Oracle creates a memory area, known as context area, for processing an SQL statement, which contains all information needed for processing the statement, for example, number of rows processed, etc.

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 the attributes like %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:

|  |  |
| --- | --- |
| Attribute | Description |
| %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. |
| %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. |
| %ISOPEN | Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement. |
| %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 would update the table and increase 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 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 |

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

### Drawbacks of Implicit Cursors

Even if your query returns only a single row, you might still decide to use an explicit cursor. The implicit cursor has the following drawbacks:

* It is less efficient than an explicit cursor (in PL/SQL Release 2.2 and earlier)
* It is more vulnerable to data errors
* It gives you less programmatic control

**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 :

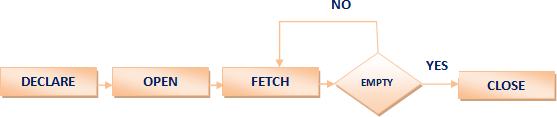
CURSOR cursor\_name IS select\_statement;

Working with an explicit cursor involves four steps:

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

## Working with PL/SQL Cursor

The following picture describes steps that you need to follow when you work with a PL/SQL cursor:

[](http://www.plsqltutorial.com/wp-content/uploads/2012/10/plsqlcursor.png)

**PL/SQL Cursor**

**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 memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open 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 above-opened cursor as follows:

CLOSE c\_customers;

**Example:**

Following is a complete example to illustrate the concepts of explicit cursors:

DECLARE

c\_idcustomers.id%type; (%type allows u to declare constants, variables)

c\_namecustomers.name%type;

c\_addrcustomers.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 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.

# PL/SQL - Records

A PL/SQL **record** is a data structure that can hold data items of different kinds. Records consist of different fields, similar to a row of a database table.

For example, you want to keep track of your books in a library. You might want to track the following attributes about each book like, Title, Author, Subject, Book ID. A record containing a field for each of these items allows treating a BOOK as a logical unit and allows you to organize and represent its information in a better way.

PL/SQL can handle the following types of records:

* Table-based
* Cursor-based records
* User-defined records

## Table-Based Records

The %ROWTYPE attribute enables a programmer to create **table-based** and**cursor-based** records.

The following example would illustrate the concept of **table-based** records. We will be using the CUSTOMERS table we had created and used in the previously:

DECLARE

customer\_reccustomers%rowtype;

BEGIN

SELECT \*into customer\_rec

FROM customers

WHERE id =5;

dbms\_output.put\_line('Customer ID: '|| customer\_rec.id);

dbms\_output.put\_line('Customer Name: '|| customer\_rec.name);

dbms\_output.put\_line('Customer Address: '||customer\_rec.address);

dbms\_output.put\_line('Customer Salary: '||customer\_rec.salary);

END;

/

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

Customer ID:5

CustomerName:Hardik

CustomerAddress:Bhopal

CustomerSalary:9000

PL/SQL procedure successfully completed.

## Cursor-Based Records

The following example would illustrate the concept of **cursor-based** records. We will be using the CUSTOMERS table we had created and used in the previous chapters:

DECLARE

CURSOR customer\_cur is

SELECT id,name,address

FROM customers;

customer\_rec customer\_cur%rowtype;

BEGIN

OPEN customer\_cur;

LOOP

FETCH customer\_cur into customer\_rec;

EXIT WHEN customer\_cur%notfound;

DBMS\_OUTPUT.put\_line(customer\_rec.id ||' '|| customer\_rec.name);

END LOOP;

END;

/

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

1Ramesh

2Khilan

3kaushik

4Chaitali

5Hardik

6Komal

PL/SQL procedure successfully completed.

## User-Defined Records

PL/SQL provides a user-defined record type that allows you to define different record structures. Records consist of different fields. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book:

* Title
* Author
* Subject
* Book ID

## Defining a Record

The record type is defined as:

TYPE type\_name IS RECORD

( field\_name1 datatype1 [NOT NULL][:= DEFAULT EXPRESSION],

field\_name2 datatype2 [NOT NULL][:= DEFAULT EXPRESSION],

...

field\_nameNdatatypeN[NOT NULL][:= DEFAULT EXPRESSION);

record-name type\_name;

Here is the way you would declare the Book record:

DECLARE

TYPE books IS RECORD

(title varchar(50),

author varchar(50),

subjectvarchar(100),

book\_id number);

book1 books;

book2 books;

## Accessing Fields

To access any field of a record, we use the dot (.) operator. The member access operator is coded as a period between the record variable name and the field that we wish to access. Following is the example to explain usage of record:

DECLARE

TYPE books is record

(titlevarchar(50),

Authorvarchar(50),

Subjectvarchar(100),

book\_id number);

book1 books;

book2 books;

BEGIN

--Book1 specification

book1.title :='C Programming';

book1.author :='Nuha Ali ';

book1.subject :='C Programming Tutorial';

book1.book\_id :=6495407;

--Book2 specification

book2.title :='Telecom Billing';

book2.author :='Zara Ali';

book2.subject :='Telecom Billing Tutorial';

book2.book\_id :=6495700;

--Print book 1 record

dbms\_output.put\_line('Book 1 title : '|| book1.title);

dbms\_output.put\_line('Book 1 author : '|| book1.author);

dbms\_output.put\_line('Book 1 subject : '|| book1.subject);

dbms\_output.put\_line('Book 1 book\_id : '|| book1.book\_id);

--Print book 2 record

dbms\_output.put\_line('Book 2 title : '|| book2.title);

dbms\_output.put\_line('Book 2 author : '|| book2.author);

dbms\_output.put\_line('Book 2 subject : '|| book2.subject);

dbms\_output.put\_line('Book 2 book\_id : '|| book2.book\_id);

END;

/

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

Book1title : C Programming

Book1author :NuhaAli

Book1subject : C ProgrammingTutorial

Book1book\_id:6495407

Book2title :TelecomBilling

Book2author :ZaraAli

Book2subject :TelecomBillingTutorial

Book2book\_id:6495700

PL/SQL procedure successfully completed.

# PL/SQL - Collections

A collection is an ordered group of elements having the same data type. Each element is identified by a unique subscript that represents its position in the collection.

PL/SQL provides three collection types:

* Index-by tables or Associative array
* Nested table
* Variable-size array or Varray

Both types of PL/SQL tables, i.e., index-by tables and nested tables have the same structure and their rows are accessed using the subscript notation. However, these two types of tables differ in one aspect; the nested tables can be stored in a database column and the index-by tables cannot.

**Index-By Table**

An **index-by** table (also called an associative array) is a set of **key-value** pairs. Each key is unique and is used to locate the corresponding value. The key can be either an integer or a string.

An index-by table is created using the following syntax. Here, we are creating an index-by table named **table\_name** whose keys will be of *subscript\_type* and associated values will be of *element\_type*

TYPE type\_name IS TABLE OF element\_type [NOT NULL] INDEX BY subscript\_type;

table\_nametype\_name;

**Nested Tables**

A **nested table** is like a one-dimensional array with an arbitrary number of elements. However, a nested table differs from an array in the following aspects:

* An array has a declared number of elements, but a nested table does not. The size of a nested table can increase dynamically.
* An array is always dense, i.e., it always has consecutive subscripts. A nested array is dense initially, but it can become sparse when elements are deleted from it.

A **nested table** is created using the following syntax:

TYPE type\_name IS TABLE OF element\_type [NOT NULL];

table\_nametype\_name;

This declaration is similar to declaration of an **index-by** table, but there is no INDEX BY clause.

A nested table can be stored in a database column and so it could be used for simplifying SQL operations where you join a single-column table with a larger table. An associative array cannot be stored in the database.

**Collection Methods**

PL/SQL provides the built-in collection methods that make collections easier to use. The following table lists the methods and their purpose:

|  |  |
| --- | --- |
| **S.N.** | **Method Name & Purpose** |
| 1 | **EXISTS(n)** Returns TRUE if the nth element in a collection exists; otherwise returns FALSE. |
| 2 | **COUNT** Returns the number of elements that a collection currently contains. |
| 3 | **LIMIT** Checks the Maximum Size of a Collection. |
| 4 | **FIRST** Returns the first (smallest) index numbers in a collection that uses integer subscripts. |
| 5 | **LAST** Returns the last (largest) index numbers in a collection that uses integer subscripts. |
| 6 | **PRIOR(n)** Returns the index number that precedes index n in a collection. |
| 7 | **NEXT(n)** Returns the index number that succeeds index n. |
| 8 | **EXTEND** Appends one null element to a collection. |
| 9 | **EXTEND(n)** Appends n null elements to a collection. |
| 10 | **EXTEND(n,i)** Appends n copies of the ith element to a collection. |
| 11 | **TRIM** Removes one element from the end of a collection. |
| 12 | **TRIM(n)** Removes n elements from the end of a collection. |
| 13 | **DELETE** Removes all elements from a collection, setting COUNT to 0. |
| 14 | **DELETE(n)** Removes the nth element from an associative array with a numeric key or a nested table. If the associative array has a string key, the element corresponding to the key value is deleted. If n is null, DELETE(n) does nothing. |
| 15 | **DELETE(m,n)** Removes all elements in the range m..n from an associative array or nested table. If m is larger than n or if m or n is null, DELETE(m,n) does nothing. |

**Collection Exceptions**

The following table provides the collection exceptions and when they are raised:

|  |  |
| --- | --- |
| **Collection Exception** | **Raised in Situations** |
| COLLECTION\_IS\_NULL | You try to operate on an atomically null collection. |
| NO\_DATA\_FOUND | A subscript designates an element that was deleted, or a nonexistent element of an associative array. |
| SUBSCRIPT\_BEYOND\_COUNT | A subscript exceeds the number of elements in a collection. |
| SUBSCRIPT\_OUTSIDE\_LIMIT | A subscript is outside the allowed range. |
| VALUE\_ERROR | A subscript is null or not convertible to the key type. This exception might occur if the key is defined as a PLS\_INTEGER range, and the subscript is outside this range. |

# PL/SQL - Exceptions

An error condition during a program execution is called an exception in PL/SQL. PL/SQL supports programmers to catch such conditions using **EXCEPTION**block in the program and an appropriate action is taken against the error condition. There are two types of exceptions:

* System-defined exceptions
* User-defined exceptions

**Syntax for Exception Handling**

The General Syntax for exception handling is as follows. Here you can list down as many as exceptions you want to handle. The default exception will be handled using *WHEN others THEN*:

DECLARE

<declarations section>

BEGIN

<executable command(s)>

EXCEPTION

<exception handling goes here >

WHEN exception1 THEN

exception1-handling-statements

WHEN exception2 THEN

exception2-handling-statements

WHEN exception3 THEN

exception3-handling-statements

........

WHEN others THEN

exception3-handling-statements

END;

Example

Let us write some simple code to illustrate the concept. We will be using the CUSTOMERS table we had created and used in the previous chapters:

DECLARE

c\_idcustomers.id%type := 8;

c\_namecustomers.name%type;

c\_addrcustomers.address%type;

BEGIN

SELECT name, address INTO c\_name, c\_addr

FROM customers

WHERE id = c\_id;

DBMS\_OUTPUT.PUT\_LINE ('Name: '|| c\_name);

DBMS\_OUTPUT.PUT\_LINE ('Address: ' || c\_addr);

EXCEPTION

WHEN no\_data\_found THEN

dbms\_output.put\_line('No such customer!');

WHEN others THEN

dbms\_output.put\_line('Error!');

END;

/

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

No such customer!

PL/SQL procedure successfully completed.

The above program displays the name and address of a customer whose ID is given. Since there is no customer with ID value 8 in our database, the program raises the run-time exception **NO\_DATA\_FOUND**, which is captured in**EXCEPTION** block.

**Raising Exceptions**

Exceptions are raised by the database server automatically whenever there is any internal database error, but exceptions can be raised explicitly by the programmer by using the command **RAISE**. Following is the simple syntax of raising an exception:

DECLARE

exception\_name EXCEPTION;

BEGIN

IF condition THEN

RAISE exception\_name;

END IF;

EXCEPTION

WHEN exception\_name THEN

statement;

END;

You can use above syntax in raising Oracle standard exception or any user-defined exception. Next section will give you an example on raising user-defined exception, similar way you can raise Oracle standard exceptions as well.

**User-defined Exceptions**

PL/SQL allows you to define your own exceptions according to the need of your program. A user-defined exception must be declared and then raised explicitly, using either a RAISE statement or the procedure DBMS\_STANDARD.RAISE\_APPLICATION\_ERROR.

The syntax for declaring an exception is:

DECLARE

my-exception EXCEPTION;

Example:

The following example illustrates the concept. This program asks for a customer ID, when the user enters an invalid ID, the exception invalid\_id is raised.

DECLARE

c\_idcustomers.id%type :=&c\_id;

c\_namecustomers.name%type;

c\_addrcustomers.address%type;

-- user defined exception

ex\_invalid\_id EXCEPTION;

BEGIN

IF c\_id<= 0 THEN

RAISE ex\_invalid\_id;

ELSE

SELECT name, address INTO c\_name, c\_addr

FROM customers

WHERE id = c\_id;

DBMS\_OUTPUT.PUT\_LINE ('Name: '|| c\_name);

DBMS\_OUTPUT.PUT\_LINE ('Address: ' || c\_addr);

END IF;

EXCEPTION

WHEN ex\_invalid\_id THEN

dbms\_output.put\_line('ID must be greater than zero!');

WHEN no\_data\_found THEN

dbms\_output.put\_line('No such customer!');

WHEN others THEN

dbms\_output.put\_line('Error!');

END;

/

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

Enter value for cc\_id: -6 (let's enter a value -6)

old 2: c\_idcustomers.id%type := &cc\_id;

new 2: c\_idcustomers.id%type := -6;

ID must be greater than zero!

PL/SQL procedure successfully completed.

**Pre-defined Exceptions**

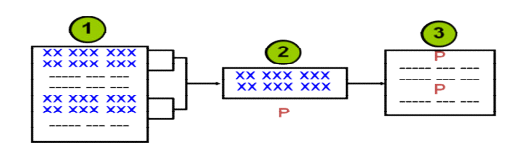
PL/SQL provides many pre-defined exceptions, which are executed when any database rule is violated by a program. For example, the predefined exception NO\_DATA\_FOUND is raised when a SELECT INTO statement returns no rows. The following table lists few of the important pre-defined exceptions:

|  |  |  |  |
| --- | --- | --- | --- |
| **Exception** | **Oracle Error** | **SQLCODE** | **Description** |
| ACCESS\_INTO\_NULL | 06530 | -6530 | It is raised when a null object is automatically assigned a value. |
| CASE\_NOT\_FOUND | 06592 | -6592 | It is raised when none of the choices in the WHEN clauses of a CASE statement is selected, and there is no ELSE clause. |
| COLLECTION\_IS\_NULL | 06531 | -6531 | It is raised when a program attempts to apply collection methods other than EXISTS to an uninitialized nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray. |
| DUP\_VAL\_ON\_INDEX | 00001 | -1 | It is raised when duplicate values are attempted to be stored in a column with unique index. |
| INVALID\_CURSOR | 01001 | -1001 | It is raised when attempts are made to make a cursor operation that is not allowed, such as closing an unopened cursor. |
| INVALID\_NUMBER | 01722 | -1722 | It is raised when the conversion of a character string into a number fails because the string does not represent a valid number. |
| LOGIN\_DENIED | 01017 | -1017 | It is raised when s program attempts to log on to the database with an invalid username or password. |
| NO\_DATA\_FOUND | 01403 | +100 | It is raised when a SELECT INTO statement returns no rows. |
| NOT\_LOGGED\_ON | 01012 | -1012 | It is raised when a database call is issued without being connected to the database. |
| PROGRAM\_ERROR | 06501 | -6501 | It is raised when PL/SQL has an internal problem. |
| ROWTYPE\_MISMATCH | 06504 | -6504 | It is raised when a cursor fetches value in a variable having incompatible data type. |
| SELF\_IS\_NULL | 30625 | -30625 | It is raised when a member method is invoked, but the instance of the object type was not initialized. |
| STORAGE\_ERROR | 06500 | -6500 | It is raised when PL/SQL ran out of memory or memory was corrupted. |
| TOO\_MANY\_ROWS | 01422 | -1422 | It is raised when s SELECT INTO statement returns more than one row. |
| VALUE\_ERROR | 06502 | -6502 | It is raised when an arithmetic, conversion, truncation, or size-constraint error occurs. |
| ZERO\_DIVIDE | 01476 | 1476 | It is raised when an attempt is made to divide a number by zero. |

**Unit V**

# PL/SQL – Procedures

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

A schema level subprogram is a **standalone subprogram**. It is created with the CREATE PROCEDURE or 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**: 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** and we will cover **PL/SQL function** in next chapter.

## Parts of a PL/SQL Subprogram

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

|  |  |
| --- | --- |
| **S.N.** | **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>

ENDprocedure\_name;

Where,

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

HelloWorld

PL/SQL procedure successfully completed.

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

BEGIN

greetings;

END;

/

The above call would display:

HelloWorld

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;

So you can drop *greetings* procedure by using the following statement:

DROP PROCEDURE greetings;

## Parameter Modes in PL/SQL Subprograms

|  |  |
| --- | --- |
| **S.N.** | **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**. |
| 2 | **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 its 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 procedure takes two numbers using IN mode and returns their minimum using 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 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 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 SQL prompt, it produces the following result:

Square of (23):529

PL/SQL procedure successfully completed.

## Methods for Passing Parameters

Actual parameters could 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 ( => ). So the procedure call would look like:

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

### MIXED NOTATION

In mixed notation, you can mix both notations in procedure call; however, the positional notation should precede the named notation.

The following call is legal:

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

But this is not legal:

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

# PL/SQL - Functions

A PL/SQL function is same as a procedure except that it returns a value. Therefore, all the discussions of the previous chapter are true for functions 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 modifying an existing function.
* The optional parameter list contains name, mode and types of the parameters. IN represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure.
* The function must contain a **return** statement.
* *RETURN* clause specifies that data type you are going to return from the function.
* *function-body* contains the executable part.
* The AS keyword is used instead of the IS keyword for creating a standalone function.

## Example:

The following example illustrates creating and calling 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 [PL/SQL Variables](http://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 above code is executed using 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, program control is transferred to the called function.

A called function performs defined task and when its return statement is executed or when it last end statement is reached, it returns program control back to the main program.

To call a function you simply need to pass the required parameters along with function name and if function returns a value then you can store 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 SQL prompt, it produces the following result:

Totalno.ofCustomers:6

PL/SQL procedure successfully completed.

## Example:

The following is one more example which 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 SQL prompt, it produces the following result:

Maximum of (23,45):45

PL/SQL procedure successfully completed.

## PL/SQL Recursive Functions

We have seen that a program or subprogram may call another subprogram. When a subprogram calls itself, it is referred to as a recursive call and the process is known as recursion.

To illustrate the concept, let us calculate the factorial of a number. Factorial of a number n is defined as:

n!= n\*(n-1)!

= n\*(n-1)\*(n-2)!

...

= n\*(n-1)\*(n-2)\*(n-3)...1

The following program calculates the factorial of a given number by calling itself recursively:

DECLARE

num number;

factorial number;

FUNCTION fact(x number)

RETURN number

IS

f number;

BEGIN

IF x=0 THEN

f:=1;

ELSE

f:= x \* fact(x-1);

END IF;

RETURN f;

END;

BEGIN

num:=6;

factorial:= fact(num);

dbms\_output.put\_line(' Factorial '||num||' is '|| factorial);

END;

/

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

Factorial6is720

PL/SQL procedure successfully completed.

# PL/SQL - Packages

PL/SQL packages are schema objects that groups logically related PL/SQL types, variables and subprograms.

A package will have two mandatory parts:

* Package specification
* Package body or definition

## Package Specification

The specification is the interface to the package. It just DECLARES the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package. In other words, it contains all information about the content of the package, but excludes the code for the subprograms.

All objects placed in the specification are called **public** objects. Any subprogram not in the package specification but coded in the package body is called a**private** object.

The following code snippet shows a package specification having a single procedure. You can have many global variables defined and multiple procedures or functions inside a package.

CREATE PACKAGE cust\_sal AS

PROCEDURE find\_sal(c\_idcustomers.id%type);

ENDcust\_sal;

/

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

Package created.

## Package Body

The package body has the codes for various methods declared in the package specification and other private declarations, which are hidden from code outside the package.

The CREATE PACKAGE BODY Statement is used for creating the package body. The following code snippet shows the package body declaration for the *cust\_sal*package created above. I assumed that we already have CUSTOMERS table created in our database as mentioned in [PL/SQL - Variables](http://www.tutorialspoint.com/plsql/plsql_variable_types.htm) chapter.

CREATE OR REPLACE PACKAGE BODY cust\_sal AS

PROCEDURE find\_sal(c\_idcustomers.id%TYPE) IS

c\_salcustomers.salary%TYPE;

BEGIN

SELECT salary INTO c\_sal

FROM customers

WHERE id =c\_id;

dbms\_output.put\_line('Salary: '||c\_sal);

ENDfind\_sal;

ENDcust\_sal;

/

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

Package body created.

## Using the Package Elements

The package elements (variables, procedures or functions) are accessed with the following syntax:

package\_name.element\_name;

Consider, we already have created above package in our database schema, the following program uses the *find\_sal* method of the *cust\_sal* package:

DECLARE

codecustomers.id%type:=&cc\_id;

BEGIN

cust\_sal.find\_sal(code);

END;

/

When the above code is executed at SQL prompt, it prompt to enter customer ID and when you enter an ID, it displays corresponding salary as follows:

Enter value forcc\_id:1

Salary:3000

PL/SQL procedure successfully completed.

## Example:

The following program provides a more complete package. We will use the CUSTOMERS table stored in our database with the following records:

Select\*from customers;

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

|1|Ramesh|32|Ahmedabad|3000.00|

|2|Khilan|25|Delhi|3000.00|

|3|kaushik|23|Kota|3000.00|

|4|Chaitali|25|Mumbai|7500.00|

|5|Hardik|27|Bhopal|9500.00|

|6|Komal|22| MP |5500.00|

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

### THE PACKAGE SPECIFICATION:

CREATE OR REPLACE PACKAGE c\_package AS

--Adds a customer

PROCEDURE addCustomer(c\_idcustomers.id%type,

c\_namecustomers.name%type,

c\_agecustomers.age%type,

c\_addrcustomers.address%type,

c\_salcustomers.salary%type);

--Removes a customer

PROCEDURE delCustomer(c\_idcustomers.id%TYPE);

--Lists all customers

PROCEDURE listCustomer;

ENDc\_package;

/

When the above code is executed at SQL prompt, it creates the above package and displays the following result:

Package created.

### CREATING THE PACKAGE BODY:

CREATE OR REPLACE PACKAGE BODY c\_package AS

PROCEDURE addCustomer(c\_idcustomers.id%type,

c\_namecustomers.name%type,

c\_agecustomers.age%type,

c\_addrcustomers.address%type,

c\_salcustomers.salary%type)

IS

BEGIN

INSERT INTO customers (id,name,age,address,salary)

VALUES(c\_id,c\_name,c\_age,c\_addr,c\_sal);

ENDaddCustomer;

PROCEDURE delCustomer(c\_idcustomers.id%type) IS

BEGIN

DELETE FROM customers

WHERE id =c\_id;

ENDdelCustomer;

PROCEDURE listCustomer IS

CURSOR c\_customers is

SELECT name FROM customers;

TYPE c\_listis TABLE OF customers.name%type;

name\_listc\_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;

ENDlistCustomer;

ENDc\_package;

/

Above example makes use of **nested table** which we will discuss in the next chapter. When the above code is executed at SQL prompt, it produces the following result:

Package body created.

### USING THE PACKAGE:

The following program uses the methods declared and defined in the package*c\_package*.

DECLARE

codecustomers.id%type:=8;

BEGIN

c\_package.addcustomer(7,'Rajnish',25,'Chennai',3500);

c\_package.addcustomer(8,'Subham',32,'Delhi',7500);

c\_package.listcustomer;

c\_package.delcustomer(code);

c\_package.listcustomer;

END;

/

When the above code is executed at 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

Customer(7):Rajnish

Customer(8):Subham

Customer(1):Ramesh

Customer(2):Khilan

Customer(3):kaushik

Customer(4):Chaitali

Customer(5):Hardik

Customer(6):Komal

Customer(7):Rajnish

PL/SQL procedure successfully completed

# PL/SQL Dynamic SQL

**Dynamic SQL** is a programming methodology for generating and running SQL statements at run time. It is useful when writing general-purpose and flexible programs like ad hoc query systems, when writing programs that must run database definition language (DDL) statements, or when you do not know at compilation time the full text of a SQL statement or the number or data types of its input and output variables.

PL/SQL provides two ways to write dynamic SQL:

* Native dynamic SQL, a PL/SQL language (that is, native) feature for building and running dynamic SQL statements
* DBMS\_SQL package, an API for building, running, and describing dynamic SQL statements

Native dynamic SQL code is easier to read and write than equivalent code that uses the DBMS\_SQL package, and runs noticeably faster (especially when it can be optimized by the compiler). However, to write native dynamic SQL code, you must know at compile time the number and data types of the input and output variables of the dynamic SQL statement. If you do not know this information at compile time, you must use the DBMS\_SQL package.

## When You Need Dynamic SQL

In PL/SQL, you need dynamic SQL to run:

* SQL whose text is unknown at compile time

For example, a SELECT statement that includes an identifier that is unknown at compile time (such as a table name) or a WHERE clause in which the number of subclauses is unknown at compile time.

* SQL that is not supported as static SQL

That is, any SQL construct not included in ["Description of Static SQL"](https://docs.oracle.com/cloud/latest/db112/LNPLS/static.htm#i7112).

If you do not need dynamic SQL, use static SQL, which has these advantages:

* Successful compilation verifies that static SQL statements reference valid database objects and that the necessary privileges are in place to access those objects.
* Successful compilation creates schema object dependencies.

## Native Dynamic SQL

Native dynamic SQL processes most dynamic SQL statements with the EXECUTE IMMEDIATE statement.

If the dynamic SQL statement is a SELECT statement that returns multiple rows, native dynamic SQL gives you these choices:

* Use the EXECUTE IMMEDIATE statement with the BULK COLLECT INTO clause.
* Use the OPEN FOR, FETCH, and CLOSE statements.

The SQL cursor attributes work the same way after native dynamic SQL INSERT, UPDATE, DELETE, MERGE, and single-row SELECT statements as they do for their static SQL counterparts

### EXECUTE IMMEDIATE Statement

The EXECUTE IMMEDIATE statement is the means by which native dynamic SQL processes most dynamic SQL statements.

If the dynamic SQL statement is **self-contained** (that is, if it has no placeholders for bind variables and the only result that it can possibly return is an error), then the EXECUTE IMMEDIATE statement needs no clauses.

If the dynamic SQL statement includes placeholders for bind variables, each placeholder must have a corresponding bind variable in the appropriate clause of the EXECUTE IMMEDIATE statement, as follows:

* If the dynamic SQL statement is a SELECT statement that can return at most one row, put out-bind variables (defines) in the INTO clause and in-bind variables in the USING clause.
* If the dynamic SQL statement is a SELECT statement that can return multiple rows, put out-bind variables (defines) in the BULK COLLECT INTOclause and in-bind variables in the USING clause.
* If the dynamic SQL statement is a DML statement without a RETURNING INTO clause, other than SELECT, put all bind variables in the USINGclause.
* If the dynamic SQL statement is a DML statement with a RETURNING INTO clause, put in-bind variables in the USING clause and out-bind variables in the RETURNING INTO clause.
* If the dynamic SQL statement is an anonymous PL/SQL block or a CALL statement, put all bind variables in the USING clause.

### OPEN FOR, FETCH, and CLOSE Statements

If the dynamic SQL statement represents a SELECT statement that returns multiple rows, you can process it with native dynamic SQL as follows:

1. Use an OPEN FOR statement to associate a cursor variable with the dynamic SQL statement. In the USING clause of the OPEN FOR statement, specify a bind variable for each placeholder in the dynamic SQL statement.

The USING clause cannot contain the literal NULL. To work around this restriction, use an uninitialized variable where you want to use NULL, as in [Example 7-3](https://docs.oracle.com/cloud/latest/db112/LNPLS/dynamic.htm#BHCBIICB).

For syntax details, see ["OPEN FOR Statement"](https://docs.oracle.com/cloud/latest/db112/LNPLS/openfor_statement.htm#i35231).

1. Use the FETCH statement to retrieve result set rows one at a time, several at a time, or all at once.

For syntax details, see ["FETCH Statement"](https://docs.oracle.com/cloud/latest/db112/LNPLS/fetch_statement.htm#i34221).

1. Use the CLOSE statement to close the cursor variable.

For syntax details, see ["CLOSE Statement"](https://docs.oracle.com/cloud/latest/db112/LNPLS/close_statement.htm#i32987).

# PL/SQL - Triggers

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 could 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 would 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 would 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, like INSERT, UPDATE, and DELETE.
* [FOR EACH ROW]: This specifies a row level trigger, i.e., the trigger would 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 SQL prompt, it produces the following result:

Trigger created.

Here following two points are important and should be noted carefully:

* 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.
* 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 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 CUSTOMERS table, 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 so old salary is not available and above result is coming as null. Now, let us perform one more DML operation on the CUSTOMERS table. Here is one UPDATE statement, which will update an existing record in the table:

UPDATE customers

SET salary = salary +500

WHERE id =2;

When a record is updated in CUSTOMERS table, 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

## Types of PL/SQL Triggers

There are two types of triggers based on the which level it is triggered.  
**1) Row level trigger**- An event is triggered for each row upated, inserted or deleted.   
**2) Statement level trigger**- An event is triggered for each sql statement executed.

## PL/SQL Trigger Execution Hierarchy

The following hierarchy is followed when a trigger is fired.  
**1)**BEFORE statement trigger fires first.  
**2)** Next BEFORE row level trigger fires, once for each row affected.   
**3)** Then AFTER row level trigger fires once for each affected row. This eventswill alternates between BEFORE and AFTER row level triggers.  
**4)** Finally the AFTER statement level trigger fires.

**For Example:** Let's create a table 'product\_check' which we can use to store messages when triggers are fired.

*CREATE TABLE product*

*(Message varchar2(50),*

*Current\_Datenumber(32)*

*);*

Let's create a BEFORE and AFTER statement and row level triggers for the product table.

**1) BEFORE UPDATE, Statement Level:** This trigger will insert a record into the table 'product\_check' before a sql update statement is executed, at the statement level.

*CREATE or REPLACE TRIGGER Before\_Update\_Stat\_product*

*BEFORE*

*UPDATE ON product*

*Begin*

*INSERT INTO product\_check*

*Values('Before update, statement level',sysdate);*

*END;*

*/*

**2) BEFORE UPDATE, Row Level:**This trigger will insert a record into the table 'product\_check' before each row is updated.

*CREATE or REPLACE TRIGGER Before\_Upddate\_Row\_product*

*BEFORE*

*UPDATE ON product*

*FOR EACH ROW*

*BEGIN*

*INSERT INTO product\_check*

*Values('Before update row level',sysdate);*

*END;*

*/*

**3) AFTER UPDATE, Statement Level:** This trigger will insert a record into the table 'product\_check' after a sql update statement is executed, at the statement level.

*CREATE or REPLACE TRIGGER After\_Update\_Stat\_product*

*AFTER*

*UPDATE ON product*

*BEGIN*

*INSERT INTO product\_check*

*Values('After update, statement level', sysdate);*

*End;*

*/*

**4) AFTER UPDATE, Row Level:**This trigger will insert a record into the table 'product\_check' after each row is updated.

*CREATE or REPLACE TRIGGER After\_Update\_Row\_product*

*AFTER*

*insert On product*

*FOR EACH ROW*

*BEGIN*

*INSERT INTO product\_check*

*Values('After update, Row level',sysdate);*

*END;*

*/*

Now lets execute a update statement on table product.

*UPDATE PRODUCT SET unit\_price = 800*

*WHERE product\_id in (100,101);*

Lets check the data in 'product\_check' table to see the order in which the trigger is fired.

*SELECT \* FROM product\_check;*

**Output:**

Mesage                                             Current\_Date

------------------------------------------------------------

Before update, statement level          26-Nov-2008  
Before update, row level                    26-Nov-2008  
After update, Row level                     26-Nov-2008  
Before update, row level                    26-Nov-2008  
After update, Row level                     26-Nov-2008  
After update, statement level            26-Nov-2008

The above result shows 'before update' and 'after update' row level events have occured twice, since two records were updated. But 'before update' and 'after update' statement level events are fired only once per sql statement.

The above rules apply similarly for INSERT and DELETE statements.

## How To know Information about Triggers.

We can use the data dictionary view 'USER\_TRIGGERS' to obtain information about any trigger.

The below statement shows the structure of the view 'USER\_TRIGGERS'

*DESC USER\_TRIGGERS;*

NAME                              Type

--------------------------------------------------------

TRIGGER\_NAME                 VARCHAR2(30)  
TRIGGER\_TYPE                  VARCHAR2(16)  
TRIGGER\_EVENT                VARCHAR2(75)   
TABLE\_OWNER                  VARCHAR2(30)  
BASE\_OBJECT\_TYPE           VARCHAR2(16)  
TABLE\_NAME                     VARCHAR2(30)  
COLUMN\_NAME                  VARCHAR2(4000)  
REFERENCING\_NAMES        VARCHAR2(128)  
WHEN\_CLAUSE                  VARCHAR2(4000)  
STATUS                            VARCHAR2(8)  
DESCRIPTION                    VARCHAR2(4000)  
ACTION\_TYPE                   VARCHAR2(11)  
TRIGGER\_BODY                 LONG

This view stores information about header and body of the trigger.

*SELECT \* FROM user\_triggers WHERE trigger\_name = 'Before\_Update\_Stat\_product';*

The above sql query provides the header and body of the trigger 'Before\_Update\_Stat\_product'.

You can drop a trigger using the following command.

*DROP TRIGGER trigger\_name;*