Triggers:

- The keyword BEFORE indicates the trigger action time. In this case, the trigger activates before each row is inserted into the table. The other permitted keyword here is AFTER.

- The keyword INSERT indicates the trigger event; that is, the type of operation that activates the trigger. In the example, INSERT operations cause trigger activation. You can also create triggers for DELETE and UPDATE operations.

- If a BEFORE trigger fails, the operation on the corresponding row is not performed.

- A BEFORE trigger is activated by the ***attempt*** to insert or modify the row, regardless of whether the attempt subsequently succeeds.

- An AFTER trigger is executed only if any BEFORE triggers and the row operation executes successfully.

- An error during either a BEFORE or AFTER trigger results in failure of the entire statement that caused trigger invocation.

- For transactional tables, failure of a statement should cause rollback of all changes performed by the statement. Failure of a trigger causes the statement to fail, so trigger failure also causes rollback. For non-transactional tables, such rollback cannot be done, so although the statement fails, any changes performed prior to the point of the error remain in effect
-

Customer table:
create table customerTrigger (id int primary key,firstName varchar(20),lastName varchar(20), email varchar(100),billingCity varchar(100),shippingCity varchar(100));

Before Insert:
Create trigger beforeInsert before insert on customerTrigger
for each row

set new.shippingCity = new.billingCity ;

Before delete:

- BEFORE DELETE triggers are fired automatically before a delete event occurs in a table
- You can access the OLD row but cannot update it.
- There is no NEW row in the BEFORE DELETE trigger.
- If you have multiple statements in the trigger_body, you need to use the BEGIN END block to wrap these statements and temporarily change the default delimiter.

create table deletedCustomer (id int primary key,firstName varchar(20),lastName varchar(20), email varchar(100), billingCity varchar(100),shippingCity varchar(100), deletedAt timestamp default now());

create trigger before_delete_customer
before delete
on customer for each row
insert into deletedCustomer(id, firstName, lastName, email, billingCity, shippingCity)
values(old.id, old.firstName, old.lastName, old.email, old.billingCity, old.shippingCity);

delete from customer where id = 1;
*select * from deletedCustomer;*

After delete:

- AFTER DELETE triggers are automatically invoked after a delete event occurs on the table.
- You can access the OLD row but cannot change it.
- There is no NEW row in the AFTER DELETE trigger.

```
create table shippingCities (city varchar(20));

insert into shippingCities(city)
select shippingCity from customer;

select * from shippingCities;

create trigger after_delete_customer
after delete
on customer for each row
delete from shippingCities where city = old.shippingCity;
```

BEFORE UPDATE AND AFTER UPDATE:

- BEFORE UPDATE Trigger in MySQL is invoked automatically whenever an update operation is fired on the table associated with the trigger.
- SYNTAX is **CREATE TRIGGER** trigger_name

  BEFORE **UPDATE**

  ON table_name **FOR** EACH ROW

  trigger_body ;

- We cannot update the OLD values in a BEFORE UPDATE trigger. We cannot create a BEFORE UPDATE trigger on a VIEW.

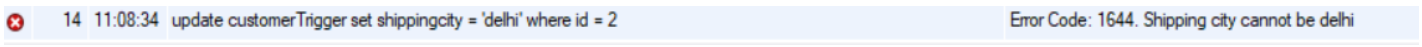
Example :

```
CREATE DEFINER=`root`@`localhost` TRIGGER `customertrigger_BEFORE_UPDATE` BEFORE UPDATE ON `customertrigger` FOR EACH ROW BEGIN

    DECLARE error_msg VARCHAR(200);

    SET error_msg = ('Shipping city cannot be delhi');

    IF new.shippingCity = 'delhi' THEN

    SIGNAL SQLSTATE '45000'

    SET MESSAGE_TEXT = error_msg;

    END IF;

END
```

In Query:

update customerTrigger set shippingcity = 'delhi' where id = 2;

Output:



AFTER UPDATE:

- The AFTER UPDATE trigger in MySQL is invoked automatically whenever an UPDATE event is fired on the table associated with the triggers.

Syntax:

```
CREATE TRIGGER trigger_name

AFTER UPDATE

ON table_name FOR EACH ROW
trigger_body ;
```

- We can access the OLD rows but cannot update them. We can access the NEW rows but cannot update them. We cannot create an AFTER UPDATE trigger on a **VIEW**.

Example:

```
CREATE DEFINER=`root`@`localhost` TRIGGER `customertrigger_AFTER_UPDATE` AFTER UPDATE
ON `customertrigger` FOR EACH ROW

BEGIN
INSERT into customerDetails VALUES (user(),
CONCAT('Update customer Record : ', OLD.firstname, ',billing city :',
OLD.billingcity, ' SHipping City : ', NEW.shippingcity));
END
```

In Query:

```
create table customerdetails(user varchar(200), details varchar(200));
update customerTrigger set shippingcity = 'jammu' where id = 2;
update customerTrigger set shippingcity = 'Kashmir' where id = 2;
Select * from customerdetails;
```

Output:

user	details
root@localhost	Update customer Record Meghna billing city :dehradun SHipping City : jammu
root@localhost	Updated customer Record :Meghna, Billing city :dehradun Shipping City : kashmir

Cursor:

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `getEmpNamesStartingWith`(in c varchar(1))
BEGIN
        declare finished int default 0;
        declare name varchar(255);
   declare allNames varchar(10000) default "";
        declare name_cursor
   cursor for
   select empName from employees where empName like concat(c,'%');
   DECLARE CONTINUE HANDLER
     FOR NOT FOUND SET finished = 1;
   open name_cursor;
   getNames: LOOP
                fetch name_cursor into name;
     if finished = 1 then
                        leave getNames;
```

```
                end if;
        set allNames = concat(name,", ",allNames);
            end LOOP;
    if allNames = "" then
                select "No names found" as result;
            else
                select allNames as result;
            end if;
END
```