PL/SQL

Procedure Language extension to SQL

Guest Lecture by
Vijaykumar Karur

Session Plan

Need for PL/SQL

- SQL – Language designed to examine and manipulate relational data, but lacks programming capabilities.
- PL/SQL paves way for programming with SQL.
- PL/SQL engine.
- Advantages –
  - Block Structure
  - Procedural Language Capability
  - Better Performance
  - Error Handling
  - Transaction Implementation

PL/SQL Block

- Declaration Section
- Execution Section
- Exception Section

```
[DECLARE]
-- declaration of variables
BEGIN
/* SQL and
PL/SQL statements */
[EXCEPTION]

END;
```

Did you observe the single and multi line commenting?.

Anonymous PL/SQL Blocks

- Do not have any name.
- Not permanently stored in the database.
- Cannot call other anonymous PL/SQL blocks as well as itself.
- What is the minimum block of PL/SQL that can be executed without any error?.

```
BEGIN
NULL;
END;
```

PL/SQL Block Execution

```
SQL> edit E:\Vijay\Seminars\HelloWorld.sql
BEGIN
      DBMS_OUTPUT.PUT_LINE('Hello World!');
END;
/
SQL> get E:\Vijay\Seminars\HelloWorld.sql
1   BEGIN
2  DBMS_OUTPUT.PUT_LINE('Hello World!');
3  * END;
SQL> SET SERVEROUTPUT ON;
SQL> @ E:\Vijay\Seminars\HelloWorld.sql
Hello World!
PL/SQL procedure successfully completed.

SQL> BEGIN
  2      DBMS_OUTPUT.PUT_LINE('Hello World!');
  3   END;
  4   .
SQL> /
Hello World!
PL/SQL procedure successfully completed.
```

Variables and Data types

- Syntax

```
variable_name [CONSTANT] datatype [NOT NULL] [:=value]
```

- Variables defined as CONSTANT and NOT NULL must be initialized.
- := or DEFAULT can be used to initialize the variables.
- Data Types

CHAR(n)	CHAR(n CHAR)
VARCHAR2(n)	VARCHAR2(n CHAR)
PLS_INTEGER	NUMBER(p, s)
BOOLEAN	DATE
TIMESTAMP	

DBMS_OUTPUT.PUT_LINE

- DBMS_OUTPUT is package and PUT_LINE is a procedure within the package.
- SET SERVEROUTPUT ON should be used to enable the package.
- This is used to display messages to the screen from anonymous PL/SQL blocks.
- Debugging is the most popular use of this package.
- Procedures available –
  - DBMS_OUTPUT.PUT
  - DBMS_OUTPUT.PUT_LINE
  - DBMS_OUTPUT.NEW_LINE

```
BEGIN
        DBMS_OUTPUT.PUT('Hello'
        DBMS_OUTPUT.PUT('World');
END;
/
```

Nothing is displayed..Why?

```
BEGIN
        DBMS_OUTPUT.PUT(‘BVBCET');
        DBMS_OUTPUT.PUT_LINE(‘HUBLI');
END;
/
```

BVBCETHUBLI

Anchored Declarations

- Usage
  - Declare a variable that directly maps to a column definition in the database.

-Are NOT NULL and CHECK constraint associated with the table column applicable to the variable declare?
-Any change to the table column would also be reflected in the variable.

```
variable_name TABLENAME.COLUMNNAME%TYPE;
```

```
vNum1 NUMBER NOT NULL:= 50;
vNum2 vNum1%TYPE := 51;
```

-NOT NULL constraint is applied to the variable.
-The value is not copied.

Arrays

```
TYPE var_array IS VARRAY(n) of <element_type>

NOTE: Arrays index starts from 1
```

- The starting index for varrays is always 1.
- You can initialize the varray elements using the constructor method of the varray type, which has the same name as the varray.
- Varrays are one-dimensional arrays.
- A varray is automatically NULL when it is declared and must be initialized before its elements can be referenced.

```
DECLARE
  TYPE ArrayNumbers IS VARRAY(5) OF NUMBER;
  vSearchNum NUMBER;
  vArray ArrayNumbers;
  vCnt NUMBER;
  vBool BOOLEAN := FALSE;
BEGIN
  vArray := ArrayNumbers(10, 20, 30, 40, 50);
  vCnt := vArray.COUNT;
  vSearchNum := '&SearchForNum';
  FOR vLoop IN 1..vCnt
  LOOP
    IF vSearchNum = vArray(vLoop) THEN
      DBMS_OUTPUT.PUT_LINE('Found');
      vBool := TRUE;
      EXIT;
   END IF;
  END LOOP;
  IF NOT vBool THEN
    DBMS_OUTPUT.PUT_LINE('Not Found');
  END IF;
END;
```

Accepting input in PL/SQL

- How to accept input from the user?.
- SET VERIFY ON/OFF
- PL/SQ

```
Enter value for CollageName:BVBCET
BVBCET
PL/SQL procedure successfully completed.
```

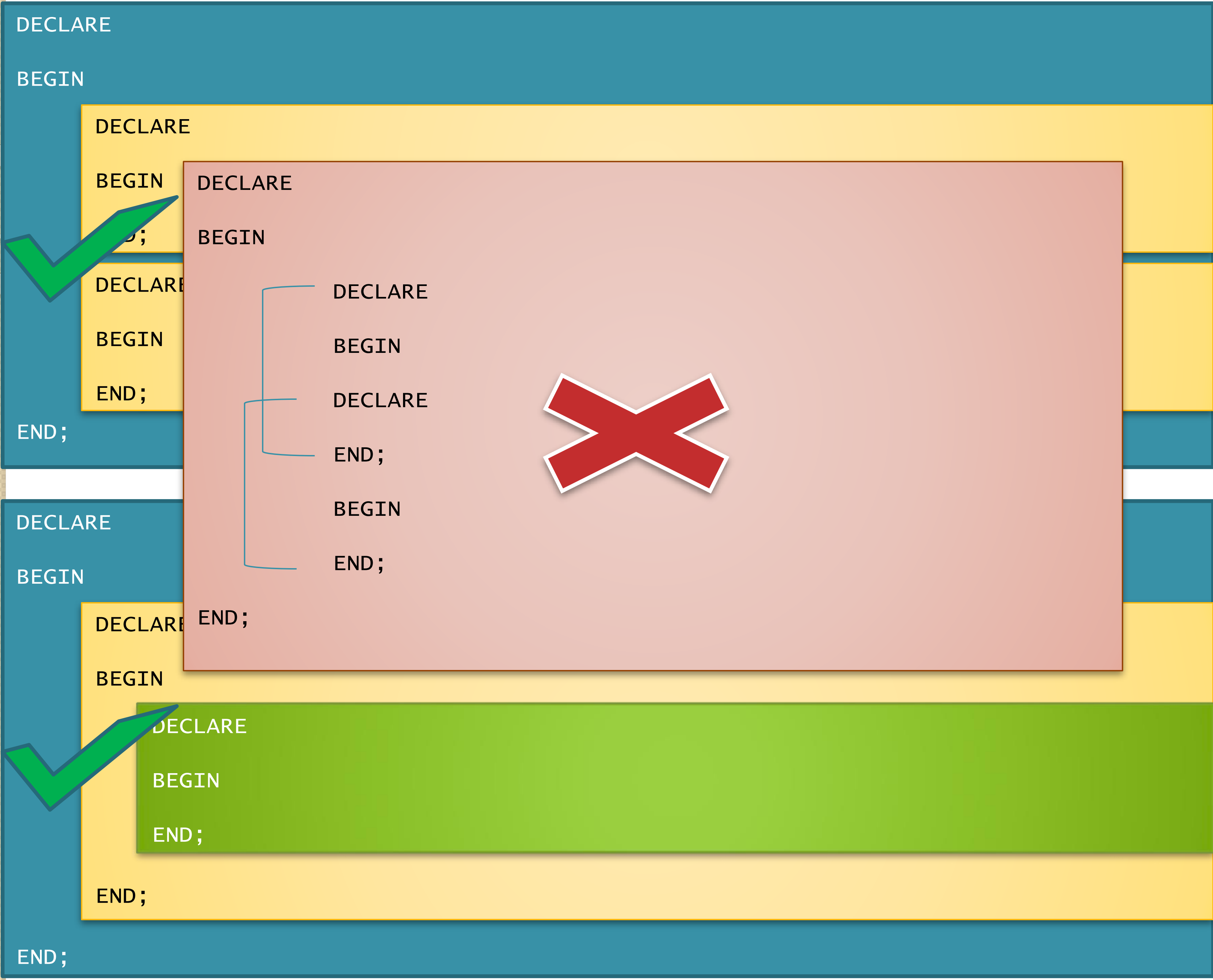
```
DECLARE
        vDeptName VARCHAR2(100);
BEGIN
        vDeptName := '&DepartmentName';
        DBMS_OUTPUT.PUT_LINE(vDeptName);
END;
/
```

?

```
DECLARE
        vCollName VARCHAR2(20);
        vCity VARCHAR2(20);
BEGIN
        vCollName := '&CollageName';
        DBMS_OUTPUT.PUT_LINE('CollegeName :'||vCollName);
        vCity := '&City';
        DBMS_OUTPUT.PUT_LINE('City :'||vCity);
END;
/
```

Nested PL/SQL Blocks

- A PL/SQL block defined within another PL/SQL block is called nested PL/SQL block.
- Can be nested in the executable section or in exception handling section.
- Overlapping of nested blocks in not allowed.



```
DECLARE
BEGIN
    DECLARE
    BEGIN
    END;
    DECLARE
    BEGIN
    END;
END;
```

```
DECLARE
BEGIN
    DECLARE
    BEGIN
        DECLARE
        BEGIN
        END;
    END;
END;
```

```
DECLARE
BEGIN
        DECLARE
        BEGIN
        DECLARE
        END;
        BEGIN
        END;
END;
```

Scope of variables

```
DECLARE
vNum NUMBER := 10;
BEGIN
        DECLARE
        vNum NUMBER := 20;
        BEGIN
        DBMS_OUTPUT.PUT_LINE('Number '||vNum);
        END;
        DECLARE
        vNum2 NUMBER := 30;
        BEGIN
        vNum := 40;
        DBMS_OUTPUT.PUT_LINE('Numb
        END;
DBMS_OUTPUT.PUT_LINE('Number '||vNum);
DBMS_OUTPUT.PUT_LINE('Number '||vNum2);
END;
```

Number 20

Number 40

Number 40

?

Qualifying Identifiers

```
<<outer>>
DECLARE
vNum NUMBER := 10;
BEGIN
      <<inner1>>
      DECLARE
      vNum NUMBER := 20;
      BEGIN
      DBMS_OUTPUT.PUT_LINE('Number '||outer.vNum);
      END;
      <<inner2>>
      DECLARE
      vNum2 NUMBER := 30;
      BEGIN
      vNum := 40;
      DBMS_OUTPUT.PUT_LINE('Numb
      END;
DBMS_OUTPUT.PUT_LINE('Number '||outer.vNum);
DBMS_OUTPUT.PUT_LINE('Number '||inner2.vNum2);
END;
```

Number 10

Number 40

Number 40

?

Conditional Statements

```
IF condition
THEN
  action;
END IF;
```

```
IF condition
THEN
  action;
ELSE
  action;
END IF;
```

```
IF condition
THEN
  action;
ELSIF condition
THEN
  action;
[ELSE
  action;]
END IF;
```

The condition must evaluate to TRUE, FALSE or NULL

Example

```
DECLARE
  vMarks NUMBER := '&Marks';
  vGrade VARCHAR2(2);
BEGIN
  IF vMarks >= 90 THEN
    vGrade := 'A+';
  ELSIF vMarks >= 80 AND vMarks < 90 THEN
    vGrade := 'A';
  ELSIF vMarks >= 70 AND vMarks < 80 THEN
    vGrade := 'B';
  ELSE
    vGrade := 'C';
  END IF;
  DBMS_OUTPUT.PUT_LINE('Grade : '||vGrade);
END;
/
```

Iterative Statements

- 1. Loop

```
DECLARE
  vNum NUMBER := 1;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE('Loop Count : '||vNum);
    vNum := vNum + 1;
    EXIT WHEN vNum > 5;
  END LOOP;
END;
/
```

- 2. Numeric FOR loop

```
BEGIN
   FOR vNum IN 1..5
   LOOP
      DBMS_OUTPUT.PUT_LINE('Number : '||vNum);
   END LOOP;
END;
```

```
Number : 1
Number : 2
Number : 3
Number : 4
Number : 5
```

- 3. Numeric FOR loop REVERSE

```
BEGIN
   FOR vNum IN REVERSE 1..5
   LOOP
      DBMS_OUTPUT.PUT_LINE('Number : '||vNum);
   END LOOP;
END;
```

```
Number : 5
Number : 4
Number : 3
Number : 2
Number : 1
```

Why are we not declaring and initializing the variable?. What if we do?

- 4. WHILE loop

```
DECLARE
  vNum NUMBER := 1;
BEGIN
  WHILE vNum <= 5
  LOOP
    DBMS_OUTPUT.PUT_LINE('Number : '||vNum);
    vNum := vNum + 1;
  END LOOP;
END;
/
```

SQL SELECT in PL/SQL

```
SELECT selct_list INTO variable_list
FROM table_list
[WHERE condition];
```

```
DECLARE
      vDeptName DEPARTMENT.DEPARTME
      vDeptId DEPARTMENT.DEPAR MENT
BEGIN
      vDeptId := '&DepartmentId';
      SELECT DEPARTMENT_NAME INTO vDeptName FROM
DEPARTMENT WHERE DEPARTMENT_ID = DeptId
      DBMS_OUTPUT.PUT_LINE('Department
vDeptName);
END;
```

What happens if no rows are returned?

What happens if more than one record is returned?

What if you have to fetch all the columns in a table and the table has a huge number of columns?

Composite Data Type

```
recordvariablename tablename%ROWTYPE;
```

```
DECLARE
         vStudent STUDENT%ROWTYPE;
         vStudentId STUDENT.STUDENT_ID%TYPE;
BEGIN
         vStudentId := '&StudentId';
         SELECT * INTO vStudent FROM STUDENT WHERE STUDENT_ID = vS
         DBMS_OUTPUT.PUT_LINE('Student Id : ' || vStudent.STUDENT_
         DBMS_OUTPUT.PUT_LINE('Student Name : ' || vStudent.STUDE
         DBMS_OUTPUT.PUT_LINE('Student Address : ' || vStudent.   UDENT_ADDRE  );
         DBMS_OUTPUT.PUT_LINE('Student Dept Id : ' || vStudent.STUDENT_DEPARTMENT_ID);
         DBMS_OUTPUT.PUT_LINE('Student Sem : ' || vStudent.STUDENT_SEM);
END;
```

What if try to print record variable?

SQL INSERT in PL/SQL

- How to insert records into a table?

```
DECLARE
       vDeptId DEPARTMENT.DEPARTMENT_ID%TYPE;
       vDeptName DEPARTMENT.DEPARTMENT_NAME%TYPE;
BEGIN
       vDeptId := '&DepartmentId';
       vDeptName := '&DepartmentName';
       INSERT INTO DEPARTMENT VALUES(vDeptId            );
END;
```

Will this work?

IINSERT INTO DEPARTMENT VALUES('&DepartmentId', '&Depart entName');

SQL UPDATE in PL/SQL

- How to update the records in a table?

```
DECLARE
	vDeptId DEPARTMENT.DEPARTMENT_ID%TYPE;
	vDeptName DEPARTMENT.DEPARTMENT_NAME%TYPE;
BEGIN
	vDeptId := '&DepartmentId';
	vDeptName := '&DepartmentNewName';
	UPDATE DEPARTMENT SET DEPARTMENT_NAM
vDeptName WHERE DEPARTMENT_ID = vDeptId;
END;
```

Will this work?

UPDATE DEPARTMENT SET DEPARTMENT_NAME = '&DepartmentNewName' WHERE DEPARTMENT_ID = '&DepartmentId';

SQL DELETE in PL/SQL

- How to delete the records in a table?

```
DECLARE
      vDeptId DEPARTMENT.DEPARTMENT_ID%TYPE;
BEGIN
      vDeptId := '&DepartmentId';
      DELETE FROM DEPARTMENT WHERE DEPARTMENT_ID =
vDeptId;
END;
```

Will this work?

DELETE FROM DEPARTMENT WHERE DEPARTMENT_ID = '&DepartmentId';

SQL%Attributes

- How to test the outcome of the SQL statements executed?
- Can be used in both execution and exception sections.

SQL%Attribute	Meaning
SQL%ROWCOUNT	No. of records affected by the most recent SQL statement.
SQL%FOUND	TRUE if the most recent SQL statement affects one or more rows.
SQL%NOTFOUND	TRUE if the most recent SQL statement does not affect any rows.
SQL%ISOPEN	Always evaluates to FALSE because PL/SQL closes implicit cursors immediately after they are executed.

DML statement status	SQL%ROWCOUNT	SQL%FOUND	SQL%NOTFOUND	SQL%ISOPEN
INSERT success	1	TRUE	FALSE	FALSE
INSERT Fail	0	FALSE	TRUE	FALSE
UPDATE/DELETE success	N	TRUE	FALSE	FALSE
UPDATE/DELETE Fail	0	FALSE	TRUE	FALSE

```
BEGIN
        UPDATE DEPARTMENT
        SET DEPARTMENT_NAME = 'Information Science'
        WHERE DEPARTMENT_ID = 'ISE';
        IF SQL%FOUND
        THEN
                DBMS_OUTPUT.PUT_LINE('Number of Rows Updated :
' || SQL%ROWCOUNT);
        ELSE
                DBMS_OUTPUT.PUT_LINE('Record not found');
        END IF;
END;
```

Operators and Functions

- Operators
  - Concatenation Operator (||)
  - Arithmetic Operators (+, -, *, /, **)
  - Relational Operators (=, !=, <, >, <=, >=)
  - Logical Operators (AND, OR, NOT)
- Functions
  - SELECT SYSDATE FROM DUAL;
  - TRIM {Leading/Trailing/Both} trim_char FROM trim_source
  - SUBSTR(string, position, substring_length)
  - TRANSLATE('char', 'from_string', 'to_string')
  - NVL(expr1, expr2)
  - INSTR(string, find_string [, start] [, occurrence])
  - LENGTH(string)
  - LPAD(string, no_of_char_reserved, padding_char)
  - RPAD(string, no_of_char_reserved, padding_char)
  - DECODE(expression, search_condn, result [,search_condn, result]…[,default])

Exception

- Exception is an identifier in PL/SQL that is raised during the execution.
- It terminates the main body and transfers the control to the exception section.
- Program execution would continue in the exception handler and then to any outer block, if it is nested.
- If Exception is not handled, the exception is propagated to the calling environment.
- Exception Types –
  - Predefined Oracle Server Exceptions
  - Non-Predefined Oracle Server Exceptions
  - User-Defined Exceptions

```
DECLARE

BEGIN

EXCEPTION
  WHEN exception1 [OR exception2 . . .] THEN
    statement1;
    statement2;
    . . .
  [WHEN exception3 [OR exception4 . . .] THE
    statement1;
    statement2;
    . . .]
  [WHEN OTHERS THEN
    statement1;
    statement2;
    . . .]
END;
```

1. Pre-Defined Oracle Server Exceptions

Oracle Error	Pre-Defined Exception	Description
ORA-1403	NO_DATA_FOUND	SELECT statement matches no rows
ORA-1422	TOO_MANY_ROWS	SELECT statement matches more than one row
ORA-0001	DUP_VAL_ON_INDEX	Unique constraint violated
ORA-1476	ZERO_DIVIDE	Division by zero
ORA-6502	VALUE_ERROR	Truncation, Arithmetic Error
ORA-1722	INVALID_NUMBER	Conversion to number failed

```
DECLARE
  vName STUDENT.STUDENT_NAME%TYPE;
BEGIN
  SELECT STUDENT_NAME INTO vName FROM STUDENT WHERE
STUDENT_ID = '2BV15IS001';
  /*SELECT STUDENT_NAME INTO vName FROM STUDENT WHERE
STUDENT_ID = '2BV15IS100';*/
  /*SELECT STUDENT_NAME INTO vName FROM STUDENT WHERE
STUDENT_SEM = 5;*/
  DBMS_OUTPUT.PUT_LINE('Student Name : '||vName);
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('No record found');
  WHEN TOO_MANY_ROWS THEN
    DBMS_OUTPUT.PUT_LINE('Too many records found');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Exception Occurred');
END;
/
```

2. Non-Predefined Oracle Server Exceptions

- Declare an exception identifier in the declaration section.
  - eNotNullExc EXCEPTION;
- Associate the exception declared with an oracle error number using PRAGMA EXCEPTION_INIT compiler directive.
  - PRAGMA EXCEPTION_INIT(eNotNullExc, -1400);
- Compiler associates an exception name to the oracle error number.
- No need to raise the exception explicitly.
- The exception can now be trapped using the name in the exception section.

  -
    ```
    EXCEPTION
        WHEN eNotNullExc THEN
                    DBMS_OUTPUT.PUT_LINE(' … ');
     END;
    ```

```
BEGIN
	INSERT INTO DEPARTMENT VALUES('IPE',NULL);
END;
```

```
DECLARE
	eNotNullException EXCEPTION;
	PRAGMA EXCEPTION_INIT(eNotNullException, -1400);
BEGIN
	INSERT INTO DEPARTMENT VALUES('IPE', NULL);
EXCEPTION
	WHEN eNotNullException THEN
		DBMS_OUTPUT.PUT_LINE('Not Null Exception');
	WHEN OTHERS THEN
		DBMS_OUTPUT.PUT_LINE('Exception');
END;
```

3. User-Defined Exceptions

- Declare an exception identifier in the declaration section.
  - eInvalidMarks EXCEPTION;
- Raise the exception explicitly in the executable section using the RAISE statement.
  - RAISE eInvalidMarks;
- Handle the exception in the exception handling section using the identifier.

  -
    ```
    EXCEPTION
    WHEN eInvalidMarks THEN
       DBMS_OUTPUT.PUT_LINE( ' … ' );
    END;
    ```

```
DECLARE
  vMarks NUMBER := '&Marks';
  vGrade VARCHAR2(2);
  eInvalidMarksExc EXCEPTION;
BEGIN
  IF vMarks < 0 OR vMarks > 100 THEN
    RAISE eInvalidMarksExc;
  END IF;
  IF vMarks >= 90 THEN
    vGrade := 'A+';
  ELSIF vMarks >= 80 AND vMarks < 90 THEN
    vGrade := 'A';
  ELSIF vMarks >= 70 AND vMarks < 80 THEN
    vGrade := 'B';
  ELSE
    vGrade := 'C';
  END IF;
  DBMS_OUTPUT.PUT_LINE('Grade : '||vGrade);
EXCEPTION
  WHEN eInvalidMarksExc THEN
    DBMS_OUTPUT.PUT_LINE('Invalid Marks');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Exception Occured');
END;
```

SQLCODE and SQLERRM

```
DECLARE
  vNum NUMBER := 10;
BEGIN
  vNum := vNum / (10 - 10); -- Divide By Zero
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Exception Occured');
    DBMS_OUTPUT.PUT_LINE('Error Code : '||SQLCODE);
    DBMS_OUTPUT.PUT_LINE('Error Msg : '||SQLERRM);
END;
/
```

Using RAISE_APPLICATION_ERROR

- RAISE_APPLICATION_ERROR(error_number, error_message);
- Report errors to your application and avoid returning unhandled exceptions.
- error_number is a value between -20000 to -20999.
- error_message is a text associated with this error, should be less than 512 characters.
- Can be handled by WHEN OTHERS exception clause.
- Can be used in both execution and exception block.

```
DECLARE
  vMarks NUMBER := '&Marks';
  vGrade VARCHAR2(2);
  eInvalidMarksExc EXCEPTION;
BEGIN
  IF vMarks < 0 OR vMarks > 100 THEN
    RAISE_APPLICATION_ERROR(-20000, 'Invalid Marks');
  END IF;
  IF vMarks >= 90 THEN
    vGrade := 'A+';
  ELSIF vMarks >= 80 AND vMarks < 90 THEN
    vGrade := 'A';
  ELSIF vMarks >= 70 AND vMarks < 80 THEN
    vGrade := 'B';
  ELSE
    vGrade := 'C';
  END IF;
  DBMS_OUTPUT.PUT_LINE('Grade : '||vGrade);
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Exception Occured');
    DBMS_OUTPUT.PUT_LINE('Error Code : '||SQLCODE);
    DBMS_OUTPUT.PUT_LINE('Error Msg : '||SQLERRM);
END;
```

Exception Propagation

```
DECLARE
  vNum NUMBER := 10;
BEGIN
  DECLARE
    vNum2 NUMBER := 20;
  BEGIN
    vNum := vNum / (vNum2 - 20); -- Divide By Zero
  END;
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Exception Occured');
    DBMS_OUTPUT.PUT_LINE('Error Code : '||SQLCODE);
    DBMS_OUTPUT.PUT_LINE('Error Msg : '||SQLERRM);
END;
```

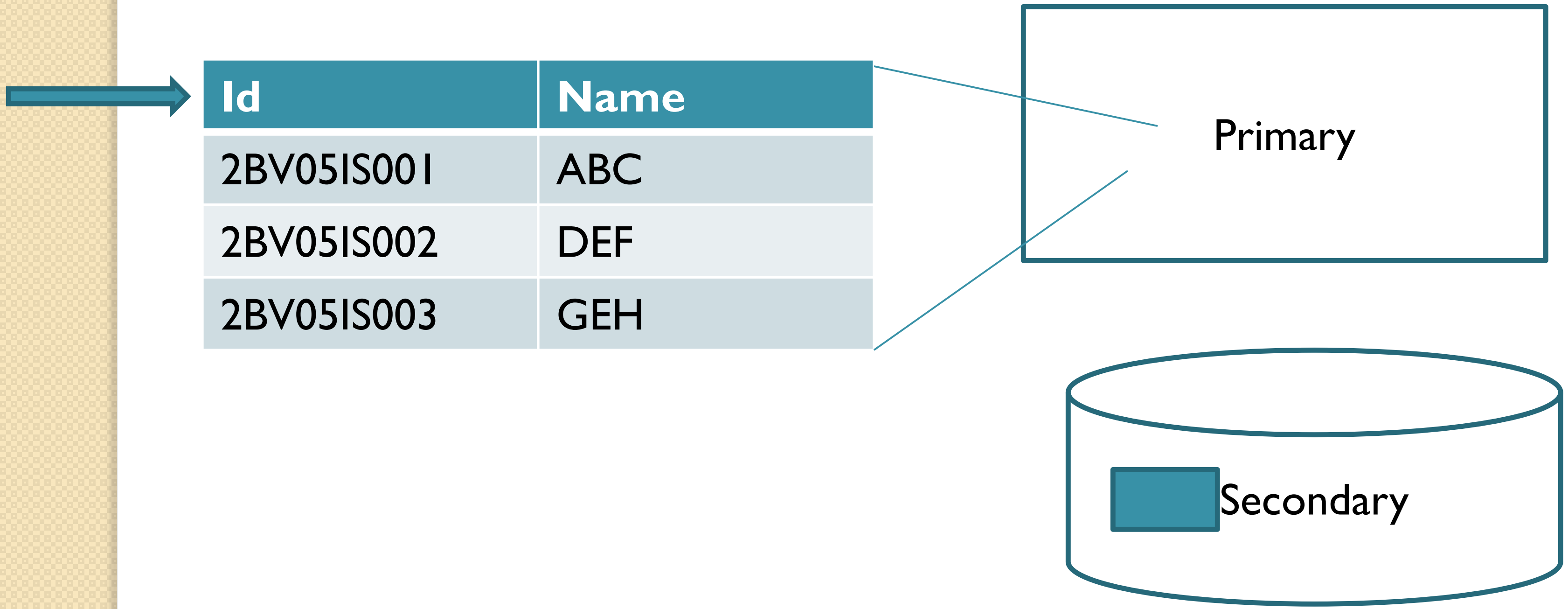
CURSOR

- A cursor is a private SQL work area.
- Every SQL statement executed by the oracle server has an individual cursor associated with it.
- There are two types of cursors –
    - Implicit Cursors
        - Declared for all DML and PL/SQL statements.
    - Explicit Cursors
        - Declared and named by the programmer.
- The cursor point to a memory region called the context area that holds the following :
    - Rows returned by the query.
    - Number of rows processed by the query.
    - A pointer to the parsed query.

Explicit Cursors

- Meant to work with the SELECT statements that return more than one record.
- Operations performed on explicit cursors are
  - Declaring the cursor.
  - Opening the cursor.
  - Fetching the records from the cursor.
  - Close the cursor.

Id	Name
2BV05IS001	ABC
2BV05IS002	DEF
2BV05IS003	GEH



1. Declaration of the cursor

- CURSOR cur_name IS query;
- cur_name is the cursor name and it can be any valid identifier.
- query can be any select statement. The select statement need not have INTO clause.

```
CURSOR curStudent1 IS SELECT * FROM STUDENT;

CURSOR curStudent2 IS
       SELECT STUDENT_DEPARTMENT_ID, COUNT(*)
       FROM STUDENT
       GROUP BY STUDENT_DEPARTMENT_ID;
```

2. Opening the cursor

- OPEN cur_name;
- Cursors can be opened in execution or exception blocks.
- If cursor is already opened it would throw CURSOR_ALREADY_OPEN runtime exception.
- Select query associated with the cursor declaration is executed only when you open the cursor.
- The OPEN command prepares the resultset and positions the cursor before the 1st row.

```
OPEN curStudent1;

IF NOT curStudent2%ISOPEN THEN
  OPEN curStudent2;
END IF;
```

3. Fetching records from the cursor

- FETCH cur_name INTO variable(s) | PL/SQL record.
- cur_name is the name of the cursor that is already opened. Fetching the cursor that is not open will result into runtime exception.
- The order and data type of the variables mentioned in the FETCH has to exactly match the list of columns in the SELECT statement.

```
FETCH curStudent1 INTO vStudRec;

FETCH curStudent2 INTO vDeptNo, vCount;
```

4. Closing the cursor

- CLOSE cur_name;
- cur_name is the opened cursor. Closing an unopened cursor would result into a runtime exception.
- Memory allocated to the cursor is released.

```
CLOSE curStudent1;

IF curStudent2%ISOPEN THEN
  CLOSE curStudent2;
END IF;
```

Explicit Cursor Attributes

- cur_name%ISOPEN – Is the cursor open?
- cur_name%ROWCOUNT – How many rows fetced so far?
- cur_name%FOUND – Has a row been fetched?
- cur_name%NOTFOUND – Has a fetch failed?

		%FOUND	%ISOPEN	%NOTFOUND	%ROWCOUNT
After	OPEN	NULL	TRUE	NULL	0
After	1st Fetch	TRUE	TRUE	FALSE	1
After	2nd Fetch	TRUE	TRUE	FALSE	2
After	n+1 Fetch	FALSE	TRUE	TRUE	n
After	Close	Exception	FALSE	Exception	Exception

```
DECLARE
  CURSOR curStudent IS
    SELECT STUDENT_DEPARTMENT_ID, COUNT(*)
    FROM STUDENT
    GROUP BY STUDENT_DEPARTMENT_ID;
  vDeptId STUDENT.STUDENT_DEPARTMENT_ID%TYPE;
  vCount NUMBER;
BEGIN
  IF NOT curStudent%ISOPEN THEN
    OPEN curStudent;
  END IF;
  LOOP
    FETCH curStudent INTO vDeptId, vCount;
    EXIT WHEN curStudent%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Deparment Id : '||vDeptId||'
No. of Students : '||vCount);
  END LOOP;
  IF curStudent%ISOPEN THEN
    CLOSE curStudent;
  END IF;
END;
```

```
DECLARE
  CURSOR curStudent IS
    SELECT *
    FROM STUDENT
    WHERE STUDENT_DEPARTMENT_ID = 'ISE';
  recStudent STUDENT%ROWTYPE;
BEGIN
  IF NOT curStudent%ISOPEN THEN
    OPEN curStudent;
  END IF;
  LOOP
    FETCH curStudent INTO recStudent;
    EXIT WHEN curStudent%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(recStudent.STUDENT_ID||'
'||recStudent.STUDENT_NAME);
  END LOOP;
  IF curStudent%ISOPEN THEN
    CLOSE curStudent;
  END IF;
END;
```

Explicit Cursors – WHILE loop

```
DECLARE
  CURSOR curStudent IS
    SELECT STUDENT_DEPARTMENT_ID, COUNT(*) FROM STUDENT
    GROUP BY STUDENT_DEPARTMENT_ID;
  vDeptId STUDENT.STUDENT_DEPARTMENT_ID%TYPE;
  vCount NUMBER;
BEGIN
  IF NOT curStudent%ISOPEN THEN
    OPEN curStudent;
  END IF;
  FETCH curStudent INTO vDeptId, vCount;
  WHILE curStudent%FOUND
  LOOP
    DBMS_OUTPUT.PUT_LINE('Deparment Id : '||vDeptId||'
No. of Students : '||vCount);
    FETCH curStudent INTO vDeptId, vCount;
  END LOOP;
  IF curStudent%ISOPEN THEN
    CLOSE curStudent;
  END IF;
END;
```

Explicit Cursor – FOR loop

```
DECLARE
   CURSOR curStudent IS
      SELECT STUDENT_DEPARTMENT_ID AS DEPTID, COUNT(*) AS COUNT
      FROM STUDENT
      GROUP BY STUDENT_DEPARTMENT_ID;
BEGIN
   FOR recStudent IN curSTUDENT
   LOOP
      DBMS_OUTPUT.PUT_LINE('Deparment Id : '||recStudent.DEPTID||'
No. of Students : '||recStudent.COUNT);
   END LOOP;
END;
```

1. The variable/record variable is declared implicitly.
2. The cursor is opened implicitly.
3. The cursor fetch happens implicitly.
4. The cursor is closed implicitly.

Implicit FOR loops

```
BEGIN
   FOR recStudent IN (SELECT * FROM STUDENT WHERE
STUDENT_DEPARTMENT_ID = 'ISE')
   LOOP
      DBMS_OUTPUT.PUT_LINE(recStudent.STUDENT_ID||'
'||recStudent.STUDENT_NAME);
   END LOOP;
END;
```

Is the above mentioned approach better than the others?.

Cursor Exceptions

```
DECLARE
 CURSOR curStudent IS SELECT * FROM
  vStudentRec STUDENT%ROWTYPE;
BEGIN
   --OPEN curStudent;

  FETCH curStudent INTO vS
  WHILE curStudent%FOUND
  LOOP
     OPEN curStudent;
    DBMS_OUTPUT.PUT_LINE(vStudentRec.STUDENT_NAME);
    FETCH curStudent INTO vStudentRec;
  END LOOP;
  CLOSE curStudent;
END;
```

INVALID_CURSOR exception is thrown

CURSOR_ALREADY_OPEN exception is thrown

Subprograms

- A subprogram is named PL/SQL block that can accept parameters and be invoked from a calling environment.
- It provides modularity, extensibility, reusability, and maintainability.

Local Procedure

Do not specify the size for formal parameters.

```
DECLARE
  vStudentId STUDENT.STUDENT_ID%TYPE := '&St dentId';
  PROCEDURE DisplayDepartmentName(pStudentId VARCHAR2)
    IS
      vDeptName DEPARTMENT.DEPARTMENT_NAME%TYPE;
    BEGIN
      SELECT DEPARTMENT_NAME INTO vDeptName
      FROM STUDENT JOIN DEPARTMENT ON
  STUDENT_DEPARTMENT_ID = DEPARTMENT_ID
      WHERE STUDENT_ID = pStudentId

      DBMS_OUTPUT.PUT_LINE('Depar
  '||vDeptName);
    END Displ yDepartmentName;

BEGIN
  DisplayDepartmentName(vStudentId);
END;
```

No declaration of the variables after the procedure implementation

Local Function

```
DECLARE
  vStudentId STUDENT.STUDENT_ID%TYPE := '&StudentId';
  vDeptName DEPARTMENT.DEPARTMENT_NAME%TYPE;
  FUNCTION GetDepartmentName(pStudentId VARCHAR2)
    RETURN VARCHAR2
    IS
      vDeptName DEPARTMENT.DEPARTMENT_NAME%TYPE;
    BEGIN
      SELECT DEPARTMENT_NAME INTO vDeptName
      FROM STUDENT JOIN DEPARTMENT ON
  STUDENT_DEPARTMENT_ID = DEPARTMENT_ID
      WHERE STUDENT_ID = pStudentId;
      RETURN vDeptName;
    END GetDepartmentName;
BEGIN
  vDeptName := GetDepartmentName(vStudentId);
  DBMS_OUTPUT.PUT_LINE('Department Name :
'||vDeptName);
END;
```

Limitations of Local procedure/function

- Are not permanently stored in the database.
- Cannot be invoked from another PL/SQL block.
- Cannot declare variables after local procedure/function implementation in the declaration.
- The degree of reusability is reduced.

Solution : Stored Procedures and Stored Functions

Stored Procedure

```
CREATE [OR REPLACE] PROCEDURE proc_name
(parameter1 [Mode] datatype1,
 parameter2 [Mode] datatype2,
 . . .)
IS|AS
 -- local variable declaration
BEGIN
 -- Execution Section
EXCEPTION
 --Exception Section
END [proc_name];
```

```
CREATE OR REPLACE PROCEDURE
DisplayDepartmentName(pStudentId VARCHAR2)
IS
    vDeptName DEPARTMENT.DEPARTMENT_NAME%TYPE;
BEGIN
    SELECT DEPARTMENT_NAME INTO vDeptName
    FROM STUDENT JOIN DEPARTMENT ON
STUDENT_DEPARTMENT_ID = DEPARTMENT_ID
    WHERE STUDENT_ID = pStudentId;

    DBMS_OUTPUT.PUT_LINE('Department Name :
'||vDeptName);
END DisplayDepartmentName;
```

```
EXEC DisplayDepartmentName('2BV15CS009');
```

```
DECLARE
    vStudentId STUDENT.STUDENT_ID%TYPE :=
'&StudentId';
BEGIN
    DisplayDepartmentName(vStudentId);
END;
```

Parameter Modes

IN	OUT	IN OUT
Default Mode	Must be specified	Must be specified
Value is passed into subprogram	Returned to the calling environment	Passed into subprogram; Returned to the calling environment
Formal parameter acts as constant	Uninitialized variable	Initialized variable
Actual parameter can be a literal, constant, expression, or initialized variable	Must be a variable	Must be variable
Can be assigned a default value	Cannot be assigned a default value	Cannot be assigned a default value

```
CREATE OR REPLACE PROCEDURE ComputeSum(pNum1 IN NUMBER,
pNum2 IN NUMBER, pNum3 OUT NUMBER)
IS
BEGIN
  pNum3 := pNum1 + pNum2;
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Exception');
END ComputeSum;
```

```
DECLARE
    vNum1 NUMBER := 10;
    vNum2 NUMBER := 20;
    vSum1 NUMBER;
    vSum2 NUMBER;
BEGIN
    ComputeSum(1, 2, vSum1);
    DBMS_OUTPUT.PUT_LINE('ComputeSum(1, 2, vSum1) :
'||vSum1);
    ComputeSum(vNum1, vNum2, vSum2);
    DBMS_OUTPUT.PUT_LINE('ComputeSum(vNum1, vNum2,
vSum2) : '||vSum2);
END;
```

Stored Function

```
CREATE [OR REPLACE] FUNCTION fun_name
(parameter1 [Mode] datatype1,
 parameter2 [Mode] datatype2,
 . . .)
RETURN DATATYPE
IS|AS
 -- local variable declaration
BEGIN
 -- Execution Section
EXCEPTION
 --Exception Section
END [fun_name];
```

```
CREATE OR REPLACE FUNCTION GetDiscountAmt(pBillAmt
NUMBER)
RETURN NUMBER
IS
BEGIN
  IF pBillAmt >= 2000 THEN
    RETURN 100;
  ELSIF pBillAmt >= 1000 AND pBillAmt < 2000 THEN
    RETURN 50;
  ELSE
    RETURN 25;
  END IF;
END;
```

```
DECLARE
  vBillAmt NUMBER := '&BillAmt';
  vFinalPrice NUMBER;
BEGIN
  vFinalPrice := vBillAmt - GetDiscountAmt(vBillAmt);
  DBMS_OUTPUT.PUT_LINE('Final Price : ' ||
vFinalPrice);
END;
```

Locations to call Stored Function

- Select list of SELECT command.
- Condition of WHERE or HAVING clause.
- ORDER BY and GROUP BY clause.
- VALUE clauses of INSERT command.
- SET clause of an UPDATE command.
- Restrictions on calling function from SQL expressions
  - The function must be stored funtion.
  - Accept only IN parameters.
  - Accept only valid SQL types, not PL/SQL specific types (e.g BOOLEAN).
  - Return data type should also be a valid SQL type.
  - You must own or have execute permission on the function.

Procedures vs. Functions

Procedures	Functions
Can be executed as a PL/SQL statement.	Can be invoked as part of an expression.
Do not contain RETURN clause in the header.	Must contain a RETURN clause in the header.
Can return none, one or many values.	Must return a single value.
Can contain a RETURN statement. Ex: RETURN;	Must contain at least one RETURN statement. Ex: RETURN vRes;

DROP procedure/function

- DROP PROCEDURE proc_name;
- DROP FUNCTION fun_name;

```
SELECT object_name, object_type
FROM user_objects
WHERE object_type IN ('PROCEDURE', 'FUNCTION')
```

Triggers

- Triggers are stored programs that are automatically executed or fired when an event occurs.
- Benefits
  - Enforcing referential integrity
  - Event Log or Access Log
  - Gather statistics on table access
  - Auditing
  - Imposing security authorization
  - Preventing invalid transactions
  - Generating derived column values
  - Modify table data when DML statements are issued against views

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
DECLARE
   Declaration-statements
BEGIN
   Executable-statements
EXCEPTION
   Exception-handling-statements
END;
```

```
ALTER TRIGGER trigger_name { ENABLE | DISABLE };
```

Constraints..

- OLD and NEW references are not available for table level triggers, rather you can use them for record level triggers.
- The mutating-table restriction prevents the trigger from querying or modifying the table that the triggering statement is modifying.
- The trigger cannot change OLD field values.
- If the triggering statement is DELETE, then the trigger cannot change NEW field values.
- An AFTER trigger cannot change NEW field values.
- An INSTEAD OF trigger is always a row-level trigger.
  An INSTEAD OF trigger can read OLD and NEW values, but cannot change them.
- WHEN clause is valid only for row level triggers.

```
CREATE OR REPLACE TRIGGER trgDeptHistory
BEFORE
INSERT OR DELETE OR UPDATE
OF DEPARTMENT_NAME
ON DEPARTMENT
REFERENCING OLD AS O NEW AS N
FOR EACH ROW
BEGIN
  INSERT INTO DEPARTMENT_HISTORY
VALUES(:O.DEPARTMENT_ID, :O.DEPARTMENT_NAME,
:N.DEPARTMENT_NAME, USER, SYSDATE);
  DBMS_OUTPUT.PUT_LINE('Changes logged into
Department History');
END;
```

```
UPDATE DEPARTMENT SET DEPARTMENT_NAME = 'Information
Science' WHERE DEPARTMENT_ID = 'ISE';
```

```
BEGIN
  UPDATE DEPARTMENT SET DEPARTMENT_NAME =
'Information Science' WHERE DEPARTMENT_ID = 'ISE';
END;
```

Instead of Trigger

- An INSTEAD OF trigger is the only way to update a view that is not inherently updatable.
- An INSTEAD OF trigger is always a row-level trigger. An INSTEAD OF trigger can read OLD and NEW values, but cannot change them.

```
CREATE OR REPLACE VIEW order_info AS
SELECT c.customer_id, c.cust_last_name,
       c.cust_first_name, o.order_id,
       o.order_date, o.order_status
FROM customers c, orders o
WHERE c.customer_id = o.customer_id;
```

```
CREATE OR REPLACE TRIGGER order_info_insert INSTEAD OF
INSERT ON order_info
DECLARE
    duplicate_info EXCEPTION;
    PRAGMA EXCEPTION_INIT (duplicate_info, -00001);
BEGIN
    INSERT INTO customers (customer_id, cust_last_name, cust_first_name)
    VALUES(:new.customer_id, :new.cust_last_name, :new.cust_first_name);

    INSERT INTO orders (order_id, order_date, customer_id)
    VALUES ( :new.order_id, :new.order_date, :new.customer_id);
EXCEPTION
    WHEN duplicate_info THEN
        RAISE_APPLICATION_ERROR ( num=> -20107, msg=> 'Duplicate
customer or order ID');

END order_info_insert; 00
```

```
INSERT INTO VALUES order_info VALUES(999, 'Smith', 'John', 2500, '13-
MAR-2001', 0);
```

Thank You