PL/SQL

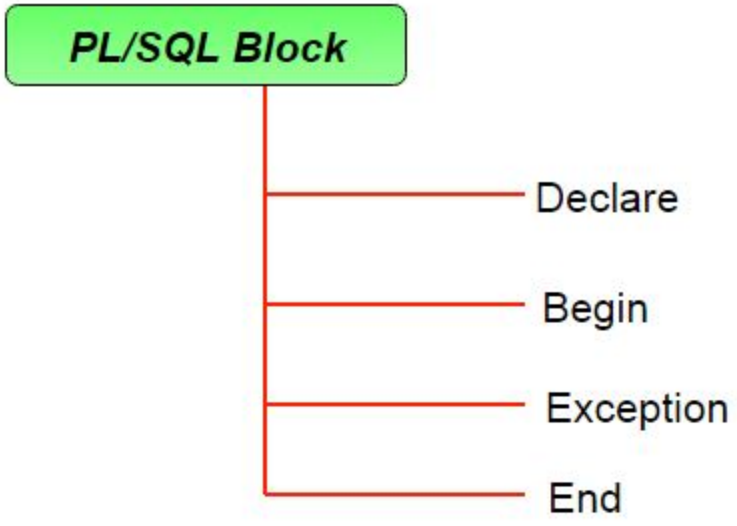
Oracle

Introduction

- PL/SQL is a block structured language that enables developers to combine the power of SQL with procedural statements.
- PL/SQL is an embedded language. PL/SQL only can execute in an Oracle Database. It was not designed to use as a standalone language like Java, C#, and C++. In other words, you cannot develop a PL/SQL program that runs on a system that does not have an Oracle Database.

Structure of PL/SQL Block:

- PL/SQL have two types of Blocks
  - Anonymous blocks
  - Named blocks
  - Declare and exceptior sections are optional.



  - Typically, each block performs a logical action in the program

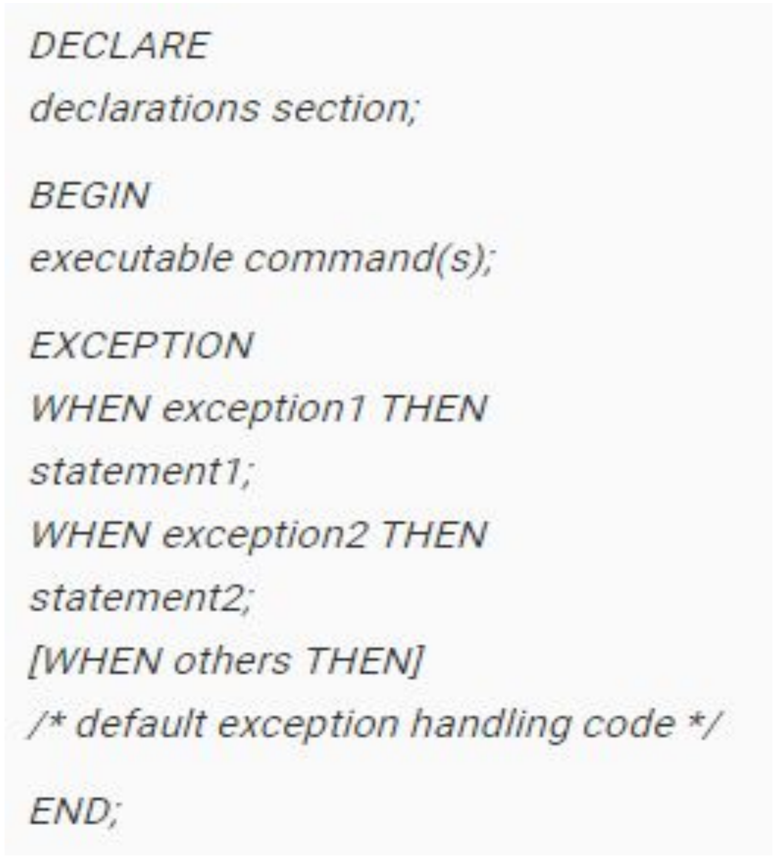
Structure of PL/SQL Block:

- Declare section starts with **DECLARE** keyword in which variables, constants, records as cursors can be declared which stores data temporarily. It basically consists definition of PL/SQL identifiers. This part of the code is optional.

- Execution section starts with **BEGIN** and ends with **END** keyword. This is a mandatory section and here the program logic is written to perform any task like loops and conditional statements. It supports all DML commands, DDL commands and SQL*PLUS built-in functions as well.

- Exception section starts with **EXCEPTION** keyword. This section is optional which contains statements that are executed when a run-time error occurs. Any exceptions can be handled in this section.

Basic structure of PL/SQL Program



```
DECLARE
declarations section;

BEGIN
executable command(s);

EXCEPTION
WHEN exception1 THEN
statement1;
WHEN exception2 THEN
statement2;
[WHEN others THEN]
/* default exception handling code */

END;
```

First Program

```
SET SERVEROUTPUT ON;
BEGIN
  DBMS_OUTPUT.put_line ('Hello World!');
END;
/
```

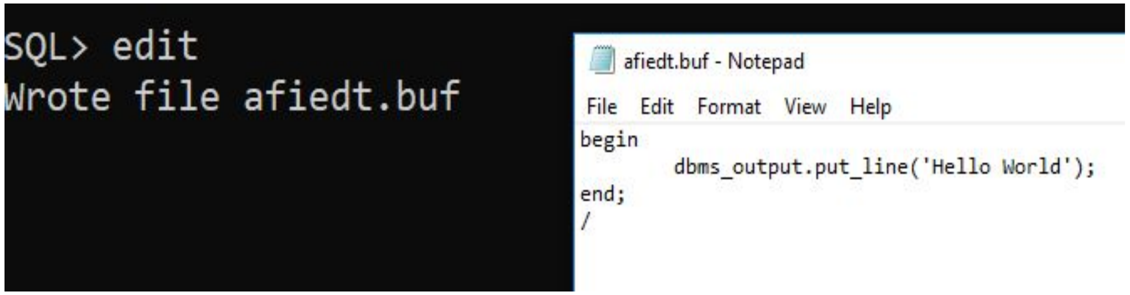
- ***Output:***

  ```
    Hello World!
  PL/SQL procedure successfully completed.
  ```

- ***Explanation:***
  - **<u>SET SERVEROUTPUT ON</u>**: It is used to display the buffer used by the dbms_output.
  - **<u>Slash (/) after END;</u>:** The slash (/) tells the SQL*Plus to execute the block.
  - **<u>PL/SQL procedure successfully completed.</u>:** It is displayed when the code is compiled and executed successfully.

First Program

- If you want to edit the code block, use the edit command. SQL*Plus will write the code block to a file and open it in a text editor as shown in the following picture:



```
SQL> edit
Wrote file afiedt.buf
```

```
begin
	dbms_output.put_line('Hello World');
end;
/
```

- You can change the contents of the file like the following:

```
begin
	dbms_output.put_line('Hello There');
end;
/
```

- And save and close the file. The contents of the file will be written to the buffer and recompiled.
- After that, you can execute the code block again, it will use the new code.

Second Program

- The next anonymous block example adds an exception-handling section which catches ZERO_DIVIDE exception raised in the executable section and displays an error message.

```
DECLARE
    v_result NUMBER;
BEGIN
    v_result := 1 / 0;
EXCEPTION
    WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE( ‘divisor cannot be zero’ );
    END;
/
```

Output:

```
divisor cannot be zero
```

PL/SQL identifiers

- PL/SQL identifiers are name given to constants, variables, exceptions, procedures, cursors, and reserved words. The identifiers consist of a letter optionally followed by more letters, numerals, dollar signs, underscores, and number signs and should not exceed 30 characters.
- By default, identifiers are not case-sensitive. So you can use integer or INTEGER to represent a numeric value. You cannot use a reserved keyword as an identifier.

The PL/SQL Comments

- Program comments are explanatory statements that can be included in the PL/SQL code that you write and helps anyone reading its source code. All programming languages allow some form of comments.

- The PL/SQL supports single-line and multi-line comments. All characters available inside any comment are ignored by the PL/SQL compiler. The PL/SQL single-line comments start with the delimiter -- (double hyphen) and multi-line comments are enclosed by /* and */.

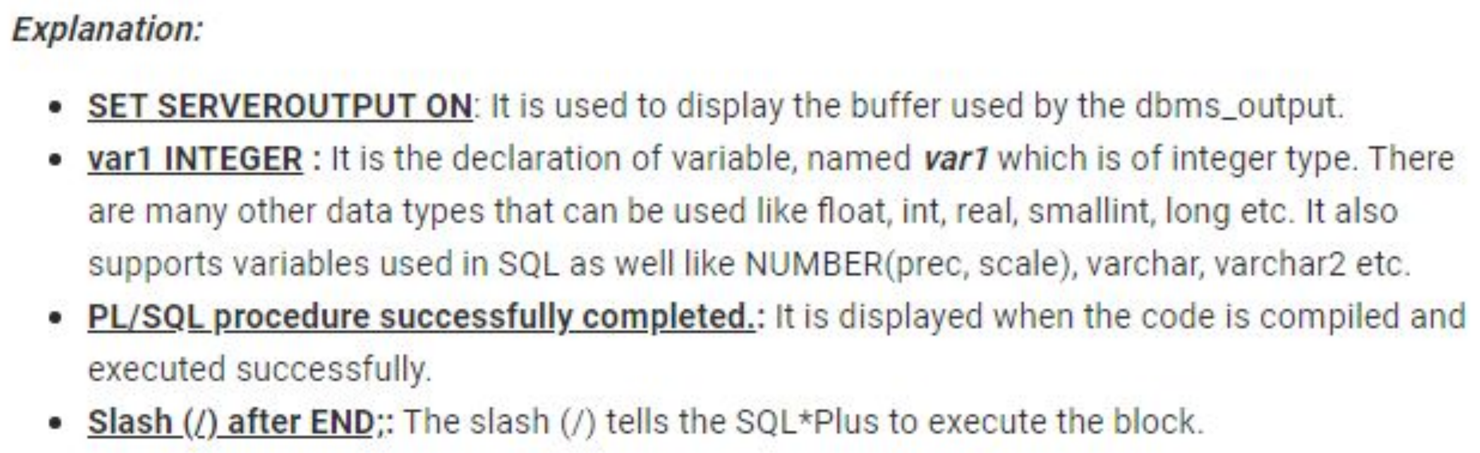
Variables

- Like several other programming languages, variables in PL/SQL must be declared prior to its use. They should have a valid name and data type as well.
- Syntax for declaration of variables:

```
variable_name datatype [NOT NULL := value ];
```

Variables

```
SQL> SET SERVEROUTPUT ON;
SQL> DECLARE
    var1 INTEGER;
    var2 REAL;
    var3 varchar2(20) ;
BEGIN
    null;
END;
/
```



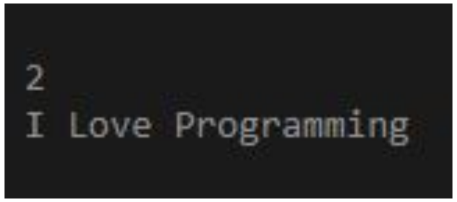
Explanation:

- <u>**SET SERVEROUTPUT ON**</u>: It is used to display the buffer used by the dbms_output.
- <u>**var1 INTEGER**</u> **:** It is the declaration of variable, named ***var1*** which is of integer type. There are many other data types that can be used like float, int, real, smallint, long etc. It also supports variables used in SQL as well like NUMBER(prec, scale), varchar, varchar2 etc.
- <u>**PL/SQL procedure successfully completed.**</u>**:** It is displayed when the code is compiled and executed successfully.
- <u>**Slash (/) after END**</u>;: The slash (/) tells the SQL*Plus to execute the block.

INITIALISING and Displaying VARIABLES

```
SQL> SET SERVEROUTPUT ON;
SQL> DECLARE
       var1 INTEGER := 2 ;
       var2 varchar2(20) := 'I Love Programming' ;
BEGIN
        dbms_output.put_line(var1);
       dbms_output.put_line(var2);
END;
```

Output:

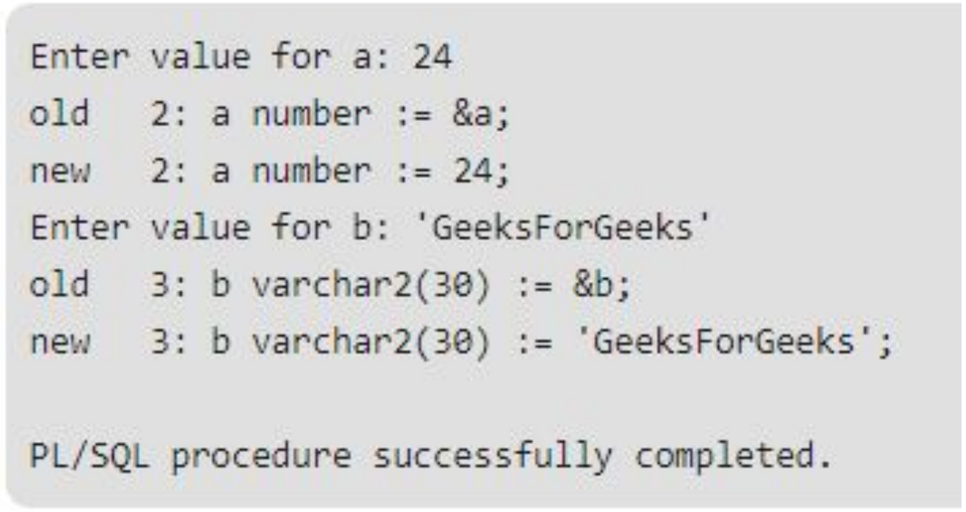


```
2
I Love Programming
```

Taking input from user

```
SQL> SET SERVEROUTPUT ON;

SQL> DECLARE

        -- taking input for variable a
        a number := &a;

        -- taking input for variable b
        b varchar2(30) := &b;

BEGIN
        null;

END;
/
```

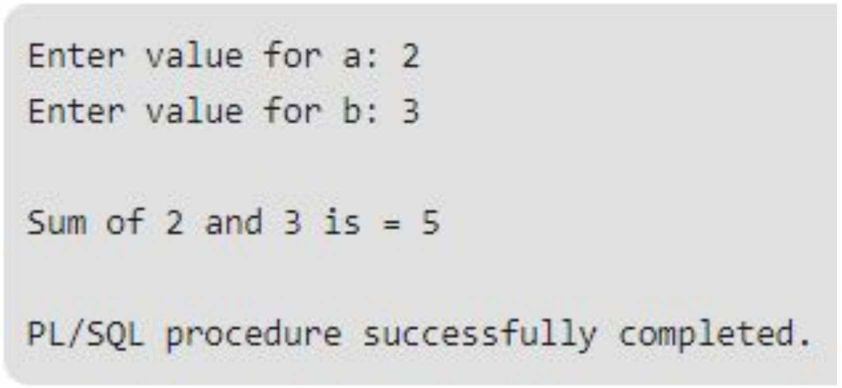
Output:



```
Enter value for a: 24
old   2: a number := &a;
new   2: a number := 24;
Enter value for b: 'GeeksForGeeks'
old   3: b varchar2(30) := &b;
new   3: b varchar2(30) := 'GeeksForGeeks';

PL/SQL procedure successfully completed.
```

PL/SQL code to print sum of two numbers taken from the user.

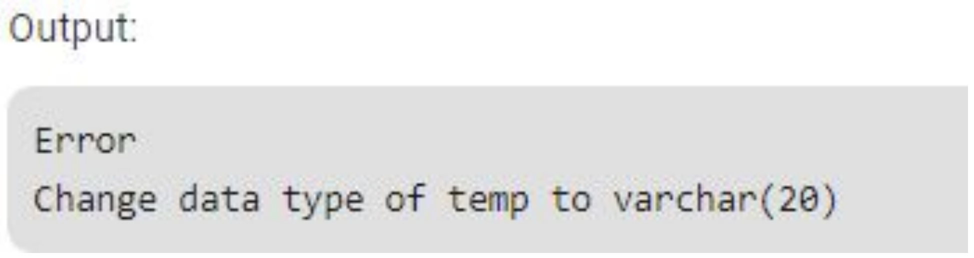
```
SQL> SET SERVEROUTPUT ON;

SQL> DECLARE

        -- taking input for variable a
        a integer := &a ;

        -- taking input for variable b
        b integer := &b ;
        c integer ;

BEGIN
        c := a + b ;
        dbms_output.put_line('Sum of '||a||' and '||b||' is = '||c);

END;
/
```



```
Enter value for a: 2
Enter value for b: 3

Sum of 2 and 3 is = 5

PL/SQL procedure successfully completed.
```

VALUE_ERROR Program

```
DECLARE
temp number;

BEGIN
temp:=‘Hello World’;

EXCEPTION
WHEN value_error THEN
dbms_output.put_line('Error');
dbms_output.put_line('Change data type of temp to varchar(20)');

END;
```

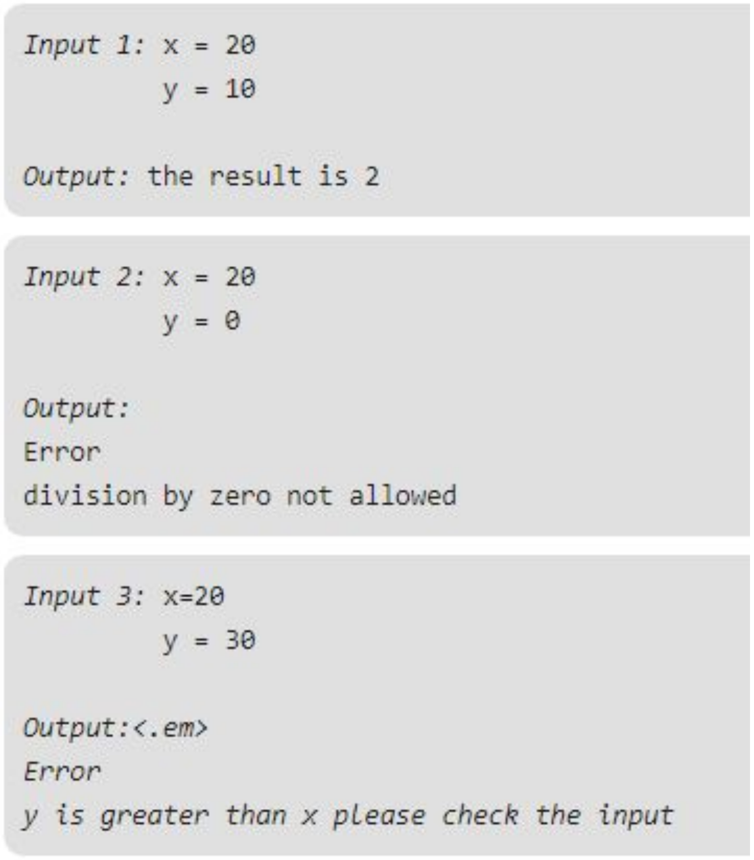

Output:

```
Error
Change data type of temp to varchar(20)
```

User defined exceptions

```
DECLARE
x int:=&x; /*taking value at run time*/
y int:=&y;
div_r float;
exp1 EXCEPTION;
exp2 EXCEPTION;
BEGIN
IF y=0 then
        raise exp1;
ELSIF y > x then
        raise exp2;
ELSE
        div_r:= x / y;
        dbms_output.put_line('the result is
      '||div_r);
END IF;
EXCEPTION
WHEN exp1 THEN
        dbms_output.put_line('Error');
        dbms_output.put_line('division by
      zero not allowed');

WHEN exp2 THEN
        dbms_output.put_line('Error');
        dbms_output.put_line('y is greater
      than x please check the input');

END;
```

User defined exceptions : Output



```
Input 1: x = 20
         y = 10

Output: the result is 2
```

```
Input 2: x = 20
         y = 0

Output:
Error
division by zero not allowed
```

```
Input 3: x=20
         y = 30

Output:<.em>
Error
y is greater than x please check the input
```

PL/SQL Execution Environment

- The PL/SQL engine resides in the Oracle engine.The Oracle engine can process not only single SQL statement but also block of many statements. The call to Oracle engine needs to be made only once to execute any number of SQL statements if these SQL statements are bundled inside a PL/SQL block.

PL/SQL Control Structures

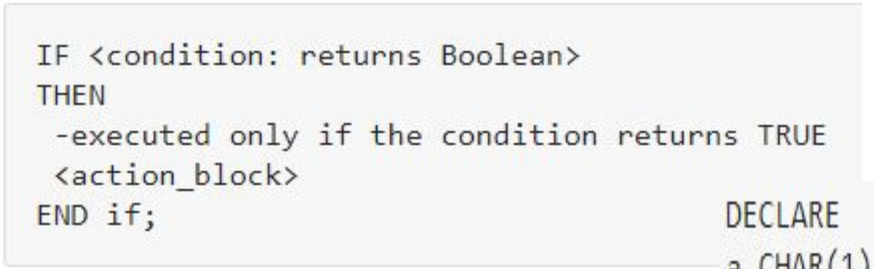
- Testing Conditions:
  - IF and
  - CASE Statements
- Controlling Loop Iterations:
  - LOOP and
  - EXIT Statements
- Sequential Control:
  - GOTO and
  - NULL Statements

IF and CASE Statements

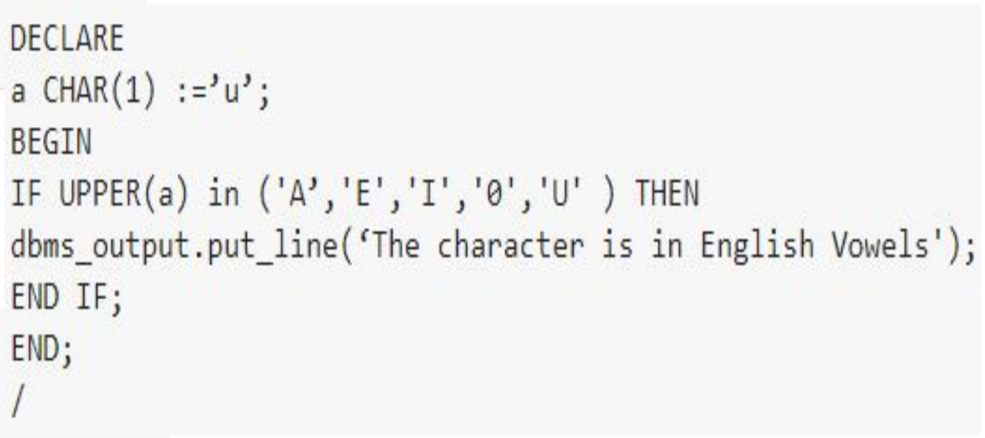
- There are three forms of IF statements:
  - IF-THEN,
  - IF-THEN-ELSE, and
  - IF-THEN-ELSIF

IF-THEN

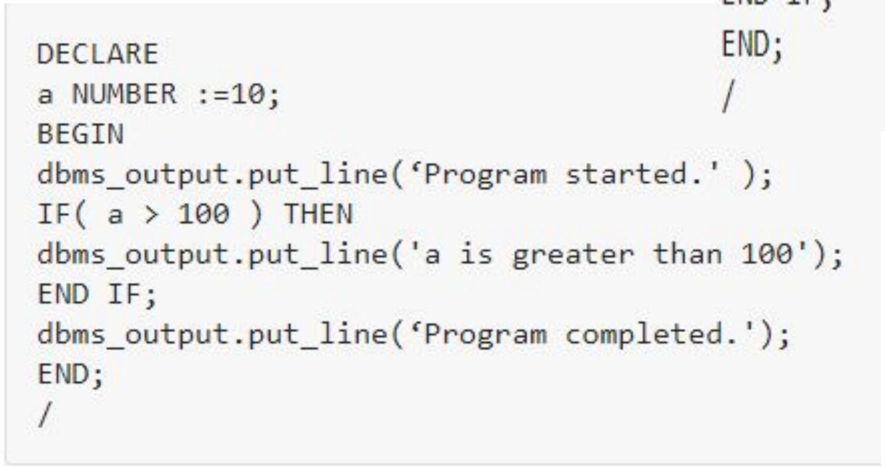
Syntax for IF THEN Statements:



```
IF <condition: returns Boolean>
THEN
 -executed only if the condition returns TRUE
 <action_block>
END if;
```

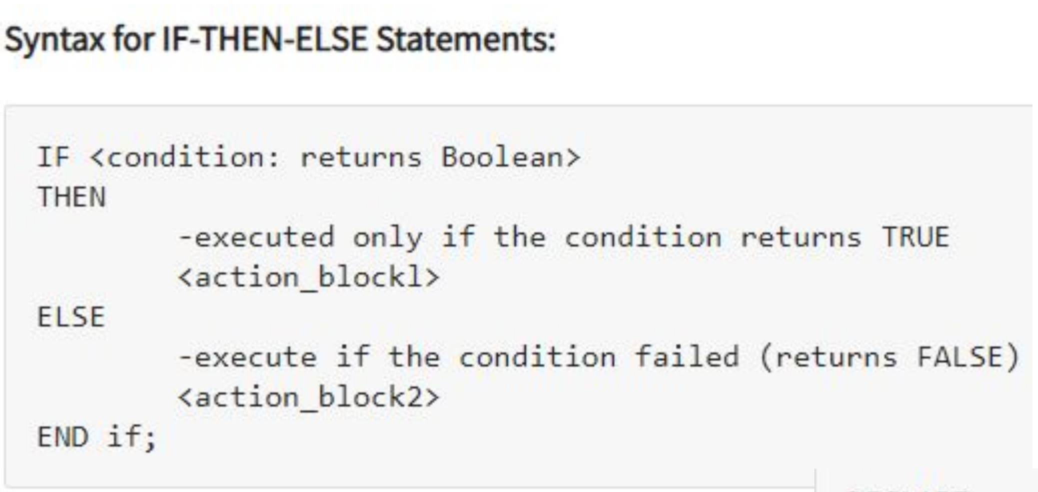


```
DECLARE
a CHAR(1) :='u';
BEGIN
IF UPPER(a) in ('A','E','I','0','U' ) THEN
dbms_output.put_line('The character is in English Vowels');
END IF;
END;
/
```



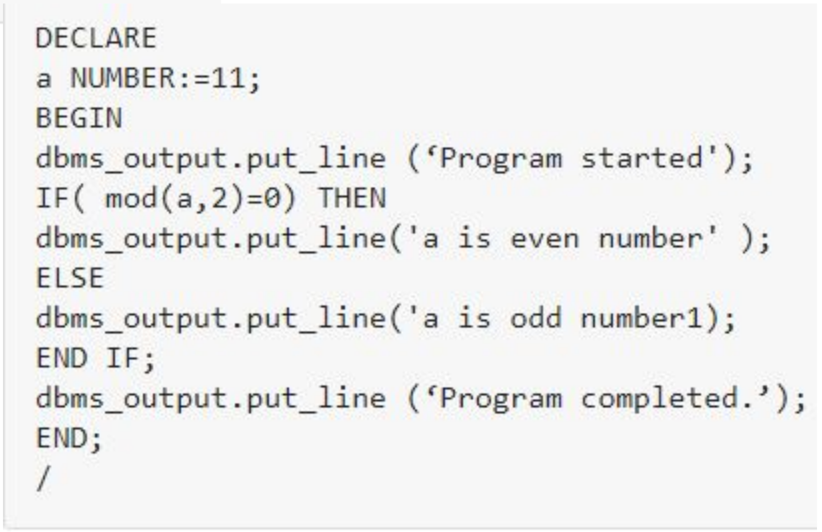
```
DECLARE
a NUMBER :=10;
BEGIN
dbms_output.put_line('Program started.' );
IF( a > 100 ) THEN
dbms_output.put_line('a is greater than 100');
END IF;
dbms_output.put_line('Program completed.');
END;
/
```

IF-THEN-ELSE Statement



Syntax for IF-THEN-ELSE Statements:

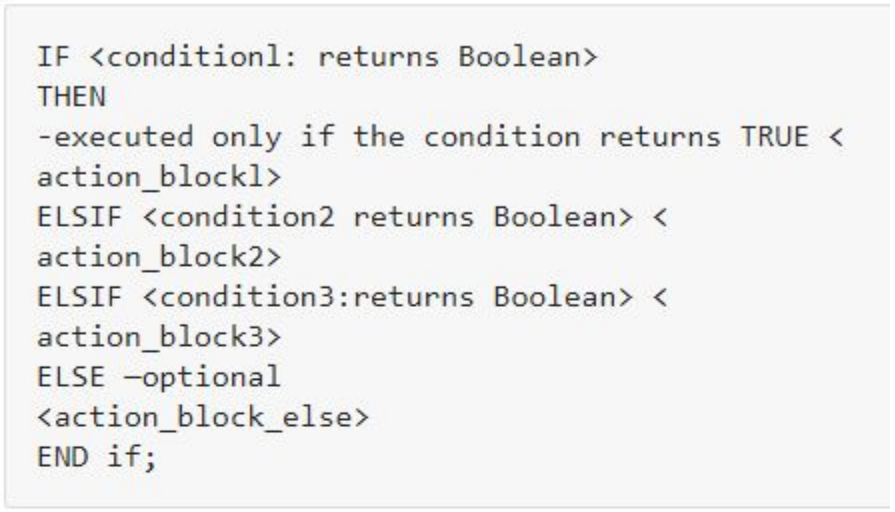
```
IF <condition: returns Boolean>
THEN
        -executed only if the condition returns TRUE
        <action_blockl>
ELSE
        -execute if the condition failed (returns FALSE)
        <action_block2>
END if;
```



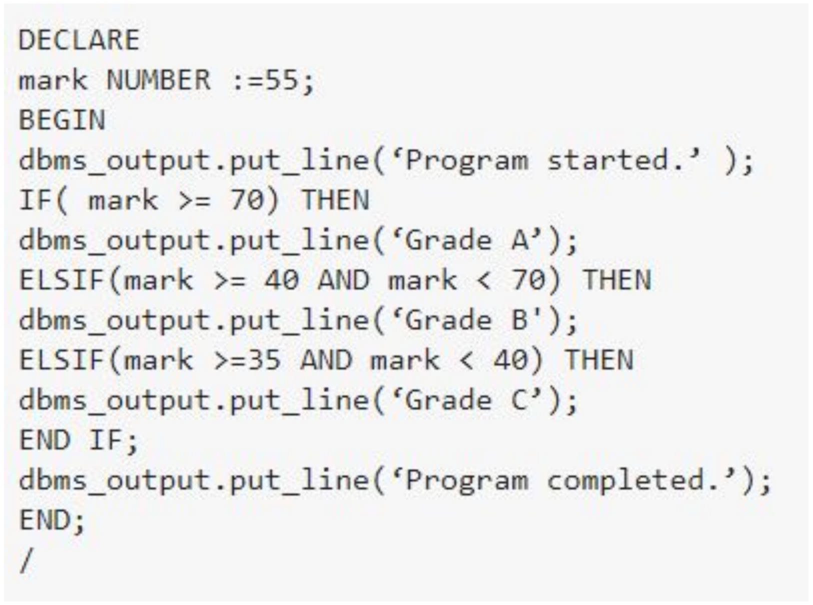
```
DECLARE
a NUMBER:=11;
BEGIN
dbms_output.put_line (‘Program started');
IF( mod(a,2)=0) THEN
dbms_output.put_line('a is even number' );
ELSE
dbms_output.put_line('a is odd number1);
END IF;
dbms_output.put_line (‘Program completed.’);
END;
/
```

IF-THEN-ELSIF

Syntax for IF-THEN-ELSIF Statements:

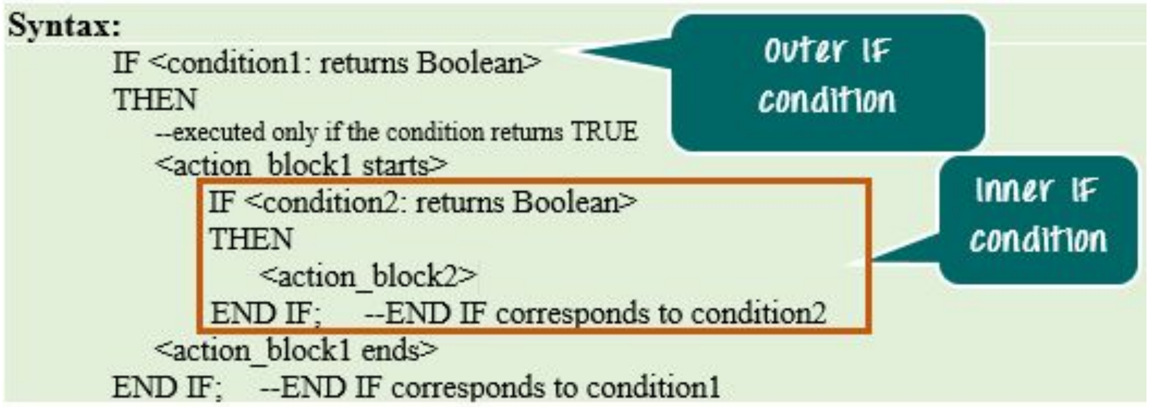


```
IF <conditionl: returns Boolean>
THEN
-executed only if the condition returns TRUE <
action_blockl>
ELSIF <condition2 returns Boolean> <
action_block2>
ELSIF <condition3:returns Boolean> <
action_block3>
ELSE —optional
<action_block_else>
END if;
```



```
DECLARE
mark NUMBER :=55;
BEGIN
dbms_output.put_line(‘Program started.’ );
IF( mark >= 70) THEN
dbms_output.put_line(‘Grade A’);
ELSIF(mark >= 40 AND mark < 70) THEN
dbms_output.put_line(‘Grade B');
ELSIF(mark >=35 AND mark < 40) THEN
dbms_output.put_line(‘Grade C’);
END IF;
dbms_output.put_line(‘Program completed.’);
END;
/
```

NESTED-IF Statement

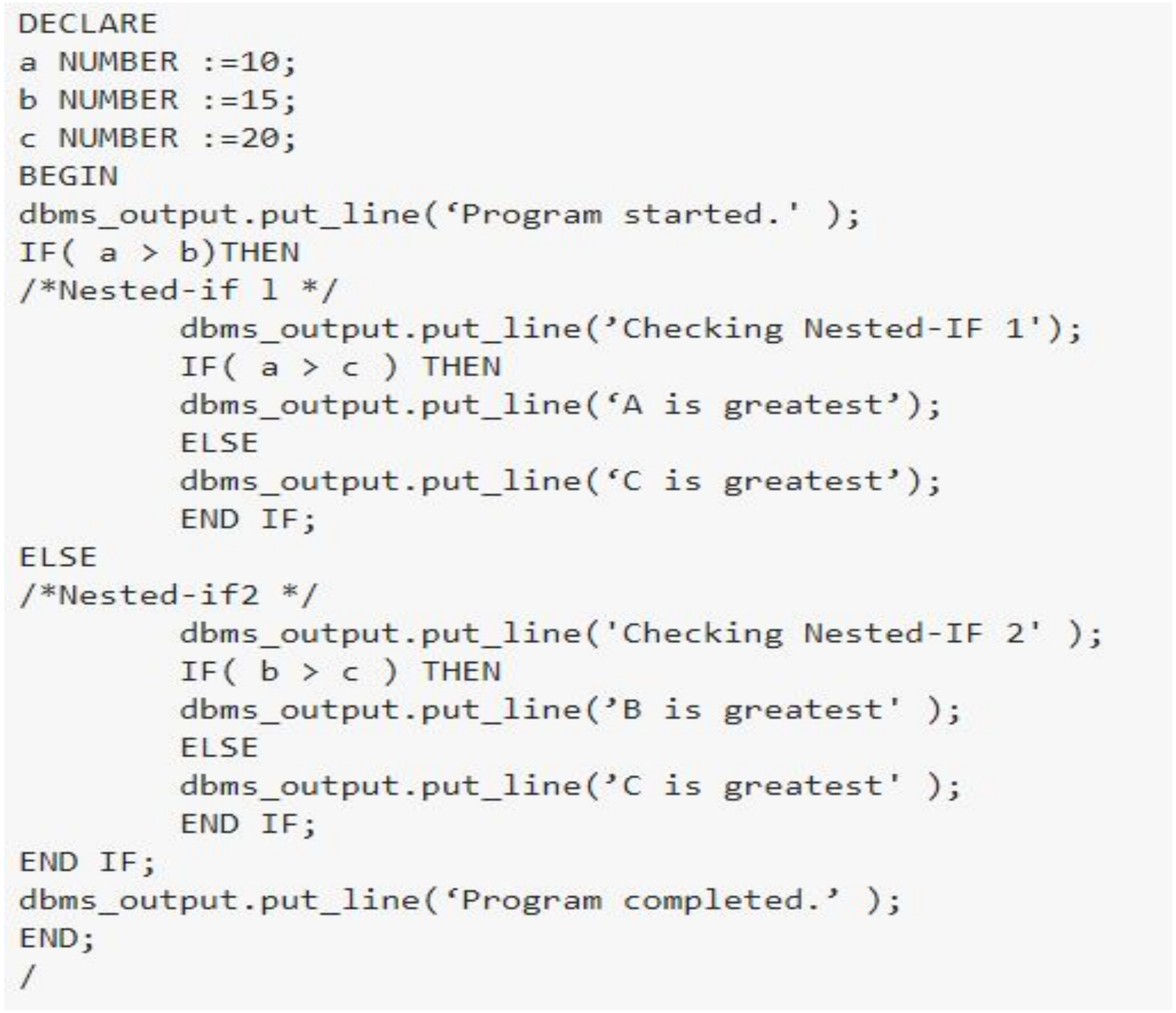


Syntax:

```
IF <condition1: returns Boolean>
THEN
   --executed only if the condition returns TRUE
   <action_block1 starts>
      IF <condition2: returns Boolean>
      THEN
         <action_block2>
      END IF;   --END IF corresponds to condition2
   <action_block1 ends>
END IF;   --END IF corresponds to condition1
```

Outer IF condition
Inner IF condition

NESTED-IF Statement



```
DECLARE
a NUMBER :=10;
b NUMBER :=15;
c NUMBER :=20;
BEGIN
dbms_output.put_line(‘Program started.' );
IF( a > b)THEN
/*Nested-if l */
        dbms_output.put_line(’Checking Nested-IF 1');
        IF( a > c ) THEN
        dbms_output.put_line(‘A is greatest’);
        ELSE
        dbms_output.put_line(‘C is greatest’);
        END IF;
ELSE
/*Nested-if2 */
        dbms_output.put_line('Checking Nested-IF 2' );
        IF( b > c ) THEN
        dbms_output.put_line(’B is greatest' );
        ELSE
        dbms_output.put_line(’C is greatest' );
        END IF;
END IF;
dbms_output.put_line(‘Program completed.’ );
END;
/
```

Case Statements

Syntax:

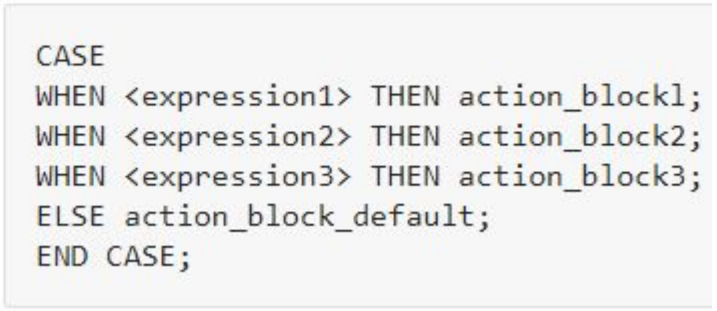
```
CASE (expression)
 WHEN <value1> THEN action_block1;
 WHEN <value2> THEN action_block2;
 WHEN <value3> THEN action_block3;
 ELSE action_block_default;
END CASE;
```

```
DECLARE
  grade CHAR(1);
BEGIN
  grade := 'B';
  CASE grade
    WHEN 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
    WHEN 'B' THEN DBMS_OUTPUT.PUT_LINE('Very Good');
    WHEN 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
    WHEN 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
    WHEN 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
    ELSE DBMS_OUTPUT.PUT_LINE('No such grade');
  END CASE;
END;
/
```

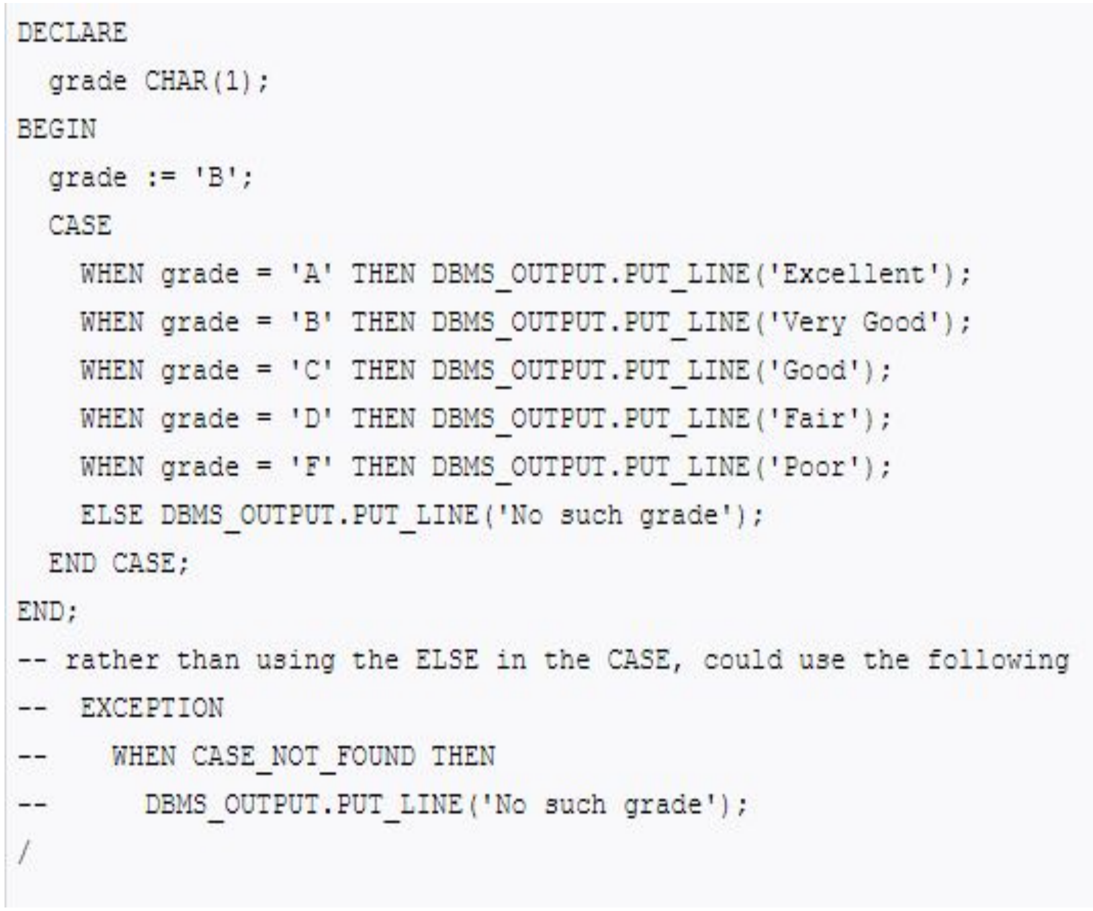
Equivalent IF THEN Ladder

```
DECLARE
  grade CHAR(1);
BEGIN
  grade := 'B';
  IF grade = 'A' THEN
    DBMS_OUTPUT.PUT_LINE('Excellent');
  ELSIF grade = 'B' THEN
    DBMS_OUTPUT.PUT_LINE('Very Good');
  ELSIF grade = 'C' THEN
    DBMS_OUTPUT.PUT_LINE('Good');
  ELSIF grade = 'D' THEN
    DBMS_OUTPUT. PUT_LINE('Fair');
  ELSIF grade = 'F' THEN
    DBMS_OUTPUT.PUT_LINE('Poor');
  ELSE
    DBMS_OUTPUT.PUT_LINE('No such grade');
  END IF;
ENd;
/
```

Searched CASE Statement



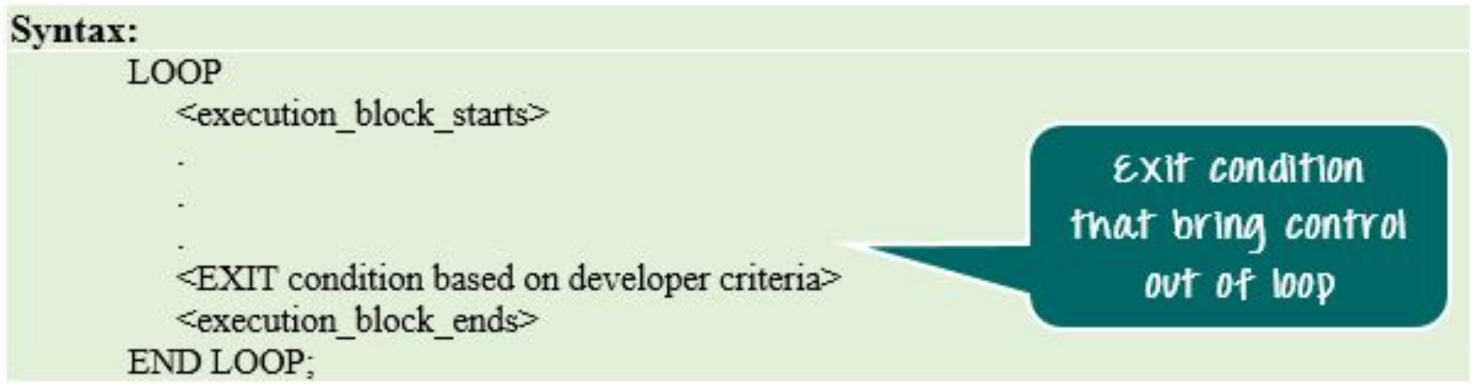
```
CASE
WHEN <expression1> THEN action_blockl;
WHEN <expression2> THEN action_block2;
WHEN <expression3> THEN action_block3;
ELSE action_block_default;
END CASE;
```



```
DECLARE
  grade CHAR(1);
BEGIN
  grade := 'B';
  CASE
    WHEN grade = 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
    WHEN grade = 'B' THEN DBMS_OUTPUT.PUT_LINE('Very Good');
    WHEN grade = 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
    WHEN grade = 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
    WHEN grade = 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
    ELSE DBMS_OUTPUT.PUT_LINE('No such grade');
  END CASE;
END;
-- rather than using the ELSE in the CASE, could use the following
--   EXCEPTION
--     WHEN CASE_NOT_FOUND THEN
--       DBMS_OUTPUT.PUT_LINE('No such grade');
/
```

LOOP Statements

- Basic loop statement
- For loop statement
- While loop statement



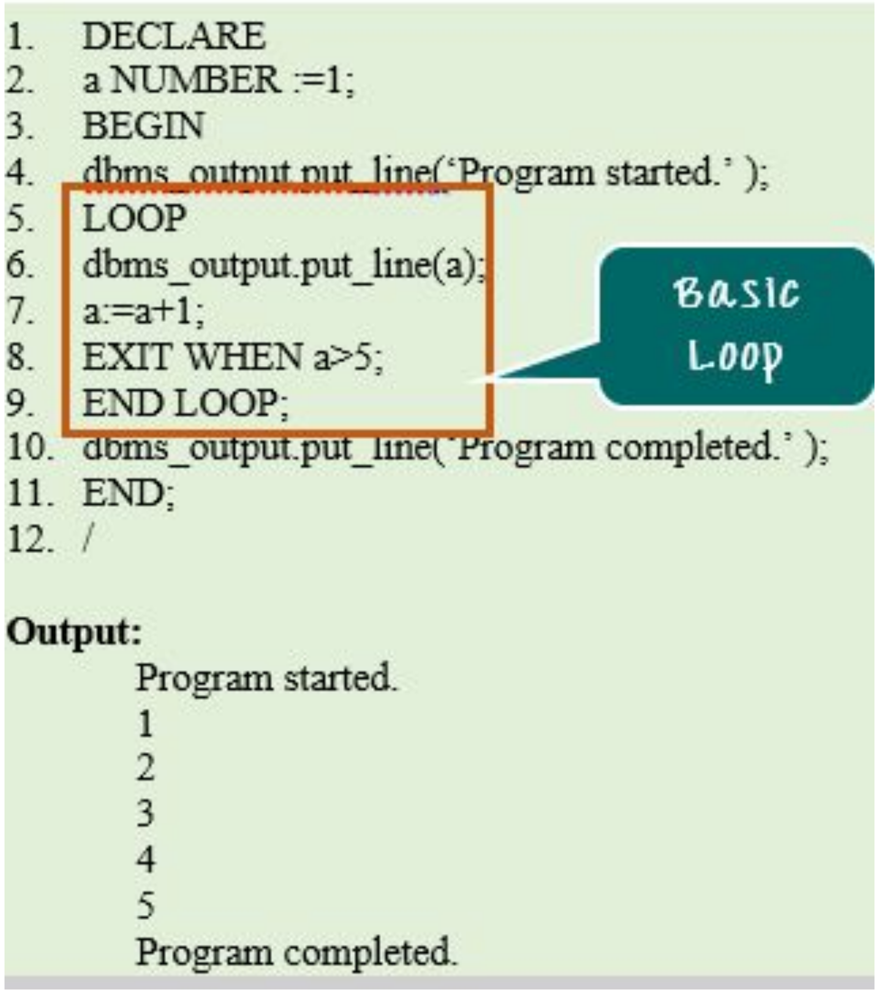
Syntax:

```
LOOP
    <execution_block_starts>
    .
    .
    .
    <EXIT condition based on developer criteria>
    <execution_block_ends>
END LOOP;
```

Exit condition that bring control out of loop

- Use an EXIT statement to stop looping and prevent an infinite loop.

Basic loop statement



```
DECLARE
a NUMBER :=1;
BEGIN
dbms_output.put_line('Program started.' );
LOOP
dbms_output.put_line(a);
a:=a+1;
EXIT WHEN a>5;
END LOOP;
dbms_output.put_line('Program completed.' );
END;
/
```

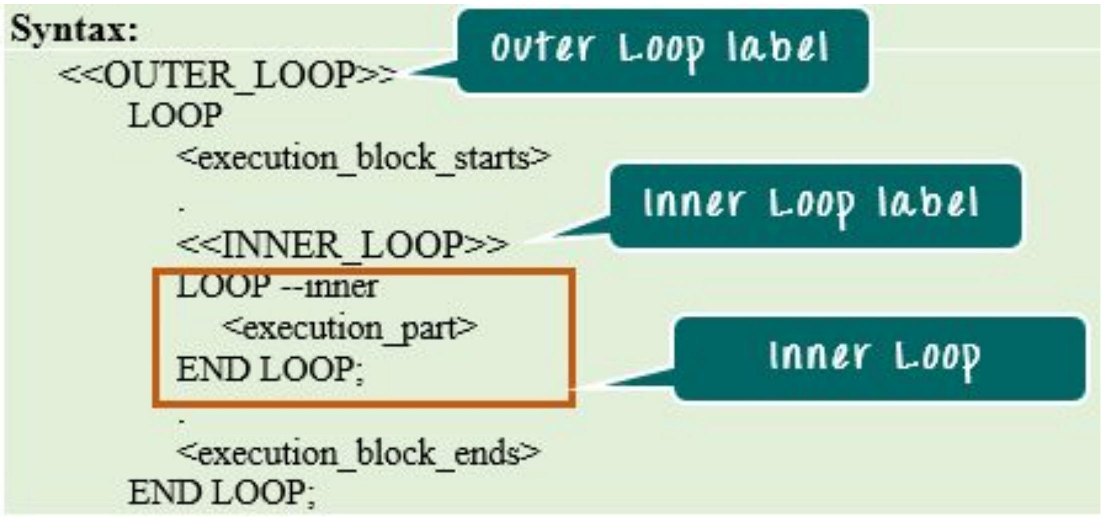
Output:

```
    Program started.
    1
    2
    3
    4
    5
    Program completed.
```

– There are two form of EXIT statements:
  - » EXIT and EXIT-WHEN.

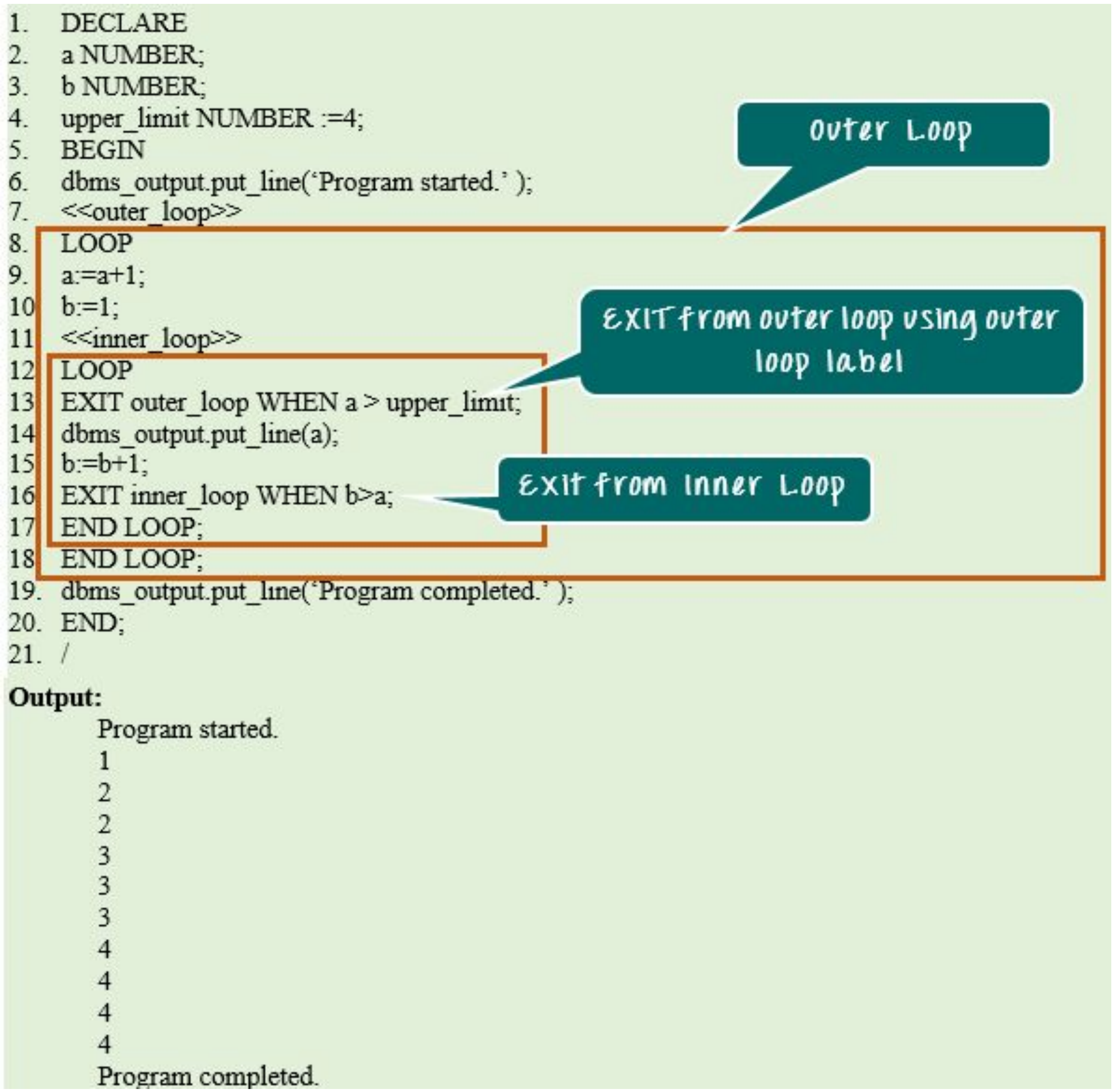
```
IF a> 5THEN
EXIT;
ENDIF;

EXIT WHEN a> 5;
```

Nested loop and labeling of loops



Syntax:

```
<<OUTER_LOOP>>
    LOOP
        <execution_block_starts>
        .
        <<INNER_LOOP>>
        LOOP --inner
            <execution_part>
        END LOOP;
        .
        <execution_block_ends>
    END LOOP;
```



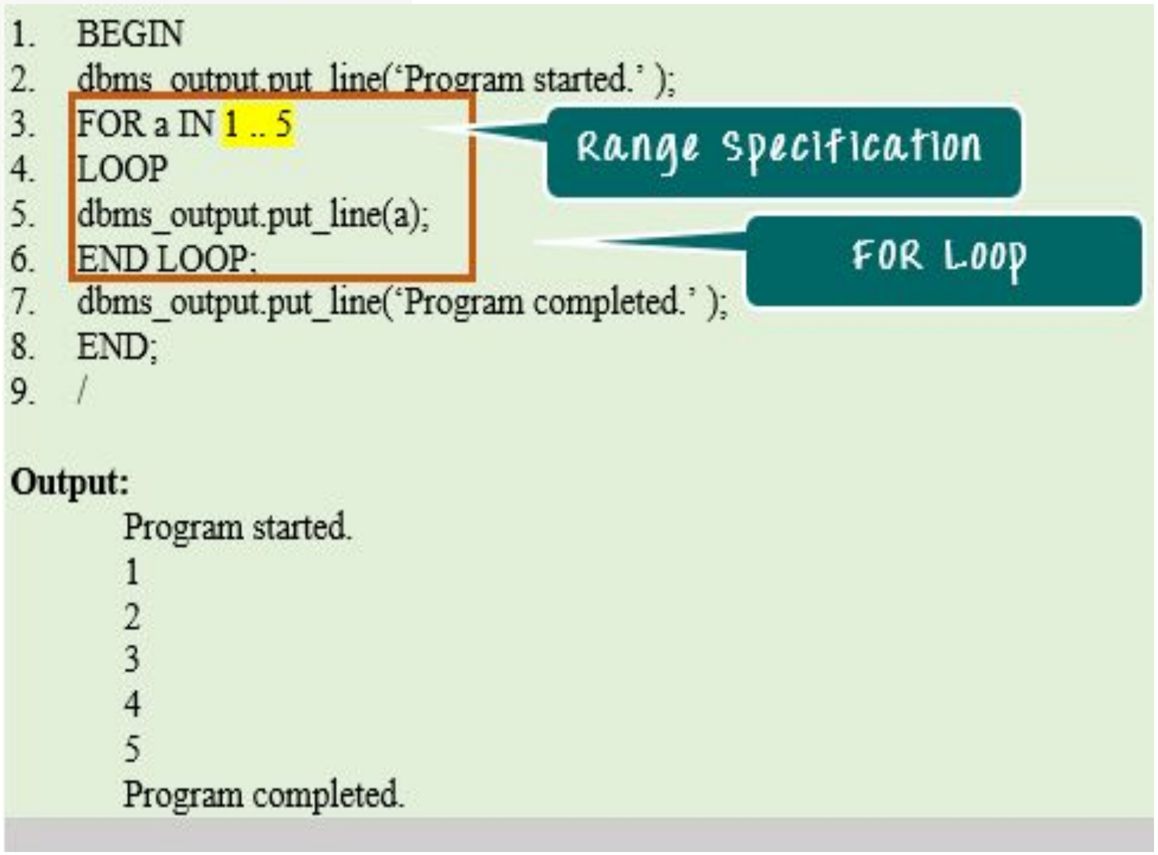
```
DECLARE
a NUMBER;
b NUMBER;
upper_limit NUMBER :=4;
BEGIN
dbms_output.put_line('Program started.' );
<<outer_loop>>
LOOP
a:=a+1;
b:=1;
<<inner_loop>>
LOOP
EXIT outer_loop WHEN a > upper_limit;
dbms_output.put_line(a);
b:=b+1;
EXIT inner_loop WHEN b>a;
END LOOP;
END LOOP;
dbms_output.put_line('Program completed.' );
END;
/
```

Output:

```
Program started.
1
2
2
3
3
3
4
4
4
4
Program completed.
```

For Loop

```
FOR <loop_variable> in <lower_limit> .. <higher_limit>
LOOP
<execution block starts>
.
.
.
<execution_block_ends>
 END LOOP;
```

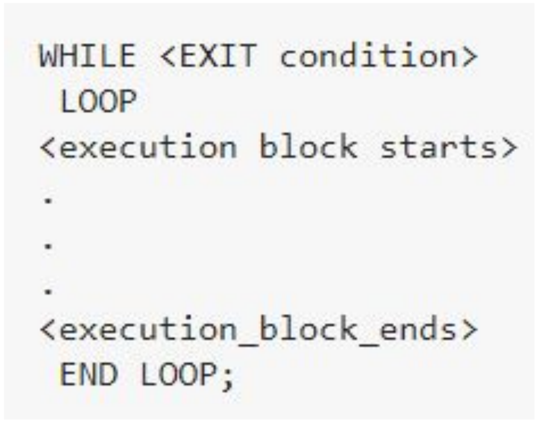


```
BEGIN
dbms_output.put_line('Program started.' );
FOR a IN 1 .. 5
LOOP
dbms_output.put_line(a);
END LOOP;
dbms_output.put_line('Program completed.' );
END;
/
```

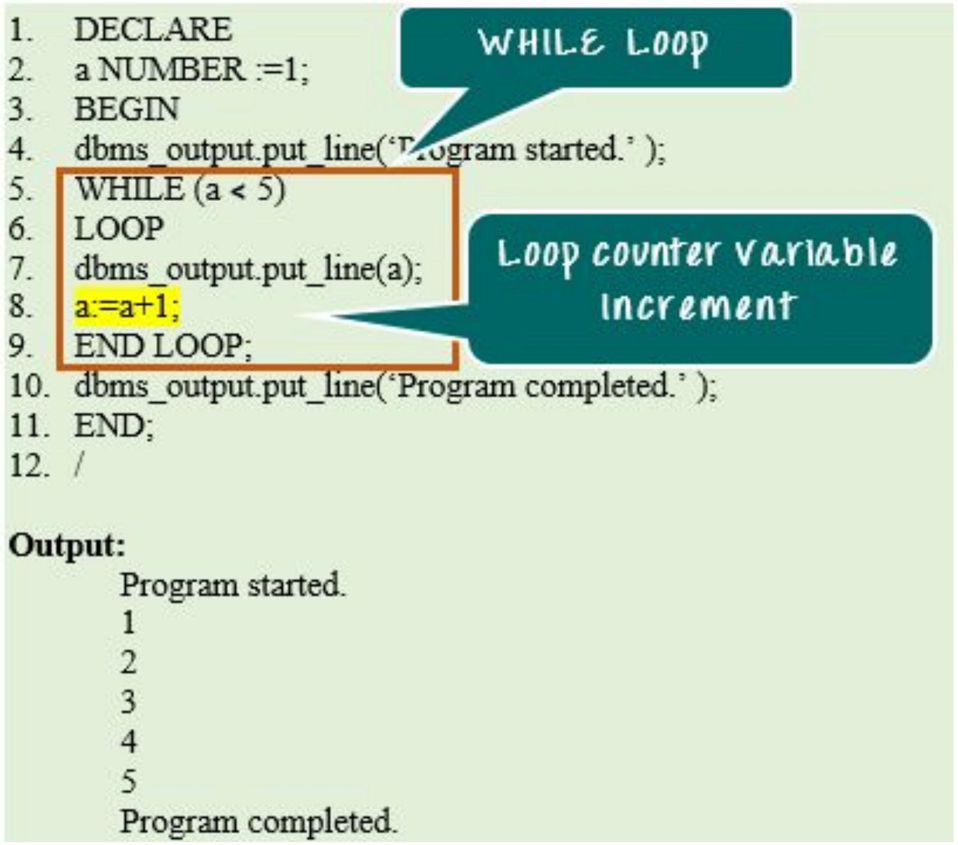
Output:

```
    Program started.
    1
    2
    3
    4
    5
    Program completed.
```

While Loop



```
WHILE <EXIT condition>
 LOOP
<execution block starts>
.
.
.
<execution_block_ends>
 END LOOP;
```

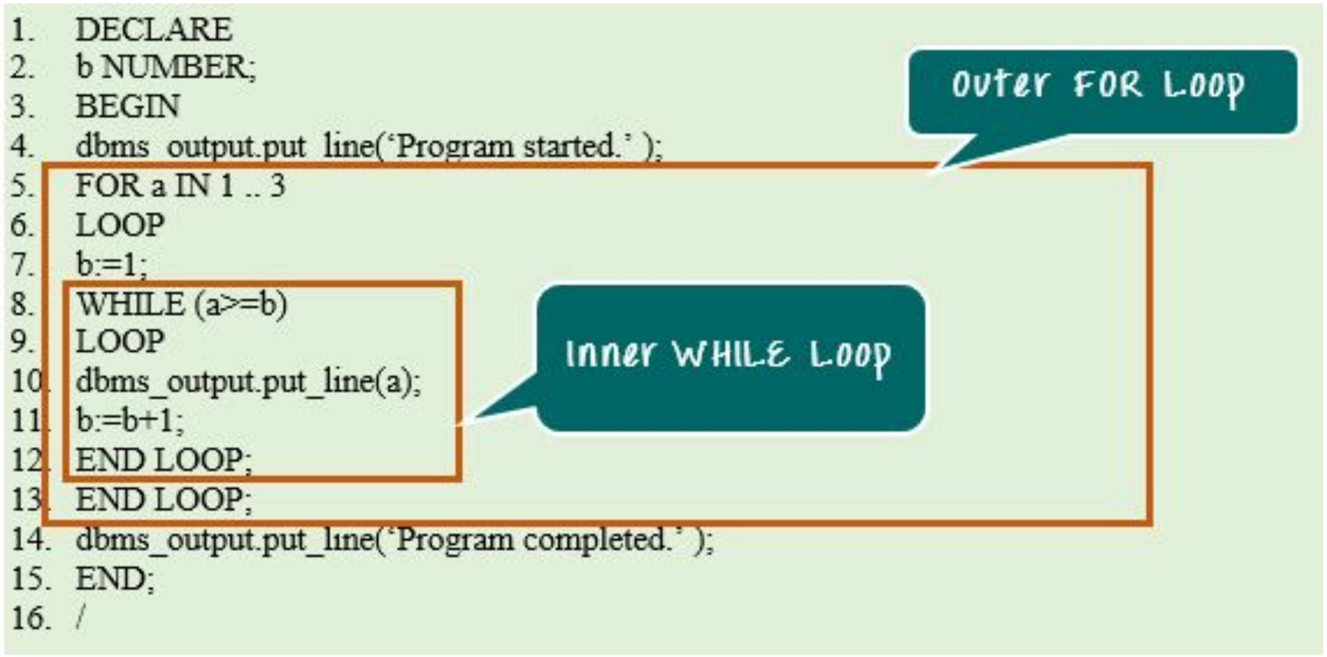


```
DECLARE
a NUMBER :=1;
BEGIN
dbms_output.put_line('Program started.' );
WHILE (a < 5)
LOOP
dbms_output.put_line(a);
a:=a+1;
END LOOP;
dbms_output.put_line('Program completed.' );
END;
/
```

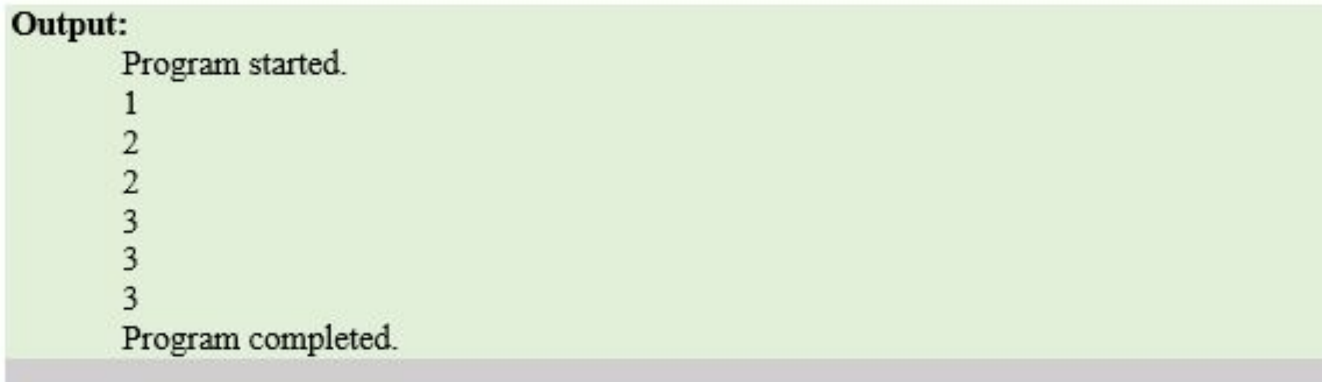
Output:

```
Program started.
1
2
3
4
5
Program completed.
```

Nesting of For and While Loop



```
DECLARE
b NUMBER;
BEGIN
dbms_output.put_line(‘Program started.’ );
FOR a IN 1 .. 3
LOOP
b:=1;
WHILE (a>=b)
LOOP
dbms_output.put_line(a);
b:=b+1;
END LOOP;
END LOOP;
dbms_output.put_line(‘Program completed.’ );
END;
/
```

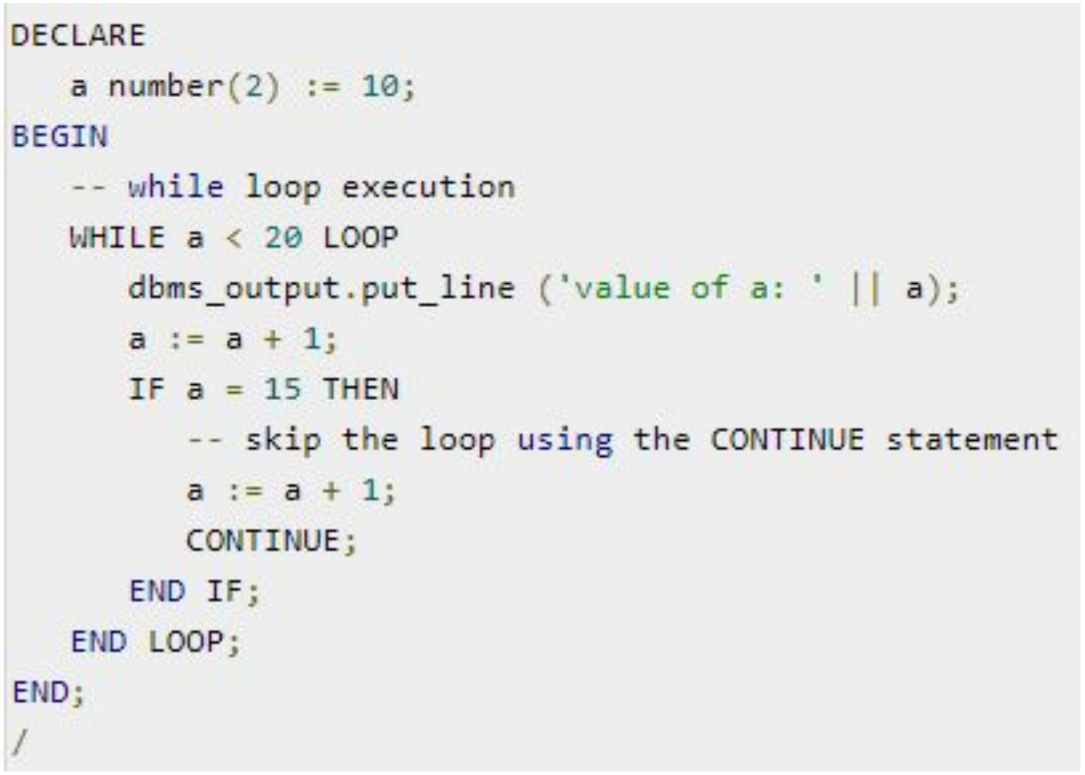


Output:

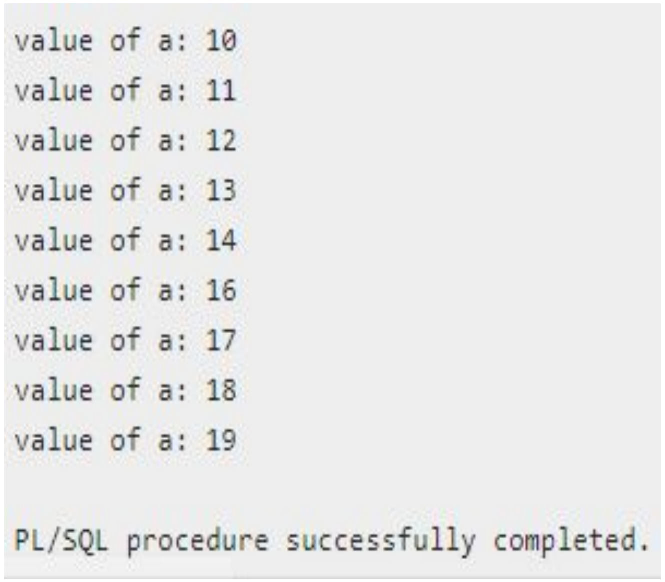
```
Program started.
1
2
2
3
3
3
Program completed.
```

CONTINUE statement

- The **CONTINUE** statement causes the loop to skip one iteration of loop based on a condition.



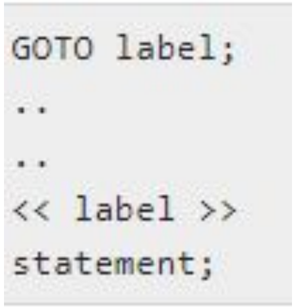
```
DECLARE
   a number(2) := 10;
BEGIN
   -- while loop execution
   WHILE a < 20 LOOP
      dbms_output.put_line ('value of a: ' || a);
      a := a + 1;
      IF a = 15 THEN
         -- skip the loop using the CONTINUE statement
         a := a + 1;
         CONTINUE;
      END IF;
   END LOOP;
END;
/
```



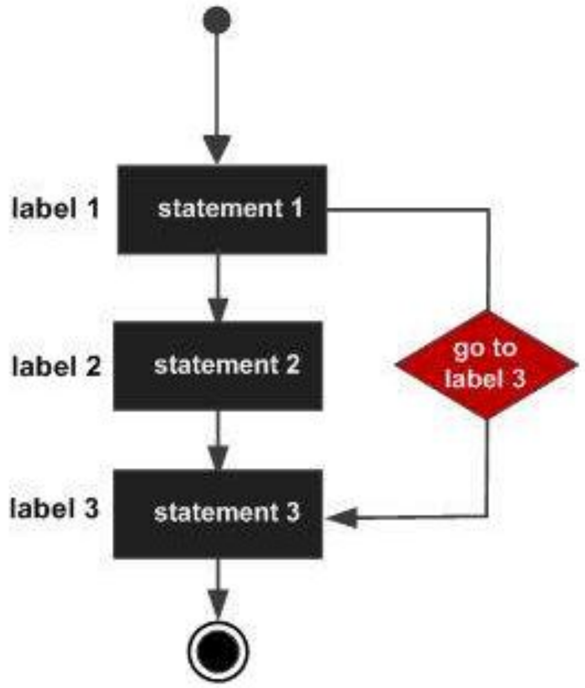
```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19

PL/SQL procedure successfully completed.
```

GOTO statement

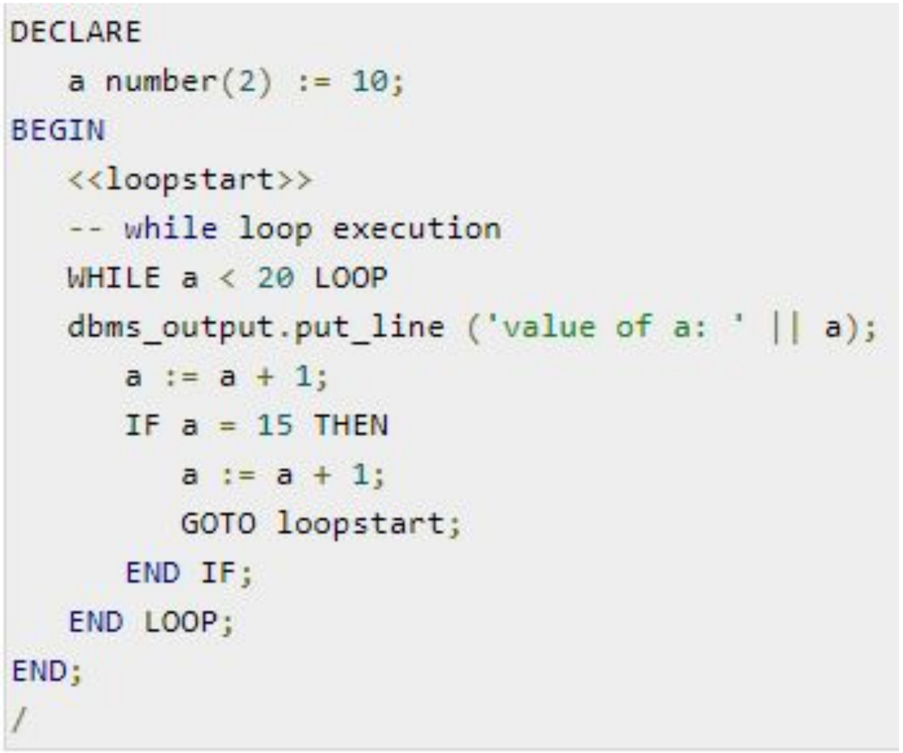
- A **GOTO** statement in PL/SQL programming language provides an unconditional jump from the GOTO to a labeled statement in the same subprogram.



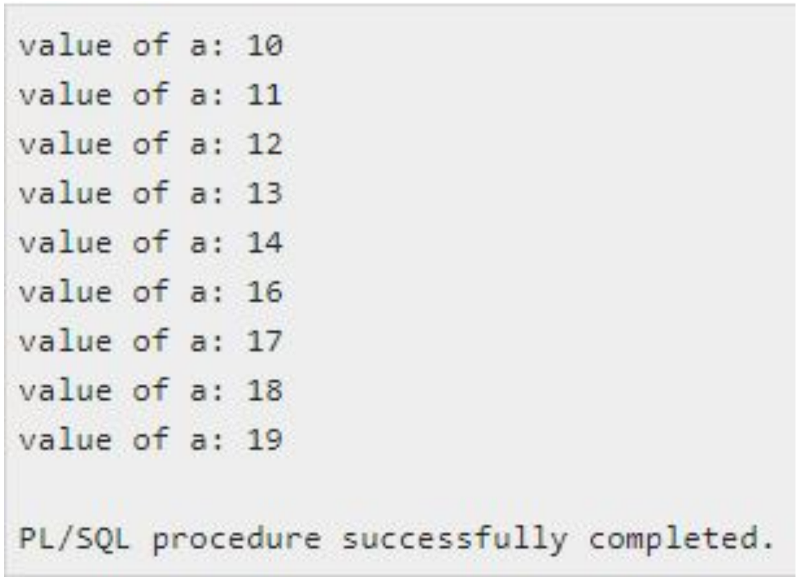
```
GOTO label;
..
..
<< label >>
statement;
```



Example of GOTO

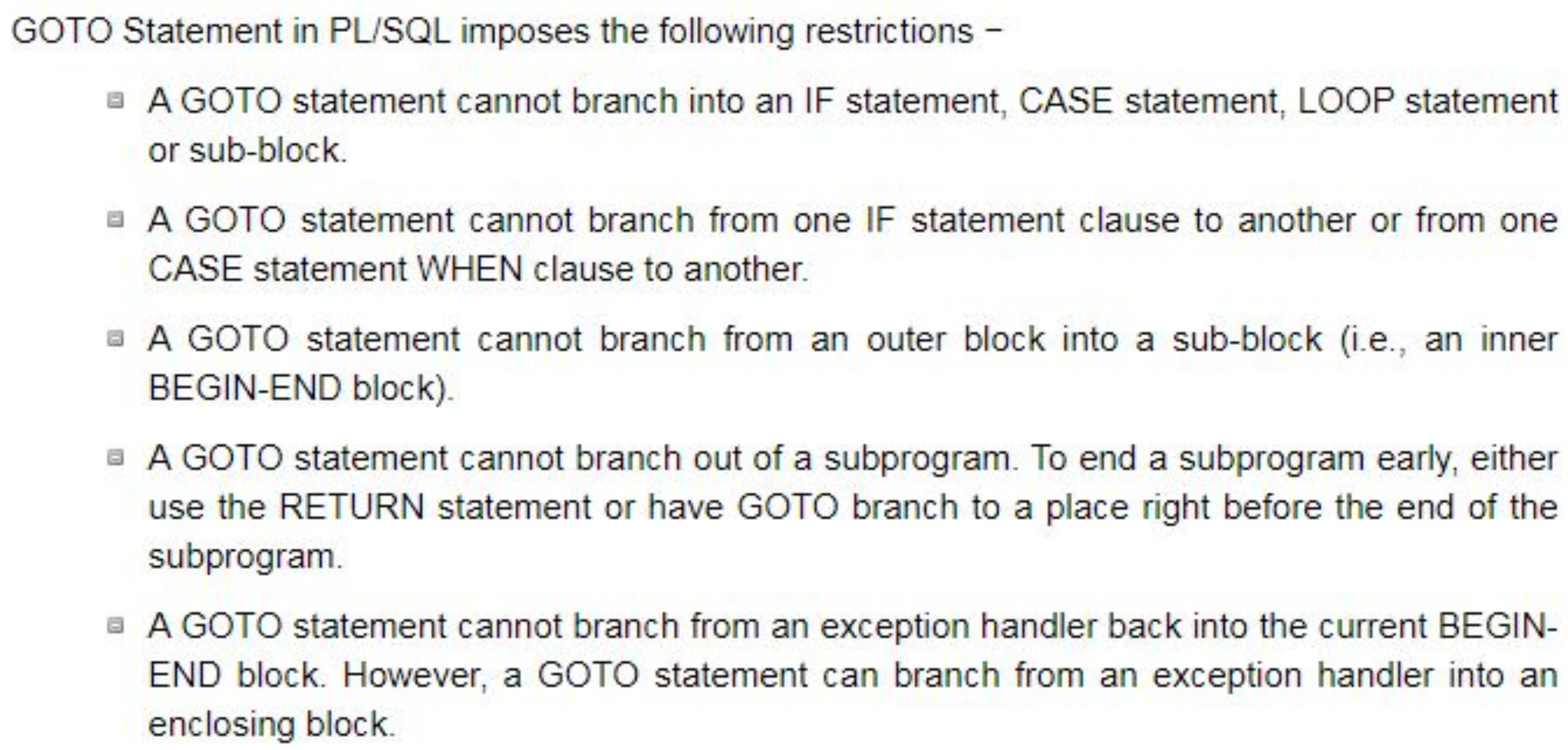


```
DECLARE
   a number(2) := 10;
BEGIN
   <<loopstart>>
   -- while loop execution
   WHILE a < 20 LOOP
   dbms_output.put_line ('value of a: ' || a);
      a := a + 1;
      IF a = 15 THEN
         a := a + 1;
         GOTO loopstart;
      END IF;
   END LOOP;
END;
/
```



```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19

PL/SQL procedure successfully completed.
```

Restrictions with GOTO Statement

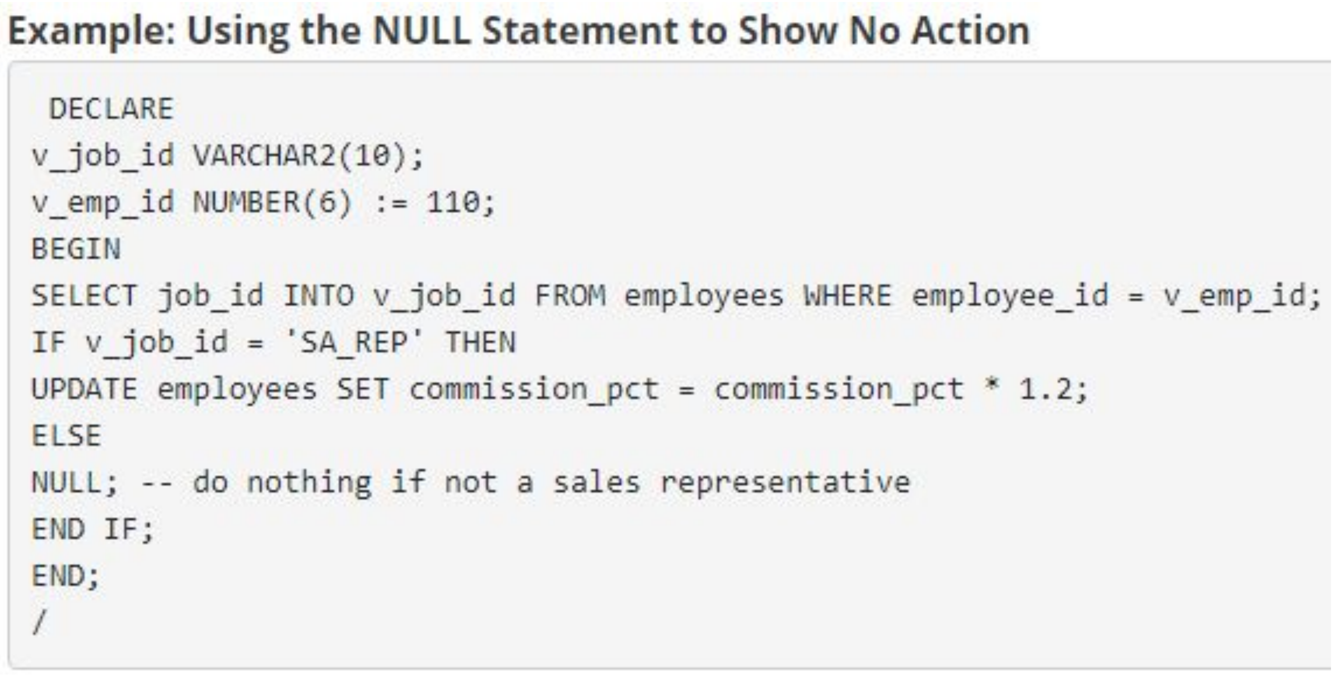


GOTO Statement in PL/SQL imposes the following restrictions −

- A GOTO statement cannot branch into an IF statement, CASE statement, LOOP statement or sub-block.
- A GOTO statement cannot branch from one IF statement clause to another or from one CASE statement WHEN clause to another.
- A GOTO statement cannot branch from an outer block into a sub-block (i.e., an inner BEGIN-END block).
- A GOTO statement cannot branch out of a subprogram. To end a subprogram early, either use the RETURN statement or have GOTO branch to a place right before the end of the subprogram.
- A GOTO statement cannot branch from an exception handler back into the current BEGIN-END block. However, a GOTO statement can branch from an exception handler into an enclosing block.

Using the NULL Statement

- The NULL statement does nothing and passes control to the next statement. Some languages refer to such instruction as a no-op (no operation).



Example: Using the NULL Statement to Show No Action

```
 DECLARE
v_job_id VARCHAR2(10);
v_emp_id NUMBER(6) := 110;
BEGIN
SELECT job_id INTO v_job_id FROM employees WHERE employee_id = v_emp_id;
IF v_job_id = 'SA_REP' THEN
UPDATE employees SET commission_pct = commission_pct * 1.2;
ELSE
NULL; -- do nothing if not a sales representative
END IF;
END;
/
```

Example: use of Null

```
declare
did varchar(20);
eid1 varchar(20):='e1’;
begin
    select dno into did from emp where eid=eid1;
    if did= 'd1' then
        update emp set ecommision=ecommision*1.2;
    else
        null;
    end if;
end;
```

PL/SQL %TYPE Attribute

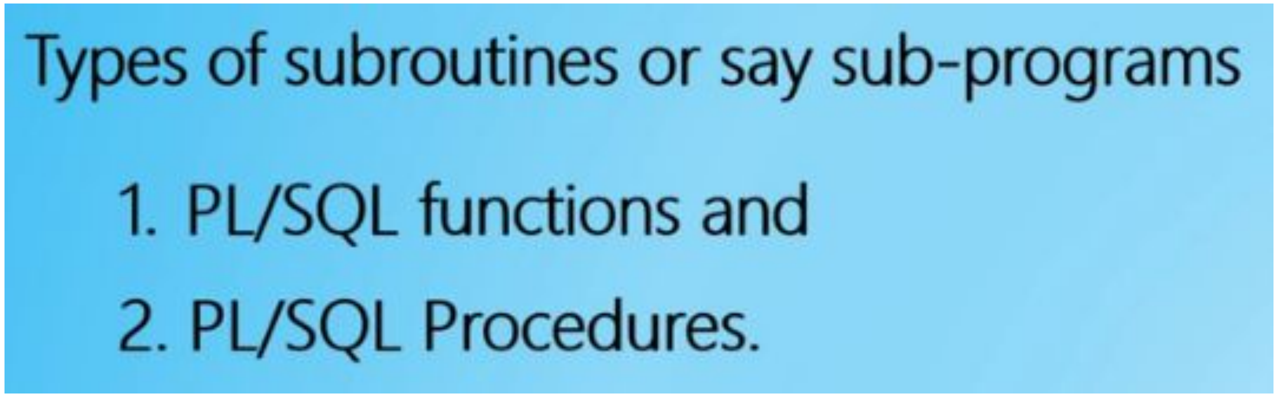
- The %TYPE attribute allow you to declare a constant, variable, or parameter to be of the same data type as previously declared variable, record, nested table, or database column.
- Syntax:

```
identifier Table.column_name%TYPE;
```

```
declare
	v_name employee.lastname%TYPE;
	v_dep  number;
	v_min_dep  v_dep%TYPE:=31;
begin
	select lastname into v_name from EMPLOYEE where DEPARTMENTID=v_min_dep;
	DBMS_OUTPUT.PUT_LINE('v_name: '||v_name);
end;
```

```
DECLARE x NUMBER := 100;
BEGIN
EXECUTE IMMEDIATE 'create table my_table (n number)';
--Second, use DBMS_UTLIITY.EXEC_DDL_STATEMENT:
--DBMS_UTILITY.EXEC_DDL_STATEMENT ( 'create table my_table (n number)');
FOR i IN 1..10
 LOOP IF MOD(i,2) = 0 THEN
 -- i is even
INSERT INTO temp VALUES (i, x, 'i is even');
ELSE
 INSERT INTO temp VALUES (i, x, 'i is odd');
 END IF;
 x := x + 100;
END LOOP;
COMMIT;
END;
```

Sub-programs in PL/SQL

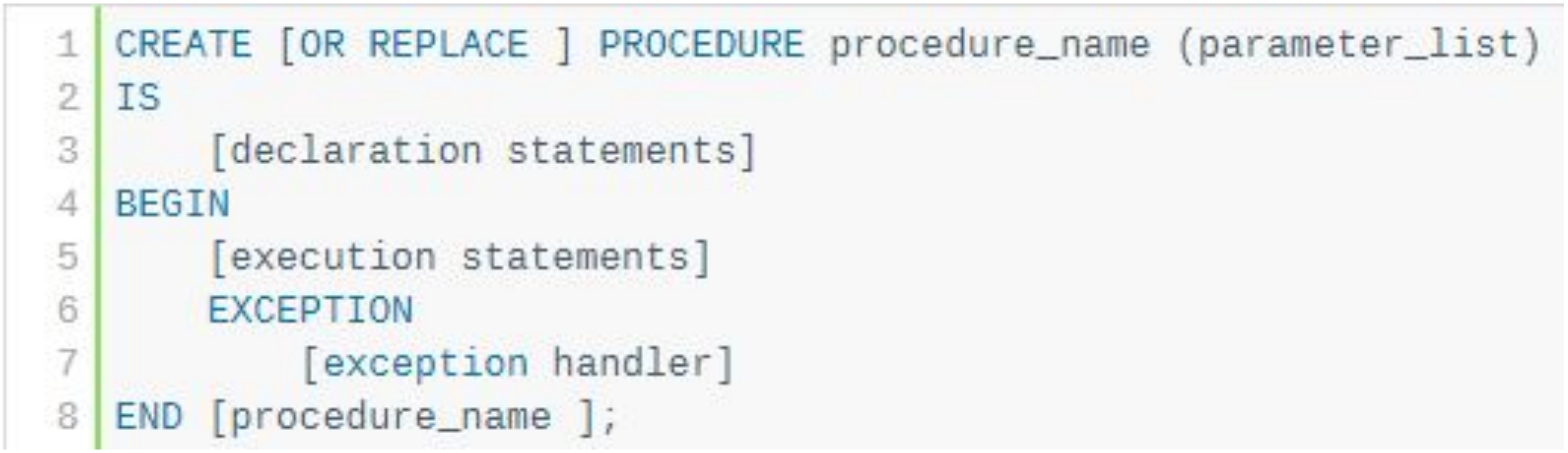


Types of subroutines or say sub-programs

1. PL/SQL functions and
2. PL/SQL Procedures.

PL/SQL Procedures

- A PL/SQL procedure is a named block stored
  - It is a reusable unit
- The basic syntax of creating a procedure in PL/SQL is as follows:



```
CREATE [OR REPLACE ] PROCEDURE procedure_name (parameter_list)
IS
    [declaration statements]
BEGIN
    [execution statements]
    EXCEPTION
        [exception handler]
END [procedure_name ];
```

- Note that OR REPLACE option allows you to overwrite the current procedure with the new code.

PL/SQL Procedures

- PL/SQL procedures have to parts
  - PL/SQL procedure header
    - It specifies procedure name and an optional parameter list.
  - PL/SQL procedure body
    - **Declarative part**
    - **Executable part**
    - **Exception-handling part**

PL/SQL Procedure Header

- A procedure begins with a header that specifies its name and an optional parameter list.
- Parameters can in one of the following modes:
  - In mode
  - Out mode
  - Inout mode

PL/SQL Procedure Header- In Mode

- An IN parameter is read-only.
- You can reference an IN parameter inside a procedure, but you cannot change its value.
- Oracle uses IN as the default mode.
  - It means that if you don’t specify the mode for a parameter explicitly, Oracle will use the IN mode.

PL/SQL Procedure Header- Out Mode

- An OUT parameter is writable.
- Typically, you set a returned value for the OUT parameter and return it to the calling program.
- Note that a procedure ignores the value that you supply for an OUT parameter.

PL/SQL Procedure Header- Inout Mode

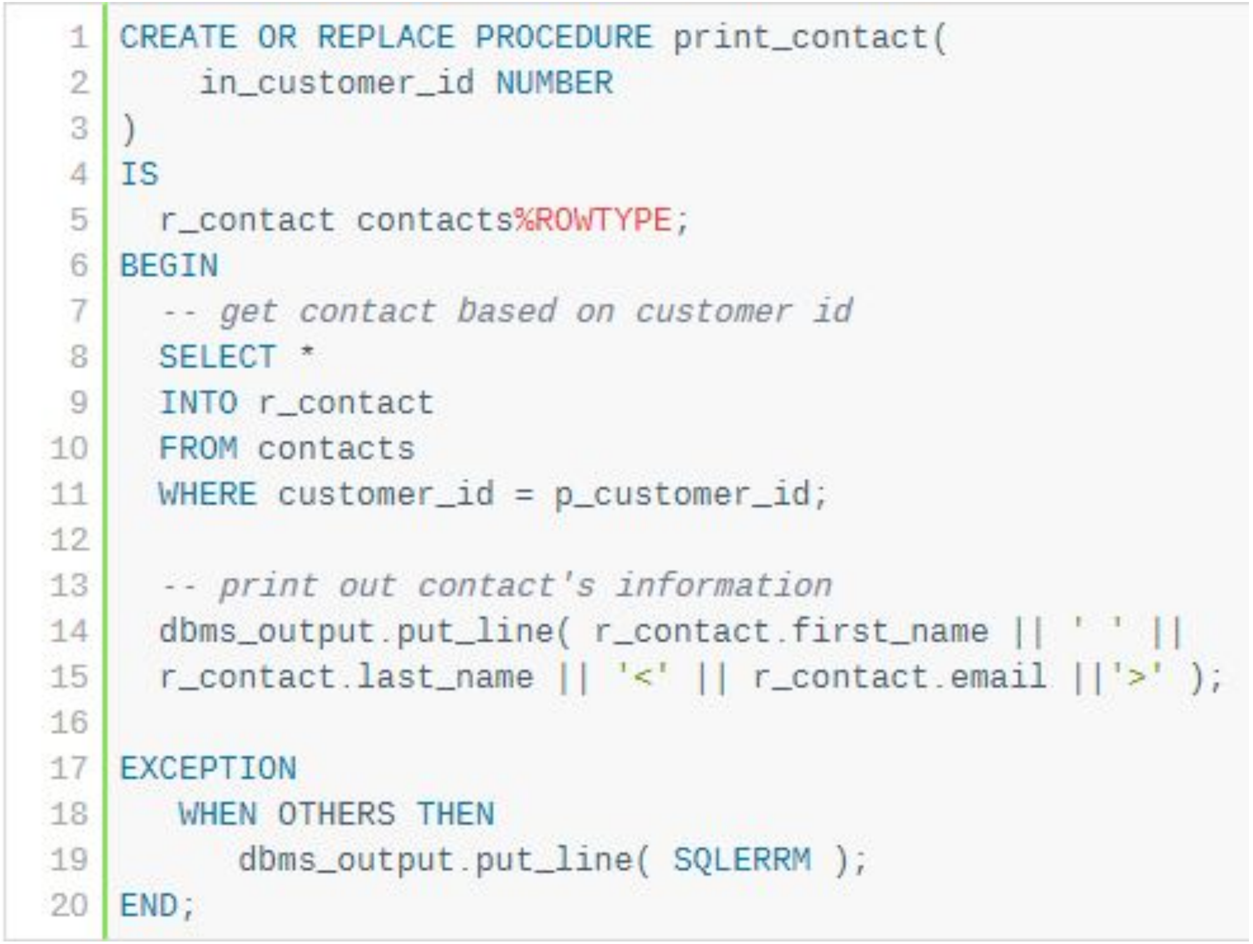
- An INOUT parameter is both readable and writable.
  - The procedure can read and modify it.

%ROWTYPE Attribute

- The %ROWTYPE attribute provides a record type that represents a row in a database table.
- The record can store an entire row of data selected from the table or fetched from a cursor or cursor variable.
- Fields in a record and corresponding columns in a row have the same names and datatypes.
- You can use the %ROWTYPE attribute in variable declarations as a datatype specifier.
- Variables declared using %ROWTYPE are treated like those declared using a datatype name.

Creating a PL/SQL procedure example

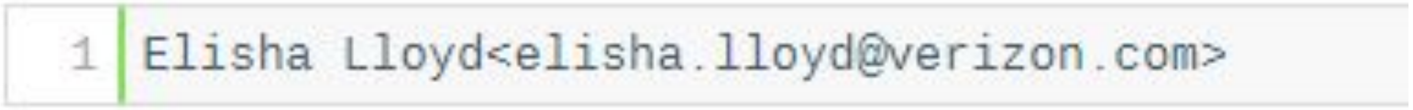
- The following procedure accepts a customer id and prints out the customer’s contact information including first name, last name, and email:



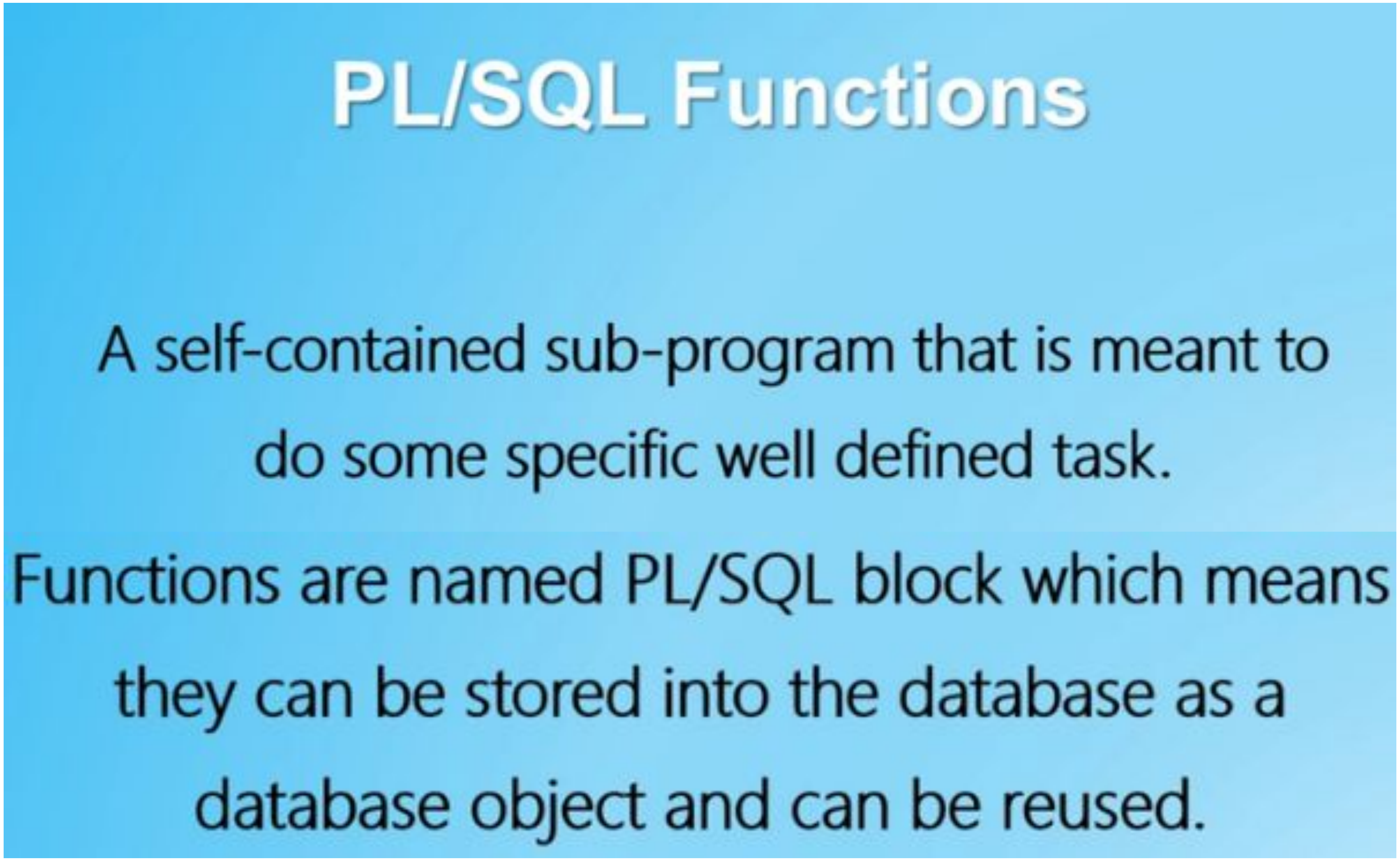
```sql
CREATE OR REPLACE PROCEDURE print_contact(
    in_customer_id NUMBER
)
IS
  r_contact contacts%ROWTYPE;
BEGIN
  -- get contact based on customer id
  SELECT *
  INTO r_contact
  FROM contacts
  WHERE customer_id = p_customer_id;

  -- print out contact's information
  dbms_output.put_line( r_contact.first_name || ' ' ||
  r_contact.last_name || '<' || r_contact.email ||'>' );

EXCEPTION
   WHEN OTHERS THEN
      dbms_output.put_line( SQLERRM );
END;
```

PL/SQL Procedures

- Use / to compile procedures
- Using the **EXECUTE/EXEC** keyword with procedure name to execute procedure
  - EXECUTE procedure_name(arguments);
  - EXEC procedure_name(arguments);
- EXEC print_contact(100);



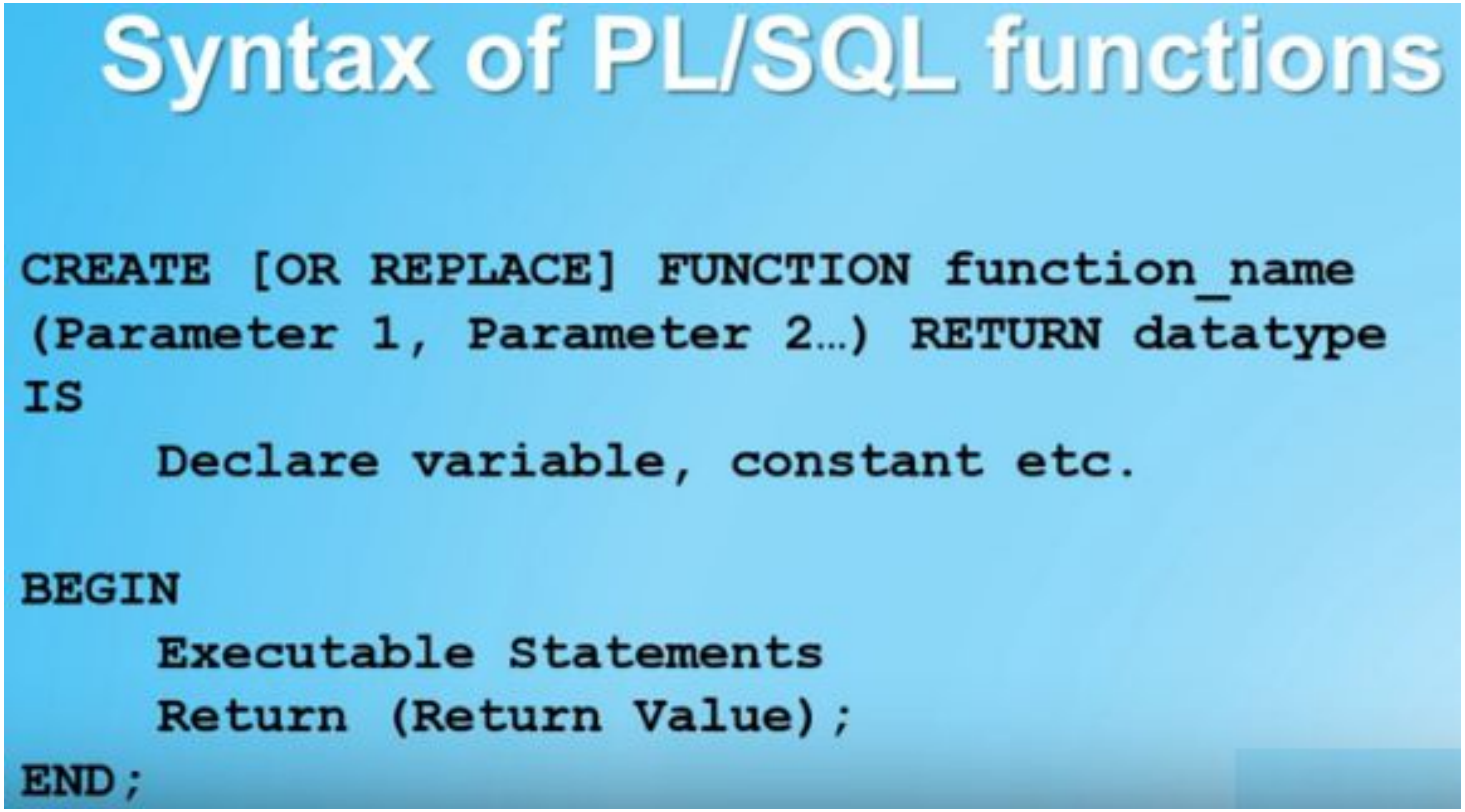
```
Elisha Lloyd<elisha.lloyd@verizon.com>
```



PL/SQL Functions

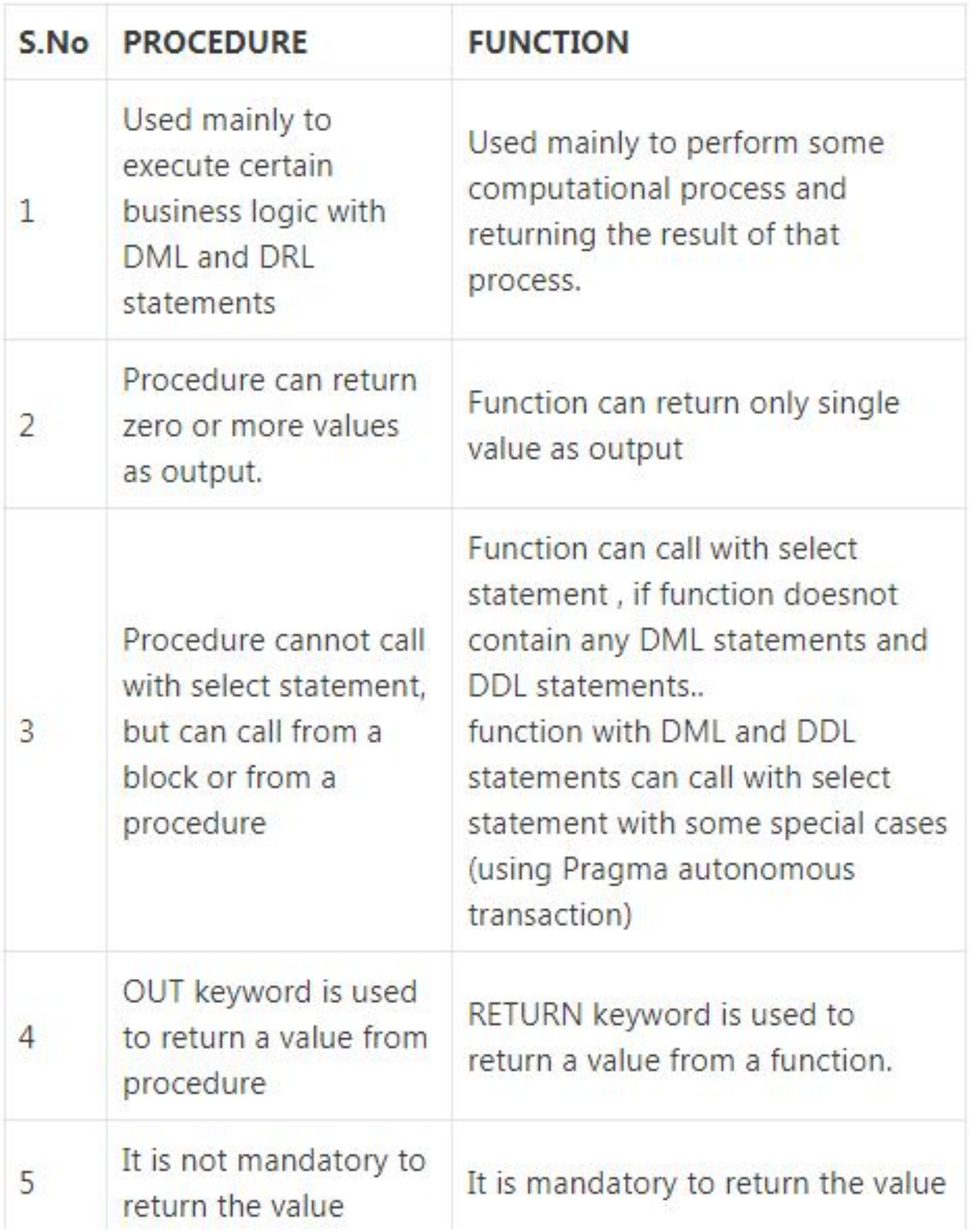
A self-contained sub-program that is meant to do some specific well defined task.

Functions are named PL/SQL block which means they can be stored into the database as a database object and can be reused.



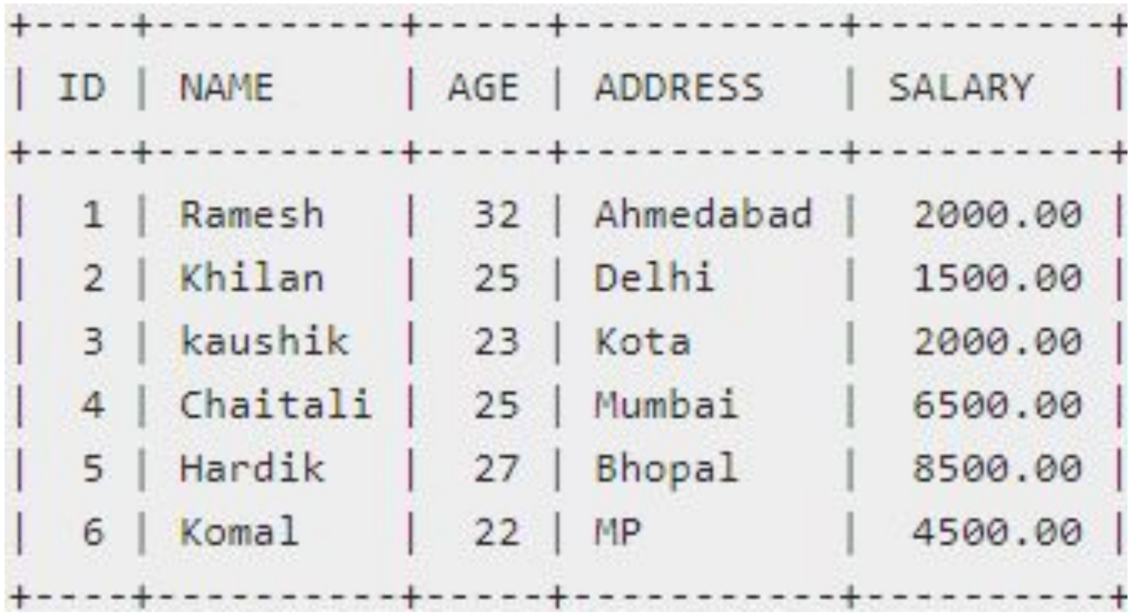
Syntax of PL/SQL functions

```
CREATE [OR REPLACE] FUNCTION function_name
(Parameter 1, Parameter 2…) RETURN datatype
IS
    Declare variable, constant etc.

BEGIN
    Executable Statements
    Return (Return Value);
END;
```



S.No	PROCEDURE	FUNCTION
1	Used mainly to execute certain business logic with DML and DRL statements	Used mainly to perform some computational process and returning the result of that process.
2	Procedure can return zero or more values as output.	Function can return only single value as output
3	Procedure cannot call with select statement, but can call from a block or from a procedure	Function can call with select statement , if function doesnot contain any DML statements and DDL statements.. function with DML and DDL statements can call with select statement with some special cases (using Pragma autonomous transaction)
4	OUT keyword is used to return a value from procedure	RETURN keyword is used to return a value from a function.
5	It is not mandatory to return the value	It is mandatory to return the value

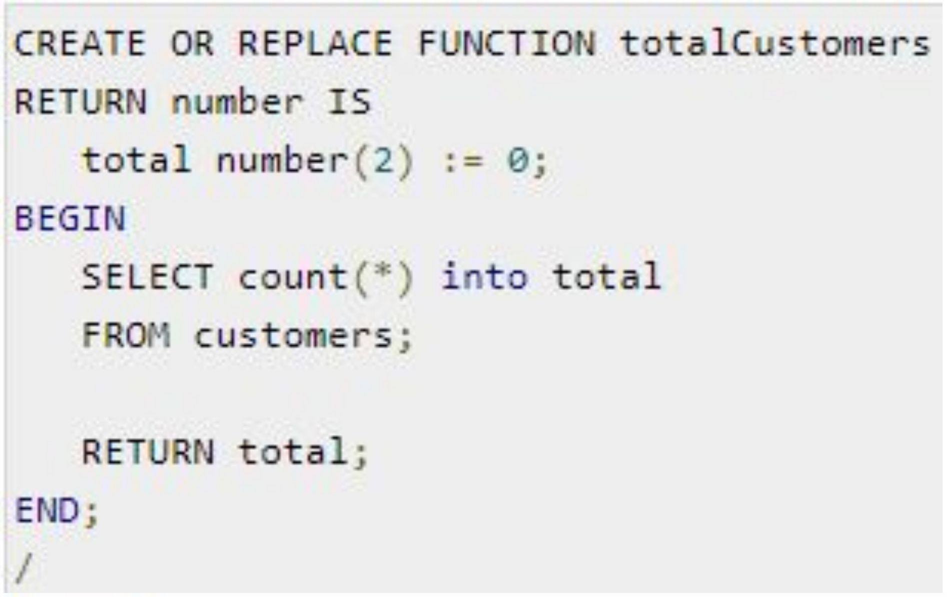
Customer table



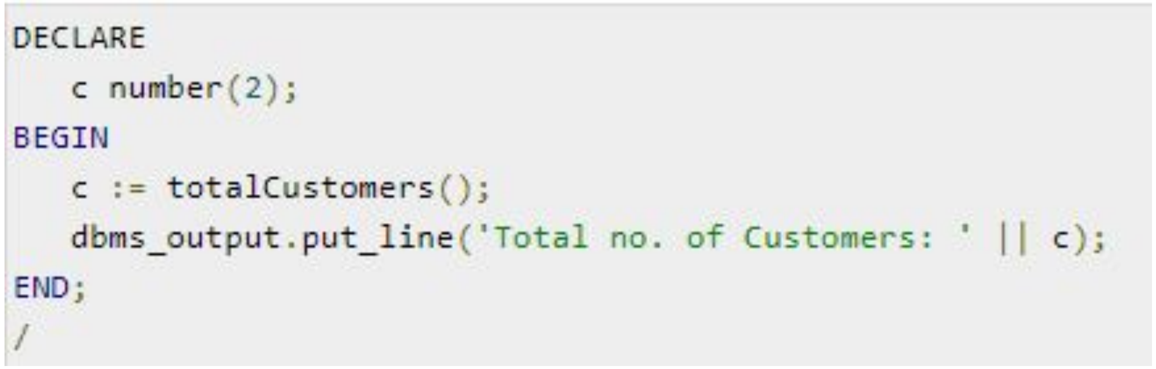
```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali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

- Write a function to count total number customers in the customer table.

PL/SQL Function

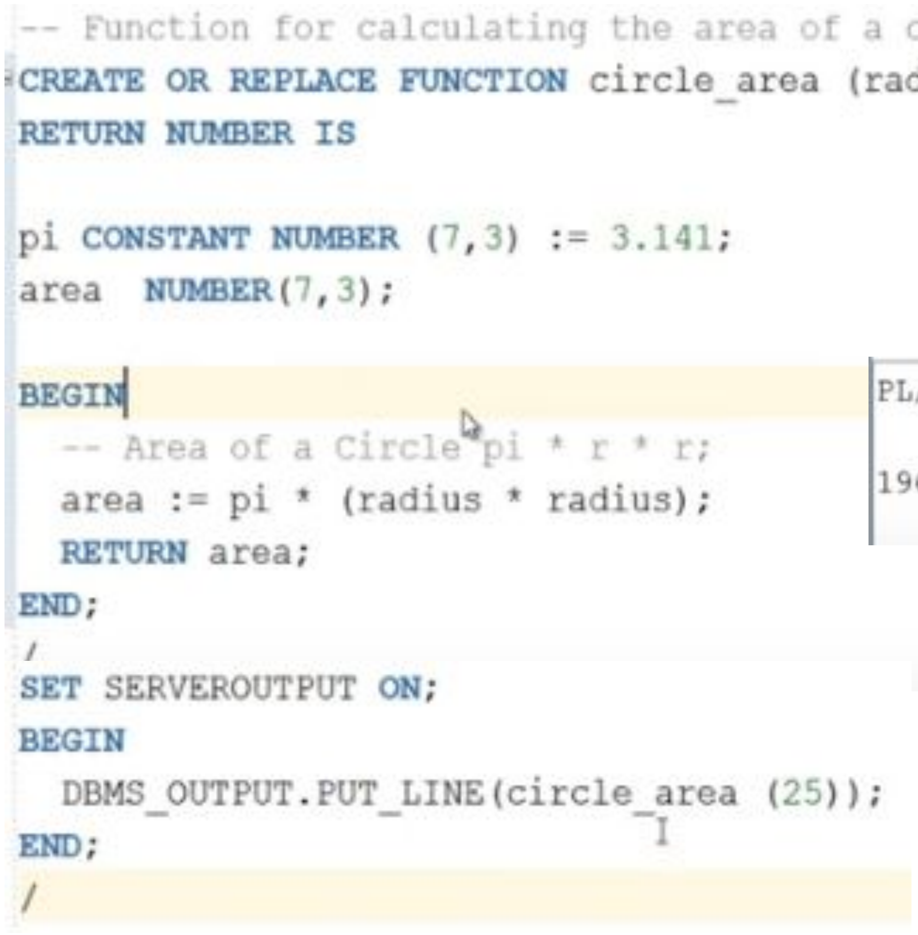


```
CREATE OR REPLACE FUNCTION totalCustomers
RETURN number IS
   total number(2) := 0;
BEGIN
   SELECT count(*) into total
   FROM customers;

   RETURN total;
END;
/
```

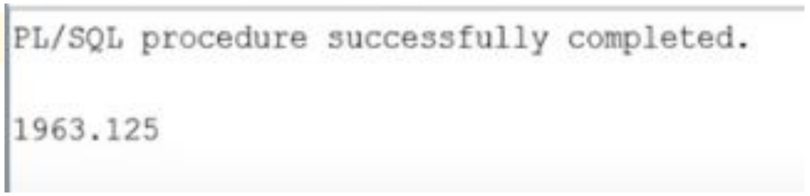


```
DECLARE
   c number(2);
BEGIN
   c := totalCustomers();
   dbms_output.put_line('Total no. of Customers: ' || c);
END;
/
```

PL/SQL Functions: Area of Circle



```
-- Function for calculating the area of a 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 of a Circle pi * r * r;
  area := pi * (radius * radius);
  RETURN area;
END;
/
SET SERVEROUTPUT ON;
BEGIN
  DBMS_OUTPUT.PUT_LINE(circle_area (25));
END;
/
```



```
PL/SQL procedure successfully completed.

1963.125
```

Cursor

- A cursor holds multiple rows returned by a SQL statement.
- Type of Cursor
  1. Implicit cursor (Generated by Oracle)
  2. Explicit cursor (Created by User)
- For steps to use a cursor
  I. Declare
  II. Open
  III. Fetch
  IV. Close

Syntax:

Cursor C1 IS select statement

Open C1

Fetch C1 INTO …. (some variables)

Close C1

Cursor Implementaion

```
Declare
      c_id customers.cid%type;
      c_name customers.cname%type;
      cursor C1 IS select cid, cname from customers;
Begin
      Open C1;
      loop
      fetch c1 into c_id, c_name;
      exit when c1%notfound;
      DBMS_OUTPUT.PUT_LINE(C_ID || ','|| C_NAME);
      end loop;
      close C1;
End;
```

Trigger

- Stored Program which are automatically fired or executed when some event occur such as DML or DDL or DB operation.

Syntax: Trigger

```
Create or Replace Trigger trigger-name
Before or After or  Instead of Insert or update or Delete
Of colName (optional)
On tableName
Referencing old as O new as N (optional)
For each row
When (condition)
Declare
        ------
Begin
        ------
Exception
        ------
End;
```

Salary Difference Trigger

```
Create table employee (eid number, ename varchar(20), ordesalary number)
Create or Replace Trigger display_salary_change
Before Insert or update or Delete
On employee
For each row
When (new.id>0)
Declare
      salary_difference number;
Begin
      salary_difference:=new.salary-old.salary;
      dbms_output.put_line(‘salary difference is: ‘ || salary_difference)
End;
```

Ordervalue Difference Trigger

```
create table customers (cid number, cname varchar(20));
insert into customers values (1, 'AA’);
insert into customers values (2, 'AB’);
insert into customers values (3, 'AC’);
insert into customers values (4, 'AD’);
insert into customers values (5, 'AE’);

Create or Replace Trigger display_ordervalue_change
Before Insert or update or Delete
On customers
For each row
When (new.cid >0)
Declare
ordervalue_difference number;Begin
ordervalue_difference:=:new.ORDERVALUE - :old.ORDERVALUE;
dbms_output.put_line('ordervalue difference is: ' || ordervalue_difference);
End;
```

Student total and percent update trigger

```
create table student(stid int not null unique, sname varchar(30), sub1 int, sub2 int,
sub3 int, total int, percentage int)

create or replace trigger std_marks
before insert
on student
for each row
begin
update student set student.total=student.sub1+student.sub2+student.sub3;
update student set student.percentage=student.total*60/100;
end;

insert into student (sid,sname,sub1,sub2,sub3) values (1,'aa', 55,55,55);
insert into student (sid,sname,sub1,sub2,sub3) values (2,'aa', 55,55,55);
```

Student total and percent update trigger

```
create or replace trigger STOTAL
before insert
on student
for each row
when (new.sid>0)
begin
:new.total:=:new.sub1+:new.sub2;
:new.percentage:=:new.total*60/100;
end;

insert into student (sid,sname,sub1,sub2)
values(1000,'aaa',25,25)
select * from student
```

Display Total Marks trigger

```
create or replace trigger displaySTOTAL
after insert
on student
for each row
when (new.sid>0)
declare sstotal number;
begin  sstotal:= :new.total;
dbms_output.put_line('the total is '||sstotal);
end;
```