Unit – II
PL / SQL and Triggers

TOPIC TO BE COVERED…..

1. Basics of PL / SQL
2. Datatypes
3. Advantages
4. Control Structures : Conditional,

   Iterative, Sequential
5. Exceptions:

   Predefined Exceptions ,User defined exceptions
6. Cursors:

   Static (Implicit & Explicit), Dynamic
7. Procedures & Functions

TOPIC TO BE COVERED…..

8. Packages :

   Package specification, Package body, Advantages of package

9. Fundamentals of Database Triggers

10. Creating Triggers

11. Types of Triggers :

    Before, after for each row, for each statement

2.1 Basics of PL / SQL

- PL/SQL is Oracle's procedural language extension to SQL, the non-procedural relational database language.
- With PL/SQL, you can use SQL statements to manipulate ORACLE data and the flow of control statements to process the data.
- Moreover, you can declare constants and variables, define subprograms (procedures and functions), and trap runtime errors.
- Thus, PL/SQL combines the data manipulating power of SQL with the data processing power of procedural languages.

2.1 Basics of PL / SQL

- Many Oracle applications are built using client- server architecture. The Oracle database resides on the server.
- The program that makes requests against this database resides on the client machine.

- This program can be written in C, Java, or PL/SQL.
- While PL/SQL is just like any other programming language, it has syntax and rules that determine how programming statements work together.
- PL/SQL is a part of the Oracle RDBMS, and it can reside in two environments, the client and the server.

2.1 Basics of PL / SQL

- As a result, it is very easy to move PL/SQL modules between server-side and client-side applications.
- When the PL/SQL engine is located on the server, the whole PL/SQL block is passed to the PL/SQL engine on the Oracle server.
- The PL/SQL engine processes the block according to the Figure 2.1.

THE PL/SQL ENGINE PROCESSES

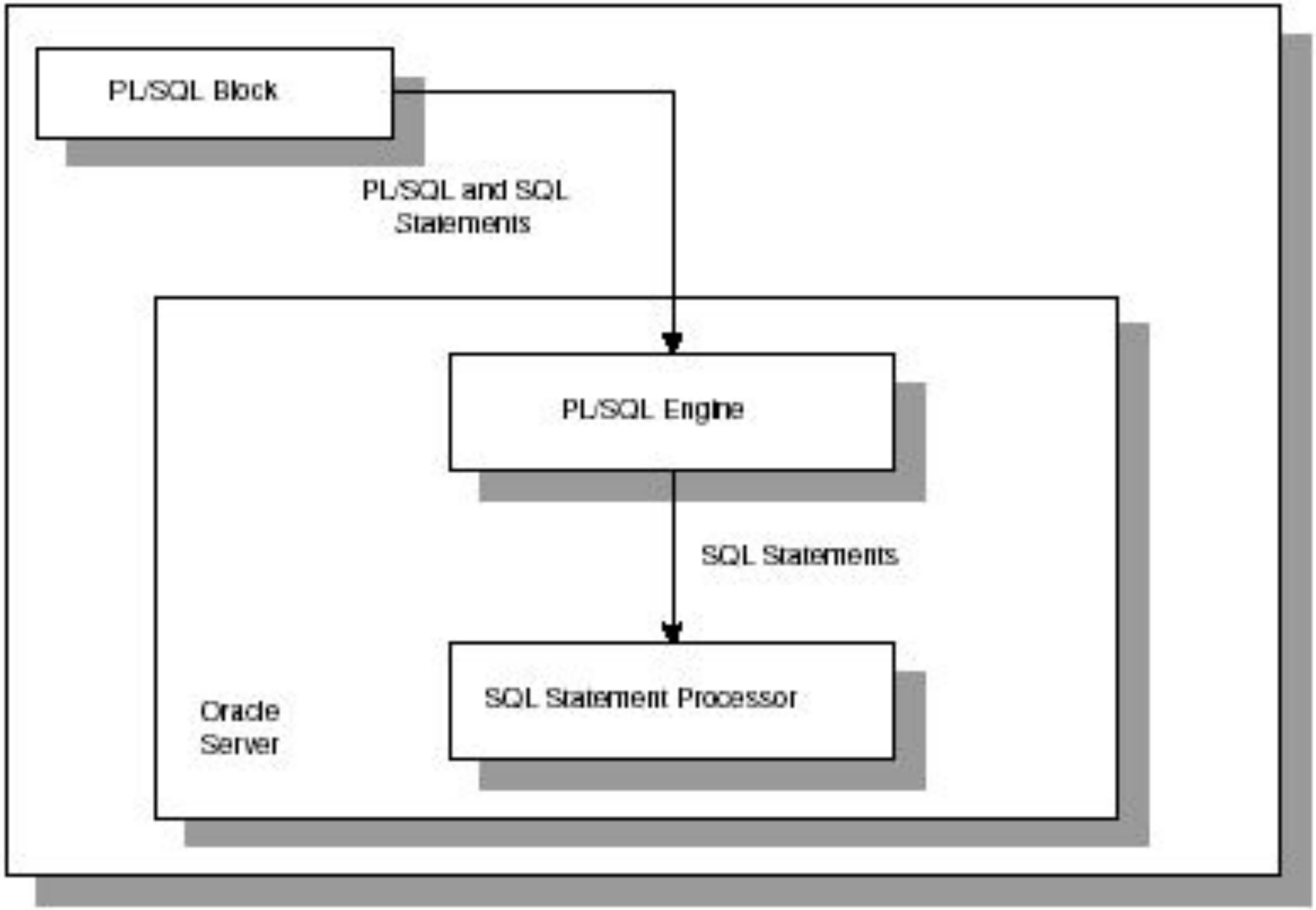


Figure 2.1 ■ The PL/SQL engine and Oracle server.

THE PL/SQL ENGINE PROCESSES

- When the PL/SQL engine is located on the client, as it is in the Oracle Developer Tools, the PL/SQL processing is done on the client side.
- All SQL statements that are embedded within the PL/SQL block are sent to the Oracle server for further processing. When PL/SQL block contains no SQL statement, the entire block is executed on the client side.

DIFFERENCE BETWEEN PL/SQL AND SQL

SQL	PL/SQL
SQL is a Structured Query Language.	PL-SQL is a procedural Structured Query Language.
SQL is executed one statement at a time.	PL/SQL is executed as a block of code.
SQL is used to write queries, DDL and DML statemen ts.	PL/SQL is used to write program blocks, functions, procedures triggers, and packages.
SQL does not support Exception Handling.	PL/SQL support Exception Handling.
SQL does not support variable declaration.	SQL does not support variable declaration.

PL/SQL BLOCKS

- PL/SQL blocks can be divided into two groups:

  Named and

  Anonymous.

- Named blocks are used when creating subroutines. These subroutines are procedures, functions, and packages.
- The subroutines can be stored in the database and referenced by their names later on.
- In addition, subroutines can be defined within the anonymous PL/SQL block.
- Anonymous PL/SQL blocks do not have names. As a result, they cannot be stored in the database and referenced later.

PL/SQL BLOCKS

- **PL/SQL blocks contain three sections**

  Declare section
  Executable section and
  Exception-handling section.

- The executable section is the only mandatory section of the block.
- Both the declaration and exception-handling sections are optional.

PL/SQL BLOCK STRUCTURE

```
            DECLARE
    ……….Optional
 <Declaration Section>
           BEGIN
    ……….Mandatory
 <Executable commands>
           EXCEPTION
    ……….Optional
 <Exception Handling>
END;
    ……….Mandatory
```

DECLARATION SECTION

- The declaration section is the first section of the PL/SQL block.
- It contains definitions of PL/SQL identifiers such as variables, constants, cursors and so on.
- **Example:**

```
DECLARE
  v_first_name    VARCHAR2(35)    ;
  v_last_name    VARCHAR2(35)    ;
  v_counter NUMBER := 0 ;
```

EXECUTABLE SECTION

- The executable section is the next section of the PL/SQL block.
- This section contains executable statements that allow you to manipulate the variables that have been declared in the declaration section.

```
BEGIN
    SELECT first_name, last_name  INTO
        v_first_name, v_last_name  FROM
        student
        WHERE student_id = 123 ;
    DBMS_OUTPUT.PUT_LINE
    (‘Student name :’ || v_first_name ||‘  ’||
    v_last_name);
END;
```

EXCEPTION-HANDLING SECTION

- The exception-handling section is the last section of the PL/SQL block.
- This section contains statements that are executed when a runtime error occurs within a block.
- Runtime errors occur while the program is running and cannot be detected by the PL/SQL compiler.

Example:

```
EXCEPTION
  WHEN NO_DATA_FOUND THEN
  DBMS_OUTPUT.PUT_LINE
  (‘ There is no student with student id 123  ’);
END;
```

HOW PL/SQL GETS EXECUTED

- Every time an anonymous block is executed, the code is sent to the PL/SQL engine on the server where it is compiled.
- The named PL/SQL block is compiled only at the time of its creation, or if it has been changed.
- The compilation process includes syntax checking, binding and p-code generation.
- Syntax checking involves checking PL/SQL code for syntax or compilation errors.
- Once the programmer corrects syntax errors, the compiler can assign a storage address to program variables that are used to hold data for Oracle. This process is called Binding.

HOW PL/SQL GETS EXECUTED

- After binding, p-code is generated for the PL/SQL block.
- P-code is a list of instructions to the PL/SQL engine.
- For named blocks, p-code is stored in the database, and it is used the next time the program is executed.
- Once the process of compilation has completed successfully, the status for a named PL/SQL block is set to VALID, and also stored in the database.
- If the compilation process was not successful, the status for a named PL/SQL block is set to INVALID.

PL/SQL IN SQL*PLUS

- SQL*Plus is an interactive tool that allows you to type SQL or PL/SQL statements at the command prompt.
- These statements are then sent to the database. Once they are processed, the results are sent back from the database and displayed on the screen.
- There are some differences between entering SQL and PL/SQL statements.

SQL EXAMPLE

- SELECT first_name, last_name
- FROM student;
- The semicolon terminates this SELECT statement. Therefore, as soon as you type semicolon and hit the ENTER key, the result set is displayed to you.

PL/SQL EXAMPLE

```
DECLARE
    v_first_name VARCHAR2(35);
    v_last_name VARCHAR2(35);
BEGIN
    SELECT first_name, last_name
    INTO v_first_name, v_last_name
    FROM student
    WHERE student_id = 123;
    DBMS_OUTPUT.PUT_LINE
    ('Student name: '||v_first_name||'
    '||v_last_name);
```

PL/SQL EXAMPLE

```
EXCEPTION
    WHEN NO_DATA_FOUND THEN
           DBMS_OUTPUT.PUT_LINE
                  ('There is no student with
    student id 123');
END;
.
/
```

PL/SQL EXAMPLE

- There are two additional lines at the end of the block containing “.” and “/”. The “.” marks the end of the PL/SQL block and is optional.
- The “/” executes the PL/SQL block and is required.
- When SQL*Plus reads SQL statement, it knows that the semicolon marks the end of the statement. Therefore, the statement is complete and can be sent to the database.
- When SQL*Plus reads a PL/SQL block, a semicolon marks the end of the individual statement within the block. In other words, it is not a block terminator.

PL/SQL EXAMPLE

- Therefore, SQL*Plus needs to know when the block has ended. As you have seen in the example, it can be done with period and forward slash.

EXECUTING PL/SQL

- PL/SQL can be executed directly in SQL*Plus.
- A PL/SQL program is normally saved with an .sql extension.
- To execute an anonymous PL/SQL program, simply type the following command at the SQL prompt:
- SQL> @DisplayAge

DATATYPES

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

VARIABLES

- Oracle allows to use variables in PL/SQL.
- In PL/SQL, variables containvalues resulting from queries to expressions.
- Variables are declared in declaration section.
- Declaring a variable:

Syntax:

VariableName datatype[NOT NULL] := initialValue;

Example:

city char(10);

counter number(2) NOT NULL :=0

VARIABLES

- **Anchored datatype:** A variable declared as anchored datatype means datatype for variable is determined based on the datatype of other object. This object can be other variable or column of table.

  Syntax:

  VariableName object% TYPE [NOT NULL]:= initialValue;

  Example:

  bal Account.ano%TYPE;

  name Customer.name%TYPE;

DECLARING A CONSTANT

- A constant is also used to store values but a value cannot be changed during program execution.

Syntax:

constantName CONSTANT datatype= initialValue;

Example:

pi CONSTANT number(3,2) := 3.14;

ASSIGNING A VALUE

- Assigning a value in two ways:

1. Using assignment operator(:=)

   Syntax:

   VariableName := Value;

   Example:

   no := 101;

2. Reading from keyboard

   Syntax:

   VariableName := &VariableName;

   Example:

   no :=&no;

ASSIGNING A VALUE

3. Selecting or fetching table data values into variables

Syntax:

SELECT col1,col2,…colN INTO var1,var2,…varN

FROM tableName WHERE condition;

GENERATING OUTPUT

- Like other programming languages, PL/SQL provides a procedure (i.e. PUT_LINE) to allow the user to display the output on the screen. For a user to able to view a result on the screen, two steps are required.
- First, before executing any PL/SQL program, type the following command at the SQL prompt (Note: you need to type in this command only once for every SQL*PLUS session):
- SQL> SET SERVEROUTPUT ON;

- or put the command at the beginning of the program, right before the declaration section.

GENERATING OUTPUT

- Second, use **DBMS_OUTPUT.PUT_LINE** in your executable section to display any message you want to the screen.
- **Syntax for displaying a message:**
- DBMS_OUTPUT.PUT_LINE(<string>);
- in which PUT_LINE is the procedure to generate the output on the screen, and DBMS_OUTPUT is the package to which the PUT_LINE belongs.
- DBMS_OUTPUT_PUT_LINE(‘My age is ‘ || num_age);

SUBSTITUTIONVARIABLES

- SQL*Plus allows a PL/SQL block to receive input information with the help of substitution variables.
- Substitution variables cannot be used to output the values because no memory is allocated for them.
- SQL*Plus will substitute a variable before the PL/SQL block is sent to the database.
- Substitution variables are usually prefixed by the ampersand(&) character or double ampersand (&&) character.

EXAMPLE

```
DECLARE
    v_student_id NUMBER := &sv_student_id;
    v_first_name VARCHAR2(35);
    v_last_name VARCHAR2(35);
BEGIN
    SELECT first_name, last_name
    INTO v_first_name, v_last_name
    FROM student
    DBMS_OUTPUT.PUT_LINE
           ('Student name: '||v_first_name||'
    '||v_last_name);
```

EXAMPLE

```
    WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('There is no such
    student');
END;
```

- When this example is executed, the user is asked to provide a value for the student ID.
- The example shown above uses a single ampersand for the substitution variable.
- When a single ampersand is used throughout the PL/SQL block, the user is asked to provide a value for each occurrence of the substitution variable.

EXAMPLE

```
BEGIN
     DBMS_OUTPUT.PUT_LINE('Today is ‘||’&sv_day');
     END;
```

This example produces the following output:

```
Enter value for sv_day: Monday
Today is Monday
```

EXAMPLE

- When a substitution variable is used in the script, the output produced by the program contains the statements that show how the substitution was done.
- If you do not want to see these lines displayed in the output produced by the script, use the SET command option before you run the script as shown below:
- **SET VERIFY OFF;**
- Using SET SERVEROUTPUT ON command we can display the output on.

EXAMPLE

- Then, the output changes as shown below:
  - **Enter value for sv_day: Monday**
  - **Enter value for sv_day: Tuesday**
  - **Today is Monday**
  - **Tomorrow will be Tuesday**
  - **PL/SQL procedure successfully completed.**
- The substitution variable sv_day appears twice in this PL/SQL block. As a result, when this example is run, the user is asked twice to provide the value for the same variable.

CONTROL STRUCTURE

- Conditional control

  IF – END IF

  IF – ELSE – END IF

  IF – ELSIF – ELSE – END IF

- Iterative control

  LOOP – EXIT WHEN – END LOOP

  FOR – LOOP – END LOOP

  WHILE – LOOP – END LOOP

- Sequential control

  GOTO statement

CONTROL STRUCTURE

- Conditional control

  IF – END IF

  IF – ELSE – END IF

  IF – ELSIF – ELSE – END IF

--- IF – END IF

```
DECLARE
    …
 BEGIN
    …
        v_commison := 7500;

    IF v_dept = 10 THEN
          v_commision := 5000;

END IF;
     …
  END;
/
```

--- IF – ELSE – END IF

```
DECLARE
     …
     …
BEGIN
          IF v_dept = 10 THEN
          v_commision := 5000;
           ELSE
          v_commision := 7500;
          END IF;
      …
      …
  END;
/
```

--- IF – ELSIF – ELSE – END IF

```
DECLARE
   BEGIN
          IF v_dept = 10 THEN
         v_commision := 5000;
     ELSIF v_dept = 20 THEN
         v_commison := 5500;
     ELSIF v_dept = 30    THEN
         v_commison := 6200;
     ELSE
         v_commision := 7500;
     END IF;
    END;
/
```

ITERATIVE CONTROL

LOOP – EXIT WHEN – END LOOP

FOR – LOOP – END LOOP

WHILE – LOOP – END LOOP

--- LOOP – EXIT WHEN – END LOOP

```
DECLARE
      v_deptno   dept.deptno%TYPE    := 50;
       v_counterinteger := 1;
      BEGIN
LOOP
         INSERT INTO dept(deptno)
                    VALUES(v_deptno);
         v_counter := v_counter + 1;
         v_deptno := v_deptno + 10;
         EXIT WHEN v_counter > 5;
     END LOOP;
     END;
/
```

--- FOR – LOOP - END LOOP

```
DECLARE
        v_deptno  dept.deptno%TYPE    := 50;
        v_counterinteger;
       …
 BEGIN
               FOR v_counter IN 1..5 LOOP
                INSERT INTO dept(deptno)
                      VALUES(v_deptno);
            v_deptno := v_deptno + 10;
       END LOOP;
     END;
/
```

--- WHILE – LOOP - END LOOP

```
DECLARE
          v_deptno    dept.deptno%TYPE    := 50;
       v_counterinteger;
      …
BEGIN
       v_counter := 1;
      WHILE v_counter <= 5 LOOP
           INSERT INTO dept(deptno)
                   VALUES(v_deptno);
           v_deptno := v_deptno + 10;
      END LOOP;
     …
END;
/
```

SEQUENTIAL CONTROL

- To alter the sequence GOTO statement is used
- **Syntax:**

```
        GOTO jumhere;
                :
                :<<jumphere>>
```

- **Example:**

```
 Begin
    dbms_output.put_line(‘Code starts’);
    GOTO jump;
```

SEQUENTIAL CONTROL

```
dbms_output.put_line(‘This statement is not
  executed…’);
<<jumphere>>  dbms_output.put_line(‘Flow
of Execution  jumped here …’);
End;
/
```

CURSORS

- A Cursor is an area in memory where the data required to execute SQL statement is stored.
- The data that is stored in the cursor is called the **Active Data Set.**
- The Size of the cursor will be same as a size to hold this data.
- The row that is being processed is called the **current Row.**
- A pointer known as a **Row pointer** is used to track the current row.

TYPES OF CURSORS

1. **Implicit Cursors**

   A cursors is called an Implicit cursor, if it is opened by Oracle itself to execute any SQL statement.

2. **Explicit Cursors**

A cursors is called an Explicit cursor, if it is opened by user to process through PL/SQL block.

1.Implicit Cursors

- A cursors is called an Implicit cursor, if it is opened by Oracle itself to execute any SQL statement.
- Oracle perform the following operations to manage an implicit cursor.

1. reserves an area in memory to store data required to execute SQL statement

2.populates this area with required data

3.processes data

4.Fress memory area

closes a cursor when processing is completed.

Attributes Of Implicit Cursors

Attributes	Description
SQL%ISOPEN	Always returns false
SQL%FOUND	If select found any record returns true else returns false.
SQL%NOTFOUND	If select no found any record returns true else returns false.
SQL%ROWCOUNT	Returns number of records processed by select, insert ,delete and update operation.

EXAMPLE

```
Declare
   branch Account.bname%TYPE;
Begin
   branch:=&branch;
UPDATE Account SET bname=UPPER(branch)
   WHERE bname=branch;
   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;
/
```

2.EXPLICIT CURSORS

- A cursors is called an Explicit cursor, if it is opened by user to process data through PL/SQL block.
- An Explicit cursor is used when there is a need to process more than one record individually.
- The step required to manage Explicit cursors

1.Declare a Cursors

2. Open a Cursors

3.Fetching Data

4.Processing Data

5.Closing Cursors

EXPLICIT CURSORS

1) **Declare a Cursor**

   Syntax:

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

   Example:

   ```
   CURSOR cAcc IS
   SELECT ano,balance,bname FROM Account;
   ```

2) **Open a Cursor**

   Syntax:

   ```
   Open   cursorName;
   ```

   Example:

   ```
   open cAcc;
   ```

EXPLICIT CURSORS

3) Fetching a cursors

   Syntax:

   FETCH CursorName INTO

   variable1,variable2..variableN;

   Example:

   FETCH CAcc INTO   no,bal,branch;

4) Processing Data

   This step involve actual processing of the table data.

   This step may involve various PL/SQL as well as
   SQL statements.

EXPLICIT CURSORS

5) Close a Cursor

Syntax:

```
Close cursorName;
```

Example:

```
Close CAcc;
```

EXAMPLE

```
Declare
   CURSOR cAcc is select ano,balance,bname from
   Account;
          no Account.ano%TYPE;  bal
   Account.balance%TYPE;  branch
   Account.bname%TYPE;
Begin
   Open cAcc;
   If cAcc%ISOPEN then
          loop
          Fetch cAcc into no,bal,branch;
          Exit when cAcc%NOTFOUND;
```

EXAMPLE

```
        If Branch='vvn' then
                insert into Acc_vvn values(no,bal);
                delete from Account where ano=no;
        end if;
        end loop;
  commit;
  else
  dbms_output.put_line('cursor can not be
  opened....');
  end if;
end;
/
```

EXAMPLE

```
 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;
/
```

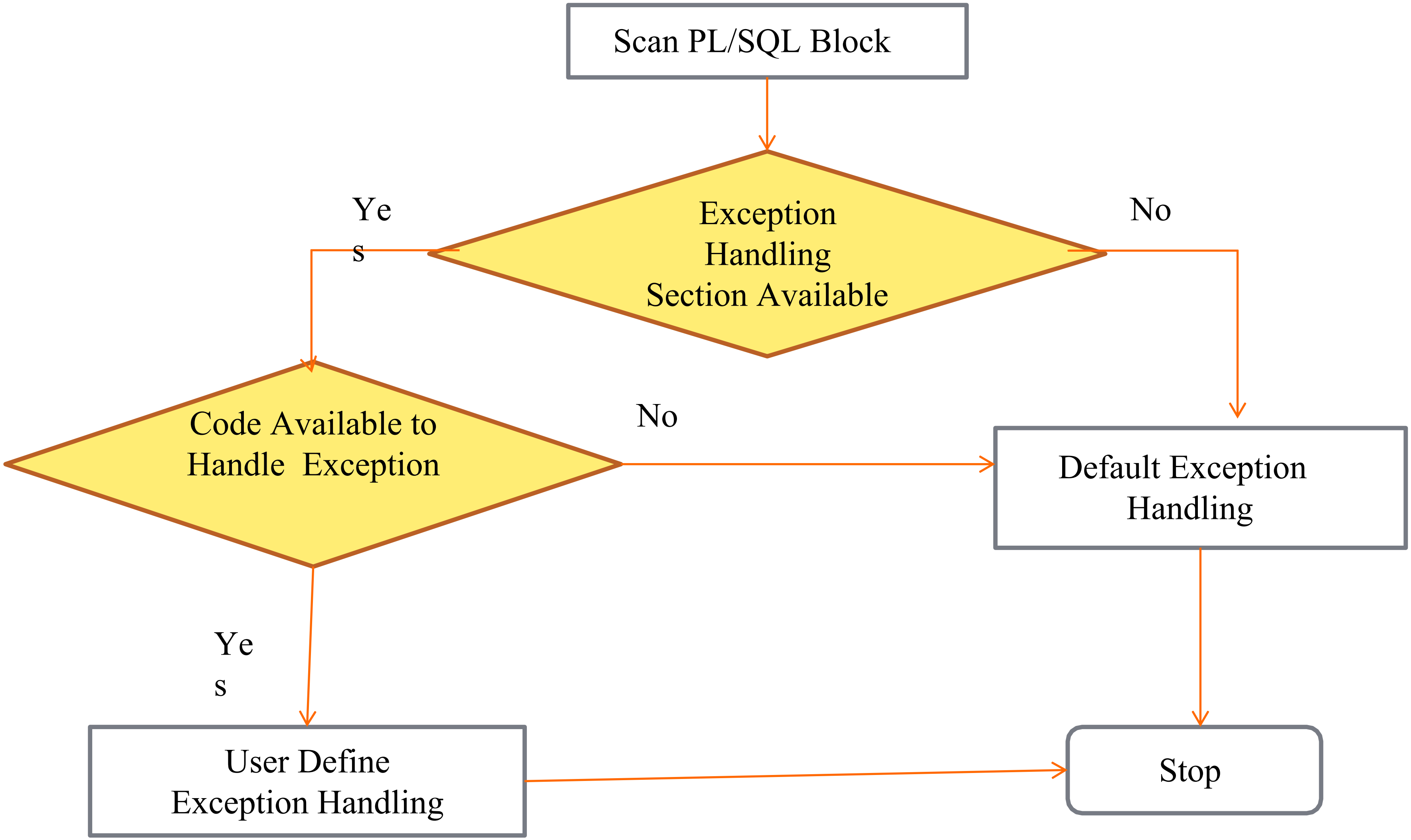
Exception Handling

- To handle various kinds of errors PL/SQL uses exception part.
- **Type of Exception**

1)Named Exception

2)Numbered Exception

3)User-define Exception


Scan PL/SQL Block
Exception Handling Section Available
Ye s
No
Code Available to Handle Exception
No
Default Exception Handling
Ye s
User Define Exception Handling
Stop

Working of the Exception Handler

NAMED EXCEPTION

- Some commonly occurring system exception are given name, and known as Named Exception.
- Oracle has defined 15 to 20 named Exception.
- Some of the named exception are listed below.
  1. DUP_VAL_ON_INDEX
  2. INVALID_CURSOR
  3. INVALID_NUMBER
  4. LOGIN_DENIED
  5. NO_DATA_FOUND
  6. NOT_LOGGED_ON

NAMED EXCEPTION

7. NOT_LOGGED_ON
8. PROGRAM_ERROR
9. TOO_MANY_ROWS
10. VALUE_ERROR
11. ZERO_DIVIDE
12. OTHER

Named Exception syntax

```
DECLARE
        exceptionName EXCEPTION;
  PRAGMA EXCEPTION_INIT(exceptionName,
  errorNumber);
  BEGIN
  ---EXECUTE Commands...
  EXCEPTION
        when exceptionName then
  ---code to handle Exception....
  END;
/
```

Named Exception example

```
Declare
        exNull EXCEPTION;
        PRAGMA EXCEPTION_INIT(exNull,-1400);
        no Account.ano%Type;
        bal Account.balance%Type;
        branch Accout.bname%Type;
Begin
        no :=&no;
        bal :=&bal;
        branch:='&branch';
        insert into Account values(no,bal,branch);
        commit;
```

Named Exception example

```
dbms_output.put_line('Record inserted
   successfully...');
Exception
   when DUP_VAL_ON_INDEX then
   dbms_output.put_line('Duplicate value found for
   primary key.');
          when exNull then


   dbms_output.put_line('Null value found for
   primary key.');
   end;
   /
```

NUMBERED EXCEPTION SYNTAX

```
 DECLARE
  exceptionName EXCEPTION;
  PRAGMA EXCEPTION_INIT(exceptionName,
  errorNumber);
  BEGIN
  ---EXECUTE Commands...
  EXCEPTION
     when exceptionName then
  ---code to handle Exception....
  END;
/
```

NUMBERED EXCEPTION EXAMPLE

```
Declare
        exNull EXCEPTION;
        PRAGMA EXCEPTION_INIT(exNull,-1400);
        no Account.ano%Type;
        bal Account.balance%Type;
        branch Accout.bname%Type;
Begin
        no :=&no;     bal :=&bal;      branch:='&branch';
        insert into Account values(no,bal,branch);
        commit;
        dbms_output.put_line('Record inserted
successfully...');
```

Numbered Exception Example

```
when DUP_VAL_ON_INDEX then
dbms_output.put_line('Duplicate value found for primary key.');
when exNull then
dbms_output.put_line('Null value found for primary key.');
end;
/
```

User define Exception

```
     Declare

            exNull EXCEPTION;
            PRAGMA EXCEPTION_INIT(exNull,-
  1400);

            myEx EXCEPTION            ----user define
  exception-----

            no Account.ano%Type;
            bal Account.balance%Type;
            branch Accout.bname%Type;
```

User define Exception

```
   Begin
        no :=&no; bal :=&bal;      branch:='&branch';  if
               bal<0 then
               raise myEx;
               end if;
        insert into Account values(no,bal,branch);
        commit;
        dbms_output.put_line('Record inserted
        successfully...');
Exception
```

User define Exception

```
when DUP_VAL_ON_INDEX then
   dbms_output.put_line('Duplicate value found for
   primary key.');

when exNull then
   dbms_output.put_line('Null value found for primary  key.');


when myEx then
   dbms_output.put_line('balance can not be negative
   value.');
end;
/
```

PROCEDURES AND FUNCTION

- A procedure or function is a group or set of SQL and PL/SQL statement that perform a specific task.
- A procedure or function is a named PL/SQL block of code.
- This block can be compiled and successfully compiled block can be stored in oracle database.
- so that is called stored procedure and function.

STRUCTURE OF PROCEDURES

Declaration

--variable declaration---

Executable commands

---statement of SQL and PL/SQL

Exception Handling

---handle exception or error--

Syntax of Procedures

```
Create or Replace PROCEDURE ProcedureName
                    (argument [IN,OUT,IN OUT] datatype,..)
IS
        <declaration section>
Begin  --mandatory ---
<executable commands>
Exception  --optional---
<Exception Handling>
End;    ---Mandatory---
```

Example of Procedures

```
create or replace procedure debitAcc
          ( no IN Account.ano%Type,amount IN Number)
   IS
          balAccount.balance%TYPE;
          NewBalanceAccount.balance%TYPE;
Begin
select balance into bal from Account where ano=no;
                    NewBalance:=bal-amount;
   update Account Set balance=NewBalance where
   ano=no;
dbms_output.put_line('Account'||no||'debited...');  End; /
```

SYNTAX OF FUNCTION

```
CREATE OR REPLACE FUNCTION Functionname
(argument IN datatype …)
RETURN datatype
IS
<Declaration Section>
BEGIN
<Executable Commands>
EXCEPTION
<Exception Handling>
END;
```

Example of Function

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

Execution of Procedure and Function

For procedure

```
EXEC debitAcc(‘A01’,1000);
```

For function

```
select getBalance(‘A01’) from dual;
```

ADVANTAGES OF PROCEDURE AND FUNCTION

- **Security**
- **Faster execution**
- **Sharing of code**
- **Productivity**
- **Integrity**

Procedure vs. Function

Function	Procedure
A function mast return a value.	A procedure can also return value but not mandatory.
A function can return only one value.	A procedure can return more than one value.
A function use **select** command for execution of it.	A procedure can not use **select** command for execution of it.
A function can not use **EXEC** command for execution of it.	A procedure use **EXEC** command for execution of it.

PACKAGE

- A package a container for other database is objects.
- A package can hold other database objects such as variables, constants, cursors, exception, procedure, function and sub-programs.
- It is one kind of database object.

STRUCTURE OF PACKAGE

1. **Package Specification**

```
    CREATE OR REPLACE PACKAGE packagename  IS
                        --Package specification. . .
                 END packagename;
```

2. **Package Body**

```
 CREATE OR REPLACE PACKAGE BODY
    packagename
                        IS
                               --Package body. . .
                        END packagename;
```

PACKAGE SPECIFICATION

```
CREATE OR REPLACE PACKAGE transaction

IS

(No IN Account.ano%TYPE, Amount IN NUMBER)  (No
IN Account.ano%TYPE) RETURN NUMBER

END transaction;
            /
```

PACKAGE BODY

```
CREATE OR REPLACE PACKAGE transaction  IS
        --define procedure ‘debitAcc’
 CREATE OR REPLACE PROCEDURE debitAcc
(no IN Account.ano%TYPE, Amount IN NUMBER)  IS
        bal Account.balance%TYPE;
        newbal Account.balance%TYPE;
BEGIN
        SELECT balance INTO bal FROM Account
        WHERE ANO = NO:
        newbalance := bal –amount;
        dbms_output.put_line (‘Account
’||no||’debited);
        END;
```

PACKAGE BODY

```
--define function ‘getBalance’
CREATE OR REPLACE FUNCTION getBalance
   (No IN Account.ano%TYPE)
RETURN NUMBER
IS
          BAL Account.balance%TYPE;
BEGIN
   SELECT balance INTO BAL FROM Account
         WHERE ANO = NO:
                    RETURN BAL;
END;
END transaction;
/
```

RUN OR REFERENCING A PACKAGE SUBPROGRAM

For procedure

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

For function

```
select transaction.getBalance(‘A01’) from dual;
```

DESTROYING A PACKAGE

Syntax:

```
drop package[Body] packageName;
```

Example:

```
drop package transaction;
```

TRIGGERS

- Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events:
- A database manipulation (DML) statement (DELETE, INSERT, or UPDATE).
- A database definition (DDL) statement (CREATES, ALTER, or DROP).
- A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).
- Triggers could be defined on the table, view, schema, or database with which the event is associated.

SYNTAX OF TRIGGERS

```
CREATE OR REPLACE TRIGGER
   triggername

[BEFORE / AFTER]
                    [INSERT, DELETE, UPDATE [OF
   column]]
                   ON Tablename  [REFERENCING
                   [OLD AS old,
   NEW AS new] ]
```

SYNTAX OF TRIGGERS

```
              [FOR EACH ROW [WHEN
  condition]
                  IS
                       <Declaration Section>
                  BEGIN
                       <Executable
  Commands>
                  EXCEPTION
                       <Exception Handling>
                  END;
/
```

EXAMPLE OF TRIGGER

```
CREATE OR REPLACE TRIGGER Invalid_bal
BEFORE INSERT
ON Account
FOR EACH ROW
BEGIN
IF NEW. Balance < 1000 THEN
        dbms_output.put_line (‘Balance is not
  sufficient…’);
END IF;
END;
/
```

GTU IMPORTANT QUESTIONS

1. Differentiate : SQL and PL/SQL.
2. Write short note on Structure of PL/SQL block.
3. Explain Anchored data type with example.
4. Find out maximum value out of given three numbers.
5. What is explicit cursor? Explain various steps to manage it.
6. Explain different types of explicit cursor.
7. State the meaning of PRAGMA EXCEPTION_INIT.
8. Explain error handling using example.
9. Display three account having top three highest balance.

GTU IMPORTANT QUESTIONS

11. Write short note on : Stored Procedure.
12. Explain procedure in detail with example.
13. Define package. Write steps to create package in PL/SQL.
14. What is trigger? Explain advantages and types of triggers.
15. Explain RAISE_APPLICATION_ERROR.

Thank you