|  |  |
| --- | --- |
| **Ex. No: 07** | **Triggers** |
| Date | 3-10-2023 |

**Objective:**

To solve the given problems using triggers.

**Software Required:**

Oracle 10g

**Description:**

Triggers are similar to stored procedures. A trigger stored in the database can include SQL and PL/SQL or Java statements to run as a unit and can invoke stored procedures. However, procedures and triggers differ in the way that they are invoked. A procedure is explicitly run by a user, application, or trigger.

**Detailed Procedure:**

The events that fire a trigger include the following:

DML statements that modify data in a table (INSERT, UPDATE, or DELETE) DDL statements System events such as startup, shutdown, and error messages User events such as logon and logoff A trigger has three basic parts:A triggering event or statement ,trigger restriction , trigger has three basic parts:riggering event or statement ,trigger restriction A trigger action

The types of triggers are:

BEFORE and AFTER Triggers

INSTEAD OF Triggers

Triggers on System Events and User Events

A database trigger is procedural code that is automatically executed in response to certain events on a particular table or view in a database. The trigger is mostly used for keeping the integrity of the information on the database. For example, when a new record (representing a new worker) is added to the employees table, new records should be created also in the tables of the taxes, vacations, and salaries.

**Sample Input /Output**

1. Create a trigger T1\_sal that prints “Salary incremented “whenever there is a increase in salary and “salary decremented “whenever there is a decrease in salary in employees table

create or replace trigger t1\_sal

after update of salary on employee

for each row

Begin

if (:new.salary>:old.salary) then

dbms\_output.put\_line('Salary Incremented');

elsif (:new.salary<:old.salary) then

dbms\_output.put\_line('Salary Decremented');

end if;

end;

2. Create a trigger T2 \_error that raises an error whenever there is no commission for manager for new insertions

create or replace trigger t2\_error

after insert on emp

for each row

begin

if(:new.comm is null) then

raise\_application\_error(-20000,'commission not given');

end if;

end;

3. Create a trigger T3 \_set\_null to set the manager id null in the departments table whenever the manager id is deleted from the employees table.

create or replace trigger t3\_set\_null

after delete on employee

for each row

begin

update departments set manager\_id=null where manager\_id=:old.manager\_id;

end;

**Question:**

1. **Create a BEFORE INSERT Trigger for the “User” Table that ensures that the passwords are at least 8 Characters.**

**CREATE OR REPLACE TRIGGER before\_insert\_password\_check**

**BEFORE INSERT ON User\_1128**

**FOR EACH ROW**

**BEGIN**

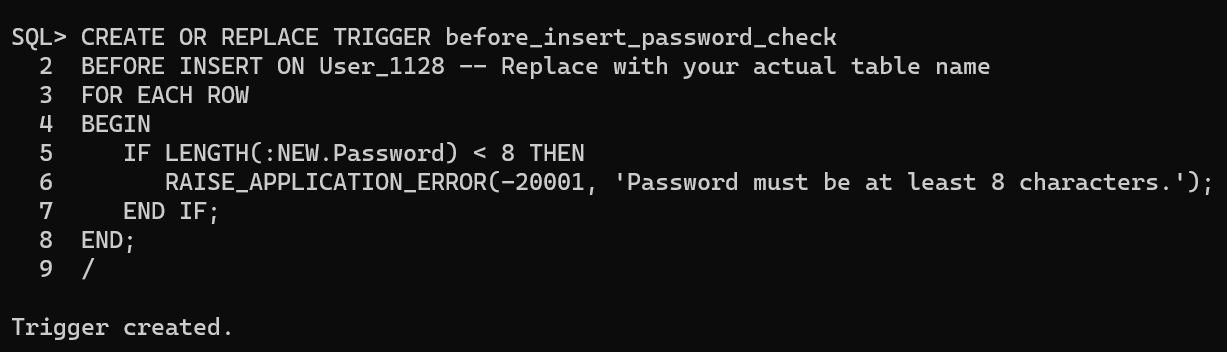
**IF LENGTH(:NEW.Password) < 8 THEN**

**RAISE\_APPLICATION\_ERROR(-20001, 'Password must be at least 8 characters.');**

**END IF;**

**END;**

**/**

****

1. **Create a BEFORE UPDATE Trigger for the “User” Table that does not allow email addresses to be null.**

**CREATE OR REPLACE TRIGGER before\_update\_email\_check**

**BEFORE UPDATE ON User\_1128**

**FOR EACH ROW**

**BEGIN**

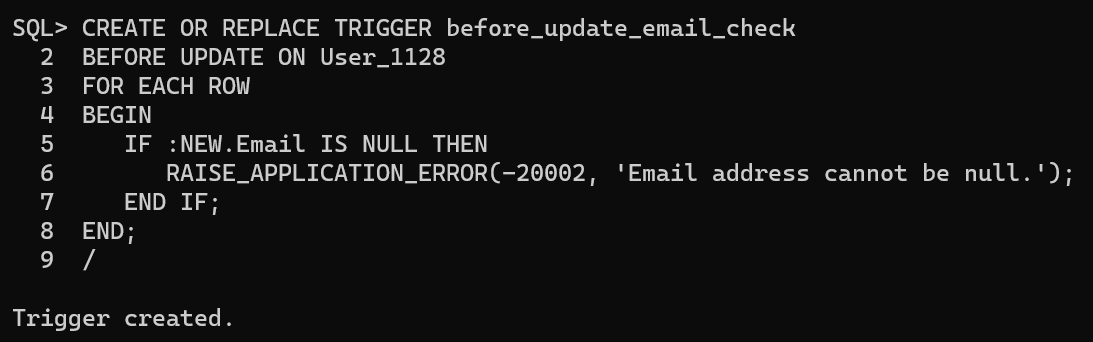
**IF :NEW.Email IS NULL THEN**

**RAISE\_APPLICATION\_ERROR(-20002, 'Email address cannot be null.');**

**END IF;**

**END;**

**/**

****

1. **Create a BEFORE DELETE Trigger for the “User” Table that prevents the deletion of users with specific email domains (like "example.com").**

**CREATE OR REPLACE TRIGGER before\_delete\_email**

**BEFORE DELETE ON User\_1128**

**FOR EACH ROW**

**BEGIN**

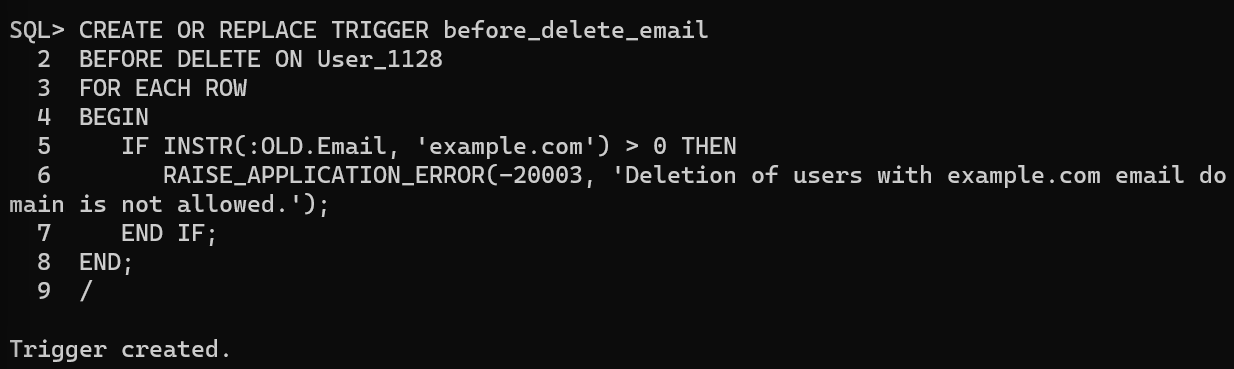
**IF INSTR(:OLD.Email, 'example.com') > 0 THEN**

**RAISE\_APPLICATION\_ERROR(-20003, 'Deletion of users with example.com email domain is not allowed.');**

**END IF;**

**END;**

**/**

****

1. **Write an AFTER INSERT trigger to count number of new tuples inserted using each**

**CREATE OR REPLACE TRIGGER after\_insert\_count\_tuples**

**AFTER INSERT ON User\_1128**

**DECLARE**

**total\_new\_rows NUMBER;**

**BEGIN**

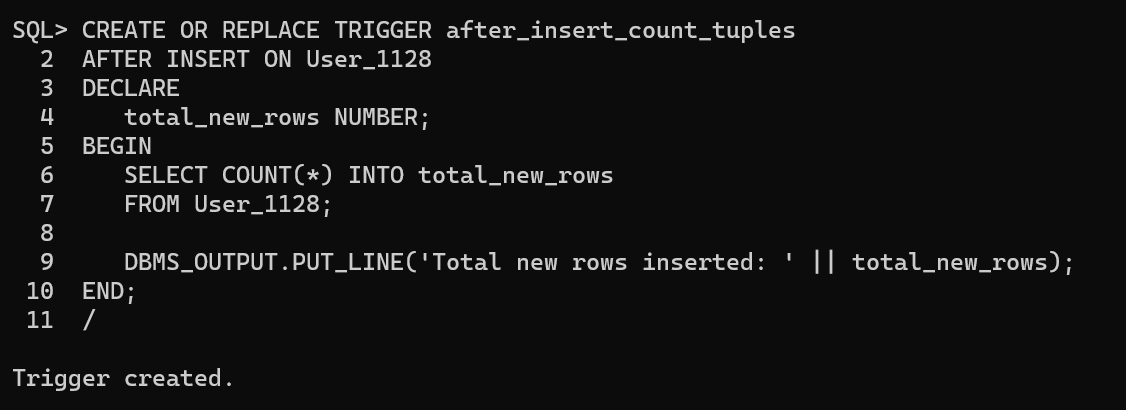
**SELECT COUNT(\*) INTO total\_new\_rows**

**FROM User\_1128;**

**DBMS\_OUTPUT.PUT\_LINE('Total new rows inserted: ' || total\_new\_rows);**

**END;**

**/**

****

1. **Create an AFTER UPDATE Trigger for the “User” Table that signals when a user's email is changed.**

**CREATE OR REPLACE TRIGGER after\_update\_email**

**AFTER UPDATE OF Email ON User\_1128**

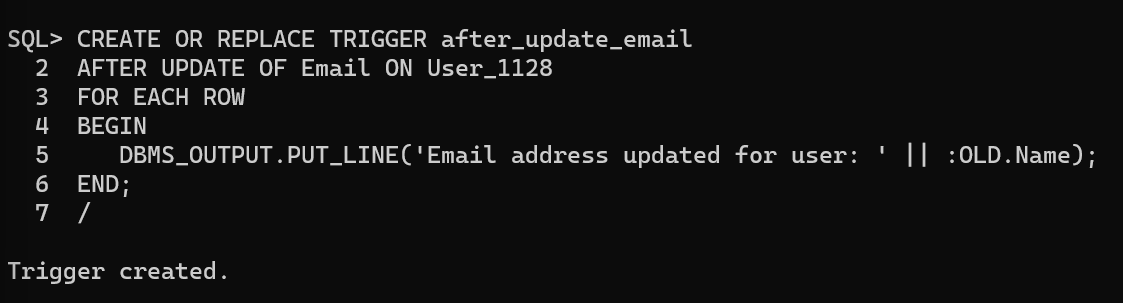
**FOR EACH ROW**

**BEGIN**

**DBMS\_OUTPUT.PUT\_LINE('Email address updated for user: ' || :OLD.Name);**

**END;**

**/**

****

1. **Create an AFTER DELETE Trigger for the “User” Table that signals when a user is deleted.**

**CREATE OR REPLACE TRIGGER after\_delete\_user\_signal**

**AFTER DELETE ON User\_1128**

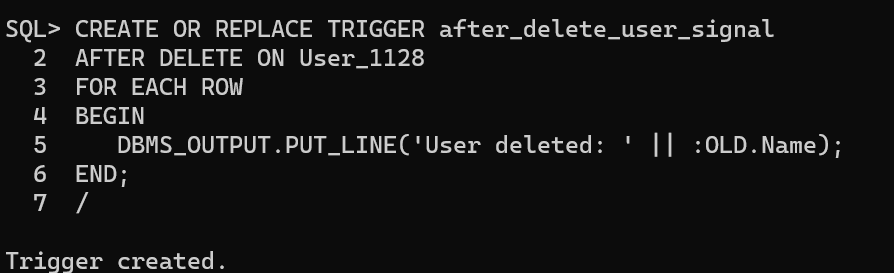
**FOR EACH ROW**

**BEGIN**

**DBMS\_OUTPUT.PUT\_LINE('User deleted: ' || :OLD.Name);**

**END;**

**/**

****

1. **Create a BEFORE INSERT Trigger for the “Event” Table that** **ensures the event's date is in the future.**

**CREATE OR REPLACE TRIGGER before\_insert\_future\_event**

**BEFORE INSERT ON Event\_1128**

**FOR EACH ROW**

**BEGIN**

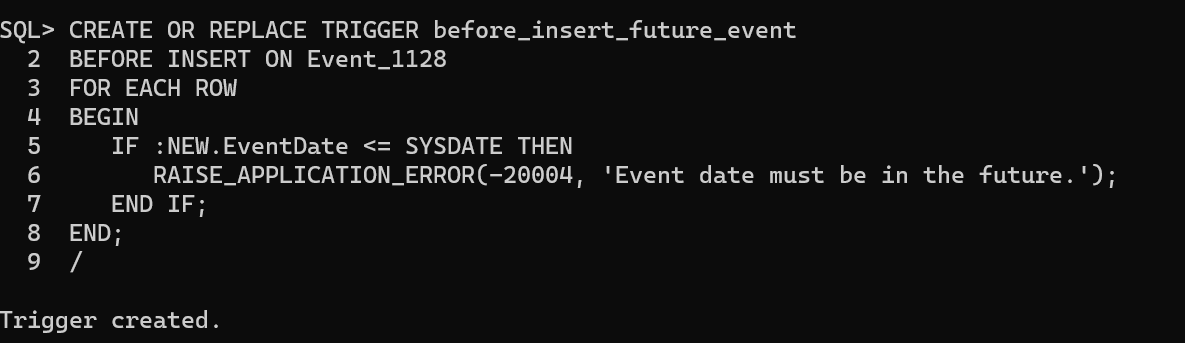
**IF :NEW.EventDate <= SYSDATE THEN**

**RAISE\_APPLICATION\_ERROR(-20004, 'Event date must be in the future.');**

**END IF;**

**END;**

**/**

****

1. **Create a BEFORE UPDATE Trigger for the “Event” Table that Ensures that the event's time is not set to before 7:00 AM (assuming you use 24-hour format for your Time column).**

**CREATE OR REPLACE TRIGGER before\_update\_event\_time\_check**

**BEFORE UPDATE ON Event\_1128**

**FOR EACH ROW**

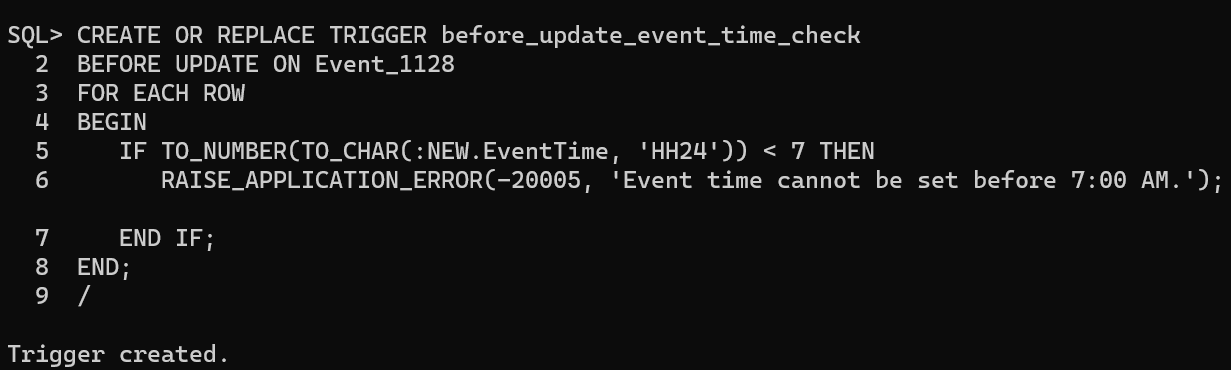
**BEGIN**

**IF TO\_NUMBER(TO\_CHAR(:NEW.EventTime, 'HH24')) < 7 THEN**

**RAISE\_APPLICATION\_ERROR(-20005, 'Event time cannot be set before 7:00 AM.');**

**END IF;**

**END;**

**/** ****

1. **Create an AFTER DELETE Trigger for the “Event” Table that signals when an event is deleted.**

**CREATE OR REPLACE TRIGGER after\_delete\_event\_signal**

**AFTER DELETE ON Event\_1128**

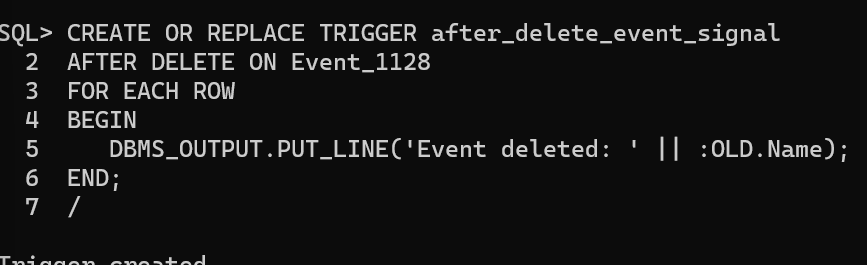
**FOR EACH ROW**

**BEGIN**

**DBMS\_OUTPUT.PUT\_LINE('Event deleted: ' || :OLD.Name);**

**END;**

**/**

****

1. **Create an AFTER UPDATE Trigger for the “Event” Table that signals when an event's time is changed.**

**CREATE OR REPLACE TRIGGER after\_update\_event\_time\_signal**

**AFTER UPDATE OF EventTime ON Event\_1128**

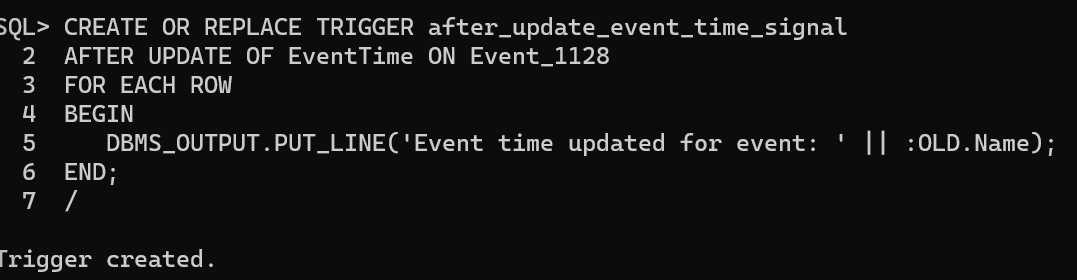
**FOR EACH ROW**

**BEGIN**

**DBMS\_OUTPUT.PUT\_LINE('Event time updated for event: ' || :OLD.Name);**

**END;**

**/**

****

1. **Create a BEFORE INSERT Trigger for the “Venue” Table that ensures the name of the venue is not empty.**

**CREATE OR REPLACE TRIGGER before\_insert\_venue\_name\_check**

**BEFORE INSERT ON Venue\_1128**

**FOR EACH ROW**

**BEGIN**

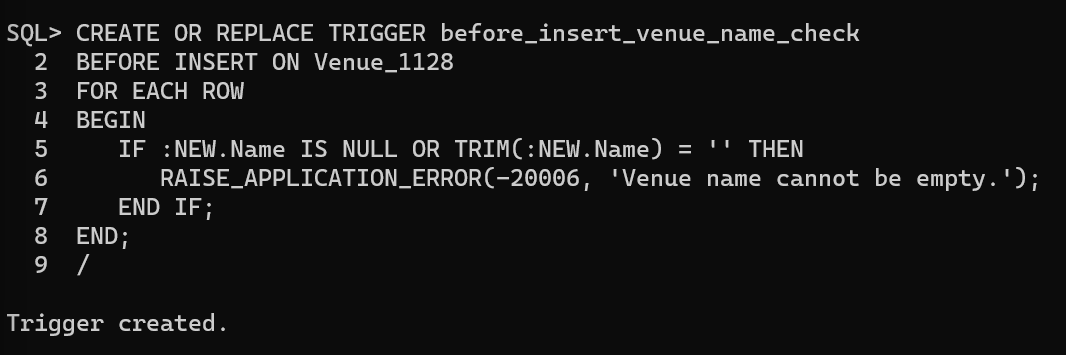
**IF :NEW.Name IS NULL OR TRIM(:NEW.Name) = '' THEN**

**RAISE\_APPLICATION\_ERROR(-20006, 'Venue name cannot be empty.');**

**END IF;**

**END;**

**/**

****

1. **Create a BEFORE DELETE Trigger for the “Venue” Table that Prevents deletion if the VenueID is less than 105.**

**CREATE OR REPLACE TRIGGER before\_delete\_venue\_check**

**BEFORE DELETE ON Venue\_1128**

**FOR EACH ROW**

**BEGIN**

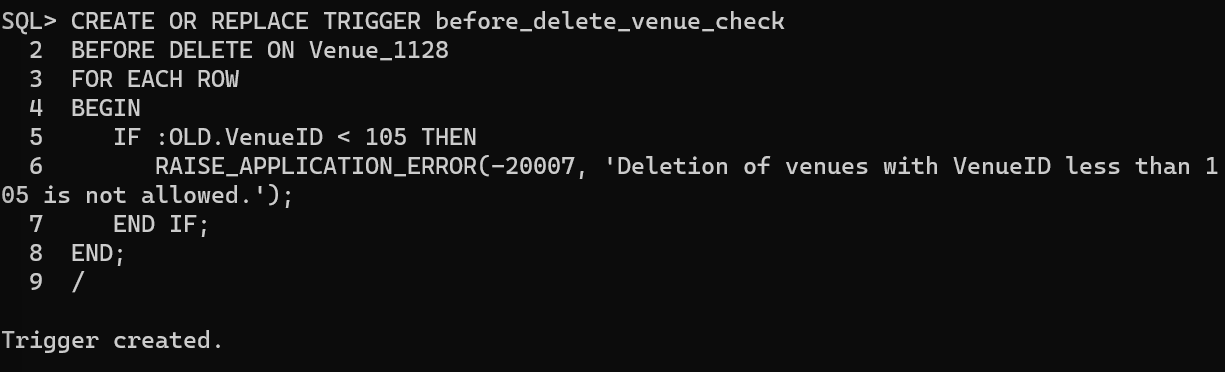
**IF :OLD.VenueID < 105 THEN**

**RAISE\_APPLICATION\_ERROR(-20007, 'Deletion of venues with VenueID less than 105 is not allowed.');**

**END IF;**

**END;**

**/**

****

1. **Create an AFTER INSERT Trigger for the “Venue” Table that signals when a new row is added to it.**

**CREATE OR REPLACE TRIGGER after\_insert\_venue\_signal**

**AFTER INSERT ON Venue\_1128**

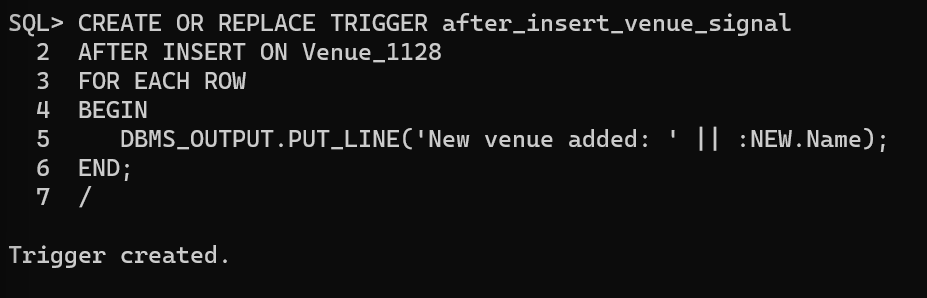
**FOR EACH ROW**

**BEGIN**

**DBMS\_OUTPUT.PUT\_LINE('New venue added: ' || :NEW.Name);**

**END;**

**/**

****

1. **Create an AFTER UPDATE Trigger for the “Venue” Table that signals when a row is updated.**

**CREATE OR REPLACE TRIGGER after\_update\_venue\_signal**

**AFTER UPDATE ON Venue\_1128**

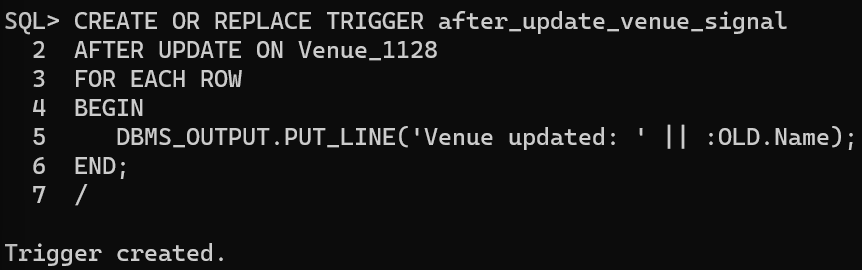
**FOR EACH ROW**

**BEGIN**

**DBMS\_OUTPUT.PUT\_LINE('Venue updated: ' || :OLD.Name);**

**END;**

**/**

****

**Result:**

The given trigger were created successfully.