**Oracle PL/SQL Trigger Tutorial: Instead of, Compound [Example]**

**What are Triggers in PL/SQL?**

Triggers are stored programs that are fired automatically when some events occur. The code to be fired can be defined as per the requirement.

Oracle has also provided the facility to mention the event upon which the trigger needs to be fire and the timing of the execution.

In this tutorial, you will learn-

* [Benefits of Triggers](https://www.guru99.com/triggers-pl-sql.html#1)
* [Types of Triggers in Oracle](https://www.guru99.com/triggers-pl-sql.html#2)
* [How to Create Trigger](https://www.guru99.com/triggers-pl-sql.html#3)
* [:NEW and :OLD Clause](https://www.guru99.com/triggers-pl-sql.html#4)
* [INSTEAD OF Trigger](https://www.guru99.com/triggers-pl-sql.html#5)
* [Compound Trigger](https://www.guru99.com/triggers-pl-sql.html#6)

**Benefits of Triggers**

Following are the 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

**Types of Triggers in Oracle**

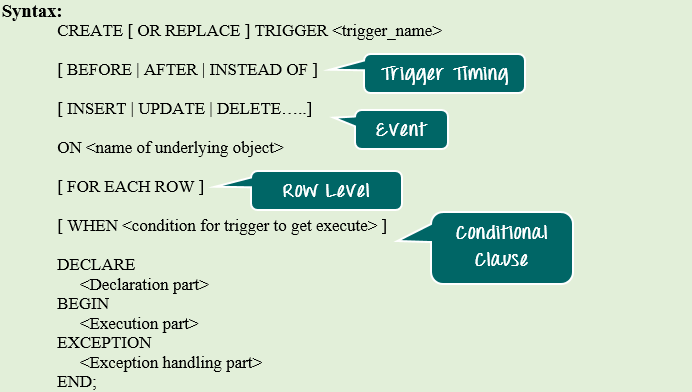
Triggers can be classified based on the following parameters.

* Classification based on the **timing**
  + BEFORE Trigger: It fires before the specified event has occurred.
  + AFTER Trigger: It fires after the specified event has occurred.
  + INSTEAD OF Trigger: A special type. You will learn more about the further topics. (only for DML )
* Classification based on the **level**
  + STATEMENT level Trigger: It fires one time for the specified event statement.
  + ROW level Trigger: It fires for each record that got affected in the specified event. (only for DML)
* Classification based on the**Event**
  + DML Trigger: It fires when the DML event is specified (INSERT/UPDATE/DELETE)
  + DDL Trigger: It fires when the DDL event is specified (CREATE/ALTER)
  + DATABASE Trigger: It fires when the database event is specified (LOGON/LOGOFF/STARTUP/SHUTDOWN)

So each trigger is the combination of above parameters.

**How to Create Trigger**

Below is the syntax for creating a trigger.

[](https://www.guru99.com/images/PL-SQL/110215_1121_TriggersinP1.png)

CREATE [ OR REPLACE ] TRIGGER <trigger\_name>

[BEFORE | AFTER | INSTEAD OF ]

[INSERT | UPDATE | DELETE......]

ON<name of underlying object>

[FOR EACH ROW]

[WHEN<condition for trigger to get execute> ]

DECLARE

<Declaration part>

BEGIN

<Execution part>

EXCEPTION

<Exception handling part>

END;

**Syntax Explanation:**

* The above syntax shows the different optional statements that are present in trigger creation.
* BEFORE/ AFTER will specify the event timings.
* INSERT/UPDATE/LOGON/CREATE/etc. will specify the event for which the trigger needs to be fired.
* ON clause will specify on which object the above-mentioned event is valid. For example, this will be the table name on which the DML event may occur in the case of DML Trigger.
* Command "FOR EACH ROW" will specify the ROW level trigger.
* WHEN clause will specify the additional condition in which the trigger needs to fire.
* The declaration part, execution part, exception handling part is same as that of the other PL/SQL blocks. Declaration part and exception handling part are optional.

**:NEW and :OLD Clause**

In a row level trigger, the trigger fires for each related row. And sometimes it is required to know the value before and after the DML statement.

Oracle has provided two clauses in the RECORD-level trigger to hold these values. We can use these clauses to refer to the old and new values inside the trigger body.

* :NEW – It holds a new value for the columns of the base table/view during the trigger execution
* :OLD – It holds old value of the columns of the base table/view during the trigger execution

This clause should be used based on the DML event. Below table will specify which clause is valid for which DML statement (INSERT/UPDATE/DELETE).

|  |  |  |  |
| --- | --- | --- | --- |
|  | **INSERT** | **UPDATE** | **DELETE** |
| **:NEW** | VALID | VALID | INVALID. There is no new value in delete case. |
| **:OLD** | INVALID. There is no old value in insert case | VALID | VALID |

**INSTEAD OF Trigger**

"INSTEAD OF trigger" is the special type of trigger. It is used only in DML triggers. It is used when any DML event is going to occur on the complex view.

Consider an example in which a view is made from 3 base tables. When any DML event is issued over this view, that will become invalid because the data is taken from 3 different tables. So in this INSTEAD OF trigger is used. The INSTEAD OF trigger is used to modify the base tables directly instead of modifying the view for the given event.

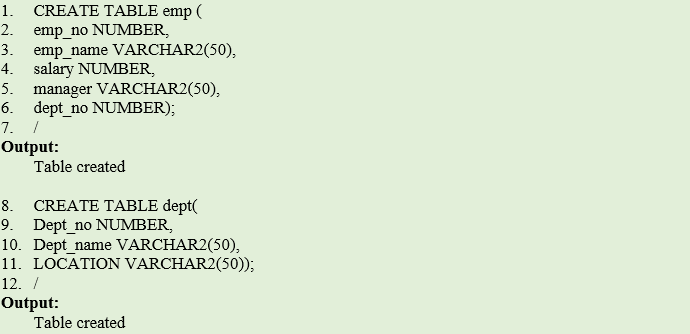
**Example 1**: In this example, we are going to create a complex view from two base table.

* Table\_1 is emp table and
* Table\_2 is department table.

Then we are going to see how the INSTEAD OF trigger is used to issue UPDATE the location detail statement on this complex view. We are also going to see how the :NEW and :OLD is useful in triggers.

* Step 1: Creating table 'emp' and 'dept' with appropriate columns
* Step 2: Populating the table with sample values
* Step 3: Creating view for the above created table
* Step 4: Update of view before the instead-of trigger
* Step 5: Creation of the instead-of trigger
* Step 6: Update of view after instead-of trigger

**Step 1)** Creating table 'emp' and 'dept' with appropriate columns

[](https://www.guru99.com/images/PL-SQL/110215_1121_TriggersinP2.png)

CREATE TABLE emp(

emp\_no NUMBER,

emp\_name VARCHAR2(50),

salary NUMBER,

manager VARCHAR2(50),

dept\_no NUMBER);

/

CREATE TABLE dept(

Dept\_no NUMBER,

Dept\_name VARCHAR2(50),

LOCATION VARCHAR2(50));

/

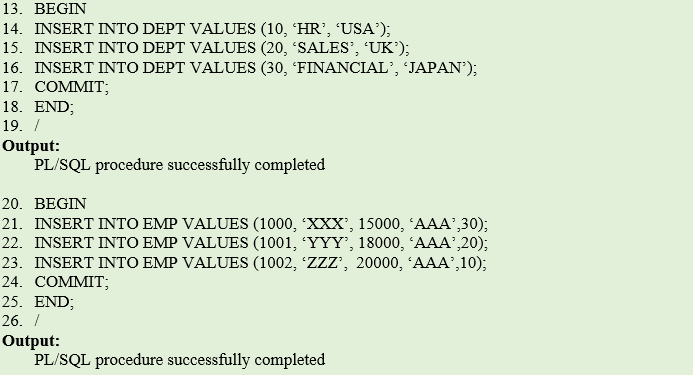
**Code Explanation**

* **Code line 1-7**: Table 'emp' creation.
* **Code line 8-12**: Table 'dept' creation.

**Output**

Table Created

**Step 2)** Now since we have created the table, we will populate this table with sample values and Creation of Views for the above tables.

[](https://www.guru99.com/images/PL-SQL/110215_1121_TriggersinP3.png)

BEGIN

INSERT INTO DEPT VALUES(10,‘HR’,‘USA’);

INSERT INTO DEPT VALUES(20,'SALES','UK’);

INSERT INTO DEPT VALUES(30,‘FINANCIAL',‘JAPAN');

COMMIT;

END;

/

BEGIN

INSERT INTO EMP VALUES(1000,'XXX5,15000,'AAA',30);

INSERT INTO EMP VALUES(1001,‘YYY5,18000,‘AAA’,20) ;

INSERT INTO EMP VALUES(1002,‘ZZZ5,20000,‘AAA',10);

COMMIT;

END;

/

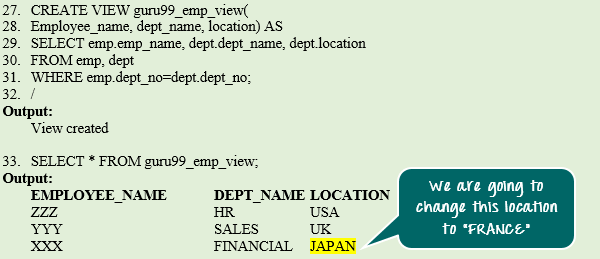
**Code Explanation**

* **Code line 13-19**: Inserting data into 'dept' table.
* **Code line 20-26:**Inserting data into 'emp' table.

**Output**

PL/SQL procedure completed

**Step 3)**Creating a view for the above created table.

[](https://www.guru99.com/images/PL-SQL/110215_1121_TriggersinP4.png)

CREATE VIEW guru99\_emp\_view(

Employee\_name:dept\_name,location) AS

SELECT emp.emp\_name,dept.dept\_name,dept.location

FROM emp,dept

WHERE emp.dept\_no=dept.dept\_no;

/

SELECT \* FROM guru99\_emp\_view;

**Code Explanation**

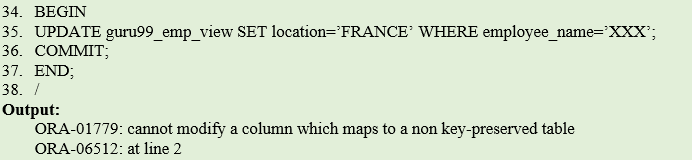
* **Code line 27-32:**Creation of 'guru99\_emp\_view' view.
* **Code line 33:** Querying guru99\_emp\_view.

**Output**

View created

|  |  |  |
| --- | --- | --- |
| **EMPLOYEE\_NAME** | **DEPT\_NAME** | **LOCATION** |
| ZZZ | HR | USA |
| YYY | SALES | UK |
| XXX | FINANCIAL | JAPAN |

**Step 4)** Update of view before instead-of trigger.

[](https://www.guru99.com/images/PL-SQL/110215_1121_TriggersinP5.png)

BEGIN

UPDATE guru99\_emp\_view SET location='FRANCE' WHERE employee\_name=:'XXX’;

COMMIT;

END;

/

**Code Explanation**

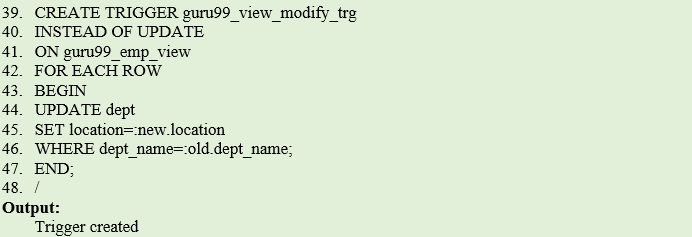
* **Code line 34-38:**Update the location of "XXX" to 'FRANCE'. It raised the exception because the DML statements are not allowed in the complex view.

**Output**

ORA-01779: cannot modify a column which maps to a non key-preserved table

ORA-06512: at line 2

**Step 5)**To avoid the error encounter during updating view in the previous step, in this step we are going to use "instead of trigger."

[](https://www.guru99.com/images/PL-SQL/110215_1121_TriggersinP6.png)

CREATE TRIGGER guru99\_view\_modify\_trg

INSTEAD OF UPDATE

ON guru99\_emp\_view

FOR EACH ROW

BEGIN

UPDATE dept

SET location=:new.location

WHERE dept\_name=:old.dept\_name;

END;

/

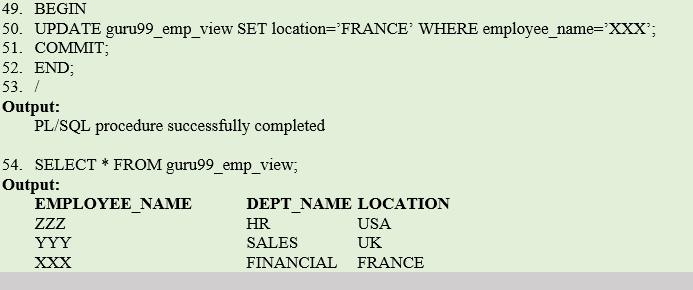
**Code Explanation**

* **Code line 39:** Creation of INSTEAD OF trigger for 'UPDATE' event on the 'guru99\_emp\_view' view at the ROW level. It contains the update statement to update the location in the base table 'dept'.
* **Code line 44:** Update statement uses ':NEW' and ': OLD' to find the value of columns before and after the update.

**Output**

Trigger Created

**Step 6)** Update of view after instead-of trigger. Now the error will not come as the "instead of trigger" will handle the update operation of this complex view. And when the code has executed the location of employee XXX will be updated to "France" from "Japan."

[](https://www.guru99.com/images/PL-SQL/110215_1121_TriggersinP7.png)

BEGIN

UPDATE guru99\_emp\_view SET location='FRANCE' WHERE employee\_name='XXX';

COMMIT;

END;

/

SELECT \* FROM guru99\_emp\_view;

**Code Explanation:**

* **Code line 49-53:** Update of the location of "XXX" to 'FRANCE'. It is successful because the 'INSTEAD OF' trigger has stopped the actual update statement on view and performed the base table update.
* **Code line 55:** Verifying the updated record.

**Output:**

PL/SQL procedure successfully completed

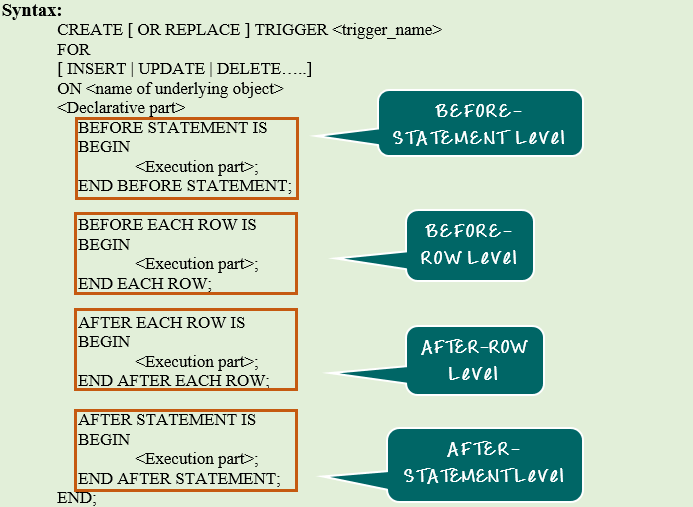
|  |  |  |
| --- | --- | --- |
| **EMPLOYEE\_NAME** | **DEPT\_NAME** | **LOCATION** |
| ZZZ | HR | USA |
| YYY | SALES | UK |
| XXX | FINANCIAL | FRANCE |

**Compound Trigger**

The Compound trigger is a trigger that allows you to specify actions for each of four timing points in the single trigger body. The four different timing point it supports is as below.

* BEFORE STATEMENT – level
* BEFORE ROW – level
* AFTER ROW - level
* AFTER STATEMENT – level

It provides the facility to combine the actions for different timing into the same trigger.

[](https://www.guru99.com/images/PL-SQL/110215_1121_TriggersinP8.png)

CREATE [ OR REPLACE ] TRIGGER <trigger\_name>

FOR

[INSERT | UPDATE | DELET.......]

ON <name of underlying object>

<Declarative part>

BEFORE STATEMENT IS

BEGIN

<Execution part>;

END BEFORE STATEMENT;

BEFORE EACH ROW IS

BEGIN

<Execution part>;

END EACH ROW;

AFTER EACH ROW IS

BEGIN

<Execution part>;

END AFTER EACH ROW;

AFTER STATEMENT IS

BEGIN

<Execution part>;

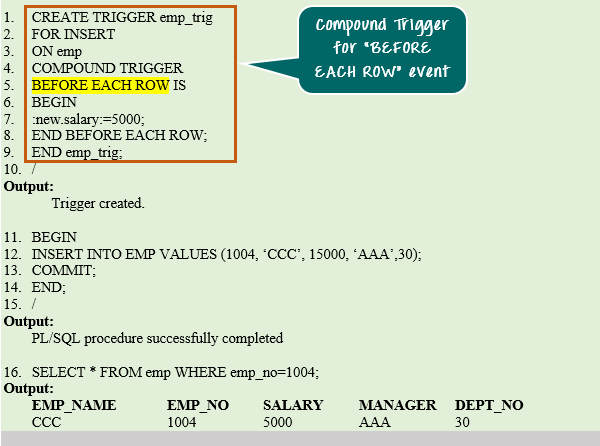
END AFTER STATEMENT;

END;

**Syntax Explanation:**

* The above syntax shows the creation of 'COMPOUND' trigger.
* Declarative section is common for all the execution block in the trigger body.
* These 4 timing blocks can be in any sequence. It is not mandatory to have all these 4 timing blocks. We can create a COMPOUND trigger only for the timings which are required.

**Example 1**: In this example, we are going to create a trigger to auto-populate the salary column with the default value 5000.

[](https://www.guru99.com/images/PL-SQL/110215_1121_TriggersinP9.png)

CREATE TRIGGER emp\_trig

FOR INSERT

ON emp

COMPOUND TRIGGER

BEFORE EACH ROW IS

BEGIN

:new.salary:=5000;

END BEFORE EACH ROW;

END emp\_trig;

/

BEGIN

INSERT INTO EMP VALUES(1004,‘CCC’,15000,‘AAA’,30);

COMMIT;

END;

/

SELECT \* FROM emp WHERE emp\_no=1004;

**Code Explanation:**

* **Code line 2-10**: Creation of compound trigger. It is created for timing BEFORE ROW- level to populate the salary with default value 5000. This will change the salary to default value '5000' before inserting the record into the table.
* **Code line 11-14**: Insert the record into 'emp' table.
* **Code line 16**: Verifying the inserted record.

**Output:**

Trigger created

PL/SQL procedure successfully completed.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **EMP\_NAME** | **EMP\_NO** | **SALARY** | **MANAGER** | **DEPT\_NO** |
| CCC | 1004 | 5000 | AAA | 30 |

**Enabling and Disabling Triggers**

Triggers can be enabled or disabled. To enable or disable the trigger, an ALTER (DDL) statement needs to be given for the trigger that disable or enable it.

Below are the syntax for enabling/disabling the triggers.

ALTER TRIGGER <trigger\_name> [ENABLE|DISABLE];

ALTER TABLE <table\_name> [ENABLE|DISABLE] ALL TRIGGERS;

**Syntax Explanation:**

* The first syntax shows how to enable/disable the single trigger.
* The second statement shows how to enable/disable all the triggers on a particular table.

**Summary**

In this chapter, we have learnt about PL/SQL triggers and their advantages. We have also learnt the different classifications and discussed INSTEAD OF trigger and COMPOUND trigger.