CodeLine

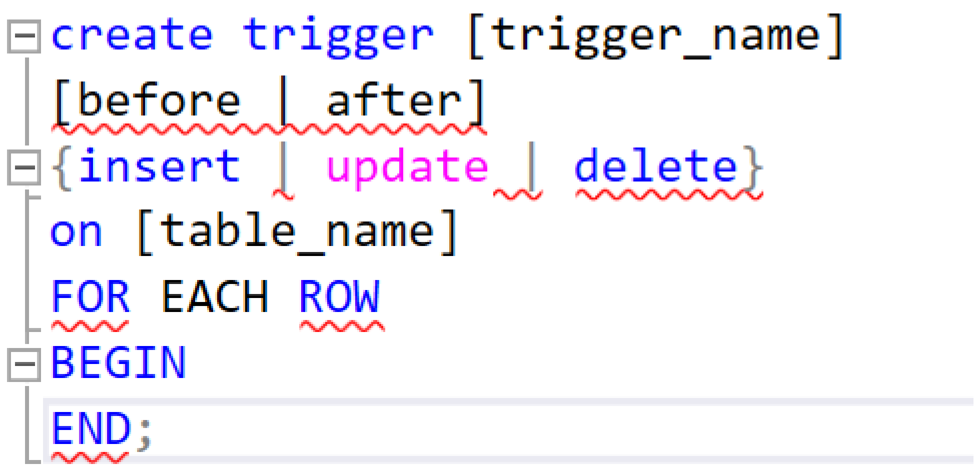
SQL TRIGGERS

What are SQL Triggers?

A trigger is a stored procedure in a database that automatically invokes whenever a special event in the database occurs. By using SQL triggers, developers can automate tasks, ensure data consistency, and keep accurate records of database activities. For example, a trigger can be invoked when a row is inserted into a specified table or when specific table columns are updated.

In simple words, a trigger is a collection of SQL statements with particular names that are stored in system memory. It belongs to a specific class of stored procedures that are automatically invoked in response to database server events. Every trigger has a table attached to it.

Syntax



```
create trigger [trigger_name]
[before | after]
{insert | update | delete}
on [table_name]
FOR EACH ROW
BEGIN
END;
```

Explanation:

- **trigger_name**: The name of the trigger to be created.
- **BEFORE | AFTER**: Specifies whether the trigger is fired before or after the triggering event (INSERT, UPDATE, DELETE).
- **{INSERT | UPDATE | DELETE}**: Specifies the operation that will activate the trigger.
- **table_name**: The name of the table the trigger is associated with.
- **FOR EACH ROW**: Indicates that the trigger is row-level, meaning it executes once for each affected row.
- **trigger_body**: The SQL statements to be executed when the trigger is fired.

Why Should You Use SQL Triggers?

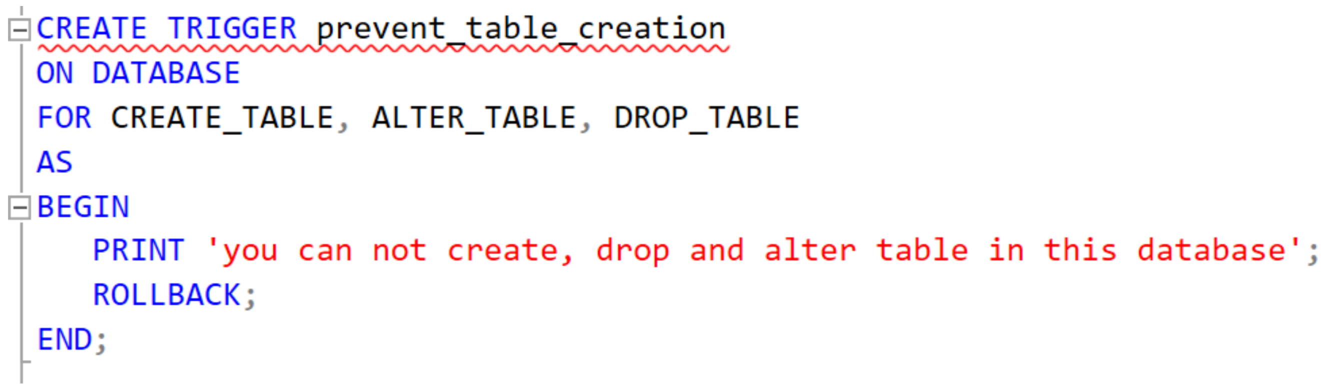
- **Automation**: Triggers handle repetitive tasks, saving our time and effort.
- **Consistency & Data Integrity**: Automatically enforcing rules ensures that our data remains clean and accurate.
- **Business Rules Enforcement**: Triggers can help ensure that changes to our database follow your business logic.
- **Audit Trails**: Track changes automatically, making it easier to monitor and record data updates.

Types of SQL Triggers

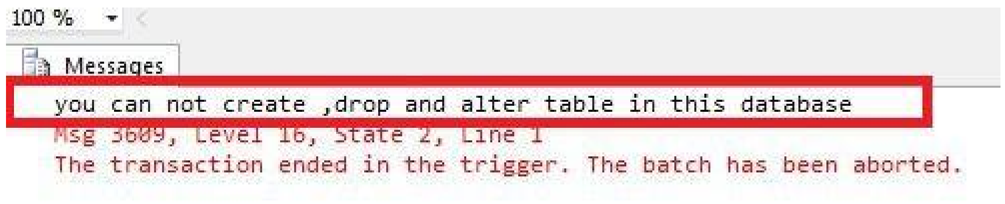
1. **DDL Triggers**

   The Data Definition Language (DDL) command events such as Create_table, Create_view, drop_table, Drop_view, and Alter_table cause the DDL triggers to be activated. They allow us to track changes in the structure of the database. The trigger will prevent any table creation, alteration, or deletion in the database.

Example: Prevent Table Deletions:



```
CREATE TRIGGER prevent_table_creation
ON DATABASE
FOR CREATE_TABLE, ALTER_TABLE, DROP_TABLE
AS
BEGIN
    PRINT 'you can not create, drop and alter table in this database';
    ROLLBACK;
END;
```

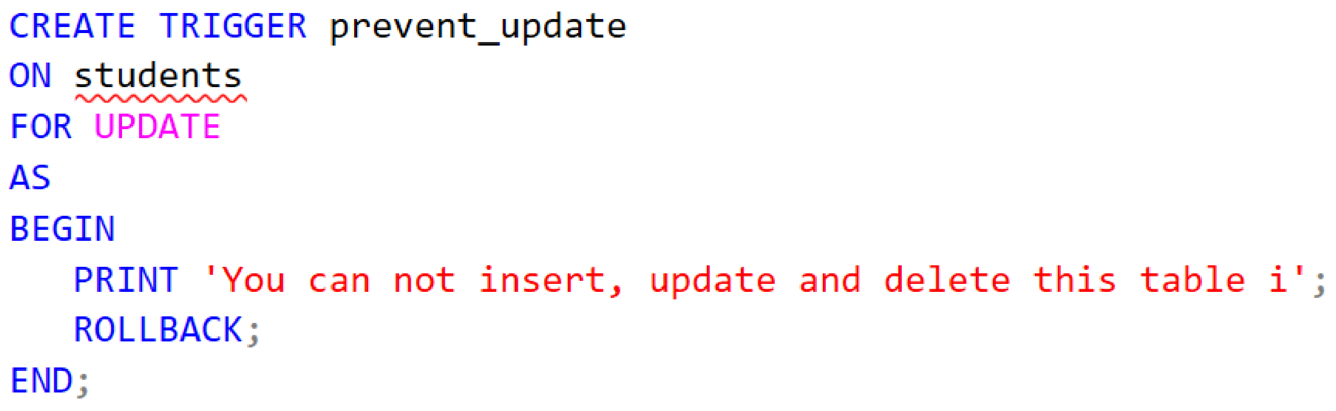


```
you can not create ,drop and alter table in this database
Msg 3609, Level 16, State 2, Line 1
The transaction ended in the trigger. The batch has been aborted.
```

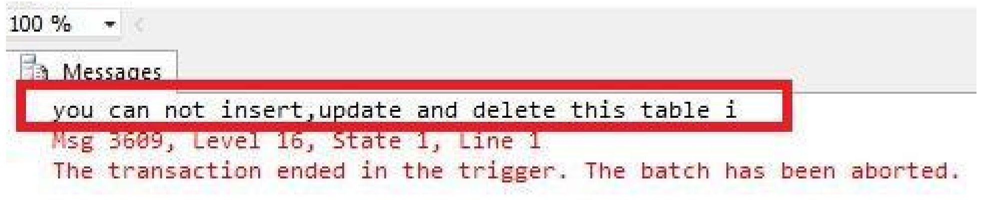
2. DML Triggers

DML triggers fire when we manipulate data with commands like INSERT, UPDATE, or DELETE. These triggers are perfect for scenarios where we need to validate data before it is inserted, log changes to a table, or cascade updates across related tables.

Example: Prevent Unauthorized Updates Let's say you want to prevent users from updating the data in a sensitive students table. We can set up a trigger to handle that:



```
CREATE TRIGGER prevent_update
ON students
FOR UPDATE
AS
BEGIN
    PRINT 'You can not insert, update and delete this table i';
    ROLLBACK;
END;
```

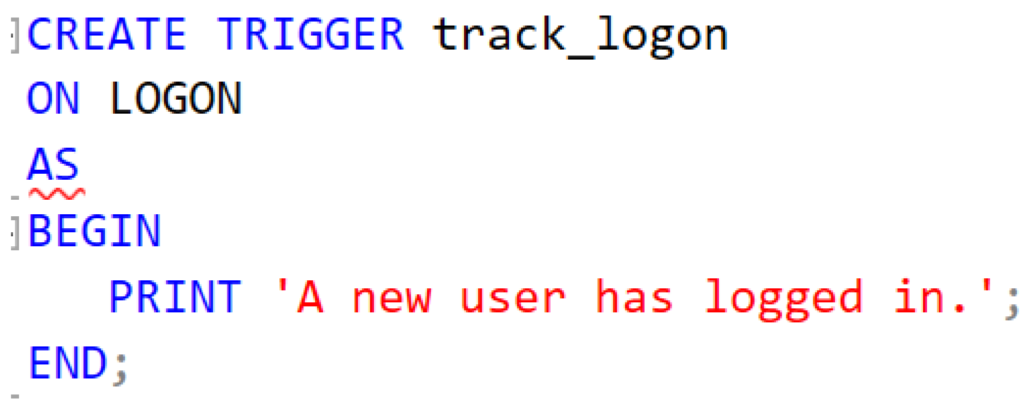


```
you can not insert,update and delete this table i
Msg 3609, Level 16, State 1, Line 1
The transaction ended in the trigger. The batch has been aborted.
```

3. Logon Triggers

These triggers are fired in response to logon events. Logon triggers are useful for monitoring user sessions or restricting user access to the database. As a result, the PRINT statement messages and any errors generated by the trigger will all be visible in the SQL Server error log.

Authentication errors prevent logon triggers from being used. These triggers can be used to track login activity or set a limit on the number of sessions that a given login can have in order to audit and manage server sessions.



```
CREATE TRIGGER track_logon
ON LOGON
AS
BEGIN
    PRINT 'A new user has logged in.';
END;
```

4. BEFORE and AFTER Triggers

SQL triggers can be specified to run **BEFORE** or **AFTER** the triggering event.

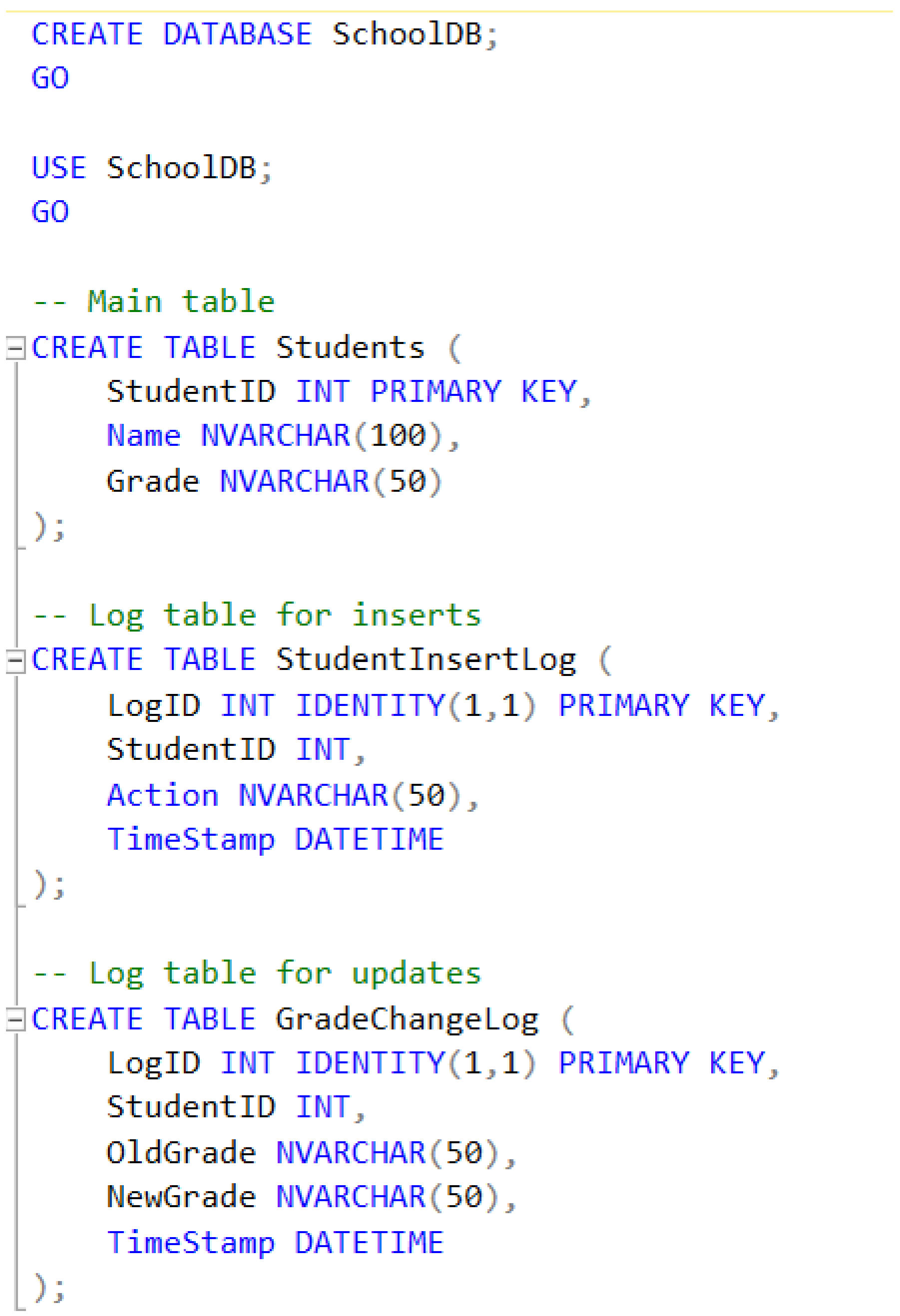
- **BEFORE Triggers**: These run before the action (INSERT, UPDATE, DELETE) is executed. They're great for **data validation** or **modifying** values before they are committed to the database. **Note**: In SQL Server, **BEFORE** triggers don't exist. Instead, we use **INSTEAD** OF triggers to achieve the same behavior.

- **AFTER Triggers**: Execute after the SQL statement completes. Useful for logging or cascading updates to other tables.

Real-World Use Cases of SQL Triggers

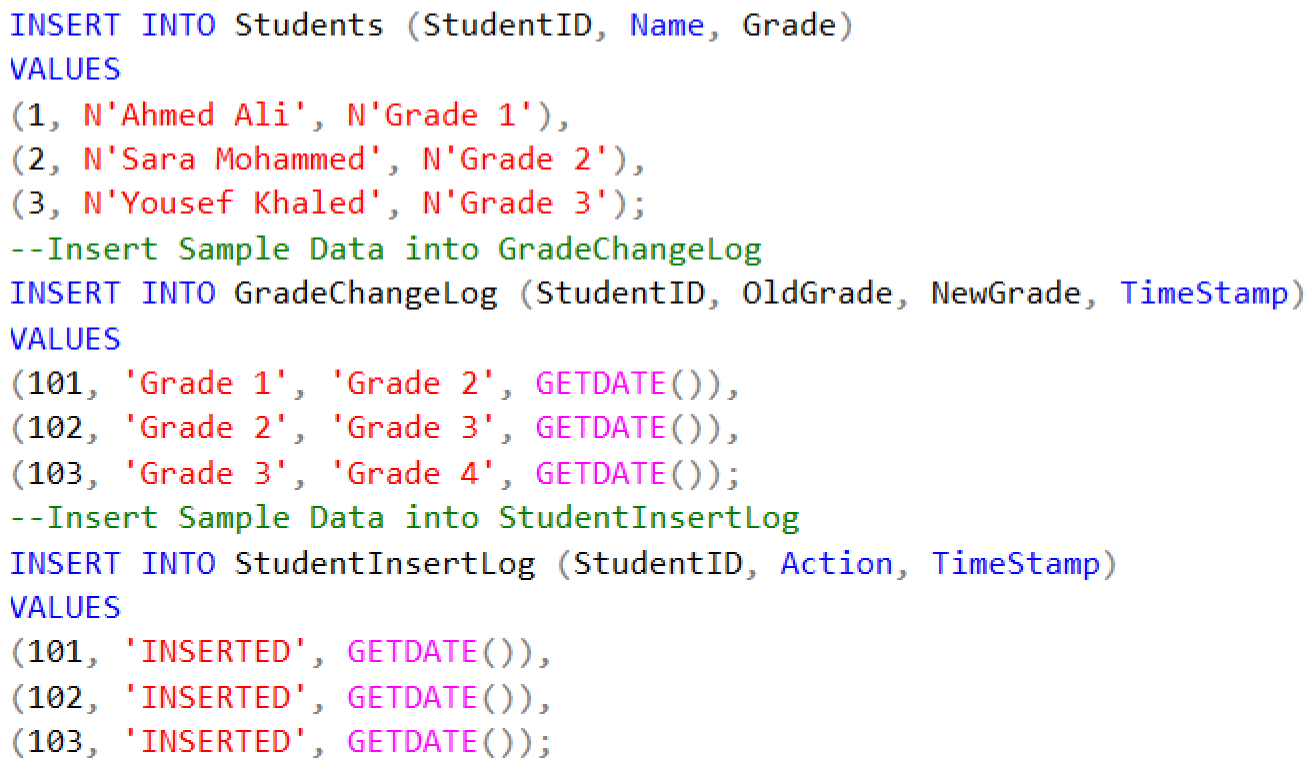
- **Student Records System**

Create DB and related tables:



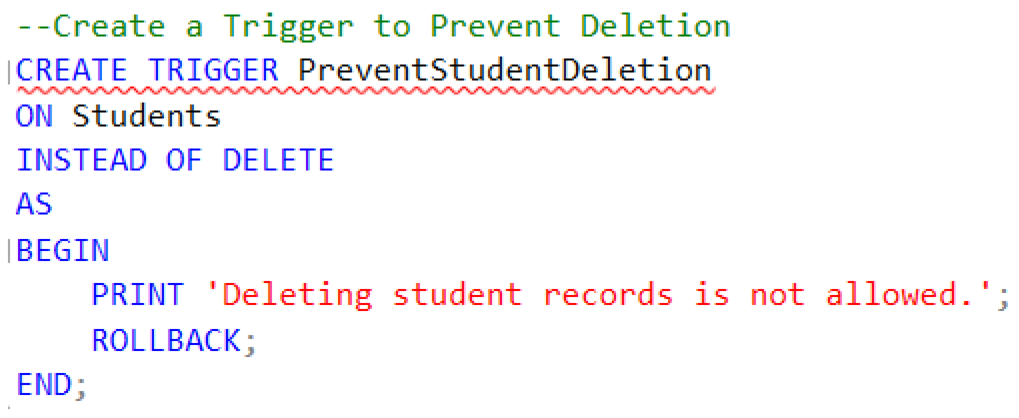
```sql
CREATE DATABASE SchoolDB;
GO

USE SchoolDB;
GO

-- Main table
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    Name NVARCHAR(100),
    Grade NVARCHAR(50)
);

-- Log table for inserts
CREATE TABLE StudentInsertLog (
    LogID INT IDENTITY(1,1) PRIMARY KEY,
    StudentID INT,
    Action NVARCHAR(50),
    TimeStamp DATETIME
);

-- Log table for updates
CREATE TABLE GradeChangeLog (
    LogID INT IDENTITY(1,1) PRIMARY KEY,
    StudentID INT,
    OldGrade NVARCHAR(50),
    NewGrade NVARCHAR(50),
    TimeStamp DATETIME
);
```

Insert Sample Data:



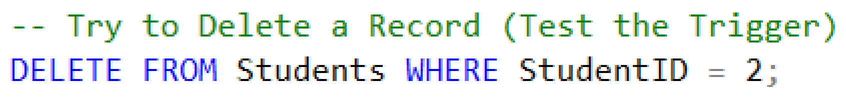
```sql
INSERT INTO Students (StudentID, Name, Grade)
VALUES
(1, N'Ahmed Ali', N'Grade 1'),
(2, N'Sara Mohammed', N'Grade 2'),
(3, N'Yousef Khaled', N'Grade 3');
--Insert Sample Data into GradeChangeLog
INSERT INTO GradeChangeLog (StudentID, OldGrade, NewGrade, TimeStamp)
VALUES
(101, 'Grade 1', 'Grade 2', GETDATE()),
(102, 'Grade 2', 'Grade 3', GETDATE()),
(103, 'Grade 3', 'Grade 4', GETDATE());
--Insert Sample Data into StudentInsertLog
INSERT INTO StudentInsertLog (StudentID, Action, TimeStamp)
VALUES
(101, 'INSERTED', GETDATE()),
(102, 'INSERTED', GETDATE()),
(103, 'INSERTED', GETDATE());
```

Create a Trigger to Prevent Deletion:



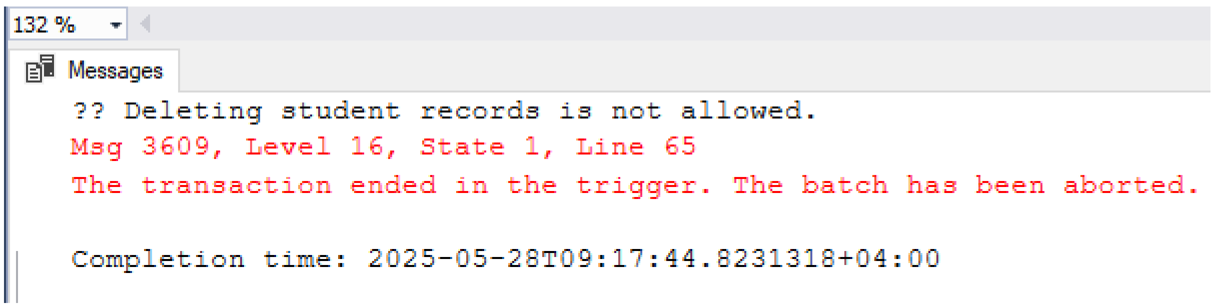
```sql
--Create a Trigger to Prevent Deletion
CREATE TRIGGER PreventStudentDeletion
ON Students
INSTEAD OF DELETE
AS
BEGIN
    PRINT 'Deleting student records is not allowed.';
    ROLLBACK;
END;
```

Try to Delete a Record:



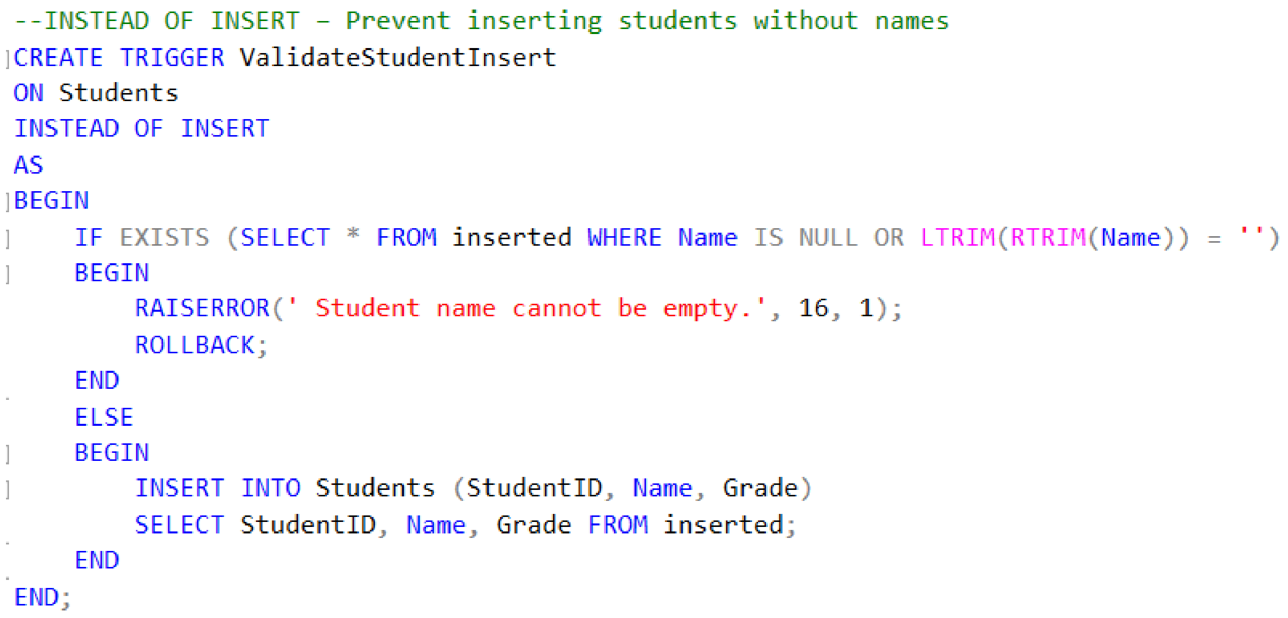
```sql
-- Try to Delete a Record (Test the Trigger)
DELETE FROM Students WHERE StudentID = 2;
```

Expected Output:



```
?? Deleting student records is not allowed.
Msg 3609, Level 16, State 1, Line 65
The transaction ended in the trigger. The batch has been aborted.

Completion time: 2025-05-28T09:17:44.8231318+04:00
```

BEFORE INSERT (via `INSTEAD OF`): Prevent inserting if name is empty:

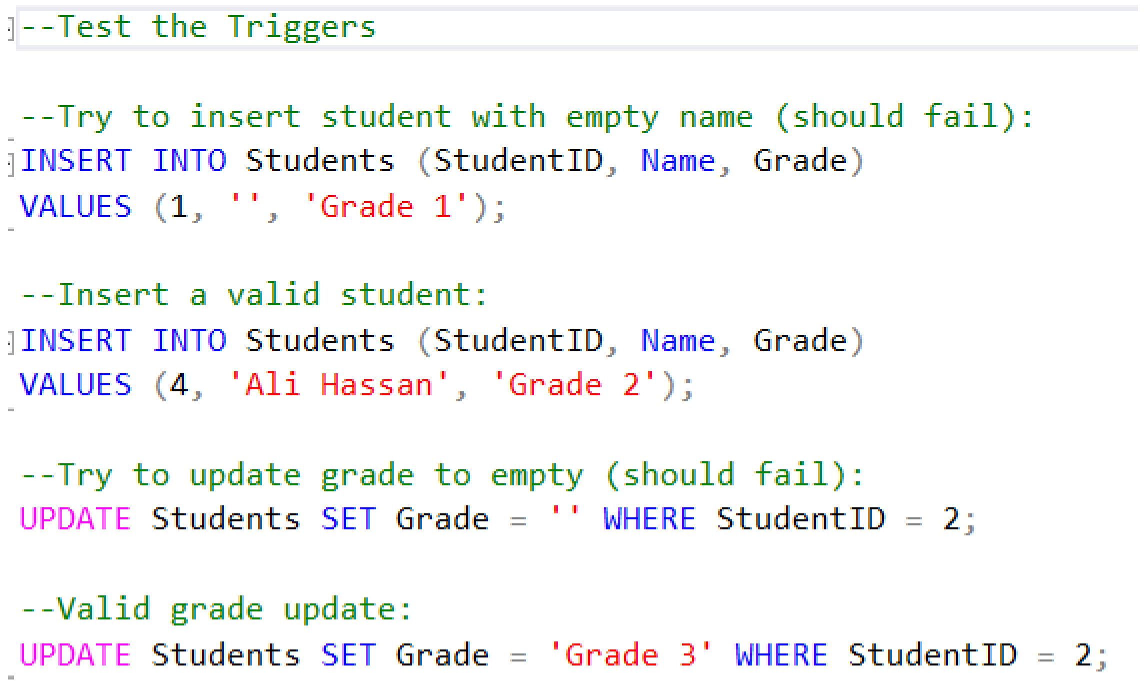


```
--INSTEAD OF INSERT - Prevent inserting students without names
CREATE TRIGGER ValidateStudentInsert
ON Students
INSTEAD OF INSERT
AS
BEGIN
    IF EXISTS (SELECT * FROM inserted WHERE Name IS NULL OR LTRIM(RTRIM(Name)) = '')
    BEGIN
        RAISERROR(' Student name cannot be empty.', 16, 1);
        ROLLBACK;
    END
    ELSE
    BEGIN
        INSERT INTO Students (StudentID, Name, Grade)
        SELECT StudentID, Name, Grade FROM inserted;
    END
END;
```

AFTER INSERT: Log inserted students:

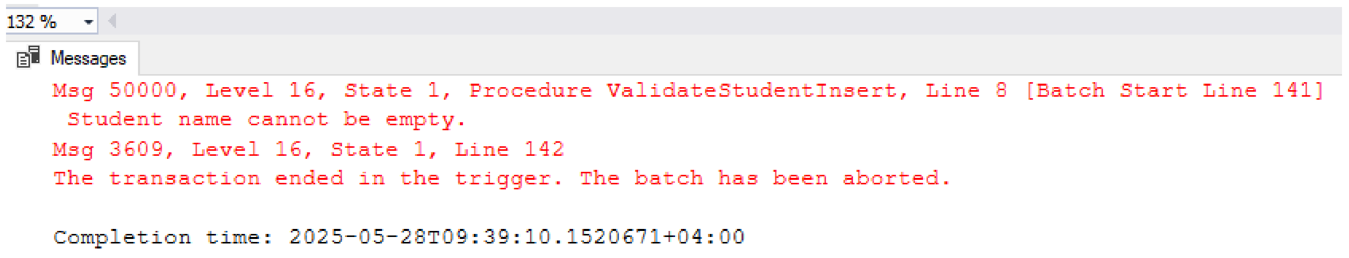
```
--AFTER INSERT - Log every new student
CREATE TRIGGER LogStudentInsert
ON Students
AFTER INSERT
AS
BEGIN
    INSERT INTO StudentInsertLog (StudentID, Action, TimeStamp)
    SELECT StudentID, 'INSERTED', GETDATE() FROM inserted;
END;
```

```
--INSTEAD OF UPDATE – Prevent setting Grade to invalid value
CREATE TRIGGER ValidateGradeUpdate
ON Students
INSTEAD OF UPDATE
AS
BEGIN
    IF EXISTS (SELECT * FROM inserted WHERE Grade IS NULL OR Grade = '')
    BEGIN
        RAISERROR(' Grade cannot be empty.', 16, 1);
        ROLLBACK;
    END
    ELSE
    BEGIN
        UPDATE Students
        SET Name = i.Name,
            Grade = i.Grade
        FROM Students s
        JOIN inserted i ON s.StudentID = i.StudentID;
    END
END;


--AFTER UPDATE – Log grade changes
CREATE TRIGGER LogGradeChange
ON Students
AFTER UPDATE
AS
BEGIN
    INSERT INTO GradeChangeLog (StudentID, OldGrade, NewGrade, TimeStamp)
    SELECT d.StudentID, d.Grade, i.Grade, GETDATE()
    FROM deleted d
    JOIN inserted i ON d.StudentID = i.StudentID
    WHERE d.Grade <> i.Grade;
END;
```

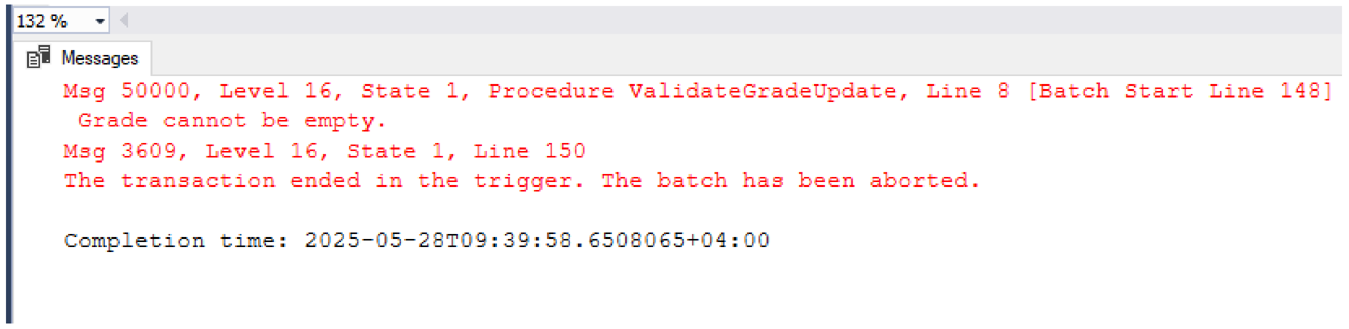


```sql
--Test the Triggers

--Try to insert student with empty name (should fail):
INSERT INTO Students (StudentID, Name, Grade)
VALUES (1, '', 'Grade 1');

--Insert a valid student:
INSERT INTO Students (StudentID, Name, Grade)
VALUES (4, 'Ali Hassan', 'Grade 2');

--Try to update grade to empty (should fail):
UPDATE Students SET Grade = '' WHERE StudentID = 2;

--Valid grade update:
UPDATE Students SET Grade = 'Grade 3' WHERE StudentID = 2;
```

Expected Result:



```
Msg 50000, Level 16, State 1, Procedure ValidateStudentInsert, Line 8 [Batch Start Line 141]
 Student name cannot be empty.
Msg 3609, Level 16, State 1, Line 142
The transaction ended in the trigger. The batch has been aborted.

Completion time: 2025-05-28T09:39:10.1520671+04:00
```



```
Msg 50000, Level 16, State 1, Procedure ValidateGradeUpdate, Line 8 [Batch Start Line 148]
 Grade cannot be empty.
Msg 3609, Level 16, State 1, Line 150
The transaction ended in the trigger. The batch has been aborted.

Completion time: 2025-05-28T09:39:58.6508065+04:00
```

Advantages of Triggers

- **Data Integrity**: Triggers help enforce consistency and business rules, ensuring that data follows the correct format.
- **Automation**: Triggers eliminate the need for manual intervention by automatically performing tasks such as updating, inserting, or deleting records when certain conditions are met.
- **Audit Trail**: Triggers can track changes in a database, providing an audit trail of INSERT, UPDATE, and DELETE operations.
- **Performance**: By automating repetitive tasks, triggers improve SQL query performance and reduce manual workload.

Conclusion

SQL triggers are a powerful feature for automating and enforcing rules in our database management system. Whether we're ensuring data integrity, automating updates, or preventing unauthorized changes, triggers can save time and improve consistency in database operations. With various types of triggers available, including DML triggers, DDL triggers, and logon triggers, they can be tailored to a wide range of use cases. Understanding and implementing SQL triggers effectively is important for maintaining a strong and efficient database.