The following material has been copied from oracle documentation.

**PL/SQL Triggers**

**Overview of Triggers**

A database trigger is a stored procedure associated with a **database table, view, or event**. The trigger can be called **once**, **when some event occurs**, or many times, **once for each row affected** by an INSERT, UPDATE, or DELETE statement. The trigger can be called **after** the event, to record it, or take some follow-up action. The trigger can be called **before** the event, to prevent erroneous operations or fix new data so that it conforms to business rules. The executable part of a trigger can contain procedural statements and SQL data manipulation statements.

Triggers are created using the SQL CREATE TRIGGER statement. This statement can be used with Object Browser, SQL Script Editor, or SQL Command Line (SQL\*Plus). The CREATE (or CREATE OR REPLACE) statement fails if any errors exist in the PL/SQL block.

**Types of Triggers**

A trigger can be a stored PL/SQL or C procedure associated with a table, view, schema, or the database itself. Oracle automatically executes a trigger when a specified event takes place, which usually is a DML statement being issued against the table. The types of triggers are:

* DML triggers on tables
* INSTEAD OF triggers on views
* System triggers on DATABASE or SCHEMA

You can create triggers to be fired on any of the following:

* DML statements (DELETE, INSERT, UPDATE)
* DDL statements (CREATE, ALTER, DROP)
* Database operations (LOGON, LOGOFF)

**Naming Triggers**

Trigger names must be unique with respect to other triggers in the same schema. Trigger names do not need to be unique with respect to other schema objects, such as tables, views, and procedures. For example, a table and a trigger can have the same name; however, to avoid confusion, this is not recommended.

**When Is a Trigger Fired?**

A trigger is fired based on a triggering statement, which specifies:

* The SQL statement or the system event, database event, or DDL event that fires the trigger body. The options include DELETE, INSERT, and UPDATE. One, two, or all three of these options can be included in the triggering statement specification.
* The table, view, database, or schema associated with the trigger.

If a trigger contained the following statement:

AFTER DELETE OR INSERT OR UPDATE ON employees ...

then any of the following statements would fire the trigger:

DELETE FROM employees WHERE ...;  
INSERT INTO employees VALUES ( ... );  
INSERT INTO employees SELECT ... FROM ... ;  
UPDATE employees SET ... ;

An UPDATE statement might include a list of columns. If a triggering statement includes a column list, the trigger is fired only when one of the specified columns is updated. If a triggering statement omits a column list, the trigger is fired when any column of the associated table is updated. A column list cannot be specified for INSERT or DELETE triggering statements. In Example 1, the audit\_sal trigger specifies the salary column, and is only fired after an UPDATE of the salary of an employee in the employees table. Updates of other columns would not fire the trigger.

***Example 1 Creating a Database Trigger with the AFTER Option***

-- create a table to use for with the trigger in this example if

-- it has not already been created previously

-- if the table does not exist, the trigger will be invalid

CREATE TABLE emp\_audit ( emp\_audit\_id NUMBER(6), up\_date DATE,

new\_sal NUMBER(8,2), old\_sal NUMBER(8,2) );

-- create or replace the trigger

CREATE OR REPLACE **TRIGGER** **audit\_sal**

AFTER UPDATE OF salary ON employees FOR EACH ROW

BEGIN

-- bind variables are used here for values

INSERT INTO emp\_audit VALUES( :OLD.employee\_id, SYSDATE,

:NEW.salary, :OLD.salary );

END;

/

-- fire the trigger with an update of salary

UPDATE employees SET salary = salary \* 1.01 WHERE manager\_id = 122;

-- check the audit table to see if trigger was fired

SELECT \* FROM emp\_audit;

**Controlling When a Trigger Is Fired**

**Firing Triggers With the BEORE and AFTER Options**

The BEFORE or AFTER option in the CREATE TRIGGER statement specifies exactly when to fire the trigger body in relation to the triggering statement that is being run. In a CREATE TRIGGER statement, the BEFORE or AFTER option is specified just before the triggering statement.

In general, you use BEFORE or AFTER triggers to achieve the following results:

* Use a BEFORE row trigger to modify the row before the row data is written to disk. See Example 2 for an example of a BEFORE trigger.
* Use an AFTER row trigger to obtain and perform operations using the row ID. See Example 1 for an example of an AFTER trigger.

***Example 2 Creating a Database Trigger With the BEFORE Option***

-- create a temporary table

CREATE TABLE emp\_sal\_log (emp\_id NUMBER, log\_date DATE,

new\_salary NUMBER, action VARCHAR2(50));

CREATE OR REPLACE TRIGGER log\_salary\_increase -- create a trigger

BEFORE UPDATE of salary ON employees FOR EACH ROW

WHEN (OLD.salary < 8000)

BEGIN

INSERT INTO emp\_sal\_log (emp\_id, log\_date, new\_salary, action)

VALUES (:NEW.employee\_id, SYSDATE, :NEW.salary, 'New Salary');

END;

/

-- update the salary with the following UPDATE statement

-- trigger fires for each row that is udpated

UPDATE employees SET salary = salary \* 1.01 WHERE department\_id = 60;

-- view the log table

SELECT \* FROM emp\_sal\_log;

**Note:**

BEFORE row triggers are slightly more efficient than AFTER row triggers. With AFTER row triggers, affected data blocks must be read (logical read, not physical read) once for the trigger and then again for the triggering statement. Alternatively, with BEFORE row triggers, the data blocks must be read only once for both the triggering statement and the trigger.

If an UPDATE or DELETE statement detects a conflict with a concurrent UPDATE statement, then Oracle performs a transparent ROLLBACK and restarts the update operation. This can occur many times before the statement completes successfully. Each time the statement is restarted, the BEFORE statement trigger is fired again. The rollback does not undo changes to any package variables referenced in the trigger. Your package should include a counter variable to detect this situation.

**Firing Triggers With the FOR EACH ROW Option**

The FOR EACH ROW option determines whether the trigger is a row trigger or a statement trigger. If you specify FOR EACH ROW, then the trigger fires once for each row of the table that is affected by the triggering statement. These triggers are referred to as row-level triggers. See the use of FOR EACH ROW in Example 1 and Example 2.

The absence of the FOR EACH ROW option indicates that the trigger fires only once for each applicable statement, but not separately for each row affected by the statement. These triggers are referred to as statement-level triggers and are useful for performing validation checks for the entire statement. In [Example 6-6](https://docs.oracle.com/cd/B25329_01/doc/appdev.102/b25108/xedev_triggers.htm#BABIIBAA), the trigger fires only once for each update of the employees table.

**Firing Triggers Based on Conditions (WHEN Clause)**

An optional trigger restriction can be included in the definition of a row trigger by specifying a Boolean SQL expression in a WHEN clause.

If included, the expression in the WHEN clause is evaluated for each row that the trigger affects. If the expression evaluates to TRUE for a row, then the trigger body is fired on behalf of that row. Otherwise, if the expression evaluates to FALSE, the trigger body is not fired. See Example 2 for an example of the use of the WHEN clause in a trigger.

The expression in a WHEN clause must be a SQL expression, and it cannot include a subquery. You cannot use a PL/SQL expression (including user-defined functions) in a WHEN clause. A WHEN clause cannot be included in the definition of a statement trigger.

**Firing Triggers With the INSTEAD OF Option**

Use the INSTEAD OF option to fire the trigger instead of executing the triggering event. Unlike other types of triggers, Oracle Database engine fires the trigger instead of executing the triggering SQL DML statement.

With an INSTEAD OF trigger, you can run an UPDATE, INSERT, or DELETE statement on a complex view that otherwise could not be updated. Also, the trigger can be used to control how updates are performed on a view. The INSTEAD OF trigger runs transparently in the background to perform the correct actions on the underlying tables of the view. The INSTEAD OF option only can only be specified for a trigger created on a view and can only be activated for each row. INSTEAD OF triggers are valid for DML events on views. They are not valid for DDL or database events.

See ["Creating a Trigger With the INSTEAD OF Option"](https://docs.oracle.com/cd/B25329_01/doc/appdev.102/b25108/xedev_triggers.htm#BABIADGG).

**Accessing Column Values in Row Triggers**

Within a trigger body of a row trigger, the PL/SQL code and SQL statements have access to the old and new column values of the current row affected by the triggering statement. Two correlation names exist for every column of the table being modified. There is one for the old column value and one for the new column value. These columns in the table are identified by :OLD.*colum\_name* and :NEW.*column\_name*. The use of :NEW and :OLD is shown in Example 1 and Example 2.

Depending on the type of triggering statement, certain correlation names might not have any meaning:

* A trigger fired by an INSERT statement has meaningful access to new column values only. Because the row is being created by the INSERT operation, the old values are null.
* A trigger fired by an UPDATE statement has access to both old and new column values for both BEFORE and AFTER row triggers.
* A trigger fired by a DELETE statement has meaningful access to old (:OLD) column values only. Because the row no longer exists after the row is deleted, the new (:NEW) values are NULL and cannot be modified.

Old and new values are available in both BEFORE and AFTER row triggers. A new column value can be assigned in a BEFORE row trigger, but not in an AFTER row trigger (because the triggering statement takes effect before an AFTER row trigger is fired). If a BEFORE row trigger changes the value of NEW.column, then an AFTER row trigger fired by the same statement sees the change assigned by the BEFORE row trigger.

Correlation names can also be used in the Boolean expression of a WHEN clause. A colon (:) must precede the OLD and NEW qualifiers when they are used in a trigger body, but a colon is not allowed when using the qualifiers in the WHEN clause.

**Detecting the DML****O****per****ation That Fired a Trigger**

If more than one type of DML operation can fire a trigger, such as ON INSERT or UPDATE, the trigger body can use the conditional predicates INSERTING, DELETING, and UPDATING to check which type of statement fires the trigger.

Within the code of the trigger body, you can execute blocks of code depending on the kind of DML operation that fired the trigger. For an example of INSERTING and UPDATING predicates, see Example 3.

***Example 3 Creating a Trigger That Fires Only Once***

-- create a log table

CREATE TABLE emp\_update\_log (log\_date DATE, action VARCHAR2(50));

-- create a trigger

CREATE OR REPLACE TRIGGER log\_emp\_update

AFTER UPDATE OR INSERT ON employees

DECLARE

v\_action VARCHAR2(50);

BEGIN

IF UPDATING THEN

v\_action := 'A row has been updated in the employees table';

END IF;

IF INSERTING THEN

v\_action := 'A row has been inserted in the employees table';

END IF;

INSERT INTO emp\_update\_log (log\_date, action)

VALUES (SYSDATE, v\_action);

END;

/

-- fire the trigger with an update

UPDATE employees SET salary = salary \* 1.01 WHERE department\_id = 60;

INSERT INTO employees VALUES(14, 'Belden', 'Enrique', 'EBELDEN','555.111.2222',

'31-AUG-05', 'AC\_MGR', 9000, .1, 101, 110);

-- view the log table

SELECT \* FROM emp\_update\_log;

-- clean up: remove the inserted record

DELETE FROM employees WHERE employee\_id = 14;

In an UPDATE trigger, a column name can be specified with an UPDATING conditional predicate to determine if the named column is being updated. For example, assume a trigger is defined as the following:

CREATE OR REPLACE TRIGGER ...  
... UPDATE OF salary ON employees ...  
BEGIN  
... IF UPDATING ('salary') THEN ... END IF;  
...

The code in the THEN clause runs only if the triggering UPDATE statement updates the salary column. This way, the trigger can minimize its overhead when the column of interest is not being changed.

**Enabled and Disabled Trigger Modes**

This section discusses enabled and disabled triggers. A trigger can be in an enabled or disabled mode:

* An enabled trigger executes its trigger body if a triggering statement is entered and the trigger restriction (if any) evaluates to TRUE.
* A disabled trigger does not execute its trigger body, even if a triggering statement is entered and the trigger restriction (if any) evaluates to TRUE.

Disable a trigger if you do not want the trigger to execute, for example during maintenance activities on the database.

**Error Conditions and Exceptions in the Trigger Body**

If a predefined or user-defined error condition or exception occurs during the execution of a trigger body, then all effects of the trigger body, as well as the triggering statement, are rolled back unless the error is trapped by an exception handler. Therefore, a trigger body can prevent the execution of the triggering statement by raising an exception. User-defined exceptions are commonly used in triggers that enforce complex security authorizations or integrity constraints.

**Designing Triggers**

This section discusses the design of triggers.

This section contains the following topics:

**Guidelines for Triggers**

Use the following guidelines when designing triggers:

* Use triggers to guarantee that when a specific operation is performed, related actions are performed.
* Do not define triggers that duplicate features already built into Oracle Database engine. For example, do not define triggers to reject bad data if you can do the same checking through declarative integrity constraints.
* Limit the size of triggers. If the logic for a trigger requires more than 60 lines of PL/SQL code, it is better to include most of the code in a stored procedure, and call the procedure from the trigger. The size of the trigger cannot be more than 32K.
* Use triggers only for centralized, global operations that should be fired for the triggering statement, regardless of which user or database application issues the statement.
* Do not create recursive triggers. For example, creating an AFTER UPDATE statement trigger on the employees table that will then issue an UPDATE statement on the same employees table, will cause the trigger to fire recursively until it has run out of memory.
* Use triggers on the database judiciously. They are executed for every user, every time the event occurs on which the trigger is created.

**Restrictions for Creating Triggers**

When creating triggers with PL/SQL code, there are some restrictions that are not required for standard PL/SQL blocks. The following sections discuss these restrictions.

**SQL Statements Allowed in Trigger Bodies**

The body of a trigger can contain DML SQL statements. It can also contain SELECT statements, but they must be SELECT... INTO... statements or the SELECT statement in the definition of a cursor.

DDL statements are not allowed in the body of a trigger and transaction control statements are not allowed in a trigger. ROLLBACK, COMMIT, and SAVEPOINT statements cannot be used. For system triggers, CREATE, ALTER, and DROP TABLE statements and ALTER...COMPILE statements are allowed.

**Note:**

A procedure called by a trigger cannot run the previous transaction control statements because the procedure runs within the context of the trigger body.

Statements inside of a trigger can reference remote schema objects. However, pay special attention when calling remote procedures from within a local trigger. If a timestamp or signature mismatch is found during execution of the trigger, then the remote procedure is not run, and the trigger is invalidated.

**System Trigger Restrictions**

Only committed triggers are fired. For example, if you create a trigger that should be fired after all CREATE events, then the trigger itself does not fire after the creation, because the correct information about this trigger was not committed at the time when the trigger on CREATE events was fired.

For example, if you execute the following SQL statement, trigger my\_trigger is not fired after the creation of my\_trigger. Oracle Database engine does not fire a trigger that is not committed.

CREATE OR REPLACE TRIGGER my\_trigger  
  AFTER CREATE ON DATABASE  
BEGIN  
  NULL;  
END;

**Privileges Needed to Work with Triggers**

To create a trigger in your schema, you must have the CREATE TRIGGER system privilege, and one of the following:

* Own the table specified in the triggering statement
* Have the ALTER privilege for the table in the triggering statement
* Have the ALTER ANY TABLE system privilege

The CREATE TRIGGER system privilege is included in predefined RESOURCE role that has been granted to the user HR.

To create a trigger on a database, you must have the ADMINISTER DATABASE TRIGGER privilege. If this privilege is later revoked, then you can drop the trigger, but not alter it.

The object privileges to the schema objects referenced in the trigger body must be granted to the trigger owner explicitly (not through a role). The statements in the trigger body operate under the privilege domain of the trigger owner, not the privilege domain of the user issuing the triggering statement. This is similar to the privilege model for stored procedures.

**Managing Triggers in the Database**

Triggers are another type of database object that you can manage with Object Browser. You can also create and update triggers with the SQL Commands page or SQL Editor page.

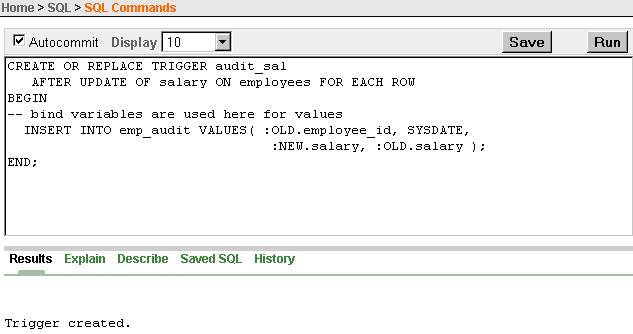
In addition, you can use SQL Command Line (SQL\*Plus) to create and update triggers.

**Creating a Trigger With the SQL Commands Page**

With the SQL Commands page, you can create and update triggers.

To create a trigger with the SQL Commands page:

1. To run the examples in this guide, you should log in as user HR with your password for the HR account.
2. On the home page, click the **SQL** icon to display the SQL page.
3. Click the **SQL** **Commands** icon to display the SQL Command page.
4. On the SQL Commands page, first enter the SQL statements to create any objects that are needed in the trigger body. For example, the emp\_audit table needs to be created before creating the audit\_sal trigger in Example 1. If a database object is referred to in the trigger code, then that object must exist for the trigger to be valid.
5. Click the **Run** button to execute the SQL statements to create any supporting objects for the trigger. If the statements run successfully, delete the statements from the SQL Commands page. Otherwise, update the statements so they run successfully.
6. On the SQL Commands page, enter the PL/SQL code to create the trigger after any objects that are needed by the trigger are created. For an example of code to create a trigger, see Example 1.

  
Click the **Run** button to execute the PL/SQL code to create the trigger. Correct the code if it does not execute successfully.

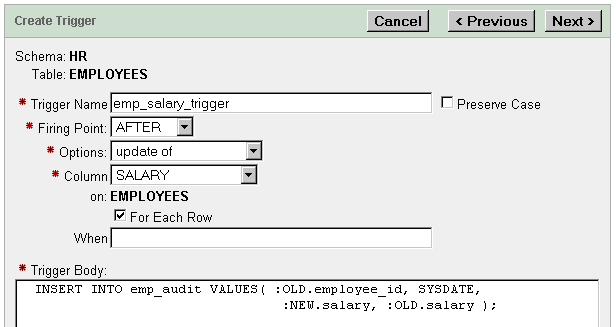
1. If you want to save the PL/SQL code for future use, click the **Save** button.
2. In the **Name** field, enter a name for the saved PL/SQL code. You can also enter an optional description. Click the **Save** button to complete the action.
3. To access the saved PL/SQL code, click the **Saved** **SQL** tab and select the name of the saved PL/SQL code that you want to access.

**Creating a Trigger with the Object Browser Page**

You can create and update triggers in the database with Object Browser.

To create a trigger with the Object Browser page:

1. To run the examples in this guide, log in as user HR with your password for the HR account.
2. Click the **Object Browser** icon on the Database Home Page.
3. Click the **Create** button, and select **Trigger** from the list.
4. Enter the name of the table (employees) that the trigger activity is based on and click the **Next** button. You can also select a table name from list.
5. In the **Trigger** **Name** field, enter the trigger name (emp\_salary\_trigger). The the **Preserve** **Case** box should be unchecked.
6. From the **Firing** **Point** list, select the firing point (AFTER).
7. From the **Options** list, select an option (update of).
8. From the **Column** list, select a column (salary).
9. Check the **For** **Each** **Row** option. Do not enter anything in the **When** field.
10. In the **Trigger** **Body** field, enter the code for the trigger body. See Example 1. Note that if a database object is referred to in the trigger body code, then that object must exist for the trigger to be valid.

  
[Description of the illustration xe\_create\_trigger.gif](https://docs.oracle.com/cd/B25329_01/doc/appdev.102/b25108/img_text/xe_create_trigger.htm)

1. Click the **Next** button.
2. Click the **SQL** button to view the SQL statements for creating the trigger.
3. Click the **Finish** button to complete the action.