

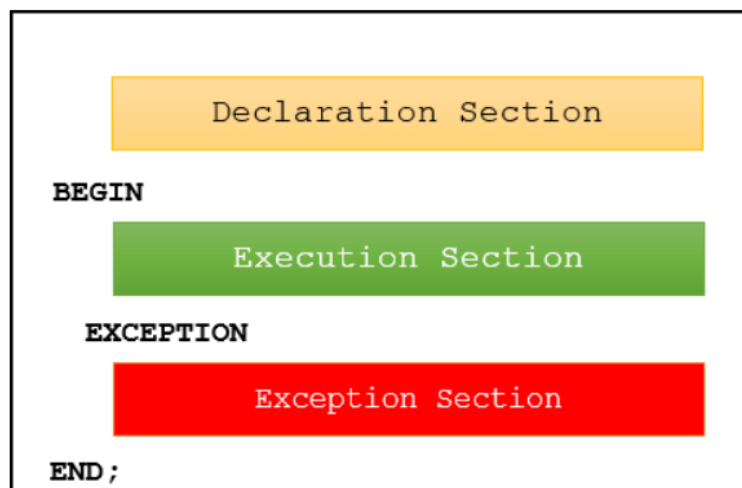
PL/SQL anonymous block overview

PL/SQL is a block-structured language whose code is organized into blocks. A PL/SQL block consists of three sections: declaration, executable, and exception-handling sections. In a block, the executable section is mandatory while the declaration and exception-handling sections are optional.

A PL/SQL block has a name. [Functions](#) or [Procedures](#) is an example of a named block. A named block is stored into the Oracle Database server and can be reused later.

A block without a name is an anonymous block. An anonymous block is not saved in the Oracle Database server, so it is just for one-time use. However, PL/SQL anonymous blocks can be useful for testing purposes.

The following picture illustrates the structure of a PL/SQL block:



1) Declaration section

A PL/SQL block has a declaration section where you [declare variables](#), allocate memory for [cursors](#), and define data types.

2) Executable section

A PL/SQL block has an executable section. An executable section starts with the keyword `BEGIN` and ends with the keyword `END`. The executable section must have at least one executable statement, even if it is the [NULL statement](#) which does nothing.

3) Exception-handling section

A PL/SQL block has an exception-handling section that starts with the keyword `EXCEPTION`. The exception-handling section is where you catch and handle exceptions raised by the code in the execution section.

Note a block itself is an executable statement, therefore you can nest a block within other blocks.

PL/SQL anonymous block example

The following example shows a simple PL/SQL anonymous block with one executable section.

```
BEGIN  
    DBMS_OUTPUT.put_line ('Hello World!');  
END;
```

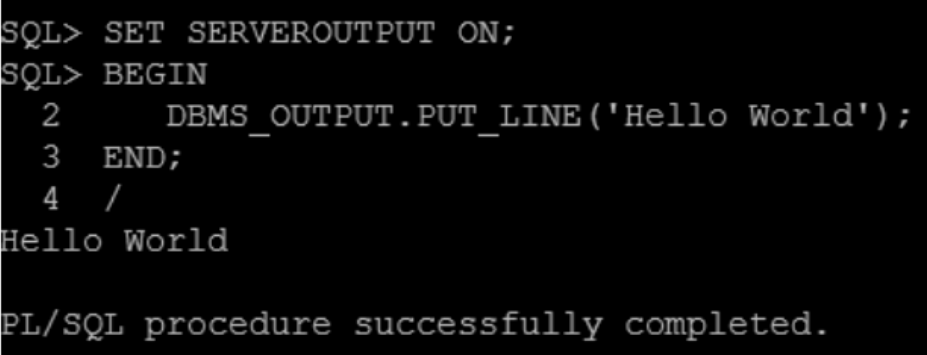
Code language: SQL (Structured Query Language) (sql)

The executable section calls the DBMS_OUTPUT.PUT_LINE procedure to display the "Hello World" message on the screen.

Execute a PL/SQL anonymous block using SQL*Plus

Once you have the code of an anonymous block, you can execute it using SQL*Plus, which is a command-line interface for executing SQL statement and PL/SQL blocks provided by Oracle Database.

The following picture illustrates how to execute a PL/SQL block using SQL*Plus:



```
SQL> SET SERVEROUTPUT ON;  
SQL> BEGIN  
  2     DBMS_OUTPUT.PUT_LINE('Hello World');  
  3 END;  
  4 /  
Hello World  
  
PL/SQL procedure successfully completed.
```

First, connect to the Oracle Database server using a username and password.

Second, turn on the server output using the SET SERVEROUTPUT ON command so that the DBMS_OUTPUT.PUT_LINE procedure will display text on the screen.

Third, type the code of the block and enter a forward slash (/) to instruct SQL*Plus to execute the block. Once you type the forward-slash (/), SQL*Plus will execute the block and display the Hello World message on the screen as shown in the illustrations.

Note that you must execute SET SERVEROUTPUT ON command in every session that you connect to the Oracle Database in order to show the message using the DBMS_OUTPUT.PUT_LINE procedure.

In PL/SQL, a variable is named storage location that stores a value of a particular data type. The value of the variable changes through the program. Before using a variable, you must declare it in the declaration section of a block.

Declaring variables

The syntax for a variable declaration is as follows:

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

Code language: SQL (Structured Query Language) (sql)

In this syntax:

- First, specify the name of the variable. The name of the variable should be as descriptive as possible, e.g., `I_total_sales`, `I_credit_limit`, and `I_sales_revenue`.
- Second, choose an appropriate [data type](#) for the variable, depending on the kind of value which you want to store, for example, number, character, Boolean, and datetime.

By convention, local variable names should start with `I_` and global variable names should have a prefix of `g_`.

The following example declares three variables `I_total_sales`, `I_credit_limit`, and `I_contact_name`:

```
DECLARE  
  I_total_sales NUMBER(15,2);  
  I_credit_limit NUMBER (10,0);  
  I_contact_name VARCHAR2(255);  
BEGIN  
  NULL;  
END;
```

Code language: SQL (Structured Query Language) (sql)

Default values

PL/SQL allows you to set a default value for a variable at the declaration time. To assign a default value to a variable, you use the assignment operator (`:=`) or the `DEFAULT` keyword.

The following example declares a variable named `I_product_name` with an initial value 'Laptop':

```
DECLARE  
  I_product_name VARCHAR2( 100 ) := 'Laptop';  
BEGIN  
  NULL;
```

END;

Code language: SQL (Structured Query Language) (sql)

It is equivalent to the following block:

DECLARE

l_product_name VARCHAR2(100) DEFAULT 'Laptop';

BEGIN

NULL;

END;

Code language: SQL (Structured Query Language) (sql)

In this example, instead of using the assignment operator `:=`, we used the `DEFAULT` keyword to initialize a variable.

NOT NULL constraint

If you impose the [NOT NULL](#) constraint on a value, then the variable cannot accept a NULL value. Besides, a variable declared with the `NOT NULL` must be initialized with a non-null value. Note that PL/SQL treats a zero-length string as a NULL value.

The following example first declares a variable named `l_shipping_status` with the `NOT NULL` constraint. Then, it assigns the variable a zero-length string.

DECLARE

l_shipping_status VARCHAR2(25) NOT NULL := 'Shipped';

BEGIN

l_shipping_status := '';

END;

Code language: SQL (Structured Query Language) (sql)

PL/SQL issued the following error:

ORA-06502: PL/SQL: numeric or value error

Code language: SQL (Structured Query Language) (sql)

Because the variable `l_shipping_status` declared with the `NOT NULL` constraint, it could not accept a NULL value or zero-length string in this case.

Variable assignments

To assign a value to a variable, you use the assignment operator (`:=`), for example:

DECLARE

l_customer_group VARCHAR2(100) := 'Silver';

BEGIN

```
I_customer_group := 'Gold';  
DBMS_OUTPUT.PUT_LINE(I_customer_group);  
END;
```

Code language: SQL (Structured Query Language) (sql)

You can assign a value of a variable to another as shown in the following example:

```
DECLARE  
  I_business_parter VARCHAR2(100) := 'Distributor';  
  I_lead_for VARCHAR2(100);  
BEGIN  
  I_lead_for := I_business_parter;  
  DBMS_OUTPUT.PUT_LINE(I_lead_for);  
END;
```

What is an Oracle trigger

A trigger is a named PL/SQL block stored in the Oracle Database and executed automatically when a triggering event takes place. The event can be any of the following:

A data manipulation language (DML) statement executed against a table e.g., INSERT, UPDATE, or DELETE. For example, if you define a trigger that fires before an INSERT statement on the customers table, the trigger will fire once before a new row is inserted into the customers table.

A data definition language (DDL) statement executes e.g., CREATE or ALTER statement. These triggers are often used for auditing purposes to record changes of the schema.

A system event such as startup or shutdown of the Oracle Database.

A user event such as login or logout.

The act of executing a trigger is also known as firing a trigger. We say that the trigger is fired.

Oracle trigger usages

Oracle triggers are useful in many cases such as the following:

Enforcing complex business rules that cannot be established using integrity constraint such as UNIQUE, NOT NULL, and CHECK.

Preventing invalid transactions.

Gathering statistical information on table accesses.

Generating value automatically for derived columns.

Auditing sensitive data.

How to create a trigger in Oracle

To create a new trigger in Oracle, you use the following CREATE TRIGGER statement:

```
CREATE [OR REPLACE] TRIGGER trigger_name  
{BEFORE | AFTER } triggering_event ON table_name  
[FOR EACH ROW]  
[FOLLOWS | PRECEDES another_trigger]  
[ENABLE / DISABLE ]
```

```
[WHEN condition]
DECLARE
  declaration statements
BEGIN
  executable statements
EXCEPTION
  exception_handling statements
END;
```

Code language: SQL (Structured Query Language) (sql)

As you can see, the trigger body has the same structure as an anonymous PL/SQL block.

1) CREATE OR REPLACE

The CREATE keyword specifies that you are creating a new trigger. The OR REPLACE keywords are optional. They are used to modify an existing trigger.

Even though the OR REPLACE keywords are optional, they appear with the CREATE keyword in most cases.

For example, if today you define a new trigger named trigger_example:

```
CREATE TRIGGER trigger_example
```

```
...
```

Code language: SQL (Structured Query Language) (sql)

And on the next day, you decide to modify this trigger.

If you do not include the OR REPLACE keywords, you will receive an error message indicating that the name of your trigger is already used by another object:

```
CREATE TRIGGER trigger_example
```

```
...
```

Code language: SQL (Structured Query Language) (sql)

Therefore, the CREATE OR REPLACE keywords will replace an existing trigger if it already exists and create a new trigger if the trigger does not:

```
CREATE OR REPLACE trigger_example
```

```
....
```

Code language: SQL (Structured Query Language) (sql)

2) Trigger name

Specify the name of the trigger that you want to create after the CREATE OR REPLACE keywords.

3) BEFORE | AFTER

The BEFORE or AFTER option specifies when the trigger fires, either before or after a triggering event e.g., INSERT, UPDATE, or DELETE

4) ON table_name

The table_name is the name of the table associated with the trigger.

5) FOR EACH ROW

The clause FOR EACH ROW specifies that the trigger is a row-level trigger. A row-level trigger fires once for each row inserted, updated, or deleted.

Besides the row-level triggers, we have statement-level triggers. A statement-trigger fire once regardless of the number of rows affected by the triggering event. If you omit the FOR EACH ROW clause, the CREATE TRIGGER statement will create a statement-level trigger.

6) ENABLE / DISABLE

The ENABLE / DISABLE option specifies whether the trigger is created in the enabled or disabled state. Note that if a trigger is disabled, it is not fired when the triggering event occurs.

By default, if you don't specify the clause ENABLE / DISABLE , the trigger is created with the enabled state.

7) FOLLOWS | PRECEDES another_trigger

For each triggering event e.g., INSERT, UPDATE, or DELETE, you can define multiple triggers to fire. In this case, you need to specify the firing sequence using the FOLLOWS or PRECEDES option.

Let's create a trigger to see understand how it works.

Creating an Oracle trigger example

Suppose we want to record actions against the customers table whenever a customer is updated or deleted. In order to do this:

First, create a new table for recording the UPDATE and DELETE events:

```
CREATE TABLE audits (  
  audit_id    NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,  
  table_name  VARCHAR2(255),  
  transaction_name VARCHAR2(10),  
  by_user     VARCHAR2(30),  
  transaction_date DATE  
);
```

Code language: SQL (Structured Query Language) (sql)

Second, create a new trigger associated with the customers table:

```
CREATE OR REPLACE TRIGGER customers_audit_trg  
  AFTER  
  UPDATE OR DELETE  
  ON customers  
  FOR EACH ROW  
DECLARE  
  l_transaction VARCHAR2(10);  
BEGIN  
  -- determine the transaction type  
  l_transaction := CASE
```

```
    WHEN UPDATING THEN 'UPDATE'  
    WHEN DELETING THEN 'DELETE'  
END;
```

-- insert a row into the audit table

```
INSERT INTO audits (table_name, transaction_name, by_user, transaction_date)  
VALUES('CUSTOMERS', l_transaction, USER, SYSDATE);
```

```
END;
```

```
/
```

Code language: SQL (Structured Query Language) (sql)

The following clause:

```
AFTER UPDATE OR DELETE ON customers
```

Code language: SQL (Structured Query Language) (sql)

will fire the trigger after a row in the table customers is updated or deleted.

Inside the trigger, we determine the current action whether it is UPDATE or DELETE and insert a row into the audits table.

The following statement updates the credit limit of the customer 10 to 2000.

```
UPDATE
```

```
  customers
```

```
SET
```

```
  credit_limit = 2000
```

```
WHERE
```

```
  customer_id =10;
```

Code language: SQL (Structured Query Language) (sql)

Now, check the contents of the table audits to see if the trigger was fired:

```
SELECT * FROM audits;
```

Code language: SQL (Structured Query Language) (sql)

Here is the output:

Oracle Trigger - AFTER UPDATE example

As you can see clearly from the output, the trigger customers_audit_trg was fired so that we have a new row inserted into the audits table.

This DELETE statement deletes a row from the customers table.

```
DELETE FROM customers
```

```
WHERE customer_id = 10;
```

Code language: SQL (Structured Query Language) (sql)

And view the data of the audits table:

```
SELECT * FROM audits;
```

Code language: SQL (Structured Query Language) (sql)

Oracle Trigger - AFTER DELETE example

The output showed that a new row has been inserted. It means that the DELETE action fired the trigger customer_audit_trg.

In this tutorial, you have learned about Oracle triggers and how to create new triggers using the CREATE TRIGGER statement.

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
```

```
Code language: SQL (Structured Query Language) (sql)
[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;
```

Code language: SQL (Structured Query Language) (sql)

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

PL/SQL Procedure – compile

Creating a PL/SQL function

Similar to a procedure, a PL/SQL function is a reusable program unit stored as a schema object in the Oracle Database. The following illustrates the syntax for creating a function:

```
CREATE [OR REPLACE] FUNCTION function_name (parameter_list)
  RETURN return_type
IS
```

Code language: SQL (Structured Query Language) (sql)
[declarative section]

```
BEGIN
  [executable section]
```

```
[EXCEPTION]
  [exception-handling section]
```

```
END;
```

A function consists of a header and body.

The function header has the function name and a RETURN clause that specifies the datatype of the returned value. Each parameter of the function can be either in the IN, OUT, or INOUT mode. For more information on the parameter mode, check it out the PL/SQL procedure tutorial

The function body is the same as the procedure body which has three sections: declarative section, executable section, and exception-handling section.

The declarative section is between the IS and BEGIN keywords. It is where you declare variables, constants, cursors, and user-defined types.

The executable section is between the BEGIN and END keywords. It is where you place the executable statements. Unlike a procedure, you must have at least one RETURN statement in the executable statement.

The exception-handling section is where you put the exception handler code.

In these three sections, only the executable section is required, the others are optional.

PL/SQL function example

The following example creates a function that calculates total sales by year.

```
CREATE OR REPLACE FUNCTION get_total_sales(
  in_year PLS_INTEGER
)
RETURN NUMBER
IS
  l_total_sales NUMBER := 0;
BEGIN
  -- get total sales
  SELECT SUM(unit_price * quantity)
  INTO l_total_sales
```

```

FROM order_items
INNER JOIN orders USING (order_id)
WHERE status = 'Shipped'
GROUP BY EXTRACT(YEAR FROM order_date)
HAVING EXTRACT(YEAR FROM order_date) = in_year;

-- return the total sales
RETURN I_total_sales;
END;

```

Introduction to PL/SQL Exceptions

PL/SQL treats all errors that occur in an anonymous block, procedure, or function as exceptions. The exceptions can have different causes such as coding mistakes, bugs, even hardware failures.

It is not possible to anticipate all potential exceptions, however, you can write code to handle exceptions to enable the program to continue running as normal.

The code that you write to handle exceptions is called an exception handler.

A PL/SQL block can have an exception-handling section, which can have one or more exception handlers.

Here is the basic syntax of the exception-handling section:

```

BEGIN
  -- executable section
  ...
  -- exception-handling section
  EXCEPTION
    WHEN e1 THEN
      -- exception_handler1
    WHEN e2 THEN
      -- exception_handler1
    WHEN OTHERS THEN
      -- other_exception_handler
END;

```

Code language: SQL (Structured Query Language) (sql)

In this syntax, e1, e2 are exceptions.

When an exception occurs in the executable section, the execution of the current block stops and control transfers to the exception-handling section.

If the exception e1 occurred, the exception_handler1 runs. If the exception e2 occurred, the exception_handler2 executes. In case any other exception raises, then the other_exception_handler runs.

After an exception handler executes, control transfers to the next statement of the enclosing block. If there is no enclosing block, then the control returns to the invoker if the exception handler is in a

subprogram or host environment (SQL Developer or SQL*Plus) if the exception handler is in an anonymous block.

If an exception occurs but there is no exception handler, then the exception propagates, which we will discuss in the unhandled exception propagation tutorial.

PL/SQL exception examples

Let's take some examples of handling exceptions.

PL/SQL NO_DATA_FOUND exception example

The following block accepts a customer id as an input and returns the customer name :

```
DECLARE
  l_name customers.NAME%TYPE;
  l_customer_id customers.customer_id%TYPE := &customer_id;
BEGIN
  -- get the customer name by id
  SELECT name INTO l_name
  FROM customers
  WHERE customer_id = l_customer_id;

  -- show the customer name
  dbms_output.put_line('Customer name is ' || l_name);

END;
/
```

Code language: SQL (Structured Query Language) (sql)

If you execute the block and enter the customer id as zero, Oracle will issue the following error:

```
ORA-01403: no data found
```

Code language: SQL (Structured Query Language) (sql)

The ORA-01403 is a predefined exception.

Note that the following line does not execute at all because control transferred to the exception handling section.

```
dbms_output.put_line('Customer name is ' || l_name);
```

Code language: SQL (Structured Query Language) (sql)

To issue a more meaningful message, you can add an exception-handling section as follows:

```
DECLARE
  l_name customers.NAME%TYPE;
  l_customer_id customers.customer_id%TYPE := &customer_id;
BEGIN
  -- get the customer
  SELECT NAME INTO l_name
  FROM customers
  WHERE customer_id = l_customer_id;
```

```
-- show the customer name
dbms_output.put_line('customer name is ' || l_name);

EXCEPTION
  WHEN NO_DATA_FOUND THEN
    dbms_output.put_line('Customer ' || l_customer_id || ' does not exist');
END;
/
```