- The name of a PL/SQL variable consists of a letter optionally followed by more letters, numerals, dollar signs, underscores, and number signs and should not exceed 30 characters.

- By default, variable names are not case-sensitive.

- You cannot use a reserved PL/SQL keyword as a variable name.

The syntax for declaring a variable is –

variable_name [CONSTANT] datatype [NOT NULL] [:= | DEFAULT initial_value]

```
sales number(10, 2);
pi CONSTANT double precision := 3.1415;
name varchar2(25);
address varchar2(100);
```

```
sales number(10, 2);
name varchar2(25);
address varchar2(100);
```

Initializing Variables in PL/SQL

```
counter binary_integer := 0;
greetings varchar2(20) DEFAULT 'Have a Good Day';
```


```
DECLARE
  a integer := 10;
  b integer := 20;
  c integer;
  f real;
BEGIN
  c := a + b;
  dbms_output.put_line('Value of c: ' || c);
  f := 70.0/3.0;
  dbms_output.put_line('Value of f: ' || f);
END;
/
```

```
DECLARE
  -- Global variables
  num1 number := 95;
  num2 number := 85;
BEGIN
  dbms_output.put_line('Outer Variable num1: ' || num1);
  dbms_output.put_line('Outer Variable num2: ' || num2);
  DECLARE
    -- Local variables
    num1 number := 195;
    num2 number := 185;
  BEGIN
    dbms_output.put_line('Inner Variable num1: ' || num1);
    dbms_output.put_line('Inner Variable num2: ' || num2);
  END;
END;
/
```

Assigning SQL Query Results to PL/SQL Variables

```
CREATE TABLE CUSTOMERS(
  ID   INT NOT NULL,
  NAME VARCHAR (20) NOT NULL,
  AGE INT NOT NULL,
  ADDRESS CHAR (25),
  SALARY   DECIMAL (18, 2),
  PRIMARY KEY (ID)
);
```

Table Created

Let us now insert some values in the table –

```
INSERT INTO CUSTOMERS
(ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, ‘Rahul', 32, 'Ahmedabad', 2000.00 );

INSERT INTO CUSTOMERS
(ID,NAME,AGE,ADDRESS,SALARY)
VALUES (2, ‘ Mohit', 25, 'Delhi', 1500.00 );

INSERT INTO CUSTOMERS
(ID,NAME,AGE,ADDRESS,SALARY)
VALUES (3, ‘Sonali', 23, 'Kota', 2000.00 );
```

The following program assigns values from the above table to PL/SQL variables using the **SELECT INTO clause** of SQL −

```
DECLARE
  c_id customers.id%type := 1;
  c_name  customers.name%type;
  c_addr customers.address%type;
  c_sal  customers.salary%type;
BEGIN
  SELECT name, address, salary INTO c_name, c_addr, c_sal
  FROM customers
  WHERE id = c_id;
  dbms_output.put_line
  ('Customer ' ||c_name || ' from ' || c_addr || ' earns ' || c_sal);
END;
/
```

```
Customer Rahul from Ahmedabad earns 2000
```


CONSTANT IN PL/SQL

```
PI CONSTANT NUMBER := 3.141592654;
DECLARE
  -- constant declaration
  pi constant number := 3.141592654;
  -- other declarations
  radius number(5,2);
  dia number(5,2);
  circumference number(7, 2);
  area number (10, 2);
BEGIN
  -- processing
  radius := 9.5;
  dia := radius * 2;
  circumference := 2.0 * pi * radius;
  area := pi * radius * radius;
  -- output
  dbms_output.put_line('Radius: ' || radius);
  dbms_output.put_line('Diameter: ' || dia);
  dbms_output.put_line('Circumference: ' || circumference);
  dbms_output.put_line('Area: ' || area);
END;
/
```

To embed single quotes within a string literal, place two single quotes next to each other as shown in the following program −

```
DECLARE
  message  varchar2(30):= 'That''s bangontheory.com!';
BEGIN
  dbms_output.put_line(message);
END;
/
```

PL/SQL - Conditions

S.No	Statement & Description
1	**IF - THEN statement** The **IF statement** associates a condition with a sequence of statements enclosed by the keywords **THEN** and **END IF**. If the condition is true, the statements get executed and if the condition is false or NULL then the IF statement does nothing.
2	**IF-THEN-ELSE statement** **IF statement** adds the keyword **ELSE** followed by an alternative sequence of statement. If the condition is false or NULL, then only the alternative sequence of statements get executed. It ensures that either of the sequence of statements is executed.
3	**IF-THEN-ELSIF statement** It allows you to choose between several alternatives.
4	**Case statement** Like the IF statement, the **CASE statement** selects one sequence of statements to execute. However, to select the sequence, the CASE statement uses a selector rather than multiple Boolean expressions. A selector is an expression whose value is used to select one of several alternatives.
5	**Searched CASE statement** The searched CASE statement **has no selector**, and it's WHEN clauses contain search conditions that yield Boolean values.
6	**nested IF-THEN-ELSE** You can use one **IF-THEN** or **IF-THEN-ELSIF** statement inside another **IF-THEN** or **IF-THEN-ELSIF** statement(s).

Syntax

Syntax for IF-THEN statement is –

```
IF condition THEN
  S;
END IF;

IF (a <= 20) THEN
  c:= c+1;
END IF;
```

Syntax

Syntax for the IF-THEN-ELSE statement is –

```
IF condition THEN
  S1;
ELSE
  S2;
END IF;

IF color = red THEN
 dbms_output.put_line('You have chosen a red car')
ELSE
 dbms_output.put_line('Please choose a color for
your car');
END IF;
```


The syntax of an IF-THEN-ELSIF Statement in PL/SQL programming language is –

```
DECLARE
  a number(3) := 100;
BEGIN
  IF ( a = 10 ) THEN
    dbms_output.put_line('Value of a is 10' );
  ELSIF ( a = 20 ) THEN
    dbms_output.put_line('Value of a is 20' );
  ELSIF ( a = 30 ) THEN
    dbms_output.put_line('Value of a is 30' );
  ELSE
     dbms_output.put_line('None of the values is matching');
  END IF;
  dbms_output.put_line('Exact value of a is: '|| a );
END;
/
```


Example:

```
DECLARE
  grade char(1) := 'A';
BEGIN
  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('Well done');
    when 'D' then dbms_output.put_line('You passed');
    when 'F' then dbms_output.put_line('Better try again');
    else dbms_output.put_line('No such grade');
  END CASE;
END;
/
```

CASE selector

```
  WHEN 'value1' THEN S1;
  WHEN 'value2' THEN S2;
  WHEN 'value3' THEN S3;
  ...
  ELSE Sn;  -- default case
END CASE;
```

The searched CASE statement has no selector and the WHEN clauses of the statement contain search conditions that give Boolean values.

The syntax for the searched case statement in PL/SQL is –

```
CASE
   WHEN selector = 'value1' THEN S1;
   WHEN selector = 'value2' THEN S2;
   WHEN selector = 'value3' THEN S3;
   ...
   ELSE Sn;  -- default case
END CASE;
```

```
DECLARE
   grade char(1) := 'B';
BEGIN
   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('Well done');
      when grade = 'D' then dbms_output.put_line('You passed');
      when grade = 'F' then dbms_output.put_line('Better try again');
      else dbms_output.put_line('No such grade');
   end case;
END;
/
```


```
DECLARE
  a number(3) := 100;
  b number(3) := 200;
BEGIN
  -- check the boolean condition
  IF( a = 100 ) THEN
  -- if condition is true then check the following
    IF( b = 200 ) THEN
    -- if condition is true then print the following
    dbms_output.put_line('Value of a is 100 and b is 200' );
    END IF;
  END IF;
  dbms_output.put_line('Exact value of a is : ' || a );
  dbms_output.put_line('Exact value of b is : ' || b );
END;
/
```


S.No	Loop Type & Description
1	**PL/SQL Basic LOOP** In this loop structure, sequence of statements is enclosed between the LOOP and the END LOOP statements. At each iteration, the sequence of statements is executed and then control resumes at the top of the loop.
2	**PL/SQL WHILE LOOP** Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
3	**PL/SQL FOR LOOP** Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
4	**Nested loops in PL/SQL** You can use one or more loop inside any another basic loop, while, or for loop.

PL/SQL - Basic Loop Statement

```
LOOP
  Sequence of statements;
END LOOP;
```

Example

```
DECLARE
  x number := 10;
BEGIN
  LOOP
    dbms_output.put_line(x);
    x := x + 10;
    IF x > 50 THEN
      exit;
    END IF;
  END LOOP;
  -- after exit, control resumes here
  dbms_output.put_line('After Exit x is: ' || x);
END;
/
```

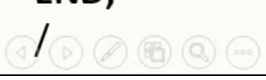
WHILE LOOP statement in PL/SQL programming language

Syntax

```
WHILE condition LOOP
  sequence_of_statements
END LOOP;
```

Example:

```
DECLARE
  a number(2) := 10;
BEGIN
  WHILE a < 20 LOOP
    dbms_output.put_line('value of a: ' || a);
    a := a + 1;
  END LOOP;
END;
/
```



Syntax

```
FOR counter IN initial_value .. final_value LOOP
  sequence_of_statements;
END LOOP;
```

```
DECLARE
  a number(2);
BEGIN
  FOR a in 10 .. 20 LOOP
    dbms_output.put_line('value of a: ' || a);
  END LOOP;
END;
/
```

Reverse FOR LOOP Statement

```
DECLARE
  a number(2) ;
BEGIN
  FOR a IN REVERSE 10 .. 20 LOOP
    dbms_output.put_line('value of a: ' || a);
  END LOOP;
END;
/
```

PL/SQL - Nested Loops

```
LOOP
  Sequence of statements1
  LOOP
    Sequence of statements2
  END LOOP;
END LOOP;
```

```
WHILE condition1 LOOP
  sequence_of_statements1
  WHILE condition2 LOOP
    sequence_of_statements2
  END LOOP;
END LOOP;
```

```
FOR counter1 IN initial_value1 .. final_value1 LOOP
  sequence_of_statements1
  FOR counter2 IN initial_value2 .. final_value2 LOOP
    sequence_of_statements2
  END LOOP;
END LOOP;
```

Example

The following program uses a nested basic loop to find the prime numbers from 2 to 100 –

```
DECLARE
  i number(3);
  j number(3);
BEGIN
  i := 2;
  LOOP
    j:= 2;
    LOOP
      exit WHEN ((mod(i, j) = 0) or (j = i));
      j := j +1;
    END LOOP;
  IF (j = i ) THEN
    dbms_output.put_line(i || ' is prime');
  END IF;
  i := i + 1;
  exit WHEN i = 50;
  END LOOP;
END;
/
```

The Loop Control Statements

S.No	Control Statement & Description
1	**EXIT statement** The Exit statement completes the loop and control passes to the statement immediately after the END LOOP.
2	**CONTINUE statement** Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
3	**GOTO statement** Transfers control to the labeled statement. Though it is not advised to use the GOTO statement in your program.

PL/SQL - EXIT Statement

```
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
      -- terminate the loop using the exit statement
      EXIT;
    END IF;
  END LOOP;
END;
/
```

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

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15

PL/SQL procedure successfully completed.
```


The EXIT WHEN Statement

```
DECLARE
  a number(2) := 10;
BEGIN
  -- while loop execution
  WHILE a < 20 LOOP
    dbms_output.put_line ('value of a: ' || a);
    a := a + 1;
    -- terminate the loop using the exit when statement
  EXIT WHEN a > 15;
  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: 15

PL/SQL procedure successfully completed.
```

STRING IN PL/SQL

The string in PL/SQL is actually a sequence of characters with an optional size specification.
The characters could be numeric, letters, blank, special characters or a combination of all.

PL/SQL offers three kinds of strings −

•**Fixed-length strings** − In such strings, programmers specify the length while declaring the string.
• The string is right-padded with spaces to the length so specified.

•**Variable-length strings** − In such strings, a maximum length up to 32,767, for the string is specified and no padding takes place.

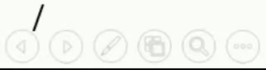
•**Character large objects (CLOBs)** − These are variable-length strings that can be up to 128 terabytes.

Declaring String Variables

Oracle database provides numerous string datatypes, such as CHAR, NCHAR, VARCHAR2, NVARCHAR2, CLOB, and NCLOB.

The datatypes prefixed with an **'N'** are **'national character set'** datatypes, that store Unicode character data.

```
DECLARE
  name varchar2(20);
  company varchar2(30);
  introduction clob;
  choice char(1);
BEGIN
  name := 'John Smith';
  company := 'Infotech';
  introduction := ' Hello! I''m John Smith from Infotech.';
  choice := 'y';
  IF choice = 'y' THEN
    dbms_output.put_line(name);
    dbms_output.put_line(company);
    dbms_output.put_line(introduction);
  END IF;
END;
/
```



PL/SQL String Functions and Operators

S.No	Function & Purpose
1	**ASCII(x);** Returns the ASCII value of the character x.
2	**CHR(x);** Returns the character with the ASCII value of x.
3	**CONCAT(x, y);** Concatenates the strings x and y and returns the appended string.
4	**INITCAP(x);** Converts the initial letter of each word in x to uppercase and returns that string.
5	**INSTR(x, find_string [, start] [, occurrence]);** Searches for **find_string** in x and returns the position at which it occurs.
6	**INSTRB(x);** Returns the location of a string within another string, but returns the value in bytes.
7	**LENGTH(x);** Returns the number of characters in x.
8	**LENGTHB(x);** Returns the length of a character string in bytes for single byte character set.
9	**LOWER(x);** Converts the letters in x to lowercase and returns that string.
10	**LPAD(x, width [, pad_string]) ;** Pads **x** with spaces to the left, to bring the total length of the string up to width characters.
11	**LTRIM(x [, trim_string]);** Trims characters from the left of **x**.
12	**NANVL(x, value);** Returns value if x matches the NaN special value (not a number), otherwise **x** is returned.
13	**NLS_INITCAP(x);** Same as the INITCAP function except that it can use a different sort method as specified by NLSSORT.

14	**NLS_LOWER(x) ;** Same as the LOWER function except that it can use a different sort method as specified by NLSSORT.
15	**NLS_UPPER(x);** Same as the UPPER function except that it can use a different sort method as specified by NLSSORT.
16	**NLSSORT(x);** Changes the method of sorting the characters. Must be specified before any NLS function; otherwise, the default sort will be used.
17	**NVL(x, value);** Returns value if **x** is null; otherwise, x is returned.
18	**NVL2(x, value1, value2);** Returns value1 if x is not null; if x is null, value2 is returned.
19	**REPLACE(x, search_string, replace_string);** Searches **x** for search_string and replaces it with replace_string.
20	**RPAD(x, width [, pad_string]);** Pads **x** to the right.
21	**RTRIM(x [, trim_string]);** Trims **x** from the right.
22	**SOUNDEX(x) ;** Returns a string containing the phonetic representation of **x**.
23	**SUBSTR(x, start [, length]);** Returns a substring of **x** that begins at the position specified by start. An optional length for the substring may be supplied.
24	**SUBSTRB(x);** Same as SUBSTR except that the parameters are expressed in bytes instead of characters for the single-byte character systems.
25	**TRIM([trim_char FROM) x);** Trims characters from the left and right of **x**.
26	**UPPER(x);** Converts the letters in x to uppercase and returns that string.

```
DECLARE
  greetings varchar2(11) := 'hello world';
BEGIN
  dbms_output.put_line(UPPER(greetings));

  dbms_output.put_line(LOWER(greetings));

  dbms_output.put_line(INITCAP(greetings));

  /* retrieve the first character in the string */
  dbms_output.put_line ( SUBSTR (greetings, 1, 1));

  /* retrieve the last character in the string */
  dbms_output.put_line ( SUBSTR (greetings, -1, 1));

  /* retrieve five characters,
    starting from the seventh position. */
  dbms_output.put_line ( SUBSTR (greetings, 7, 5));

  /* retrieve the remainder of the string,
    starting from the second position. */
  dbms_output.put_line ( SUBSTR (greetings, 2));

  /* find the location of the first "e" */
  dbms_output.put_line ( INSTR (greetings, 'e'));
END;
/
```

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

```
HELLO WORLD
hello world
Hello World
h
d
World
ello World
2

PL/SQL procedure successfully completed.
```

```
DECLARE
  greetings varchar2(30) := '......Hello World.....';
BEGIN
  dbms_output.put_line(RTRIM(greetings,'.'));
  dbms_output.put_line(LTRIM(greetings, '.'));
  dbms_output.put_line(TRIM( '.' from greetings));
END;
/
```

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

```
......Hello World
Hello World.....
Hello World

PL/SQL procedure successfully completed.
```

PL/SQL - Arrays

The PL/SQL programming language provides a data structure called the **VARRAY**, which can store a fixed-size sequential collection of elements of the same type.

A varray type is created with the **CREATE TYPE** statement.
You must specify the maximum size and the type of elements stored in the varray.

```
CREATE OR REPLACE TYPE varray_type_name IS VARRAY(n) of <element_type>
```

Where,
- ***varray_type_name*** **is a valid attribute name,**
- ***n*** **is the number of elements (maximum) in the varray,**
- ***element_type*** **is the data type of the elements of the array.**

Maximum size of a varray can be changed using the **ALTER TYPE** statement.

```
CREATE Or REPLACE TYPE namearray IS VARRAY(3) OF VARCHAR2(10);
/
```

```
TYPE namearray IS VARRAY(5) OF VARCHAR2(10);

Type grades IS VARRAY(5) OF INTEGER;
```

```
DECLARE
  type namesarray IS VARRAY(5) OF VARCHAR2(10);
   type grades IS VARRAY(5) OF INTEGER;
  names namesarray;
  marks grades;
  total integer;
BEGIN
  names := namesarray('Kavita', 'Pritam', 'Ayan', 'Rishav', 'Aziz');
  marks:= grades(98, 97, 78, 87, 92);
  total := names.count;
  dbms_output.put_line('Total '|| total || ' Students');
  FOR i in 1 .. total LOOP
    dbms_output.put_line('Student: ' || names(i) || '
    Marks: ' || marks(i));
  END LOOP;
END;
/
```

```
Total 5 Students
Student: Kavita  Marks: 98
Student: Pritam  Marks: 97
Student: Ayan  Marks: 78
Student: Rishav  Marks: 87
Student: Aziz  Marks: 92

PL/SQL procedure successfully completed.
```


Please note −

•In Oracle environment, the starting index for varrays is always 1.

•You can initialize the varray elements using the constructor method of the varray type, which has the same name as the varray.

•Varrays are one-dimensional arrays.

•A varray is automatically NULL when it is declared and must be initialized before its elements can be referenced.

PL/SQL - Procedures

The PL/SQL stored procedure or simply a procedure is a PL/SQL block which performs one or more specific tasks.

It is just like procedures in other programming languages.

The procedure contains a header and a body.

- **Header:** The header contains the name of the procedure and the parameters or variables passed to the procedure.

- **Body:** The body contains a declaration section, execution section and exception section similar to a general PL/SQL block.

How to pass parameters in procedure:

When you want to create a procedure or function, you have to define parameters .

There is three ways to pass parameters in procedure

IN parameters:

The IN parameter can be referenced by the procedure or function.
The value of the parameter cannot be overwritten by the procedure or the function.

OUT parameters:

The OUT parameter cannot be referenced by the procedure or function, but the value of the parameter can be overwritten by the procedure or function.

INOUT parameters:

The INOUT parameter can be referenced by the procedure or function and the value of the parameter can be overwritten by the procedure or function.

A procedure may or may not return any value.

Syntax for creating procedure:

```
CREATE [OR REPLACE] PROCEDURE procedure_name
    [ (parameter [,parameter]) ]
IS
   [declaration_section]
BEGIN
   executable_section
[EXCEPTION
   exception_section]
END [procedure_name];
```


Create procedure example

In this example, we are going to insert record in user table. So you need to create user table first.

Table creation:

```
create table user(id number(10) primary key,name varchar2(100));
```

Now write the procedure code to insert record in user table.

Procedure Code:

```
create or replace procedure "INSERTUSER" (id IN NUMBER, name IN VARCHAR2)
is
begin
insert into user values(id,name);
end;
/
```


PL/SQL program to call procedure

Let's see the code to call above created procedure.

```
BEGIN
   insertuser(101,'Rahul');
   dbms_output.put_line('record inserted successfully');
END;

/
```

Now, see the "USER" table, you will see one record is inserted.

ID	Name
101	Rahul

PL/SQL Function

The PL/SQL Function is very similar to PL/SQL Procedure.

The main difference between procedure and a function is, a function must always return a value, and on the other hand a procedure may or may not return a value.

Except this, all the other things of PL/SQL procedure are true for PL/SQL function too.

Syntax to create a function:

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

- **Function_name:** specifies the name of the function.
- **[OR REPLACE]** option allows modifying an existing function.
- The **optional parameter list** contains name, mode and types of the parameters.
- **IN** represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure.

The function must contain a return statement.

- RETURN clause specifies that 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.

PL/SQL Function Example

Let's see a simple example to **create a function**.

```
create or replace function adder(n1 in number, n2 in number)
return number
is
n3 number(8);
begin
n3 :=n1+n2;
return n3;
end;
/
```

Now write another program to **call the function**.

```
DECLARE
   n3 number(2);
BEGIN
   n3 := adder(11,22);
   dbms_output.put_line('Addition is: ' || n3);
END;
/
```

Output:

```
Addition is: 33 Statement processed.
 0.05 seconds
```

```
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;
/
```

```
Total no. of Customers: 4
PL/SQL procedure successfully completed.
```

PL/SQL Cursor

When an SQL statement is processed, Oracle creates a memory area known as context area.

A cursor is a pointer to this context area. It contains all information needed for processing the statement.

In PL/SQL, the context area is controlled by Cursor.

A cursor contains information on a select statement and the rows of data accessed by it.

A cursor is used to referred to a program to fetch and process the rows returned by the SQL statement, one at a time.

There are two types of cursors:

- **Implicit Cursors**
- **Explicit Cursors**

PL/SQL Implicit Cursors

The implicit cursors are automatically generated by Oracle while an SQL statement is executed, if you don't use an explicit cursor for the statement.

These are created by default to process the statements when DML statements like
INSERT, UPDATE, DELETE etc. are executed.

Orcale provides some attributes known as Implicit cursor's attributes to check the status of DML operations. Some of them are**:**
%FOUND, %NOTFOUND, %ROWCOUNT and %ISOPEN.

For example:

When you execute the SQL statements like INSERT, UPDATE, DELETE then the cursor attributes tell whether any rows are affected and how many have been affected.

If you run a SELECT INTO statement in PL/SQL block, the implicit cursor attribute can be used to find out whether any row has been returned by the SELECT statement.

It will return an error if there no data is selected.

Attribute	Description
%FOUND	Its return value is TRUE if DML statements like INSERT, DELETE and UPDATE affect at least one row or more rows or a SELECT INTO statement returned one or more rows. Otherwise it returns FALSE.
%NOTFOUND	Its return value is TRUE if DML statements like INSERT, DELETE and UPDATE affect no row, or a SELECT INTO statement return no rows. Otherwise it returns FALSE. It is a just opposite of %FOUND.
%ISOPEN	It always returns FALSE for implicit cursors, because the SQL cursor is automatically closed after executing its associated SQL statements.
%ROWCOUNT	It returns the number of rows affected by DML statements like INSERT, DELETE, and UPDATE or returned by a SELECT INTO statement.

Create procedure:

```
DECLARE
  total_rows number(2);
BEGIN
  UPDATE  customers
  SET salary = salary + 5000;
  IF sql%notfound THEN
    dbms_output.put_line('no customers updated');
  ELSIF sql%found THEN
    total_rows := sql%rowcount;
    dbms_output.put_line( total_rows || ' customers updated ');
  END IF;
END;
/
```

Output:
6 customers updated
PL/SQL procedure successfully completed.

PL/SQL Explicit Cursors

The Explicit cursors are defined by the programmers to gain more control over the context area.

These cursors 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.

CURSOR cursor_name **IS** select_statement;

Steps:

You must follow these steps while working with an explicit cursor.

1. Declare the cursor to initialize in the memory.
2. Open the cursor to allocate memory.
3. Fetch the cursor to retrieve data.
4. Close the cursor to release allocated memory.

Declare the cursor:

It defines the cursor with a name and the associated SELECT statement.

Syntax for explicit cursor decleration

```
CURSOR name IS
 SELECT statement;
```

Open the cursor:

It is used to allocate memory for the cursor and make it easy to fetch the rows returned by the SQL statements into it.

Syntax for cursor open:

```
OPEN cursor_name;
```

Fetch the cursor:

It is used to access one row at a time. You can fetch rows from the above-opened cursor as follows:

Syntax for cursor fetch:

```
FETCH cursor_name INTO variable_list;
```

Close the cursor:

It is used to release the allocated memory. The following syntax is used to close the above-opened cursors.

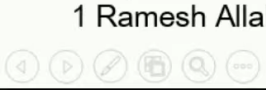
Syntax for cursor close:

```
Close cursor_name;
```

```
DECLARE
  c_id customers.id%type;
  c_name customers.name%type;
  c_addr customers.address%type;
  CURSOR c_customers is
    SELECT id, name, address FROM customers;
BEGIN
  OPEN c_customers;
  LOOP
    FETCH c_customers into c_id, c_name, c_addr;
    EXIT WHEN c_customers%notfound;
    dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
  END LOOP;
  CLOSE c_customers;
END;
/
```

Output:

1 Ramesh Allahabad 2 Suresh Kanpur 3 Mahesh Ghaziabad 4 Chandan Noida 5 Alex Paris 6 Sunita Delhi

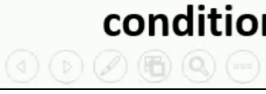


PL/SQL SELECT INTO

PL/SQL SELECT INTO statement is the simplest and fastest way to fetch a single row from a table into variables.

The following illustrates the syntax of the PL/SQL SELECT INTO statement:

```
SELECT
  select_list
INTO
  variable_list
FROM
  table_name
WHERE
  condition;
```



PL/SQL SELECT INTO – selecting one column example

```
DECLARE
  l_customer_name customers.name%TYPE;
BEGIN
  -- get name of the customer 100 and assign it to l_customer_name
  SELECT name INTO l_customer_name
  FROM customers
  WHERE customer_id = 100;
  -- show the customer name
  dbms_output.put_line( v_customer_name );
END;
```


PL/SQL SELECT INTO – selecting a complete row example

```
DECLARE
  r_customer customers%ROWTYPE;
BEGIN
  -- get the information of the customer 100
  SELECT * INTO r_customer
  FROM customers
  WHERE customer_id = 100;
  -- show the customer info
  dbms_output.put_line( r_customer.name || ', website: ' || r_customer.website );
END;
```

Here is the output:

Verizon, website: http://www.vertex.com

PL/SQL

BASIC PART -1
WITH
EXAMPLES

PL/SQL
is a combination of SQL along with the procedural features of programming languages.

PL/SQL
is one of three key programming languages embedded in the Oracle Database, along with SQL itself and Java.

•PL/SQL is a completely portable, high-performance transaction-processing language.

•PL/SQL provides a built-in, interpreted and OS independent programming environment.

•PL/SQL can also directly be called from the command-line **SQL*Plus interface**.

•Direct call can also be made from external programming language calls to database.

•PL/SQL's general syntax is based on that of ADA and Pascal programming language.

•Apart from Oracle, PL/SQL is available **in-memory database** and **IBM DB2**.

- PL/SQL is tightly integrated with SQL.
- It offers extensive error checking.
- It offers numerous data types.
- It offers a variety of programming structures.
- It supports structured programming through functions and procedures.
- It supports object-oriented programming.
- It supports the development of web applications and server pages.


PL/SQL has the following advantages −

•SQL is the standard database language and PL/SQL is strongly integrated with SQL.

•PL/SQL allows sending an entire block of statements to the database at one time.

•This reduces network traffic and provides high performance for the applications.

•PL/SQL gives high productivity to programmers as it can query, transform, and update data in a database.

•PL/SQL saves time on design and debugging by strong features, such as exception handling, encapsulation, data hiding, and object-oriented data types.

•Applications written in PL/SQL are fully portable.
•PL/SQL provides high security level.
•PL/SQL provides access to predefined SQL packages.
•PL/SQL provides support for Object-Oriented Programming.
•PL/SQL provides support for developing Web Applications and Server Pages.

The primary role one can expect by opting PL/SQL is the **Developer role.**

Other positions to consider are:

- Analyst role
- Unix Developer/Administrator
- DBA
- PL/SQL performance optimization developer
- Pro*c developer
- SQL developer
- PL/SQL administrator
- ETL developer
- Informatica Developer
- DB2 professional

BASIC SYNTAX

```
DECLARE
  <declarations section>
BEGIN
  <executable command(s)>
EXCEPTION
  <exception handling>
END;
```

```
DECLARE
  message  varchar2(20):= 'Hello, World!';
BEGIN
  dbms_output.put_line(message);
END;
/
```

The PL/SQL Delimiters

A delimiter is a symbol with a special meaning.

Delimiter	Description
+, -, *, /	Addition, subtraction/negation, multiplication, division
%	Attribute indicator
'	Character string delimiter
.	Component selector
(,)	Expression or list delimiter
:	Host variable indicator
,	Item separator
"	Quoted identifier delimiter
=	Relational operator
@	Remote access indicator
;	Statement terminator
:=	Assignment operator
=>	Association operator
||	Concatenation operator
**	Exponentiation operator
<<, >>	Label delimiter (begin and end)
/*, */	Multi-line comment delimiter (begin and end)
--	Single-line comment indicator
..	Range operator
<, >, <=, >=	Relational operators
<>, '=, ~=, ^=	Different versions of NOT EQUAL

The PL/SQL Comments

```
DECLARE
  -- variable declaration
  message  varchar2(20):= 'Hello, World!';
BEGIN
  /*
  *  PL/SQL executable statement(s)
  */
  dbms_output.put_line(message);
END;
/
```

DATA TYPES

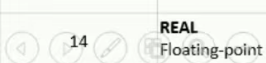
S.No	Category & Description
1	**Scalar** Single values with no internal components, such as a **NUMBER, DATE,** or **BOOLEAN**.
2	**Large Object (LOB)** Pointers to large objects that are stored separately from other data items, such as text, graphic images, video clips, and sound waveforms.
3	**Composite** Data items that have internal components that can be accessed individually. For example, collections and records.
4	**Reference** Pointers to other data items.

PL/SQL Scalar Data Types and Subtypes come under the following categories −

S.No	Date Type & Description
1	**Numeric** Numeric values on which arithmetic operations are performed.
2	**Character** Alphanumeric values that represent single characters or strings of characters.
3	**Boolean** Logical values on which logical operations are performed.
4	**Datetime** Dates and times.

PL/SQL Numeric Data Types and Subtypes

S.No	Data Type & Description
1	**PLS_INTEGER** Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits
2	**BINARY_INTEGER** Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits
3	**BINARY_FLOAT** Single-precision IEEE 754-format floating-point number
4	**BINARY_DOUBLE** Double-precision IEEE 754-format floating-point number
5	**NUMBER(prec, scale)** Fixed-point or floating-point number with absolute value in range 1E-130 to (but not including) 1.0E126. A NUMBER variable can also represent 0
6	**DEC(prec, scale)** ANSI specific fixed-point type with maximum precision of 38 decimal digits
7	**DECIMAL(prec, scale)** IBM specific fixed-point type with maximum precision of 38 decimal digits
8	**NUMERIC(pre, secale)** Floating type with maximum precision of 38 decimal digits
9	**DOUBLE PRECISION** ANSI specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits)
10	**FLOAT** ANSI and IBM specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits)
11	**INT** ANSI specific integer type with maximum precision of 38 decimal digits
12	**INTEGER** ANSI and IBM specific integer type with maximum precision of 38 decimal digits
13	**SMALLINT** ANSI and IBM specific integer type with maximum precision of 38 decimal digits
14	**REAL** Floating-point type with maximum precision of 63 binary digits (approximately 18 decimal digits)




Following is a valid declaration –

```
DECLARE
  num1 INTEGER;
  num2 REAL;
  num3 DOUBLE PRECISION;
BEGIN
  null;
END;
/
```


PL/SQL Character Data Types and Subtypes

S.No	Data Type & Description
1	**CHAR** Fixed-length character string with maximum size of 32,767 bytes
2	**VARCHAR2** Variable-length character string with maximum size of 32,767 bytes
3	**RAW** Variable-length binary or byte string with maximum size of 32,767 bytes, not interpreted by PL/SQL
4	**NCHAR** Fixed-length national character string with maximum size of 32,767 bytes
5	**NVARCHAR2** Variable-length national character string with maximum size of 32,767 bytes
6	**LONG** Variable-length character string with maximum size of 32,760 bytes
7	**LONG RAW** Variable-length binary or byte string with maximum size of 32,760 bytes, not interpreted by PL/SQL
8	**ROWID** Physical row identifier, the address of a row in an ordinary table
9	**UROWID** Universal row identifier (physical, logical, or foreign row identifier)

PL/SQL User-Defined Subtypes

- A subtype is a subset of another data type, which is called its base type.
- A subtype has the same valid operations as its base type, but only a subset of its valid values.
- PL/SQL predefines several subtypes in package **STANDARD**.

For example, PL/SQL predefines the subtypes **CHARACTER** and **INTEGER** as follows −

You can define and use your own subtypes.

```
DECLARE
  SUBTYPE name IS char(20);
  SUBTYPE message IS varchar2(100);
  salutation name;
  greetings message;
BEGIN
  salutation := 'Reader ';
  greetings := 'Welcome to the World of PL/SQL';
  dbms_output.put_line('Hello ' || salutation || greetings);
END;
/
```