

# PL/SQL Introduction

Last Updated: 03-04-2018

PL/SQL is a block structured language that enables developers to combine the power of SQL with procedural statements. All the statements of a block are passed to oracle engine all at once which increases processing speed and decreases the traffic.

## **Disadvantages of SQL:**

- SQL doesn't provide the programmers with a technique of condition checking, looping and branching.
- SQL statements are passed to Oracle engine one at a time which increases traffic and decreases speed.
- SQL has no facility of error checking during manipulation of data.

## **Features of PL/SQL:**

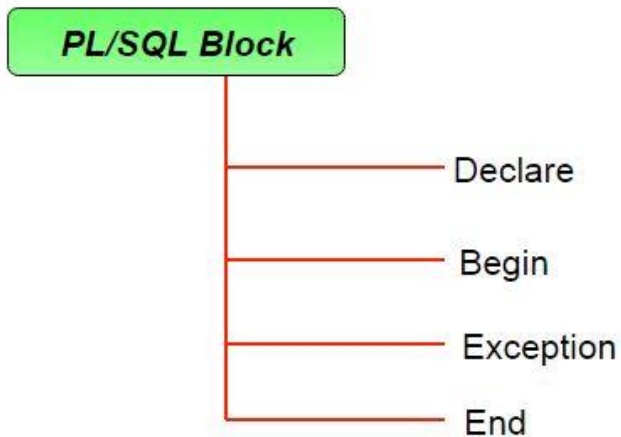
1. PL/SQL is basically a procedural language, which provides the functionality of decision making, iteration and many more features of procedural programming languages.
2. PL/SQL can execute a number of queries in one block using single command.
3. One can create a PL/SQL unit such as procedures, functions, packages, triggers, and types, which are stored in the database for reuse by applications.
4. PL/SQL provides a feature to handle the exception which occurs in PL/SQL block known as exception handling block.
5. Applications written in PL/SQL are portable to computer hardware or operating system where Oracle is operational.
6. PL/SQL Offers extensive error checking.

### Differences between SQL and PL/SQL:

SQL	PL/SQL
SQL is a single query that is used to perform DML and DDL operations.	PL/SQL is a block of codes that used to write the entire program blocks/ procedure/ function, etc.
It is declarative, that defines what needs to be done, rather than how things need to be done.	PL/SQL is procedural that defines how the things needs to be done.
Execute as a single statement.	Execute as a whole block.
Mainly used to manipulate data.	Mainly used to create an application.
Cannot contain PL/SQL code in it.	It is an extension of SQL, so it can contain SQL inside it.

### Structure of PL/SQL Block:

PL/SQL extends SQL by adding constructs found in procedural languages, resulting in a structural language that is more powerful than SQL. The basic unit in PL/SQL is a block. All PL/SQL programs are made up of blocks, which can be nested within each other.



Typically, each block performs a logical action in the program. A block has the following structure:

```
DECLARE
    declaration statements;

BEGIN
    executable statements

EXCEPTIONS
    exception handling statements

END;
```

- 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.

## PL/SQL identifiers

There are several PL/SQL identifiers such as variables, constants, procedures, cursors, triggers etc.

### 1. **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 ];
```

Example to show how to declare variables in PL/SQL :

```
SQL> SET SERVEROUTPUT ON;
```

```
SQL> DECLARE
```

```
    var1 INTEGER;
```

```
    var2 REAL;
```

```
    var3 varchar2(20) ;
```

```
BEGIN
```

```
    null;
```

```
END;
```

```
/
```

Output:

```
PL/SQL procedure successfully completed.
```

### **Explanation:**

- **SET SERVEROUTPUT ON:** It is used to display the buffer used by the dbms\_output.
- **var1 INTEGER :** 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.
- **PL/SQL procedure successfully completed.:** It is displayed when the code is compiled and executed successfully.
- **Slash (/) after END::** The slash (/) tells the SQL\*Plus to execute the block.

### 1.1) INITIALISING VARIABLES:

The variables can also be initialised just like in other programming languages. Let us see an example for the same:

```
SQL> SET SERVEROUTPUT ON;
```

```
SQL> DECLARE
```

```
    var1 INTEGER := 2 ;
```

```
    var3 varchar2(20) := 'welcome' ;
```

```
BEGIN
```

```
    null;
```

```
END;
```

```
/
```

Output:

```
PL/SQL procedure successfully completed.
```

***Explanation:***

- **Assignment operator (:=)** : It is used to assign a value to a variable.

### 2. Displaying Output:

The outputs are displayed by using DBMS\_OUTPUT which is a built-in package that enables the user to display output, debugging information, and send messages from PL/SQL blocks, subprograms, packages, and triggers. Let us see an example to see how to display a message using PL/SQL :

```
SQL> SET SERVEROUTPUT ON;
```

```
SQL> DECLARE
```

```
    var varchar2(40) := 'Welcome to PLSQL' ;
```

```
BEGIN
```

```
    dbms_output.put_line(var) ;
```

```
END;
```

```
/
```

Output:

```
Welcome to PLSQL
```

```
PL/SQL procedure successfully completed.
```

***Explanation:***

- *dbms\_output.put\_line* : This command is used to direct the PL/SQL output to a screen.

### 3. Using Comments:

Like in many other programming languages, in PL/SQL also, comments can be put within the code which has no effect in the code. There are two syntaxes to create comments in PL/SQL :

- **Single Line Comment:** To create a single line comment , the symbol `--` is used.
- **Multi Line Comment:** To create comments that span over several lines, the symbol `/*` and `*/` is used.

Example to show how to create comments in PL/SQL :

```
SQL> SET SERVEROUTPUT ON;

SQL> DECLARE

    -- I am a comment, so i will be ignored.

    var varchar2(40) := ' Welcome to PLSQL ';

BEGIN

    dbms_output.put_line(var);

END;

/
```

Output:

```
Welcome to PLSQL
PL/SQL procedure successfully completed.
```

### 4. Taking input from user:

Just like in other programming languages, in PL/SQL also, we can take input from the user and store it in a variable. Let us see an example to show how to take input from users in PL/SQL:

```
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: 'Hello'
old   3: b varchar2(30) := &b;
new   3: b varchar2(30) := 'Hello';

PL/SQL procedure successfully completed.

```

5. **(\*\*\*)** *Let us see an example on PL/SQL to demonstrate all above concepts in one single block of code.*

**filter\_none**

--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.

### 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.

## Cursors in PL/SQL

### Cursor in SQL

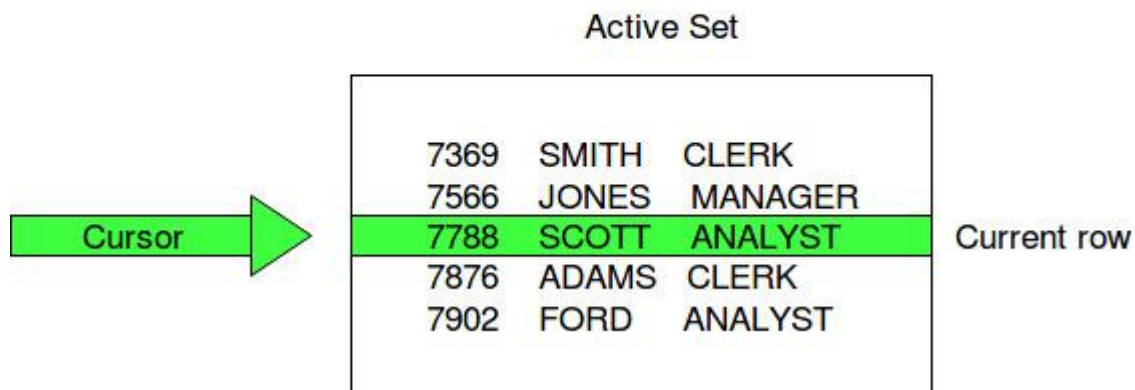
To execute SQL statements, a work area is used by the Oracle engine for its internal processing and storing the information. This work area is private to SQL's operations. The 'Cursor' is the PL/SQL construct that allows the user to name the work area and access the stored information in it.

### Use of Cursor

The major function of a cursor is to retrieve data, one row at a time, from a result set, unlike the SQL commands which operate on all the rows in the result set at one time. Cursors are used when the user needs to update records in a singleton fashion or in a row by row manner, in a database table.

The Data that is stored in the Cursor is called the *Active Data Set*. Oracle DBMS has another predefined area in the main memory Set, within which the cursors are opened. Hence the size of the cursor is limited by the size of this pre-defined area.

## Cursor Functions



### Cursor Actions

- **Declare Cursor:** A cursor is declared by defining the SQL statement that returns a result set.
- **Open:** A Cursor is opened and populated by executing the SQL statement defined by the cursor.
- **Fetch:** When the cursor is opened, rows can be fetched from the cursor one by one or in a block to perform data manipulation.
- **Close:** After data manipulation, close the cursor explicitly.
- **Deallocate:** Finally, delete the cursor definition and release all the system resources associated with the cursor.



## Types of Cursors

Cursors are classified depending on the circumstances in which they are opened.

- **Implicit Cursor:** If the Oracle engine opened a cursor for its internal processing it is known as an Implicit Cursor. It is created “automatically” for the user by Oracle when a query is executed and is simpler to code.
- **Explicit Cursor:** A Cursor can also be opened for processing data through a PL/SQL block, on demand. Such a user-defined cursor is known as an Explicit Cursor.

### Explicit cursor

An explicit cursor is defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row. A suitable name for the cursor.

General syntax for creating a cursor:

```
CURSOR cursor_name IS select_statement;
```

cursor\_name – A suitable name for the cursor.

select\_statement – A select query which returns multiple rows

### How to use Explicit Cursor?

There are four steps in using an Explicit Cursor.

1. DECLARE the cursor in the Declaration section.
2. OPEN the cursor in the Execution Section.
3. FETCH the data from the cursor into PL/SQL variables or records in the Execution Section.
4. CLOSE the cursor in the Execution Section before you end the PL/SQL Block.

### Syntax:

```
DECLARE variables;
```

```
records;
```

```
create a cursor;
```

```
BEGIN
```

```
OPEN cursor;
```

```
FETCH cursor;
```

```
process the records;
```

```
CLOSE cursor;
```

```
END;
```

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 –

- Implicit cursors
- Explicit cursors

## Implicit Cursors

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

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

In PL/SQL, you can refer to the most recent implicit cursor as the **SQL cursor**, which always has attributes such as **%FOUND**, **%ISOPEN**, **%NOTFOUND**, and **%ROWCOUNT**. 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	<b>%FOUND</b> 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	<b>%NOTFOUND</b> 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	<b>%ISOPEN</b> Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.

4	<b>%ROWCOUNT</b> 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

Explicit cursors are programmer-defined cursors for gaining more control over the **context area**. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.

The syntax for creating an explicit cursor is –

```
CURSOR cursor_name IS select_statement;
```

Working with an explicit cursor includes the following steps –

- Declaring the cursor for initializing the memory
- Opening the cursor for allocating the memory
- Fetching the cursor for retrieving the data
- Closing the cursor to release the allocated memory

## 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 &minua;

```
DECLARE
    c_id customers.id%type;
    c_name customer.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;
/
```

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.

## Sum Of Two Numbers in PL/SQL

Here, first, we take three variables x, y, and z and assign the value in x and y and after addition of both the numbers, we assign the resultant value to z and print z.

Examples:

Input : 15 25

Output : 40

Input : 250 400

Output : 650

Below is the required implementation:

```
declare
    -- declare variable x, y
    -- and z of datatype number
x number(5);
y number(5);
z number(7);

begin

    -- Here we Assigning 10 into x
x:=10;

    -- Assigning 20 into x
y:=20;

    -- Assigning sum of x and y into z
z:=x+y;

    // Print the Result
dbms_output.put_line('Sum is '||z);
end;

/

-- Program End
```

### Output :

```
Sum is 30
```

## Trigger

Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events –

- A **database manipulation (DML)** statement (DELETE, INSERT, or UPDATE)
- A **database definition (DDL)** statement (CREATE, ALTER, or DROP).
- A **database operation** (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

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

### Benefits of Triggers

Triggers can be written for the following purposes –

- Generating some derived column values automatically

- Enforcing referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

## Creating Triggers

The syntax for creating a trigger is –

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

Where,

- CREATE [OR REPLACE] TRIGGER trigger\_name – Creates or replaces an existing trigger with the *trigger\_name*.
- {BEFORE | AFTER | INSTEAD OF} – This specifies when the trigger will be executed. The INSTEAD OF clause is used for creating trigger on a view.
- {INSERT [OR] | UPDATE [OR] | DELETE} – This specifies the DML operation.
- [OF col\_name] – This specifies the column name that will be updated.
- [ON table\_name] – This specifies the name of the table associated with the trigger.
- [REFERENCING OLD AS o NEW AS n] – This allows you to refer new and old values for various DML statements, such as INSERT, UPDATE, and DELETE.
- [FOR EACH ROW] – This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
- WHEN (condition) – This provides a condition for rows for which the trigger would fire. This clause is valid only for row-level triggers.

### Example

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

```
Select * from customers;
```

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

The following program creates a **row-level** trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values –

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON customers
FOR EACH ROW
WHEN (NEW.ID > 0)
DECLARE
    sal_diff number;
BEGIN
    sal_diff := :NEW.salary - :OLD.salary;
    dbms_output.put_line('Old salary: ' || :OLD.salary);
    dbms_output.put_line('New salary: ' || :NEW.salary);
    dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/
```

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

Trigger created.

The following points need to be considered here –

- OLD and NEW references are not available for table-level triggers, rather you can use them for record-level triggers.
- If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.
- The above trigger has been written in such a way that it will fire before any DELETE or INSERT or UPDATE operation on the table, but you can write your trigger on a single or multiple operations, for example BEFORE DELETE, which will fire whenever a record will be deleted using the DELETE operation on the table.

## Triggering a Trigger

Let us perform some DML operations on the CUSTOMERS table. Here is one INSERT statement, which will create a new record in the table –



```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (7, 'Kriti', 22, 'HP', 7500.00 );
```

When a record is created in the CUSTOMERS table, the above create trigger, **display\_salary\_changes** will be fired and it will display the following result –

```
Old salary:
New salary: 7500
Salary difference:
```

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

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

When a record is updated in the CUSTOMERS table, the above create trigger, **display\_salary\_changes** will be fired and it will display the following result –

```
Old salary: 1500
New salary: 2000
Salary difference: 500
```

## PL/SQL Procedure

**Summary:** in this tutorial, you will learn how to create, compile, and execute a PL/SQL procedure from the Oracle SQL Developer.

### PL/SQL procedure syntax

A PL/SQL procedure is a reusable unit that encapsulates specific business logic of the application. Technically speaking, a PL/SQL procedure is a named **block** stored as a schema object in the Oracle Database.

The following illustrates the basic syntax of creating a procedure in PL/SQL:

```
CREATE [OR REPLACE ] PROCEDURE procedure_name (parameter_list)
IS
```

```
[declaration statements]
```

```
BEGIN
```

```
[execution statements]
```

## EXCEPTION

[exception handler]

END [procedure\_name];

### PL/SQL procedure header

A procedure begins with a header that specifies its name and an optional parameter list.

Each parameter can be in either **IN**, **OUT**, or **INOUT** mode. The parameter mode specifies whether a parameter can be read from or write to.

**IN**

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.

**OUT**

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.

**INOUT**

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

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

### PL/SQL procedure body

Similar to an [anonymous block](#), the procedure body has three parts. The executable part is mandatory whereas the declarative and exception-handling parts are optional. The executable part must contain at least one executable statement.

#### 1) Declarative part

In this part, you can declare [variables](#), [constants](#), [cursors](#), etc. Unlike an [anonymous block](#), a declaration part of a procedure does not start with the `DECLARE` keyword.

## 2) Executable part

This part contains one or more statements that implement specific business logic. It might contain only a [NULL statement](#).

## 3) Exception-handling part

This part contains the code that handles [exceptions](#).

### Creating a PL/SQL procedure example

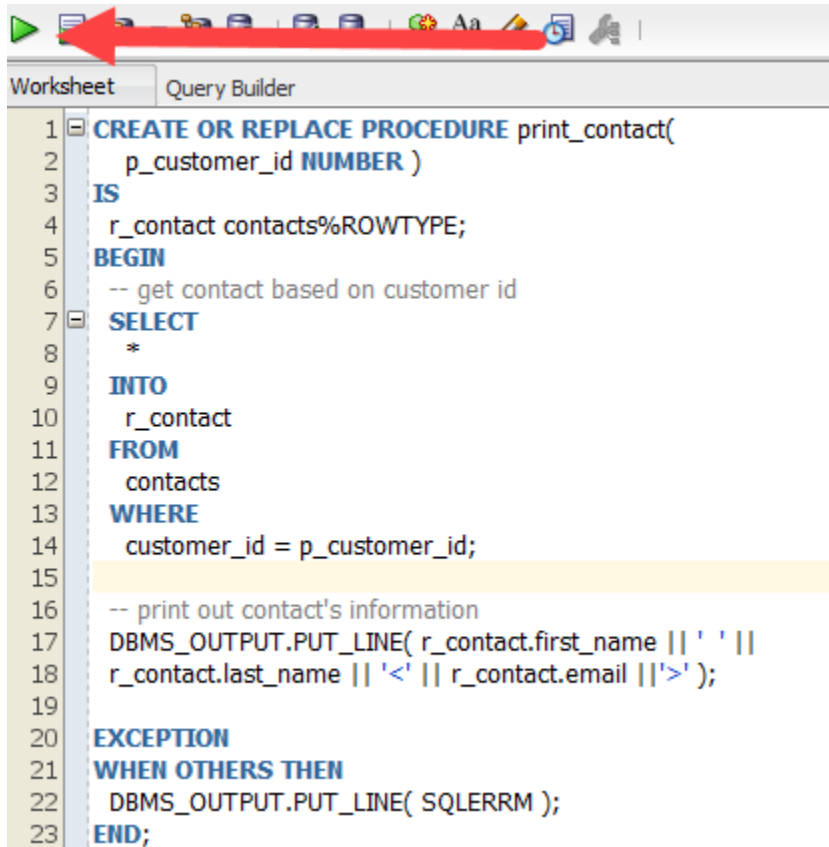
The following procedure accepts a customer id and prints out the customer's contact information including first name, last name, and email:

```
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 || '>' );

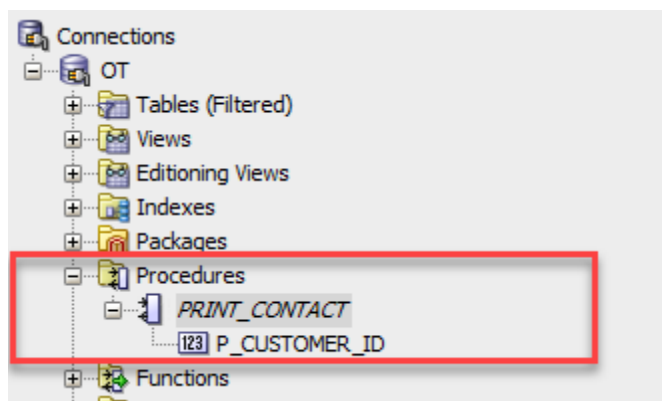
EXCEPTION
    WHEN OTHERS THEN
        dbms_output.put_line( SQLERRM );
END;
```

To compile the procedure, you click the **Run Statement** button as shown in the following picture:



```
1 CREATE OR REPLACE PROCEDURE print_contact(  
2   p_customer_id NUMBER )  
3 IS  
4   r_contact contacts%ROWTYPE;  
5 BEGIN  
6   -- get contact based on customer id  
7   SELECT  
8     *  
9   INTO  
10    r_contact  
11  FROM  
12    contacts  
13  WHERE  
14    customer_id = p_customer_id;  
15  
16  -- print out contact's information  
17  DBMS_OUTPUT.PUT_LINE( r_contact.first_name || ' ' ||  
18    r_contact.last_name || '<' || r_contact.email || '>' );  
19  
20 EXCEPTION  
21 WHEN OTHERS THEN  
22   DBMS_OUTPUT.PUT_LINE( SQLERRM );  
23 END;
```

If the procedure is compiled successfully, you will find the new procedure under the **Procedures** node as shown below:



### Executing a PL/SQL procedure

The following shows the syntax for executing a procedure:

```
EXECUTE procedure_name( arguments );
```

Or

```
EXEC procedure_name( arguments );
```

For example, to execute the `print_contact` procedure that prints the contact information of customer id 100, you use the following statement:

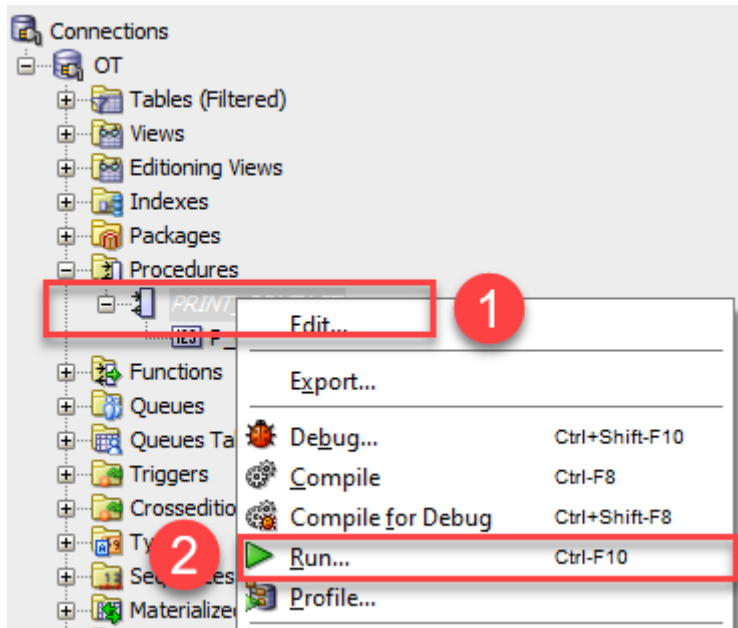
```
EXEC print_contact(100);
```

Here is the output:

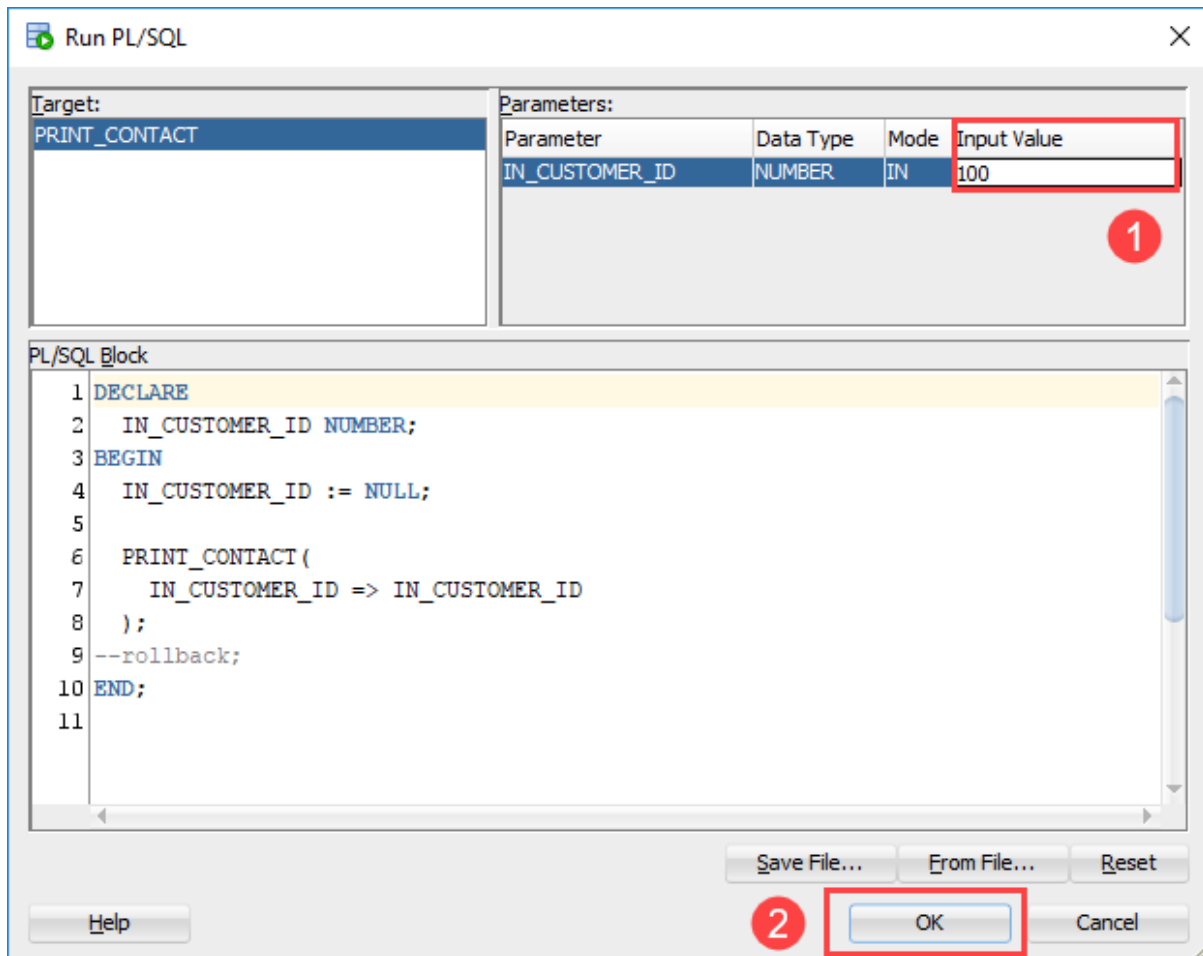
Elisha Lloyd<elisha.lloyd@verizon.com>

You can also execute a procedure from the Oracle SQL Developer using the following steps:

- 1) Right-click the procedure name and choose **Run...** menu item



- 2) Enter a value for the `in_customer_id` parameter and click **OK** button.



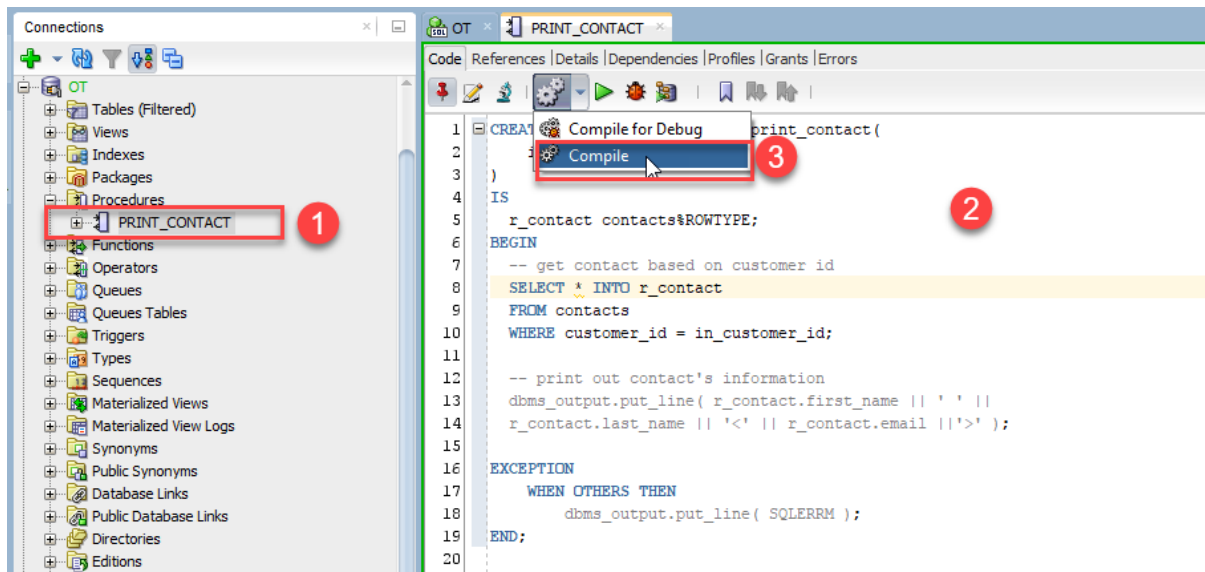
3) The following shows the result

```
Connecting to the database Local.
Elisha Lloyd<elisha.lloyd@verizon.com>
Process exited.
Disconnecting from the database Local.
```

### Editing a procedure

To change the code of an existing procedure, you can follow these steps:

- Step 1. Click the procedure name under **Procedures** node.
- Step 2. Edit the code of the procedure.
- Step 3. Click Compile menu option to recompile the procedure.



## Removing a procedure

To delete a procedure, you use the `DROP PROCEDURE` followed by the procedure's name that you want to drop as shown in the following syntax:

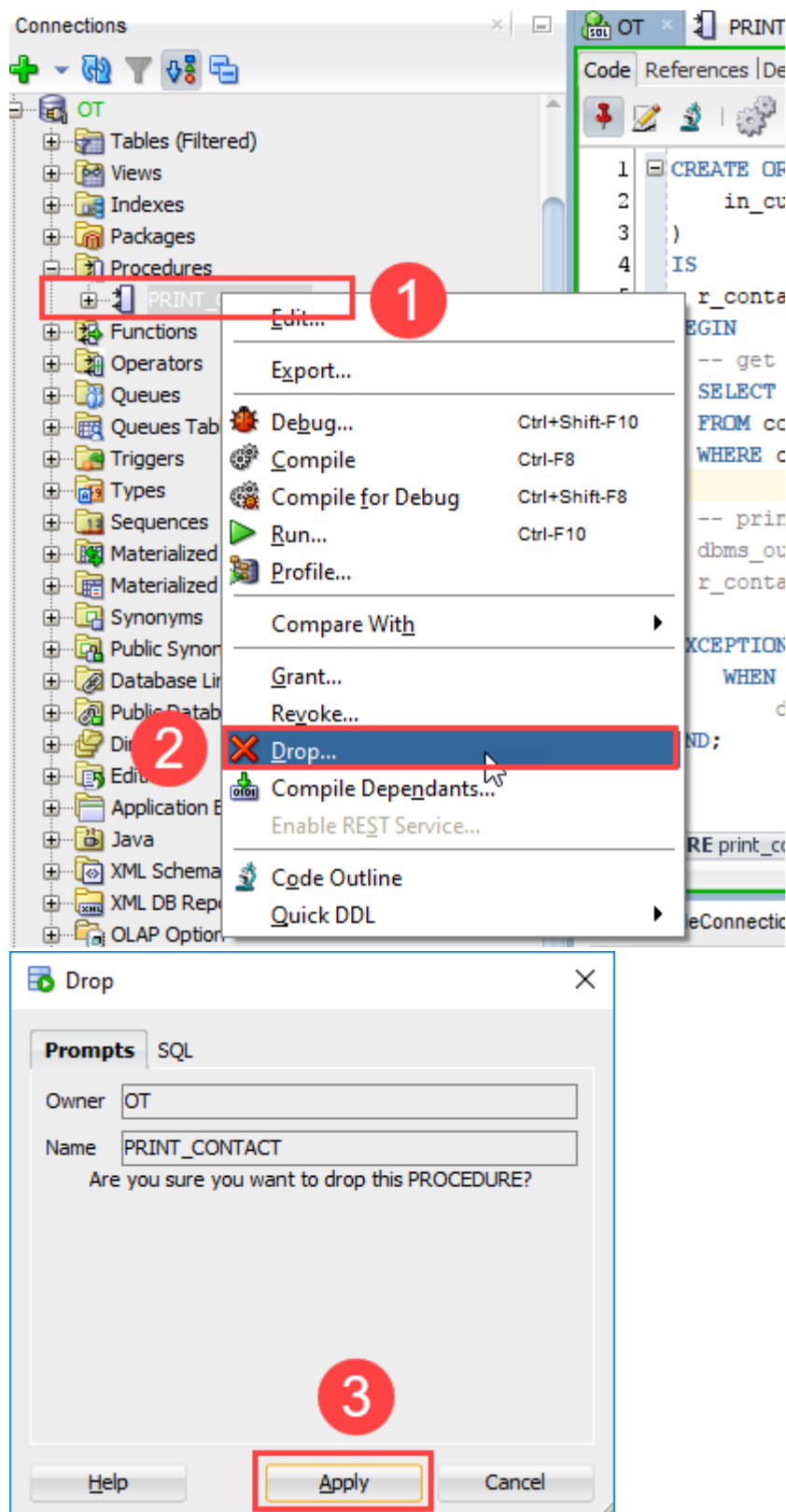
```
DROP PROCEDURE procedure_name;
```

For example, the following statement drops the `print_contact` procedure :

```
DROP PROCEDURE print_contact;
```

The following illustrates the steps of dropping a procedure using SQL Developer:

- Step 1. Right click on the procedure name that you want to drop
- Step 2. Choose the **Drop...** menu option
- Step 3. In the Prompts dialog, click the **Apply** button to remove the procedure.



In this tutorial, you have learned how to create a PL/SQL procedure and execute it from Oracle SQL Developer.