Triggers are database objects that are associated with a table and are executed automatically in response to certain events on the table. There are two main types of triggers: `BEFORE` triggers and `AFTER` triggers. `BEFORE` triggers are executed before the event (e.g., `INSERT`, `UPDATE`, `DELETE`) occurs, while `AFTER` triggers are executed after the event.

Here are two code examples for creating triggers for each trigger type (BEFORE and AFTER) along with examples for each trigger event (INSERT, UPDATE, DELETE). I'll use SQL syntax for the examples:

Insert– used ton inset the data into the table
,update– used to update the data into the table
,delete – used to delete the data into the table

Example 1: BEFORE Trigger for INSERT

```sql
-- Creating a BEFORE INSERT trigger
CREATE TRIGGER before_insert_trigger
BEFORE INSERT ON your_table
FOR EACH ROW
BEGIN
  -- Your trigger logic here before the INSERT operation
  -- For example, you can modify the data before it is inserted
  SET NEW.column_name = UPPER(NEW.column_name);
END;
```

In this example, the `before_insert_trigger` is set to execute before any `INSERT` operation on `your_table`. It converts the value of `column_name` to uppercase before the actual insertion.

Example 2: AFTER Trigger for UPDATE

```sql
-- Creating an AFTER UPDATE trigger
CREATE TRIGGER after_update_trigger
AFTER UPDATE ON your_table
FOR EACH ROW
BEGIN
  -- Your trigger logic here after the UPDATE operation
  -- For example, you can log the changes in another table
  INSERT INTO audit_table (changed_column, new_value, old_value)
  VALUES ('column_name', NEW.column_name, OLD.column_name);
END;
```

This trigger, `after_update_trigger`, is executed after any `UPDATE` operation on `your_table`. It logs the changes made to the `column_name` in an `audit_table`.

Example 3: BEFORE Trigger for DELETE

```sql
-- Creating a BEFORE DELETE trigger
CREATE TRIGGER before_delete_trigger
BEFORE DELETE ON your_table
FOR EACH ROW
BEGIN
  -- Your trigger logic here before the DELETE operation
  -- For example, you can prevent deletion based on certain conditions
  IF OLD.column_name = 'important_value' THEN
    SIGNAL SQLSTATE '45000'
    SET MESSAGE_TEXT = 'Cannot delete rows with important_value.';
  END IF;
END;
```

This trigger, `before_delete_trigger`, is executed before any `DELETE` operation on `your_table`. It prevents the deletion of rows where the value of `column_name` is 'important_value'.

Example 4: AFTER Trigger for INSERT

```sql
-- Creating an AFTER INSERT trigger
CREATE TRIGGER after_insert_trigger
AFTER INSERT ON your_table
FOR EACH ROW
BEGIN
  -- Your trigger logic here after the INSERT operation
  -- For example, you can update a counter or perform additional actions
  UPDATE counter_table SET count = count + 1;
END;
```

The `after_insert_trigger` is executed after any `INSERT` operation on `your_table`. In this example, it increments a counter in a separate `counter_table` after each insertion.

These examples showcase how triggers can be used to execute custom logic before or after specific events (INSERT, UPDATE, DELETE) on database tables.

Certainly! Let's explore additional examples with triggers:

Example 5: BEFORE Trigger for UPDATE with Validation

```sql
-- Creating a BEFORE UPDATE trigger with validation
CREATE TRIGGER before_update_validation_trigger
BEFORE UPDATE ON your_table
FOR EACH ROW
BEGIN
  -- Your trigger logic here before the UPDATE operation
  -- For example, you can validate the new value of a column
  IF NEW.column_name < 0 THEN
    SIGNAL SQLSTATE '45000'
    SET MESSAGE_TEXT = 'Column value cannot be negative.';
  END IF;
END;
```

In this case, the `before_update_validation_trigger` prevents updates that would set the value of `column_name` to a negative number.

Example 6: AFTER Trigger for DELETE with Logging

```sql
-- Creating an AFTER DELETE trigger with logging
CREATE TRIGGER after_delete_logging_trigger
AFTER DELETE ON your_table
FOR EACH ROW
BEGIN
  -- Your trigger logic here after the DELETE operation
  -- For example, you can log deleted rows into a history table
  INSERT INTO deleted_history_table (column1, column2, deleted_at)
  VALUES (OLD.column1, OLD.column2, NOW());
END;
```

The `after_delete_logging_trigger` logs deleted rows into a `deleted_history_table` with additional information such as the timestamp of deletion (`deleted_at`).

Example 7: BEFORE Trigger for INSERT with Default Values

```sql
```

```
-- Creating a BEFORE INSERT trigger to set default values
CREATE TRIGGER before_insert_default_values_trigger
BEFORE INSERT ON your_table
FOR EACH ROW
BEGIN
  -- Your trigger logic here before the INSERT operation
  -- Set default values if certain columns are not provided
  IF NEW.column_name IS NULL THEN
    SET NEW.column_name = 'default_value';
  END IF;
END;
```

The `before_insert_default_values_trigger` sets a default value for `column_name` if it is not provided during the `INSERT` operation.

Example 8: AFTER Trigger for UPDATE with Notification

```sql
-- Creating an AFTER UPDATE trigger with notification
CREATE TRIGGER after_update_notification_trigger
AFTER UPDATE ON your_table
FOR EACH ROW
BEGIN
  -- Your trigger logic here after the UPDATE operation
  -- Send a notification, e.g., via email or message queue
  CALL send_notification('Table updated', CONCAT('Row ID ', NEW.id, ' has been updated.'));
END;
```

The `after_update_notification_trigger` sends a notification (using a hypothetical `send_notification` function) after any update operation on `your_table`. This could be useful for alerting users or other systems about changes.

These examples demonstrate the flexibility of triggers in enforcing business rules, logging changes, setting default values, and triggering notifications based on various database events.