PL/SQL Programming

- PL/SQL stands for Procedural Language/SQL with SQL features

- The basic unit in any PL/SQL program is a block

- All PL/SQL programs are composed of blocks

Basic Block Structure

```
DECLARE
        /*Declarative section is here */
BEGIN
        /* Executable section is here */
```

```
EXCEPTION
        /* Exception section is here */
END;
```

▪Only the executable section is required; the other two are optional

Declaration syntax

▪Variables, Exceptions and Cursors are declared in the declarative section of the block

▪The general syntax for declaring a variable is

```
    Variable_name   type[CONSTANT][NOT
                                                NULL][:=VALUE];
```

- Where variable_name is the name of the variable, type is the data type, and value is the initial value of the variable

- For example, the following are all legal variable declarations:

```
 Description      VARCHAR2 (50);
NumberSeats     NUMBER := 45;
Counter              NUMBER := 0;
```

- If CONSTANT is present in the variable declaration, the variable must be initialized, and its value cannot be changed from the initial value

```
StudentID  CONSTANT NUMBER(5) := 10000;
```

- There can be only one variable declaration per line in the declarative section

- The following section is illegal, since two variables are declared in the same line:

```
        FirstName, LastName  VARCHAR2 (20);
```

Using %Type

```
     sailor_id   sailors.sid%TYPE;
```

- By using %TYPE, sailor_id will have type whatever type the sid column of the sailors table has

Example Program1:

```
DECLARE

s_id number(2):=99;
s_name varchar2(20):='John';
s_rating number(2):=7;
s_age number(3,1):=34.6;

BEGIN

INSERT INTO Sailors
VALUES(s_id,s_name,s_rating,s_age);

END;
```

SELECT * FROM Sailors

SID	NAME	RATING	AGE
22	Dustin	7	45
29	Brutus	1	33
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35
64	Horatio	7	35
71	Zorba	10	16
74	Horatio	9	35
85	Art	3	25.5
95	Bob	3	63.5
99	John	7	34.6

Example Program2:

```
BEGIN

UPDATE Sailors SET sid=77 where sid=99;

END;
```

SELECT * FROM Sailors

SID	NAME	RATING	AGE
22	Dustin	7	45
29	Brutus	1	33
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35
64	Horatio	7	35
71	Zorba	10	16
74	Horatio	9	35
85	Art	3	25.5
95	Bob	3	63.5
77	John	7	34.6

Example Program3:

```
BEGIN

DELETE FROM Sailors WHERE sid=77;

END;
```

SELECT * FROM Sailors

SID	NAME	RATING	AGE
22	Dustin	7	45
29	Brutus	1	33
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35
64	Horatio	7	35
71	Zorba	10	16
74	Horatio	9	35
85	Art	3	25.5
95	Bob	3	63.5

Example Program4:

```
set serveroutput on;
DECLARE

s_id sailors.sid%TYPE;

BEGIN

SELECT sid into s_id FROM Sailors WHERE sid=22;
dbms_output.put_line('s_id value is '||s_id);

END;
```

Output:

```
s_id value is 22
```

PL/SQL Control Structures

IF-THEN-ELSE

- The syntax for an IF-THEN-ELSE statement is

```
 IF Boolean_expression1 THEN
       sequence_of_statements1;
[ELSIF Boolean_expression2 THEN
       sequence_of_statements2;]
 …
 [ELSE
       sequence_of_statements3;]
 END IF;
```

LOOPS

Simple Loops

- The most basic kind of loops, simple loops, have this syntax:

```
        LOOP
        Sequence_of_statements;
        END LOOP;
```

- Sequence_of_statements will be executed infinitely, since this loop has no stopping condition

- We can add one via the EXIT statement, which has this syntax:

```
        EXIT [WHEN condition];
```

WHILE Loops

- The syntax for a WHILE LOOP is

```
WHILE condition LOOP
        sequence_of_statements;
END LOOP;
```

- The condition is evaluated before each iteration of the loop. If it evaluates to TRUE, sequence_of_statements is executed. If condition evaluates to FALSE or NULL, the loop is finished and control resumes after the END LOOP statement

Numeric FOR Loops

- The number of iterations for simple loops and WHILE loops is not known in advance—it depends on the loop condition

- Numeric FOR loops, on the other hand, have a defined number of iterations

- The syntax is

```
  FOR loop_counter IN [REVERSE] low_bound..
                                              high_bound LOOP
             Sequence_of_statements;
 END LOOP;
```

- Where loop_counter is an variable, low_bound and high_bound specify the number of iterations, and sequence_of_statements is the contents of the loop

- The bounds of the loop are evaluated once. This determines the total number of iterations. Loop_counter will take on the values ranging from low_bound to high_bound, incrementing by 1 each time, until the loop is complete

Example1

```
BEGIN
for m1 in 1..50  loop
   dbms_output.put_line(m1);
end loop;
END;               Output: 1 2 3 4 … 50
```

Example2

```
BEGIN
for m1 in reverse 1..50  loop
   dbms_output.put_line(m1);
end loop;
END;
```

Output: 50 49 48 . . . 1

EXCEPTION HANDLING

Exception: An exception is a runtime error. Exceptions are declared in the declaration section of the block, raised in the executable section and handled in the exception section

▪There are two types of exceptions:

- Pre-defined
- User-defined

Predefined Exceptions

- Oracle has predefined several exceptions that correspond to the most common Oracle errors. Like the predefined data types (NUMBER, VARCHAR2, and so on), the identifiers for these exceptions are defined in package STANDARD

- Because of this, they are already available to the program—it is not necessary to declare them in the declarative section like a user-defined exception

- These predefined exceptions are described in Table 3.2

Oracle Error	Equivalent Exception	Description
ORA-0001	DUP_VAL_ON_INDEX	Unique constraint violated
ORA-0051	TIMEOUT_ON_RESOURCE	Time-out occurred while waiting for resource
ORA-0061	TRANSACTION_BACKED_OUT	The transaction was rolled back due to deadlock
ORA-1001	INVALID_CURSOR	Illegal cursor operation
ORA-1012	NOT_LOGGED_ON	Not connected to Oracle
ORA-1017	LOGIN DEFINED	Invalid user name/password
ORA-1403	NO_DATA_FOUND	No data found

ORA-1422	TOO_MANY_ROWS	A SELECT..INTO statement matches more than one row
ORA-1476	ZERO_DIVIDE	Division by zero
ORA-1722	INVALID_NUMBER	Conversion to a number failed—for example,’1A’ is not valid
ORA-6500	STORAGE_ERROR	Internal PL/SQL error raised if PL/SQL runs out of memory
ORA-6501	PROGRAM_ERROR	Internal PL/SQL error
ORA-6502	VALUE_ERROR	Truncation, arithmetic, or conversion error
ORA-6504	ROWTYPE_MISMATCH	Host cursor variable and PL/SQL cursor variable have incompatible row types
ORA-6511	CURSOR_ALREADY_OPEN	Attempt to open a cursor that is already open

Table 3.2

- Short descriptions of some of the predefined exceptions follow

INVALID_CURSOR

- This error is raised when an illegal cursor operation is performed, such as attempting to close a cursor that is already closed

- The analogous situation of attempting to open a cursor that is already open causes CURSOR_ALREADY_OPEN to be raised

NO_DATA_FOUND

- This exception can be raised when a SELECT..INTO statement does not return any rows. If the statement returns more than one row, TOO_MANY_ROWS is raised

INVALID_NUMBER

- This exception is raised in an SQL statement when an attempt to converting string to a number fails

Ex:

```
declare
        a number:=4;
        c number;
begin
        c:=a/0;
exception
        when ZERO_DIVIDE then
        dbms_output.put_line('divide by zero exception');
end;
```

User-Defined Exceptions

- A user-defined exception is an error that is defined by the program

- User-defined exceptions are declared in the declarative section of a PL/SQL block. Just like variables, exceptions have a type (EXCEPTION)

- For example:

```
DECLARE
            ex EXCEPTION;
```

Raising Exceptions

- When the error associated with an exception occurs, the exception is raised

- User-defined exceptions are raised explicitly via the RAISE statement, while predefined exceptions are raised implicitly when their associated Oracle error occurs

Ex:

```
declare
        a number(2);
        b number(2);
        c number(2);
        ex exception;
begin
        a:=&a;
        b:=&b;
        if(b=0) then
                raise ex;
        else
                c:=a/b;
                dbms_output.put_line(c);
        end if;
exception
        when ex then
                dbms_output.put_line('divide by zero');
        end;
```

Handling Exceptions

- When an exception is raised, control passes to the exception section of the block

- The exception section consists of handlers for all the exceptions

- An exception handler contains the code that is executed when the error associated with the exception occurs, and the exception is raised

- The syntax for the exception section is as follows:

```
EXCEPTION
        WHEN exception_name THEN
                sequence_of_statements1;
        WHEN exception_name THEN
                sequence_of_statements2;
        WHEN OTHERS THEN
                sequence_of_statements3;
END
```

The OTHERS Exception Handler

- The OTHERS handler will execute when exception does not match with the list of exceptions. It should always be the last handler in the block

Example program

```
set serveroutput on

DECLARE
s_sname sailors.sname%TYPE;
BEGIN

select sname into s_sname from sailors where sid=75;

EXCEPTION
  WHEN NO_DATA_FOUND THEN
           dbms_output.put_line('NO DATA WAS FOUND!');
  WHEN OTHERS THEN
           dbms_output.put_line('SELECT CANNOT BE EXECUTED!');
END;
```

Output: NO DATA WAS FOUND!

CURSORS

- Oracle creates a memory area, known as context area, for processing an SQL statement, which contains all information needed for processing the statement, for example, number of rows processed, etc.

- A cursor is a pointer to this context area.

- PL/SQL controls the context area through a cursor.

- There are two types of cursors:
  - Implicit cursors
  - Explicit cursors

Implicit Cursors

- Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement.

- Programmers cannot control the implicit cursors and the information in it.

- In PL/SQL, you can refer to the most recent implicit cursor as the **SQL cursor**, which always has the attributes like %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT

- Any SQL cursor attribute will be accessed as **sql%attribute_name**

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

EMP_NO	EMP_NAME	EMP_DEPT	EMP_SALARY
1	Forbs ross	Web Developer	45000
2	marks jems	Program Developer	38000
3	Saulin	Program Developer	34000
4	Zenia Sroll	Web Developer	42000

Ex:

```
DECLARE
   total_rows number(2);
BEGIN
   UPDATE customers SET salary = salary + 5000;
  IF sql%notfound THEN
    dbms_output.put_line('no customers selected');
  ELSIF sql%found THEN
   total_rows := sql%rowcount;
   dbms_output.put_line( total_rows || ' customers selected ');
  END IF;
END;
```

Explicit cursors

- Explicit cursors are programmer defined cursors for gaining more control over the **context area**.

- An explicit cursor should be defined in the declaration section of the PL/SQL Block.

- It is created on a SELECT Statement which returns more than one row.

- The four PL/SQL steps necessary for explicit cursor processing are as follows:

1. Declare the cursor
2. Open the cursor for a query
3. Fetch the results into PL/SQL variables
4. Close the cursor

- The following PL/SQL program illustrates a cursor

```
DECLARE
s_id number(2);
s_name varchar2(20);
s_rating number(2);
s_age number(3,1);

Cursor  C_Sailors   IS
SELECT sid,sname,rating,age
FROM  sailors
where rating=7;
```

```
BEGIN
OPEN  C_Sailors;
LOOP
    /* Retrieve each row of the active set into PL/SQL variables */
    FETCH C_Sailors INTO s_id,s_name,s_rating,s_age;
   /* If there are no more rows to fetch, exit the loop */
   EXIT WHEN C_Sailors%NOTFOUND;
   dbms_output.put_line(s_id||s_name||s_rating||s_age);
END LOOP;

 /*Free resources used by the query */
 CLOSE   C_Sailors;
END;
```

Output:

```
22 Dustin 7 45
64 Horatio 7 35
```

PROCEDURES AND FUNCTIONS

- These are also known as subprograms

PROCEDURE

Creating a Procedure

The syntax for the CREATE OR REPLACE PROCEDURE statement is

```
CREATE [OR REPLACE] PROCEDURE  procedure_name
                 [(argument[{IN|OUT|INOUT}] type,
                 …
                 argument[{IN|OUT|INOUT)] {IS|AS}
                 procedure_body
```

- procedure_name is the name of the procedure to be created
- argument is the name of a procedure parameter
- type is the type of the associated parameter
- procedure_body is a PL/SQL block
- In order to change the code of a procedure, the procedure must be dropped and then re-created. The OR REPLACE keywords allow this to be done in one operation

- IN -- parameter is considered read-only-it cannot be changed
- OUT -- parameter is considered write-only-it can only be assigned to and cannot be read from
- INOUT -- parameter can be read from and written to

Structure of a procedure:

```
CREATE OR REPLACE PROCEDURE procedure_name
AS
        Declarative section
        Executable section
EXCEPTION
        Exception section
END [procedure_name];
```

Example Program:

```
CREATE OR REPLACE PROCEDURE
circle_area_circum (radius in number,area out number,
circumference out number) is

BEGIN
area:=3.14 * radius * radius;
circumference:=2 * 3.14 * radius;
END circle_area_circum;
/
```

Procedure Usage for the above:

```
set serveroutput on
```

```
DECLARE
r number(10):=2;
a number(10,3):=0;
c number(10,5):=0;

BEGIN
circle_area_circum(r,a,c);    --- call to procedure
dbms_output.put_line(r||'  '||a||'   '||c);
END;
/
```

Out put:

```
2    12.56   12.56
```

FUNCTION

Creating a Function

- A function is very similar to a procedure
- Both take arguments, which can be of different modes
- Both are different forms of PL/SQL blocks, with a declarative executable, and exception section
- Both can be stored in the database or declared within a block
- However, a procedure call is a PL/SQL statement by itself, while a function call is called as part of an expression

Function Syntax

- The syntax for creating a stored function is very similar to the syntax for a procedure

```
CREATE[OR REPLACE] FUNCTION function_name
        [(argument[{IN|OUT|INOUT}]type,
        …
        argument[{IN|OUT|INOUT}]type)]
        RETURN return_type{IS|AS}
        Function_body
```

- RETURN statement is used to return control to the calling environment with a value

- The general syntax of the RETURN statement is

  RETURN expression;

- where expression is the value to be returned

Example Program:

```
create or replace function concatenate(pfirst in varchar2,
plast in varchar2) return varchar2 is
begin
return pfirst||plast;  /* returning concatenated name  */
end concatenate;
```

Function Usage for the above:

```
set serveroutput on
declare
```

```
fname varchar2(10):='sree';
lname varchar2(10):='nidhi';
coname varchar2(20);
begin
coname:=concatenate(fname,lname);
dbms_output.put_line(coname);
end;
```

Output: sreenidhi

Note: function can be dropped using the following command.

```
      DROP FUNCTION function_name;
```

EMP_NO	EMP_NAME	EMP_DEPT	EMP_SALARY
1	Forbs ross	Web Developer	45000
2	marks jems	Program Developer	38000
3	Saulin	Program Developer	34000
4	Zenia Sroll	Web Developer	42000

Example: create a function to return employee name based on the employee number

```
CREATE or REPLACE FUNCTION fun1(no in number) RETURN varchar2 IS
name varchar2(20);
BEGIN
  select emp_name into name from emp where emp_no = no;
  return name;
END;
```

Usage:

```
select fun1(2) from emp;          o/p: marks jems
          (or)
```

Call function from the PL/SQL program

Packages

- Packages are PL/SQL constructs that allow related objects to be stored together
- A package has two separate parts - specification and body

Package Specification

- The package specification (also known as the package header) contains information about the contents of the package

Syntax:

```
CREATE[OR REPLACE]PACKAGE package_name
{IS|AS}
        procedure_specification|
        function_specification|
        variable_declaration|
        type_definition|
        exception_declaration|
        cursor_declaration
END [package_name];
```

- All objects placed in the specification are called **public** objects. Any subprogram not in the package specification but coded in the package body is called a **private** object.

Package Body

- The package body is a separate data dictionary object from the package header
- It cannot be successfully compiled without the header
- The body contains the code for the forward subprogram declarations in the package header

Package Initialization

```
CREATE OR REPLACE PACKAGE BODY
package_name {IS|AS}
        …
BEGIN
        Initialization_code;
END [package_name];
```

Example Program:

Package Specification:

```
CREATE OR REPLACE PACKAGE DEMO AS
-- Function Specs goes here
FUNCTION fact(n in number) return number;
-- Procedure one specs goes here
PROCEDURE circle(radius in number, area out number,
cir out number);
-- Procedure two Specs goes here
PROCEDURE Aggregation(minimum out number,
maximum out number, sum1 out number);
END DEMO;
```

Example Program:
package body:

```
CREATE OR REPLACE PACKAGE BODY DEMO AS
FUNCTION fact (n in number) return number is
pno number(3):= 0;
c   number:= 1;
BEGIN
 pno:= n;
 WHILE pno >= 1
  Loop
            c:= c * pno;
            pno:= pno - 1;
  END LOOP;
return c;
END fact;
```

```
PROCEDURE circle (radius in number, area out number, cir out
number) AS
BEGIN
area:= (22/7) * radius * radius;
cir:= 2* (22/7) * radius;
END CIRCLE;
PROCEDURE Aggregation(minimum out number, maximum out
number, sum1 out number) AS
BEGIN
SELECT MIN(rating) INTO minimum FROM sailors;
SELECT MAX(rating) INTO maximum FROM sailors;
SELECT SUM(rating) INTO sum1 FROM sailors;
END Aggregation;
END DEMO;
```

Example Program:

Package usage:

```
SET SERVEROUTPUT ON
DECLARE
        x1 number:= 2; -- radius
        f1 number:= 5; -- fact
        x  number:= 0;
        sa number:= 0;
        sb number:= 0;
        smin number:= 0;
        smax number:= 0;
        ssum number:= 0;
BEGIN
   x:= demo.FACT(f1);
  dbms_output.put_line('Factorial for  '||f1||' is '||x);
  demo.CIRCLE(x1,sa,sb);
  dbms_output.put_line('Area is '||sa||'   '||'Circumference is '||sb);
  demo.Aggregation(smin,smax,ssum);
  dbms_output.put_line('Min '||smin||'   '||'Max '||smax||' Sum '||ssum);
END;
```

Note: package can be dropped using the following command.

```
    DROP PACKAGE package_name;
```

Triggers

•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

Syntax for Creating a Trigger

```
CREATE [OR REPLACE] TRIGGER trigger_name
   BEFORE | AFTER | INSTEAD OF
   [INSERT, UPDATE, DELETE [COLUMN NAME..]
   ON table_name
    Referencing [ OLD AS OLD | NEW AS NEW ]
    FOR EACH ROW | FOR EACH STATEMENT [ WHEN Condition ]
DECLARE
    [declaration_section;]
BEGIN
      [executable_section;]
EXCEPTION
      [exception_section;]
END;
```

Inserting Trigger

This trigger execute BEFORE to convert ename field lowercase to uppercase.

```
CREATE or REPLACE TRIGGER trg1
    BEFORE
    INSERT ON emp1
    FOR EACH ROW
BEGIN
    :new.ename := upper(:new.ename);
END;
/
```

Restriction to Deleting Trigger

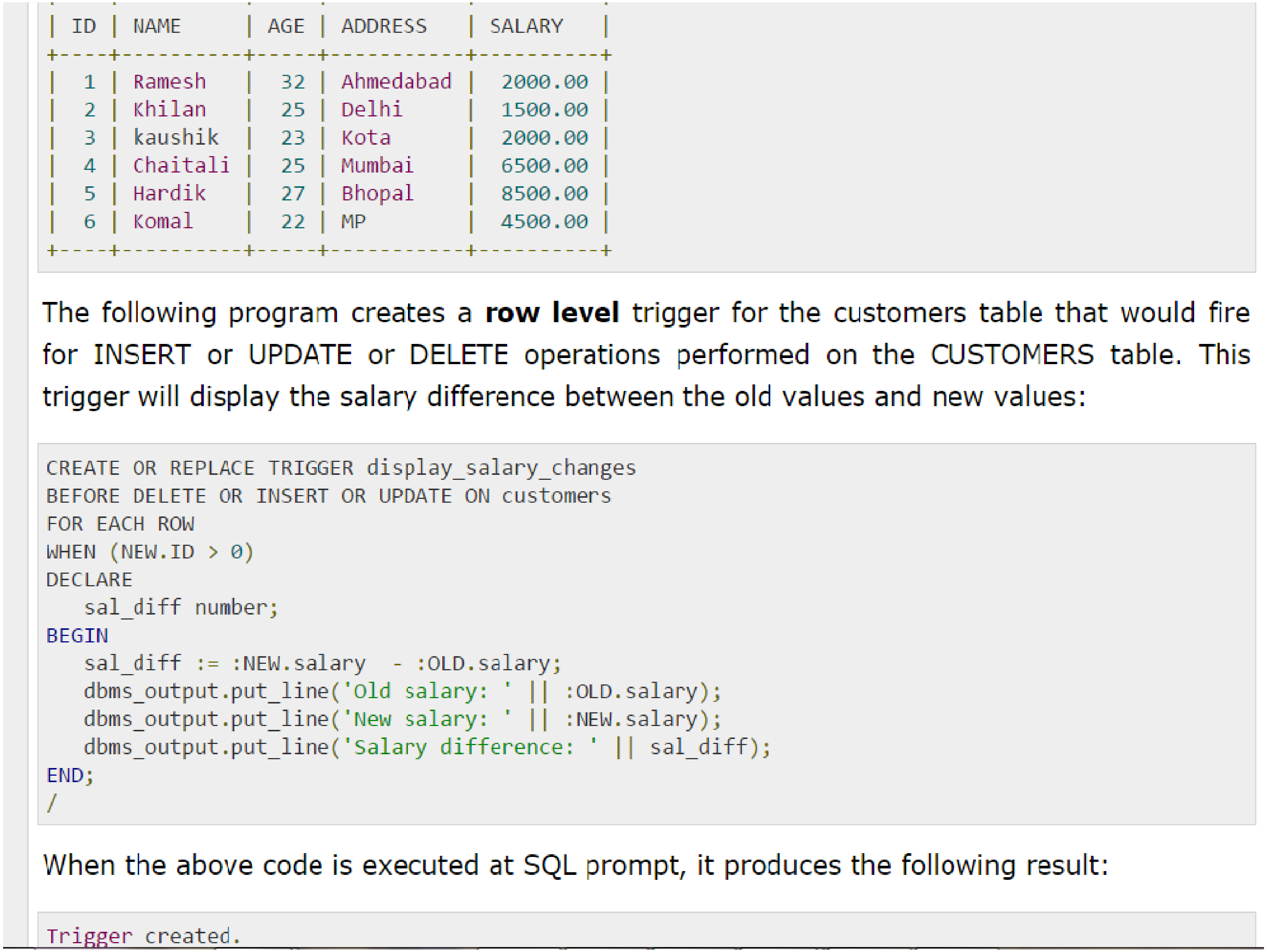
This trigger is preventing to deleting row.

Delete Trigger Code:

```
CREATE or REPLACE TRIGGER trg1
    AFTER
    DELETE ON emp1
    FOR EACH ROW
BEGIN
    IF :old.eno = 1 THEN
        raise_application_error(-20015, 'You can't delete this row');
    END IF;
END;
/
```

Delete Trigger Result :

```
SQL>delete from emp1 where eno = 1;
Error Code: 20015
Error Name: You can't delete this row
```



```
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

The following program creates a **row level** trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values:

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON customers
FOR EACH ROW
WHEN (NEW.ID > 0)
DECLARE
   sal_diff number;
BEGIN
   sal_diff := :NEW.salary  - :OLD.salary;
   dbms_output.put_line('Old salary: ' || :OLD.salary);
   dbms_output.put_line('New salary: ' || :NEW.salary);
   dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Trigger created.
```

Let us perform some DML operations on the CUSTOMERS table. Here is one INSERT statement, which will create a new record in the table:

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (7, 'Kriti', 22, 'HP', 7500.00 );
```

When a record is created in CUSTOMERS table, above create trigger **display_salary_changes** will be fired and it will display the following result:

```
Old salary:
New salary: 7500
Salary difference:
```

Because this is a new record so old salary is not available and above result is coming as null. Now, let us perform one more DML operation on the CUSTOMERS table. Here is one UPDATE statement, which will update an existing record in the table:

```
UPDATE customers
SET salary = salary + 500
WHERE id = 2;
```

When a record is updated in CUSTOMERS table, above create trigger **display_salary_changes** will be fired and it will display the following result:

```
Old salary: 1500
New salary: 2000
Salary difference: 500
```