PL/SQL

Friday, May 24, 2024 4:22 PM

Programming language used for managing data in databases.

Basic of PL/SQL -

PL/SQL engine compiles PL/SQL code into byte-code and executes the executable code.
Once we submit PL/SQL code to the Oracle Database server, the PL/SQL engine collaborates with the SQL engine to compile and execute the code.
PL/SQL engine runs the procedural elements while the SQL engine processes the SQL statements.

PL/SQL Anonymous Block (1 time use)

- It is a block structured language
- Useful for testing purposes
- Block consists of
    - **Declaration**
        - Declare variables, allocate memory for cursors, and define data types
    - **Executable (Mandatory)**
        - Starts with BEGIN and ends with keyword END
        - Must have 1 statement to execute even if it is NULL
    - **Exception-handling**
        - Starts with the keyword EXCEPTION
        - Here we can catch and handle exceptions raised in execution section.

BEGIN
  DBMS_OUTPUT.put_line ('Hello World!');
END;
 	
begin
null;
end;
	SET SERVEROUTPUT ON command in
	message using the DBMS_OUTPUT.PUT_LINE procedure

DECLARE
  l_message VARCHAR2(255) := 'Hello World!';
BEGIN
  DBMS_OUTPUT.PUT_LINE(l_message);
END;
	Hello World!

DECLARE
    v_result NUMBER;
BEGIN
  v_result := 1 / 0;
  EXCEPTION
    WHEN ZERO_DIVIDE THEN
      DBMS_OUTPUT.PUT_LINE(SQLERRM);
END;
	ORA-01476: divisor is equal to zero

PL/SQL Named Block (Reusable)--> stored in oracle database server - ex- Functions and Procedures

DATA TYPES -

1) Scalar
   Store single values -- number,boolean,character,datetime
2) Composite
   Store multiple values -- record and collection

Number,binary_float,binary_double - sql data types.

- **PLS_INTEGER** -- PL/SQL --> requires less storage than number range from -2,147,483,648 to 2,147,483,647.
- **Boolean** --- TRUE,FALSE AND NULL -- we can use in IF,THEN,CASE,LOOPS
- **Character** -- char(n) - fixed length - range from 1 to 32,767 bytes.
- **Varchar(n)** - varying length character -range from 1 to 32,767 bytes.
  long, raw, long raw, rowid, and UROWID.
- **Date time** -- Date, Timestamp

Variables

- Variable is named storage location that stores a value of a particular data type.
- Before using a variable, you must declare it in the declaration section of a block.
- To assign a value to a variable, you use the assignment operator (:=)

variable_name datatype [NOT NULL] [:= initial_value];	

DECLARE
  l_product_name VARCHAR2(100) DEFAULT 'Laptop';
BEGIN
  NULL;
END;
	DECLARE
	  l_product_name VARCHAR2(100) := 'Laptop';
	BEGIN
	  NULL;
	END;

DECLARE
  l_shipping_status VARCHAR2(25) NOT NULL := 'Shipped';
BEGIN
  l_shipping_status := '';
END;
	ORA-06502: PL/SQL: numeric or value error

Anchored Declaration -
If the data type of the table column changes,
we must adjust the program to make it work with the new type
DECLARE
  l_customer_name customers.name%TYPE;
  l_credit_limit customers.credit_limit%TYPE;
BEGIN
  SELECT
    name, credit_limit
  INTO
    l_customer_name, l_credit_limit
  FROM
    customers
  WHERE	

```
  customer_id = 38;

 DBMS_OUTPUT.PUT_LINE(l_customer_name || ':' || l_credit_limit );
END;
/
```

PL/SQL Comments - single line -> "--" , Multi-line -> "/*………………………………*/"

Constants-

Unlike a variable, a constant holds a value that does not change throughout the execution of the program.

PL/SQL IF Statement

IF statement to either execute or skip a sequence of statements based on a specified condition.
IF statement has three forms:

1) IF THEN

```
IF condition THEN
   statements;
END IF;
```

2) IF THEN ELSE

```
declare
a number(10) := 19;
begin
   if a>2 then
   DBMS_OUTPUT.PUT_LINE( a );
        else
      DBMS_OUTPUT.PUT_LINE( 'not found' );
        end if;
end;
```

3) IF THEN ELSIF

```
declare
a number(10) := 19;
begin
   if a<2 then
   DBMS_OUTPUT.PUT_LINE( a );
        elsif a>10 and a<20 then
        DBMS_OUTPUT.PUT_LINE( a || ' yes' );
        else
     DBMS_OUTPUT.PUT_LINE( 'not found' );
        end if;
end;
```

```
IF condition_1 THEN
  statements_1
ELSIF condition_2 THEN
  statements_2
[ ELSIF condition_3 THEN
    statements_3
]
...
[ ELSE
    else_statements
]
END IF;
```

4) Nested IF statement

```
declare
a number(10) := 19;
begin
   if a>2 then
      DBMS_OUTPUT.PUT_LINE( a );
      if a>10 and a<20 then
           DBMS_OUTPUT.PUT_LINE( a || ' yes' );
      end if;
        else
      DBMS_OUTPUT.PUT_LINE( 'not found' );
        end if;
end;
```

```
IF condition_1 THEN
   IF condition_2 THEN
      nested_if_statements;
   END IF;
ELSE
   else_statements;
END IF;
```

PL/SQL CASE statement

The searched CASE statement follows the rules below:

- The conditions in the WHEN clauses are evaluated in order, from top to bottom.
- The sequence of statements associated with the WHEN clause whose condition evaluates to TRUE is executed.
  If more than one condition evaluates to TRUE, only the first one executes.
- If no condition evaluates to TRUE, the else_statements in the ELSE clause executes.
  If you skip the ELSE clause and no expressions are TRUE, a CASE_NOT_FOUND exception is raised.

From <https://www.oracletutorial.com/plsql-tutorial/plsql-case-statement/>

```
CASE selector
WHEN selector_value_1 THEN
   statements_1
WHEN selector_value_1 THEN
   statement_2
...
ELSE
   else_statements
END CASE;
```

```
declare
a varchar(20) := 'Z';
b varchar(20);
begin
 CASE a
  when 'A' then
     b := 'Excellent';
  when 'B' then
     b := 'Fair';
  when 'C' then
     b := 'Good';
       else
```

```
declare
a varchar(20) := 'Z';
b varchar(20);
begin
 CASE a
  when 'A' then
     b := 'Excellent';
  when 'B' then
     b := 'Fair';
  when 'C' then
     b := 'Good';
      end case;
```

```
    b := 'Poor';
  end case;
dbms_output.put_line(b);
end;
```

```
dbms_output.put_line(b);
end;

o/p =
 ORA-06592: CASE not found while executing CASE statement ORA-06512: at line 13 ORA-06512: at "SYS.DBMS_SQL", line 1721
```

PL/SQL GOTO Statement

GOTO label_name;

```
BEGIN
 GOTO second_message;

 <<first_message>>
 DBMS_OUTPUT.PUT_LINE( 'Hello' );
 GOTO the_end;

 <<second_message>>
 DBMS_OUTPUT.PUT_LINE( 'PL/SQL GOTO Demo' );
 GOTO first_message;

 <<the_end>>
 DBMS_OUTPUT.PUT_LINE( 'and good bye...' );

END;
```

GOTO statement restrictions

1) First, you cannot use a GOTO statement to transfer control into an IF, CASE or LOOP statement, the same for the sub-block
2) Second, you cannot use a GOTO statement to transfer control from one clause to another in the IF statement e.g., from IF clause to ELSIF or ELSE clause, or from one WHEN clause to another in the CASE statement.
3) Third, you cannot use a GOTO statement to transfer control out of a subprogram or into an exception handler.
4) Fourth, you cannot use a GOTO statement to transfer control from an exception handler back into the current block.

PL/SQL NULL Statement

The NULL statement does nothing except that it passes control to the next statement.

PL/SQL LOOP

LOOP statement is a control structure that repeatedly executes a block of code until a specific condition is met or until you manually exit the loop.

LOOP

```
declare
a number := 0;
begin
   <<outer_loop>>
        loop
                a := a+1;
                dbms_output.put_line(a);
                if a>10 then
         exit;
        end if;
        End loop outer_loop;
end;
```

WHILE LOOP - when no of execution is unknown

```
declare
a number := 1;
begin
           while  a <= 4 loop
      dbms_output.put_line (a);
        a := a+ 1;
        exit when a>3;
                end loop;
 dbms_output.put_line (a);
end;
```

```
WHILE condition
LOOP
   statements;
END LOOP;
```

FOR LOOP - when no of executions is known

```
FOR index IN lower_bound .. upper_bound
LOOP
   statements;
END LOOP;
```

```
BEGIN
 FOR a IN 1..5
 LOOP
   DBMS_OUTPUT.PUT_LINE( a );
 END LOOP;
END;
```

FOR LOOP with REVERSE keyword

```
FOR index IN REVERSE lower_bound .. upper_bound
   LOOP
   statements;
END LOOP;
```

PL/SQL CONTINUE statement

The CONTINUE statement allows you to exit the current loop iteration and immediately continue on to the next iteration of that loop.

We can use in every loop like FOR,WHILE,LOOP

```
IF condition THEN
   CONTINUE;
END IF;
```

CONTINUE WHEN condition;

```
BEGIN
 FOR n_index IN 1 .. 10
 LOOP
```

```
BEGIN
 FOR n_index IN 1 .. 10
 LOOP
  -- skip odd numbers
  IF MOD( n_index, 2 ) = 1 THEN
```

	-- skip even numbers
	CONTINUE
	WHEN MOD(n_index, 2) = 0;
	DBMS_OUTPUT.PUT_LINE(n_index);
	END LOOP;
	END;
		CONTINUE;
		END IF;
		DBMS_OUTPUT.PUT_LINE(n_index);
		END LOOP;
		END;

PL/SQL SELECT INTO

To fetch data of a single row from a table into variables.
If the SELECT statement returns more than one row, Oracle will raise the TOO_MANY_ROWS exception. If the SELECT statement does not return any row, Oracle will raise the NO_DATA_FOUND exception.

SELECT
select_list
INTO
variable_list
FROM
table_name
WHERE
condition;
	declare
	name customer.customer_name%type;
	begin
	SELECT customer_name INTO name from customer
	WHERE customer_id = 1;
	dbms_output.put_line(name);
	end;
		declare
		name1 customer%rowtype;
		begin
		SELECT * INTO name1 from customer
		WHERE customer_id = 4;
		dbms_output.put_line(name1.customer_name || name1.customer_id);
		end;

Oracle Trigger :- **Named PL/SQL blocks which are stored in the database**

Events -	Types -	Uses -
DML, DDL, System, User event	◆ DML Triggers ◆ DDL Triggers (Auditing changes) ◆ System/ Database Event Triggers (e.g. log off/log on) ◆ Instead-of Triggers ◆ Compound Triggers	◆ Enforce business rules ◆ Gain strong control over the security ◆ Collect statistical Information ◆ Automatically generate values ◆ Prevent invalid Transactions

How to create a trigger in Oracle -

--Header of trigger CREATE [OR REPLACE] TRIGGER trigger_name {BEFORE | AFTER } triggering_event ON table_name [FOR EACH ROW] [FOLLOWS | PRECEDES another_trigger] [ENABLE / DISABLE] [WHEN condition] -- Body of Trigger DECLARE declaration statements BEGIN executable statements EXCEPTION exception_handling statements END;	create table audits(audit_id number, table_name varchar(20), transaction_name varchar(20), table_date Date); -- Header Create or replace trigger customer_audit_trg AFTER UPDATE OR DELETE on customer FOR EACH ROW -- Body Declare Customer_name varchar(20); Begin Customer_name := CASE WHEN Updating THEN 'UPDATE' WHEN Deleting THEN 'DELETE' End; /* UPDATE customer SET Customer_name = 'Shivu' WHERE customer_id =2; END;*/ DELETE FROM customer WHERE customer_name = 'Arun'; End; Select * from audits;

Oracle Statement-level Triggers

A statement-level trigger is fired whenever a trigger event occurs on a table regardless of how many rows are affected. In other words, a statement-level trigger executes once for each transaction.
For example, if you update 1000 rows in a table, then a statement-level trigger on that table would only be executed once.

Oracle Row-level Triggers (DATA AUDITING / DATA VALIDATION)

Row-level triggers fires once for each row affected by the triggering event such as INSERT, UPDATE, or DELETE.

DML Trigger -->

create or replace trigger bi_emp_trg
BEFORE INSERT ON employee
FOR EACH ROW
ENABLE
DECLARE
user_emp varchar(20);
BEGIN
select user into user_emp from dual;
dbms_output.put_line('You just inserted Mr. '||
user_emp);
END;
	create or replace trigger bu_emp_trg
	BEFORE UPDATE ON employee
	FOR EACH ROW
	ENABLE
	DECLARE
	user_emp varchar(20);
	BEGIN
	select user into user_emp from dual;
	dbms_output.put_line('You just updated Mr. '|| user_emp);
	END;
		-- For auditing purposes -
		
		create table emp(
		emp_name varchar(30)
		);
		
		create table sh_audit (
		new_name varchar(30),
		old_name varchar(30),
		user_name varchar(30),
		entry_date varchar(30),	

```
Insert into employee values(
   'Ironman'
   )

o/p  =
1 row(s) inserted.
You just inserted Mr. APEX_PUBLIC_USER
```

```
Update employee set emp_name = 'Adultman' where emp_name =
'Ironman';



o/p  =
1 row(s) updated.
You just updated Mr. APEX_PUBLIC_USER
```

```
   operation varchar(30)
);

create or replace trigger bi_emp
        Before
                 INSERT OR UPDATE OR DELETE
        on
                Emp
                FOR EACH ROW
                ENABLE
        DECLARE
           v_user varchar(30);
           v_date varchar(30);
        BEGIN
                SELECT user, To_char(sysdate,'DD/MON/YYYY HH24:MI:SS') into v_user,v_date from
                dual;

               IF INSERTING THEN
                    Insert into sh_audit( new_name,
                                                old_name,
                                                user_name,
                                                entry_date,
                                                operation)
                                  values (:NEW.emp_name,
                                                  NULL,
                                                  v_user,
                                                  v_date,
                                                  'Insert');
                END IF;
        end;


select * from sh_audit;

insert into emp values ('Shailvi');
```

```
create table emp1(
   emp_name varchar(30)
);

desc emp;

create table emp_backup1 as select * from emp1 where 1=2;

create or replace trigger emp_backup1
before INSERT OR UPDATE OR DELETE on emp1
FOR EACH ROW
ENABLE
BEGIN
        IF INSERTING THEN
                insert into emp_backup1(emp_name) values (:NEW.emp_name);
        END IF;
END;

select * from emp1;
select * from emp_backup1;

Insert into emp1 values('TIlak');
Insert into emp1 values('Shailvi');
```

Oracle INSTEAD OF Triggers

An INSTEAD OF trigger is a trigger that allows you to update data in tables via their view which cannot be modified directly through DML statements.

PL/SQL Cursor

Cursor is a pointer to memory area called context area. Context area is a memory region inside the process global area or PGA assigned to hold the info about the processing of a SELECT statement or DML statement.
It is a pointer that points a result set of query.

Cursor is defined as private worked area where the SQL statements (Select & DML) are executed.

Types -

- ► Implicit

  Whenever Oracle executes an SQL statement such as SELECT INTO, INSERT, UPDATE, and DELETE, it automatically creates an implicit cursor

- ► Explicit

How to create Cursor -

select * from customer;	-- Cursor Parameters / parameterized customers	-- Cursor Parameters / parameterized customers with default value

```
desc customer;

Declare
        c_name1 varchar(20);
-- declare the cursor
        CURSOR cur_customer IS
                select CUSTOMER_NAME from customer
                where CUSTOMER_ID <5;
Begin
        OPEN cur_customer;
        LOOP
                Fetch cur_customer into c_name1;
                        Dbms_ouptut.put_line(c_name1);
                Exit when cur_customer%notfound;
        END LOOP;
        CLOSE cur_customer;
END;
```

```
select * from customer;
declare
        v_name varchar (20);
        CURSOR cur_Bottle (var_c_id number) IS
        SELECT customer_name from customer
                Where
                customer_ID < var_c_id;
BEGIN
        OPEN cur_bottle(5);
        LOOP
                FETCH customer_name into v_name;
                exit when cur_bottle%notfound;
         Dbms_output.put_line(v_name);
        END LOOP;
        close cur_bottle;
END;
/
```

```
select * from customer;
declare
        v_name varchar (20);
        v_cid number(10);
        CURSOR cur_Bottle (var_c_id number:=5) IS
        SELECT customer_name from customer
                Where
                customer_ID < var_c_id;
BEGIN
        OPEN cur_bottle; -- if we give any parameter value o/p will be a/q to this not default value
        LOOP
                FETCH customer_name into v_name,v_cid;
                exit when cur_bottle%notfound;
         Dbms_output.put_line(v_name||' '||v_Cid);
        END LOOP;
        close cur_bottle;
END;
/
```

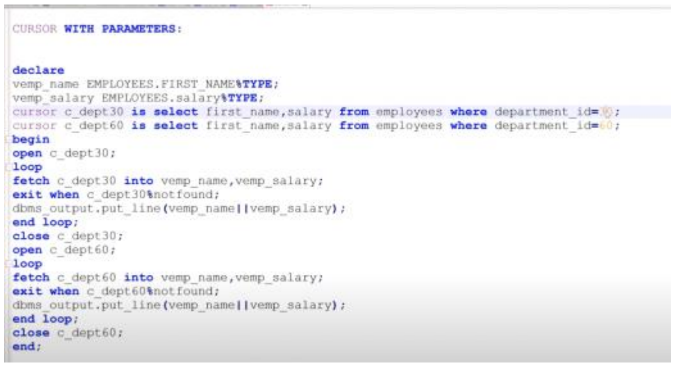
```
-- cursor FOR LOOP
declare
        CURSOR cur_Dev is
        select customer_name,city from customer
          where
   customer_id >3;
BEGIN
   For l_index in cur_Dev
   LOOP
          DBMS_OUTPUT.PUT_LINE (l_index.customer_name||' '||l_index.city);
        END lOOP;
END;
/
select * from customer;
```

```
declare
        CURSOR cur_Reb(var_c_id number) is
        select customer_id,customer_name from customer
        where
        customer_id > var_c_id;
BEGIN
        for L_indx in cur_Reb(2) loop
        Dbms_output.put_line(l_indx.customer_id||' '||l_indx.customer_name);
        end loop;
end;
/
```

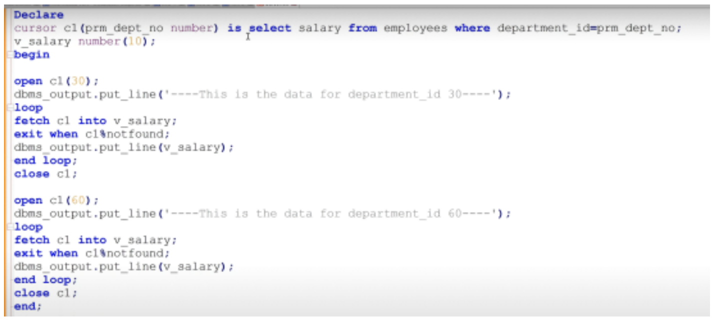
Explicit Cursor Attributes - %isopen,%found,%notfound, %rowcount

To know how many Cursor is opened - select * from v$open_cursor where user_name ='schema_name';

Cursor parameters -



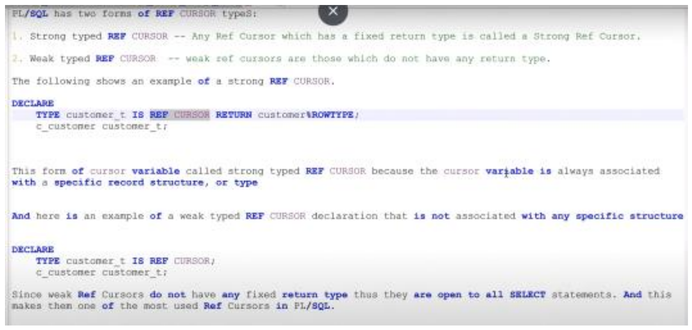
```
CURSOR WITH PARAMETERS:


declare
vemp_name EMPLOYEES.FIRST_NAME%TYPE;
vemp_salary EMPLOYEES.salary%TYPE;
cursor c_dept30 is select first_name,salary from employees where department_id=30;
cursor c_dept60 is select first_name,salary from employees where department_id=60;
begin
open c_dept30;
loop
fetch c_dept30 into vemp_name,vemp_salary;
exit when c_dept30%notfound;
dbms_output.put_line(vemp_name||vemp_salary);
end loop;
close c_dept30;
open c_dept60;
loop
fetch c_dept60 into vemp_name,vemp_salary;
exit when c_dept60%notfound;
dbms_output.put_line(vemp_name||vemp_salary);
end loop;
close c_dept60;
end;
```



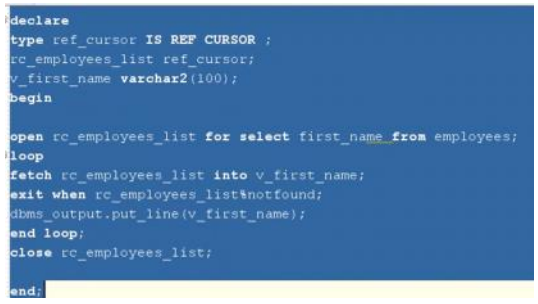
```
Declare
cursor c1(prm_dept_no number) is select salary from employees where department_id=prm_dept_no;
v_salary number(10);
begin

open c1(30);
dbms_output.put_line('----This is the data for department_id 30----');
loop
fetch c1 into v_salary;
exit when c1%notfound;
dbms_output.put_line(v_salary);
end loop;
close c1;

open c1(60);
dbms_output.put_line('----This is the data for department_id 60----');
loop
fetch c1 into v_salary;
exit when c1%notfound;
dbms_output.put_line(v_salary);
end loop;
close c1;
end;
```

Instead of using many times dept_id = 30,60 we directly apply parameters to overcome writing code
From <https://www.oracletutorial.com/plsql-tutorial/plsql-cursor/>

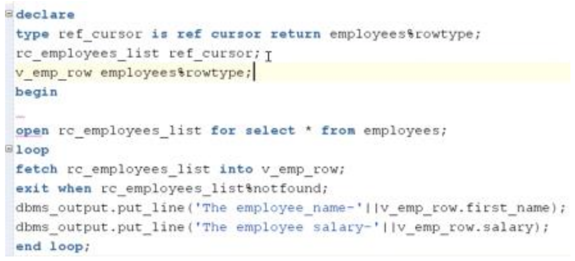
REF Cursor



```
PL/SQL has two forms of REF CURSOR typeS:

1. Strong typed REF CURSOR -- Any Ref Cursor which has a fixed return type is called a Strong Ref Cursor.

2. Weak typed REF CURSOR  -- weak ref cursors are those which do not have any return type.

The following shows an example of a strong REF CURSOR.

DECLARE
    TYPE customer_t IS REF CURSOR RETURN customer%ROWTYPE;
    c_customer customer_t;


This form of cursor variable called strong typed REF CURSOR because the cursor variable is always associated
with a specific record structure, or type


And here is an example of a weak typed REF CURSOR declaration that is not associated with any specific structure


DECLARE
    TYPE customer_t IS REF CURSOR;
    c_customer customer_t;

Since weak Ref Cursors do not have any fixed return type thus they are open to all SELECT statements. And this
makes them one of the most used Ref Cursors in PL/SQL.
```

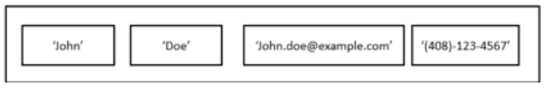


```
declare
type ref_cursor IS REF CURSOR ;
rc_employees_list ref_cursor;
v_first_name varchar2(100);
begin

open rc_employees_list for select first_name from employees;
loop
fetch rc_employees_list into v_first_name;
exit when rc_employees_list%notfound;
dbms_output.put_line(v_first_name);
end loop;
close rc_employees_list;

end;
```



```
declare
type ref_cursor is ref cursor return employees%rowtype;
rc_employees_list ref_cursor;
v_emp_row employees%rowtype;
begin

open rc_employees_list for select * from employees;
loop
fetch rc_employees_list into v_emp_row;
exit when rc_employees_list%notfound;
dbms_output.put_line('The employee name-'||v_emp_row.first_name);
dbms_output.put_line('The employee salary-'||v_emp_row.salary);
end loop;
```

PL/SQL Record - composite data structure that consists of multiple fields; each has its own value.



If we want to change datatype or width of the variable of the column we wont do directly so PL/Sql resolves this issue

Types of records -

%TYPE - Anchored datatype variable
%ROWTYPE - Record datatype variable

Syntax - variable_name table_name/cursor_name%ROWTYPE
How to access - record_name.column_name

1) Table - based

DECLARE record_name table_name%ROWTYPE;	

2) Cursor - based

DECLARE
  CURSOR c_contacts IS
    SELECT first_name, last_name, phone
    FROM contacts;
  r_contact c_contacts%ROWTYPE;
	DECLARE
	  record_name cursor_name%ROWTYPE;

3) Programmer/user - defined - want to create a record whose structure is not based on the existing ones

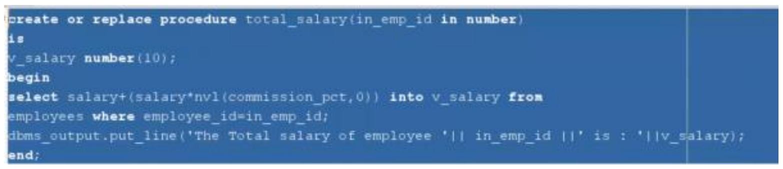
To declare a programmer-defined record, you use the following steps:

1. Define a record type that contains the structure you want in your record.
2. Declare a record based on the record type.

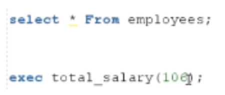
PL/SQL Procedure

procedure is a named block stored as a schema object in the Oracle Database.

PL/SQL Functions - A self- contained sub-program that is meant to do some specific well defined task.
Functions are named PL/SQL block which means they can be stored into the database as a database object and can be reused

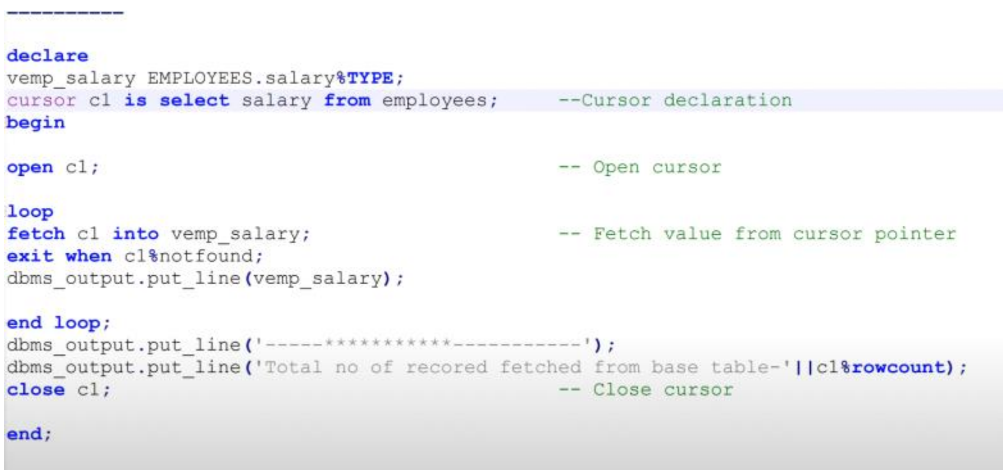


```
create or replace procedure total_salary(in_emp_id in number)
is
v_salary number(10);
begin
select salary+(salary*nvl(commission_pct,0)) into v_salary from
employees where employee_id=in_emp_id;
dbms_output.put_line('The Total salary of employee '|| in_emp_id ||' is : '||v_salary);
end;
```

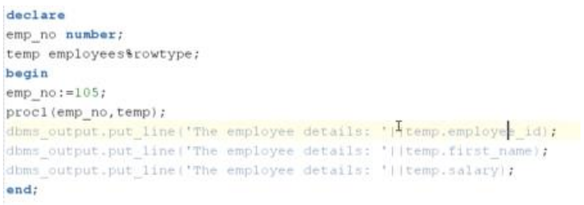


```
select * From employees;


exec total_salary(100);
```

Want for multiple rows -



```
-----------

declare
vemp_salary EMPLOYEES.salary%TYPE;
cursor c1 is select salary from employees;      --Cursor declaration
begin

open c1;                                         -- Open cursor

loop
fetch c1 into vemp_salary;                       -- Fetch value from cursor pointer
exit when c1%notfound;
dbms_output.put_line(vemp_salary);

end loop;
dbms_output.put_line('-----***********-----------');
dbms_output.put_line('Total no of recored fetched from base table-'||c1%rowcount);
close c1;                                        -- Close cursor

end;
```



```
declare
emp_no number;
temp employees%rowtype;
begin
emp_no:=105;
proc1(emp_no,temp);
dbms_output.put_line('The employee details: '||temp.employee_id);
dbms_output.put_line('The employee details: '||temp.first_name);
dbms_output.put_line('The employee details: '||temp.salary);
end;
```

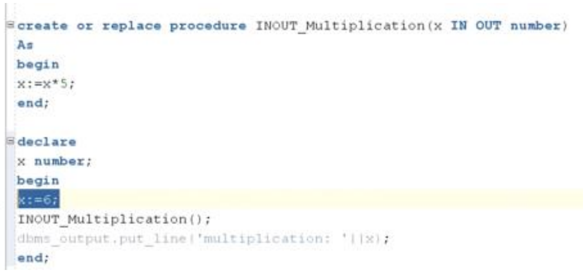
Tyes of subroutines or say sub-programs

1) PL/SQL Functions
2) PL/SQL Procedures

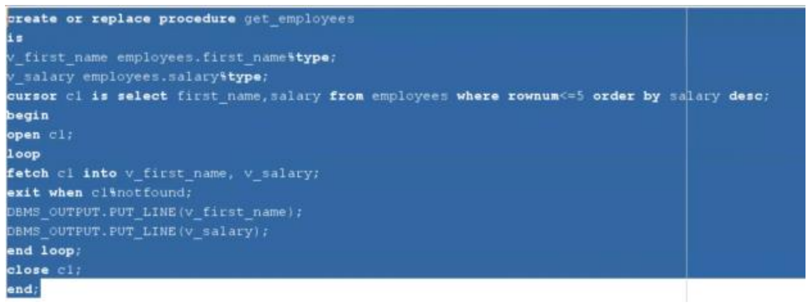
-- Calculate area of the circle
 	
create or replace function circle_area (radius Number)
return number IS
pi constant number (7,3) := 3.141;
area number (7,3);
BEGIN
area := pi * (radius * radius);
return area;
end;
/
	begin
	dbms_output.put_line(circle_area(25));
	end;
	/
		CREATE [OR REPLACE] FUNCTION function_name (parameter_list)
		RETURN return_type
		IS
		[declarative section]
		
		BEGIN
		
		[executable section]
		
		[EXCEPTION]
		
		[exception-handling section]
		
		END;

PL/SQL stored procedures - It is a self-contained subprogram that is meant to do some specific tasks.

Procedures are named PL/SQL blocks thus they can be reused, because they are stored into the database as a database object.
Unlike PL/SQL functions a stored procedure doesn't return any value.



```
create or replace procedure INOUT_Multiplication(x IN OUT number)
As
begin
x:=x*5;
end;

declare
x number;
begin
x:=6;
INOUT_Multiplication();
dbms_output.put_line('multiplication: '||x);
end;
```



```
create or replace procedure get_employees
is
v_first_name employees.first_name%type;
v_salary employees.salary%type;
cursor c1 is select first_name,salary from employees where rownum<=5 order by salary desc;
begin
open c1;
loop
fetch c1 into v_first_name, v_salary;
exit when c1%notfound;
DBMS_OUTPUT.PUT_LINE(v_first_name);
DBMS_OUTPUT.PUT_LINE(v_salary);
end loop;
close c1;
end;
```

create or replace procedure pr_shail IS
    var1 varchar(20) := 'Shailvi';
    var2 varchar(20) := 'Accen.com';
Begin
    dbms_output.put_line ('whats up internet? I am '
    || var1 || ' from ' || var2);
end pr_shail;
/
	execute pr_shail;
	---------------- 	
	
	exec pr_shail;
	-------------- 	
	begin
	    pr_shail;
	end;
	/
	
	Note : values returned by procedures can't be assigned into any variables unlike function
		create or replace procedure custom (cust_id number) -- cust_id is formal parameter
		is
		begin
		update customer set city = 'Deccan' where customer_id = cust_id;
		dbms_output.put_line('City updated successfully');
		end;
		/
		
		execute custom(2); -- 2 is actual parameter

Calling notation for subroutines --

Calling notation is a way of providing values to the parameters of a subroutine such as PL/SQL function or a stored procedure .

Types - Positional, Named, mixed calling notational

1) Positional notation -- We have to specify the value for each formal parameter in a sequential manner.

2) Named notation -- Pass values to the formal parameters using their name. This will in turn let you assign values to only required or say mandatory parameters.
   In order to assign values to the formal parameters using their names we use association operator. (Formal parameters => value)

```
create or replace function add_num
(var1 number, var2 number default 0, var3 number) return number
is
begin
        dbms_output.put_line('var1 -> '|| var1);
    dbms_output.put_line('var2 -> '|| var2);
    dbms_output.put_line('var3 -> '|| var3);
        return var1+var2+var3;
end;
/

--- positional calling notation --it will not work (3,,5)
begin
        dbms_output.put_line(add_num(3,,5));
end;
/

--if we try with named calling notation it will work
declare var_result number;
begin
        var_result := add_num (var3 => 5, var1 => 2);
        dbms_output.put_line('result -> '|| var_result);
end;
/
```

3) Mixed notation --

Packages -

Packages are stored libraries in the database which allows us to group related PL/SQL objects under one name.
It is named PL/SQL bocks which means permanently stored into database schema and can be referenced or reused by our program.
It includes stored procedures, PL/SQL functions, Database cursors, Type declarations and variables.

It is divided into 2 parts --
Specification -- header (mandatory) -- where we put the declaration of all the package elements. Whatever elements we declare here in this section are publicly available and can be referenced outside of the package.
Body -- optional -- provide actual structure to all the package elements.
-- It contains the implementation of the elements listed in the package specification. Contains both declaration of variables and as well as the definition of all the package elements.

Create Package --