PL/SQL (Procedural Language/SQL)

CSE-302 (Database Management System Sessional)

Introduction

- The development of database applications typically requires language constructs similar to those that can be found in programming languages such as C, C++, or Pascal.
- These constructs are necessary in order to implement complex data structures and algorithms.
- A major restriction of the database language SQL, however, is that many tasks cannot be accomplished by using only the provided language elements.

What is PL/SQL?

- PL/SQL stands for Procedural Language extension of SQL that offers language constructs similar to those in imperative programming languages.
- It is a combination of SQL along with the procedural features of programming languages
- It allows users and designers to develop complex database applications that require the usage of control structures and procedural elements such as procedures, functions, and modules.
- It was developed by Oracle Corporation in the early 90’s to enhance the capabilities of SQL.

Why do we need PL/SQL?

- Though SQL is a very powerful, set-oriented language, it cannot be used to implement all business logic and end-user functionality needed in our applications. That brings us to PL/SQL.
- Suppose you need to insert 10000 records every day from a file or need to check for valid data among 1000 records.

  Are you going to write 10000 lines of INSERT statements???? Or

  Are you going to check 10000 records one by one to find the valid data?

Advantages of PL/SQL

- Can include **error handling, exception handling** and control structures
- Can be **stored and used** by various application programs or users
- SQL statements are passed to Oracle engine one at a time which increases traffic and decreases speed. PL/SQL can **execute a number of queries** in one block using single command.
- Applications written in PL/SQL are **portable** to computer hardware or operating system where Oracle is operational.

Basic Structure of PL/SQL Block

PL/SQL Block consists of three sections:

- The Declaration section (optional)
- The Execution section (required)
- The Exception Handling (or Error) section (optional)

Every PL/SQL statement ends with a semicolon

PL/SQL Block: Syntax

```
[DECLARE]
    /* Variable declaration */
BEGIN
    /* Program Execution */
[EXCEPTION]
    /* Exception handling */
END;
```

Declarative Section

- Identified by **DECLARE** keyword.
- Used to define identifiers (**variables**, **constants, records as cursors, types, exceptions)** referenced in the block.
- Variable:
  - Reserve a temporary storage area in memory. (Used to store query results)
  - Manipulated without accessing a physical storage medium.
- Constant:
  - Its assigned value doesn't change during execution.
- Forward Execution:
  - Identifiers must be declared before they can be referenced.

Executable Section

- Identified by **BEGIN** keyword.
  - Mandatory
  - Can consists of several SQL and/or PL/SQL statements
  - End with **END** keyword
- Used to access & manipulate data within the block.

Exception-Handling Section

- Identified by **EXCEPTION** keyword.
- Used to display messages or identify other actions to be taken when an error occurs
- Addresses errors that occur during a statement's execution.

  Examples: No rows returned or divide by zero errors

END Keyword

- Used to close a PL/SQL block.
- Always followed by a semicolon.

A Simple Example of PL/SQL Block (1)

Print the classic "Hello World!" using PL/SQL

```
BEGIN
   dbms_output.put_line ('Hello World!');
END;
```

DBMS_OUTPUT.PUT_LINE procedure will write the passing string into the Oracle buffer. In order to print the content of the Oracle buffer into screen, use the **SET SERVEROUTPUT ON** command.

A Simple Example of PL/SQL Block (2)

```
DECLARE
      msg VARCHAR2 (100) := 'Hello  World!';
BEGIN
      DBMS_OUTPUT.put_line (msg);
END;
```

Output:
Hello World!

Example of PL/SQL Nested Block

```
DECLARE
        msg1 VARCHAR2 (100) := 'Hello';
BEGIN
  DECLARE
      msg2 VARCHAR2 (100) := msg1 || ' World!';
  BEGIN
      DBMS_OUTPUT.put_line (msg2);
  END;

END;
```

Running SQL inside PL/SQL Block

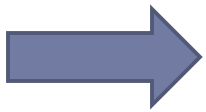
Suppose, we want to show the customer name from customer table where customer ID is ‘C_0000001’.

SQL:

SELECT cust_name from customer where cust_id=‘C_0000001’

PL/SQL:

```
DECLARE
c_name VARCHAR2 (50);
BEGIN
SELECT name INTO c_name  FROM customer
WHERE cust_id = 'C_0000001';
DBMS_OUTPUT.put_line (c_name);
END;
```



```
Tahrima Tusi

Statement processed.


0.01 seconds
```

Types of Block

Procedure:

These programs do not return a value directly; mainly used to perform an action.

Function:

These programs return a single value; mainly used to compute and return a value.

Anonymous block:

These programs have no names.

Procedure

- Also called “Stored Procedures”
- **Named block**
- Can process several variables
- **Returns no values**
- Interacts with application program using **IN, OUT** or **IN OUT parameters**
- Procedure is a **subprogram** that performs particular task
- It is created with the **CREATE PROCEDURE** and **stored** in the database
- It is **invoked** by another subprogram or block
- It can be deleted with the **DROP PROCEDURE**

Procedure: Syntax

```
CREATE [OR REPLACE]
PROCEDURE procedure_name
[( argument [IN|OUT|IN OUT]
datatype,
argument [IN|OUT|IN OUT]
datatype )]
 AS
    /* declaration section */
BEGIN
   /* executable section - required */
EXCEPTION
  /* error handling statements */
END;
```

```
CREATE OR REPLACE
PROCEDURE hello_msg
AS
msg varchar2(100):= 'Hello
World!';
BEGIN
dbms_output.put_line
(msg);
END;
```

Program Output:

Procedure created.

Executing a Procedure

A procedure can be called in two ways −

i. Using EXECUTE keyword

EXECUTE [Procedure Name];

or

EXECUTE [Procedure Name]([Parameter]);

Example: EXECUTE hello_msg;

ii. Calling the name of the procedure from a PL/SQL block

```
BEGIN
    hello_msg;
END;
```

Drop Procedure hello_msg;

Parameters Modes in PL/SQL Subprogram

IN:

- An IN parameter lets you pass a value to the subprogram.
- It is a read-only parameter.
- Inside the subprogram, an IN parameter acts like a constant. It cannot be assigned a value.
- You can pass a constant, literal, initialized variable, or expression as an IN parameter.
- You can also initialize it to a default value; however, in that case, it is omitted from the subprogram call.
- It is the default mode of parameter passing. That means if you don't specify the mode for a parameter explicitly, Oracle will use the IN mode.
- Parameters are passed by reference.

Parameters Modes in PL/SQL Subprogram

OUT:

- An OUT parameter returns a value to the calling program.
- Inside the subprogram, an OUT parameter acts like a variable.
- You can change its value and reference the value after assigning it.
- The actual parameter must be variable and it is passed by value.

IN OUT:

- An IN OUT parameter passes an initial value to a subprogram and returns an updated value to the caller. It can be assigned a value and its value can be read.
- The actual parameter corresponding to an IN OUT formal parameter must be a variable, not a constant or an expression.
- Actual parameter is passed by value.

Example of IN & OUT Mode

This program finds the minimum of two values, here procedure takes two numbers using IN mode and returns their minimum using OUT parameters.

```
CREATE PROCEDURE findMin(x IN number, y IN number, z OUT number)
AS
BEGIN
        IF x < y THEN
        z:= x;
        ELSE z:= y;
        END IF;
END;
```

```
DECLARE
a number;
b number;
c number;
 BEGIN
        a:= 23; b:= 45;
        findMin(a, b, c);
       dbms_output.put_line(' Minimum of (23, 45) : ' || c);
END;
```

Practice Problems

1. Write a procedure to compute the square of value of a passed value (using IN OUT parameter mode).
2. Write a procedure that shows the following output: [Use ‘||’ operator]

Hello MIST

Hello DBMS

Procedure Example

```
Create Procedure
Insert_Customer(customer_id IN varchar2)
AS
BEGIN
insert into customer values (customer_id,
   NULL,NULL,NULL,NULL);
END;
```

```
BEGIN
Insert_Customer('C_0000010');
END;
```

Function

- Named block that is stored on the server.
- REPLACE option allows the modification of an existing function.
- Accepts zero or more input parameters; parameter list contains name, mode (In, OUT) and types of the parameters.
- Must contain a **return** statement; Returns one value.
- **Basic Syntax:**

```
CREATE [OR REPLACE] FUNCTION function_name
[( argument IN/OUT datatype , argument IN/OUT datatype] )]
RETURN datatype
AS
/* declaration section */
BEGIN
/* executable section - required */
EXCEPTION
/* error handling statements */
END;
```

Function Example

```
CREATE OR REPLACE FUNCTION
get_email (customerid VARCHAR2)
RETURN VARCHAR2
AS
v_email customer.email%TYPE;
BEGIN
SELECT email INTO v_email FROM customer
WHERE cust_id = customerid;
RETURN v_email;
END;
```

Recursive Function Possible???

```
BEGIN
DBMS_OUTPUT.PUT_LINE(get_email('C_0000001'));
END;
```

```
tusi@gmail.com

Statement processed.

0.00 seconds
```

Task: Write a procedure that shows the Customer's Email id whose customer id is 'C_0000001' .

Anonymous Block

- Not stored since it cannot be referenced by a name.
- Usually embedded in an application program, stored in a script file, or manually entered when needed.

PL/SQL Variables

- Reserves a temporary storage area in the computer's memory
- Each variable must have
  - A name
  - A data type
  - Form is <variable> <data type>
- Variables can be initialized
- Variable name can consist of up to 30 characters, numbers, or special symbols
- Variable name must begin with a character

PL/SQL Variables Initialization

- Syntax:

  Variable_name data type [NOT NULL := value];

- NOT NULL is an optional specification on the variable.
- Use DEFAULT keyword or assignment operator (:=).
- Each variable declaration is a separate statement and must be terminated by a semicolon.
- When a variable is specified as NOT NULL, you **MUST** initialize the variable when it is declared.
- Non-numeric data types must be enclosed in single quotation marks.

PL/SQL Variables Example

DECLARE

salary number (6);

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

We can assign values to variables in the two ways:

- Assign values to variables: variable_name:= value;
- Assign values to variables directly from the database columns by using a SELECT.. INTO statement.

  SELECT *column_name*

  INTO *variable_name* FROM *table_name* [WHERE condition];

Example

Display the email id and phone number of a customer with id ‘C_0000002’

```
DECLARE
var_cust_id varchar2(20):='C_0000002';
var_email varchar2(50);
var_phone varchar2(20);
```

```
fahmida@gmail.com
01829431158
The customer C_0000002 has email fahmida@gmail.com and 01829431158
```

```
BEGIN
select email , phone into var_email,var_phone from customer where cust_id=var_cust_id;
dbms_output.put_line(var_email);
dbms_output.put_line(var_phone);
dbms_output.put_line('The customer ' || var_cust_id|| ' has email ' || var_email|| ' and '|| var_phone);
END;
```

Practice Problem

Print the account balance of a Cutomer with Cust_id 'C00000000005' and display it on the screen.

Scope of PL/SQL Variables

- PL/SQL allows the nesting of Blocks within Blocks.
- Based on their declaration we can classify variables into two types.

  Local variables - These are declared in a inner block and cannot be referenced by outside Blocks.

  Global variables - These are declared in a outer block and can be referenced by its itself and by its inner blocks.

Scope of PL/SQL Variables: Example

```
DECLARE
    var_num1 number (10);
    BEGIN
    var_num1 := 100;

        DECLARE
        var_mult number
          (10);
        BEGIN
        var_mult :=
          var_num1 * 100;
        END;
END;
```

```
DECLARE
var_num1 number(10);
BEGIN
    var_num1 := 100;
    DECLARE
    var_mult number (10);
    BEGIN
    var_mult := var_num1 * 100;
    END;
    var_mult := 900;
END;
```

ORA-06550: line 12, column 1: PLS-00201: identifier 'VAR_MULT' must be declared

PL/SQL Constants

constant_name CONSTANT data type := VALUE;

- ***constant_name*** is the name of the constant i.e. similar to a variable name.
- The word *CONSTANT* is a reserved word and ensures that the value does not change.
- ***VALUE*** - It is a value which must be assigned to a constant when it is declared. You cannot assign a value later.

PL/SQL Records

- A **record** is a data structure that can hold data items of different kinds.
- It's a **composite data types**, which means it is a combination of different scalar data types like **char, varchar, number** etc.
- Records consist of different fields, similar to a row of a database table.
- Records can be three types:
  - User Defined
  - Table Based
  - Cursor Based

User Defined Records

- User-defined record type allows user to define different record structures. These records consist of different fields.
- General Syntax:

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

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

User Defined Records: Example

```
DECLARE
    TYPE employee_type
    IS
    RECORD (
        employee_id number(5),
        employee_first_name  varchar2(25),
        employee_last_name  employee.last_name%type,
        employee_dept employee.dept%type) ;
        employee_rec1 employee_type;
        employee_rec2 employee_type;
```

Note:

- %TYPE Takes the data type from the column of a table
- %TYPE syntax: **<variable> <table>.<column>%TYPE**

Table Based Records

- If all the fields of a record are based on the columns of a table, we can declare the record as follows:

  record_name table_name%ROWTYPE;

  %ROWTYPE creates a record with fields for each column of the specified table

- The syntax is:

```
DECLARE
      variable_name data_type;
      row_variable table%ROWTYPE;
BEGIN
      SELECT column name1, column name2, …… INTO
      row_variable FROM table_name WHERE
      column_name =  variable_name;
END;
```

The variables are then accessed as: **row_variable.column name**

Table Based Records: Example

```
DECLARE
customer_rec customer%rowtype;
BEGIN
  SELECT * INTO customer_rec FROM customer WHERE
  Cust_id = 'C_0000003';
   dbms_output.put_line('Customer ID: ' || customer_rec.Cust_id);
   dbms_output.put_line('Customer Name: '||customer_rec.name);
END;
```

```
Customer ID: C_0000003
Customer Name: Khairul Mahbub

Statement processed.


0.08 seconds
```

PL/SQL Conditional Statement

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

Nested IF..ELSE:

```
IF condition1 THEN
      statement1;
ELSE
    IF condition2 THEN
          statement2;
    ELSIF condition3 THEN
          statement3;
    END IF;
END IF;
```

PL/SQL Conditional Statement

```
DECLARE
price_val number(10);
outp varchar2(100);

BEGIN
SELECT price into price_val from menu
    where menu_id='M_0000007';
IF price_val<1000 THEN
    outp:='Price value is less than 1000';
ELSIF price_val> 1500 THEN
    outp:='Price value is greater than 1500';
ELSE
    outp:='Price value is between 1000 to
    1500';
END IF;
dbms_output.put_line(outp);
END;
```

Nested IF..ELSE:

```
DECLARE
price_val number(10);
outp varchar2(100);
BEGIN
SELECT price into price_val from menu
    where menu_id='M_0000007';
IF  price_val<1000  THEN
    outp:='Price value is less than 1000';
ELSE
    IF  price_val> 1500 THEN
            outp:='Price value is greater than
    1500';
    ELSIF price_val =2000 THEN
            outp:='Price value is 2000';
    END IF;
END IF;
dbms_output.put_line(outp);

END;
```

PL/SQL Iterative Statement-Loop

There are three types of loop:

- Simple loop
- While loop
- For loop

Simple Loop

- Syntax:

```
LOOP
    Statements;
    EXIT [WHEN condition];
END LOOP;
```

- Initialize a variable before the loop body.
- Increment the variable in the loop.
- Use a EXIT WHEN statement to exit from the Loop.
- If you use a EXIT statement without WHEN condition, the statements in the loop is executed only once.

Example:

```
DECLARE
counter number(12):=1;
BEGIN
LOOP
dbms_output.put_line('The value of
    counter is '||counter);
counter:=counter+1;
EXIT WHEN counter=5;
END LOOP;
END;
```

```
The value of counter is 1
The value of counter is 2
The value of counter is 3
The value of counter is 4
```

While Loop

- Syntax:

WHILE <condition>

LOOP statements;

END LOOP;

- Initialize a variable before the loop body.
- Increment the variable in the loop.
- EXIT WHEN statement and EXIT statements can be used in while loops but it's not done often.

Example:

```
DECLARE
counter number(12):=1;
BEGIN
WHILE counter<=4
LOOP
dbms_output.put_line('The value of
    counter is '||counter);
counter:=counter+1;
END LOOP;
END;
```

```
The value of counter is 1
The value of counter is 2
The value of counter is 3
The value of counter is 4
```

For Loop

- Syntax:

FOR counter IN **start_val..end_val**

LOOP

statements;

END LOOP;

- The **counter** variable is **implicitly declared** in the declaration section, so it's not necessary to declare it explicitly.
- The counter variable is **incremented by 1** and does not need to be incremented explicitly.
- EXIT WHEN statement and EXIT statements can be used in for loops but it's not done often.

Example:

```
DECLARE
counter number(12):=1;
BEGIN
FOR counter IN 1..4
LOOP
dbms_output.put_line('The value of
    counter is '||counter);
END LOOP;
END;
```

```
The value of counter is 1
The value of counter is 2
The value of counter is 3
The value of counter is 4
```

Nested Loop

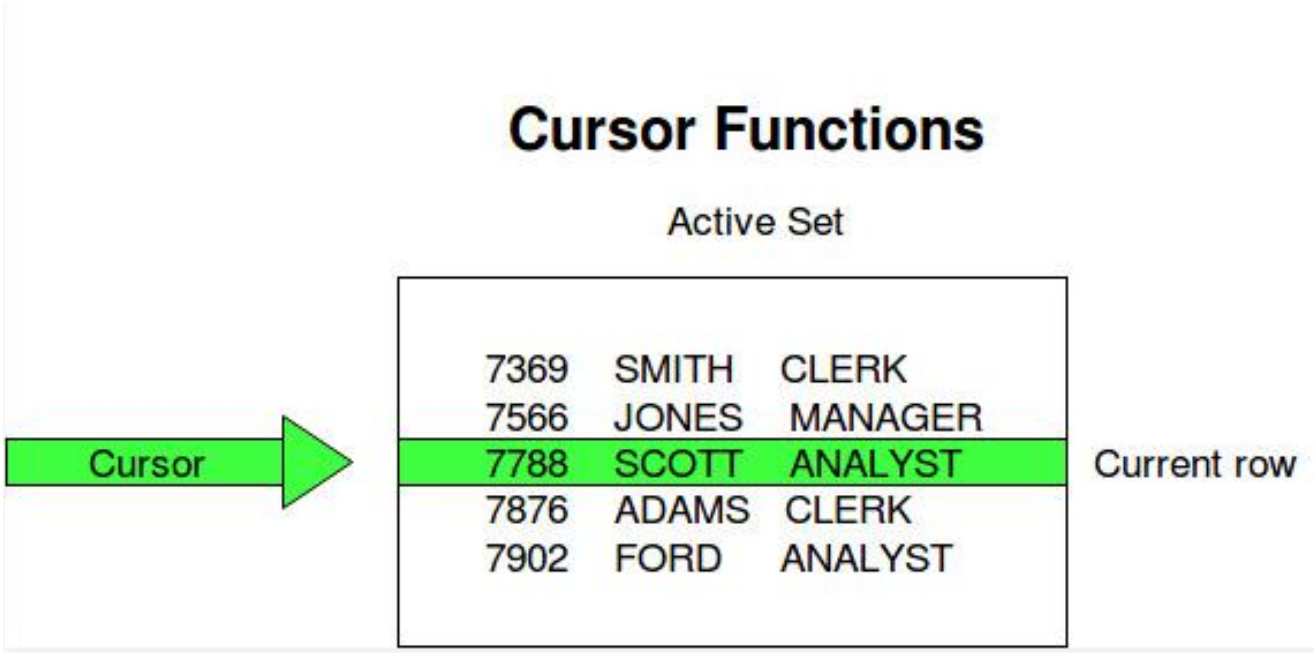
```
DECLARE
    loop_counter number(10) := 1;
BEGIN

    WHILE loop_counter <= 10
    LOOP
        dbms_output.put_line('The value of OUTER WHILE LOOP counter is '
          || loop_counter);
        FOR counter IN 1 .. 3
            LOOP dbms_output.put_line('The value of NESTED FOR LOOP
              counter is ' || counter);
        END LOOP;
        loop_counter := loop_counter + 1;
    END LOOP;

END
```

TRY IT!!!

Cursor

- A cursor is a pointer to the temporary work area created in the system memory when a SQL statement is executed and accessed the stored information.
- The major function of a cursor is to retrieve data, one row at a time, from a result set, unlike the SQL commands which operate on all the rows in the result set at one time.
- Cursors are used when the user needs to update records in a singleton fashion or in a row by row manner, in a database table.
- A cursor can hold more than one row, but can process only one row at a time. The set of rows (data) the cursor holds is called the *active* set.

Cursor


Cursor Functions
Active Set
7369 SMITH CLERK
7566 JONES MANAGER
7788 SCOTT ANALYST
7876 ADAMS CLERK
7902 FORD ANALYST
Cursor
Current row

Types of Cursors

- **Implicit Cursor:**
  - Automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement.
  - Programmers cannot control the implicit cursors and the information in it.
  - Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement.
  - For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.
- **Explicit Cursor:**
  - It is a program defined cursor and created on a SELECT Statement which returns more than one row.
  - Even though the cursor stores multiple records, **only one record can be processed at a time**, which is called as current row. When you fetch a row the current row position moves to next row.

Cursor Properties

%FOUND, %NOTFOUND: a record can/cannot be fetched from the cursor

%ROWCOUNT: the number of rows fetched from the cursor so far

%ISOPEN: the cursor has been opened

Cursor Properties

Attributes	Return Values	Example
%FOUND	Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.	SQL%FOUND
%NOTFOUND	The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.	SQL%NOTFOUND
%ROWCOUNT	Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.	SQL%ROWCOUNT
%ISOPEN	Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.	SQL%ISOPEN

Cursor (Explicit) Actions

- Cursors are defined and manipulated using-
  - Declare: For initializing the memory
  - Open: For allocating the memory
  - Fetch: For retrieving the data
  - Close: For releasing the allocated memory

Declaring Cursor

- Cursor name-similar to a pointer variable.
- There is no INTO clause.
- Syntax:

  CURSOR <cursor name>IS

  <select-expression>;

- Example:

  CURSOR emp_cursor IS

  SELECT emp_id, name from employee where name LIKE 'A%';

Opening a Cursor

- Opening the cursor allocates the memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it.
- Open cursor must be closed.
- Syntax:

  OPEN <cursor name>;

- Example:

  OPEN emp_cursor;

** You must use the same cursor name if you want data from that cursor

Fetching a Cursor

- Fetching the cursor involves accessing one row at a time.
- Syntax:

  FETCH <cursor name>

  INTO<host variables>;

- Example:

  FETCH emp_cursor

  INTO e_id,e_name;

Closing the Cursor

- Closing the cursor means releasing the allocated memory.
- Reopening the same cursor will reset it to point to the beginning of the returned table.
- Syntax:

  CLOSE <cursor name>;

- Example:

  CLOSE emp_cursor ;

Example of Cursor

```
DECLARE
CURSOR emp_cursor IS
SELECT emp_id, name FROM employeeB WHERE name LIKE
   'A%';
emp_val emp_cursor%ROWTYPE;
BEGIN
OPEN emp_cursor;
FETCH emp_cursor INTO emp_val;
DBMS_OUTPUT.PUT_LINE(emp_val.emp_id);
CLOSE emp_cursor;
END;
```

```
2011004

Statement processed.


0.07 seconds
```

Example of Implicit Cursor

```
DECLARE
var_rows number(5);
BEGIN
UPDATE menu
SET price = price + 10;
IF SQL%NOTFOUND THEN
dbms_output.put_line('None of the prices where updated');
ELSIF SQL%FOUND THEN
var_rows := SQL%ROWCOUNT;
dbms_output.put_line('Prices for ' || var_rows || ' foods are updated');
END IF;
END;
```

Example of Explicit Cursor

Simple Loop

```
DECLARE
CURSOR menu_cursor IS
SELECT menu_name, price FROM menu
    WHERE menu_name LIKE 'B%';
menu_val menu_cursor%ROWTYPE;
BEGIN
OPEN menu_cursor;
LOOP
FETCH menu_cursor INTO menu_val;
EXIT WHEN menu_cursor%NOTFOUND;
DBMS_OUTPUT.PUT_LINE(menu_val.menu
    _name);
END LOOP;
CLOSE menu_cursor;
END;
```

For Loop

```
DECLARE
CURSOR menu_cursor  IS
SELECT menu_name, price FROM menu
    WHERE menu_name LIKE 'B%';
BEGIN
FOR menu_val IN menu_cursor
LOOP
EXIT WHEN
    menu_cursor%NOTFOUND;
DBMS_OUTPUT.PUT_LINE(menu_val.
    menu_name);
END LOOP;
END;
```

```
Blueberry Cheese Cake
Brownie
Beef Burger with Cheese
```

Exception Handling

- A PL/SQL block may contain statements that specify exception handling routines.
- Each error or warning during the execution of a PL/SQL block raises an exception.
- Two types of exceptions:
  - System Defined: Automatically raised whenever corresponding errors or warnings occur.
  - User Defined: Must be raised explicitly in a sequence of statements using raise <exception name>.
- Syntax:

```
DECLARE
    //Declaration section
BEGIN
    //Execution section
EXCEPTION
    WHEN ex_name1
            THEN -Error handling statements
    ……
    WHEN Others
            THEN -Error handling statements
    END;
```

Exception Handling

- When an exception is raised, Oracle searches for an appropriate exception handler in the exception section.
- For example in the above example, if the error raised is '**ex_name1** ', then the error is handled according to the statements under it.
- Since, it is not possible to determine all the possible runtime errors during testing of the code, the 'WHEN Others' exception is used to manage the exceptions that are not explicitly handled.
- Only one exception can be raised in a Block and the control does not return to the Execution Section after the error is handled.

System Exceptions

Exception Name	Reason	Error Number
CURSOR_ALREADY_OPEN	When you open a cursor that is already open.	ORA-06511
INVALID_CURSOR	When you perform an invalid operation on a cursor like closing a cursor, fetch data from a cursor that is not opened.	ORA-01001
NO_DATA_FOUND	When a SELECT...INTO clause does not return any row from a table.	ORA-01403
TOO_MANY_ROWS	When you SELECT or fetch more than one row into a record or variable.	ORA-01422
ZERO_DIVIDE	When you attempt to divide a number by zero.	ORA-01476

Example of Exception Handling

```
DECLARE
employee_sal NUMBER(10,3); employee_id VARCHAR2(12);
too_high_sal exception;
BEGIN
    SELECT emp_id, salary into employee_id, employee_sal from employeeB where
    name='Shahriar Nazim';
IF employee_sal * 1.05 > 2500 THEN
    raise too_high_sal;
END IF;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
            DBMS_OUTPUT.PUT_LINE('No data found');
    WHEN too_high_sal THEN
            DBMS_OUTPUT.PUT_LINE('High Salary');
END;
```

```
High Salary

Statement processed.


0.00 seconds
```

Raise Application Error

- It is also possible to use procedure raise_application_error().
- This procedure has two parameters <error number> and <message text>.
- <error number> is a negative integer defined by the user and must range between -20000 and -20999.
- <error message> is a string with a length up to 2048 characters.

- If the procedure raise application error is called from a PL/SQL block, processing the PL/SQL block terminates and all database modifications are undone, that is, an implicit rollback is performed in addition to displaying the error message.

Example of Raise Application Error

```
DECLARE
employee_sal NUMBER(10,3);
employee_id VARCHAR2(12);
BEGIN
SELECT emp_id, salary into employee_id, employee_sal from
    employeeB where name='Shahriar Nazim';
IF employee_sal * 1.05 > 2500 THEN
raise_application_error(-20010,'Salary is too high');
END IF;
END;
```

```
ORA-20010: Salary is too high
```

"Be healthy and stay safe"