00 );
INSERT INTO CUSTOMER (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (2, 'Khilan', 25, 'Delhi', 1500.00 );
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (3, 'kaushik', 23, 'Kota', 2000.00 );
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (4, 'Chaitali', 25, 'Mumbai', 6500.00 );
```

Assigning values from the above table to PL/SQL variables

```
DECLARE
  c_id customer.id%type := 1;
 c_name  customer.name%type;
  c_addr customer.address%type;
```

```
BEGIN
  SELECT name, address INTO c_name,
  c_addr FROM customer   WHERE id = c_id;
  dbms_output.put_line('Customer ' ||c_name || ' from ' || c_addr);
```

```
END;
/
```

Updating

Example 2:

```
DECLARE
  c_id customer.id%type := 1;
  c_sal  customer.salary%type;
BEGIN
  SELECT  salary INTO  c_sal
  FROM customer WHERE id = c_id;
  IF (c_sal <= 2000) THEN
    UPDATE customer
    SET salary =  salary + 1000
      WHERE id = c_id;
    dbms_output.put_line ('Salary updated');
  END IF;
END;
/
```

Subprogram

A **subprogram** is a program unit/module that performs a particular task. These subprograms are combined to form larger programs. This is basically called the 'Modular design'. A subprogram can be invoked by another subprogram or program which is called the **calling program**.

PL/SQL provides two kinds of subprograms:

- **Functions**: these subprograms return a single value, mainly used to compute and return a value.
- **Procedures**: these subprograms do not return a value directly, mainly used to perform an action.

Procedures and Functions are saved in the database as **database objects.**

Parts of a PL/SQL Subprogram

Like anonymous PL/SQL blocks and, the named blocks a subprograms will also have following three parts:

- Declarative Part
- Executable Part
- Exception-handling Part

Terminologies in PL/SQL Subprograms

Parameter:

The parameter is variable or placeholder of any valid PL/SQL datatype through which the PL/SQL subprogram exchange the values with the main code. This parameter allows to give input to the subprograms and to extract from these subprograms.

- These parameters should be defined along with the subprograms at the time of creation.
- These parameters are included in the calling statement of these subprograms to interact the values with the subprograms.
- The datatype of the parameter in the subprogram and in the calling statement should be same.
- The size of the datatype should not be mention at the time of parameter declaration, as the size is dynamic for this type.

Based on their purpose parameters are classified as: **IN**, **OUT** and **IN OUT**

Terminologies cont...

IN:

- This parameter is used for giving input to the subprograms.
- It is a read-only variable inside the subprograms, their values cannot be changed inside the subprogram.
- In the calling statement these parameters can be a variable or a literal value or an expression, for example, it could be the arithmetic expression like '5*8' or 'a/b' where 'a' and 'b' are variables.
- By default, the parameters are of IN type.

OUT:

- This parameter is used for getting output from the subprograms.
- It is a read-write variable inside the subprograms, their values can be changed inside the subprograms.
- In the calling statement, these parameters should always be a variable to hold the value from the current subprograms.

Terminologies cont...

IN OUT:

- This parameter is used for both giving input and for getting output from the subprograms.
- It is a read-write variable inside the subprograms, their values can be changed inside the subprograms.
- In the calling statement, these parameters should always be a variable to hold the value from the subprograms.

RETURN:

RETURN is the keyword that actually switch the control from the subprogram to the calling statement. In subprogram RETURN simply means that the control needs to exit from the subprogram. Once the RETURN keyword is encountered in the subprogram, the code after this will be skipped. Normally, parent or main block will call the subprograms, and then the control will shift from those parent block to the called subprograms. RETURN in the subprogram will return the control back to their parent block. In the case of functions RETURN statement also returns the value. The datatype of this value is always mentioned at the time of function declaration. The datatype can be of any valid PL/SQL data type.

Procedure

Syntax:

```
CREATE [OR REPLACE] PROCEDURE procedure_name [(parameter_name [IN | OUT | IN OUT] type [, ...])]
{IS | AS}
BEGIN
 < procedure_body >
EXCEPTION
  < exception handling part >
END;
```

Explanation:

- CREATE PROCEDURE is to create a new procedure. Keyword 'OR REPLACE' instructs is to replace the existing procedure (if any) with the current one.
- Procedure name should be unique.
- The optional parameter list contains name, mode and types of the parameters.
- Keyword '**IS**' will be used, when the procedure is nested into some other blocks. If the procedure is standalone then '**AS**' will be used.
- Procedure-body contains the executable part.
- Exception handling part contains the exception handling part.

Standalone Procedure Example

```
CREATE OR REPLACE PROCEDURE Greetings
AS
BEGIN
  dbms_output.put_line('Hello World!');
END;
/
```

Executing the Standalone Procedure

A standalone procedure can be called in two ways:

1. Using the EXECUTE keyword as shown below

   ```
   EXECUTE Greetings;
   ```

2. Calling the name of the procedure from a PL/SQL block as shown below

   ```
   BEGIN
     Greetings;
   END;
   /
   ```

Deleting Procedure

A standalone procedure is deleted with the DROP PROCEDURE statement. Syntax for deleting a procedure is: **DROP PROCEDURE procedure-name**;

So you can drop greetings procedure by using the following statement: **DROP PROCEDURE Greetings;**

IN & OUT mode example

```
DECLARE
  a,b,c number;
PROCEDURE findMin(x IN number, y IN number,
                              z OUT number)  IS
BEGIN
  IF x < y THEN
    z:= x;
  ELSE
    z:= y;
  END IF;
END;
```

```
/*continuation of program */
BEGIN
  a:= 23;
  b:= 45;
  findMin(a, b, c);
  dbms_output.put_line(' Minimum of (23, 45) : ' || c);
END;
/
```

Nested Procedure Example cont…

```
DECLARE
   a number;
PROCEDURE squareNum(x IN OUT number) IS
BEGIN
  x := x * x;
END;
BEGIN
   a:= 23;
   squareNum(a);
   dbms_output.put_line(' Square of (23): ' || a);
END;
/
```

Methods for Passing Parameters

Assume there is an procedure findMin(a, b, c, d). Actual parameters could be passed in three ways:

- **Positional notation:** the first actual parameter is substituted for the first formal parameter; the second actual parameter is substituted for the second formal parameter, and so on e.g. call the procedure as: findMin(m, n, o, p);
- **Named notation:** the actual parameter is associated with the formal parameter using the arrow symbol (=>). So the procedure call would look like: findMin(a=>m, b=>n, c=>o, d=>p);
- **Mixed notation:** In mixed notation, you can mix both notations in procedure call; however, the positional notation should precede the named notation.

  This call is legal: findMin(m, n, o, d=>p);

  But this is not legal: findMin(a=>x, b, c, d);

Function

A PL/SQL function is same as a procedure except that it returns a value. Therefore, all the discussions of the previous chapter are true for functions too.

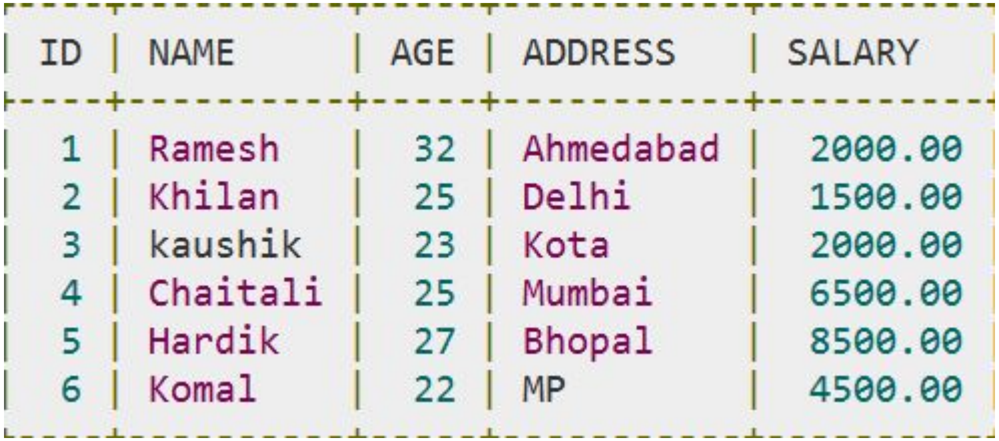
Creating Function Syntax:

```
CREATE [OR REPLACE] FUNCTION function_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
RETURN return_datatype
{IS | AS}
BEGIN
  < function_body >
EXCEPTION
  < exception handling part >
END;
```

Standalone Function Example

```
CREATE OR REPLACE FUNCTION GetTotalCustomers
RETURN number
IS
  total number(2) := 0;
BEGIN
  SELECT count(*) into total FROM customer;
  RETURN total;
END;
/
```

View structure with data



```
+----+----------+-----+-----------+----------+
| 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 |
+----+----------+-----+-----------+----------+
```

Calling a Function

To call a function you simply need to pass the required parameters along with function name and if function returns a value then you can store returned value. Following program calls the function **GetTotalCustomers** from an anonymous block:

```
DECLARE
  c number(2);
BEGIN
  c := GetTotalCustomers();
  dbms_output.put_line('Total no. of Customers: ' || c);
END;
/
```

Nested Function Example

```
DECLARE
  a,b,c number;
FUNCTION findMax(x IN number, y IN number)
RETURN number
IS
   z number;
BEGIN
  IF x > y THEN
    z:= x;
  ELSE
    Z:= y;
  END IF;

  RETURN z;
END;
```

```
/* continuation of program */
BEGIN
  a:= 23;
  b:= 45;

  c := findMax(a, b);
  dbms_output.put_line(' Maximum of (23,45): ' || c);
END;
/
```

PL/SQL Recursive Function

When a subprogram calls itself, it is referred to as a recursive call and the process is known as recursion. The following program calculates the factorial of a given number by calling itself recursively:

```
DECLARE
  num number;
  factorial number;

FUNCTION Fact(x number)
RETURN number
IS
  f number;
BEGIN
  IF x=0 THEN
    f := 1;
  ELSE
    f := x * Fact(x-1);
  END IF;
RETURN f;
END;
```

```
/* continuation  of program */
BEGIN
  num:= 6;
  factorial := Fact(num);
  dbms_output.put_line(' Factorial '|| num || ' is ' || factorial);
END;
/
```

Deleting Function

Once you have created your function, you might find that you need to remove it from the database.

Syntax: DROP FUNCTION function_name;

Example: DROP FUNCTION GetTotalCustomers;

DROP FUNCTION Fact;

Difference between Function and Procedure

Procedure	Function
Used mainly to execute certain process	Used mainly to perform some calculation
Use OUT parameter to return the value	Use RETURN to return the value
It is not mandatory to return the value	It is mandatory to return the value
RETURN will simply exit the control from subprogram.	RETURN will exit the control from subprogram and also returns the value
Return datatype is not specified at the time of creation	Return datatype is mandatory at the time of creation

Exception

An error condition during a program execution is called an **exception** in PL/SQL. PL/SQL supports programmers to catch such conditions using EXCEPTION block in the program and an appropriate action is taken against the error condition. There are two types of exceptions:

- System-defined exceptions
- User-defined exceptions

Syntax for Exception Handling

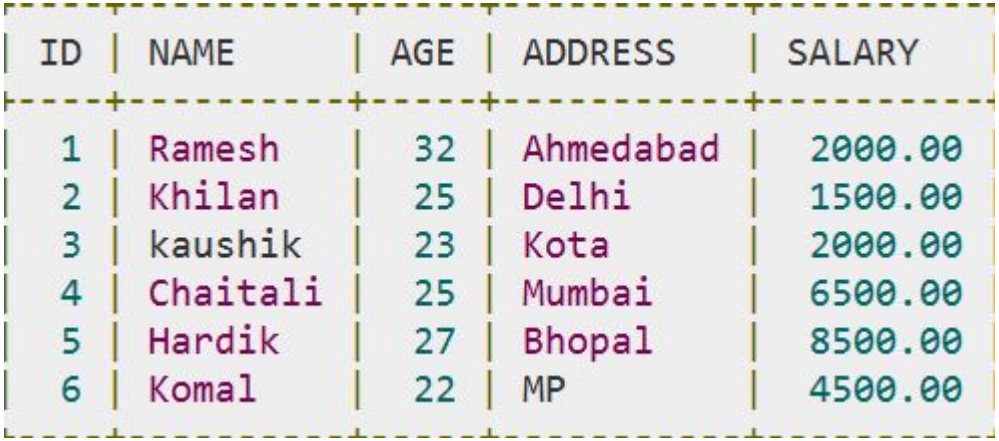
```
DECLARE
  <declarations section>
BEGIN
  <executable command(s)>
EXCEPTION
  <exception handling goes here >
  WHEN exception1 THEN
     exception1-handling-statements
/*continuation of program */
WHEN exception2  THEN
     exception2-handling-statements
  ........
  WHEN others THEN
     exception-handling-statements
END;
```

Exception Example

```
DECLARE
  c_id customer.id%type := 8;
  c_name  customer.name%type;
  c_addr customer.address%type;
BEGIN
  SELECT  name, address INTO  c_name, c_addr FROM customer WHERE id = c_id;
  DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
  DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
EXCEPTION
  WHEN no_data_found THEN
    dbms_output.put_line('No such customer!');
  WHEN others THEN
    dbms_output.put_line('Error!');
END;
/
```

Since there is no customer with ID value 8 in the table, the program raises the run-time exception NO_DATA_FOUND, which is captured in EXCEPTION block.

Customer



```
+----+----------+-----+-----------+---------+
| 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 |
+----+----------+-----+-----------+---------+
```

System-defined Exception

These exception are pre-defined and are automatically raised by Oracle whenever an exception is encountered. Each exception is assigned a unique number and a name.

Error Name	Error No	Description
ACCESS_INTO_NULL	ORA-06530	It is raised when a null object is automatically assigned a value.
CASE_NOT_FOUND	ORA-06592	It is raised when none of the choices in the WHEN clauses of a CASE statement is selected, and there is no ELSE clause.
ZERO_DIVIDE	ORA-01476	It is raised when an attempt is made to divide a number by zero.
TOO_MANY_ROWS	ORA-01422	It is raised when s SELECT INTO statement returns more than one row.
INVALID_NUMBER	ORA-01722	It is raised when the conversion of a character string into a number fails because the string does not represent a valid number.
NO_DATA_FOUND	ORA-01403	It is raised when a SELECT INTO statement returns no rows.
PROGRAM_ERROR	ORA-06504	It is raised when PL/SQL has an internal problem.

System-defined Exception Cont…

Error Name	Error No	Description
CURSOR_ALREADY_OPEN	ORA-06511	A program attempts to open an already open cursor. A cursor must be closed before it can be reopened. A cursor FOR loop automatically opens the cursor to which it refers, so your program cannot open that cursor inside the loop.
DUP_VAL_ON_INDEX	ORA-00001	A program attempts to store duplicate values in a column that is constrained by a unique index.
VALUE_ERROR	ORA-06502	An arithmetic, conversion, truncation, or size-constraint error occurs. For example, when your program selects a column value into a character variable, if the value is longer than the declared length of the variable, PL/SQL aborts the assignment and raises VALUE_ERROR. In procedural statements, VALUE_ERROR is raised if the conversion of a character string into a number fails. (In SQL statements, INVALID_NUMBER is raised.)

System-defined Exception Example

```
DECLARE
  stock_price NUMBER := 9.73;
  net_earnings NUMBER := 0;
  pe_ratio NUMBER;
BEGIN
  pe_ratio := stock_price / net_earnings; -- Calculation might cause division-by-zero error.
  DBMS_OUTPUT.PUT_LINE('Price/earnings ratio = ' || pe_ratio);
EXCEPTION  -- exception handlers begin
-- Only one of the WHEN blocks is executed.
  WHEN ZERO_DIVIDE THEN  -- handles 'division by zero' error
    DBMS_OUTPUT.PUT_LINE('Company must have had zero earnings.');
    pe_ratio := NULL;
  WHEN OTHERS THEN  -- handles all other errors
    DBMS_OUTPUT.PUT_LINE('Some other kind of error occurred.');
    pe_ratio := NULL;
END;  -- exception handlers and block end here
/
```

Raising Exceptions

Exceptions are raised by the database automatically whenever there is any internal database error, but exceptions can be raised explicitly by the programmer by using the command **RAISE**. Following is the simple syntax of raising an exception:

```
DECLARE
  exception_name EXCEPTION;
BEGIN
  IF condition THEN
    RAISE exception_name;
  END IF;
EXCEPTION
  WHEN exception_name THEN
  statement;
END;
```

User-defined Exception

PL/SQL allows you to define your own exceptions according to the need of your program. A user-defined exception must be declared and then raised explicitly, using RAISE statement.

Example

```
DECLARE
  c_id customer.id%type := &cc_id;
  c_name  customer.name%type;
  c_addr customer.address%type;
  -- user defined exception
  ex_invalid_id  EXCEPTION;
BEGIN
  IF c_id <= 0 THEN
    RAISE ex_invalid_id;
  ELSE
     SELECT  name, address INTO  c_name, c_addr FROM customer WHERE id =
c_id;
    DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
    DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
  END IF;
```

```
/*Continuation of program */
EXCEPTION
  WHEN ex_invalid_id THEN
    dbms_output.put_line('ID must be greater than zero!');
  WHEN no_data_found THEN
    dbms_output.put_line('No such customer!');
  WHEN others THEN
    dbms_output.put_line('Error!');
END;
/
```

Assigning name and error number to user-defined exception

A user-defined exception can be assigned a name and an error number by using PRAGMA pre-compiler directive. This directive binds the specified error number to a user-defined exception name. You can use more than one PRAGMA EXCEPTION_INIT directives. The syntax is:

exceptionname EXCEPTION;

PRAGMA EXCEPTION_INIT(exceptionname, errorcode);

Example

```
DECLARE
  vcomm Employee.comm%TYPE; veno Employee.empno%TYPE;
  Invalid_comm EXCEPTION;
  PRAGMA EXCEPTION_INIT(Invalid_comm, -20000);
BEGIN
   veno: =&veno;
   SELECT comm INTO vcomm FROM Employee WHERE empno=veno;
   IF vcomm<0 THEN
      RAISE Invalid_comm;
   ELSE
       DBMS_OUTPUT.PUT_LINE(vcomm);
END IF;
```

```
/*Continuation of program */
EXCEPTION
   WHEN Invalid_comm THEN
      DBMS_OUTPUT.PUT_LINE(SQLERRM||’ ’||’Negative commission);
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE(’No such id’);
END;
/
```

Guidelines for Avoiding and Handling PL/SQL Errors and Exceptions

Because reliability is crucial for database programs, use both error checking and exception handling to ensure your program can handle all possibilities:

- ❑ Add exception handlers whenever there is any possibility of an error occurring. Errors are especially likely during arithmetic calculations, string manipulation, and database operations. Errors could also occur at other times, for example if a hardware failure with disk storage or memory causes a problem that has nothing to do with your code; but your code still needs to take corrective action.
- ❑ Add error-checking code whenever you can predict that an error might occur if your code gets bad input data. Expect that at some time, your code will be passed incorrect or null parameters, that your queries will return no rows or more rows than you expect.
- ❑ Carefully consider whether each exception handler should commit the transaction, roll it back, or let it continue. Remember, no matter how severe the error is, you want to leave the database in a consistent state and avoid storing any bad data.
- ❑ Test your code with different combinations of bad data to see what potential errors arise.