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.

❖ **Use Of Database Triggers:**

- A trigger can permit DML statements against a table only if they are issued, during regular business hours or on predetermined weekdays.
- Used to keep an audit trail of a table.
- Prevent invalid transaction.
- Enforce complex security authorizations.

❖ **Database triggers V/S Procedures**

- Triggers do not accept parameters whereas procedures can.
- A trigger is executed impilcitly by Oracle engine.
- A procedure is executed explicitly.

❖ **How to apply database triggers**

A trigger has three basic parts:

1. A triggering event.
2. A trigger restriction.
3. A trigger action.

1. **Triggering event or statement:** It is a SQL statement that causes a trigger to be fired. Can be INSERT, UPDATE or DELETE statement.

2. **Trigger Restriction:** Specifies Boolean expression that must be **true** for the trigger to fire. It functions is to conditionally control the execution of a trigger. Specified in **WHEN** Clause.

3. **Trigger Action:** PL/SQL code to be executed when a triggering statement is encountered and any trigger restriction evaluates to TRUE.

Creating Triggers
Syntax :

```
CREATE [OR REPLACE ] TRIGGER <trigger_name>
        {BEFORE | AFTER | INSTEAD OF }
        {INSERT [OR] | UPDATE [OR] | DELETE}[OF col_name]
ON <table_name>
        [REFERENCING OLD AS old NEW AS new]
        [FOR EACH ROW [WHEN (condition)]]
DECLARE
        <variable declarations>;
        <constant declarations>;
BEGIN
        <PL/SQL Subprogram body>;
EXCEPTION
        <Exception PL/SQL block>;
END;
```

Where,
• **CREATE [OR REPLACE] TRIGGER trigger_name :** Creates or replace 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.

❖ Benefits of Triggers

- 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

For Example: The price of a product changes constantly. It is important to maintain the history of the prices of the products.

We can create a trigger to update the 'product_price_history' table when the price of the product is updated in the 'product' table.

1) **Create the 'product' table and 'product_price_history' table**

```
CREATE TABLE product_price_history
(product_id number(5),
product_name varchar2(32),
supplier_name varchar2(32),
unit_price number(7,2) );

CREATE TABLE product
(product_id number(5),
product_name varchar2(32),
supplier_name varchar2(32),
unit_price number(7,2) );
```

2) **Create the price_history_trigger and execute it.**

```
CREATE or REPLACE TRIGGER p1
BEFORE UPDATE OF unit_price
ON product
FOR EACH ROW
BEGIN
```

```
        INSERT INTO product_price_history
        VALUES (:old.product_id,:old.product_name,:old.supplier_name,:old.unit_price);
        END;/
```

3) Lets update the price of a product.
update product set unit_price = 250 where product_id = 101;

Once the above update query is executed, the trigger fires and updates the 'product_price_history' table.

4) If you ROLLBACK the transaction before committing to the database, the data inserted to the table is also rolled back.

❖ <u>Types of PL/SQL Triggers</u>

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 updated, 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 event will 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_Date number(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       19-Sep-2013
Before update row level             19-Sep-2013
After update row level              19-Sep-2013
Before update row level             19-Sep-2013
After update row level              19-Sep-2013
After update, statement level       19-Sep-2013
```

The above result shows 'before update' and 'after update' row level events have occurred 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.

❖ Working with Views

For our next example, we will need to create a view (of PERSON table):

```
        CREATE OR REPLACE
        VIEW PERSON_VIEW AS
        SELECT NAME FROM PERSON;
```

Now, we know that updating (or inserting) into a view is kind of pointless; however, we can Provide this functionality using a trigger!

For example:

```
        CREATE OR REPLACE TRIGGER PERSON_VIEW_INSERT
        INSTEAD OF INSERT ON PERSON_VIEW
        FOR EACH ROW
        BEGIN
                DBMS_OUTPUT.PUT_LINE('INSERTING: ' || :NEW.NAME);
                -- we can also do
                -- INSERT INTO PERSON(ID,NAME,DOB) VALUES
        (N,:NEW.NAME,SYSDATE);
END;/
```

When we do an insert statement on PERSON_VIEW:

```
INSERT INTO PERSON_VIEW(NAME) VALUES ('SUPERMAN');
```

Which produces the result:

```
INSERTING: SUPERMAN
```

❖ Trigger Exceptions (introduction)

- Sometimes, an error (or exception) is raised for a good reason. For example, to prevent some action that improperly modifies the database.
- Let's say that our database should not allow anyone to modify their DOB (after the person is in the database, their DOB is assumed to be static). Anyway, we can create a trigger that would prevent us from updating the DOB:

```
        CREATE OR REPLACE TRIGGER PERSON_DOB
        BEFORE UPDATE OF DOB ON PERSON
        FOR EACH ROW
        BEGIN
                RAISE_APPLICATION_ERROR(-20000,'CANNOT CHANGE DATE OF
        BIRTH');
        END;
```

Notice the format of the trigger declaration. We explicitly specify that it will be called **BEFORE UPDATE OF DOB ON PERSON.**

When we do the actual update for example:

```
UPDATE PERSON SET DOB = SYSDATE;
```

We end up with an error, that says we CANNOT CHANGE DATE OF BIRTH.

```
SQL> UPDATE PERSON SET DOB = SYSDATE;
UPDATE PERSON SET DOB = SYSDATE
*
ERROR at line 1:
```

```
ORA-20000: CANNOT CHANGE DATE OF BIRTH
ORA-06512: at "PARTICLE.PERSON_DOB", line 2
ORA-04088: error during execution of trigger 'PARTICLE.PERSON_DOB'
```

You should also notice the error code of ORA-20000. This is our -20000 parameter to RAISE APPLICATION ERROR.

❖ Enabling and disabling triggers

- A database trigger may be either in enabled state or disabled state.
- Disabling trigger my improve performance when a large amount of table data is to be modified, because the trigger code is not executed.
- Enabling or disabling can be done using alter statement

Syntax
ALTER TRIGGER TriggerName DISABLE/ENABLE;

For example,
UPDATE students set name = upper(name);

will run faster if all the triggers fired by UPDATE on STUDENTS table are disabled. Triggers can be disabled as follows:

ALTER TRIGGER payments_bu_row DISABLE;

A disabled trigger is not fired until it is enabled.
A disabled trigger can be enable with the use of following command.

ALTER TRIGGER payments_bu_row ENABLE;

The following command disables all triggers of STUDENTS table.

ALTER TABLE students DISABLE ALL TRIGGERS;
Dropping A Trigger:

You can drop a trigger using the following command.
DROP TRIGGER Triggername;

❖ 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'.

❖ <u>CYCLIC CASCADING in a TRIGGER</u>

- This is an undesirable situation where more than one trigger enters into an infinite loop.
- While creating a trigger we should ensure the such a situation does not exist.

- The below example shows how Trigger's can enter into cyclic cascading.
- Let's consider we have two tables 'abc' and 'xyz'. Two triggers are created.

1. The INSERT Trigger, triggerA on table 'abc' issues an UPDATE on table 'xyz'.

2. The UPDATE Trigger, triggerB on table 'xyz' issues an INSERT on table 'abc'.

- In such a situation, when there is a row inserted in table 'abc', trigger A fires and will update able 'xyz'.
- When the table 'xyz' is updated, triggerB fires and will insert a row in table 'abc'.

- This cyclic situation continues and will enter into a infinite loop, which will crash the database.