PL/SQL

PL/SQL stands for Procedural Language extension of SQL.

PL/SQL is a combination of SQL along with the procedural features of programming languages.

Properties of programming as well as the great interaction with database.

PLSQL IS NOT A CASE SENSITIVE LANG.

COMMENTS IN PLSQL

The PL/SQL compiler ignores comments but you should not.

Single-line comments begin with a double hyphen (--)

Multiline comments begin with a slashasterisk (/*), end with an asterisk-

WHERE AND HOW TO RUN ORACLE PL/SQL IN WINDOWS ?

- YOU HAVE ORACLE 9i/10g/11g in your system.
- THEN FOLLOWS THESE STEPS
- 1) OPEN SQL PROMPT

- USERNAME :user
- PASSWORD:user

Basic construct of a PL/SQL Program

```
DECLARE
/* Variables can be declared here */
BEGIN
/* Executable statements can be written here */
EXCEPTION
/* Error handlers can be written here. */
END;
```

IMPORTANT PL SQL CONCEPTS

DECLARE :- if you want to decalre a variable in plsql program then it takes place in declare section

BEGIN:- is used to start the working of program and end is used to terminate the begin.

Delimiter is used to run (/)

WHAT TO DO PREVIOUSLY

- SET SERVEROUTPUT ON ; is run before every time when you compiled a program in a session.

- **To write program, use Notepad through Oracle using ED command.**

- SQL> ED ProName
- Type the program Save & Exit
- To Run the program

- SQL> @ProName

To Display something on Screen

- DBMS_OUTPUT.PUT_LINE command for e.g. if sal=10 and you want to print it
- Then it looks like
- dbms_output.put_line('the salary is ' ||sal);

Value assign in variable

```
Declare
Num number(11);
Begin
Num:=5;
```

User’s input for a variable

```
DECALRE
N NUMBER(11);
BEGIN
N:=&N;
DBMS_OUTPUT.PUT_LINE(‘THE VALUE IS ‘||N);
END;
/
```

Sample program to print your ‘hello world’

```
BEGIN
Dbms_output.put_line(‘hello world’);
End;
/
```

/ is used to terminate plsql program called as delimeter

Ex :- PL/SQL to find addition of two numbers

```
DECLARE
  A INTEGER := &A; B INTEGER := &B; C INTEGER;
BEGIN
   C :=  A + B;
    DBMS_OUTPUT.PUT_LINE('THE SUM IS '||C);
   END;
/
```

Use of Commit, Savepoint & Rollback in PL/SQL

- We can use these commands in PL/SQL. We can use any (Insert, Delete or Update)
- operations for Savepoint & Rollback.
- The following program inserts a record into Sailors table then updates a record before we Commit a Savepoint is defined and we can use Rollback to undo the operations we have done after the Savepoint (i.e. deleting a Sailors record is undone). We can define number of Savepoint statements in a program and Rollback to any point.
- BEGIN
- INSERT INTO SAILORS VALUES('32','HEMANT',10, 30);
- SAVEPOINT S1;
- DELETE FROM SAILORS WHERE SID='22';

- ROLLBACK TO S1; COMMIT;
- END;
- /

IF STATEMENT

IF STATEMENT WORKS AS SIMILAR AS C OR C++

Common syntax

```
IF condition THEN
statement 1;
ELSE
statement 2;
END IF;
```

For Nested IF—ELSE Statement we can use IF--ELSIF—ELSE as follows

```
IF(TEST_CONDITION) THEN
SET OF STATEMENTS
ELSIF (CONDITION)THEN
 SET OF STATEMENTS
END IF;
```

Conditional statement IF then else

```
DECLARE
Age number(11);
Begin
Age:=&age;
If age>18 then
Dbms_output.put_line('u can vote');
Else
Dbms_output.put_line('u cannot vote');
End if;
End;
/
```

USE OF IF WITH SQL TABLE

```
Declare
A number(11);
Begin
Select salary into a from emp where name='ram';
If a>1000 then
Update emp set bonus=bonus+1000 where name='ram';
Else
Update emp set bonus=bonus+500 where name='ram';
End if;
End;
/
```

To print Pat salary from employees table using pl program

- Declare
- n number(11);
- Begin
- Select salary **<u>into</u>** n from employees where first_name='Pat';
- Dbms_output.put_line('the Pat sal is ' ||n);
- End;
- /

INTO COMMAND

INTO command is used to catch a value in variable from table.

Only one value must be returned

For e.g. in the above example if there are two people who’s name is john then it shows error

Some Points regarding SELECT --- INTO

- We have to ensure that the SELECT….INTO statement should return one & only one row. If no row is selected then exception NO_DATA_FOUND is raised. If more than one row is selected then exception TOO_MANY_ROWS is raised.
- To handle the situation where no rows selected or so many rows selected we can use
- Exceptions. We have two types of exception, User-Defined and Pre-Defined Exceptions.

Program with User-Defined Exception

- DECLARE
- N INTEGER:=&N; A EXCEPTION;
- B EXCEPTION; BEGIN
- IF MOD(N,2)=0 THEN RAISE A;
- ELSE RAISE B; END IF; EXCEPTION
- WHEN A THEN
- DBMS_OUTPUT.PUT_LINE('THE INPUT IS EVEN.....');
  WHEN B THEN
- DBMS_OUTPUT.PUT_LINE('THE INPUT IS ODD.....');
  END;
- /

Program with Pre-Defined Exception

```
DECLARE
SID VARCHAR2(10);
BEGIN
SELECT SID INTO SID FROM SAILORS WHERE
   SNAME='&SNAME'; DBMS_OUTPUT.PUT_LINE(SID);
EXCEPTION
WHEN NO_DATA_FOUND THEN
   DBMS_OUTPUT.PUT_LINE('No Sailors with given SID
   found'); WHEN TOO_MANY_ROWS THEN
DBMS_OUTPUT.PUT_LINE('More than one Sailors with
   same name found');
END;
/
```

LOOPS IN PLSQL

1) SIMPLE LOOP
2) WHILE LOOP
3) FOR LOOP

GOTO STATEMENTS

<<LABEL>>

SET OF STATEMENTS GOTO LABEL;

- **<u>FOR LOOP</u>**

```
FOR <VAR> IN [REVERSE]
   <INI_VALUE>..<END_VALUE>
SET OF STATEMENTS
END LOOP;
```

- **<u>WHILE LOOP</u>**

```
WHILE (CONDITION)
LOOP
SET OF STATEMENTS
END LOOP;
```

- **<u>LOOP STATEMENT</u>**

```
LOOP
 SET OF STATEMENTS
IF (CONDITION) THEN
EXIT
 SET OF STATEMENTS
END LOOP;
```

FOR LOOP

- Print number from 1 to 10 using for loop

- BEGIN
- FOR i in 1 ..10 loop
- Dbms_output.put_line(i);
- End loop
- End;
- /
- (For has NO need to initialize explicitly but it need in while)

While loop

- PRINT NUMBERS FROM 1 TO 10 USING WHILE LOOP

- Declare
- i number(3):=0;
- Begin
- While i<=10 loop
- i:=i+1;
- Dbms_output.put_line(i);
- End loop;
- End;
- /

SIMPLE LOOP

- LOOP
- Statement 1;
- Statement 2;
- Exit condition
- End loop;

Error Handling using RAISE_APPLICATION_ERROR

- Procedure RAISE_APPLICATION_ERROR is used to generate user-defined errors in the PL/SQL. The general syntax is
- RAISE_APPLICATION_ERROR(ErrorCode, Error_Message [, TRUE/FALSE]); The valid Error_Code is in range from –20000 to –20999.
- The Error_Message length is maximum 2048 bytes.
- The optional third parameter TRUE indicates that error message is put in stack. If FALSE
- is mentioned then error replaces all previous errors.

Example

```
DECLARE
A INTEGER:=&A;
B INTEGER:=&B;
C INTEGER;
BEGIN IF
(B=0)THEN
RAISE_APPLICATION_ERROR(-20001,'DIVISION BY
   ZERO'); ELSE
 C:=A/B;
DBMS_OUTPUT.PUT_LINE('RESULT IS :'||C); END IF;
END;
/
```

FUNCTIONS

- A function is a named PL/SQL Block which is similar to a procedure. The major difference between a procedure and a function is, a function must always return a value, but a procedure may or may not return a value.

FUNCTION EXAMPLE

- Create function funname
- Return number is
- a number(10);
- Begin
- Select avg(sal) into a from emp;
- return a;
- End;
- /

HOW TO EXECUTE FUNCTION ?

1) SELECT FUNCTIONNAME FROM DUAL;

2)DBMS_OUTPUT.PUT_LINE(FUNCTIONNAME);

- -- Assume file name Fun
- CREATE OR REPLACE FUNCTION FUN(A NUMBER) RETURN NUMBER IS
- BEGIN
- RETURN (A*A);
- END FUN;
- /
-

- -- Assume file name testFun
- DECLARE
- X NUMBER:=&X; S NUMBER; BEGIN S:=FUN(X);
- DBMS_OUTPUT.PUT_LINE('SQUARE OF A NUMBER'|| S);
- END
- /

- **Output**

SQL> @Fun

Function created. SQL> @testFun

ENTER VALUE FOR X: 10

OLD 2: X NUMBER:=&X; NEW 2: X NUMBER:=10; SQUARE OF A NUMBER100

PL/SQL procedure successfully completed.

STORED PROCEDURE

- SOMETHING LIKE FUNCIONS IN C/C++.
- A **stored procedure** is a subroutine available to applications that access a relational database system. A stored procedure (sometimes called a **proc**, **sproc**, **StoPro**, **StoredProc**, **sp** or **SP**) is actually stored in the database data dictionary.
- A procedure is similar to an anonymous PL/SQL Block but it is named for repeated usage.

A procedure may or may not return any value

Common syntax

- CREATE [OR REPLACE] PROCEDURE procedure_name
- [(parameter [,parameter])]
- IS
- [declaration_section]
- BEGIN
- executable_section
- [EXCEPTION exception_section]
- END [procedure_name];

EXAMPLE WITHOUT PARAMETER

- **CREATE OR REPLACE PROCEDURE** MYSTPROC **IS**
- **BEGIN**
- DBMS_OUTPUT.PUT_LINE('Hello World!');
- **END**;
- /

When you create a procedure or function, you may define parameters. There are three types of parameters that can be declared

1)IN
2)OUT
3)IN OUT

PARAMETER TYPES

- **1) IN type parameter:** These types of parameters are used to send values to stored procedures.
  2) OUT type parameter: These types of parameters are used to get values from stored procedures. This is similar to a return type in functions.
  3) IN OUT parameter: These types of parameters are used to send values and get values from stored procedures.

For writing Procedures we can directly type at SQL prompt or create a file.

SQL> ed File_Name

- **Type & save procedure.**
- **To create Procedure (before calling from other program.)**

SQL> @File_Name

- **To use/call procedure, write a PL/SQL code and include call in the code using**

Pro_Name(Par_List);

Or you can execute from SQL Prompt as

SQL>execute Pro_Name(Par_List)

For dropping Procedure/Function (Function described next)

SQL>DROP PROCEDURE Pro_Name;

SQL>DROP FUNCTION Fun_Name;

1) IN PARAMETER

```
  -- Assume file name P1
CREATE OR REPLACE PROCEDURE P1(A NUMBER) AS
 BEGIN
DBMS_OUTPUT.PUT_LINE('A:'||A);
END P1;
/
```

Now write PL/SQL code to use procedure in separate file.

```
-- Assume file name testP1
DECLARE
BEGIN
P1(100);
END;
/
```

Output SQL> @P1

Procedure created. SQL>@testP1

A:100

PL/SQL procedure successfully completed.

2) OUT Parameter

```
-- Assume file name P3
CREATE OR REPLACE PROCEDURE P3(A OUT
   NUMBER) AS
BEGIN
A:=100;
DBMS_OUTPUT.PUT_LINE('A:'|| A);
END P3;
/
```

- --Assume file name testp3
- DECLARE
- X NUMBER;
- BEGIN
- X:=50;
- DBMS_OUTPUT.PUT_LINE(‘X:'|| X);
- P3(X);
- DBMS_OUTPUT.PUT_LINE(‘X:'|| X);
- END;
- /

Output

SQL> @P3

Procedure created. SQL>@testP3

X:50

A:100

X:100

PL/SQL procedure successfully completed.

DIFF B/W PROCEDURE AND FUNCTION

- The functions can return only one value and procedures not. Functions can be call from SQL Statements, procedures not and there are some things that you can
do in a stored procedure that you can not do in a function.

FOR E.G.

- Create a stored procedure that adds 1000 to each employees commission watch for Null values
- Create procedure st_proc as
- Begin
- Update emp set comm=nvl(comm,0)+1000;
- End;
- /

- The **NVL function** is used to replace NULL values by another value. value_in if the **function** to test on null values. The value_in field can have a datatype char, varchar2, date or number datatype.

Procedure can call at any time using

- Execute st_proc; -------→ procedure name
- OR
- Exec st_proc;----------→procedure name

CURSORS

- A cursor is a temporary work area created in the system memory when a SQL statement is executed. A cursor contains information on a select statement and the rows of data accessed by it.
- Cursors provide a way for your program to select multiple rows of data from the database and then to process each row individually.
- There are two types of cursors in PL/SQL:
- **1)IMPLICIT CURSORS**
- **2) EXPLICIT CURSORS**

A CURSOR CAN HOLD MORE THAN ONE ROW, BUT CAN PROCESS ONLY ONE ROW AT A TIME.

Implicit Cursors

- These are created by default when DML statements like, INSERT, UPDATE, and DELETE statements are executed.
- The set of rows returned by query is called active set.
- Oracle provides few attributes called as implicit cursor attributes to check the status of DML operations. The are as follows
- ***1) %FOUND***
- ***2) %NOTFOUND***
- ***3) %ROWCOUNT***
- ***4) %ISOPEN***.

FOR E.G.

```
DECLARE
n number(5);
BEGIN
UPDATE emp SET salary = salary + 1000;
 IF SQL%NOTFOUND THEN
dbms_output.put_line('No sal are updated');
ELSIF SQL%FOUND THEN
n := SQL%ROWCOUNT;
 dbms_output.put_line('Sal for ' ||n || 'employees are updated');
END IF;
 END;
/
```

EXPLANATION

- ***%FOUND->***The return value is TRUE, if the DML statements like INSERT, DELETE and UPDATE affect at least one row and if SELECT ….INTO statement return at least one row.
- ***%NOTFOUND-***>same as above but if not found.
- ***%ROWCOUNT*** ->Return the number of rows affected by the DML operations

Explicit Cursors

- Explicit cursors are declared by and used by user to process multiple rows ,returned by select statement.
- An **explicit cursor** is defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row. We can provide a suitable name for the cursor.

Explicit cursor management

- 1)Declare the cursor
- 2)Open the cursor
- 3)Fetch data from cursor
- 4)Close the cursor

Declaring the cursor

- CURSOR cursor_name IS select_statement;
- For e.g.
- Cursor cursor_name is
- Select name from emp where dept='maths'

Opening the cursor

- Open cursor_name
- For e.g.
- Open c_name
- Where c_name is the name of cursor.

Fetching data from cursor

- The fetch statement retrieves the rows from the active set one row at a time. The fetch statement is used usually used in conjunction with iterative process (looping statements)
- Syntax: FETCH cursor-name INTO record-list
- Ex: LOOP
- ----------
- ----------
- FETCH c_name INTO name;
- ----------
- END LOOP;

Closing a cursor:

- Closing statement closes/deactivates/disables the previously opened cursor and makes the active set undefined.
- Syntax : CLOSE cursor_name

Cursor can store multiple rows at a time but without loop cursor cannot fetch multiple rows but only print very first row from your result e.g. on next slide

Without loop example

- declare
- a emp%rowtype;
- cursor cc is ----------→cursor name
- select * from emp where sal>1000;
- begin
- open cc;----------→open cursor
- fetch cc into a;----------→fetch cursor
- dbsm_output.put_line(a.ename || a.job);--→print multiple rows
- close cc;
- end;
- /
- output:-allen salesman

USING LOOP FOR FETCHING MULTIPLE ROWS THROUGH CURSORS

- declare
- cursor qaz is
- select ename,job from emp;
- a qaz%rowtype;----------→a of cursor type variable
- begin
- open qaz;----------→open cursor
- loop
- fetch qaz into a;----------→fetch cursor
- exit when qaz%notfound;----------→exit when not found
- dbms_output.put_line(a.ename || a.job);
- end loop;
- end;
- /

Another Cursor example

- The HRD manager has decided to raise the salary for all the employees in the physics department by 0.05 whenever any such raise is given to the employees, a record for the same is maintained in the emp_raise table (the data table definitions are given below). Write a PL/SQL block to update the salary of each employee and insert the record in the emp_raise table.

Tables

- **Table: employee**
- emp_code varchar (10)
- emp_name varchar (10)
- dept varchar (15)
- job varchar (15)
- salary number (6,2)
- **Table: emp_raise**
- emp_code varchar(10)
- raise_date Date
- raise_amt Number(6,2)

- DECLARE
- CURSOR c_emp IS ----------→cursor name
- SELECT emp_code, salary FROM employee----→query which stored in cursor
- WHERE dept = 'physics';
- a employee.emp_code %TYPE;-------→variable declare
- b employee.salary %TYPE;
- BEGIN
- OPEN c_emp;----------→open cursor
- LOOP----------→fetching records using loop
- FETCH c_emp INTO a, b;-----→ fetching records and stored in
- UPDATE employee SET salary : = b + (b* 0.05)
- WHERE emp_code = str_e;
- INSERT INTO emp_raise
- VALUES (str_emp_code, sysdate, num_salary * 0.05);
- END LOOP;
- Commit;
- CLOSE c_emp;
- END;

Triggers

- A procedure that starts automatically if specified changes occur to the DBMS
- A trigger is a pl/sql block structure which is fired when a DML statements like Insert, Delete, Update is executed on a database table. A trigger is triggered automatically when an associated DML statement is executed.

Triggers (Active database)

- Three parts:
  - Event (activates the trigger)
  - Condition (tests whether the triggers should run) [Optional]
  - Action (what happens if the trigger runs)

- Semantics:
  - When event occurs, and condition is satisfied, the action is performed.

Triggers – Event,Condition,Action

- Events could be :

```
BEFORE|AFTER INSERT|UPDATE|DELETE ON <tableName>
```

e.g.: `BEFORE INSERT ON Professor`

- Condition is SQL expression or even an SQL query (query with non-empty result means TRUE)

- Action can be many different choices :
  - SQL statements , even DDL and transaction-oriented statements like “commit”.

Trigger Syntax

CREATE TRIGGER <triggerName>

BEFORE|AFTER INSERT|DELETE|UPDATE

[OF <columnList>] ON <tableName>|<viewName>

[REFERENCING [OLD AS <oldName>] [NEW AS <newName>]]

[FOR EACH ROW] (default is “FOR EACH STATEMENT”)

[WHEN (<condition>)]

<PSM body>;

Example Trigger

Assume our DB has a relation schema :Pfessor (pNum, pName, salary)

We want to write a trigger that :

Ensures that any new professor inserted has salary >= 60000

Example Trigge

Event

```
CREATE TRIGGER minSalary "BEFORE
  INSERT" ON Professor

      for what conte
BEGIN
    check for violation here ?

END;
```

Condition

Example Trigger

```
CREATE TRIGGER minSalary BEFORE
  INSERT ON Professor

      FOR EACH ROW

BEGIN

    Violation of Minimum Professor
  Salary?

END;
```

trigger is performed for each row inserted

Example Trigger

```
CREATE TRIGGER minSalary BEFORE
  INSERT ON Professor
      FOR EACH ROW
BEGIN
  IF (:new.salary < 60000)
    THEN RAISE_APPLICATION_ERROR
  (-20004,  'Violation of Minimum
  Professor Salary');
  END IF;

END;
```

Example trigger

```
CREATE TRIGGER minSalary BEFORE INSERT ON Professor
  FOR EACH ROW

DECLARE temp int;        -- dummy variable not
  needed

BEGIN
  IF (:new.salary < 60000)
      THEN RAISE_APPLICATION_ERROR (-20004,
            'Violation of Minimum Professor
  Salary');
  END IF;

temp := 10;        -- to illustrate declared
  variables

END;
```

Details of Trigger Example

- BEFORE INSERT ON Professor
  - This trigger is checked before the tuple is inserted
- FOR EACH ROW
  - specifies that trigger is performed for each row inserted
- :new
  - refers to the new tuple inserted
- If (:new.salary < 60000)
- then an application error is raised and hence the row is not inserted; otherwise the row is inserted. Use error code: -20004;
  - this is in the valid range

Example Trigger Using Condition

Condition

```
CREATE TRIGGER minSalary BEFORE INSERT ON Professor
FOR EACH ROW
WHEN (new.salary < 60000)
BEGIN
   RAISE_APPLICATION_ERROR (-20004,          'Violation
   of Minimum Professor Salary');
END;
```

- Conditions can refer to old/new values of tuples modified by the statement activating the trigger.

Triggers: REFERENCING

```
CREATE TRIGGER minSalary BEFORE INSERT ON Professor

REFERENCING NEW as newTuple

FOR EACH ROW

WHEN (newTuple.salary < 60000)

BEGIN
  RAISE_APPLICATION_ERROR (-20004,
  'Violation of Minimum Professor Salary');
END;
```

Example Trigger

```
CREATE TRIGGER minSalary
     BEFORE UPDATE ON Professor
REFERENCING OLD AS oldTuple NEW as newTuple
FOR EACH ROW
WHEN (newTuple.salary < oldTuple.salary)
BEGIN
   RAISE_APPLICATION_ERROR (-20004, 'Salary
   Decreasing !!');
END;
```

- Ensure that salary does not decrease

Another Trigger Example

```
CREATE TRIGGER  youngSailorUpdate
   AFTER  INSERT ON SAILORS
REFERENCING NEW TABLE AS NewSailors
FOR EACH STATEMENT
   INSERT
        INTO YoungSailors(sid, name, age, rating)
        SELECT sid, name, age, rating
        FROM NewSailors N
        WHERE N.age <= 18
```

Types of Trigger

Defined by triggering transaction & level at which the trigger is executed

This section describes the different types of triggers:

- Row level triggers
  - Trigger once for "each row" in a transaction.
- Statement level triggers
  - Triggers execute once for "each transaction".
- Before and after triggers
  - Triggers can be executed immediately before and after INSERT, UPDATE and DELETE.

Row vs Statement Level Trigger

- Example: Consider a relation schema

  Account (num, amount)

  where we will allow creation of new accounts

  only during normal business hours.

Example: Statement level trigger

```
CREATE TRIGGER MYTRIG1
BEFORE INSERT ON Account
FOR EACH STATEMENT                --- is default
BEGIN
   IF (TO_CHAR(SYSDATE,'dy') IN ('sat','sun'))
   OR
   (TO_CHAR(SYSDATE,'hh24:mi') NOT BETWEEN '08:00' AND
  '17:00')
    THEN
      RAISE_APPLICATION_ERROR(-20500,'Cannot   create
  new account now !!');
   END IF;
END;
```

Some Points about Triggers

- Check the system tables :
  - `user_triggers`
  - `user_trigger_cols`
  - `user_errors`

- `ORA-04091`: mutating relation problem
  - In a **row level trigger**, you **cannot** have the body refer to the table specified in the event

- Also `INSTEAD OF` triggers can be specified on views

To Show Compilation Errors

```
SELECT line, position, text
FROM user_errors
WHERE name = 'MY_TRIGGER'
    AND TYPE = 'TRIGGER‘
```

- In SQL*Plus, you can also use the following shortcut:

“SQL> SHOW ERRORS TRIGGER MY_TRIGGER”

Mutating Trigger

“Trigger that triggers itself”

Constraints versus Triggers

- Constraints are useful for database consistency
  - More opportunity for optimization
  - Not restricted into insert/delete/update

- Triggers are flexible and powerful
  - Alerts
  - Event logging for auditing
  - Security enforcement
  - Analysis of table accesses (statistics)
  - Workflow and business intelligence …

- But can be hard to understand ……
  - Several triggers (Arbitrary order → unpredictable !?)
  - Chain triggers (When to stop ?)
  - Recursive triggers (Termination?)

Purpose of Triggers

- To generate data automatically
- To enforce complex integrity constraints
- To customize complex security authorization
- To maintain replicate tables
- To audit data modifications.

Type of triggers

- Row Triggers and Statement Trigger BEFORE and AFTER Triggers

TRIGGER RESTRICTION IS OPTIONAL (WHEN CLAUSE)

ONE TRIGGER MAY FIRE ANOTHER DATABASE TRIGGERS

- Create trigger abcd
- Before insert or update of sal on emp
- For each row
- when(new.sal>3000)
- begin
- :new.mgr:=1000;
- end;
- /

Explanation of last example

- If sal of any employee is updated and greater than 3000 then whose mgr values set to 1000.

SUPPOSE WE HAVE TWO TABLES ONE IS PRODUCT AND OTHER IS ORDER LIKE BIG BAZAR

PRODUCT AND ORDER TABLES

PNAME	PID	QTY

OPID	DESCRIPTIO	

- If qty from product table fall within 100 then automatically an order of that product is placed in order table.

- Create trigger abcd
- After update of qty on product
- For each row
- When(new.qty<100)
- Begin
- Insert into order values(:new.pid);
- End;
- /

EXCEPTION HANDLING
WHAT IS EXCEPTION …?

- AN EXCEPTION IS AN ERROR PL/SQL THAT IS RAISED DURING PROGRAM EXECUTION
- AN EXCEPTION CAN BE RAISED BY
- 1) IMPLICITLY BY THE ORACLE SERVER
- 2) Explicitly by the program

Type of Exception

- There are 3 types of Exceptions.

  a) Named System Exceptions
  b) Unnamed System Exceptions
  c) User-defined Exceptions

1) Named System Exceptions

- System exceptions are automatically raised by Oracle, when a program violates a RDBMS rule.
- For e.g.
- 1)CURSOR_ALREADY_OPEN
- 2)NO_DATA_FOUND
- 3)TOO_MANY_ROWS
- 4)ZERO_DIVIDE

TOO_MANY_ROWS EXAMPLE

- SUPPOSE YOU WANT TO RETRIEVE ALL EMPLOYEES WHOSE NAME='JOHN'
- DECLARE
- a varchar(12)
- SELECT LAST_NAME into a from employees where first_name='john'
- Dbms_output.put_line('john last name is ' ||a);
- End;
- /

But if too many people have first_name='john' then using exception handling

- DECLARE
- a varchar(12)
- SELECT LAST_NAME into a from employees where first_name='john'
- Dbms_output.put_line('john last name is ' ||a);
- End;
- /
- Exception
- When too_many_rows then
- Dbms_output.put_line('your st. gets many rows ');
- End;
- /

2)Unnamed System Exceptions

- Those system exception for which oracle does not provide a name is known as unamed system exception
- There are two ways to handle unnamed sysyem exceptions:
  1. By using the WHEN OTHERS exception handler, or
  2. By associating the exception code to a name and using it as a named exception

Unnamed System Exceptions CONT..

- We can assign a name to unnamed system exceptions using a **Pragma** called **EXCEPTION_INIT.**
  EXCEPTION_INIT will associate a predefined Oracle error number to a programmer_defined exception name.

FOR E.G.

```
DECLARE
AAA EXCEPTION; -------→AAA IS EXCEPTION NAME
 PRAGMA --------------------→USE TO DEFINE UNMANED EXCEPTION
  EXCEPTION_INIT (AAA, -2292); -------------------------→MUST BE VALID EXCEPTION NUMBER
BEGIN
 Delete FROM SUPPLIER where SUPPLIER_ID= 1;
EXCEPTION
  WHEN AAA
  THEN Dbms_output.put_line(' $$Child records are present for this product_id.');
END;
/
```

WHAT HAPPENS IN PREVIOUS EXAMPLE

- IF U WANT TO DELETE SUPPLIER_ID=1 THEN AN EXCEPTION OCCURS WHICH WILL PRINT AS WHICH IS IN DBMS_OUTPUT.
- ACTUALLY THIS ECEPTION WORKS ON PARENT CHILD DELETION AND THE ERROR NUMBER SIGNIFIES THE TYPE OF ERR AND FOR USER EASE WE MAKE A USEFUL OR UNDERSTANDABLE PRINT STATEMENT

3) User-defined Exceptions

- Apart from sytem exceptions we can explicity define exceptions based on business rules. These are known as user-defined exceptions.

- DECLARE
- my-exception EXCEPTION;
- ----
- ----
- Raise name_of_exception;

FOR E.G.

- DECLARE
- ----
- Zero_commission Exception;
- BEGIN
- IF commission=0 THEN
- RAISE zero_commission
- EXCEPTION
- WHEN zero_commission THEN
- Process the error
- END;

For example
When the user enters an invalid ID, the exception invalid_id is raised

```
DECLARE
  c_id customers.id%type := &cc_id;-------------→input id at run time
  c_name  customers.name%type; --------------→c_name variable as same datatype
  c_addr customers.address%type;                    as customer name datatype

  -- user defined exception
  ex_invalid_id  EXCEPTION;------------→exception name
BEGIN
  IF c_id <= 0 THEN
    RAISE ex_invalid_id;----------------→raise user condition
  ELSE
    SELECT  name, address INTO  c_name, c_addr
    FROM customers
    WHERE id = c_id;
```

```
DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
    DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
  END IF;
EXCEPTION
  WHEN ex_invalid_id THEN -------→user exception
    dbms_output.put_line('ID must be greater than zero!');
  WHEN no_data_found THEN      -------→predefined exception
    dbms_output.put_line('No such customer!');
  WHEN others THEN----------------→predefined exception
    dbms_output.put_line('Error!');
END;
/
```

PACKAGES

- A Package is a container that may have many functions and procedures within it.
- A **PACKAGE** is a group of programmatic constructs combined.
- A package is a schema object that groups logically related PL/SQL types, items, and subprograms

A PACKAGE EXISTS IN TWO PARTS:

- ***Package Specification*:-** The **specification** is the interface to your applications it declares the types, variables, constants, exceptions, cursors, and subprograms available for use. ***Package Body:-***This contains the definition of the constructs prototyped in the spec. It may also contain the private or locally defined program units, which can be used within the scope of the package body only..

OOP'S

- PACKAGES DEMONSTRATE ENCAPSULATION, DATA HIDING, SUBPROGRAM OVERLOADING AND MODULARIZATION IN PL/SQL

SIMPLE EXAMPLE ON PACKAGE

- STEP NO 1:- Package specification created first means the definition of constructs in pacakage

```
CREATE OR REPLACE PACKAGE PKG_NAME IS
PROCEDURE P_ENAME(P_VAR VARCHAR2);
END;
/
```

STEP NO 2) Creating package body

- CREATE OR REPLACE PACKAGE BODY PKGNAME IS
  PROCEDURE P_ENAME(P_VAR VARCHAR2) IS
  BEGIN
  DBMS_OUTPUT.PUT_LINE(P_VAR);
  END;
  END PKGNAME;→PACKAGE END
  /

CALLING PACAKGE

- EXECUTE IS USED TO CALL A PACAKAGE
- EXEC PKG_UTIL.P_ENAME(‘MIRZA');
- OUTPUT:-MIRZA