# Practical 5 : Triggers

## Learning objectives:

1. Overview of triggers
2. Types of triggers
3. Creating and modifying triggers
4. Drop, enable and disable triggers

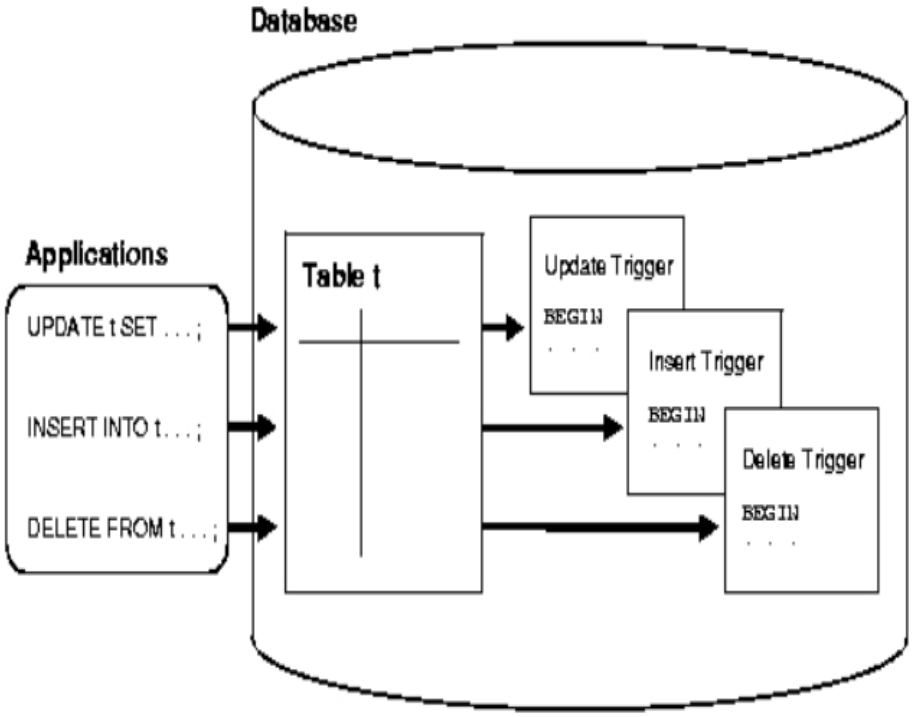
**References**

PL/SQL Language Reference https://docs.oracle.com/en/database/oracle/oracle-database/18/lnpls/database-pl-sql-language-reference.pdf

https://docs.oracle.com/en/database/oracle/oracle-database/19/tdddg/using-triggers.html

## Syntax of trigger

```
CREATE OR REPLACE TRIGGER trigger_name
  [BEFORE / AFTER]
  [DELETE / INSERT / UPDATE OF column_name]
  ON table_name
  [FOR EACH ROW]
  [WHEN (condition)]
  [DECLARE]
  [variable_name   data type[:=initial_value]]
  BEGIN
    PL/SQL instructions;
  END;
/
```




**Consider this new scenario:**

- We now need to store four more details in the **ORDERS** table, whenever an order is made.
  ***orderAmount, discount, totalDiscount and finalTotal***

- We also need to accumulate **the monthly spending** by each customers.
  (discount is defined as a percent, i.e. 5% to a maximum of 50%).

To capture the ***orderAmount, totalDiscount and finalTotal*** we need to modify the existing **ORDERS** table:

```
alter table orders
add (orderAmount   Number(7,2) Default 0.00,
     discount      Number(2,2) Default 0.05,
     totalDiscount Number(7,2) Default 0.00,
     finalTotal    Number(7,2) Default 0.00);
```

Check the new structure and data for correctness:

```
SQL>desc Orders

SQL>Select orderNumber, orderAmount, discount, totalDiscount,
           finalTotal from Orders;
```

***orderAmount, totalDiscount and finalTotal*** can only be calculated after we insert the necessary data into the **ORDERDETAILS**.
For each order detail, we need to calculate:

```
orderAmount= orderDetails.priceEach * orderDetails.quantityOrdered
```

```
totalDiscount = orderAmount * orders.Discount
finalTotal    = orderAmount - totalDiscount
```

Every time a new record is inserted into OrderDetails table, the above calculations should be accumulated automatically to the same OrderNumber record in the ORDERS table.

As we accumulate the amounts of the ORDERS table, we should also accumulate the amount spent by each customer in a month automatically

Run the **`cusMonth.sql`** to create the CUSTOMERS_MONTHLY table.

```
create table CUSTOMERS_MONTHLY
(    customerNumber number(11) NOT NULL,
     yearNo         number(4) NOT NULL,
     monthNo        number(2) NOT NULL,
     monthly_Amount number(9,2),
     primary key (customerNumber, yearNo,monthNo)
);
```

To demonstrate the workings of the TRIGGER, we need to remove the **ORDERDETAILS** records and re-insert them and as each record is inserted, the DBMS will automatically do the necessary calculations.

We will need to create two triggers that will do the following:

a. Monitor any insert into the **ORDERDETAILS** table. After an insert, automatically calculate and update the **ORDERS** table on the
   **`OrderAmount, TotalDiscount and FinalTotal`** columns

b. Monitor any updates of the column **`FinalTotal`** on the **ORDERS** table. When this column is updated, the customer's monthly spending should also be updated.

**General use of TRIGGERS**

- Triggers are to "monitor" changes to the database
- Take the appropriate "action" so that important data (and information) are not lost due to update and delete.
- Most of the time, triggers will have instructions to store important data/information in a log or audit file.
- Triggers can also stop a DML statement such as **INSERT, UPDATE, DELETE** from executing by invoking a system error (use a ***BEFORE*** trigger)
  - i.e. use ***RAISE_APPLICATION_ERROR***
  - If ***RAISE_APPLICATION_ERROR*** is used on ***AFTER*** trigger, the effect of the DML statement has taken place and may need to **ROLLBACK**

# Task

a. Remove the ORDERDETAILS records

```
Delete ORDERDETAILS;
```

b. Create the triggers.

```
Start c:\trig1
Start c:\trig2
```

**Note:**

**trig1.sql** will automatically update ***orderAmount, totalDiscount and finalTotal*** on the **ORDERS** table every time a record is successfully inserted into the **ORDERDETAILS** table.

```
CREATE OR REPLACE TRIGGER TRG_UPT_ORDERS_AMT
AFTER INSERT ON orderDetails
FOR EACH ROW
DECLARE
  v_OrderAmount   NUMBER := 0.00;
  v_TotalDiscount NUMBER := 0.00;
  v_FinalTotal    NUMBER := 0.00;
  v_discount      number(2,2);

BEGIN
  select discount into v_discount
    from Orders
    where orderNumber = :new.orderNumber;

  v_OrderAmount   := :new.priceEach * :new.quantityOrdered;
  v_TotalDiscount := ROUND( v_OrderAmount * v_discount,2);
  v_FinalTotal    := v_OrderAmount - v_TotalDiscount;

  UPDATE ORDERS
  SET OrderAmount   = OrderAmount + v_OrderAmount,
     TotalDiscount = TotalDiscount + v_TotalDiscount,
     FinalTotal    = FinalTotal + v_FinalTotal
  WHERE orderNumber = :new.orderNumber;
END;
/
```

**Note:**

| Operation | OLD Value | NEW Value |
|---|---|---|
| INSERT | NULL | the new inserted value |
| UPDATE | the column value before the update | the column value after the update |
| DELETE | the column value before deletion | NULL |

**trig2.sql** will automatically update the customer's monthly spending every time there is an update of `finalTotal` on the ORDERS table.

```
CREATE OR REPLACE TRIGGER TRG_UPT_CUST_MONTHLY
AFTER UPDATE OF FinalTotal ON Orders
FOR EACH ROW
DECLARE
  v_custNum CUSTOMERS.CUSTOMERNUMBER%TYPE;

BEGIN
  Select customerNumber into v_custNum
  From CUSTOMERS_MONTHLY
  Where customerNumber = :new.customerNumber AND
       YearNo  = EXTRACT(Year From(:new.OrderDate)) AND
       MonthNo = EXTRACT(Month From(:new.OrderDate));

  IF SQL%FOUND THEN
     UPDATE CUSTOMERS_MONTHLY
       SET Monthly_Amount = Monthly_Amount + :new.FinalTotal - :old.FinalTotal
       WHERE customerNumber = :new.customerNumber       AND
       YearNo  = EXTRACT(Year From(:new.OrderDate)) AND
       MonthNo = EXTRACT(Month From(:new.OrderDate));
  END IF;

EXCEPTION
  WHEN NO_DATA_FOUND THEN
     INSERT INTO CUSTOMERS_MONTHLY VALUES
       (:new.customerNumber,EXTRACT(Year From(:new.OrderDate)),
            EXTRACT(Month From(:new.OrderDate)),:new.FinalTotal);
END;
/
```

c. To fire the trigger, insert one orderDetails record.

```
insert into orderdetails values(10100,'S18_1749',30,136,3);
```

d. Check the updates.

```
Select orderNumber, OrderAmount, TotalDiscount, FinalTotal
   from Orders where orderNumber = 10100;

Select * from orderDetails;

Select * from customers_monthly;
```

e. Insert two more test data.

```
insert into orderdetails values(10100,'S18_2248',50,55.09,2);
insert into orderdetails values(10100,'S18_4409',22,75.46,4);
```

f. Check the updates.

g. Remove all the changes done to the database.

```
ROLLBACK;

Select orderNumber, OrderAmount, TotalDiscount, FinalTotal
  from Orders where orderNumber = 10100;

Select * from orderDetails;

Select * from customers_monthly;
```

h. Execute the script **`Insert_Orderdetails.txt`** to re-insert data into the **orderDetails** table. As each record is inserted, the correct amount will be accumulated to the respective **ORDERS** and **CUSTOMERS_MONTHLY** records.

i. Check the updates.

In **Practical 3**, there is a procedure **`PRC_PRICE_INCR(xxx)`** that accepts an input **ProductCode** and updates the price of that product. The procedure will also write a record to the **Price_Audit** table, used to keep track of the changes to the product.

However, a record will only be written to the **Price_Audit** table ***if the user executes*** the **`PRC_PRICE_INCR(xxx)`** procedure. If an update to the product price was done by other means (e.g. user type the update DML directly or run another procedure to update), then the changes will not be recorded for future reference.

# Task

a. Create a trigger to automatically record the price changes of any product:

```
CREATE OR REPLACE TRIGGER TRG_INSERT_Price_Audit
AFTER UPDATE OF MSRP ON Products
FOR EACH ROW
BEGIN
   INSERT INTO Price_Audit VALUES
   (:new.productCode, :new.productName,
    :old.MSRP, :new.MSRP, USER, SYSDATE);
END;
/
```

Once this trigger is created (and enabled), all changes to ***MSRP*** in **PRODUCTS** table will be automatically recorded into the **Price_Audit** table without any further intervention from the user.

b. Modify the **`PRC_PRICE_INCR(xxx)`** procedure by **removing** the code that does the ***INSERT*** to the **Price_Audit** table.

c. Execute **`PRC_PRICE_INCR(xxx)`** with sample test data and check the **Price_Audit** table for updated rows.

## Fire Once Trigger/ Statement-level Trigger

The trigger examples given above will activate for every row of record being manipulated (such triggers are called as row-level triggers). We may also create a trigger that only fire once even though the ***INSERT, UPDATE, DELETE*** events may have manipulated more than one row of records.

Consider the following:

We now want to perform a simple tracking of the **EMPLOYEES** table. We would like to record the number of ***INSERT,UPDATE,DELETE*** activities that is performed on the **EMPLOYEES** table. We do not need to know the exact number of records manipulated, we just want to know if some DML activities are being performed.

Firstly, create the **Employees_audit** table:

```
CREATE TABLE Employees_Audit
(  TransDate        Date,
   No_of_Insert     Number(3),
   No_of_Update     Number(3),
   No_of_Delete     Number(3)
);
```

Next, examine and create the **trig3.sql** code. This is an example of a fire once trigger.

```
CREATE OR REPLACE TRIGGER Track_Employees
BEFORE INSERT OR
      UPDATE OR
      DELETE
      ON employees
DECLARE
  v_date date;

BEGIN
  select TransDate into v_date
  from Employees_Audit
  where to_char(TransDate,'DD-MON-YY') = to_char(SYSDATE,'DD-MON-YY');
-- check if there is already a record for today ---

  if SQL%FOUND THEN
    CASE
      WHEN INSERTING THEN
        UPDATE Employees_AUDIT
          SET No_Of_Insert = No_Of_Insert + 1
        where to_char(TransDate,'DD-MON-YY') = to_char(SYSDATE,'DD-MON-YY');
      WHEN UPDATING THEN
        UPDATE Employees_AUDIT
          SET No_Of_Update = No_Of_Update + 1
        where to_char(TransDate,'DD-MON-YY') = to_char(SYSDATE,'DD-MON-YY');
      WHEN DELETING THEN
        UPDATE Employees_AUDIT
          SET No_Of_Delete = No_Of_Delete + 1
```

```
        where to_char(TransDate,'DD-MON-YY') = to_char(SYSDATE,'DD-MON-YY');
    END CASE;
  end if;

  exception
    when no_data_found then
    CASE
      WHEN INSERTING THEN
        INSERT INTO Employees_AUDIT VALUES(sysdate,1,0,0);
      WHEN UPDATING THEN
        INSERT INTO Employees_AUDIT VALUES(sysdate,0,1,0);
      WHEN DELETING THEN
        INSERT INTO Employees_AUDIT VALUES(sysdate,0,0,1);
    END CASE;
END;
/
```

What is the difference between a ***row-level trigger*** and a ***statement-level trigger***?

- If you perform an update that will affect 10 rows, a statement level trigger will fire once and a row level trigger will fire 10 times.
- A statement level trigger does not have access to individual column values. A row level trigger does have access to column values.

## INSTEAD OF Trigger for Updating a View

Recall that a complex view cannot be updated. For example, consider the following requirement to show each employee's reporting line:

```
Select employeeNumber, lastName, firstName, jobTitle, ReportsTo
From employees
Order by ReportsTO;
```

What if we also want the names of the manager to be displayed?

The following view will list employees that manages other employees.

```
CREATE OR REPLACE VIEW MANAGER_VIEW AS
  select distinct(Wkr.reportsTo) AS ManagerID, Mgr.lastname,
       Mgr.firstname, Mgr.jobTitle
  from employees Mgr, employees Wkr
  where (Wkr.reportsTo = Mgr.employeeNumber);
```

If you need to change the title of empid (5) to 'Senior Sales Manager', the following is the DML

```
update manager_view
set jobTitle = 'Senior Sales Rep'
where ManagerID = 1621;
```

What is the error message?

We will use a INSTEAD OF trigger to update a complex view:

```
CREATE OR REPLACE TRIGGER update_mgr_view
INSTEAD OF UPDATE ON manager_view
FOR EACH ROW
BEGIN
-- allow the following updates to the underlying employees table
   UPDATE employees
      SET JobTitle = :NEW.jobTitle
   WHERE employeeNumber = :OLD.ManagerId;
END;
/
```

**Disabling/enabling a Specific Trigger**

```
ALTER TRIGGER Track_Employees DISABLE/ENABLE;
```

**Disabling All Triggers on a Table**

```
ALTER TABLE products DISABLE ALL TRIGGERS;
```

**Enabling All Triggers for a Table**

```
ALTER TABLE products ENABLE ALL TRIGGERS;
```

**When a table is removed using the DROP TABLE, all triggers associated with the table will also be dropped.**

# Exercises

1. Add a new column to the CUSTOMERS table to store each customer's cumulative purchase to date. Create the appropriate trigger to accumulate this amount automatically.

2. Add a total_units_sold column to the PRODUCTS table. Create a trigger to accumulate the total units sold for each product.

3. Create a trigger name trg_upd_prodqty that will automatically update the product quantityInStock each time an order transaction is processed.

4. Create an audit table to keep track of the relevant product information and audit information for DML events performed on the PRODUCTS records. Then, create a trigger that will insert into the audit table for the monitored DML events [You decide what DML events to monitor.]

5. Modify the EMPLOYEES table to satisfy the following requirements:
   When an employee record is created, the HireDate will automatically be assigned the system's current date. Furthermore, the employee must be at least 18 years old at the time of hire. Create a trigger to perform this task. If this age restriction is not followed, the record cannot be inserted.
   [ Note 1 : Use ***RAISE_APPLICATION_ERROR***

   Note 2: A direct insert will cause a run-time error which can be avoided using a stored procedurwhich contains a compiler directive
   ***PRAGMA EXCEPTION_INIT(exception_name, -Oracle_error_number); ]***

6. Add two more columns to the EMPLOYEES table, SALARY and SAL_GRADE. The SALARY column is the employee's monthly pay and the SAL_GRADE column will indicate the employee's salary grade. Create a lookup table to store the salary grade. Each salary grade will have a minimum and maximum salary amount. Create a trigger to ensure that an employee's salary is always within the range of his/her salary grade. Test this trigger with a few UPDATE statements to the salary.

7. The records in ORDER_DETAILS table cannot be deleted. Create a trigger to ensure that all transactions are preserved.
   (Do you need to create a trigger to protect the ORDERS records as well? Why or why not?)

8. Create a trigger that will display a warning message whenever an update on the credit limit of customers is performed on the last 3 days of a month.