Programming in Oracle with PL/SQL

Procedural Language Extension to SQL

PL/SQL

- Allows using general programming tools with SQL, for example: loops, conditions, functions, etc.
- This allows a lot more freedom than general SQL, and is lighter-weight than JDBC.
- We write PL/SQL code in a regular file, for example PL.sql, and load it with @PL in the sqlplus console.

PL/SQL Blocks

- PL/SQL code is built of Blocks, with a unique structure.
- There are two types of blocks in PL/SQL:

1. **Anonymous Blocks:** have no name (like scripts)
   - can be written and executed immediately in SQLPLUS
   - can be used in a trigger
2. **Named Blocks:**
   - Procedures
   - Functions

Anonymous Block Structure:

DECLARE (optional)

/* Here you declare the variables you will use in this block */

BEGIN (mandatory)

/* Here you define the executable statements (what the block DOES!)*/

EXCEPTION (optional)

/* Here you define the actions that take place if an exception is thrown during the run of this block */

END; (mandatory)

/

Always put a new line with only a / at the end of a block! (This tells Oracle to run the block)

A correct completion of a block will generate the following message:

PL/SQL procedure successfully completed

An Example of PL/SQL

```
DECLARE
    NO      NUMBER(4);
     I        NUMBER(4);
     FACT NUMBER(4);
BEGIN
     NO := &NO;
     I :=  NO;
     FACT := 1;
     WHILE ( I >= 1)
     LOOP
           FACT := FACT  *  I;
           I := I – 1;
     END LOOP;
     DBMS_OUTPUT.PUT_LINE(‘FACTORIAL OF ‘ ||NO||’ IS ‘||FACT);
END;
/
```

DECLARE

Syntax

```
identifier [CONSTANT] datatype [NOT NULL]
    [:= | DEFAULT expr];
```

Examples

Notice that PL/SQL includes all SQL types, and more…

```
Declare
  birthday      DATE;
  age             NUMBER(2) NOT NULL := 27;
  name           VARCHAR2(13)  := 'Levi';
  magic         CONSTANT NUMBER := 77;
  valid         BOOLEAN NOT NULL := TRUE;
```

Declaring Variables with the %TYPE Attribute

Examples

Accessing column sname in table Sailors

```
DECLARE
   sname                       Sailors.sname%TYPE;
   fav_boat                    VARCHAR2(30);
   my_fav_boat                 fav_boat%TYPE :=
'Pinta';
...
```

Accessing another variable

Declaring Variables with the %ROWTYPE Attribute

Declare a variable with the type of a ROW of a table.

Accessing table Reserves

```
reserves_record
Reserves%ROWTYPE;
```

And how do we access the fields in reserves_record?

```
reserves_record.sid:=9;
Reserves_record.bid:=877;
```

Creating a PL/SQL Record

A record is a type of variable which we can define (like ‘struct’ in C or ‘object’ in Java)

```
DECLARE
   TYPE sailor_record_type IS RECORD
      (sname        VARCHAR2(10),
       sid          VARCHAR2(9),
       age          NUMBER(3),
      rating        NUMBER(3));
   sailor_record   sailor_record_type;
...
BEGIN
   Sailor_record.sname:=‘peter’;
   Sailor_record.age:=45;
…
```

Cursor

> Enables users to loop round a selection of data.

> Stores data select from a query in a temp area for use when opened.

> Use complex actions which would not be feasible in standard SQL selection queries

Syntax for Cursors

- Declared as a variable in the same way as standard variables
- Identified as cursor type
- SQL included

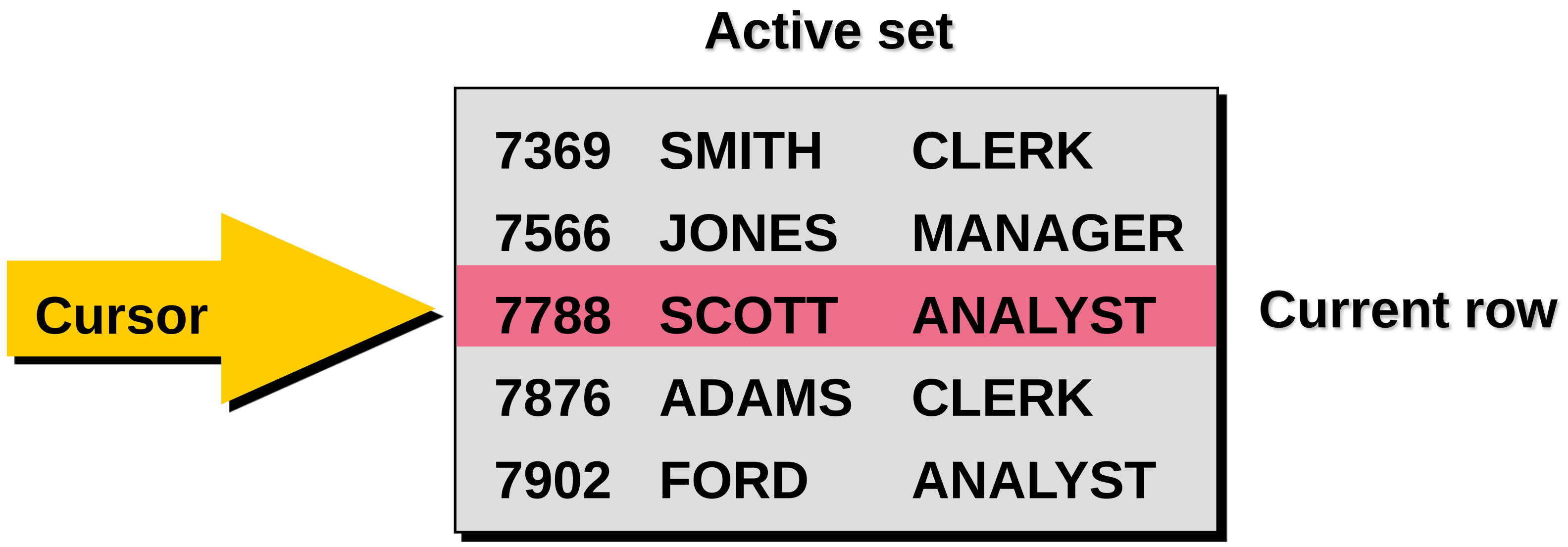
E.g.

```
Cursor cur_emp is
      Select emp_id, surname ‘name’, grade,
salary
            From employee
               Where regrade is true;
```

Cursor

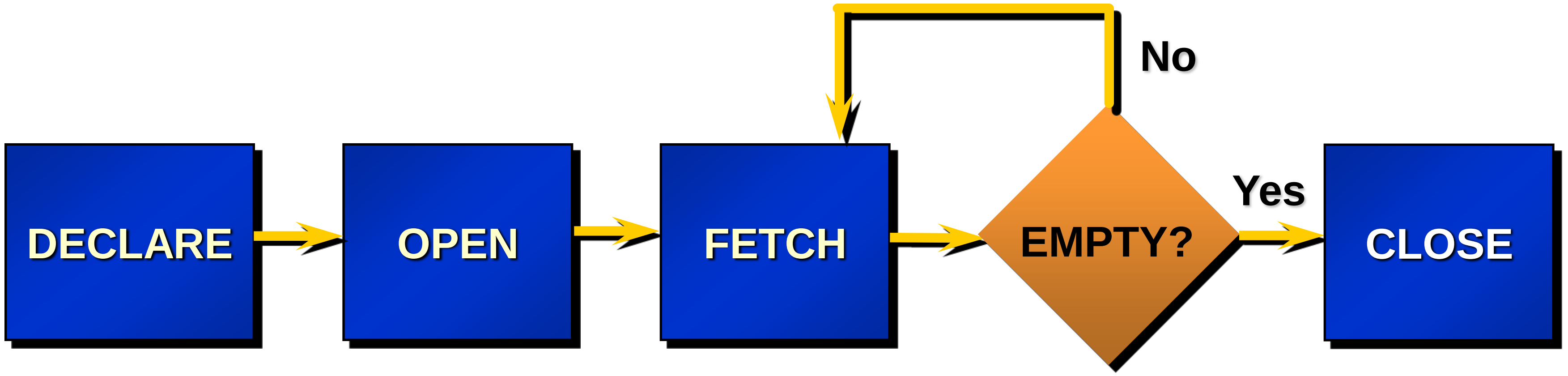
- A cursor is a temp store of data.
- The data is populated when the cursor is opened.
- Once opened the data must be moved from the temp area to a local variable to be used by the program. These variables must be populated in the same order that the data is held in the cursor.
- The data is looped round till an exit clause is reached.

Cursor Functions



7369	SMITH	CLERK
7566	JONES	MANAGER
7788	SCOTT	ANALYST
7876	ADAMS	CLERK
7902	FORD	ANALYST

Controlling Cursor



- Create a named SQL area

- Identify the active set

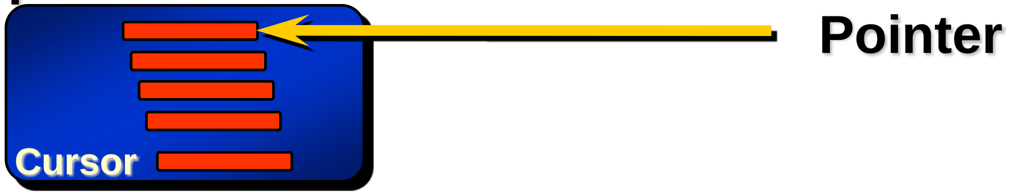
- Load the current row into variables

- Test for existing rows
- Return to FETCH if rows found

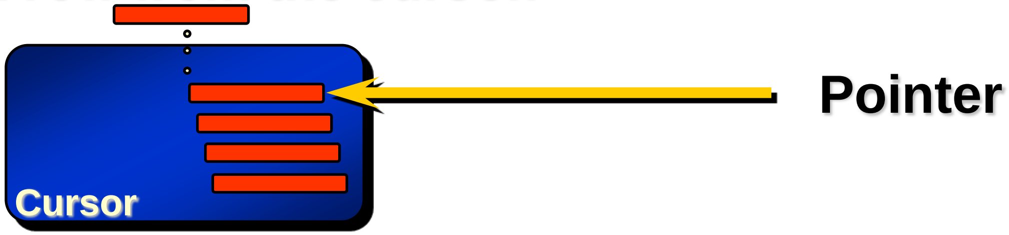
- Release the active set

Controlling Cursor…

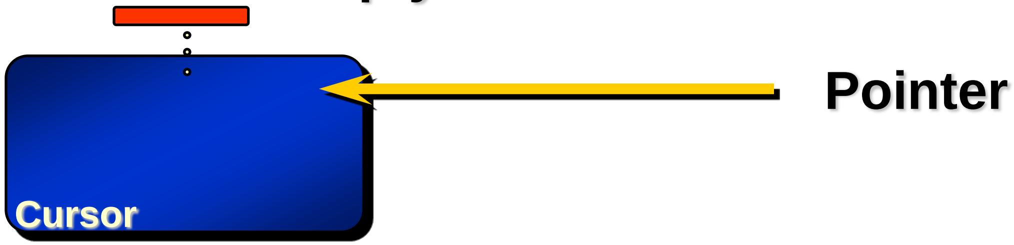
Open the cursor.



Fetch a row from the cursor.



Continue until empty.

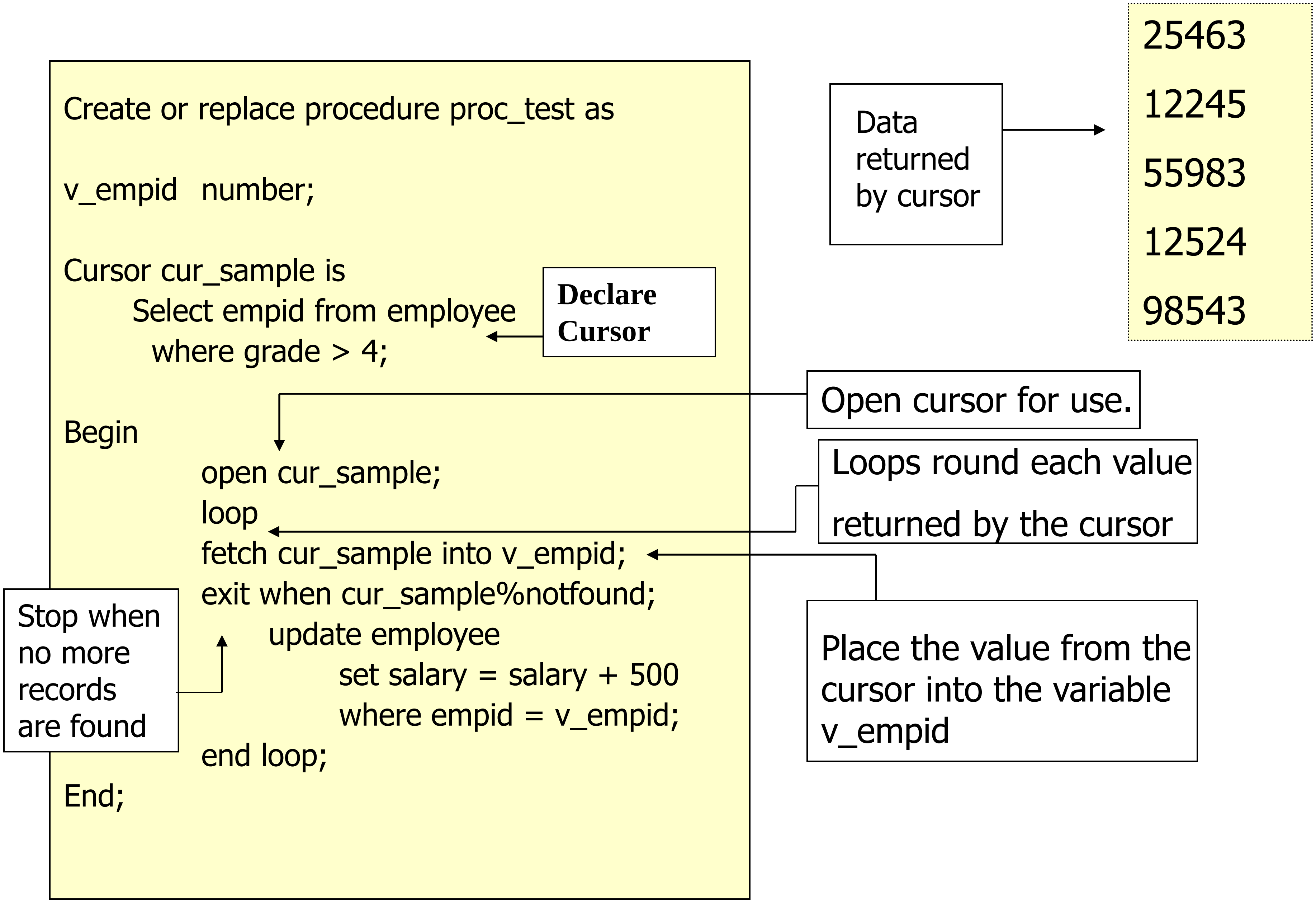


Close the cursor.

Cursor Attributes

Obtain status information about a cursor.

Attribute	Type	Description
%ISOPEN	Boolean	Evaluates to TRUE if the cursor is open
%NOTFOUND	Boolean	Evaluates to TRUE if the most recent fetch does not return a row
%FOUND	Boolean	Evaluates to TRUE if the most recent fetch returns a row; complement of %NOTFOUND
%ROWCOUNT	Number	Evaluates to the total number of rows returned so far



```
Create or replace procedure proc_test as

v_empid   number;

Cursor cur_sample is
       Select empid from employee
         where grade > 4;

Begin
             open cur_sample;
             loop
             fetch cur_sample into v_empid;
             exit when cur_sample%notfound;
                    update employee
                           set salary = salary + 500
                           where empid = v_empid;
             end loop;
End;
```

Declare Cursor
Data returned by cursor
25463
12245
55983
12524
98543
Open cursor for use.
Loops round each value returned by the cursor
Place the value from the cursor into the variable v_empid
Stop when no more records are found

Cursor FOR Loops

- Syntax

```
FOR record_name IN cursor_name LOOP
  statement1;
  statement2;
  . . .
END LOOP;
```

- The cursor FOR loop is a shortcut to process cursors.
- Implicitly opens, fetches, and closes cursor.
- The record is implicitly declared.

Cursor FOR Loops: An Example

- Retrieve employees one by one until no more are left.
- Example

```
DECLARE
  CURSOR emp_cursor IS
    SELECT ename, deptno
    FROM   emp;
BEGIN
  FOR emp_record IN emp_cursor LOOP
         -- implicit open and implicit fetch occur
    IF emp_record.deptno = 30 THEN
      ...
  END LOOP; -- implicit close occurs
END;
```

SQL Cursor (Implicit Cursor)

SQL cursor is automatically created after each SQL query. It has 4 useful attributes:

SQL%ROWCOUNT	**Number of rows affected by the most recent SQL statement (an integer value).**
SQL%FOUND	**Boolean attribute that evaluates to TRUE if the most recent SQL statement affects one or more rows.**
SQL%NOTFOUND	**Boolean attribute that evaluates to 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.**

An Example of Implicit (SQL) Cursor

```
BEGIN
     UPDATE EMP_MSTR SET BRANCH_NO = &BRANCH_NO
                                                       WHERE  EMP_NO = &EMP_NO;
     IF SQL%FOUND THEN
            dbms_output.put_line(‘Employee Successfully Transferred’);
     END IF;
     IF SQL%NOTFOUND THEN
           dbms_output.put_line(‘Employee Number Does Not Exists’);
     END IF;
END;
/
```

Printing Output

- You need to use a function in the DBMS_OUTPUT package in order to print to the output
- If you want to see the output on the screen, you must type the following (before starting):

  set serveroutput on format wrapped size 1000000

- Then print using
  - dbms_output. put_line(*your_string*);
  - dbms_output.put(*your_string*);

Input and output example

```
set serveroutput on format wrap size 1000000
ACCEPT high PROMPT 'Enter a number: '

DECLARE
i   number_table.num%TYPE:=1;
BEGIN
  dbms_output.put_line('Look Ma, I can print from PL/SQL!!!');
  WHILE i <= &high LOOP
     INSERT INTO number_table
     VALUES(i);
     i := i + 1;
  END LOOP;
END;
```

Reminder- structure of a block

```
DECLARE          (optional)
    /* Here you declare the variables you will use in this
       block */
BEGIN            (mandatory)
    /* Here you define the executable statements (what
       the block DOES!)*/
EXCEPTION        (optional)
    /* Here you define the actions that take place if an
       exception is thrown during the run of this block */
END;             (mandatory)
/
```

Functions and Procedures

- Up until now, our code was in an anonymous block
- It was run immediately
- It is useful to put code in a function or procedure so it can be called several times
- Once we create a procedure or function in a Database, it will remain until deleted (like a table).

Creating Procedures

```
CREATE [OR REPLACE] PROCEDURE
procedure_name
 [(parameter1 [mode1] datatype1,
  parameter2 [mode2] datatype2,
  . . .)]
IS|AS
PL/SQL Block;
```

- Modes:
  - IN: procedure must be called with a value for the parameter. Value cannot be changed
  - OUT: procedure must be called with a variable for the parameter. Changes to the parameter are seen by the user (i.e., call by reference)
  - IN OUT: value can be sent, and changes to the parameter are seen by the user
- Default Mode is: IN

Example

```
create or replace procedure
P1 IS
 a number(2) := 2;
 b number(2) := 3;
BEGIN
 dbms_output.put_line(a+b);
END P1;
```

Syntax for executing the procedure:

Syntax:
exec Procedure_Name;
e.g.
exec p1;

Calling the Procedure

```
declare
begin
    p1;
end;
/
```

Deleting A Stored Procedure

A Prcedure can be deleted by using the following syntax.

Syntax:

```
DROP PROCEDURE <ProcedureName>;
```

Example:

```
DROP PROCEDURE P1;
```

Errors in a Procedure

- When creating the procedure, if there are errors in its definition, they will not be shown
- To see the errors of a procedure called *myProcedure*, type

```
SHOW ERRORS PROCEDURE myProcedure
```

  in the SQLPLUS prompt
- For functions, type

```
SHOW ERRORS FUNCTION myFunction
```

Creating a Function

- Almost exactly like creating a procedure, but you supply a return type

```
CREATE [OR REPLACE] FUNCTION
function_name
 [(parameter1 [mode1] datatype1,
  parameter2 [mode2] datatype2,
  . . .)]
RETURN datatype
IS|AS
PL/SQL Block;
```

A Function

```
create or replace function
rating_message(rating IN NUMBER)
return VARCHAR2
AS
BEGIN
   IF rating > 7 THEN
      return 'You are great';
   ELSIF rating >= 5 THEN
      return  'Not bad';
   ELSE
      return 'Pretty bad';
   END IF;
END;
/
```

NOTE THAT YOU DON'T SPECIFY THE SIZE

Calling the function

```
declare
    paulRate:=9;
Begin
dbms_output.put_line(ratingMessage(paulRate));
end;
/
```

Creating a function:

```
create or replace function squareFunc(num in number)
return number
is
BEGIN
return num*num;
End;
/
```

Using the function:

```
BEGIN
dbms_output.put_line(squareFunc(3.5));
END;
/
```

Deleting A Stored Function

A Function can be deleted by using the following syntax.

Syntax:

```
DROP FUNCTION <FunctionName>;
```

Example:

```
DROP FUNCTION squareFunc;
```

Triggers

- Triggers are special procedures which we want activated when someone has performed some action on the DB.
- For example, we might define a trigger that is executed when someone attempts to insert a row into a table, and the trigger checks that the inserted data is valid.

Types of Triggers

1. Row Triggers:

> A row trigger is fired each time a row in the table is affected by the triggering statement.

> For example, if an UPDATE statement updates multiple rows of a table, a row trigger is fired once for each row affected by the UPDATE statement.

> If the triggering statement affects no rows, the trigger is not executed at all.

> Row triggers should be used when some processing is required whenever a triggering statement affects a single row in table.

2. Statement Triggers:

> A statement trigger is fired once on behalf of the triggering statement, independent of the number of rows the triggering statement affects (even if no rows are affected).

> Statement triggers should be used when a triggering statement affects rows in a table but the processing required is completely independent of the number of rows affected.

Before V/S After Triggers

Before Trigger:

BEFORE triggers execute the trigger action before the triggering statement. These types of triggers are commonly used in the following situations:

> BEFORE triggers are used when the trigger action should determine whether or not the triggering statement should be allowed to complete. By using BEFORE trigger, user can eliminate unnecessary processing of the triggering statement.

> BEFORE triggers are used to derive specific column values before completing a triggering INSERT or UPDATE statement.

After Triggers:

AFTER trigger executes the trigger action after the triggering statement is executed. These types of triggers are commonly used when the triggering statement should used in the following situations:

> AFTER triggers are used when the triggering statement should complete before executing the trigger action.

> If a BEFORE trigger is already present, an AFTER trigger can perform different actions on the same triggering statement.

Syntax of Trigger

```
CREATE OR REPLACE TRIGGER
<trigger_name>
[BEFORE/AFTER]
[DELETE/INSERT/UPDATE OF <column_name]
ON <table_name>
[FOR EACH ROW]
BEGIN
          PL/SQL instructions;

          ...............
END;
```

Example

```
Create or replace trigger t1
        after insert on Emp_Mstr
Begin
 dbms_output.put_line('New Row
                       inserted');
End t1;
```

Deleting A Trigger

A Trigger can be deleted by using the following syntax.

Syntax:

```
DROP TRIGGER <TriggerName>;
```

Example:

```
DROP TRIGGER t1;
```

Thank you!!!

By : Vishal Parikh
Assistant Professor, CE Dept.
Nirma Institute of Technology,
Nirma University.