**PL/SQL:**

PL/SQL which is a **block-structured** language; this means that the PL/SQL programs are divided and written in logical blocks of code.

Each block consists of three sub-parts –

|  |  |
| --- | --- |
| **S.No** | **Description** |
| 1 | **Declarations**  This section starts with the keyword **DECLARE**. It is an optional section and defines all variables, cursors, subprograms, and other elements to be used in the program. |
| 2 | **Executable Commands**  This section is enclosed between the keywords **BEGIN** and **END** and it is a mandatory section. It consists of the executable PL/SQL statements of the program. It should have at least one executable line of code, which may be just a **NULL command** to indicate that nothing should be executed. |
| 3 | **Exception Handling**  This section starts with the keyword **EXCEPTION**. This optional section contains **exception(s)** that handle errors in the program. |

**NOTE:**

Every PL/SQL statement ends with a semicolon (;). PL/SQL blocks can be nested within other PL/SQL blocks using **BEGIN** and **END**.

Following is the basic structure of a PL/SQL block –

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

The 'Hello World' Example

|  |
| --- |
| DECLARE  message varchar2(20):= 'Hello, World!';  BEGIN  dbms\_output.put\_line(message);  END; |

**OUTPUT:**

|  |
| --- |
| Hello World  PL/SQL procedure successfully completed. |

## The PL/SQL Identifiers

PL/SQL identifiers are 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 Delimiters

A delimiter is a symbol with a special meaning. Following is the list of delimiters in PL/SQL −

|  |  |
| --- | --- |
| **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

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 \*/.

|  |
| --- |
| DECLARE  -- variable declaration  message varchar2(20):= 'Hello, World!';  BEGIN  /\*  \* PL/SQL executable statement(s)  \*/  dbms\_output.put\_line(message);  END; |

## PL/SQL Program Units

A PL/SQL unit is any one of the following −

* PL/SQL block
* Function
* Package
* Package body
* Procedure
* Trigger
* Type
* Type body

Each of these units will be discussed in the following chapters.

**PL/SQL DATATYPE:**

The PL/SQL variables, constants and parameters must have a valid data type, which specifies a storage format, constraints, and a valid range of values.

|  |  |
| --- | --- |
| 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](https://www.tutorialspoint.com/plsql/plsql_data_types.htm) and Subtypes

PL/SQL Scalar [Data Types](https://www.tutorialspoint.com/plsql/plsql_data_types.htm) and Subtypes come under the following categories −

|  |  |
| --- | --- |
| S.No | 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 | **Date time**  Dates and times. |

## PL/SQL Numeric [Data Types](https://www.tutorialspoint.com/plsql/plsql_data_types.htm) and Subtypes

PL/SQL provides subtypes of [data types](https://www.tutorialspoint.com/plsql/plsql_data_types.htm). For example, the data type NUMBER has a subtype called INTEGER. You can use the subtypes in your PL/SQL program to make the [data types](https://www.tutorialspoint.com/plsql/plsql_data_types.htm) compatible with [data types](https://www.tutorialspoint.com/plsql/plsql_data_types.htm) in other programs while embedding the PL/SQL code in another program, such as a Java program.

|  |  |
| --- | --- |
| S.No | 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)  **INT**  ANSI specific integer type with maximum precision of 38 decimal digits |
|  | **INTEGER**  ANSI and IBM specific integer type with maximum precision of 38 decimal digits |
|  | **SMALLINT**  ANSI and IBM specific integer type with maximum precision of 38 decimal digits |
|  | **REAL**  Floating-point type with maximum precision of 63 binary digits (approximately 18 decimal digits) |

**Example:**

|  |
| --- |
| DECLARE  num1 INTEGER;  num2 REAL;  num3 DOUBLE PRECISION;  BEGIN  null;  END; |

## PL/SQL Character [Data Types](https://www.tutorialspoint.com/plsql/plsql_data_types.htm) and Subtypes

|  |  |
| --- | --- |
| S.No | 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 |
|  | **LONG RAW**  Variable-length binary or byte string with maximum size of 32,760 bytes, not interpreted by PL/SQL |
|  | **ROWID**  Physical row identifier, the address of a row in an ordinary table |
|  | **UROWID**  Universal row identifier (physical, logical, or foreign row identifier) |

## PL/SQL Boolean [Data Types](https://www.tutorialspoint.com/plsql/plsql_data_types.htm)

The **BOOLEAN** data type stores logical values that are used in logical operations. The logical values are the Boolean values **TRUE** and **FALSE** and the value **NULL**.

However, SQL has no data type equivalent to BOOLEAN. Therefore, Boolean values cannot be used in −

* SQL statements
* Built-in SQL functions (such as **TO\_CHAR**)
* PL/SQL functions invoked from SQL statements

## PL/SQL Date time and Interval Types

The **DATE** datatype is used to store fixed-length datetimes, which include the time of day in seconds since midnight. Valid dates range from January 1, 4712 BC to December 31, 9999 AD.

The default date format is set by the Oracle initialization parameter NLS\_DATE\_FORMAT. For example, the default might be 'DD-MON-YY', which includes a two-digit number for the day of the month, an abbreviation of the month name, and the last two digits of the year. For example, 01-OCT-12.

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Valid Date time Values** |  |
| YEAR | -4712 to 9999 (excluding year 0) | Any nonzero integer |
| MONTH | 01 to 12 | 0 to 11 |
| DAY | 01 to 31 (limited by the values of MONTH and YEAR, according to the rules of the calendar for the locale) | Any nonzero integer |
| HOUR | 00 to 23 | 0 to 23 |
| MINUTE | 00 to 59 | 0 to 59 |
| SECOND | 00 to 59.9(n), where 9(n) is the precision of time fractional seconds | 0 to 59.9(n), where 9(n) is the precision of interval fractional seconds |
| TIMEZONE\_HOUR | -12 to 14 (range accommodates daylight savings time changes) | Not applicable |
| TIMEZONE\_MINUTE | 00 to 59 | Not applicable |
| TIMEZONE\_REGION | Found in the dynamic performance view V$TIMEZONE\_NAMES | Not applicable |
| TIMEZONE\_ABBR | Found in the dynamic performance view V$TIMEZONE\_NAMES | Not applicable |

## PL/SQL Large Object (LOB) [Data Types](https://www.tutorialspoint.com/plsql/plsql_data_types.htm)

Large Object (LOB) [data types](https://www.tutorialspoint.com/plsql/plsql_data_types.htm) refer to large data items such as text, graphic images, video clips, and sound waveforms. LOB [data types](https://www.tutorialspoint.com/plsql/plsql_data_types.htm) allow efficient, random, piecewise access to this data.

|  |  |  |
| --- | --- | --- |
| **Data Type** | **Description** | **Size** |
| BFILE | Used to store large binary objects in operating system files outside the database | System-dependent. Cannot exceed 4 gigabytes (GB). |
| BLOB | Used to store large binary objects in the database. | 8 to 128 terabytes (TB) |
| CLOB | Used to store large blocks of character data in the database. | 8 to 128 TB |
| NCLOB | Used to store large blocks of NCHAR data in the database. | 8 to 128 TB |

## 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 −

|  |
| --- |
| SUBTYPE CHARACTER IS CHAR;  SUBTYPE INTEGER IS NUMBER(38,0); |

**Example:**

|  |
| --- |
| 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; |

## NULLs in PL/SQL

PL/SQL NULL values represent **missing** or **unknown data** and they are not an integer, a character, or any other specific data type. Note that **NULL** is not the same as an empty data string or the null character value **'\0'**. A null can be assigned but it cannot be equated with anything, including itself.

**Variable:**

 A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in PL/SQL has a specific data type, which determines the size and the layout of the variable's memory; the range of values that can be stored within that memory and the set of operations that can be applied to the variable.

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.

## Variable Declaration in PL/SQL

PL/SQL [variables](https://www.tutorialspoint.com/plsql/plsql_variable_types.htm) must be declared in the declaration section or in a package as a global variable. When you declare a variable, PL/SQL allocates memory for the variable's value and the storage location is identified by the variable name.

**Syntax:**

|  |
| --- |
| variable\_name [CONSTANT] datatype [NOT NULL] [:= | DEFAULT initial\_value] |

**Example:**

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

When you provide a size, scale or precision limit with the data type, it is called a **constrained declaration**.

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

## Initializing [Variables](https://www.tutorialspoint.com/plsql/plsql_variable_types.htm) in PL/SQL

Whenever you declare a variable, PL/SQL assigns it a default value of NULL. If you want to initialize a variable with a value other than the NULL value, you can do so during the declaration, using either of the following −

* The **DEFAULT** keyword
* The **assignment** operator

For example –

|  |
| --- |
| counter binary\_integer := 0;  greetings varchar2(20) DEFAULT 'Have a Good Day'; |

**NOTE:**

You can also specify that a variable should not have a **NULL** value using the **NOT NULL** constraint. If you use the NOT NULL constraint, you must explicitly assign an initial value for that variable.

It is a good [programming](https://www.tutorialspoint.com/plsql/plsql_variable_types.htm) practice to initialize [variables](https://www.tutorialspoint.com/plsql/plsql_variable_types.htm) properly otherwise, sometimes programs would produce unexpected results.

|  |
| --- |
| 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; |

**OUTPUT:**

|  |
| --- |
| Value of c: 30  Value of f: 23.333333333333333333  PL/SQL procedure successfully completed. |

## Variable Scope in PL/SQL

PL/SQL allows the nesting of blocks, i.e., each program block may contain another inner block. If a variable is declared within an inner block, it is not accessible to the outer block. However, if a variable is declared and accessible to an outer block, it is also accessible to all nested inner blocks. There are two types of variable scope −

* **Local**[variables](https://www.tutorialspoint.com/plsql/plsql_variable_types.htm) − [Variables](https://www.tutorialspoint.com/plsql/plsql_variable_types.htm) declared in an inner block and not accessible to ou
* ter blocks.
* **Global**[variables](https://www.tutorialspoint.com/plsql/plsql_variable_types.htm) − [Variables](https://www.tutorialspoint.com/plsql/plsql_variable_types.htm) declared in the outermost block or a package.

|  |
| --- |
| DECLARE  -- Global [variables](https://www.tutorialspoint.com/plsql/plsql_variable_types.htm)  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](https://www.tutorialspoint.com/plsql/plsql_variable_types.htm)  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; |

OUTPUT:

|  |
| --- |
| Outer Variable num1: 95  Outer Variable num2: 85  Inner Variable num1: 195  Inner Variable num2: 185  PL/SQL procedure successfully completed. |

## Assigning SQL Query Results to PL/SQL [Variables](https://www.tutorialspoint.com/plsql/plsql_variable_types.htm)

You can use the **SELECT INTO** statement of SQL to assign values to PL/SQL [variables](https://www.tutorialspoint.com/plsql/plsql_variable_types.htm). For each item in the **SELECT list**, there must be a corresponding, type-compatible variable in the **INTO list**. The following example illustrates the concept. Let us create a table named CUSTOMERS −

|  |
| --- |
| 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, 'Ramesh', 32, 'Ahmedabad', 2000.00 );  INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)  VALUES (2, 'Khilan', 25, 'Delhi', 1500.00 );  INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)  VALUES (3, 'kaushik', 23, 'Kota', 2000.00 );    INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)  VALUES (4, 'Chaitali', 25, 'Mumbai', 6500.00 );    INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)  VALUES (5, 'Hardik', 27, 'Bhopal', 8500.00 );  INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)  VALUES (6, 'Komal', 22, 'MP', 4500.00 ); |

The following program assigns values from the above table to PL/SQL [variables](https://www.tutorialspoint.com/plsql/plsql_variable_types.htm) 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 Ramesh from Ahmedabad earns 2000  PL/SQL procedure completed successfully |

## Declaring a Constant

A constant is declared using the **CONSTANT** keyword. It requires an initial value and does not allow that value to be changed. For example −

|  |
| --- |
| 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; |

|  |
| --- |
| Radius: 9.5  Diameter: 19  Circumference: 59.69  Area: 283.53  Pl/SQL procedure successfully completed. |

## The PL/SQL Literals

A literal is an explicit numeric, character, string, or Boolean value not represented by an identifier. For example, TRUE, 786, NULL, '[tutorialspoint](https://www.tutorialspoint.com/plsql/plsql_constants.htm)' are all literals of type Boolean, number, or string. PL/SQL, literals are case-sensitive. PL/SQL supports the following kinds of literals −

* Numeric Literals
* Character Literals
* String Literals
* BOOLEAN Literals
* Date and Time Literals

**Numeric Literals**

* 050 78 -14 0 +32767
* 6.6667 0.0 -12.0 3.14159 +7800.00
* 6E5 1.0E-8 3.14159e0 -1E38 -9.5e-3

**Character Literals**

'A' '%' '9' ' ' 'z' '('

**String Literals**

'Hello, world!'

'[Tutorials](https://www.tutorialspoint.com/plsql/plsql_constants.htm) Point'

'19-NOV-12'

**BOOLEAN Literals**

TRUE, FALSE, and NULL.

**Date and Time Literals**

DATE '1978-12-25';

TIMESTAMP '2012-10-29 12:01:01';

|  |
| --- |
| DECLARE  message varchar2(30):= 'That''s [tutorialspoint](https://www.tutorialspoint.com/plsql/plsql_constants.htm).com!';  BEGIN  dbms\_output.put\_line(message);  END; |

|  |
| --- |
| That's [tutorialspoint](https://www.tutorialspoint.com/plsql/plsql_constants.htm).com!  PL/SQL procedure successfully completed. |

**Decisions:**

[**IF - THEN statement**](https://www.tutorialspoint.com/plsql/plsql_if_then.htm)

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.

[**IF-THEN-ELSE statement**](https://www.tutorialspoint.com/plsql/plsql_if_then_else.htm)

**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.

[**IF-THEN-ELSIF statement**](https://www.tutorialspoint.com/plsql/plsql_if_then_elsif.htm)

It allows you to choose between several alternatives.

[**Case statement**](https://www.tutorialspoint.com/plsql/plsql_case_statement.htm)

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.

[**Searched CASE statement**](https://www.tutorialspoint.com/plsql/plsql_searched_case.htm)

The searched CASE statement **has no selector**, and it's WHEN clauses contain search conditions that yield Boolean values.

[**Nested IF-THEN-ELSE**](https://www.tutorialspoint.com/plsql/plsql_nested_if.htm)

You can use one **IF-THEN** or **IF-THEN-ELSIF** statement inside another **IF-THEN** or **IF-THEN-ELSIF** statement(s).

**LOOPS:**

[**PL/SQL Basic LOOP**](https://www.tutorialspoint.com/plsql/plsql_basic_loop.htm)

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.

[**PL/SQL WHILE LOOP**](https://www.tutorialspoint.com/plsql/plsql_while_loop.htm)

Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.

[**PL/SQL FOR LOOP**](https://www.tutorialspoint.com/plsql/plsql_for_loop.htm)

Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.

## Labeling a PL/SQL Loop

PL/SQL [loops](https://www.tutorialspoint.com/plsql/plsql_loops.htm) can be labeled. The label should be enclosed by double angle brackets (<< and >>) and appear at the beginning of the LOOP statement. The label name can also appear at the end of the LOOP statement. You may use the label in the EXIT statement to exit from the loop.

|  |
| --- |
| DECLARE  i number(1);  j number(1);  BEGIN  << outer\_loop >>  FOR i IN 1..3 LOOP  << inner\_loop >>  FOR j IN 1..3 LOOP  dbms\_output.put\_line('i is: '|| i || ' and j is: ' || j);  END loop inner\_loop;  END loop outer\_loop;  END; |

|  |
| --- |
| i is: 1 and j is: 1  i is: 1 and j is: 2  i is: 1 and j is: 3  i is: 2 and j is: 1  i is: 2 and j is: 2  i is: 2 and j is: 3  i is: 3 and j is: 1  i is: 3 and j is: 2  i is: 3 and j is: 3  PL/SQL procedure successfully completed. |

## The Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

PL/SQL supports the following control statements. Labeling [loops](https://www.tutorialspoint.com/plsql/plsql_loops.htm) also help in taking the control outside a loop.

[**EXIT statement**](https://www.tutorialspoint.com/plsql/plsql_exit_statement.htm)

The Exit statement completes the loop and control passes to the statement immediately after the END LOOP.

[**CONTINUE statement**](https://www.tutorialspoint.com/plsql/plsql_continue_statement.htm)

Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

[**GOTO statement**](https://www.tutorialspoint.com/plsql/plsql_goto_statement.htm)

Transfers control to the labeled statement. Though it is not advised to use the GOTO statement in your program.

**STRINGS:**

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](https://www.tutorialspoint.com/plsql/plsql_strings.htm) −

* **Fixed-length**[strings](https://www.tutorialspoint.com/plsql/plsql_strings.htm) − in such [strings](https://www.tutorialspoint.com/plsql/plsql_strings.htm), programmers specify the length while declaring the string. The string is right-padded with spaces to the length so specified.
* **Variable-length**[strings](https://www.tutorialspoint.com/plsql/plsql_strings.htm) − in such [strings](https://www.tutorialspoint.com/plsql/plsql_strings.htm), 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](https://www.tutorialspoint.com/plsql/plsql_strings.htm) that can be up to 128 terabytes.

PL/SQL [strings](https://www.tutorialspoint.com/plsql/plsql_strings.htm) could be either variables or literals. A string literal is enclosed within quotation marks. For example,

'This is a string literal.' Or 'hello world'

## 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.

If you need to declare a variable-length string, you must provide the maximum length of that string. For example, the VARCHAR2 data type.

|  |
| --- |
| 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; |

|  |
| --- |
| John Smith  Infotech  Hello! I'm John Smith from Infotech.  PL/SQL procedure successfully completed |

To declare a fixed-length string, use the CHAR datatype. Here you do not have to specify a maximum length for a fixed-length variable. If you leave off the length constraint, Oracle Database automatically uses a maximum length required. The following two declarations are identical −

red\_flag CHAR(1) := 'Y';

red\_flag CHAR := 'Y';

**STRING FUNCTIONS:**

|  |  |
| --- | --- |
| **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](https://www.tutorialspoint.com/plsql/plsql_strings.htm) 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. |

### **Example 1**

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 –

**Example1:**

|  |
| --- |
| HELLO WORLD  hello world  Hello World  h  d  World  ello World  2  PL/SQL procedure successfully completed. |

### **Example 2**

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;

/

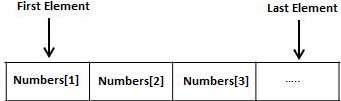
When the above code is executed at the SQL prompt, it produces the following result −

|  |
| --- |
| ......Hello World  Hello World.....  Hello World  PL/SQL procedure successfully completed. |

**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 is used to store an ordered collection of data, however it is often better to think of an array as a collection of variables of the same type.

All varrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



An array is a part of collection type data and it stands for variable-size arrays. We will study other collection types in a later chapter **'PL/SQL Collections'**.

Each element in a **varray** has an index associated with it. It also has a maximum size that can be changed dynamically.

## Creating a Varray 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.

The basic syntax for creating a VARRAY type at the schema level is −

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.

For example,

CREATE Or REPLACE TYPE namearray AS VARRAY(3) OF VARCHAR2(10);

/

Type created.

The basic syntax for creating a VARRAY type within a PL/SQL block is −

TYPE varray\_type\_name IS VARRAY(n) of <element\_type>

For example −

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

Type grades IS VARRAY(5) OF INTEGER;

Let us now work out on a few examples to understand the concept −

### **Example 1**

The following program illustrates the use of varrays −

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;

/

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

|  |
| --- |
| 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. |

**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.

### **Example 2**

Elements of a varray could also be a %ROWTYPE of any database table or %TYPE of any database table field. The following example illustrates the concept.

We will use the CUSTOMERS table stored in our database as −

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 example makes the use of **cursor**, which you will study in detail in a separate chapter.

DECLARE

CURSOR c\_customers is

SELECT name FROM customers;

type c\_list is varray (6) of customers.name%type;

name\_list c\_list := c\_list();

counter integer :=0;

BEGIN

FOR n IN c\_customers LOOP

counter := counter + 1;

name\_list.extend;

name\_list(counter) := n.name;

dbms\_output.put\_line('Customer('||counter ||'):'||name\_list(counter));

END LOOP;

END;

/

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

Customer(1): Ramesh

Customer(2): Khilan

Customer(3): kaushik

Customer(4): Chaitali

Customer(5): Hardik

Customer(6): Komal

PL/SQL procedure successfully completed.

**subprogram**:

A **subprogram** is a program unit/module that performs a particular task. These subprograms are combined to form larger programs. This is basically called the 'Modular design'. A subprogram can be invoked by another subprogram or program which is called the **calling program**.

A subprogram can be created −

* At the schema level
* Inside a package
* Inside a PL/SQL block

At the schema level, subprogram is a **standalone subprogram**. It is created with the CREATE PROCEDURE or the CREATE FUNCTION statement. It is stored in the database and can be deleted with the DROP PROCEDURE or DROP FUNCTION statement.

A subprogram created inside a package is a **packaged subprogram**. It is stored in the database and can be deleted only when the package is deleted with the DROP PACKAGE statement. We will discuss packages in the chapter **'PL/SQL - Packages'**.

PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters. PL/SQL provides two kinds of subprograms −

* **Functions** − These subprograms return a single value; mainly used to compute and return a value.
* [Procedures](https://www.tutorialspoint.com/plsql/plsql_procedures.htm) − These subprograms do not return a value directly; mainly used to perform an action.

This chapter is going to cover important aspects of a **PL/SQL procedure**. We will discuss **PL/SQL function** in the next chapter.

## Parts of a PL/SQL Subprogram

Each PL/SQL subprogram has a name, and may also have a parameter list. Like anonymous PL/SQL blocks, the named blocks will also have the following three parts −

|  |  |
| --- | --- |
| **S.No** | **Parts & Description** |
| 1 | **Declarative Part**  It is an optional part. However, the declarative part for a subprogram does not start with the DECLARE keyword. It contains declarations of types, cursors, constants, variables, exceptions, and nested subprograms. These items are local to the subprogram and cease to exist when the subprogram completes execution. |
| 2 | **Executable Part**  This is a mandatory part and contains statements that perform the designated action. |
| 3 | **Exception-handling**  This is again an optional part. It contains the code that handles run-time errors. |

## Creating a Procedure

A procedure is created with the **CREATE OR REPLACE PROCEDURE** statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows −

CREATE [OR REPLACE] PROCEDURE procedure\_name

[(parameter\_name [IN | OUT | IN OUT] type [, ...])]

{IS | AS}

BEGIN

< procedure\_body >

END procedure\_name;

Where,

* *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.

### **Example**

The following example creates a simple procedure that displays the string 'Hello World!' on the screen when executed.

CREATE OR REPLACE PROCEDURE greetings

AS

BEGIN

dbms\_output.put\_line('Hello World!');

END;

/

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

Procedure created.

## Executing a Standalone Procedure

A standalone procedure can be called in two ways −

* Using the **EXECUTE** keyword
* Calling the name of the procedure from a PL/SQL block

The above procedure named **'greetings'** can be called with the EXECUTE keyword as −

EXECUTE greetings;

The above call will display −

Hello World

PL/SQL procedure successfully completed.

The procedure can also be called from another PL/SQL block −

BEGIN

greetings;

END;

/

The above call will display −

Hello World

PL/SQL procedure successfully completed.

## Deleting a Standalone Procedure

A standalone procedure is deleted with the **DROP PROCEDURE** statement. Syntax for deleting a procedure is −

DROP PROCEDURE procedure-name;

You can drop the greetings procedure by using the following statement −

DROP PROCEDURE greetings;

## Parameter Modes in PL/SQL Subprograms

The following table lists out the parameter modes in PL/SQL subprograms −

|  |  |
| --- | --- |
| **S.No** | **Parameter Mode & Description** |
| 1 | **IN**  An IN parameter lets you pass a value to the subprogram. **It is a read-only parameter**. Inside the subprogram, an IN parameter acts like a constant. It cannot be assigned a value. You can pass a constant, literal, initialized variable, or expression as an IN parameter. You can also initialize it to a default value; however, in that case, it is omitted from the subprogram call. **It is the default mode of parameter passing. Parameters are passed by reference**. |
| 2 | **OUT**  An OUT parameter returns a value to the calling program. Inside the subprogram, an OUT parameter acts like a variable. You can change its value and reference the value after assigning it. **The actual parameter must be variable and it is passed by value**. |
| 3 | **IN OUT**  An **IN OUT** parameter passes an initial value to a subprogram and returns an updated value to the caller. It can be assigned a value and the value can be read.  The actual parameter corresponding to an IN OUT formal parameter must be a variable, not a constant or an expression. Formal parameter must be assigned a value. **Actual parameter is passed by value.** |

### **IN & OUT Mode Example 1**

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;

/

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

|  |
| --- |
| Minimum of (23, 45) : 23  PL/SQL procedure successfully completed. |

### **IN & OUT Mode Example 2**

This procedure computes the square of value of a passed value. This example shows how 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;

/

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

|  |
| --- |
| Square of (23): 529  PL/SQL procedure successfully completed. |

## Methods for Passing Parameters

Actual parameters can be passed in three ways −

* Positional notation
* Named notation
* Mixed notation

### **Positional Notation**

In positional notation, you can call the procedure as −

findMin(a, b, c, d);

In positional notation, the first actual parameter is substituted for the first formal parameter; the second actual parameter is substituted for the second formal parameter, and so on. So, **a** is substituted for **x, b** is substituted for **y, c** is substituted for **z** and **d** is substituted for **m**.

### **Named Notation**

In named notation, the actual parameter is associated with the formal parameter using the **arrow symbol ( => )**. The procedure call will be like the following −

findMin(x => a, y => b, z => c, m => d);

 A function is same as a procedure except that it returns a value. Therefore, all the discussions of the previous chapter are true for [functions](https://www.tutorialspoint.com/plsql/plsql_functions.htm) too.

## 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 outside and OUT represents the parameter that will be used to return a value outside of the procedure.
* 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, which we had created in the [PL/SQL Variables](https://www.tutorialspoint.com/plsql/plsql_variable_types.htm) chapter −

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 |

+----+----------+-----+-----------+----------+

CREATE OR REPLACE FUNCTION totalCustomers

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. Following program calls the function **totalCustomers** from an anonymous block −

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  PL/SQL procedure successfully completed. |

### **Example**

The following example demonstrates Declaring, Defining, and Invoking a Simple PL/SQL Function that computes and returns the maximum of two values.

DECLARE

a number;

b number;

c number;

FUNCTION findMax(x IN number, y IN number)

RETURN number

IS

z number;

BEGIN

IF x > y THEN

z:= x;

ELSE

Z:= y;

END IF;

RETURN z;

END;

BEGIN

a:= 23;

b:= 45;

c := findMax(a, b);

dbms\_output.put\_line(' Maximum of (23,45): ' || c);

END;

/

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

|  |
| --- |
| Maximum of (23,45): 45  PL/SQL procedure successfully completed. |

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**.

You can name a cursor so that it could be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of [cursors](https://www.tutorialspoint.com/plsql/plsql_cursors.htm) −

* Implicit [cursors](https://www.tutorialspoint.com/plsql/plsql_cursors.htm)
* Explicit [cursors](https://www.tutorialspoint.com/plsql/plsql_cursors.htm)

## Implicit [Cursors](https://www.tutorialspoint.com/plsql/plsql_cursors.htm)

Implicit [cursors](https://www.tutorialspoint.com/plsql/plsql_cursors.htm) 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](https://www.tutorialspoint.com/plsql/plsql_cursors.htm) 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**. The SQL cursor has additional attributes, **%BULK\_ROWCOUNT** and **%BULK\_EXCEPTIONS**, designed for use with the **FORALL** statement. The following table provides the description of the most used attributes −

|  |  |
| --- | --- |
| **S.No** | **Attribute & Description** |
| 1 | **%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. |
| 2 | **%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. |
| 3 | **%ISOPEN**  Always returns FALSE for implicit [cursors](https://www.tutorialspoint.com/plsql/plsql_cursors.htm), because Oracle closes the SQL cursor automatically after executing its associated SQL statement. |
| 4 | **%ROWCOUNT**  Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement. |

Any SQL cursor attribute will be accessed as **sql%attribute\_name** as shown below in the example.

### **Example**

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 will update the table and increase the salary of each customer by 500 and use the **SQL%ROWCOUNT** attribute to determine the number of rows affected −

DECLARE

total\_rows number(2);

BEGIN

UPDATE customers

SET salary = salary + 500;

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;

/

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

|  |
| --- |
| 6 customers selected  PL/SQL procedure successfully completed. |

If you check the records in customers table, you will find that the rows have been updated −

Select \* from customers;

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2500.00 |

| 2 | Khilan | 25 | Delhi | 2000.00 |

| 3 | kaushik | 23 | Kota | 2500.00 |

| 4 | Chaitali | 25 | Mumbai | 7000.00 |

| 5 | Hardik | 27 | Bhopal | 9000.00 |

| 6 | Komal | 22 | MP | 5000.00 |

+----+----------+-----+-----------+----------+

## Explicit [Cursors](https://www.tutorialspoint.com/plsql/plsql_cursors.htm)

Explicit [cursors](https://www.tutorialspoint.com/plsql/plsql_cursors.htm) are programmer-defined [cursors](https://www.tutorialspoint.com/plsql/plsql_cursors.htm) 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

## Declaring the Cursor

Declaring the cursor defines the cursor with a name and the associated SELECT statement. For example −

CURSOR c\_customers IS

SELECT id, name, address FROM customers;

## Opening the Cursor

Opening the cursor allocates the memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open the above defined cursor as follows −

OPEN c\_customers;

## Fetching the Cursor

Fetching the cursor involves accessing one row at a time. For example, we will fetch rows from the above-opened cursor as follows −

FETCH c\_customers INTO c\_id, c\_name, c\_addr;

## Closing the Cursor

Closing the cursor means releasing the allocated memory. For example, we will close the above-opened cursor as follows −

CLOSE c\_customers;

### **Example**

Following is a complete example to illustrate the concepts of explicit [cursors](https://www.tutorialspoint.com/plsql/plsql_cursors.htm) &minua;

DECLARE

c\_id customers.id%type;

c\_name customerS.No.ame%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;

/

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

|  |
| --- |
| 1 Ramesh Ahmedabad  2 Khilan Delhi  3 kaushik Kota  4 Chaitali Mumbai  5 Hardik Bhopal  6 Komal MP    PL/SQL procedure successfully completed. |