1. What is PL/SQL? Write advantages of PL/SQL over SQL

- Oracle programming language – SQL , provides various functionalities required to manage a database.
- SQL is so much powerful in handling data and various database objects.
- SQL does not provide basic procedural capabilities
- In SQL, it is not possible to control execution of SQL statements based on some condition or user inputs.
- Oracle provides **PL/SQL** (**Procedural Language / Structured Query Language**) to overcome disadvantages of SQL.
- PL/SQL is super set of SQL.
- PL/SQL supports all the functionalities provided by SQL along with its own **procedural capabilities**.

Advantages of PL/SQL

1) **Procedural Capabilities:**
   - PL/SQL provides procedural capabilities such as **condition checking, branching and looping**.
   - This enables programmer to control execution of a program based on some conditions and user inputs.
2) **Support to variables:**
   - PL/SQL supports declaration and use of variables.
3) **Error Handling:**
   - When an error occurs, user friendly message can be displayed.
   - Also, execution of program can be controlled instead of abruptly terminating the program.
4) **User Defined Functions:**
   - Along with a large set of in-build functions, PL/SQL also supports **user defined functions** and **procedures**.
5) **Portability:**
   - It means, programs can be **transferred and executed** from any other computer hardware and operating system, where Oracle is operational.
6) **Sharing of Code:**
   - This code can be **accessed and shared by different applications**.
   - This code can be executed by other programming language like JAVA.
7) **Efficient Execution:**
   - PL/SQL sends an entire block of SQL statements to the Oracle engine and executed in one go.
   - This reduces **network traffic and improves efficiency** of execution.
   - In case of **SQL**, all statements are transferred **one by one**.

2. Describe the structure of generic PL/SQL block OR Draw structure of PL/SQL block. Explain purpose of each section of PL/SQL block.

- PL/SQL code is grouped into structures called **block**.
- A block is called a **named block**, if it is given particular name to identify.
- A block is called an **anonymous block**, if it is not given any name.

- **Named blocks** are created while creating **database objects** such as function, procedure, package and trigger.

- A block of PL/SQL code contains three sections given as below:
    1) **Declarations**
    2) **Executable Commands**
    3) **Exception Handling**

1) **Declarations:**
    - This section starts with the keyword '**DECLARE'**.
    - It defines and initializes **variables** and **cursors** used in the block.
    - This section is **optional section**

2) **Executable Commands:**
    - This section starts with the keyword '**BEGIN'**.
    - This is the only **mandatory section** in the PL/SQL block.
    - It contains various SQL and PL/SQL statements providing functionalities like **data retrieval, manipulation, looping and branching**.

3) **Exception Handling:**
    - This section starts with the keyword '**EXCEPTION'**.
    - This section handles errors that arise in '**executable commands**' section.
    - This section is **optional section**.

- **The structure of a typical PL/SQL block can be given as**

```
below: DECLARE                  -- Optional
            <Declaration Section>
BEGIN                           -- Mandatory
            <Executable Commands>
EXCEPTION                       -- Optional
            <Exception Handling>
END ;                           -- Mandatory
```

- Notice that **DECLARE** and **EXCEPTION** are **optional** while **BEGIN** and **END** are **mandatory**.
- Also, **' ; '** at the end to **terminate the block**.

3. List out PL/SQL data types and explain any two in detail. OR Write short note: PL/SQL data types

- PL/SQL is super set of the SQL. So, it supports all the data types provided by SQL.
- Along with this, in PL/SQL Oracle provides **subtypes of the data types**.
- **For example**, the data type **NUMBER** has a subtype called **INTEGER**.
- These subtypes can be used in PL/SQL block to make the data type compatible with the data types of the other programming languages.

- **The various data types can be given as below:**

Category	Data Type	Sub types/values
Numerical	NUMBER	BINARY_INTEGER, DEC, DECIMAL, DOUBLE PRECISION, FLOAT, INTEGER, INT, NATURAL, POSITIVE, REAL, SMALLINT
Character	CHAR, LONG, VARCHAR2	CHARACTER,VARCHAR, STRING, NCHAR, NVARCHAR2
Date	DATE	
Binary	RAW, LONG RAW	
Boolean	BOOLEAN	Can have value like TRUE, FALSE and NULL.
RowID	ROWID	Stores values of address of each record.

4. Explain Variables. What are various ways to assign values to the variables?

- Variables are used to store values that may be change during the execution of program.
- In PL/SQL, variables contain values resulting from queries or expression.
- Variables are declared in Declaration section of the PL/SQL block.
- It must be assign valid data type and can also be initialized if necessary.

Declaring a Variable

Syntax:

variableName datatype [NOT NULL] := initialValue**;**

- **datatype** can be any valid data type supported by PL/SQL.
- **':= '** used for assignment operation.
- If variable need to be initialized then **initialValue** can be assigned at declaration time.
- If **NOT NULL** is included in declaration, variable cannot have **NULL** value during program execution and such variable must be initialized.

Example 1 : Some of the valid variable declaration are given

```
below: no      NUMBER(3);
value   DECIMAL;
city    CHAR(10);
name    VARCHAR(10);
counter         NUMBER(2)    NOT NULL :=  0
```

Assigning a Value

- There are three ways to assign value to a variable as given below:

1) **By using Assignment Operator:**

   Syntax:

   variableName := value **;**

   - A **value** can be a constant value or result of some expression or return value of some function.

2) **By Reading from the Keyword:**

   Syntax:

   variableName := &variableName **;**

   - This is similar to **scanf()** function. Whenever **'&'** is encountered, a value read from the keyboard and assign it to variable.
   - **For example**, following statement assigns value read from the keyboard to variable '**no**'.

   no := &no;

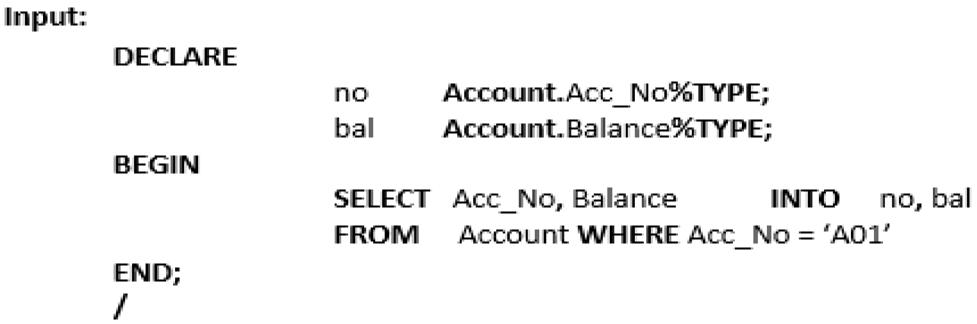
3) **Selecting or Fetching table data values into Variables:**

Syntax:

SELECT col1, col2, …, colN **INTO** var1, var2, …, varN
FROM tableName **WHERE** condition **;**

- This statement retrieves values for specified column and stores them in given variable.
- **Data type** and **size of variables** must be compatible with the relative columns.
- A condition in **WHERE** clause must be such that it selects **only single record**. This statement cannot work if multiple records are selected.

Example 3 : Following PL/SQL block stores account number and balance for account 'A01' into variable 'no' and 'bal'.



```
Input:
        DECLARE
                        no      Account.Acc_No%TYPE;
                        bal     Account.Balance%TYPE;
        BEGIN
                        SELECT  Acc_No, Balance        INTO    no, bal
                        FROM    Account WHERE Acc_No = 'A01'
        END;
        /
```

5. Explain anchored data type with example.

- A variable can be declared as having **anchored data type.** It means**, datatype** for variable is determined based on the data type of the **other object.**
- This object can be other **variable or a column** of the table.
- This provides ability to match the data types of the variables with the data types of the columns defined in the database.
- If data type of column is changed, then the data type of variable will also **changed automatically**.

Advantage: This reduces **maintenance cost** and allow a program to **adapt changes** made in tables.

Syntax:

variableName object%TYPE [NOT NULL] := initialValue **;**

- **object** can be any variable declared previously or column of a database.
- To refer of a column of particular table, column name must be combined with table name, as describe in below **example**.

Example : Some of the valid anchored variable declaration.

```
no      Account.Acc_No%TYPE;
bal     Account.Balance%TYPE;
name    Customer.name%TYPE;
```

6. Declaring a Constant

- A constant is also used to store value like a variable.
- But, unlike variable, a value stored in constant **cannot be changed** during program execution.

Syntax:

```
constantName  CONSTANT          datatype   := initialValue ;
```

- A constant must be initialized at declaration time.
- For example, following statement declared a constant named '**pi**'.

```
pi        CONSTANT      NUMBER(3,2)   :=    3.14 ;
```

7. Displaying Messages

- To display messages or any output on the screen in PL/SQL, following statement is used.

Syntax:

```
dbms_output.put_line ( message );
```

- A **dbms_output** is a package, which provides functions to **accumulate information** in a buffer.
- A **put_line** is a function, which display messages on the screen.
- A **message** is a character string to be displayed.
- To display data of other data type, they must be concatenated with some character string.
- The environment parameter, - **SERVEROUTPUT** - must be **ON** to display messages on screen.

Example 4 : Following statements display various outputs on the screen.

```
dbms_output.put_line ( 'Hi  Hello World…' );            Hi Hello World…
dbms_output.put_line ( 'Sum = ' || 25 );                 Sum = 25
dbms_output.put_line ( 'PI = ' || pi );                  PI =  3.14
dbms_output.put_line ( 'Square of ' || 3 || ' is ' || 9 );   Square of 3 is 9
```

8. Comments

- Comments are statement that will not get executed even though they are present in the program code.
- Comments are used to increase readability of a program.
- **In PL/SQL, a comment has two forms:**

1) -- (Double hyphen or double dash) (Single Line Comment):

- Treats single line as a comment.

```
-- This single line is a comment.
```

2) /* …. */ (Multiple Line Comment):

- Treats multiple lines as comment.

```
/*      This statement is spread over two line and
        both lines are treated as comments          */
```

9. Creating and Executing a PL/SQL Block

- **To create and execute a PL/SQL block, follow the steps given below:**
  - Open any editor like as notepad. An **EDIT** command can be used on **SQL** prompt to open a notepad from the **SQL * PLUS** environment.
  - The following syntax creates and opens a file:

```
EDIT    filename
```

Example:

```
EDIT    D:/PLSQL/test.sql
```

- Create and open a file named **'test.sql'**.
- Write a program code or statements in a file and save it.
- File should have **'.sql'** extension and last statement in file should be **'/'.**
- To **execute** this block, use any of the following commands on prompt.

```
RUN     fileName
START   fileName        OR
@       fileName
```

- **Example:** Following command executes a block saved in file 'test.sql' created.

```
@       D:/PLSQL/test.sql
```

10. Control Structures

- In PL/SQL , the flow of execution can be controlled in three different manners as given below:
  1) **Conditional Control**
  2) **Iterative Control**
  3) **Sequential Control**

- For various example, Account table that is given below is used:

Account

Acc_No	Balance	B_Name
A01	1000	Rjt
A02	4000	Ahmd
A03	3000	Srt
A04	5000	Brd

1) Conditional Control:

- To control the execution of block of code based on some condition, PL/SQL provides the **IF** statement.
- The **IF – THEN – ELSEIF – ELSE – END IF** construct can be used to execute specific part of the block based on the condition provided.

Syntax:

```
IF       condition       THEN
         -- Execute commands
ELSE IF  condition       THEN
         -- Execute command
         .

         .
ELSE     -- Execute command
END IF;
```

Example : Write a program to read a number from user and determine whether it is odd or even.

Input:

```
DECLARE
        no      NUMBER;                 -- declare a variable to store number
BEGIN
        -- read a number from the user.
        dbms_output.put_line ( 'Enter value for no: ' );
        no := &no;
        -- Check result of MOD function
        IF      MOD (no, 2) = 0THEN
                dbms_output.put_line ( 'Given Number '|| no || ' is EVEN.' );
        ELSE
                dbms_output.put_line ( 'Given Number '|| no || ' is ODD.' );
        END IF ;
END ;
/
```

Output:

```
Enter value for no: 7
Old 5: no := &no;
New 5: no := 7;
Given Number 7 is ODD.
```

- Observe the output. A message is displayed automatically to enter value for variable suffixed with '&'.
- It is also displays the old and new values for that variable. And at the end final message is displayed, whether number is even or odd.
- Also do not forget to set **SERVEROUTPUT** on.

Example : Write a program to debit a given account. Read account number and amount to be debited. Debit the balance if the resulting balance is not less than zero. This means, a balance in account should not go to negative while withdrawing amount.

Input:

```
DECLARE
        -- declare required variables
        no      Account.Acc_No%TYPE;
        bal     Account.Balance%TYPE;
        newBalance      Account.Balance%TYPE;
        amount          NUMBER(7,2);
BEGIN
        -- read account number and amount to be debited
                no := &no;
                amount := &amount;
```

```
                    -- retrieve the current balance for given account
                            SELECT Balance INTO bal
                            FROM    Account Where Acc_No = no ;
                    -- calculate a new balance
                            newBalance := bal – amount ;
                    -- Update balance if new balance zero or positive
                            IF newBalance >= 0 THEN
                                    UPDATE   Account    SET    Balance =  newBalance
                                    WHERE    Acc_No  =  no ;
                                    dbms_output.put_line('Account Debited Successfully..');
                            ELSE
                                    dbms_output.put_line('Not Sufficient Balance..');
                            END IF ;
            END ;
            /
```

Output 1 :

```
Enter value for no : A01
Enter value for amount : 3000
Not Sufficient balance..
PL/SQL procedure successfully completed.
```

Output 2 :

```
Enter value for no : A03
Enter value for amount : 2000
Account Debited Successfully..
PL/SQL procedure successfully completed.
```

2) Iterative Control:

- Iterative control allows a group of statements to execute **repeatedly** in a program. It is called **Looping**.
- PL/SQL provides three constructs to implement loops, as listed below:
    1. **LOOP**
    2. **WHILE**
    3. **FOR**
- In **PL/SQL**, any loop starts with a **LOOP** keyword and it terminates with an **END LOOP** keyword.
- Each loop requires a **conditional statement** to control the number of times a loop is executed.

1. LOOP

Syntax:

```
LOOP
        -- Execute commands..
END LOOP
```

- **LOOP** is an infinite loop. It executes commands in its body infinite times.
- So, it requires an **EXIT** statement within its body to **terminate the loop** after executing specific iteration.

Example : Display number from 1 to 5 along with their square values using LOOP construct.

Input:

```
DECLARE
        -- declare required variable
                counter   NUMBER(3)    :=    1 ;
BEGIN
        -- display headers for output
                dbms_output.put_line (‘Value     ’ || ‘  Square’);
        -- Traverse loop
                LOOP
                        EXIT WHEN counter > 5;
                        dbms_output.put_line ( ‘ ’ || counter || ‘ ’ || counter*counter);
                        counter := counter + 1;
                END LOOP;

END ;
/
```

Output:

```
Value           Square
  1               1
  2               4
  3               9
  4               16
  5               25
PL/SQL procedure successfully completed.
```

- Instead of using **“EXIT WHEN counter > 5;”** we can also use following **IF .. END IF** block.

```
IF      counter > 5  THEN
        EXIT;
END IF;
```

2. WHILE

Syntax:

```
WHILE    Condition
LOOP
        -- Execute Commands..
END LOOP;
```

- The **WHILE** loop executes commands in its body as long as the condition remains **TRUE**.
- The loop **terminates** when the condition evaluates to **FALSE** or **NULL**.
- The **EXIT** statement can also be used to exit the loop.

Example : Display numbers from 1 to 5 along with their square values using WHILE construct.

Input:

```
DECLARE
        -- declare required variable
                counter    NUMBER(3)   := 1;
BEGIN
        -- display headers for output
                dbms_output.put_line (‘    Value    ’ || ‘     Square’);
        -- traverse loop
            WHILE counter <= 5

                LOOP
                        dbms_output.put_line ( ‘ ’ || counter || ‘ ’ || counter*counter);
                        counter := counter + 1;
                END LOOP;
END;
/
```

Output:

Same as previous example 7.

3. **FOR**

Syntax:

```
FOR     variable   IN     [ REVERSE ]      start … end
LOOP
        -- Execute command
END LOOP
```

- Here, a **variable** is a loop control variable. It is declared implicitly by PL/SQL.
- So, it should not be **declared explicitly**.
- The **FOR LOOP** variable is always **incremented by 1** and any other increment value cannot be specified.
- A **start** and **end** specifies **the lower and upper bound** for the loop control variable.
- If **REVERSE** keyword is provided, loop is executed in reverse order (**From end to start**).

Example : Display numbers from 1 to 5 along with their square values using FOR construct.

Input:

```
DECLARE
        -- declare required variable
                counter   NUMBER(3)   := 1;
BEGIN
        -- display header for output
                dbms_output.put_line (‘   Value    ’ || ‘    Square’);
        -- traverse loop
                FOR   counter   IN      1 … 5
                LOOP
                        dbms_output.put_line ( ‘ ’ || counter || ‘ ’ || counter*counter);
                END LOOP;

END;
/
```

Output:

Same as previous example 7.

3) Sequential Control

- Normally, execution proceeds sequentially within the block of code.
- Sequence can be changed conditionally as well as unconditionally.
- To alter the sequence **unconditionally**, the **GOTO** statement can be used.

Syntax:

```
GOTO      jumpHere;
            :

            :
<<  jumpHere >>
```

- The **GOTO** statement makes flow of execution to jump at **<< jumpHere >>**.
- The jump is unconditional.

Example : The following code illustrates the use of the GOTO statement.

Input:

```
BEGIN
        dbms_output.put_line ('Code Starts.');
        dbms_output.put_line ('Before GOTO statement..');
        GOTO    jump;
                dbms_output.put_line ('This statement will not get executed..');
        << jump >>
        dbms_output.put_line ('Flow of execution jumped here..');
END;
/
```

Output:

```
Code Starts.
Before GOTO Statement..
Flow of execution jumped here..
```

- Here, **third put_line** statement did not execute. Because **GOTO** statement, which made the flow of execution to jump at **<< jump >>** and executed **forth put_line** statement.

11. Transactional Control

- Transactional control commands such as **COMMIT**, **ROLLBACK** and **SAVEPOINT** can also be used with PL/SQL code block to control the transaction.

Example: Debit the given account with specified amount. If resultant balance is negative, rollback the operation. Else, commit the transaction.

Input:

```
DECLARE
        -- declare required variables
                no        Account.Acc_No%TYPE;
                bal       Account.Balance%TYPE;
                amount  NUMBER(7,2);
BEGIN
        -- read account number and amount to be debited
                no := &no;
                amount := &amount;
        -- create savepoint
                SAVEPOINT      negativeBalance;
        -- update balance
                UPDATE    Account SET Balance = Balance – amount
```

```
                    WHERE Acc_No = no;
    -- read the new balance
                SELECT  Balance INTO bal FROM  Account
                WHERE Acc_No = no;
    -- display updated balance
                dbms_output.put_line(' Updated balance is '|| bal || ' …');

                -- if balance is negative then undo the debit operation
                        IF bal < 0 THEN
                                dbms_output.put_line(' Debit operation rollback. ');
                                ROLLBACK TO SAVEPOINT       negativeBalance;
                        ELSE
                                dbms_output.put_line(' Debit operation Committed. ');
                                COMMIT;
                        END IF ;
    END ;
    /
```

- Above given program reads an account number and amount to be debited from the user.
- Before updating the balance, a **savepoint** is created.
- Once a balance is **updated**, new balance is **retrieved** and **checked**.
- If it is **negative**, update operation is **rollback** otherwise the change is **saved permanently** using **COMMIT**

12. Explain procedure in detail with suitable example. OR Write short-note on Stored Procedures. OR Explain how to create and execute Stored Procedure?

- A **procedure** or **function** is a group of PL/SQL statements that performs specific task.
- A procedure and function is a **named PL/SQL block of code**. This block can be compiled and successfully compiled block can be stored in Oracle database. This procedure and function is called **Stores Procedure or Function**.
- We can pass parameters to procedures and functions. So that their execution can be changed dynamically.

Difference between Procedures and Functions are given below:

Functions	Procedures
○ A function **must** return a value.	○ A procedure **may or may** not return a value.
○ A function can return **only one value**.	○ A procedure can return **multiple values**.
○ A function can be used with **SELECT** statement, like in-built SQL functions.	○ A procedure **cannot** be used with SELECT statement.
○ A function cannot directly execute using **EXEC** command.	○ A procedure can directly execute using **EXEC** command.

- **Advantages of procedures and functions:**

1) Security:

- We can improve security by giving rights (privilege) to selected person to execute procedures and functions.

2) **Faster Execution:**
   - Code of procedures and functions are already compiled and no need to compile it at run time. So, require less time to execute.
3) **Sharing of code:**
   - Once procedure is created and stored, it can be used by more than one user.
   - This requires allocating memory to store code only once rather than allocating memory for multiple copies.
   - This utilized **memory efficiently**.
4) **Productivity:**
   - Code written in procedure is shared by all programmers. This eliminates redundant coding by multiple programmers so overall improvement in productivity.
5) **Integrity:**
   - A procedure or function needs to be tested only once to verify its working.
   - After this, Oracle is responsible to maintain its integrity.
   - **Example:** If tables is **altered or destroyed**, for which procedure is created. Oracle automatically makes this procedure or function **invalid**.

Structure of a Procedure and Function

- **A procedure and function has three section as describe below:**

1) **Declaration:**
   - This section defines **variable, constants, cursors, exception and other procedure and function**.
   - These objects are local to procedure or function and they become invalid once the procedure or function exits.
2) **Executable Commands:**
   - This section contains SQL and PL/SQL statements that perform a specific task assign to procedure or function.
   - Parameter passed to procedures or functions are utilized here. Data is return back to the calling function or procedure from this section.
3) **Exception Handling:**
   - This section contains code that deals with exceptions generate during the execution of code.

Creating a Procedure

Syntax:

```
CREATE   [OR REPLACE]    PROCEDURE proc_name (argument  [IN, OUT, IN OUT]  datatype)
IS
        Declaration section
BEGIN
        Execution section
EXCEPTION
        Exception section
END ;
```

- While declaring a local variable, size cannot be specified. Only **datatype** needs to be specified of a variable.
- When this procedure is executed first time, the code will be compiled.
- If there is no error then a procedure is created and stored in oracle database.

Explanation:

1) **CREATE:** It will create a procedure.
2) **REPLACE:** It will re-create a procedure if it already exists.If it re-created then oracle recompile it automatically.
3) **We can pass parameters to the procedures in three ways:**
   1. **IN-parameters:** These types of parameters are used **to send values to stored procedures**.
   2. **OUT-parameters:** These types of parameters are used **to get values from stored procedures**. This is similar to a return type in functions but procedure can return values for more than one parameters.
   3. **IN OUT-parameters:** This type of parameter allows us **to pass values into a procedure and get output values from the procedure**.
4) **IS** : It indicates the beginning of the body of the procedure. The code between **IS** and **BEGIN** forms the **Declaration section**.
5) **BEGIN:** It contains the executable statement.
6) **Exception:** It contains exception handling part. This section is **optional**.
7) **END:** It will end the procedure.

- By using **CREATE OR REPLACE** together the procedure is created if it does not exist and if it exists then it is replaced with the current code.

Executing a Procedure

- There are two ways to execute a procedure:

1) **From the SQL prompt:**

   Syntax:

   ```
   EXECUTE  [or EXEC]  procedure_name (parameter) ;
   ```

2) **To execute procedure from PL/SQL block. OR Within another procedure – simply use the procedure name.**

   Syntax:

   ```
   procedure_name (parameter) ;
   ```

- A store procedure cannot be used with **SELECT** statement.

Example: (Using IN)

```
CREATE  OR REPLACE  PROCEDURE  get_studentname_by_id   (id IN NUMBER)
IS
BEGIN
        SELECT   studentname    FROM stu_tbl
          WHERE studentID = id ;
END;
```

Execute:

```
EXECUTE   get_studentname_by_id  (10) ;     OR
get_studentname_by_id (10) ;
```

Explanation: Above procedure gives the name of student whose id is 10.

Example 12 : Create a procedure debitAcc which debit a given account with specified amount. Input:

```
CREATE  OR REPLACE PROCEDURE debitAcc  (no IN  Account.Acc_No%TYPE, amount IN  NUMBER)
IS
BEGIN
                    --declare local variable
                              bal        Account.Balance%TYPE ;
                              newBalance      Account.Balance%TYPE ;

                    --Retrieve current balance for given account
                              SELECT Balance INTO bal FROM Account
                              WHERE Acc_No = no ;
                    -- calculate balance
                              newBalance := bal - amount ;
                    -- update balance without worrying for negative balance
                              UPDATE Account SET balance = newBalance
                              WHERE Acc_No = no ;
                    -- display confirmation message
                              dbms_output.put_line( ' Account ' || no || ' debited.. ');

END ;
/
```

Output:

```
Procedure created.
```

Execute:

```
debitAcc  ( 'A01', 1000) ;                              OR
EXEC     debitAcc ( 'A01', 1000) ;
```

- If any error encountered during compilation , to display error following statement can be used:

```
SELECT * FROM  user_error;
```

Creating a Function

Syntax:

```
CREATE   [OR REPLACE]    FUNCTION func_name (argument IN dataType…) RETURN
          dataType
IS
          Declaration section
BEGIN
          Execution section
EXCEPTION
          Exception section
END ;
```

- While declaring a local variable, size cannot be specified. Only **datatype** needs to be specified of a variable.

- When this function is executed first time, the code will be compiled.
- If there is no error then a function is created and stored in oracle database.
- A function must return one value back to calling environment. **It cannot return more than one value like procedure**.

Explanation:

1) **CREATE:** It will create a function.
2) **REPLACE:** It will re-create a function if it already exists. If it re-created then oracle recompile it automatically.
3) **IN-parameters:** These types of parameters are used **to send values to stored Functions.**
4) **RETURN:** Function return value having data type **dataType.**

Executing a Function

- There are two ways to execute a procedure:
  1) **From the SQL prompt it should be used with SELECT statement.**
     Syntax:
     ```
     SELECT    function_name    (parameter)    FROM dual ;
     ```
  2) **To execute function from the PL/SQL block.**
     Syntax:
     ```
     fuction_name    (parameters) ;
     ```
- A stored function cannot be executed using **EXCE** command like **procedures**.

Example 13 : Create a function getBalance which the balance for the given account.

Input:

```
CREATE  OR REPLACE    FUNCTION    getBalance   (no     IN Account.Acc_No%TYPE )
            RETURN      NUMBER
IS
            --declare local variable
            bal         Account.Balance%TYPE ;
BEGIN
            --Retrieve current balance for given account
                        SELECT Balance INTO bal FROM Account
                        WHERE     Acc_No = no ;
            -- Return balance
                        RETURN     bal ;
END ;
/
```

Output:

```
Function Created.
```

Execute:

```
getBalance  ( 'A03') ;                          OR
SELECT    getBalance ( 'A03' )    FROM     dual ;
```

Destroying Procedure and Function

- To destroy a stored procedure:

  Syntax:

```
DROP    PROCEDURE    procedureName ;
```

Example:

```
DROP    PROCEDURE    debitAcc ;
```

Output:

```
Procedure dropped.
```

- To destroy a stored function:

  Syntax:

  ```
  DROP    FUNCTION     functionName ;
  ```

  Example:

  ```
  DROP    FUNCTION     getBalance ;
  ```

  Output:

  ```
  Function dropped.
  ```

Packages

- A package is one kind of database object.
- It is used to group together **logically related objects** like variables, constants, cursors, exceptions, procedures and functions.
- A successfully compiled package is stored in oracle database like procedures and functions.
- Unlike procedure and functions, **package itself cannot be called**.
- **Example:** In a banking system, all objects associated with transaction related activities can be grouped together in a package. Other package may contain objects associated with some other activities, like a procedure 'debitAcc' and function 'getBalance' can be grouped together in some common package.

Advantages

- Advantages of package are given below:

  1) **Modularity:**
     - Package provides modular approach to programming.
     - It is always to better to write more than **one smaller programs** instead of **one large program**.
  2) **Security:**
     - Programs can be created to provide various functionalities and can be group together into packages.
     - Privileges can be granted to these packages rather than entire tables. So, **privileges can be granted efficiently**.
  3) **Improved Performance:**
     - An entire package, including all objects within it, is loaded into memory when the first component is accessed.
     - This eliminates additional calls to other related objects which results in reduced disk I/O.
     - So, performance can be improved.
  4) **Sharing of Code:**
     - Once a package is created, objects in that package can be shared among multiple users.
     - This reduces the **redundant coding**.
  5) **Overloading of procedures and functions:**

- Procedures and functions can be overloaded using packages.

Structure of a Package

- A package contains two sections:
  1) **Package Specification**
  2) **Package Body**
- **While creating packages, package specification and package body are created separately.**

1) **Package Specification:**
   - Various objects (like variables, constants etc..) to be held by package are declared in this section.
   - This declaration is global to the package, means accessible from anywhere in the package.

Syntax:

```
CREATE   OR   REPLACE   PACKAGE   packageName
IS
                    -- Package Specification
END    packageName;
```

- Package specification consists of list of variables, constants, functions, procedures and cursors.

Example 14 : Create a package transaction that contains procedure 'debitAcc' and function 'getBalance' created earlier.

Input:

```
CREATE   OR  REPLACE   PACKAGE      transaction

IS
          PROCEDURE    debitAcc ( no IN Account.Acc_No%TYPE, amount IN NUMBER) ;
          FUNCTION     getBalance (no IN Account.Acc_No%TYPE) RETURN
          NUMBER ;
END       transaction ;
/
```

Output:

```
Package Created.
```

2) **Package Body:**
   - It contains the **formal definition** of all the objects declared in the specification section.

Syntax:

```
CREATE   OR   REPLACE   PACKAGE   BODY    packageName
IS
                    -- package body
END    packageName;
```

- If a package contains only **variables, constants and exceptions** then package body is **optional**.

Example 15 : Create a package body for package transaction that contains procedure 'debitAcc' and function 'getBalance' created earlier.

Input:

```
CREATE   OR  REPLACE   PACKAGE   BODY   transaction
IS
     -- define procedure 'debitAcc'
          CREATE  OR REPLACE    PROCEDURE    debitAcc   (no     IN
                    Account.Acc_No%TYPE,  amount     IN     NUMBER)
          IS
                    bal        Account.Balance%TYPE ;
                    newBalance      Account.Balance%TYPE ;
          BEGIN
                    SELECT Balance INTO bal FROM Account
                    WHERE    Acc_No = no ;
                    newBalance := bal - amount ;
                    UPDATE Account SET balance = newBalance
                    WHERE    Acc_No = no ;
                    dbms_output.put_line( ‘ Account ‘ || no || ‘ debited.. ‘);
          END ;
     -- define function ‘getBalance’
          CREATE  OR REPLACE    FUNCTION   getBalance   (no    IN
                    Account.Acc_No%TYPE )      RETURN     NUMBER
          IS
                    bal        Account.Balance%TYPE ;
          BEGIN
                    SELECT Balance INTO bal FROM Account
                    WHERE    Acc_No = no ;

                    RETURN     bal ;
          END ;
END      transaction ;
/
```

Output:

Package body created.

- Package body is created, objects contained in package are stored in Oracle database.

Referencing a Package Subprogram

- To access objects specified inside a package, following syntax can be used.

  packageName.object

- The use of **‘.’ (dot)** to combine package name and procedure name to access procedure.

  Example 16 : Provide statement to debit an account ‘A01’ by amount 1000.

  Input:

  ```
  EXEC    transaction.debitAcc ( ‘A01’, 1000);
  ```

  Output:

  ```
  Account A01 debited..
  ```

  Example 17 : Provide statement to getbalance for an account ‘A01’.

  Input:

  ```
  SELECT    transaction.getBalance ( ‘A01’ )   FROM     dual ;
  ```

  Output:

  ```
  TRANSACTION.GETBALANCE (‘A01’)
  ```

```
-------------------------------------------------------------
                           4000
```

- It is possible to have same name for two different objects held by two different packages.
- In this situation, package name provides **unique identification** between such objects.

Destroying a Package

Syntax:

```
DROP   PACKAGE  [ BODY ]       packageName ;
```

- If **BODY** option is provided, body of the specified package is deleted and leaves package specification unchanged.
- If **BODY** option is not provided, both sections of the package are deleted.

Triggers

- A **trigger** is a group or set of SQL and PL/SQL statements that are executed by **Oracle itself**.
- The main characteristic of the trigger is that it is **fired automatically** when DML statements like Insert, Delete, and Update is executed on a table.
- **The advantages of triggers are as given below:**
  - To prevent **misuse** of database.
  - To implement **automatic backup** of the database.
  - To implement **business rule constraints**, such as balance should not be **negative**.
  - Based on change in one table, we want to update other table.
  - To **track the operation** being performed on specific tables with details like operation, time when it is performed, user name who performed it, etc..

- **Difference between Triggers and Procedures:**

Triggers	Procedures
○ Triggers are invoked implicitly.	○ Procedures need to invoke by users explicitly.
○ Trigger cannot accept parameters.	○ Procedures can accept parameter.
○ Trigger cannot return a value.	○ Procedures can return values
○ Trigger can only be executed, whenever an event (insert, delete, and update) is fired on the table on which the trigger is defined.	○ To execute a procedure, EXEC command is used.

- Oracle allows to implement various integrity constraints while creating a table to restrict data in tables. These constraints are called **declarative integrity Constraints**.

- **Difference between Triggers and Declarative Integrity Constraints:**

Triggers	Declarative Integrity Constraint
○ Trigger is applicable to those data that are loaded before the trigger is created.	○ A constraint is applicable to all the data stored in a table.
○ Trigger can implement transitional constraint. **Example:** ask for password for specific operation.	○ Implementation of transitional constraint is not possible with declarative constraint.
○ Triggers do not guarantee.	○ Constraint guarantees all data in a table conforms the rules implemented.

Structure of a Trigger

- A trigger contains three basic sections:
    1) **Triggering Event or Statement**
    2) **Trigger Restriction**
    3) **Trigger Action**

1) **Triggering Event or Statement:**
    - It is an SQL statement that causes a trigger to be fired.
    - It can be **INSERT**, **UPDATE** or **DELETE** statement for specific table.
2) **Trigger Restriction:**
    - It specifies a **condition** that must be true for a trigger to fire.
    - It is specified using **WHEN** clause.
3) **Trigger Action:**
    - It contains SQL and PL/SQL statements as well as stored procedures that are executed when trigger is fired.
    - This block specifies actions that need to be performed whenever a trigger is fired.

Types of Triggers

- Triggers can be classified based on two different criteria:
    1) **Based on number of times trigger action is executed.**
        a) Row Trigger
        b) Statement Trigger
    2) **Based on timing when trigger action is executed.**
        a) Before Trigger
        b) After Trigger

1) **Based on number of times trigger action is executed:**

Row Trigger	Statement Trigger
○ Fired **each time** the table is affected by the triggering statement.	○ Fired only **once.**
○ **Example:** If an **UPDATE** statement updates **multiple rows** of a table, a row trigger is fired **once for each row** affected by the **UPDATE** statement	○ **Example:** If an **UPDATE** statement updates **multiple rows** of a table, statement trigger is fired only **once.**
○ If no rows are affected by the triggering statement, a trigger will **not be executed**.	○ Trigger will be **executed once**, if no rows are affected by the triggering statement.

2) **Based on timing when trigger action is executed:**

Before Trigger	After Trigger
○ Trigger is executed before the triggering statement.	○ Trigger is executed after the triggering statement.
○ Used to determines whether the triggering statement should be allowed to execute or not.	○ Used when there is a need for a triggering statement to complete execution before trigger.

- These types are used in combination, provides total four types of triggers:
    1) **Before Statement:** Execute trigger once before triggering statement.
    2) **Before Row:** Execute trigger multiple times before triggering statement.

3) **After Statement:** Execute trigger once after triggering statement.
4) **After Row:** Execute trigger multiple times after triggering statement.

Creating a Trigger

Syntax:

```
CREATE   [OR REPLACE]   TRIGGER     trigger_name
         [BEFORE / AFTER]
         [INSERT / UPDATE / DELETE    [of columnName]  ]
ON       table_name
         [REFERENCING    [OLD AS old, NEW AS new]  ]
         [FOR EACH ROW [WHEN condition] ]
DECLARE
         Declaration section
BEGIN
        Executable statements
EXCEPTION
        Exception handling
END ;
/
```

Explanation of Keywords:

Keyword	Specifies..
REPLACE	Re- creates the trigger if it already exists.
BEFORE	Before updating the table, trigger should be fired.
OF columnName	This clause is used when you want to trigger an event only when a specific column is updated. This clause is mostly used with update triggers.
AFTER	After updating the table, trigger should be fired.
DELETE	Indicates that trigger will be fired on DELETE operation.
INSERT	Indicates that trigger will be fired on INSERT operation.
UPDATE	Indicates that trigger will be fired on UPDATE operation.
ON	Specifies table or view for which trigger is defined.
REFERENCING	Specifies correlation names – OLD and NEW- that specify old and new value for a record during triggering statement. For UPDATE, both are applicable; for INSERT only NEW is applicable; for DELETE only OLD is applicable.
FOR EACH ROW	Creates ROW type trigger. If omitted, statement type trigger is created.
WHEN condition	Specifies condition as a trigger restriction. The trigger is fired only for rows that satisfy the condition specified. This clause is valid only for row type triggers.

Example 18 : Using trigger, display message if balance is negative during insert operation on Account table.

Input:

```
CREATE   OR REPLACE   TRIGGER     balNegative
         BEFORE INSERT
ON       Account
         FOR EACH ROW
```

```
            BEGIN
                    IF        :NEW.Balance  <  0        THEN
                    dbms_output.put_line ('Balance is negative..');
Output:             END IF ;
            END ;
            /
            Trigger created.
```

Example 19 : Insert a new record in Account table having balance -1000.

Input:

```
            INSERT INTO Account values ( 'A07', -1000, 'Vrl') ;
```

Output:

```
            Balance is negative..
```

- The message 'Balance is negative..' indicates that trigger has executed before the insert operation.

Destroying a Trigger

Syntax:

```
            DROP   TRIGGER     triggerName ;
```

Example: `DROP   TRIGGER     balNegative ;`

Output: Triggered dropped.

RAISE_APPLICATION_ERROR procedure

- Trigger cannot use commands like **ROLLBACK**, **COMMIT** and **SAVEPOINT**. So, it is not possible to **undo** some operation.
- Oracle provides a procedure called **RAISE_APPPLICATION_ERROR** that can be used to generate errors and display user-defined error messages.
- When such kind of error is generated, program will simply terminate.
- So, this procedure can be used with trigger to **prevent execution of operation** which break integrity rules.

Syntax:

```
            RAISE_APPLICATION_ERROR ( errorNumber, errorMessage) ;
```

- errorNumber is negative number indicating error.
- errorMessage is a character string.
- This procedure raises an error, **terminates sub-program**, **rollback** any database changes made by that sub-program and display user defined **errorNumber** and **errorMessage**.

Example 20 : Using trigger, prevent the insertion operation if balance being inserted is negative.

Input:

```
            CREATE   OR REPLACE   TRIGGER    balNegative
                      BEFORE INSERT
            ON        Account
            FOR  EACH  ROW
            BEGIN
                    IF        :NEW.Balance  <  0        THEN
                    RAISE_APPLICATION_ERROR ( -20000, 'Balance is negative') ;
                    END IF ;
            END ;
            /
```

Output:

```
Trigger created.
```

Example 21 : Insert a new record in Account table having balance -5000.

Input:

```
INSERT INTO Account VALUES ( 'A08', -5000, 'Vrl');
```

Output:

```
INSERT INTO Account VALUES ( 'A08', -5000, 'Vrl');
ERROR at line 1:
ORA-20000: Balance is negative
```

- **Disadvantage of trigger:**
  - It is not possible to **track** or **debug** triggers.
  - Triggers can execute every time some field in database is updated. If a field is likely to be updated often, it is a **system overhead**.
  - It is easy to view table relationships, constraints, indexes, stored procedure in database but **triggers are difficult to view**.

- To process table data, it must be stored into variables and this task performed by **SELECT…INTO…** statement.
- But, this statement suffers from a limitation that it can store data only from a single record by using **WHERE** clause. It cannot be used with multiple records.
- So, if there is a need to simply display all records of an **Account** table using PL/SQL block, it is not possible.
- **Cursor** provides solution to this problem.

Cursors

- Whenever an SQL statement is executed, Oracle reserves a private **SQL area in memory**.
- The data required to execute the statement are **loaded** in this memory area from the **hard disk**.
- Once data are stored in memory, they are processed as per the operation.
- After processing is finished, updated data are stored back to the hard disk and **memory is freed**.
- Cursor comes into picture for this kind of processing.
- **A Cursor is an area in memory where the data required to execute SQL statement.**
- So, a cursor referred as work area.
- So, the **size of the cursor** will be the same as a size to hold this data.
- **Active Data Set:** The data (Set of rows) that is stored in the cursor is called Active Data Set.
- **Result Set:** Data is stored in cursor because of some SQL statement. So, it is called Result Set.
- **Current Row:** The row that is being processed is called the Current Row.
- **Row Pointer:** A pointer that is used to track the current row is known as Row Pointer.
- **Cursor Attributes:** Multiple cursor variables are used to indicate the current status of the processing being done by the cursor. These kinds of variables are known as Cursor Attributes.

- **Various cursor attributes are described in given below table:**

Attribute Name	Description
%ISOPEN	If cursor is **open**, returns **TRUE**. Else returns **False**.
%ISFOUND	If record **fetched** successfully, returns **TRUE**. Else returns **FALSE**.
%NOTFOUND	If record was **not fetched** successfully, returns **TRUE**. Else returns **FALSE**.
%ROWCOUNT	Returns **number of records** processed by the cursor.

- **There are two types of cursors in PL/SQL:**
  1) **Implicit Cursor**
  2) **Explicit Cursor**

Account:

Acc_No	**Balance**	**B_Name**
A01	2000	RJT
A02	5000	AHMD
A03	3000	SRT
A04	6000	RJT

1) Implicit Cursor:

- A cursor is called an **Implicit Cursor**, if it is opened by **Oracle itself** to execute SQL Statement like **SELECT**, **INSERT**, **UPDATE** or **DELETE**.
- It is opened and managed by Oracle itself. So, user needs not to care about it.
- We cannot use implicit cursors for **user defined work**.
- Oracle performs following operation to **manage** an implicit cursor:
  - Reserve an area in memory to store data required to execute SQL statement.
  - Occupy this area with required data.
  - Processes data.
  - Frees memory area by **closes a cursor**, when processing is completed.
- The syntax to use attributes of implicit cursor can be given as:

  SQL%AttributeName
- The value of the cursor attribute always refers to the SQL command that was **executed most recently**.
- **Before open** implicit cursor, its attribute contains **NULL** as value.

- **The meaning of cursor attribute in context of implicit cursor are described in given below table:**

Attribute	Description
SQL%ISOPEN	Always returns **FALSE**, because Oracle automatically **closes cursors** after executing SQL statement.
SQL%FOUND	If **SELECT** found any record or **INSERT**, **UPDATE** and **DELETE** affected any record then return **TRUE**. Else returns **FALSE**.
SQL%NOTFOUND	If **SELECT** found **no** any record or **INSERT**, **UPDATE** and **DELETE** affected **no** any record then returns **TRUE**. Else returns **FALSE**.
SQL%ROWCOUNT	Returns **number of records processed** by **SELECT, UPDATE, INSERT** or **DELETE** operations.

Example 22 : In Account table, branch names are stored in upper case letters. Convert branch name into lower case letters for a branch specified by the user. Also display how many accounts are affected.

Input:

```
DECLARE
        -- Declare required variables
        branch Account.B_Name%TYPE;
BEGIN
        -- read a number from the user
                branch := &branch;
        -- modify branch name
                UPDATE Account SET B_Name = LOWER(branch)
                WHERE    B_Name = branch;
        -- display number of record updated if any
                IF SQL%FOUND THEN
                        dbms_output.put_line(' Total '|| SQL%ROWCOUNT || '
                        records are updated.');
               ELSE
                      dbms_output.put_line(' Given branch not available.');

               END IF ;
       END;
        /
```

Output 1:

```
Enter value for branch: 'surat'
Given branch not available
```

Output 2 :

```
Enter value for branch:'RJT'
Total 2 records are updated
```

- If your **Account** table defines as **foreign key** referencing **Branch** table then this kind of update operation will get **failed**.

2) Explicit Cursor:

- A cursor is called **Explicit Cursor**, if it is **opened by user** to process data through PL/SQL block.
- It is opened by user. So, user has to take care about managing it.
- It is **used** when there is a need to process **more than one record individually**.
- Even though the cursor stores multiple records, only one record can be processed at a time, which is called as **current row**.
- Following steps required to manage an explicit cursor:

- Declare a cursor
- Open a cursor
- Fetching data
- Processing data
- Closing cursor

1) **Declare a Cursor:**

Syntax:

```
CURSOR   cursorName    IS    SELECT …. ;
```

- A cursor with **cursorName** is declared.
- It is mapped to a query given by **SELECT** statement.
- Here, only cursor will be declared. No any memory is allocated yet.

Example:

```
CURSOR   cursorAcc    IS
SELECT     Acc_No, Balance, B_Name    FROM Account ;
```

2) **Open a Cursor:**

- Once cursor is declared we can open it.
- When cursor is opened following operations are performed:
  - ✓ Memory is allocated to store the data.
  - ✓ Execute SELECT statement associated with cursor.
  - ✓ Create active data set by retrieving data from table.
  - ✓ Set the cursor row pointer to point to first record in active data set.

Syntax:

```
OPEN cursorName ;
```

3) **Fetching Data:**

- We cannot process selected row directly. We have to **fetch column values** of a row into **memory variables**.
- This is done by **FETCH** statement.

Syntax:

```
FETCH   cursorName   INTO    variable1, variable2………. ;
```

- Retrieve data from the current row in the active data set and stores them in given variables.
- Data from a **single row** are fetched at a time.
- After fetching data, **updates row pointer** to point the **next row** in an active data set.
- **Variables** should be **compatible** with the columns specified in the SELECT statement.

  Example:

  ```
  FETCH   cursorAcc    INTO    no, balance, bname ;
  ```

- Fetched account number, balance and branch name from **current row** in active data set and **store** them in respective variables.
- To process **more than one record**, the **FETCH** statement is enclosed within loop like **LOOP … END LOOP** can be used.

4) **Processing data:**

- This step involves actual processing of current row by using PL/SQL as well as SQL statements..

5) **Closing Cursor:**

- A cursor should be closed after the processing of data completes. Once you close the cursor it will release memory allocated for that cursor.
- If user forgets to close the cursor, it will be automatically closed after termination of the program.

Syntax:

```
CLOSE     cursorName ;
```

- The syntax to use attributes of explicit cursor can be given as:

  SQL%AttributeName

- **The meaning of cursor attribute in context of explicit cursor are described in given below table:**

Attribute	Description
SQL%ISOPEN	If explicit cursor is open, returns **TRUE**. Else Return **False**.
SQL%FOUND	If record was fetched successfully in last **FETCH** statement then return **TRUE**. Else returns **FALSE** indicating no more records available in active data set.
SQL%NOTFOUND	If record was not fetched successfully in last **FETCH** statement returns **TRUE**. Else returns **FALSE**.
SQL%ROWCOUNT	Returns **number of records fetched** from active data set. It is set to **ZERO** when cursor is **opened**.

Example 23 : Transfer all the accounts belonging to 'RJT' branch from Account table into another table 'Account_RJT' having only 2 column Acc_No and Balance. If table is not available then first create it.

Input:

```
DECLARE
        -- declare a cursor
                CURSOR  cursorAcc  IS
                SELECT Acc_No, Balance, B_Name FROM Account ;
        --declare required variables
                no     Account.Acc_No%TYPE ;
                balance    Account.Balance%TYPE ;
                branch    Account.B_Name%TYPE ;
BEGIN
        --open a cursor
                OPEN   cursorAcc ;
        --if cursor is opened successfully then process data
        --Else display error message
                IF cursorAcc%ISOPEN THEN
                --traverse loop
                    LOOP
                --fetch data from cursor row into variavbles
                        FETCH cursorAcc INTO no, balance, branch;
                --if no record available in active data set then exit from loop
                        EXIT WHEN cursorAcc%NOTFOUND ;
                --process data. If record belongs to 'RJT' branch, transfer it
                        IF branch = 'RJT' THEN
                        -- insert record into Account_RJT table
                            INSERT INTO Account_RJT VALUES(no, balance);
                        --delete record from the Account table
```

```
                                        DELETE FROM Account WHERE Acc_No = no ;
                                    END IF ;
                                END LOOP;
                        --commit operations
                                COMMIT;

                        ELSE
                                    dbms_output.put_line ('Cursor cannot be opened.') ;
                        END IF ;
            END ;
            /
```

- After executing this PL/SQL block, display data from Account and Account_RJT tables.

Example 24 : Display data from the Account table.

Input:

```
SELECT * FROM Account ;
```

Output:

```
        Acc_No          Balance         B_Name
    -------------------------------------------------------------
        A02             5000            AHMD
        A03             3000            SRT
```

Example 25 : Display data from the Account_RJT table.

Input:

```
SELECT * FROM Account_RJT;
```

Output:

```
        Acc_No          Balance
    -----------------------------------------
        A01             2000
        A04             6000
```

Cursor FOR Loop

- **FETCH** statement can fetch data from single row of an **active data set**. But, there will be a need to process **multiple rows** most of the times.
- So, **FETCH** statement is enclosed within a loop to process multiple rows.
- For that oracle provide another loop statement that is a variation of the basic **FOR** loop.

  Syntax:

  ```
  FOR    variable    IN    cursorName
  LOOP
          -- Execute commands
  END LOOP;
  ```

- **Above syntax performs following operations automatically:**
  - A given **variable** is created of the **%ROWTYPE** and refer to the entire row.
  - Specified cursor is opened.
  - Data from the row of the active data set are fetched into given variable for each iteration of the loop.

  - **Exits** from the loop and **closes** the cursor.

- Here, variable of %ROWTYPE is refer to entire row.
- The individual fields of the record can be accessed as given below:

  variableName.columnName

Example 26 : Transfer all the accounts belonging to 'RJT' branch from Account table into another table 'Account_RJT' having only 2 column Acc_No and Balance. If table is not available then first create it.

Input:

```
DECLARE
        -- declare a cursor
                CURSOR   cursorAcc  IS
                SELECT Acc_No, Balance, B_Name FROM Account ;
BEGIN
        --use of a cursor FOR loop. varAcc is declared as a type of %ROWTYPE
                FOR   varAcc   IN    cursorAcc
                LOOP
                --process data. If record belongs to 'RJT' branch, transfer it
                        IF varAcc.B_Name = 'RJT' THEN
                        -- insert record into Account_RJT table
                             INSERT INTO Account_RJT VALUES(varAcc.ACC_No,
                                        varAcc.Balance);
                        --delete record from the Account table
                             DELETE    FROM   Account
                                WHERE Acc_No = varAcc.ACC_No ;
                        END IF ;
                END LOOP;
                --commit operations
                     COMMIT;
                ELSE
                        dbms_output.put_line ('Cursor cannot be opened.') ;
                END IF ;
END ;
/
```

- Same output can be observe as given example 3 and 4 after executing this PL/SQL block.

Parameterized Cursors

- Up to this point, active data set contains all the records from the given table.
- Now if we want to create an active data set that contains only selected records from the given table.
- **For example**, we want to create an active data set that contains records belonging to '**RJT'** branch only not all records of **Account** table.
- For this purpose, oracle allows to pass **parameters** to cursor that can be used to provide condition with **WHERE** clause.
- If parameters are passed to cursor, that cursor is called **parameterized cursor**.
- Syntax to declare **parameterized cursor** is:

  Syntax:

  ```
  CURSOR   cursorName  (variableName  datatype)   IS   SELECT….. ;
  ```

- While opening cursor, parameter can be passed using following syntax**:**

Syntax:

```
OPEN     cursorName (value / variable / expression);
```

Example 27 : Transfer accounts of ‘RJT’ branch of Account to ‘Account_RJT’ table.

Input:

```
DECLARE
        -- declare a cursor
        CURSOR cursorAcc (brName Account.B_Name%TYPE) IS
                SELECT     Acc_No, Balance, B_Name    FROM Account
                WHERE      B_Name = brName;
        --declare required variables
                no     Account.Acc_No%TYPE ;
                balance    Account.Balance%TYPE ;
                branch     Account.B_Name%TYPE ;
BEGIN
        --open a cursor
                OPEN    cursorAcc  (‘RJT’);
        --if cursor is opened successfully then process data
        --Else display error message
                IF cursorAcc%ISOPEN THEN
                --traverse loop
```

```
          LOOP
        --fetch data from cursor row into variavbles
            FETCH cursorAcc INTO no, balance, branch;
        --if no record available in active data set then exit from loop
            EXIT WHEN cursorAcc%NOTFOUND ;
        --process data
        --no need to check whether record belongs to 'RJT' branch
            -- insert record into Account_RJT table
               INSERT INTO Account_RJT VALUES(no, balance);
            --delete record from the Account table
               DELETE FROM Account WHERE Acc_No = no ;
          END LOOP;
        --commit operations
          COMMIT;
        ELSE
            dbms_output.put_line ('Cursor cannot be opened.') ;
        END IF ;
END ;
/
```

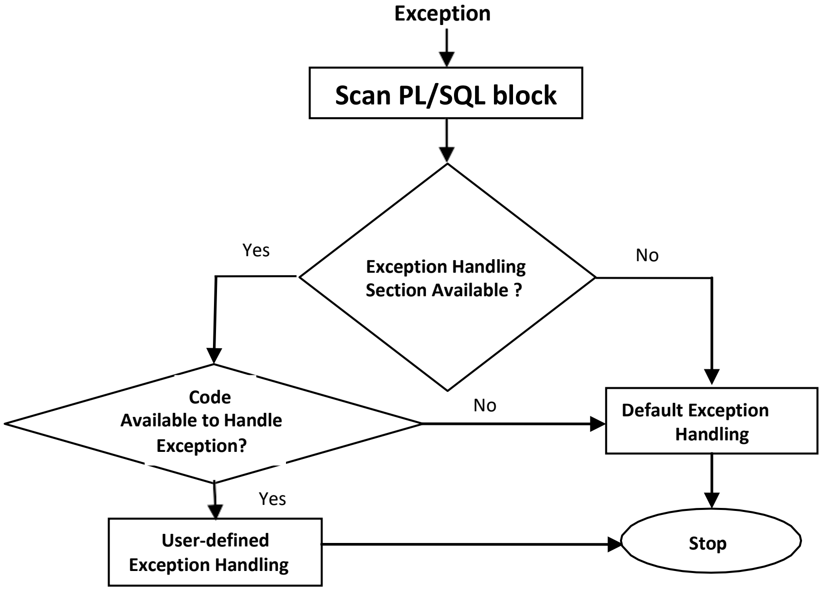
- Observe the data of Account and Account_RJT tables.

Exception Handling

- **Run-time** errors can be handled in some useful way rather than getting system specific message and terminating program directly. It's called **exception handling**.

Built-In Exception Handler

- An error occurs at run-time is called an **exception**.
- These errors may involve the operation like divide by zero, access to unauthorized data, etc.. in PL/SQL block.
- A block of code that attempts to **resolve** exceptions is known as **exception handler**.
- Following diagram shows the working of exception handler:


Exception
Scan PL/SQL block
Exception Handling Section Available ?
Yes
No
Code Available to Handle Exception?
No
Default Exception Handling
Yes
User-defined Exception Handling
Stop

There are two types of exception:

1) **System Exception:** In PL/SQL, various run-time errors are associated with different exceptions. These types of exceptions are known as system exception.
2) **User-defined Exception:** User also can define their own exception is known as user-defined exception.

- Exception handler scans the PL/SQL block to check existence of the **Exception Handling** section within block.
- If it is available then it is checked to find the code to handle exception.

Syntax:

```
        EXCEPTION
                WHEN    exceptionName       THEN
                        -- code to handle exception.
```

- Here, it contains more than on **WHEN** clauses.
- An **exceptionName** is a character string represents an exception to be handled.
- If **exception handling section is available** and an exception is **raised** then the appropriate code is executed. **Otherwise** an exception is handled using **default exception handling** code that is simply displaying an error message or terminating the program.

Types of Exceptions

- Exception can be either **System Exception (Pre-defined Exception)** or **User-define Exception**.
- System Exceptions can be further divide in to two parts:

1) **Named Exceptions**
2) **Numbered Exceptions**

1) **Named Exceptions:**
   - Particular name given to some common system exceptions **is known as Named Exception.**
   - Oracle has defined 15 to 20 named exceptions.
   - Some of the named exceptions are listed in below table:

Exception	**Raised When…**
INVALID_NUMBER	TO_NUMBER function failed in converting string to number.
NO_DATA_FOUND	SELECT … INTO statement couldn’t find data.
ZERO_DIVIDE	Divide by zero error occurred.
TOO_MANY_ROWS	SELECT … INTO statement found more than one record.
LOGIN_DENIED	Invalid usename or password found while logging.
NOT_LOGGED_ON	Statements tried to execute without logging.
INVALID_CURSOR	A cursor is attempted to use which is not open.
PROGRAM_ERROR	PL/SQL found internal problem.
DUP_VAL_ON_INDEX	Duplicate value found in column defined as unique or primary key.
VALUE_ERROR	Error occurred during conversion of data.
OTHERS	Stands for all other exceptions.

2) **Numbered Exceptions:**
   - These exceptions are identified by using **negative signed number**, such as -1200.
   - Oracle has defined more than **20000** numbered exceptions.

- **Named exceptions** also can be associated with numbers. So they can be considered as a **sub-set of numbered exceptions**.

3) **User-defined Exceptions:**
   - User also can define their own exceptions are known as **user define exceptions**.
   - These exceptions are used to **validate business rules** like balance for any account should not be negative value.
   - User-defined exceptions need to be **declared, raised and handled explicitly**.

     Syntax for exception declaration:

     ```
     exceptionName        EXCEPTION;
     ```

     Syntax for exception raised:

     ```
     RAISE     exceptionName;
     ```

Handling Named Exceptions

Example 28 : Create an Account with Acc_No as a primary key. Write a PL/SQL block to insert a record in this table. Also handle named exceptions DUP_VAL_ON_INDEX. Which is raised on encountering duplicate value for primary or unique key. (Assume table is available)

Input:

```
DECLARE
        -- declare required variable
                no    Account.Acc_No%TYPE;
                bal    Account.Balance%TYPE;
                branch    Account.B_Name%TYPE;
BEGIN
        --read an account number, balance and branch name for new record
                no := &no;
                bal  :=   &bal;
                branch :=  &branch;
        --insert record into Account table
                INSERT INTO Account VALUES (no, bal, branch);
        --commit and display message confirming insertion
                COMMIT;
                dbms_output.put_line('Record inserted successfully.');
EXCEPTION
        --handle named exception
                WHEN   DUP_VAL_ON_INDEX     THEN
                        dbms_output.put_line('Duplicate value found for primary
                key.');
END;
/
```

Output 1 :

```
Enter value for no: 'A01'
Enter value for bal: 5000
Enter value for no: 'RJT'
Record inserted successfully.
```

Output 2 :

```
Enter value for no: 'A01'
Enter value for bal: 10000
Enter value for no: 'SRT'
Duplicate value found for primary key.
```

- Here, instead of displaying system error message, user-defined error message is displayed.

Handling Numbered Exception

- A **WHEN** clause in exception handling section required **a character string** representing exception name to be handled.
- So, numbered exceptions **cannot** be handled directly like named exception.
- To handle numbered exceptions, they need **to be bound with some names**. This binding is provided in declaration section.
- After that it can be handle like named exception in exception section.

  Syntax:

```
DECLARE
        exceptionName           EXCEPTION;
        PRAGMA    EXCEPTION_INIT (exceptionName, errorName);
BEGIN
        --execute commans . . .
EXCEPTION
        WHEN    excepetionName      THEN
                -- code to Handle Exception . . .
END ;
/
```

  - A **PRAGMA** is **a call to pre-compiler** that **binds** the numbered exception to some **name**.
  - A function **EXCEPTION_INIT** takes two parameters: one is exception **name** and **number** of the exception to be handled.
  - Once **binding is provided**, exception can be handle in exception handling section using **WHEN** clause.

Example 29 : Along with named exception, in above example also handle numbered exception with number -1200, which is raised on encountering for primary or NOT NULL key. (Assume table is available)

Input:

```
DECLARE
        -- declare exception and bind it.
                exNull    EXCEPTION;
                PRAGMA      EXCEPTION_INIT (exNull, -1200);
        -- declare required variable
                no    Account.Acc_No%TYPE;
                bal    Account.Balance%TYPE;
                branch    Account.B_Name%TYPE;
BEGIN
        --read an account number, balance and branch name for new record
```

```
         no := &no;
         bal :=  &bal;
        branch  := &branch;
    --insert record into Account table
        INSERT INTO Account VALUES (no, bal, branch);
    --commit and display message confirming insertion
        COMMIT;
        dbms_output.put_line('Record inserted successfully.');
EXCEPTION
    --handle named exception
        WHEN   DUP_VAL_ON_INDEX     THEN
            dbms_output.put_line('Duplicate value found for primary
            key.');
    --handle numbered exception
        WHEN   exNull    THEN
            dbms_output.put_line('Null value found for primary key.')
END;
/
```

Output 1 :

```
Enter value for no: 'A02'
Enter value for bal: 6000
Enter value for no: 'RJT'
Record inserted successfully.
```

Output 2 :

```
Enter value for no: null
Enter value for bal: 10000
Enter value for no: 'SRT'
Null value found for primary key.
```

- Here, numbered exception **-1200** is bound with name '**exNull'**.
- So, we need to declare '**exNull'** first and then bound using **EXCEPTION_INIT** function.

Handling User-defined Exceptions

- In this case, user has to take care about **declaring an exception, raising it based on some condition** and then **handle** it.

  Syntax:

```
DECLARE
    exceptionName      EXCEPTION ;
BEGIN
    --SQL and PL/SQL statement
    IF condition THEN
        RAISE     exceptionName
    END IF ;
EXCEPTION
    WHEN    exceptionName     THEN
        -- codeto Handle Exception
END ;
```

```
/
```

- A user-define exception can be define in **declaration** section.
- A **RAISE** clause raises an exception and **transfer control** of execution from **executable commands section** to **exception handling section**.
- This exception handled in exception handling section.

Example 30 : In above example, raise an exception if inserted balance is negative value and display error message rather than inserting record in a table. (Assume that table is available).

Input:

```
DECLARE
        -- declare exception and bind it
                exNull EXCEPTION;
                PRAGMA    EXCEPTION_INIT (exNull, -1200);
        --declare exception
                myEx    EXCEPTION;
        -- declare required variable
                no    Account.Acc_No%TYPE;
                bal    Account.Balance%TYPE;
                branch    Account.B_Name%TYPE;
BEGIN
        --read an account number, balance and branch name for new record
                no := &no;
                bal  :=   &bal;
                branch :=  &branch;
        --check balance, if negative, raise 'myEx' exception
                IF bal > 0 THEN
                        RAISE    myEx;
                END IF ;
        --insert record into Account table
                INSERT INTO Account VALUES (no, bal, branch);
        --commit and display message confirming insertion
                COMMIT;
                dbms_output.put_line('Record inserted successfully.');
EXCEPTION
        --handle named exception
                WHEN   DUP_VAL_ON_INDEX    THEN
                        dbms_output.put_line('Duplicate value found for primary
                key.');
        --handle numbered exception
                WHEN   exNull    THEN
                        dbms_output.put_line('Null value found for primary key.')
        --handle user-defined exception
                WHEN   myEx    THEN
                        dbms_output.put_line('Balance cannot be negative value.')

END;
/
```

Output 1 :

```
Enter value for no: ‘A03’
Enter value for bal: 6000
Enter value for no: ‘RJT’
Record inserted successfully.
```

Output 2 :

```
Enter value for no: ‘A04’
Enter value for bal: -10000
Enter value for no: ‘SRT’
Balance cannot be negative value.
```