Ex. No: 07	PL/SQL – TRIGGER OPERATIONS
Date	12-03-2024

Objective:
To execute the given queries using Trigger Operations.

Description
TRIGGER:
Trigger operations are actions or events that initiate a specific process or set of actions within a system or program. These triggers can be based on various conditions such as time, user input, data changes, or external events. When a trigger is activated, it prompts the system to perform predefined operations or tasks automatically. This automation helps streamline workflows, improve efficiency, and ensure timely responses to critical events. Common examples of trigger operations include sending automated emails based on user actions, updating database records when certain conditions are met, or executing specific tasks in response to external API calls.
A trigger in SQL is a concise piece of code that runs automatically when certain events occur on a table. These events can be INSERT, UPDATE, or DELETE operations.
Triggers act as guardians, ensuring data integrity by responding to events and enforcing rules.
SYNTAX:
CREATE TRIGGER Trigger_Name
[BEFORE | AFTER] [INSERT | UPDATE | DELETE] ON Table_Name
[FOR EACH ROW | FOR EACH COLUMN]
[trigger_body];

TYPES OF TRIGGERS:

BEFORE Triggers:
These triggers fire before the specified event (e.g., INSERT, UPDATE, or DELETE) takes place. Useful for validation or modification before data changes.

AFTER Triggers:
These triggers fire after the event completes successfully.Often used for logging, auditing, or additional actions.

Row-Level Triggers (FOR EACH ROW):
These triggers operate on each affected row individually.Commonly used for enforcing referential integrity or maintaining history.

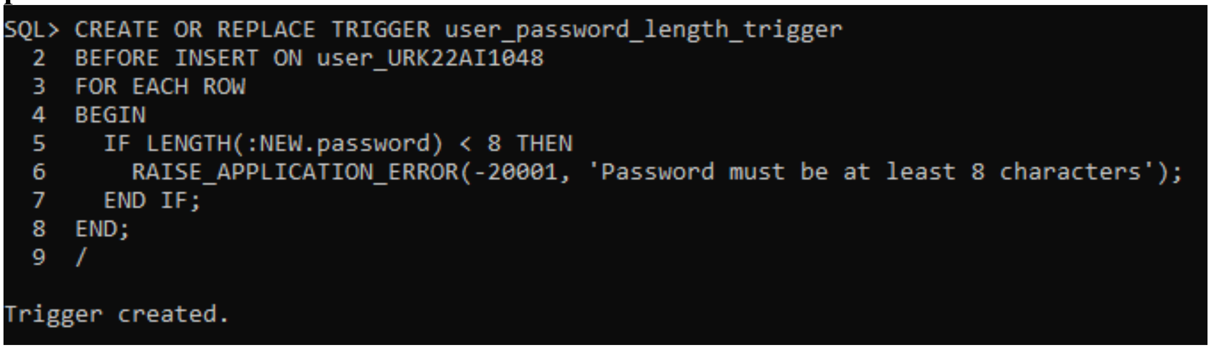
Statement-Level Triggers (FOR EACH STATEMENT):
These triggers operate on the entire set of affected rows.Useful for bulk operations or aggregations.

Nested Triggers:
Some databases allow triggers to invoke other triggers.Use with caution to avoid infinite loops.

Instead Of Triggers:
These triggers replace the default action for a specific event.Often used with views or complex data modifications.

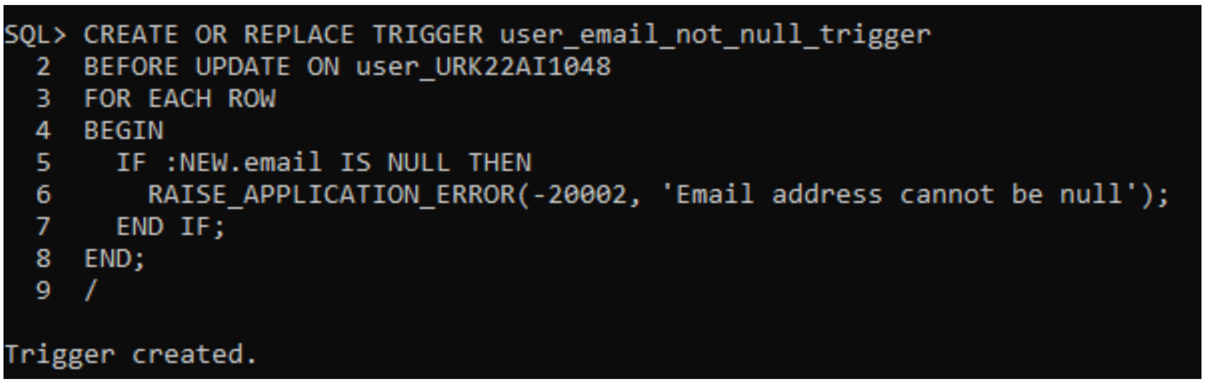
Questions

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



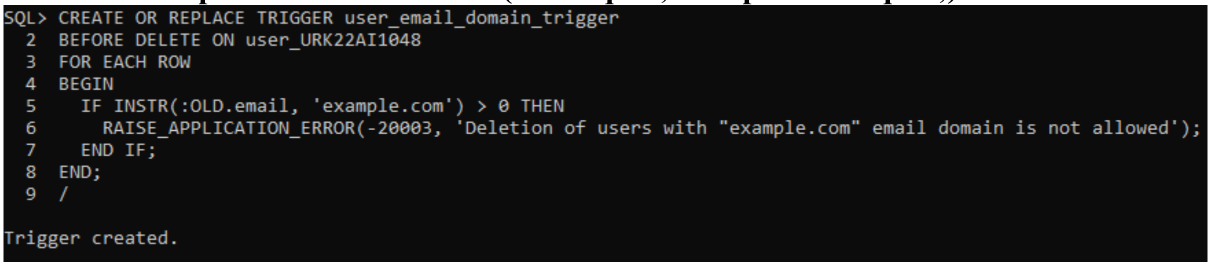
```
SQL> CREATE OR REPLACE TRIGGER user_password_length_trigger
BEFORE INSERT ON user_URK22AI1048
FOR EACH ROW
BEGIN
  IF LENGTH(:NEW.password) < 8 THEN
    RAISE_APPLICATION_ERROR(-20001, 'Password must be at least 8 characters');
  END IF;
END;
/

Trigger created.
```

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



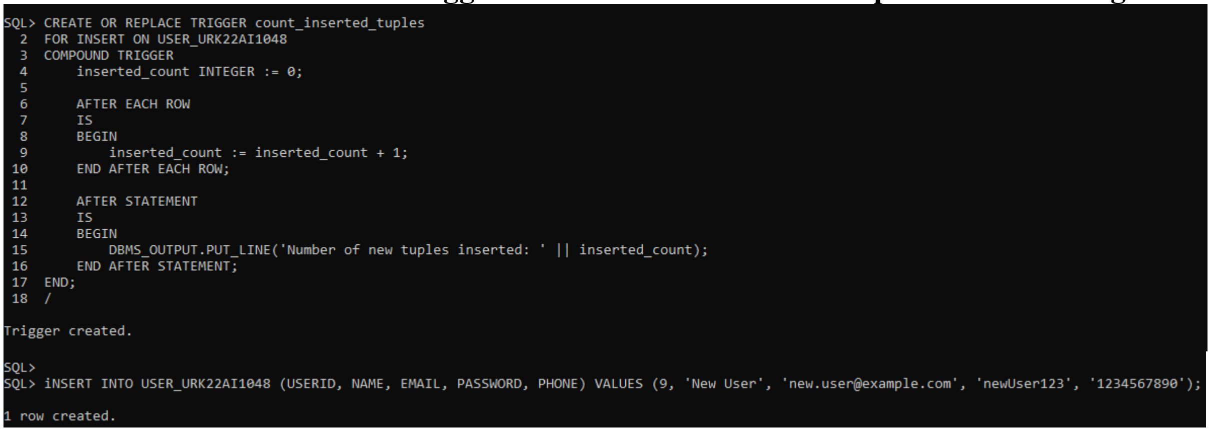
```
SQL> CREATE OR REPLACE TRIGGER user_email_not_null_trigger
BEFORE UPDATE ON user_URK22AI1048
FOR EACH ROW
BEGIN
  IF :NEW.email IS NULL THEN
    RAISE_APPLICATION_ERROR(-20002, 'Email address cannot be null');
  END IF;
END;
/

Trigger created.
```

3. Create a BEFORE DELETE Trigger for the “User” Table that prevents the deletion of users with specific email domains (like &quot;example.com&quot;).

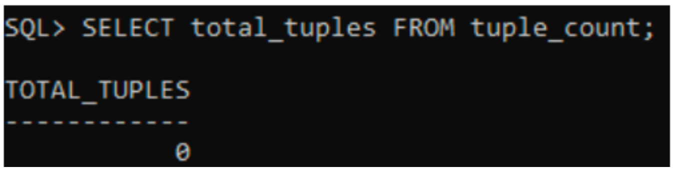


```
SQL> CREATE OR REPLACE TRIGGER user_email_domain_trigger
BEFORE DELETE ON user_URK22AI1048
FOR EACH ROW
BEGIN
  IF INSTR(:OLD.email, 'example.com') > 0 THEN
    RAISE_APPLICATION_ERROR(-20003, 'Deletion of users with "example.com" email domain is not allowed');
  END IF;
END;
/

Trigger created.
```

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

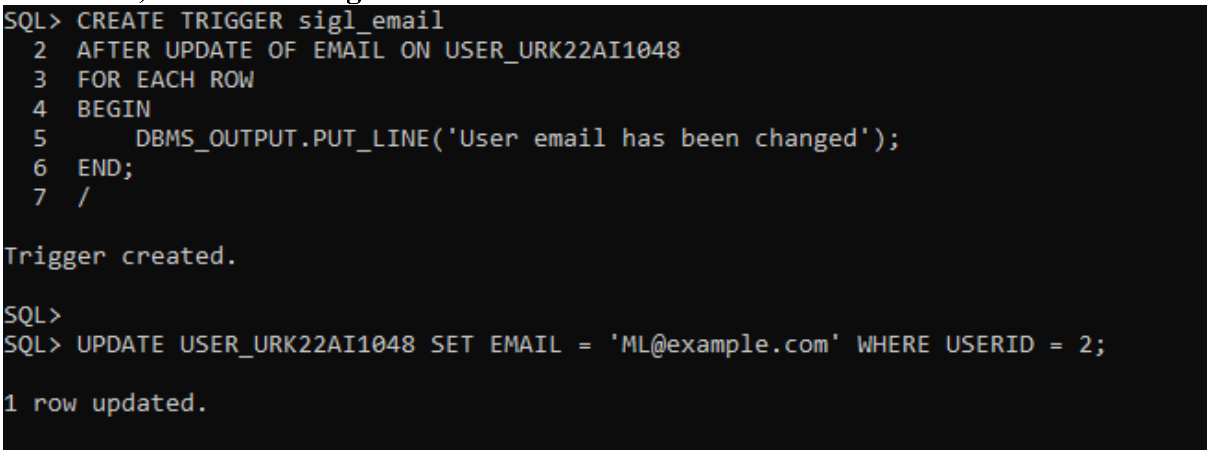


```
SQL> CREATE OR REPLACE TRIGGER count_inserted_tuples
FOR INSERT ON USER_URK22AI1048
COMPOUND TRIGGER
    inserted_count INTEGER := 0;

    AFTER EACH ROW
    IS
    BEGIN
        inserted_count := inserted_count + 1;
    END AFTER EACH ROW;

    AFTER STATEMENT
    IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Number of new tuples inserted: ' || inserted_count);
    END AFTER STATEMENT;
END;
/

Trigger created.

SQL>
SQL> iNSERT INTO USER_URK22AI1048 (USERID, NAME, EMAIL, PASSWORD, PHONE) VALUES (9, 'New User', 'new.user@example.com', 'newUser123', '1234567890');

1 row created.
```



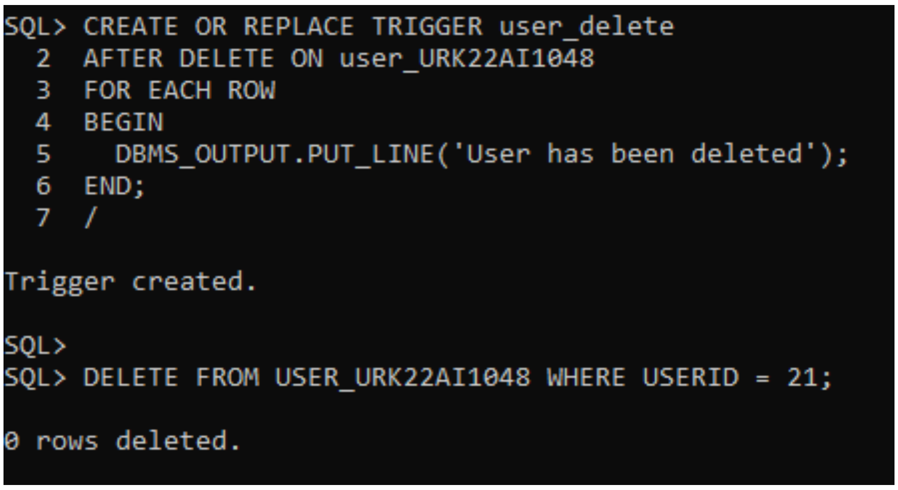
```
SQL> SELECT total_tuples FROM tuple_count;

TOTAL_TUPLES
------------
           0
```

5. Create an AFTER UPDATE Trigger for the “User” Table that signals when a user&#39;s email is changed.



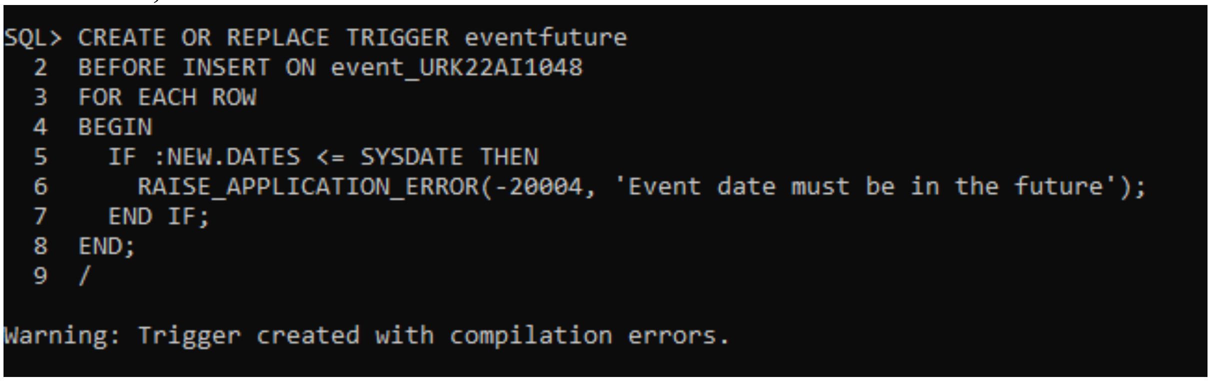
```
SQL> CREATE TRIGGER sigl_email
     AFTER UPDATE OF EMAIL ON USER_URK22AI1048
     FOR EACH ROW
     BEGIN
         DBMS_OUTPUT.PUT_LINE('User email has been changed');
     END;
     /

Trigger created.

SQL>
SQL> UPDATE USER_URK22AI1048 SET EMAIL = 'ML@example.com' WHERE USERID = 2;

1 row updated.
```

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



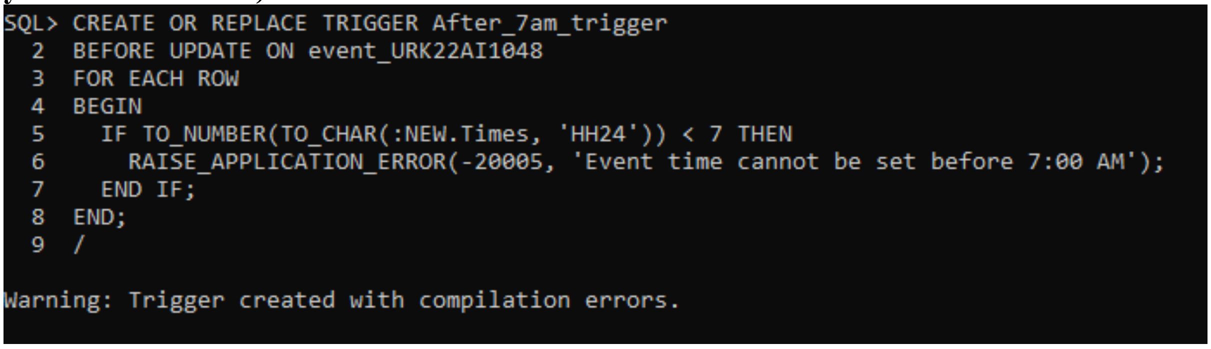
```
SQL> CREATE OR REPLACE TRIGGER user_delete
     AFTER DELETE ON user_URK22AI1048
     FOR EACH ROW
     BEGIN
       DBMS_OUTPUT.PUT_LINE('User has been deleted');
     END;
     /

Trigger created.

SQL>
SQL> DELETE FROM USER_URK22AI1048 WHERE USERID = 21;

0 rows deleted.
```

7. Create a BEFORE INSERT Trigger for the “Event” Table that ensures the event&#39;s date is in the future.



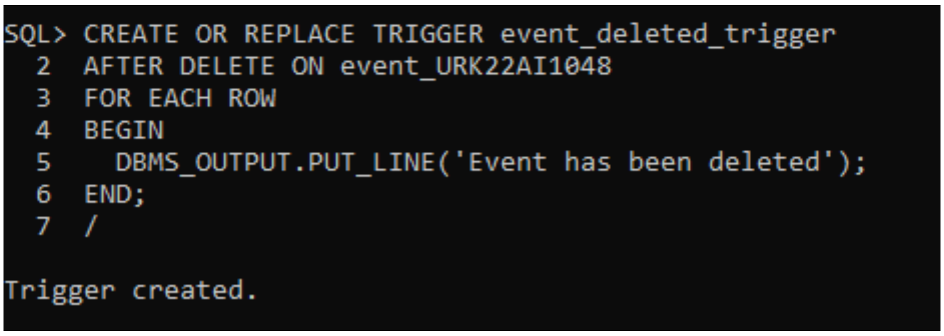
```
SQL> CREATE OR REPLACE TRIGGER eventfuture
     BEFORE INSERT ON event_URK22AI1048
     FOR EACH ROW
     BEGIN
       IF :NEW.DATES <= SYSDATE THEN
         RAISE_APPLICATION_ERROR(-20004, 'Event date must be in the future');
       END IF;
     END;
     /

Warning: Trigger created with compilation errors.
```

8. Create a BEFORE UPDATE Trigger for the “Event” Table that Ensures that the event&#39;s time is not set to before 7:00 AM (assuming you use 24-hour format for your Time column).



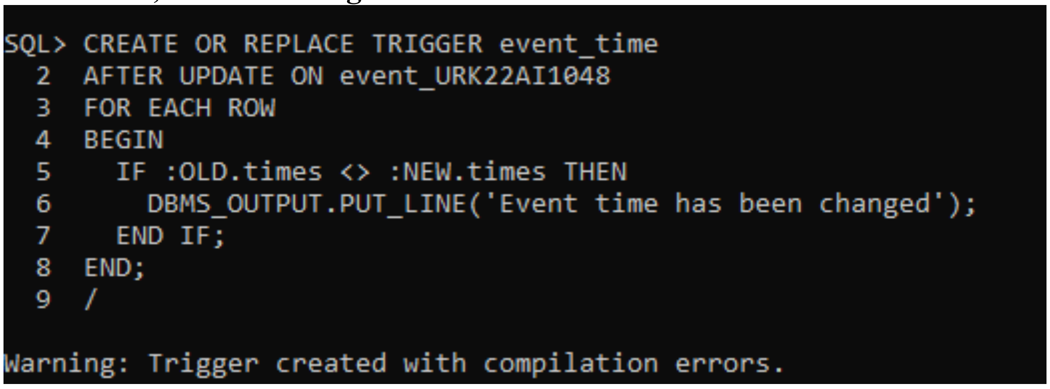
```
SQL> CREATE OR REPLACE TRIGGER After_7am_trigger
  2  BEFORE UPDATE ON event_URK22AI1048
  3  FOR EACH ROW
  4  BEGIN
  5    IF TO_NUMBER(TO_CHAR(:NEW.Times, 'HH24')) < 7 THEN
  6      RAISE_APPLICATION_ERROR(-20005, 'Event time cannot be set before 7:00 AM');
  7    END IF;
  8  END;
  9  /

Warning: Trigger created with compilation errors.
```

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



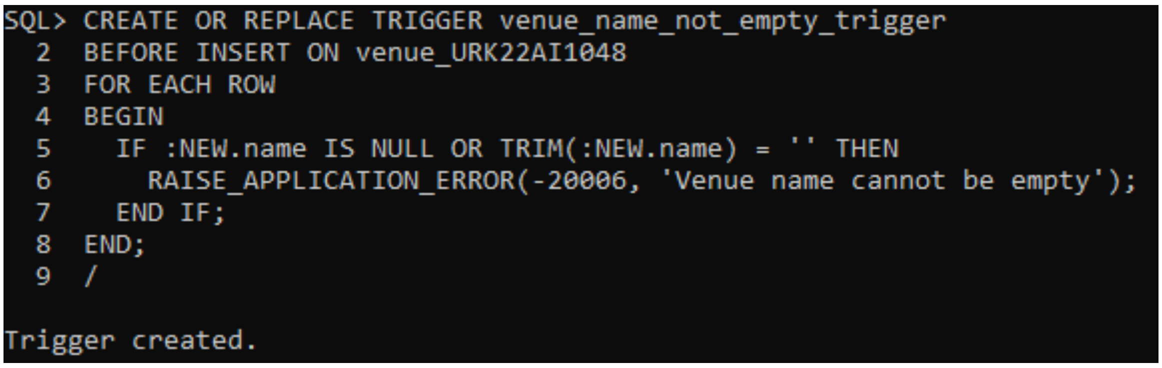
```
SQL> CREATE OR REPLACE TRIGGER event_deleted_trigger
  2  AFTER DELETE ON event_URK22AI1048
  3  FOR EACH ROW
  4  BEGIN
  5    DBMS_OUTPUT.PUT_LINE('Event has been deleted');
  6  END;
  7  /

Trigger created.
```

10. Create an AFTER UPDATE Trigger for the “Event” Table that signals when an event&#39;s time is changed.



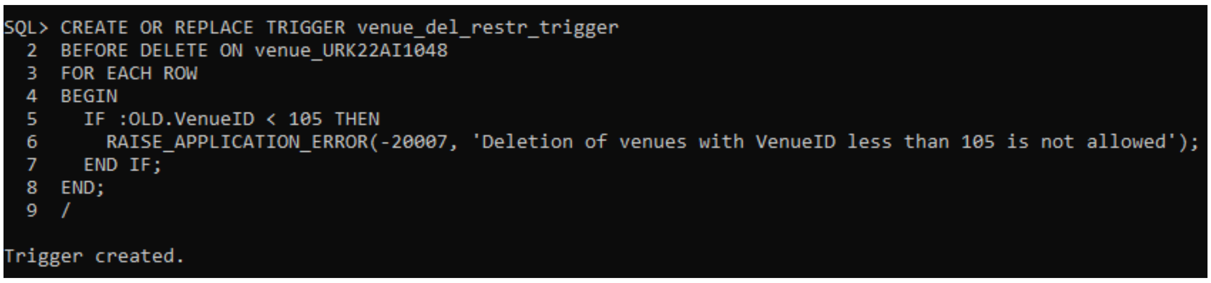
```
SQL> CREATE OR REPLACE TRIGGER event_time
  2  AFTER UPDATE ON event_URK22AI1048
  3  FOR EACH ROW
  4  BEGIN
  5    IF :OLD.times <> :NEW.times THEN
  6      DBMS_OUTPUT.PUT_LINE('Event time has been changed');
  7    END IF;
  8  END;
  9  /

Warning: Trigger created with compilation errors.
```

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



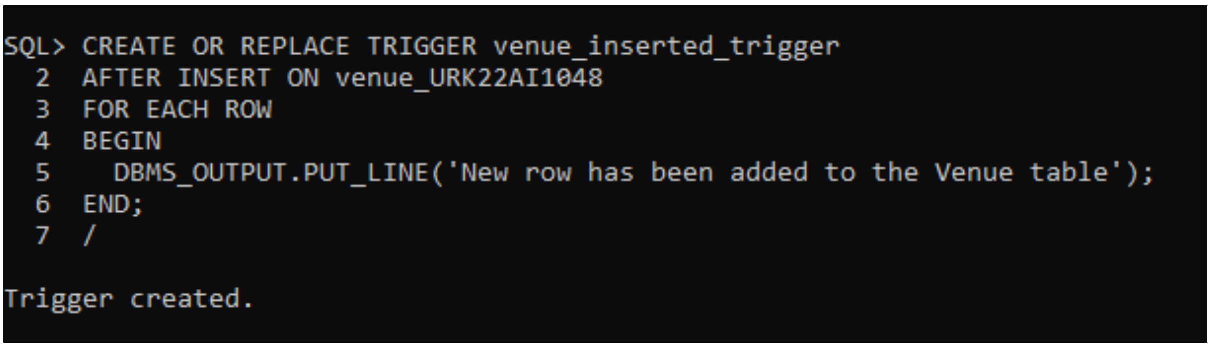
```
SQL> CREATE OR REPLACE TRIGGER venue_name_not_empty_trigger
  BEFORE INSERT ON venue_URK22AI1048
  FOR EACH ROW
  BEGIN
    IF :NEW.name IS NULL OR TRIM(:NEW.name) = '' THEN
      RAISE_APPLICATION_ERROR(-20006, 'Venue name cannot be empty');
    END IF;
  END;
  /

Trigger created.
```

12. Create a BEFORE DELETE Trigger for the "Venue" Table that Prevents deletion if the Venueid is less than 105.



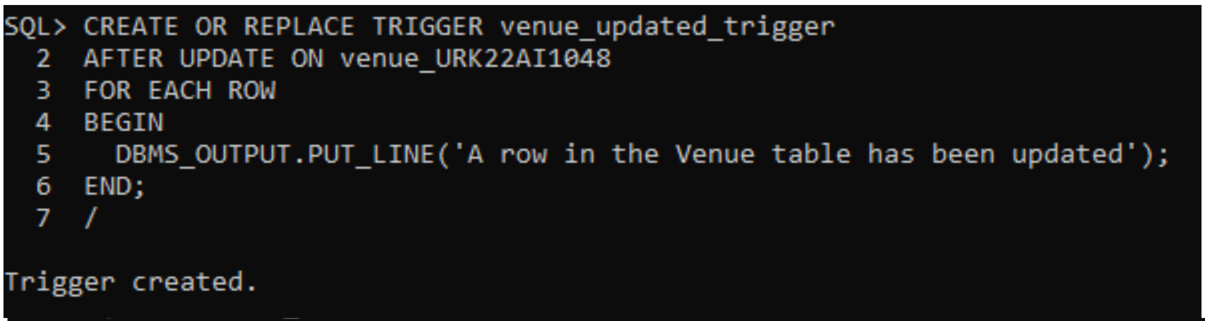
```
SQL> CREATE OR REPLACE TRIGGER venue_del_restr_trigger
  BEFORE DELETE ON venue_URK22AI1048
  FOR EACH ROW
  BEGIN
    IF :OLD.VenueID < 105 THEN
      RAISE_APPLICATION_ERROR(-20007, 'Deletion of venues with VenueID less than 105 is not allowed');
    END IF;
  END;
  /

Trigger created.
```

13. Create an AFTER INSERT Trigger for the "Venue" Table that signals when a new row is added to it.

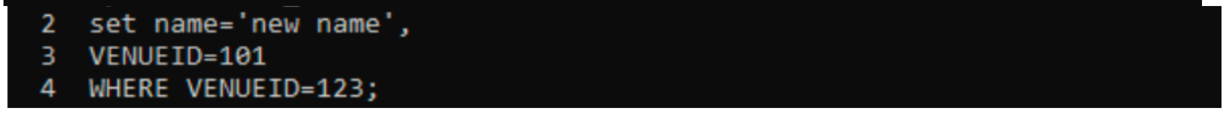


```
SQL> CREATE OR REPLACE TRIGGER venue_inserted_trigger
  AFTER INSERT ON venue_URK22AI1048
  FOR EACH ROW
  BEGIN
    DBMS_OUTPUT.PUT_LINE('New row has been added to the Venue table');
  END;
  /

Trigger created.
```

14. Create an AFTER UPDATE



```
SQL> CREATE OR REPLACE TRIGGER venue_updated_trigger
  AFTER UPDATE ON venue_URK22AI1048
  FOR EACH ROW
  BEGIN
    DBMS_OUTPUT.PUT_LINE('A row in the Venue table has been updated');
  END;
  /

Trigger created.
```



```
  set name='new name',
  VENUEID=101
  WHERE VENUEID=123;
```

Result:

The given queries executed by the set operations and joins successfully.