PL/SQL PROGRAMMING

OBJECTIVES:

- Procedure
- Functions
- Cursors
- Triggers

Procedure

- A ***PL/SQL procedure*** is a named block that performs one or more actions. It allows you to wrap complex business logic and reuse it.
- It is a subprogram that performs a specific action.
- Can be called from any PL/SQL program.
- Has two parts:
  - Specification
    - Begins with the keyword PROCEDURE and ends with the procedure name or a parameter list
  - Body
    - Begins with the keyword IS (or AS) and ends with the keyword END followed by an optional procedure name
- Declarative Part
  - It contains local declarations
  - Keyword DECLARE is NOT used to declare variables and exceptions.
  - All declarations are placed in between the IS and BEGIN keywords.
- Executable Part
  - It contains statements placed between the keywords BEGIN and EXCEPTION (or END).
  - At least one statement must appear in executable part of a procedure.
- Exception Handling Part
  - Contains exception handlers placed between keywords EXCEPTION and END.

Note: At least one statement must appear in the executable part of a procedure (may be NULL). A procedure is called a PL/SQL statement.

Syntax:

```
CREATE [OR REPLACE] PROCEDURE <procedure-name>[(parameter[, parameter]...)]
{IS | AS}
Variable  declarations;

BEGIN
        PL/SQL executable statements
```

```
[EXCEPTION exception handlers]
END  [PROCEDURE_NAME];
/
```

Note: OR REPLACE option is used to modify a previously written stored procedure.

Invoking a procedure

- A procedure can be called from any PL/SQL program by specifying their names followed by the parameters.
  Syntax:
  <procedure_name>[(parameter1,   paremeter2,…)];

- Procedures can also be invoked from the SQL prompt, using the EXECUTE command.
  Syntax:
  SQL>EXECUTE <procedure_name>[(parameter1, paremeter2,…)];

Example: Let's create a procedure FINDMIN to find out minimum of two numbers.

```
PROCEDURE findMin(x IN number, y IN number, z OUT number)
IS
BEGIN
   IF x < y THEN
      z:= x;
   ELSE
      z:= y;
   END IF;
 END;
```

Above procedure taking two numbers as input values using the IN mode and returns minimum of two numbers using the OUT parameters.

An IN parameter lets you pass a value to the subprogram. **It is a read-only parameter**. An OUT parameter returns a value to the calling program. Inside the subprogram, an OUT parameter acts like a variable. **IN OUT**

An **IN OUT** parameter passes an initial value to a subprogram and returns an updated value to the caller. It can be assigned a value and the value can be read.

Now, execute above stored procedure code as follows by writing a PL/SQL blokc as follows:

```
DECLARE
   a number;
   b number;
   c number;
  BEGIN
   a:= 23;
   b:= 45;
   findMin(a, b, c);

   dbms_output.put_line(' Minimum of (23, 45) : ' || c);

  findMin(30,10, c);
 dbms_output.put_line(' Minimum of ('||a||',' ||b||') is :  '||c);

END;
/
```

Observer the outcome and code.

Task 1: Let's create a procedure by the name INCR_RATING in which sailors id i.e. SID and rating value to which increment made are passed as parameter.

```
CREATE OR REPLACE PROCEDURE proc_incr_rating (p_sno NUMBER,p_increment   NUMBER)
        IS
                                vrating NUMBER;
                        BEGIN
                                SELECT rating
                                INTO vrating
                                FROM sailors
                                WHERE sid=p_sno;
                                UPDATE sailors
                                SET rating=rating+p_increment
                                 WHERE sid=p_sno;
                         END proc_incr_rating;
                                 /
```

- **How to invoke the procedure:**

  //First displays the content of sailors table and remembers ratings of sid 29.

```
SET SERVEROUTPUT ON SIZE 1000000;
execute proc_incr_rating(29,1);
```

  //Now display the content of sailors table and observer ratings of sid 29

<u>Parameters:</u>

- Subprograms pass information using parameters.
- Types of parameters are:
  - Formal parameters:
    - Are variables declared in a subprogram specification and referenced in the subprogram body
  - Actual parameters:
    - Are variables or expressions referenced in the parameter list of subprogram call.

Formal Parameters:

```
Create PROCEDURE proc_ incr_rating (p_sno number, p_increment number) IS
BEGIN
 UPDATE sailors
      SET rating = rating + p_increment
      WHERE sid = p_sno;
END proc_incr_rating;
/
```

Anonymous PL/SQL Block Calling the Procedure

Actual Parameters:

```
DECLARE
        var_sno   sailors.sid%TYPE  := &var_sno;
        var_increment sailors.rating%TYPE := &var_increment;
BEGIN
        proc_incr_rating(var_sno,   var_increment);
END;
/
```

➢ **<u>FUNCTIONS:</u>** A Function is a subprogram which performs an action and returns a value**.** It can be stored in the database as a schema object for repeated execution.

Creating a Function

A function has two parts:

- Function specification
  - Begins with the keyword FUNCTION and ends with the RETURN clause, which specifies the data type of the return value.
- Function body
  - Begins with the keyword IS (or AS) and ends with the keyword END followed by an optional function name.

RETURN Statement

➢ RETURN statement immediately returns control to the caller environment.

➢ Execution then resumes with the statement following the subprogram call.

➢ RETURN statement used in functions must contain an expression, which is evaluated when the RETURN statement is executed.

➢ The resulting value is assigned to the function identifier. Therefore, a function must contain at least one RETURN statement.

Syntax:
```
CREATE [OR REPLACE] FUNCTION function_name[(parameter[, parameter]...)]
 RETURN type
 {IS | AS}

 variable declarations;
 BEGIN
   function_body


[EXCEPTION exception handlers]
END  [function_name];
 /
```

Task 2: To see how a function can be created; Let's first create one table and insert data into it:

```
create table Employee(
            empno number,
            sal number,
            comm number
   );
insert into employee values (1,100,10);
insert into employee values (2,200,20);
insert into employee values (3,300,30);
```

Let's creates a function to calculate net salary based on various conditions. Employee number is passed as parameter. The function returns the value of the increment (refer above employee table).

```
CREATE OR REPLACE FUNCTION func_revised_salary (p_empno  NUMBER)
RETURN NUMBER
IS
  vincr  employee.sal%TYPE;
  vnet   employee.sal%TYPE;
  vempno employee.empno%TYPE;
  vsal   employee.sal%TYPE;
  vcomm employee.comm%TYPE;
BEGIN
    SELECT empno ,sal  ,comm
    INTO  vempno ,vsal ,vcomm
                 FROM employee
                   WHERE empno=p_empno;
    vnet:=vsal+vcomm;
    IF vsal <=20000 THEN
          vincr :=0.20*vnet;
    ELS IF vsal >20000 AND vsal <=30000 THEN
       vincr :=0.30*vnet;
    ELSE
       vincr :=0.40*vnet;
   END IF;
  END IF;
   RETURN (vincr+vsal);
END func_revised_salary ;
  /
```

- **How to call/invoke the function:** There are 2 ways to call functions:
  a) A function can be called from any SQL expression as follows:
  Example:

  select empno, sal, func_revised_salary(empno), comm from employee;

b) The function can be called from any PL/SQL block.

```
SET SERVEROUTPUT ON SIZE 1000000;
   DECLARE
     v_incr_sal NUMBER(7,2);
   BEGIN
     v_incr_sal := proc_revised_salary(2);
     DBMS_OUTPUT.PUT_LINE('incremented   salary  is:'||v_incr_sal);
   END;
   /
```

c) We can also use function call in WHERE clause as follows:

Task 3: create one more function as below:

Now insert into employee table some null values for 'comm' column.

```
insert into employee values (4,400,NULL);
insert into employee values (5,500,NULL);

CREATE FUNCTION func_retrn_comm (p_empno  NUMBER)
RETURN NUMBER
IS
  vincr  employee.sal%TYPE;
  vnet   employee.sal%TYPE;
  vempno employee.empno%TYPE;
  vsal   employee.sal%TYPE;
  vcomm employee.comm%TYPE;
BEGIN
    SELECT empno , sal ,NVL(comm,0)
         INTO  vempno ,vsal ,vcomm
       FROM employee
                WHERE empno=p_empno;
       RETURN (vcomm);
END  func_retrn_comm;
  /
```

Note: The **NVL** function is used to replace NULL values by another value. If 'comm' column is null then NVl set 0 on that fields. Using NVL we can replace the null value by 0,if we will not do this then any operation by null ,result will be null. For example if we'll not convert this and in case ' comm' is null then always any arithmetic operation(such as addition) with 'comm' as one of the operand will be null.

After creation of this function perform below query:

```
select e.empno, e.sal, func_revised_salary(e.empno), e.comm
       from employee e
         where e.comm=func_retrn_comm(5);
```

Note: A user-defined function must:

- be a stored function
- be a single row function and not a group function
- Should accept only IN parameters and their data types must be valid SQL datatypes viz. CHAR, DATE, or NUMBER and not PL/SQL types such as BOOLEAN, RECORD, or TABLE.

Removing Procedure: In order to drop a procedure, one must either own the procedure or have a DROP ANY PROCEDURE system privilege.

Syntax:

```
drop procedure <procedure_name>;
```

Example:

```
drop procedure proc_incr_rating;
```

Removing Functions: To drop a stored function

Syntax:

```
DROP FUNCTION function_nameC
```

Example:

```
DROP  FUNCTION  proc_revised_salary;
```

- **CURSORS:** It 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. This temporary work area is used to store the data retrieved from the database, and manipulate this data. A cursor can hold more than one row, but can process only one row at a time. The set of rows the cursor holds is called the *active* set.

  We use explicit cursor to fetch rows returned by a query. We retrieve the rows into the cursor using a query and then fetch the each row one at a time from the cursor. We typically use the following five steps when using a explicit cursor:

  i) Declare variables to store the column values for a row.
  ii) Declare the cursor, which contains a query.
  iii) Open the cursor.
  iv) Fetch the rows from the cursor one at a time and store the column values in the variables declared in Step i. We would then do something with those variables; such as display them on the screen, use them in a calculation, and so on.
  v) Close the cursor.

Step i): Declare the Variables to Store the Column Values
The first step is to declare the variables that will be used to store the column values. These variables must be compatible with the column types.

Example: The following example declares four variables to store the sid, sname, rating and age columns from the sailors table:

```
DECLARE
  v_ sailor_id  sailors.sid%TYPE;
  v_name      sailors.sname%TYPE;
  v_rating     sailors.rating%TYPE;
  v_age        sailors.age%TYPE;
```

Step ii): Declare the Cursor

This step is to declare the cursor. A cursor declaration consists of a name that you assign to the cursor and the query we want to execute. The cursor declaration, like all other declarations, is placed in the declaration section.

The **syntax** for declaring a cursor is as follows:

```
CURSOR cursor_name IS
                    SELECT_SQL_statement;
```

where

a) cursor_name is the name of the cursor.
b) SELECT_SQL_statement is the SQL query.

Example: The following example declares a cursor named **v_sailor_cursor** whose query retrieves sid, sname, rating and age columns from sailors table:

```
CURSOR v_sailor_cursor IS
  SELECT sid, sname,rating,age
  FROM sailors  ORDER BY sid;
```

NOTE: The query will not executed until we open the cursor.

Step iii): Open the Cursor

We open a cursor using the OPEN statement, which must be placed in the executable section of the PL/SQL block.

Syntax : OPEN cursor_name;

Example: OPEN v_sailor_cursor;

Step iv): Fetch the rows i.e. records from the Cursor

Next is to fetch i.e. retrieve the rows from the cursor of active set to the variable one at a time, which is done using the FETCH statement. The FETCH statement reads the column values into the variables declared in Step i).

We can use any loop structure to fetch the records from the cursor in to variables one row at a time.

Syntax: FETCH cursor_name INTO variable[, variable ...];

Here

cursor_name is the name of the cursor.
variable is the variable into which a column value from the cursor is stored. You need to provide matching variables for each column value.

Example: We can retrieves a row from v_sailor_cursor and stores the column values in the v_sailor_id, v_name, v_rating and v_age variables created earlier in Step i) using FETCH as :

```
FETCH  v_sailor_cursor
               INTO v_sailor_id, v_name, v_rating,v_age;
```

Since a cursor may contain many rows, you need a loop to read them. To figure out when to end the loop, you can use the Boolean variable v_sailor_cursor%NOTFOUND. This variable is true when there are no more rows to read in v_sailor_cursor.
The following example shows a loop:

```
 LOOP
   FETCH  v_sailor_cursor
          INTO v_sailor_id, v_name, v_rating,v_age;
   EXIT  WHEN  v_sailor_cursor%NOTFOUND;
DBMS_OUTPUT.PUT_LINE( 'v_sailor_id = ' || v_sailor_id || ', v_name = ' || v_name ||
                      ', v_rating = ' || v_rating||', v_age = ' || v_age);
  END LOOP;
```

Notice that we have used DBMS_OUTPUT.PUT_LINE() to display the v_sailor_id, v_name, v_rating and v_age variables that were read for each row using cursor from active data set.

Step v): Close the Cursor

Close the cursor using the CLOSE statement. Close statement disables the cursor and active set becomes undefined. Closing a cursor frees up system resources (such as memory) and its data sent occupied by it.

Syntax: CLOSE cursor_name ;

Example: CLOSE v_sailor_cursor;

Task 4: To run the PL/SQL having cursor the complete picture will be like this:

```
 SET SERVEROUTPUT ON SIZE 1000000;
DECLARE
  v_sailor_id sailors.sid%TYPE;
v_name      sailors.sname%TYPE;
v_rating    sailors.rating%TYPE;
v_age     sailors.age%TYPE;
CURSOR v_sailor_cursor IS
          SELECT sid,sname,rating,age
                  FROM sailors
                         ORDER BY sid;
 BEGIN
```

```
            OPEN v_sailor_cursor;
  LOOP
            FETCH  v_sailor_cursor
                       INTO v_sailor_id, v_name, v_rating,v_age;

       EXIT WHEN v_sailor_cursor%NOTFOUND;
   DBMS_OUTPUT.PUT_LINE( 'v_sailor_id = ' || v_sailor_id || ', v_name = ' || v_name ||
                                    ', v_rating = ' || v_rating||', v_age = ' || v_age);
  END LOOP;
 CLOSE v_sailor_cursor;
END;
/
```

<u>Explicit Cursor Attributes:</u> Oracle provided attributes to control the data processing while using cursors. Whenever any cursor is opened and used Oracle creates a set of four system variable via which oracle keeps track of the current status of the cursor which we can access. Use these attributes to avoid errors while accessing cursors through OPEN, FETCH and CLOSE Statements.

Error occurs: **a)** when tried to open a cursor which is not closed in the previous operation.
b) When to try to fetch a cursor after the last operation.

Attributes available to check the status of an explicit cursor.

Attributes	Return values	Example
%FOUND	TRUE, if fetch statement returns at least one row.	Cursor_name%FOUND
	FALSE, if fetch statement doesn't return a row.	
%NOTFOUND	TRUE, , if fetch statement doesn't return a row.	Cursor_name%NOTFOUND
	FALSE, if fetch statement returns at least one row.	
%ROWCOUNT	The number of rows fetched by the fetch statement	Cursor_name%ROWCOUNT
	If no row is returned, the PL/SQL statement returns an error.	
%ISOPEN	TRUE, if the cursor is already open in the program	Cursor_name%ISNAME
	FALSE, if the cursor is not opened in the program.	

We can use in above performed task cursor attributes appropriately. [Refer Oracle Developer 2000 to see how implicit cursor use attributes]

Task 5: Let's use in the above example certain cursor variables/attributes to control the execution of the cursor:

First Create a table *TempInfo (name varchar2(20) ,rec_date date ,age number(3,1)*) ;

Now execute the below cursor:

```
    SET SERVEROUTPUT ON SIZE 1000000;
    DECLARE
      v_sailor_id sailors.sid%TYPE;
    v_name       sailors.sname%TYPE;
    v_rating     sailors.rating%TYPE;
    v_age      sailors.age%TYPE;
    CURSOR v_sailor_cursor IS
               SELECT sid, sname,rating,age  FROM sailors  ORDER BY sid DESC;
  BEGIN
```

```
                    OPEN v_sailor_cursor;
          IF v_sailor_cursor%ISOPEN THEN
            LOOP
                FETCH v_sailor_cursor     INTO v_sailor_id, v_name, v_rating,v_age;
                    EXIT WHEN v_sailor_cursor%NOTFOUND;
              DBMS_OUTPUT.PUT_LINE( 'v_sailor_id = ' || v_sailor_id || ', v_name = ' || v_name ||
                                          ', v_rating = ' || v_rating||', v_age = ' || v_age);
              INSERT INTO TempInfo VALUES (v_name,sysdate,v_age);
           END LOOP;
           COMMIT;
           CLOSE v_sailor_cursor;
        ELSE
              DBMS_OUTPUT.PUT_LINE( 'Unable to open the cursor');
          ENDIF;
     END;
      /
```

Task 6: showing use of %ROWCOUNT and %NOTFOUND attributes for exit conditions in a loop.

```
SET SERVEROUTPUT ON SIZE 1000000;
DECLARE
         v_sailor_id sailors.sid%TYPE;
         v_rating      sailors.rating%TYPE;
        CURSOR v_sailor_cursor IS      SELECT sid, sname,rating,age   FROM sailors;
 BEGIN
                 OPEN v_sailor_cursor;
        DBMS_OUTPUT.PUT_LINE( ' SID         Rating ');
        DBMS_OUTPUT.PUT_LINE( ' ------         -------- );
    LOOP
            FETCH v_sailor_cursor  INTO v_sailor_id, v_rating;
          EXIT WHEN v_sailor_cursor %ROWCOUNT >4 OR v_sailor_cursor%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE( v_sailor_id ||'          ' || v_rating);
    END LOOP;
   CLOSE v_sailor_cursor;
END;
/
```

➢ **Triggers:** A *trigger* is a procedure that is run (or *fired)* automatically by the database when a specified DML statement (INSERT, UPDATE, or DELETE) is run against a certain database table.

When a trigger fires: A trigger may fire before or after a DML statement runs. Also, because a DML statement can affect more than one row, the code for the trigger may be run once for every row affected (a row-level trigger), or just once for all the rows (a statement-level trigger). For example, if you create a row level trigger that fires for an UPDATE on a table, and you run an UPDATE statement that modified ten rows of that table, then that trigger would run ten times. If, however, your trigger was a statement-level trigger, the trigger would fire once for the whole UPDATE statement, regardless of the number of rows affected.

There is another difference between a row-level trigger and a statement-level trigger: A row level trigger has access to the old and new column values when the trigger fires as a result of an UPDATE statement on that column. The firing of a row-level trigger may also be limited using a trigger *condition*; for example, you could set a condition that limits the trigger to fire only when a column value is less than a specified value.

Example : SetupP for the trigger:

Triggers are useful for doing advanced auditing of changes made to column values. In this example we will have a trigger that records when a product's price is lowered by more than 25 percent; when this occurs, the trigger will add a row to the product_price_audit table. For this create two tables:

```
  create table products (
         product_id INTEGER PRIMARY KEY,
        product_price   number(5,2)
);
 create table product_price_audit (
        product_id INTEGER CONSTRAINT price_audit_fk_products REFERENCES
        products(product_id),
        old_price NUMBER(5,2),
        new_price   NUMBER(5,2)
        );
```

As you can see, the product_id column of the product_price_audit table is a foreign key to the product_id column of the products table. The old_price column will be used to store the old price of a product prior to the change, and the new_price column will be used to store the new price after the change.

<u>Creating atrigger:</u> The simplified syntax for the CREATE TRIGGER statement is as follows:

```
CREATE [OR REPLACE] TRIGGER trigger_name
  {BEFORE | AFTER | INSTEAD OF } trigger_event
ON table_name
  [FOR EACH ROW]
  [{FORWARD | REVERSE} CROSSEDITION]
  [{FOLLOWS | PRECEDES} schema.other_trigger}
  [{ENABLE | DISABLE}]
 [WHEN   trigger_condition]]
BEGIN
     trigger_body
END  trigger_name;
```

where

OR REPLACE means the trigger is to replace an existing trigger, if present.
trigger_name is the name of the trigger.
BEFORE means the trigger fires before the triggering event is performed. **AFTER** means the trigger fires after the triggering event is performed. **INSTEAD OF** means the trigger fires instead of performing the triggering event.
trigger_event is the event that causes the trigger to fire.
table_name is the table that the trigger references.
FOR EACH ROW means the trigger is a row-level trigger, that is, the code contained within *trigger_body* is run for each row when the trigger fires. If you omit FOR EACH ROW, the trigger is a

statement-level trigger, which means the code within *trigger_body* is run once when the trigger fires.

{FORWARD | REVERSE} CROSSEDITION is new for Oracle Database 11*g* and will typically be used by database administrators or application administrators. **A FORWARD** cross edition trigger is intended to fire when a DML statement makes a change in the database while an online application currently accessing the database *is being patchedor upgraded* (FORWARD is the default); the code in the trigger body must be designedto handle the DML changes when the application patching or upgrade is complete. **A REVERSE** cross edition trigger is similar, except it is intended to fire and handle DML
changes made *after the online application has been patched or upgraded*.

{FOLLOWS | PRECEDES} *schema.other_trigger* is new for Oracle Database 11*g* and specifies whether the firing of the trigger follows or precedes the firing of another trigger specified in *schema.other_trigger*. You can create a series of triggers that fire in a specific order.

{ENABLE | DISABLE} is new for Oracle Database 11*g* and indicates whether the trigger is initially enabled or disabled when it is created (the default is ENABLE). You enable a disabled trigger by using the ALTER TRIGGER *trigger_name* ENABLE statement or by enabling all triggers for a table using ALTER TABLE *table_name* ENABLE ALL TRIGGERS.

trigger_condition is a Boolean condition that limits when a trigger actually runs its code.

trigger_body contains the code for the trigger.

The example trigger we'll see in this section fires before an update of the price column in the products table; therefore, we'll name the trigger before_product_price_update. Also, because we want to use the price column values before and after an UPDATE statement modifies the price column's value, we must use a row-level trigger. Finally, we want to audit a price change when the new price is lowered by more than 25 percent of the old price; therefore, we'll need to specify a trigger condition to compare the new price with the old price.

The following statement creates the **before_product_price_update** trigger:

Task 7:

```
CREATE  TRIGGER   before_product_price_update
      BEFORE UPDATE OF product_price
      ON products
      FOR EACH ROW WHEN (new.product_price < old.product_price * 0.75)
 BEGIN
      dbms_output.put_line('product_id = ' || :old.product_id);
      dbms_output.put_line('Old price = ' || :old.product_price);
      dbms_output.put_line('New price = ' || :new.product_price);
      dbms_output.put_line('The price reduction is more than 25%');
  INSERT INTO product_price_audit (product_id, old_price, new_price)
           VALUES ( :old.product_id, :old.product_price, :new.product_price);
 END   before_product_price_update;
 /
```

There are five things you should notice about this statement:

- **BEFORE UPDATE OF price** means the trigger fires before an update of the price column.
- **FOR EACH ROW** means this as a row-level trigger, that is, the trigger code contained within the BEGIN and END keywords runs once for each row modified by the update.
- The trigger condition is (new.price < old.price * 0.75), which means the trigger fires only when the new price is less than 75 percent of the old price (that is,when the price is reduced by more than 25 percent).
- The new and old column values are accessed using the :old and :new aliases in the trigger.

- The trigger code displays the product_id, the old and new prices, and a message stating that the price reduction is more than 25 percent. The code then adds a row to product_price_audit table containing the product_id and the old and new prices.

Firing a trigger:

To see the output from the trigger, you need to run the SET SERVEROUTPUT ON command:

SET SERVEROUTPUT ON

To fire the before_product_price_update trigger, you must reduce a product's price by more than 25 percent. Go ahead and perform the following INSERT and UPDATE statements to reduce the price:

// Lets first observe the data of products table.

```
insert into products values (515.99);
insert into products values(1049.99);

UPDATE  products_temp
                SET product_price = product_price * .7
                        WHERE product_id IN (5, 10);
```

// Now again observe the data of products table.

Dropping a trigger: We can drop a trigger using DROP TRIGGER. The following example drops the before_product_ price_update trigger:

DROP TRIGGER before_product_price_update;

General Syntax for using FOR LOOP with cursor:

Use the cursor FOR loop when you want to fetch and process every record in a cursor.

```
FOR record_name IN cusror_name
LOOP
    process the row...
END LOOP;
```

When using FOR LOOP you need not declare a record or variables to store the cursor values, need not open, fetch and close the cursor. These functions are accomplished by the FOR LOOP automatically.

Task 8: Create a table ***EMP_TBL(Ename,empid,salary)*** populate with few rows data and execute below code.

```
SET SERVEROUTPUT ON SIZE 1000000;
DECLARE
  CURSOR emp_cur IS
           SELECT Ename,empid,salary FROM emp_tbl;
emp_rec emp_cur%rowtype;
 BEGIN
  FOR emp_rec in emp_cur
  LOOP
dbms_output.put_line(emp_cur.first_name||''||emp_cur.last_name||emp_cur.sal
ary);
 END LOOP;
END;
/
```

In example, when the FOR loop is processed a record 'emp_rec'of structure 'emp_cur' gets created, the cursor is opened, the rows are fetched to the record 'emp_rec' and the cursor is closed after the last row is processed.Using FOR Loop in program, we can reduce the number of lines in the program.

Task 9: Following to display the name and rating of each sailor whose age is greater than 30 years.

```
SET SERVEROUTPUT ON SIZE 1000000;
DECLARE
   CURSOR sail_cur
     IS
       SELECT sname, rating
              FROM sailors WHERE age>30
                     ORDER BY rating DESC;
BEGIN
   FOR sail_rec IN sail_cur
   LOOP
      DBMS_OUTPUT.PUT_LINE
          ('Sailor: ' || sail_rec.sname || ' have ratings ' ||
                        sail_rec.rating||'and age'||' sail_rec.age');
   END LOOP;
END;
 /
```

Exercise 1:

a) Consider a StoreFact table consists of number and factorial columns. Write a PL/SQL code block to calculate the factorial of all numbers which are available in StoreFact table. If computed factorial of the number is greater than 500 then store the computed factorial of the number correspondingly in StoreFact table against the respective number else display the number and its calculated factorial.

Hint: You first create StoreFact(NO,fact) table and insert few data for NO column.

```
SET SERVEROUTPUT ON SIZE 1000000;
DECLARE
 CURSOR f_cur IS select NO from StoreFact;
 n_counter  StoreFact.NO%TYPE;
 n_factorial NUMBER := 1;
 n_temp      NUMBER;
BEGIN
OPEN  f_cur;
LOOP
FETCH f_cur INTO n_counter;
 EXIT WHEN f_cur%NOTFOUND;
 DBMS_OUTPUT.PUT_LINE('No Fetched :'||n_counter);
 n_temp :=n_counter;
 n_factorial:=1;
 WHILE n_counter >0
 LOOP
   n_factorial := n_factorial * n_counter;
    n_counter  := n_counter - 1;
   END LOOP;
if n_factorial<500THEN
 DBMS_OUTPUT.PUT_LINE('factorial of ' || n_temp ||  ' is ' || n_factorial);
      ELSE
           UPDATE StoreFact SET NO=n_factorial WHERE StoreFact.NO=n_temp;
            DBMS_OUTPUT.PUT_LINE('Factorial inserted into table StoreFact');

   END IF;
END LOOP;
close f_cur;
END;
/
```

b) In the above code use appropriately other cursor attributes and now re-write the above program using **function**.

Exercise 2: Create a trigger that allows only insertion into SAILORS table to be performed when rating of sailors >5 but not allows any other DML operation. If violation happens write appropriate message.

Exercise 3: a) Write a procedure *findName(tsid IN number(3) ,name OUT varchar2(12))* to find name of sailors with specified sid.
Now call the procedure in below PL/SQL block as follows:

```
SET SERVEROUTPUT ON SIZE 1000000;
DECLARE
tm_sid numer(3);
tname  varchar2(12);
BEGIN
tm_sid:=22;
findName(tm_sid,tname );
dbms_output.put_line(tname);
END;
/
```

b) Re-write above procedure using function to retrieve sailors name with specified sid.

Exercise 4: Write procedure using cursor and for loop to display complete details of a specific sailor name.