|  |
| --- |
| **Triggers and Transactions** |
| LAB MANUAL 09 |

What is an Oracle Trigger?

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

* A data manipulation language  (DML) statement executed against a table e.g., [INSERT](https://www.oracletutorial.com/oracle-basics/oracle-insert/), [UPDATE](https://www.oracletutorial.com/oracle-basics/oracle-update/), or [DELETE](https://www.oracletutorial.com/oracle-basics/oracle-delete/). For example, if you define a trigger that fires before an INSERT statement on the customers table, the trigger will fire once before a new row is inserted into the customers table.
* A data definition language (DDL) statement executes e.g., [CREATE](https://www.oracletutorial.com/oracle-basics/oracle-create-table/) or [ALTER](https://www.oracletutorial.com/oracle-basics/oracle-alter-table/) statement. These triggers are often used for auditing purposes to record changes of the schema.
* A system event such as [startup](https://www.oracletutorial.com/oracle-administration/oracle-startup/) or [shutdown](https://www.oracletutorial.com/oracle-administration/oracle-shutdown/) of the Oracle Database.
* A user event such as login or logout.

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

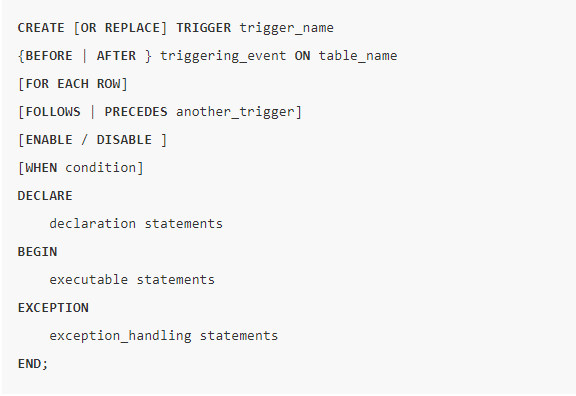
Oracle trigger usages

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

* Enforcing complex business rules that cannot be established using integrity constraint such as [UNIQUE](https://www.oracletutorial.com/oracle-basics/oracle-unique-constraint/), [NOT NULL](https://www.oracletutorial.com/oracle-basics/oracle-not-null/), and [CHECK](https://www.oracletutorial.com/oracle-basics/oracle-check-constraint/).
* Preventing invalid transactions.
* Gathering statistical information on table accesses.
* Generating value automatically for derived columns.
* Auditing sensitive data

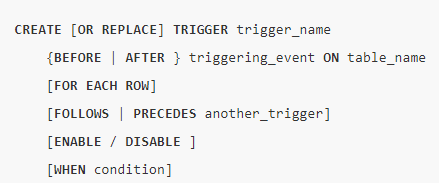
## How to create a trigger in Oracle

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

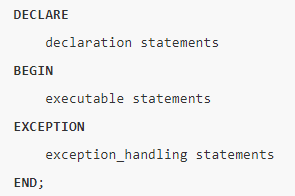


Let’s examine the syntax of the CREATE TRIGGER statement in more detail.

A trigger has two main parts: header and body. The following illustrates the trigger header:



And this is the trigger body:



As you can see, the trigger body has the same structure as an [anonymous PL/SQL block](https://www.oracletutorial.com/plsql-tutorial/plsql-anonymous-block/).

### 1) CREATE OR REPLACE

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

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

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



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

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



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



### 2) Trigger name

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

### 3) BEFORE | AFTER

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

### 4) ON table\_name

The table\_name is the name of the table associated with the trigger.

### 5) FOR EACH ROW

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

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

### 6) ENABLE / DISABLE

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

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

### 7) FOLLOWS | PRECEDES another\_trigger

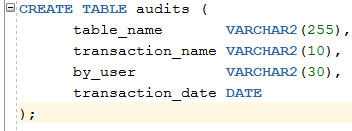
For each triggering event e.g., INSERT, UPDATE, or DELETE, you can define multiple triggers to fire. In this case, you need to specify the firing sequence using the FOLLOWS or PRECEDES option.

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

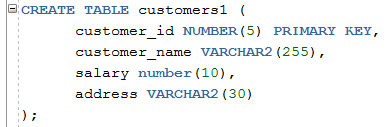
## Creating an Oracle trigger example

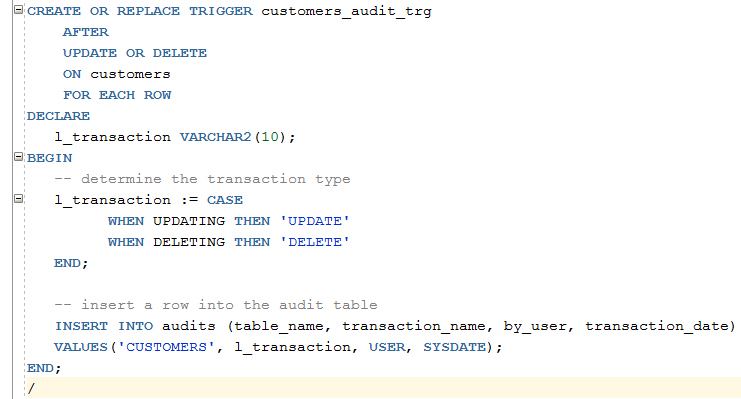
Suppose we want to record actions against the customers table whenever a customer is updated or deleted. In order to do this:

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



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





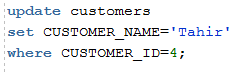
The following clause:



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

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

The following statement updates the Name of the customer Farah to Tahir.



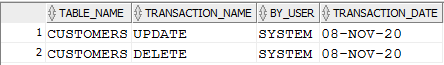
Now, check the contents of the table audits to see if the trigger was fired:



This DELETE statement deletes a row from the customers table.



And view the data of the audits table:



The output showed that a new row has been inserted. It means that the DELETE action fired the trigger customer\_audit\_trg.

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

## Introduction to Oracle Statement-level triggers

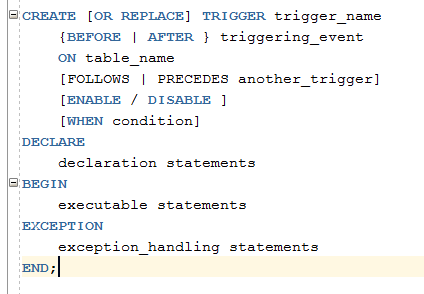
A statement-level trigger is fired whenever a trigger event occurs on a table regardless of how many rows are affected. In other words, a statement-level trigger executes once for each transaction.

For example, if you update 1000 rows in a table, then a statement-level trigger on that table would only be executed once.

Due to its features, a statement-level trigger is not often used for data-related activities like auditing the changes of the data in the associated table. It’s typically used to enforce extra security measures on the kind of transaction that may be performed on a table.

By default, the statement CREATE TRIGGER creates a statement-level trigger when you omit the FOR EACH ROW clause.

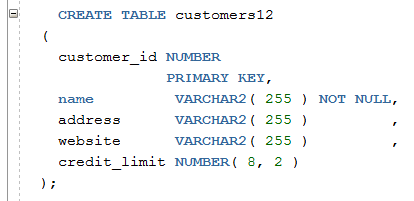
Here is the basic syntax of creating a statement-level trigger:



Note that the meanings of each clause are already explained in the trigger section:

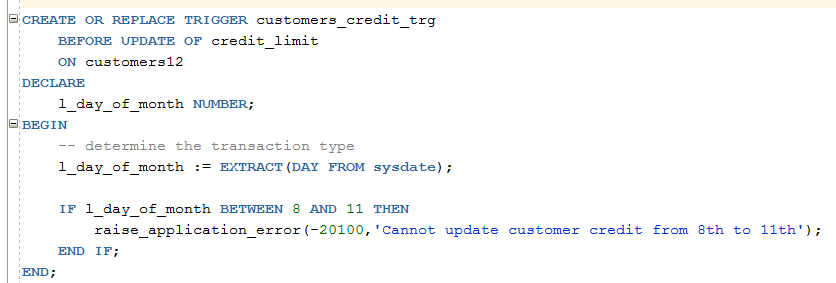
Oracle Statement-level Trigger example

We’ll use the table customers table given



Suppose, you want to restrict users to update credit of customers from 8th to 11th of every month so that you can close the financial month

To enforce this rule, you can use this statement-level trigger:



Let’s examine the trigger.

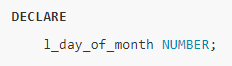
First, create a new trigger customers\_credit\_trg. The OR REPLACE modifies the trigger if it already exists:



Next, instruct the Oracle to fire the trigger only before update event for the credit\_limit column of the customers table. If you update values in other columns rather than the credit\_limit column, the trigger will not execute.



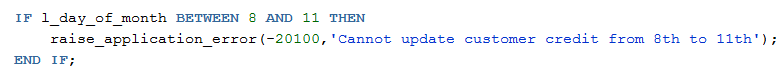
Then, declare a variable to hold the current day of the month:



After that, get the current day of the month using the [EXTRACT()](https://www.oracletutorial.com/oracle-date-functions/oracle-extract/) function:

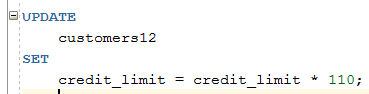


Finally, check if the current day of the month is between 8th and 11th; use the procedure raise\_application\_error to raise a user-defined error:

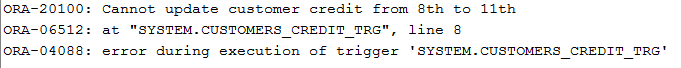


### Testing the Oracle statement-level trigger

The following statement uses the UPDATE statement to increase the credit limit of all customer 10%:



Oracle issued the following error:



# Transactions

In this section, we will discuss the transactions in PL/SQL. A database **transaction** is an atomic unit of work that may consist of one or more related SQL statements. It is called atomic because the database modifications brought about by the SQL statements that constitute a transaction can collectively be either committed, i.e., **made permanent to the database or rolled back (undone) from the database.**

A successfully executed SQL statement and a committed transaction are not same. Even if an SQL statement is executed successfully, unless the transaction containing the statement is committed, it can be rolled back and all changes made by the statement(s) can be undone.

Starting and Ending a Transaction

A transaction has a **beginning** and an **end**. A transaction starts when one of the following events take place −

* The first SQL statement is performed after connecting to the database.
* At each new SQL statement issued after a transaction is completed.

A transaction ends when one of the following events take place −

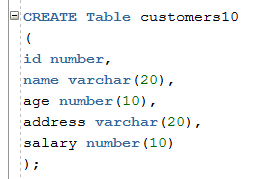
* A **COMMIT** or a **ROLLBACK** statement is issued.
* A **DDL** statement, such as **CREATE TABLE** statement, is issued; because in that case a COMMIT is automatically performed.
* A **DCL** statement, such as a **GRANT** statement, is issued; because in that case a COMMIT is automatically performed.
* User disconnects from the database.
* User exits from **SQL\*PLUS** by issuing the **EXIT** command, a COMMIT is automatically performed.
* SQL\*Plus terminates abnormally, a **ROLLBACK** is automatically performed.
* A **DML** statement fails; in that case a ROLLBACK is automatically performed for undoing that DML statement.

**Committing a Transaction**

A transaction is made permanent by issuing the SQL command COMMIT. The general syntax for the COMMIT command is −

**COMMIT;**

For example,





**Rolling Back Transactions**

Changes made to the database without COMMIT could be undone using the ROLLBACK command.

The general syntax for the ROLLBACK command is −



When a transaction is aborted due to some unprecedented situation, like system failure, the entire transaction since a commit is automatically rolled back. If you are not using **savepoint**, then simply use the following statement to rollback all the changes –



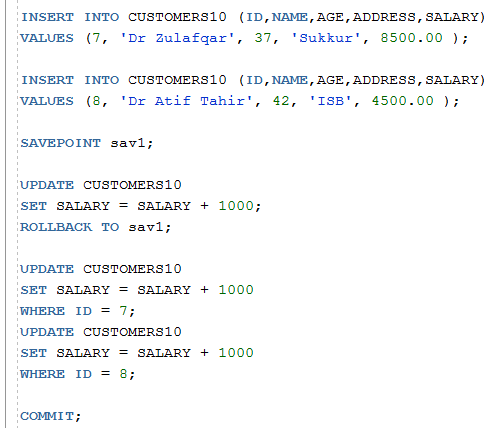
### Savepoints

Savepoints are sort of markers that help in splitting a long transaction into smaller units by setting some checkpoints. By setting Savepoints within a long transaction, you can roll back to a checkpoint if required. This is done by issuing the **SAVEPOINT** command.

The general syntax for the SAVEPOINT command is –



For example



**ROLLBACK TO sav1** − this statement rolls back all the changes up to the point, where you had marked Savepoints sav1.

After that, the new changes that you make will start.

## Automatic Transaction Control

To execute a **COMMIT** automatically whenever an **INSERT, UPDATE** or **DELETE** command is executed, you can set the **AUTOCOMMIT** environment variable as −



You can turn-off the auto commit mode using the following command −

