- **My First PL/SQL Program.**

```
Declare
message varchar(20):='Hello World';
begin
dbms_output.put_line(message);
end;
/
```

- **If a given number is multiple of 10 then increment the number by 1 else, no action should take place.**

```
Declare
num int:=100;
begin
if(mod(num,10))=00
then num:=num+1;
else
null;
end if;
dbms_output.put_line(num);
end;
/
```

- **FIND THE GREATES AMONG TWO NUMBER**

```
Declare
a number;
b number;
begin
a:= :a;
b:= :b;
if a>b
 then
dbms_output.put_line('Large number is '||a);
else if b>c
then
dbms_output.put_line('Large number is '||b);
else
dbms_output.put_line('Both are same ');
end if;
end if;
end;
/
```

- **FIND THE GREATES AMONG THREE NUMBER**

```
Declare
   a number:=:a;
   b number:=:b;
   c number:=:c;
Begin
   dbms_output.put_line('Enter a:'||a);
   dbms_output.put_line('Enter b:'||b);
   dbms_output.put_line('Enter c:'||c);

if (a>b) and (a>c)
   then
   dbms_output.put_line('A is GREATEST'||A);
elsif (b>a) and (b>c)
   then
   dbms_output.put_line('B is GREATEST'||B);
else
   dbms_output.put_line('C is GREATEST'||C);
end if;
End;
/
```

- **Find Whether entered character is Vowel or Consonant**

```
DECLARE
C CHAR:= :c;
BEGIN
IF C='A' OR C='E' OR C='I' OR C='O' OR C='U'
THEN
DBMS_OUTPUT.PUT_LINE('VOWEL');
ELSE
DBMS_OUTPUT.PUT_LINE('CONSONANT');
END IF;
END;
/
```

- **Display the Grade name**

```
DECLARE
M1 NUMBER(2):=:M1;
M2 NUMBER(2):=:M2;
M3 NUMBER(2):=:M3;
TOTMARK NUMBER(5,2);
AVE NUMBER(5,2):=0;
BEGIN
TOTMARK:=M1+M2+M3;
AVE:=TOTMARK/3;
IF AVE>=60 THEN
DBMS_OUTPUT.PUT_LINE('THE DIVISION IS FIRST
'||AVE);
ELSIF AVE<60 AND AVE>=50 THEN
DBMS_OUTPUT.PUT_LINE('THE DIVISION IS SECOND
'||AVE);
ELSIF AVE<50 AND AVE>=35 THEN
DBMS_OUTPUT.PUT_LINE('THE DIVISION IS THIRD
'||AVE);
ELSE
DBMS_OUTPUT.PUT_LINE('FAIL '||AVE);
END IF;
END;
/
```

- **PL/SQL Program to display 1 to 10 numbers in descending (Reverse) order.**

```
DECLARE
      loop_start Integer := 1;
   BEGIN
      FOR i IN REVERSE loop_start..10 LOOP
  DBMS_OUTPUT.PUT_LINE('Number is ' || i);
      END LOOP;
   END;
  /
```

- **PL/SQL Program to accept a number from user and print number in reverse order.**

```
Declare
  num1 number(5);
  num2 number(5);
  rev number(5);
begin
  num1:=:num1;
  rev:=0;
  while num1>0
  loop
    num2:=num1 mod 10;
    rev:=num2+(rev*10);
    num1:=floor(num1/10);
  end loop;
  dbms_output.put_line('Reverse number is: '||rev);
end;
/
```

- **PL/SQL Program to Find Factorial of a**
- **PL/SQL Program to Find Factorial of a Number**

Number using For Loop.

```
Declare
   n number;
   fac number:=1;
   i number;

begin
   n:=:n;

   for i in 1..n
   loop
      fac:=fac*i;
   end loop;

   dbms_output.put_line('factorial of '||n ||'
is ' ||fac);
End;
/
```

using While Loop.

```
Declare
  n_counter   NUMBER := 5;
  n_factorial NUMBER := 1;
  n_temp      NUMBER;
BEGIN
  n_temp := n_counter;
  WHILE n_counter > 0
  LOOP
    n_factorial := n_factorial * n_counter;
    n_counter   := n_counter - 1;
  END LOOP;

  DBMS_OUTPUT.PUT_LINE('factorial of ' || n_temp ||
                 ' is ' || n_factorial);

END;
/
```

- **PL/SQL Program to display odd numbers Between 10 to 50.**

```
Declare
n number := 11;
begin
while n<=50
loop
dbms_output.put_line(n);
n := n+2;
end loop;
end;
/
```

- **PL/SQL Program to display numbers from 1 to 10.**

-using loop statement

```
Declare
I number;
BEGIN
   I:=1;
Loop
  Dbms_output.put_line(I);
   I:=I+1;
   Exit when I>10;
End loop;
END;
/
```

- using FOR loop statement

```
Begin
For I in 1..10
loop

Dbms_output.put_line(I);
End loop;
End;
/
```

- **PL/SQL Program to display numbers**
- **PL/SQL Program to display numbers from 50**

from 50 to 60 using loop. (When number is greater than 60 display the message ''Exiting from loop'')

```
Declare
a number:= 50;
begin
loop
  Dbms_output.put_line(a);
a := a+1;
exit when a>60;
end loop;
dbms_output.put_line ('Exiting From Loop');
end;
/
```

to 60 using FOR loop. (When number is greater than 60 display the message ''Exiting from loop'')

```
Begin
For I in 50..60
loop
   Dbms_output.put_line(I);
End loop;
dbms_output.put_line ('Exiting From Loop');
end;
/
```

- **Exception Handling:-**

(The below program displays the name and address of a customer whose ID is given. if there is no customer with ID value which we have provided in our database, the program raises the run-time exception **'No Such Customer!'**, which is captured in the EXCEPTION block.)

- **We will be using the CUSTOMER table we had created.**

```
{create table customer(id number(10),name
varchar(20),address varchar(20))
insert into customer
values(8,'Ram','Kopargaon')}

DECLARE
  c_id customer.id%type :=8;
  c_name  customer.name%type;
  c_addr customer.address%type;
BEGIN
  SELECT  name, address INTO  c_name,
c_addr
  FROM customer
  WHERE id = c_id;
  DBMS_OUTPUT.PUT_LINE ('Name: '||
c_name);
  DBMS_OUTPUT.PUT_LINE ('Address: ' ||
c_addr);
```

```
EXCEPTION
  WHEN no_data_found THEN
    dbms_output.put_line('No such customer!');
  WHEN others THEN
    dbms_output.put_line('Error!');
END;
/
```

- **Procedures.**

Where,

Syntax:-

```
CREATE [OR REPLACE] PROCEDURE
procedure_name

[(parameter_name [IN | OUT | IN OUT]
type [, ...])]

{IS | AS}

BEGIN

  < procedure_body >

END procedure_name;
```

- *procedure-name* specifies the name of the procedure.
- [OR REPLACE] option allows the modification of an existing procedure.
- The optional parameter list contains name, mode and types of the parameters. **IN** represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
- *procedure-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone procedure.

- **Creating a Simple or Standalone Procedure.**

```
CREATE OR REPLACE PROCEDURE greetings
AS
BEGIN
  dbms_output.put_line('Hello World!');
END;
/
```

- **Deleting a Standalone Procedure**

```
DROP PROCEDURE procedure-name;
```

- **IN & OUT Mode Example**

This program finds the minimum of two values. Here, the procedure takes two numbers using the IN mode and returns their minimum using the OUT parameters.

```
DECLARE
  a number;
  b number;
  c number;
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;
BEGIN
  a:= 23;
  b:= 45;
  findMin(a, b, c);
  dbms_output.put_line(' Minimum of (23, 45) : ' || c);
END;
/
```

- **IN & OUT Mode Example**

This procedure computes the square of value of a passed value. This example shows how outside and OUT represents the parameter that will be used to return a value outside of the procedure.

we can use the same parameter to accept a value and then return another result.

```
DECLARE
  a number;
PROCEDURE squareNum(x IN OUT number) IS
BEGIN
 x := x * x;
END;
BEGIN
  a:= 23;
  squareNum(a);
dbms_output.put_line(' Square of (23): ' || a);
END;
/
```

- **Creating a Function**

A standalone function is created using the CREATE FUNCTION statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows –

```
CREATE [OR REPLACE] FUNCTION
function_name
[(parameter_name [IN | OUT | IN OUT] type
[, ...])]
RETURN return_datatype
{IS | AS}
BEGIN
   < function_body >
END [function_name];
```

Where,

function-name specifies the name of the function.

[OR REPLACE] option allows the modification of an existing function.

The optional parameter list contains name, mode and types of the parameters. IN represents the value that will be passed from

CREATE OR REPLACE FUNCTION totalCustomers

The function must contain a return statement.

The RETURN clause specifies the data type you are going to return from the function.

function-body contains the executable part.

The AS keyword is used instead of the IS keyword for creating a standalone function.

Example

The following example illustrates how to create and call a standalone function. This function returns the total number of CUSTOMERS in the customers table.

We will use the CUSTOMERS table

```
Select * from customers;

+----+----------+-----+-----------+----------+
| 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 |
+----+----------+-----+-----------+----------+
```

Following program calls the function totalCustomers from an anonymous block –

```
RETURN number IS
  total number(2) := 0;
BEGIN
  SELECT count(*) into total
  FROM customers;
    RETURN total;
END;
/
```

When the above code is executed using the SQL prompt, it will produce the following result –

```
Function created.
```

- **Calling a Function**

While creating a function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. When a program calls a function, the program control is transferred to the called function.

A called function performs the defined task and when its return statement is executed or when the last end statement is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name and if the function returns a value, then you can store the returned value.

```
DECLARE

  c number(2);

BEGIN

  c := totalCustomers();

  dbms_output.put_line('Total no. of Customers: ' ||
c);

END;

/
```

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

```
Total no. of Customers: 6
```

- **Cursors:-**

%ISOPEN: -Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL

A cursor is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the active set.

- 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. Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

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

%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.

- Triggers

Triggers are stored programs, which are automatically executed or fired when some statement.

%ROWCOUNT: -Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

- **We will be using the CUSTOMER table we had created.**

{create table customer(id number(10),name varchar(20),address varchar(20))
insert into customer values(8,'Ram','Kopargaon')}

```
DECLARE
  total_rows number(2);
BEGIN
  UPDATE customer
  SET address = 'Shirdi';
  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 syntax for creating an explicit cursor is –

```
CURSOR cursor_name IS select_statement;
```

Working with an explicit cursor includes the following steps –Declaring the cursor for initializing the memory
-Opening the cursor for allocating the memory
-Fetching the cursor for retrieving the data
-Closing the cursor to release the allocated memory

- Creating Triggers

The syntax for creating a trigger is –

```
CREATE [OR REPLACE ] TRIGGER trigger_name
```

events occur. Triggers are, in fact, written to be executed in response to any of the following events –

-A database manipulation (DML) statement (DELETE, INSERT, or UPDATE)

-A database definition (DDL) statement (CREATE, ALTER, or DROP).

-A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers can be defined on the table, view, schema, or database with which the event is associated.

Benefits of Triggers:-

Triggers can be written for the following purposes –

- Generating some derived column values automatically
- Enforcing referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

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

Where,

- CREATE [OR REPLACE] TRIGGER trigger_name – Creates or replaces an existing trigger with the *trigger_name*.
- {BEFORE | AFTER | INSTEAD OF} – This specifies when the trigger will be executed. The INSTEAD OF clause is used for creating trigger on a view.
- {INSERT [OR] | UPDATE [OR] | DELETE} – This specifies the DML operation.
- [OF col_name] – This specifies the column name that will be updated.
- [ON table_name] – This specifies the name of the table associated with the trigger.
- [REFERENCING OLD AS o NEW AS n] – This allows you to refer new and old values for various DML statements, such as INSERT, UPDATE, and DELETE.

- [FOR EACH ROW] – This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected. Otherwise the trigger will execute just

```
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;
```

once when the SQL statement is executed, which is called a table level trigger.

- WHEN (condition) – This provides a condition for rows for which the trigger would fire. This clause is valid only for row-level triggers.

Example

To start with, we will be using the CUSTOMERS table we had created and used in the previous chapters –

```
Select * from customers;
+----+----------+-----+-----------+----------+
| 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 –

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 –

```
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;
/
```

- ## Triggering a Trigger

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 the CUSTOMERS table, the 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, old salary is not available and the above result comes as null. Let us now perform one more DML operation on the CUSTOMERS table. The UPDATE statement will update an existing record in the table –

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

When a record is updated in the CUSTOMERS table, the 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
```

- ## Package

Packages are schema objects that groups logically related PL/SQL types, variables, and subprograms.

and other private declarations, which are hidden from the code outside the package.

The **CREATE PACKAGE BODY** Statement is used for creating the package body. The following code snippet shows the package body declaration for the ***cust_sal*** package created

A package will have two mandatory parts –

- Package specification
- Package body or definition

-Package Specification

The specification is the interface to the package. It just **DECLARES** the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package. In other words, it contains all information about the content of the package, but excludes the code for the subprograms.

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.

The following code snippet shows a package specification having a single procedure. You can have many global variables defined and multiple procedures or functions inside a package.

```
CREATE PACKAGE cust_sal AS
  PROCEDURE find_sal(c_id customers.id%type);
END cust_sal;
/
```

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

```
Package created.
```

Package Body

The package body has the codes for various methods declared in the package specification above. I assumed that we already have CUSTOMERS table created in our database as mentioned in the PL/SQL - Variables chapter.

```
CREATE OR REPLACE PACKAGE BODY cust_sal AS

  PROCEDURE find_sal(c_id customers.id%TYPE) IS
  c_sal customers.salary%TYPE;
  BEGIN
    SELECT salary INTO c_sal
    FROM customers
    WHERE id = c_id;
    dbms_output.put_line('Salary: '|| c_sal);
  END find_sal;
END cust_sal;
/
```

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

```
Package body created.
```

Using the Package Elements

The package elements (variables, procedures or functions) are accessed with the following syntax –

```
package_name.element_name;
```

Consider, we already have created the above package in our database schema, the following program uses the ***find_sal*** method of the ***cust_sal*** package –

```
DECLARE
  code customers.id%type := &cc_id;
BEGIN
  cust_sal.find_sal(code);
END;
/
```

O/P:-it prompts to enter the customer ID and when you enter an ID, it displays the corresponding salary as follows –

```
Enter value for cc_id: 1
Salary: 3000
```