

Lesson 01 Demo 06

Implementing Transaction Management

Objective: To implement transaction management to execute all the SQL statements together, ensuring data integrity and consistency within the database

Tool required: Eclipse IDE

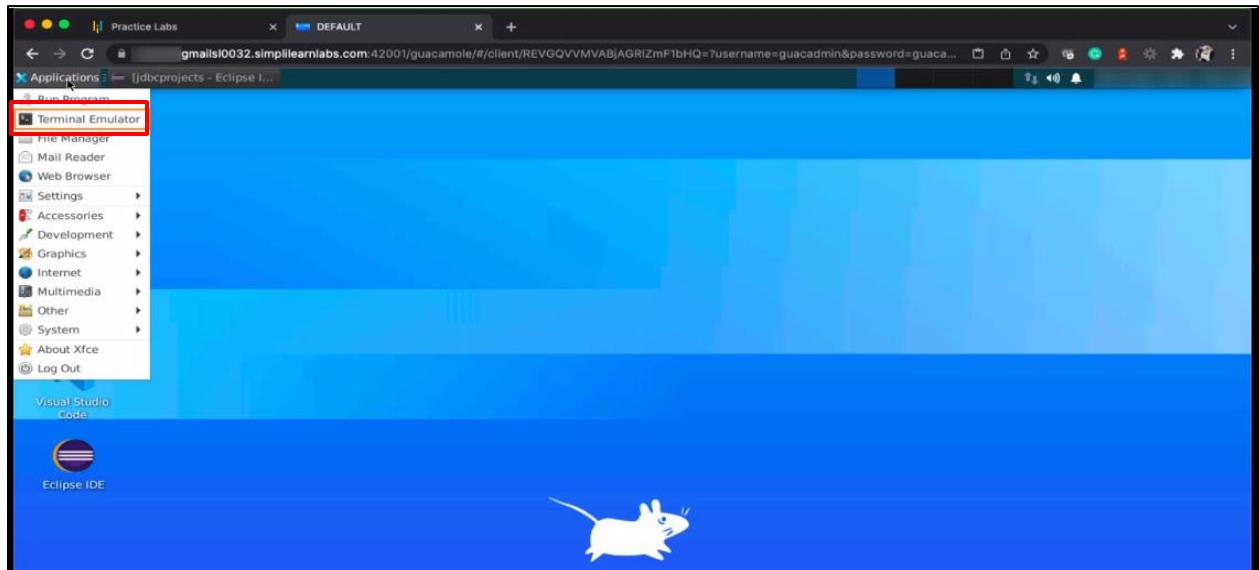
Prerequisites: None

Steps to be followed:

1. Create new table name orders
2. Create the executionTransaction method
3. Call the created method and checking the output
4. Use the auto-commit feature
5. Use insert and update operations
6. Use the delete operation

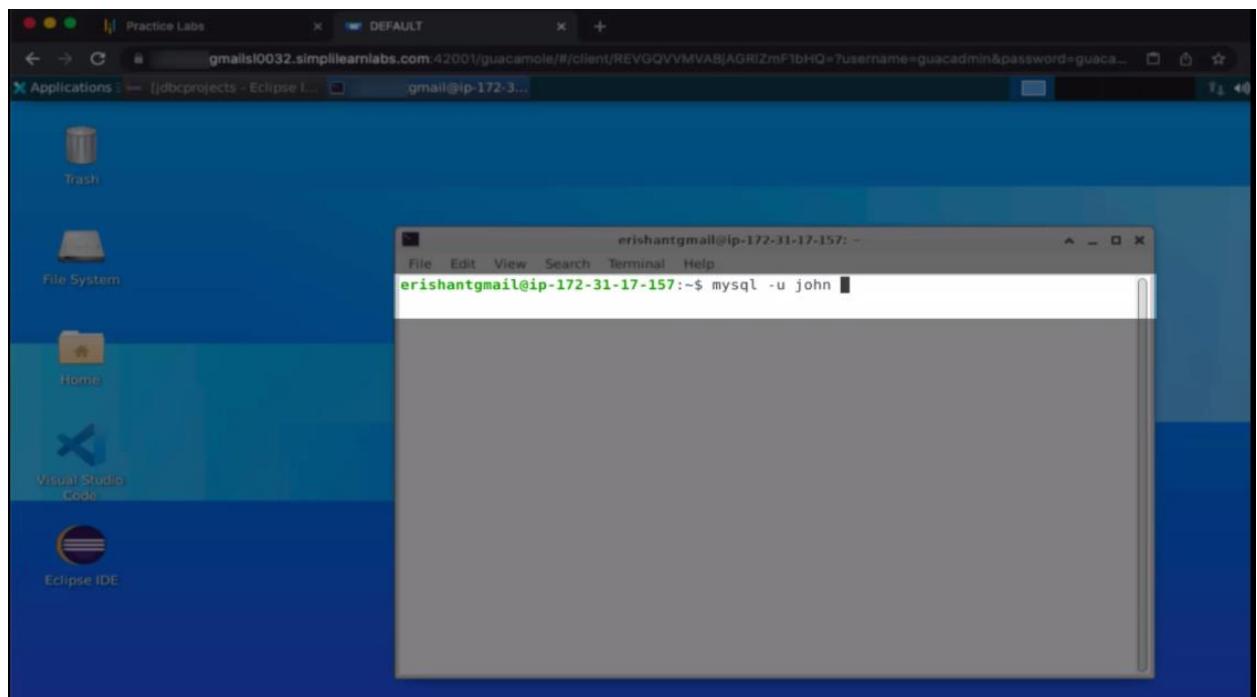
Step 1: Create new table name orders

1.1 Open the Terminal Emulator



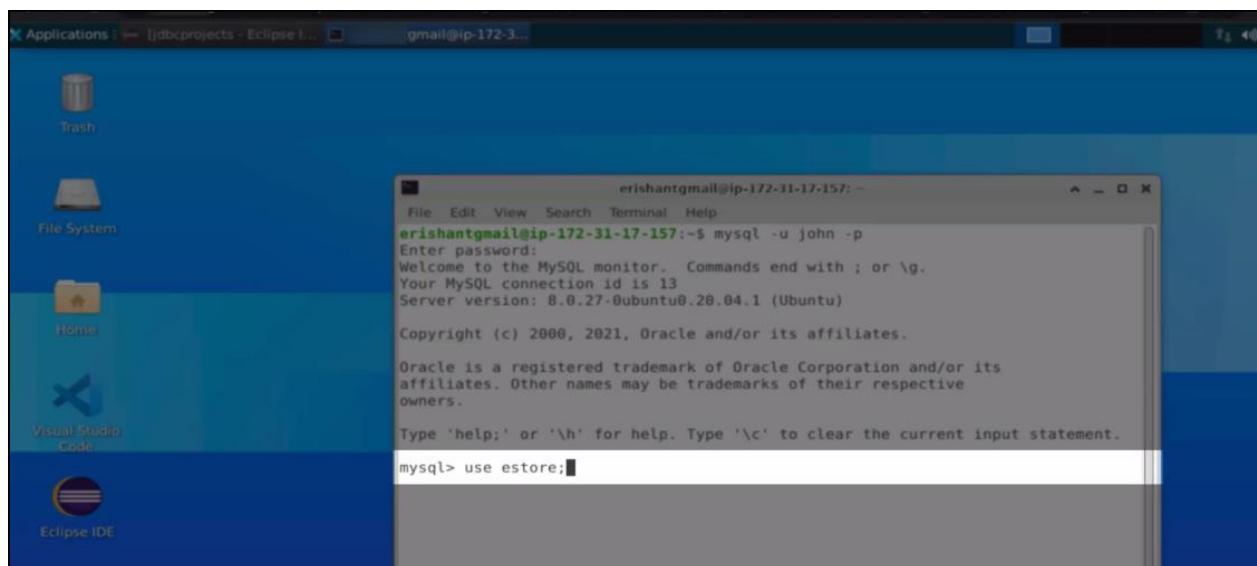
1.2 Login to MySQL using the command:

mysql -u john -p

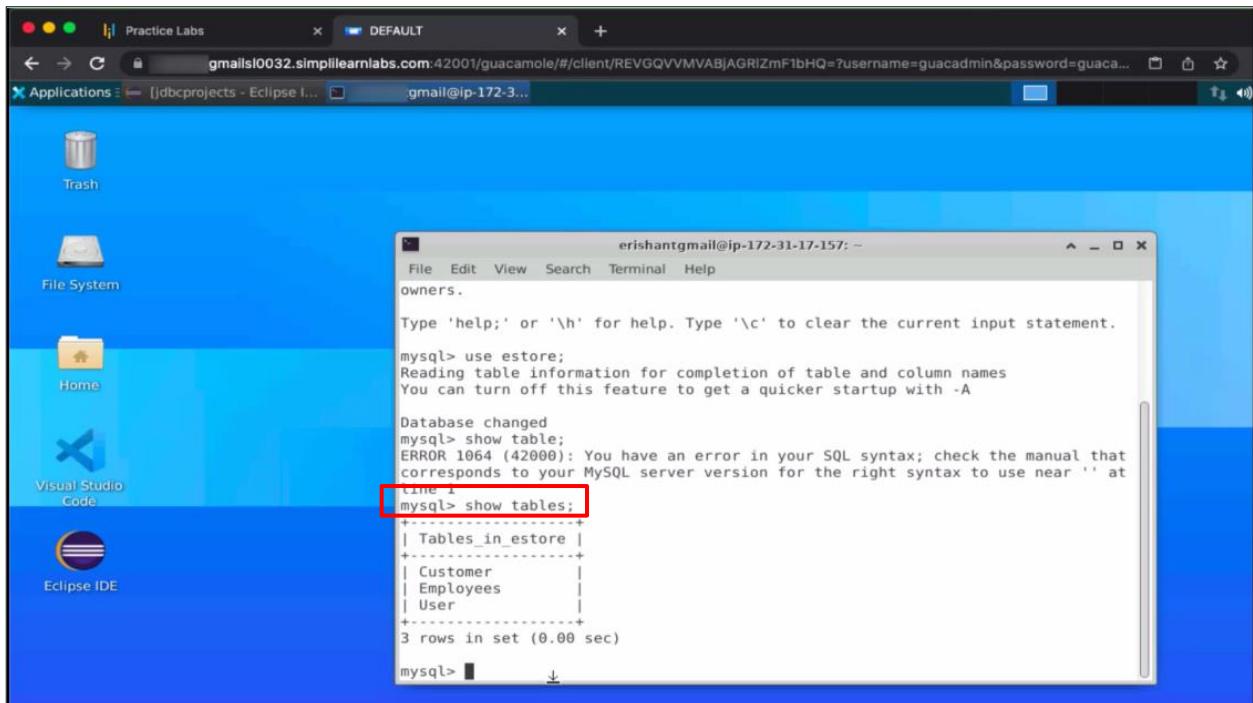


Note: A user named john has already been created for the database.

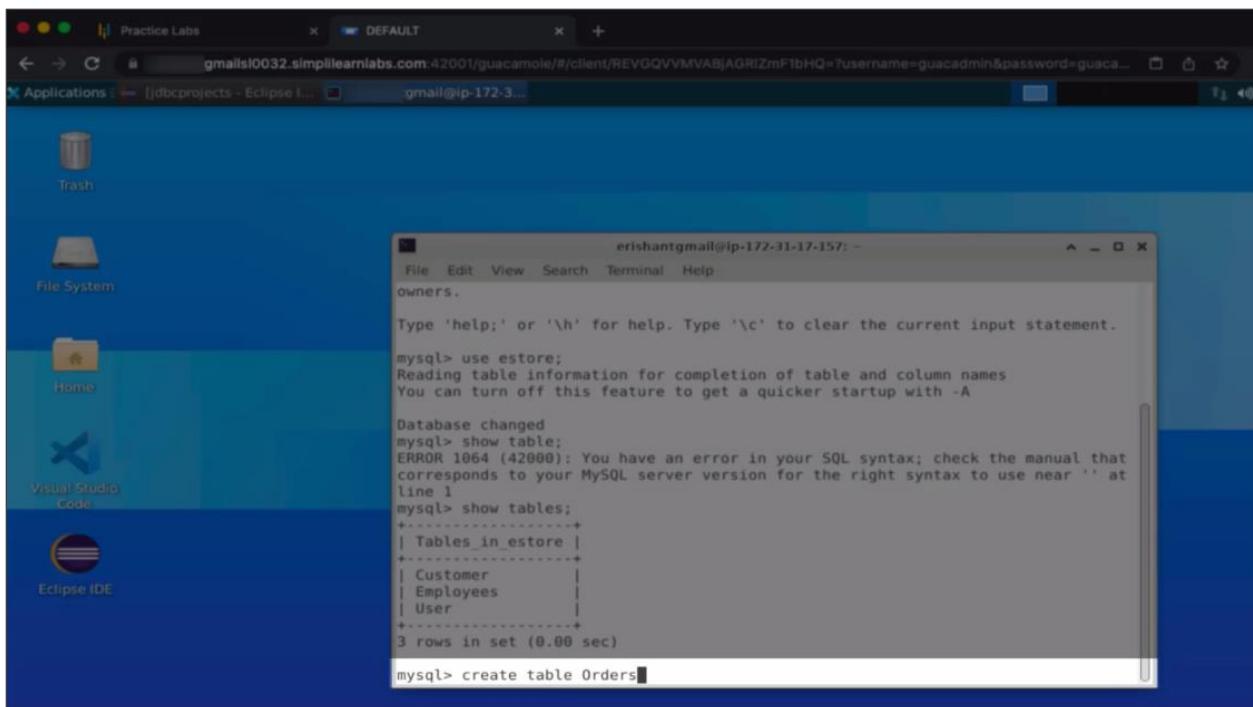
1.3 Enter the command **use estore;** to change the database



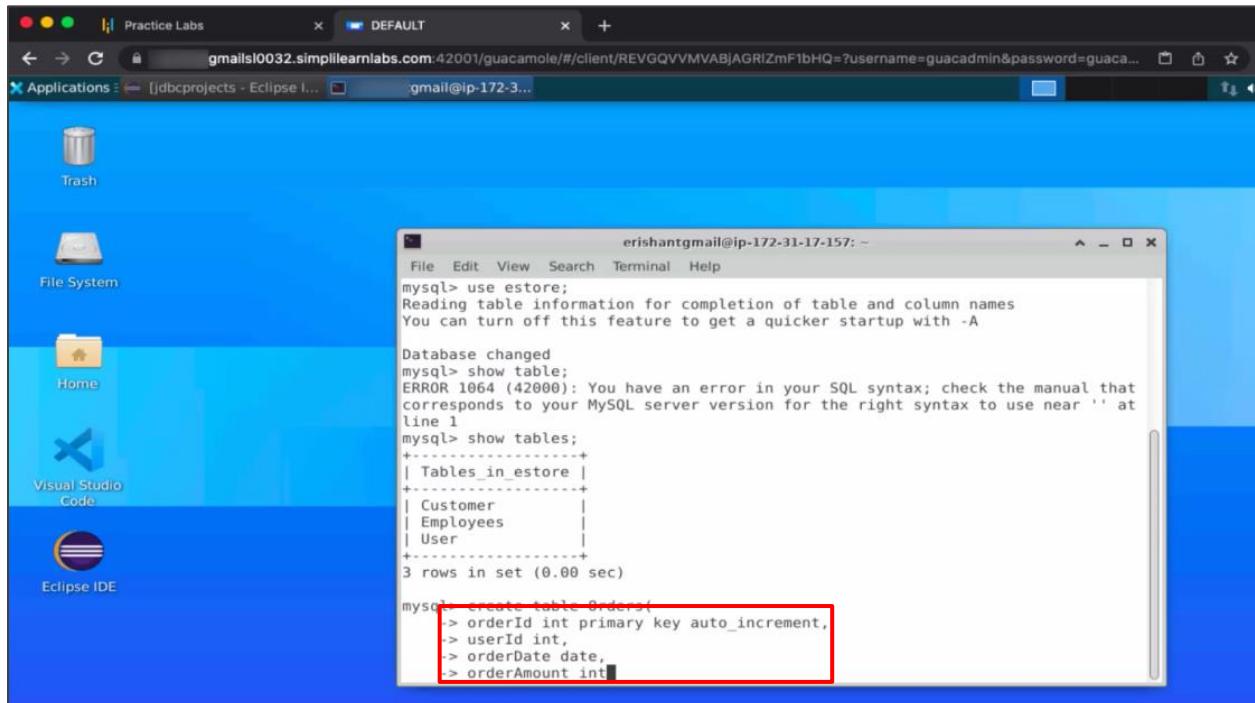
1.4 Run the **show tables;** command to list the tables in the **estore** database



1.5 Write **create table Orders** to create a new table in the **estore** database



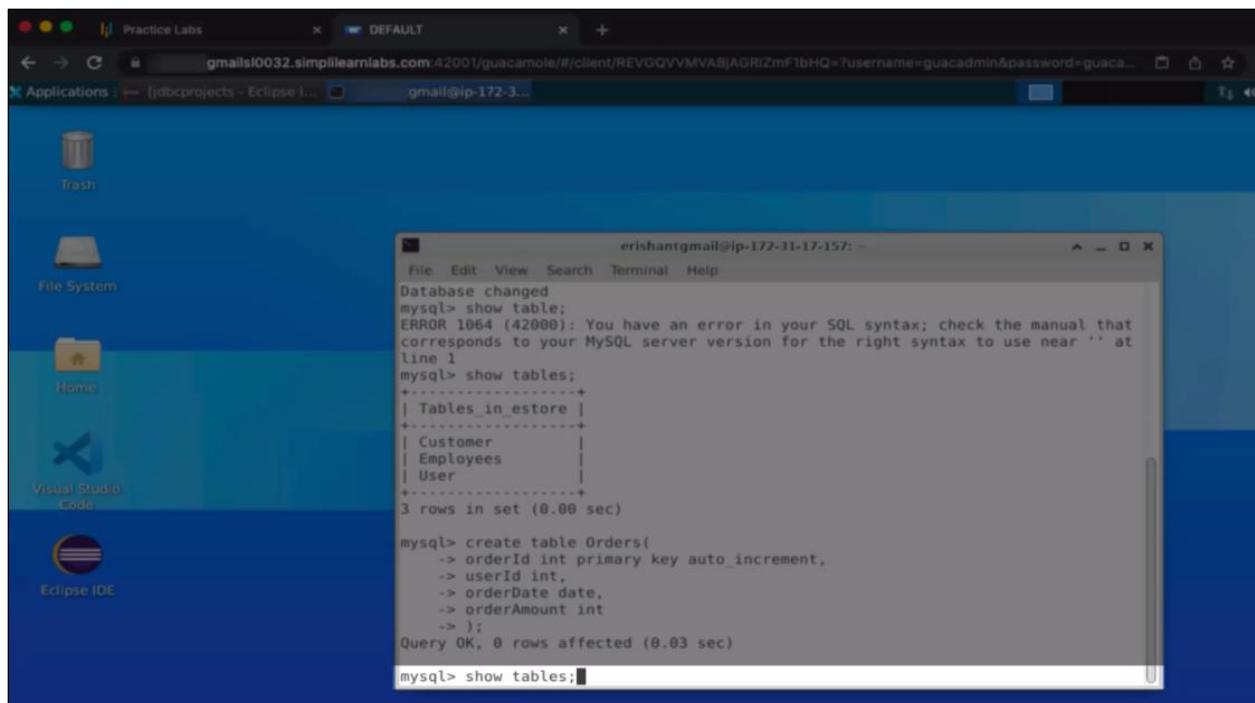
1.6 Define **attributes** for the table Orders



```
erishant@gmailip-172-31-17-157: ~
File Edit View Search Terminal Help
mysql> use estore;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A
Database changed
mysql> show table;
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that
corresponds to your MySQL server version for the right syntax to use near '' at
line 1
mysql> show tables;
+-----+
| Tables_in_estore |
+-----+
| Customer          |
| Employees         |
| User              |
+-----+
3 rows in set (0.00 sec)

mysql> create table Orders(
    > orderId int primary key auto_increment,
    > userId int,
    > orderDate date,
    > orderAmount int;
```

1.7 Run the **show tables;** command

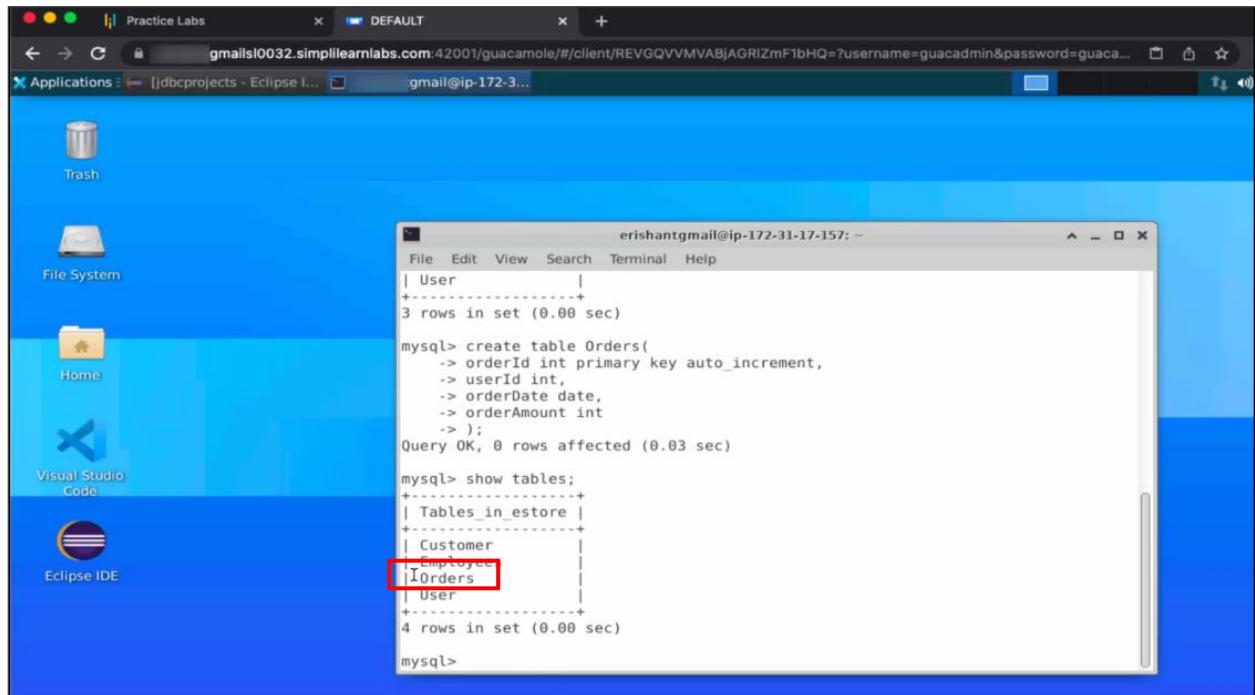


```
erishant@gmailip-172-31-17-157: ~
File Edit View Search Terminal Help
Database changed
mysql> show table;
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that
corresponds to your MySQL server version for the right syntax to use near '' at
line 1
mysql> show tables;
+-----+
| Tables_in_estore |
+-----+
| Customer          |
| Employees         |
| User              |
+-----+
3 rows in set (0.00 sec)

mysql> create table Orders(
    > orderId int primary key auto_increment,
    > userId int,
    > orderDate date,
    > orderAmount int
    > );
Query OK, 0 rows affected (0.03 sec)

mysql> show tables;
```

You can see the **Orders** table created.



The screenshot shows a Linux desktop environment with a blue Gnome-style interface. A terminal window is open in the foreground, displaying MySQL command-line interface output. The terminal window title is "erishant@gmail.com@ip-172-31-17-157: ~". The command history shows the creation of the "Orders" table and a subsequent "show tables" command which lists the "Orders" table among other existing tables like "Customer", "Employee", and "User". The "Orders" table entry is highlighted with a red box.

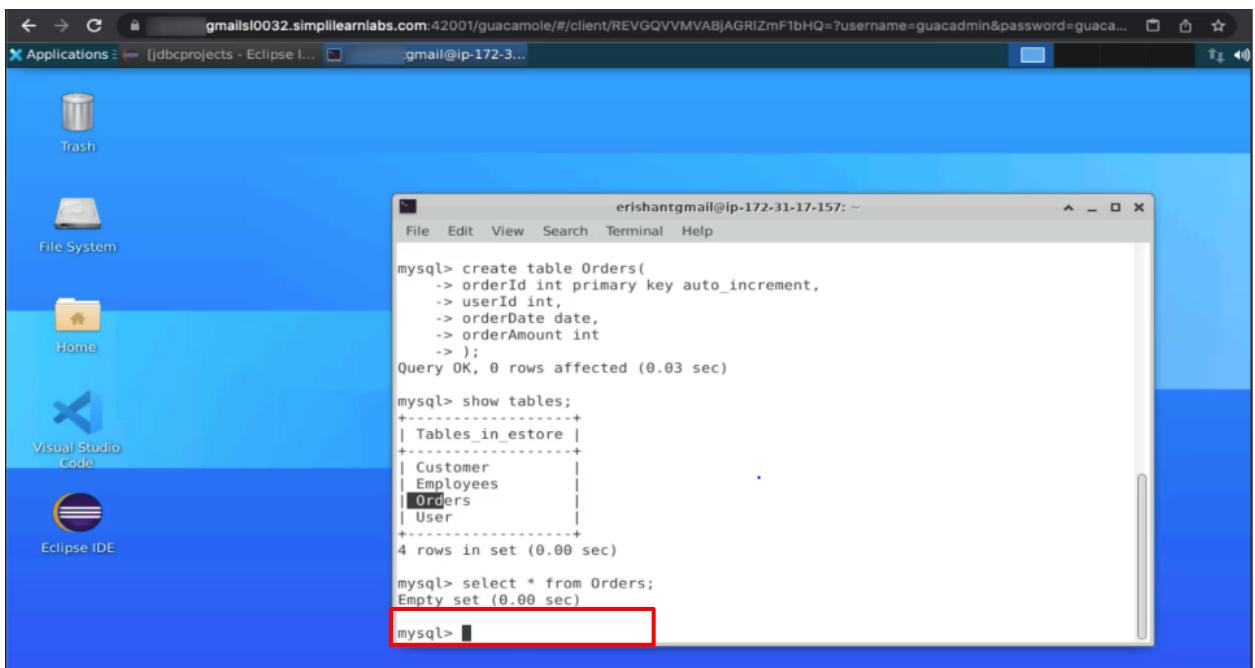
```
erishant@gmail.com@ip-172-31-17-157: ~
File Edit View Search Terminal Help
| User
+-----+
3 rows in set (0.00 sec)

mysql> create table Orders(
-> orderId int primary key auto_increment,
-> userId int,
-> orderDate date,
-> orderAmount int
-> );
Query OK, 0 rows affected (0.03 sec)

mysql> show tables;
+-----+
| Tables_in_estore |
+-----+
| Customer
| Employee
| Orders
| User
+-----+
4 rows in set (0.00 sec)

mysql>
```

1.8 Run the **select * from Orders;** command to see the empty set in the Orders table



The screenshot shows a Linux desktop environment with a blue Gnome-style interface. A terminal window is open in the foreground, displaying MySQL command-line interface output. The terminal window title is "erishant@gmail.com@ip-172-31-17-157: ~". The command history shows the creation of the "Orders" table and a subsequent "show tables" command which lists the "Orders" table among other existing tables like "Customer", "Employee", and "User". The "Orders" table entry is highlighted with a red box. Below it, a "select * from Orders;" command is run, resulting in an "Empty set (0.00 sec)" message, also highlighted with a red box.

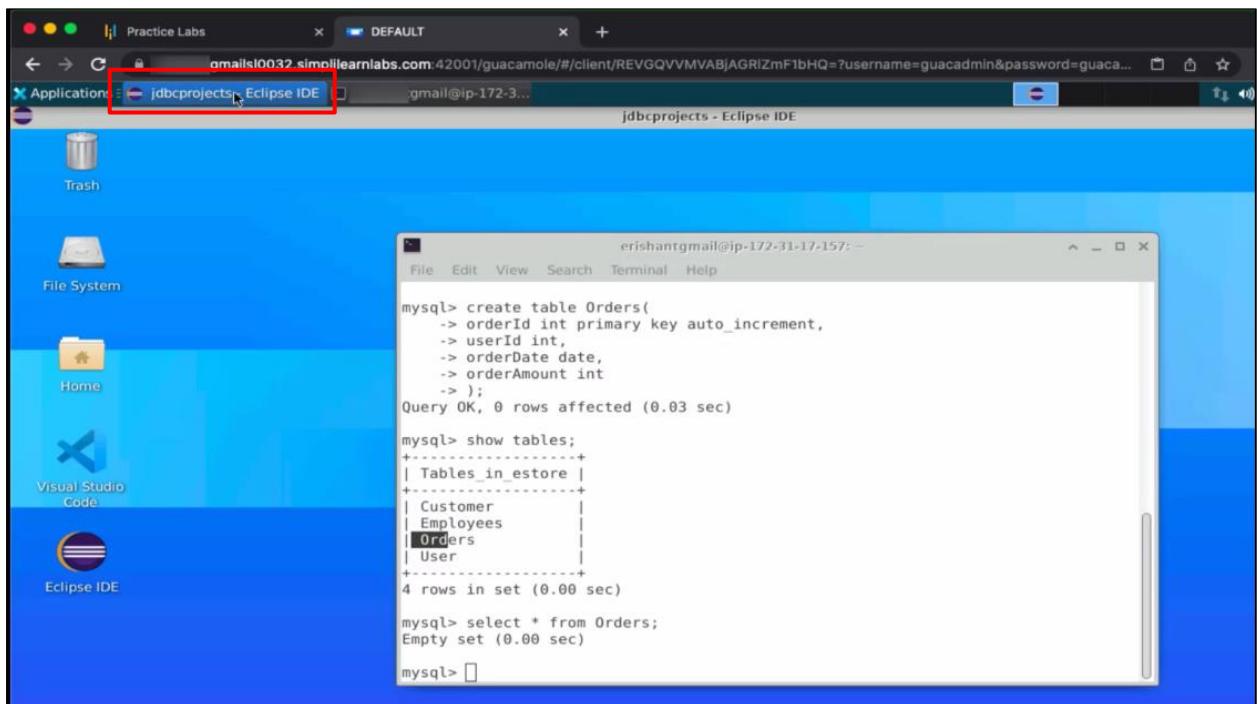
```
erishant@gmail.com@ip-172-31-17-157: ~
File Edit View Search Terminal Help
mysql> create table Orders(
-> orderId int primary key auto_increment,
-> userId int,
-> orderDate date,
-> orderAmount int
-> );
Query OK, 0 rows affected (0.03 sec)

mysql> show tables;
+-----+
| Tables_in_estore |
+-----+
| Customer
| Employee
| Orders
| User
+-----+
4 rows in set (0.00 sec)

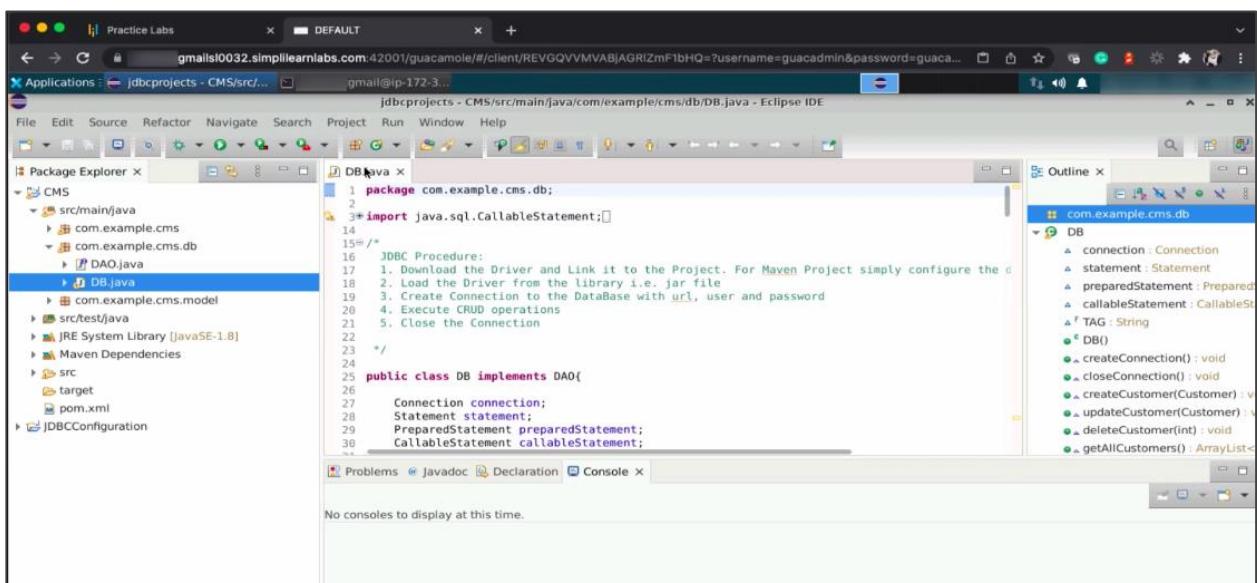
mysql> select * from Orders;
Empty set (0.00 sec)
mysql>
```

Step 2: Create the executionTransaction method

2.1 Open Eclipse IDE



2.2 Open the DB.java file



2.3 Create an `executeTransaction` method to write transaction management code

```

 259     statement.addBatch(sql1);
 260     statement.addBatch(sql2);
 261     statement.addBatch(sql3);
 262     //statement.addBatch(sql4);
 263     //statement.addBatch(sql5);

 264     int[] results = statement.executeBatch();

 265     /*String sql = "insert into Employees values(null, ?, ?, ?)";
 266     preparedStatement = connection.prepareStatement(sql);

 267     preparedStatement.setString(1, "George");
 268     preparedStatement.setString(2, "george@example.com");
 269     preparedStatement.setInt(3, 30000);
 270     preparedStatement.addBatch();

 271     preparedStatement.setString(1, "Harry");
 272     preparedStatement.setString(2, "harry@example.com");
 273     preparedStatement.setInt(3, 34000);
 274     preparedStatement.addBatch();

 275     preparedStatement.setString(1, "Sia");
 276     preparedStatement.setString(2, "sia@example.com");
 277     preparedStatement.setInt(3, 40000);
 278     preparedStatement.addBatch();

 279     int[] results = preparedStatement.executeBatch();*/
 280
 281     System.out.println("Batch Executed");
 282
 283 } catch (Exception e) {
 284     System.out.println("Exception Occurred: "+e);
 285 }
 286
 287 }

 288 public void executeTransaction I
 289
 290 void is an invalid type for the variable executeTransaction
  
```

Writable Smart Insert 295 : 25 : 9461

2.4 Write a `try-catch` block to manage all potential errors during code execution

```

 263     //statement.addBatch(sql5);

 264     int[] results = statement.executeBatch();

 265     /*String sql = "insert into Employees values(null, ?, ?, ?)";
 266     preparedStatement = connection.prepareStatement(sql);

 267     preparedStatement.setString(1, "George");
 268     preparedStatement.setString(2, "george@example.com");
 269     preparedStatement.setInt(3, 30000);
 270     preparedStatement.addBatch();

 271     preparedStatement.setString(1, "Harry");
 272     preparedStatement.setString(2, "harry@example.com");
 273     preparedStatement.setInt(3, 34000);
 274     preparedStatement.addBatch();

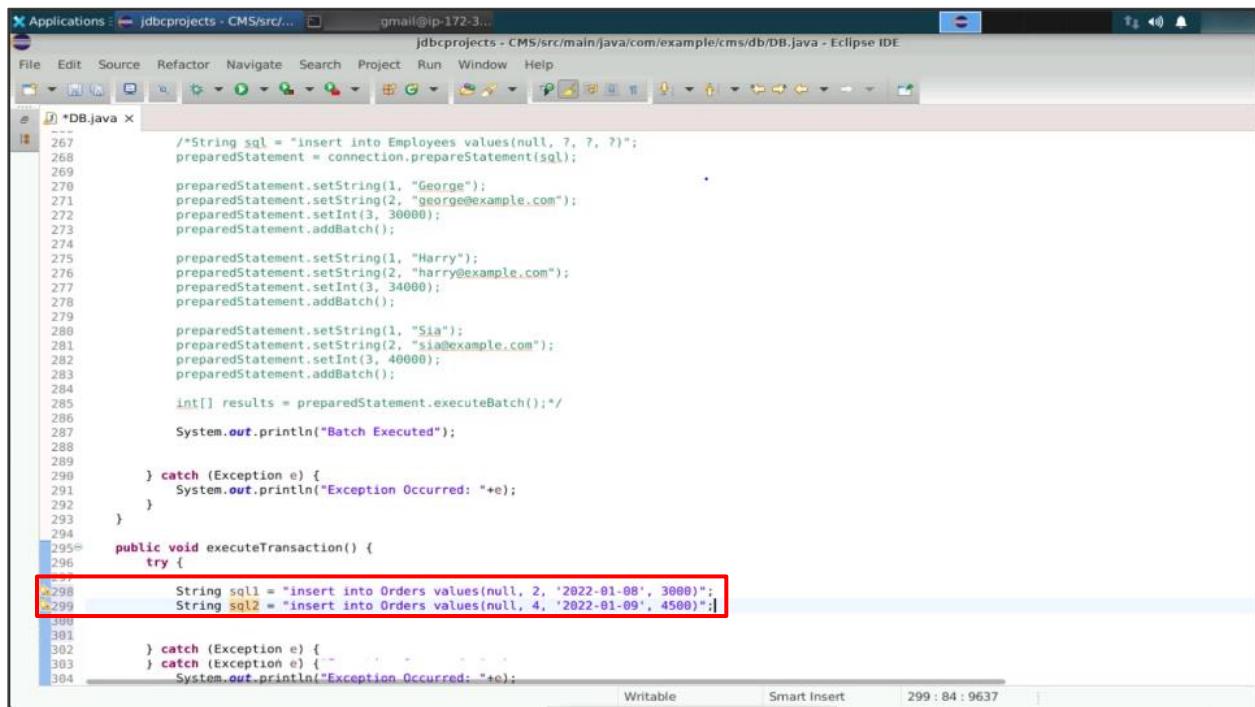
 275     preparedStatement.setString(1, "Sia");
 276     preparedStatement.setString(2, "sia@example.com");
 277     preparedStatement.setInt(3, 40000);
 278     preparedStatement.addBatch();

 279     int[] results = preparedStatement.executeBatch();*/
 280
 281     System.out.println("Batch Executed");
 282
 283 } catch (Exception e) {
 284     System.out.println("Exception Occurred: "+e);
 285 }
 286
 287 }

 288 public void executeTransaction() {
 289     try {
 290         /* */
 291         } catch (Exception e) {
 292             // TODO: handle exception
 293         }
 294     }
  
```

Syntax error on token "try", delete this token Writable Smart Insert 298 : 27 [9]

2.5 Create two SQL statements to perform the **insert** operation



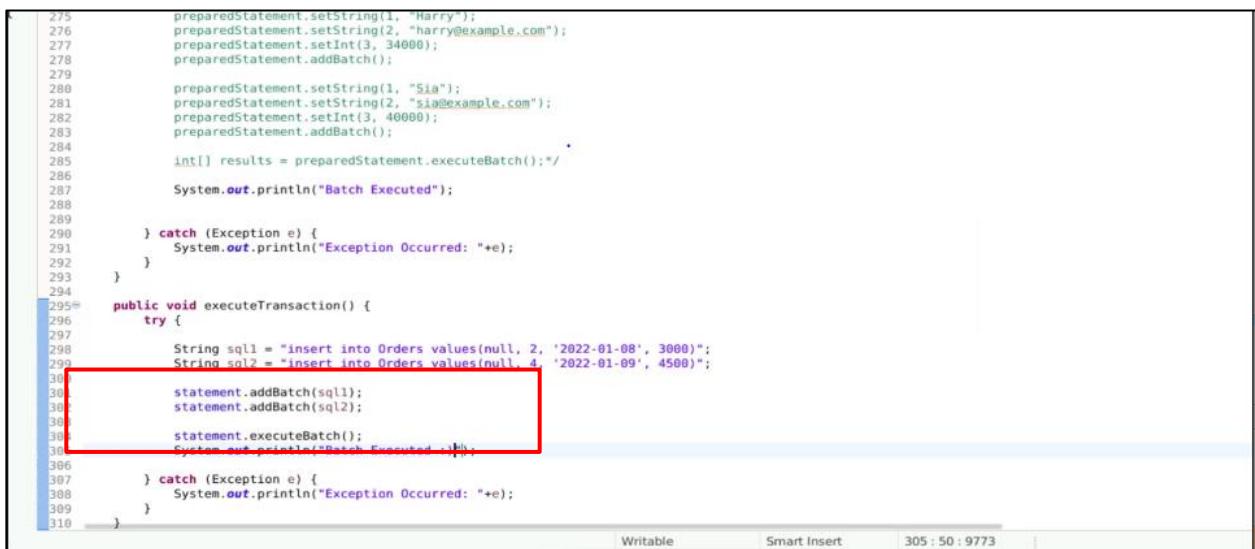
```

 267  *String sql = "insert into Employees values(null, ?, ?, ?)";
 268  preparedStatement = connection.prepareStatement(sql);
 269
 270  preparedStatement.setString(1, "George");
 271  preparedStatement.setString(2, "george@example.com");
 272  preparedStatement.setInt(3, 30000);
 273  preparedStatement.addBatch();
 274
 275  preparedStatement.setString(1, "Harry");
 276  preparedStatement.setString(2, "harry@example.com");
 277  preparedStatement.setInt(3, 34000);
 278  preparedStatement.addBatch();
 279
 280  preparedStatement.setString(1, "Sia");
 281  preparedStatement.setString(2, "sia@example.com");
 282  preparedStatement.setInt(3, 40000);
 283  preparedStatement.addBatch();
 284
 285  int[] results = preparedStatement.executeBatch();/*
 286
 287  System.out.println("Batch Executed");
 288
 289
 290  } catch (Exception e) {
 291      System.out.println("Exception Occurred: "+e);
 292  }
 293
 294
 295  public void executeTransaction() {
 296      try {
 297
 298          String sql1 = "insert into Orders values(null, 2, '2022-01-08', 3000)";
 299          String sql2 = "insert into Orders values(null, 4, '2022-01-09', 4500)";
 300
 301
 302      } catch (Exception e) {
 303          System.out.println("Exception Occurred: "+e);
 304      }
 305
 306
 307  } catch (Exception e) {
 308      System.out.println("Exception Occurred: "+e);
 309  }
 310
}

```

The code block shows Java code for performing batch insert operations. It defines a class with methods for preparing statements, adding batches, and executing them. Two specific SQL statements are defined: one for the 'Employees' table and two for the 'Orders' table. The 'executeBatch()' method is called on each prepared statement, and the results are stored in an array. Finally, a catch block handles any exceptions that might occur during the execution.

2.6 Create a batch and execute the batch statement to run the SQL statements created above



```

 275  preparedStatement.setString(1, "Harry");
 276  preparedStatement.setString(2, "harry@example.com");
 277  preparedStatement.setInt(3, 34000);
 278  preparedStatement.addBatch();
 279
 280  preparedStatement.setString(1, "Sia");
 281  preparedStatement.setString(2, "sia@example.com");
 282  preparedStatement.setInt(3, 40000);
 283  preparedStatement.addBatch();
 284
 285  int[] results = preparedStatement.executeBatch();/*
 286
 287  System.out.println("Batch Executed");
 288
 289
 290  } catch (Exception e) {
 291      System.out.println("Exception Occurred: "+e);
 292  }
 293
 294
 295  public void executeTransaction() {
 296      try {
 297
 298          String sql1 = "insert into Orders values(null, 2, '2022-01-08', 3000)";
 299          String sql2 = "insert into Orders values(null, 4, '2022-01-09', 4500)";
 300
 301          statement.addBatch(sql1);
 302          statement.addBatch(sql2);
 303
 304          statement.executeBatch();
 305          System.out.println("Batch Executed");
 306
 307      } catch (Exception e) {
 308          System.out.println("Exception Occurred: "+e);
 309      }
 310
}

```

This code block is similar to the previous one but uses a different approach. It creates a new statement object and adds the two 'Orders' insert statements to it using the 'addBatch()' method. Then, it executes the batch using 'executeBatch()' and prints a message indicating the batch was executed. A red box highlights the section of code where the statements are added to the batch.

Step 3: Call the created method and checking the output

3.1 Open the App.java file

```

    Applications : jdbcprojects - CMS/src...      gmail@ip-172-3...
    jdbcprojects - CMS/src/main/java/com/example/cms/db/DB.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer X DB.java X Outline X
  CMS
    src/main/java
      com.example.cms
        App.java
        DAO.java
        DB.java
        com.example.cms.model
      src/test/java
      JRE System Library [JavaSE-1.8]
      Maven Dependencies
    src
      target
      pom.xml
  JDBCConfiguration

273     preparedStatement.addBatch();
274
275     preparedStatement.setString(1, "Harry");
276     preparedStatement.setString(2, "harry@example.com");
277     preparedStatement.setInt(3, 34000);
278     preparedStatement.addBatch();
279
280     preparedStatement.setString(1, "Sia");
281     preparedStatement.setString(2, "sia@example.com");
282     preparedStatement.setInt(3, 40000);
283     preparedStatement.addBatch();
284
285     int[] results = preparedStatement.executeBatch();/*
286
287     System.out.println("Batch Executed");
288
289
290     } catch (Exception e) {
291         System.out.println("Exception Occurred: "+e);
292     }

```

No consoles to display at this time.

3.2 Write db.executeTransaction(); to call the created method

```

  DB.java *App.java X
10
11 //db.createCustomer(customer);
12 //db.updateCustomer(customer);
13
14 //db.deleteCustomer(3);
15
16 //System.out.println();
17
18 ArrayList<Customer> customers = db.getAllCustomers();
19 customers.forEach( cRef -> System.out.println(cRef));
20
21 db.closeConnection();/*
22
23 Scanner scanner = new Scanner(System.in);
24 //System.out.println("Enter Name: ");
25 //String name = scanner.nextLine();
26
27 //System.out.println("Enter Password: ");
28 //String password = scanner.nextLine();
29
30 //System.out.println("Enter Customer ID:");
31 //int cid = scanner.nextInt();
32
33 //scanner.close();
34
35
36 DB db = new DB();
37 db.createConnection();
38 //db.executeProcedure(name, password);
39 //db.executeProcedure(cid);
40
41 //db.executeSQLStatementsInBatch();
42
43 db.executeTransaction();
44
45 db.closeConnection();
46
47

```

3.3 Run the code, and you will see the output **Batch Executed :)** with no errors

```

1 jdbcprojects - CMS/src... gmail@ip-172-3...
2 jdbcprojects - CMS/src/main/java/com/example/cms/App.java - Eclipse IDE
3 File Refactor Navigate Search Project Run Window Help
4 Run App Problems Javadoc Declaration Console
5 <terminated> App [Java Application] /usr/eclipse/plugins/org.eclipse.jst.j.openjdk.hotspot.jre...
6 [DB] Driver Loaded
7 [DB] Connection Created
8 [DB] Executed :
9 [DB] Connection Closed. Close Status: true
10
11 //db.createCustomer(customer);
12 //db.updateCustomer(customer);
13
14 //db.deleteCustomer(3);
15
16 //System.out.println();
17
18 ArrayList<Customer> customers = db.getAllCustomers();
19 customers.forEach(cRef -> System.out.println(cRef));
20
21 db.closeConnection();*/
22
23 //Scanner scanner = new Scanner(System.in);
24 //System.out.println("Enter Name: ");
25 //String name = scanner.nextLine();
26
27 //System.out.println("Enter Password: ");
28 //String password = scanner.nextLine();
29
30 //System.out.println("Enter Customer ID:");
31 //int cid = scanner.nextInt();
32
33 //scanner.close();
34
35 DB db = new DB();
36 db.createConnection();
37 //db.executeProcedure(name, password);
38 //db.executeProcedure(cid);
39
40 //db.executeSQLStatementsInBatch();
41 db.executeTransaction();
42
43 db.closeConnection();
44
45 //Scanner scanner = new Scanner(System.in);
46 //System.out.println("Enter Name: ");
47 //String name = scanner.nextLine();
48
49 //System.out.println("Enter Password: ");
50 //String password = scanner.nextLine();
51
52 //System.out.println("Enter Customer ID: ");
53 //int cid = scanner.nextInt();
54
55 //scanner.close();
56
57 DB db = new DB();
58 db.createConnection();
59 //db.executeProcedure(name, password);
60 //db.executeProcedure(cid);
61
62 //db.executeSQLStatementsInBatch();
63
64 db.executeTransaction();
65
66 db.closeConnection();
67

```

3.4 Return to the Terminal Emulator and run the **select * from Orders;** command

```

1 Applications jdbcprojects - CMS/src... gmail@ip-172-31-17-157: ~
2 jdbcprojects - CMS/src/main/java/com/example/cms/App.java - Eclipse IDE
3 File Edit View Search Terminal Help
4 mysql> create table Orders(
5      -> orderId int primary key auto_increment,
6      -> userId int,
7      -> orderDate date,
8      -> orderAmount int
9      -> );
10 Query OK, 0 rows affected (0.03 sec)
11
12 mysql> show tables;
13 +-----+-----+
14 | Tables_in_estore | Customer |
15 +-----+-----+
16 | Employees          |
17 | Orders              |
18 | User                |
19 +-----+-----+
20
21 4 rows in set (0.00 sec)
22
23 mysql> select * from Orders;
24 Empty set (0.00 sec)
25
26 mysql> select * from Orders;
27

```

You will see the following orders inserted in the **Orders** table:

The screenshot shows the Eclipse IDE interface with multiple windows open. The left-hand window displays Java code for a class named DB.java, which interacts with a MySQL database to create customers, update customer details, and manage orders. The right-hand window is a terminal window titled 'erishant@gmail.ip-172-31-17-157 ~' showing MySQL command-line interactions. The terminal output includes:

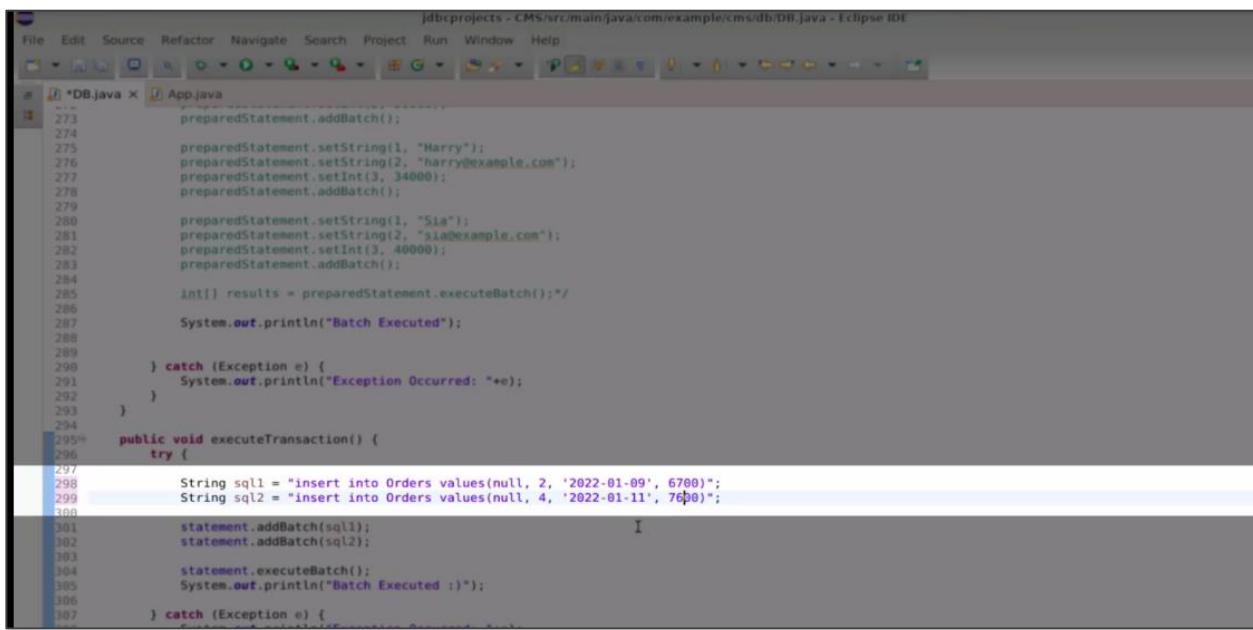
```
mysql> show tables;
+-----+
| Tables_in_estore |
+-----+
| Customer          |
| Employees         |
| Orders            |
| User              |
+-----+
4 rows in set (0.00 sec)

mysql> select * from Orders;
Empty set (0.00 sec)

mysql> select * from Orders;
+-----+
| orderId | userId | orderDate | orderAmount |
+-----+
| 1       | 2      | 2022-01-08 | 3000        |
| 2       | 4      | 2022-01-09 | 4500        |
+-----+
2 rows in set (0.00 sec)
```

Step 4: Use the auto-commit feature

4.1 Return to the DB.java file and change it to insert two more orders



The screenshot shows the Eclipse IDE interface with the title bar "jdbcprojects - CMS/src/main/java/com/example/cms/db/DB.java - Eclipse IDE". The code editor displays Java code for a class named DB. The code includes preparing statements, adding them to a batch, and executing the batch. It also handles exceptions and defines a method to execute a transaction. The code is annotated with line numbers from 273 to 307.

```
File Edit Source Refactor Navigate Search Project Run Window Help
File *DB.java X App.java
273     preparedStatement.addBatch();
274
275     preparedStatement.setString1("Marry");
276     preparedStatement.setString2("harry@example.com");
277     preparedStatement.setInt(3, 34000);
278     preparedStatement.addBatch();
279
280     preparedStatement.setString1("Sia");
281     preparedStatement.setString2("sia@example.com");
282     preparedStatement.setInt(3, 40000);
283     preparedStatement.addBatch();
284
285     int[] results = preparedStatement.executeBatch();/*
286
287     System.out.println("Batch Executed");
288
289 } catch (Exception e) {
290     System.out.println("Exception Occurred: "+e);
291 }
292 }
293 */
294
295 public void executeTransaction() {
296     try {
297
298         String sql1 = "insert into Orders values(null, 2, '2022-01-09', 6700)";
299         String sql2 = "insert into Orders values(null, 4, '2022-01-11', 7600)";
300
301         statement.addBatch(sql1);
302         statement.addBatch(sql2);
303
304         statement.executeBatch();
305         System.out.println("Batch Executed :)");
306
307     } catch (Exception e) {
```

4.2 Add the auto-commit feature in a **try-catch** block to disable auto-commit and add **commit()**

```

    279     preparedStatement.setString(1, "Sia");
    280     preparedStatement.setString(2, "sia@example.com");
    281     preparedStatement.setInt(3, 40000);
    282     preparedStatement.addBatch();
    283
    284     int[] results = preparedStatement.executeBatch();/*
    285
    286     System.out.println("Batch Executed");
    287
    288
    289 } catch (Exception e) {
    290     System.out.println("Exception Occurred: "+e);
    291 }
    292
    293
    294 public void executeTransaction() {
    295     try {
    296         connection.setAutoCommit(false);
    297
    298         String sql1 = "insert into Orders values(null, 2, '2022-01-09', 6700)";
    299         String sql2 = "insert into Orders values(null, 4, '2022-01-11', 7600)";
    300
    301         statement.addBatch(sql1);
    302         statement.addBatch(sql2);
    303
    304         statement.executeBatch();
    305
    306         connection.commit(); // We will ourselves do the commit
    307         System.out.println("Batch Executed and Transaction Committed :)");
    308
    309     } catch (Exception e) {
    310         System.out.println("Exception Occurred: "+e);
    311     }
    312 }
    313
    314
    315 }
    316
  
```

Writable Smart Insert 308 : 74 : 9896

4.3 Add a **rollback** method within the catch block for any errors that may occur during execution

```

    285     int[] results = preparedStatement.executeBatch();/*
    286
    287     System.out.println("Batch Executed");
    288
    289
    290 } catch (Exception e) {
    291     System.out.println("Exception Occurred: "+e);
    292 }
    293
    294 public void executeTransaction() {
    295     try {
    296         connection.setAutoCommit(false);
    297
    298         String sql1 = "insert into Orders values(null, 2, '2022-01-09', 6700)";
    299         String sql2 = "insert into Orders values(null, 4, '2022-01-11', 7600)";
    300
    301         statement.addBatch(sql1);
    302         statement.addBatch(sql2);
    303
    304         statement.executeBatch();
    305
    306         connection.commit(); // We will ourselves do the commit
    307         System.out.println("Batch Executed and Transaction Committed :)");
    308
    309     } catch (Exception e) {
    310         System.out.println("Exception Occurred: "+e);
    311         try {
    312             System.out.println("Rolling Back the Transaction");
    313             connection.rollback();
    314         } catch (SQLException e1) {
    315             e1.printStackTrace();
    316         }
    317     }
    318
    319 }
    320
    321 }
    322
  
```

Writable Smart Insert 314 : 65 : 10072

4.4 Re-run this code, and you will see **Batch Executed and Transaction committed :)** in the output without error

```

//db.createCustomer(customer);
//db.updateCustomer(customer);

//db.deleteCustomer(3);

//System.out.println();

ArrayList<Customer> customers = db.getAllCustomers();
customers.forEach(cRef -> System.out.println(cRef));

db.closeConnection();*/

//Scanner scanner = new Scanner(System.in);
//System.out.println("Enter Name: ");
//String name = scanner.nextLine();

//System.out.println("Enter Password: ");
//String password = scanner.nextLine();

//System.out.println("Enter Customer ID:");
//int cid = scanner.nextInt();

//scanner.close();

DB db = new DB();
db.createConnection();
//db.executeProcedure(name, password);
//db.executeProcedure(cid);

//db.executeSQLStatementsInBatch();

db.beginTransaction();

db.closeConnection();

```

4.5 Go back to the **Terminal Emulator** and run the **select * from Orders;** command. You will see two more orders inserted in the Orders table.

```

//db.createCustomer(customer);
//db.updateCustomer(customer);

//db.deleteCustomer(3);

//System.out.println();

ArrayList<Customer> customers = db.getAllCustomers();
customers.forEach(cRef -> System.out.println(cRef));
db.closeConnection();*/
//Scanner scanner = new Scanner(System.in);
//System.out.println("Enter Name: ");
//String name = scanner.nextLine();

//System.out.println("Enter Password: ");
//String password = scanner.nextLine();

//System.out.println("Enter Customer ID:");
//int cid = scanner.nextInt();

//scanner.close();

DB db = new DB();
db.createConnection();
//db.executeProcedure(name, password);
//db.executeProcedure(cid);

//db.executeSQLStatementsInBatch();

```

Step 5: Use the insert and update operations

5.1 Write an insert and an update operation. Insert a new order and update the old order with ID 4

```
 285
 286     int[] results = preparedStatement.executeBatch();/*
 287
 288     System.out.println("Batch Executed");
 289
 290
 291 } catch (Exception e) {
 292     System.out.println("Exception Occurred: "+e);
 293 }
 294 }
 295
 296 public void executeTransaction() {
 297     try {
 298
 299         connection.setAutoCommit(false);
 300
 301         String sql1 = "insert into Orders values(null, 5, '2022-01-09', 7700)";
 302         String sql2 = "update Orders set amount = 7600 where orderId = 4";
 303
 304         statement.addBatch(sql1);
 305         statement.addBatch(sql2);
 306
 307         statement.executeBatch();
 308         connection.commit(); // We will ourselves do the commit
 309         System.out.println("Batch Executed and Transaction Committed :)");
 310
 311     } catch (Exception e) {
 312         System.out.println("Exception Occurred: "+e);
 313         try {
 314             System.out.println("Rolling Back the Transaction...");
 315             connection.rollback();
 316         } catch (SQLException el) {
 317             el.printStackTrace();
 318         }
 319     }
 320 }
```

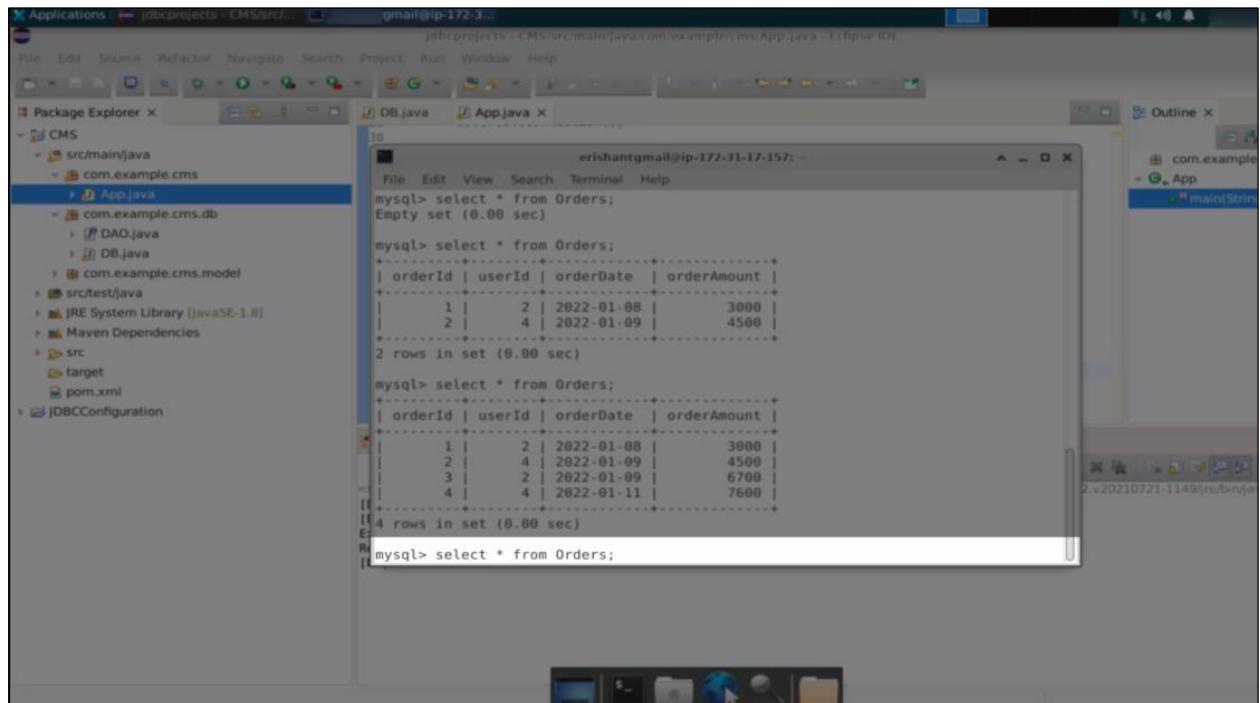
5.2 Re-run the code, and you will see **Exception Occurred** in the console

The screenshot shows the Eclipse IDE interface with the following details:

- Title Bar:** jdbcprojects - CMS/src/... and jdbcprojects - CMS/src/main/java/com/example/cms/App.java - Eclipse IDE
- Toolbar:** Standard Eclipse toolbar with icons for Refactor, Navigate, Search, Project, Run, Window, Help, and various file operations.
- Left Panel:** Shows the project structure with "App.java" selected.
- Right Panel:** Contains tabs for Problems, Javadoc, Declaration, and Console. The Console tab is active, displaying the following terminal output:

```
<terminated> App [Java Application] /usr/eclipse/plugins/org.eclipse.justj.openjdk.hotspot.jdk11
[DB] Driver Loaded
[DB] Connection Created
Exception Occurred: java.sql.BatchUpdateException: Unknown column 'amount'
Rolling Back the Transaction...
[DB] Connection Closed. Close Status: true
```
- Code Editor:** Displays the Java code for the `App.java` file, which includes database operations like creating and updating customers, and executing SQL statements in a batch.

5.3 Return to the terminal and run the **select * from Orders;** command. You will see that the new order is not inserted, and the old order is also not updated



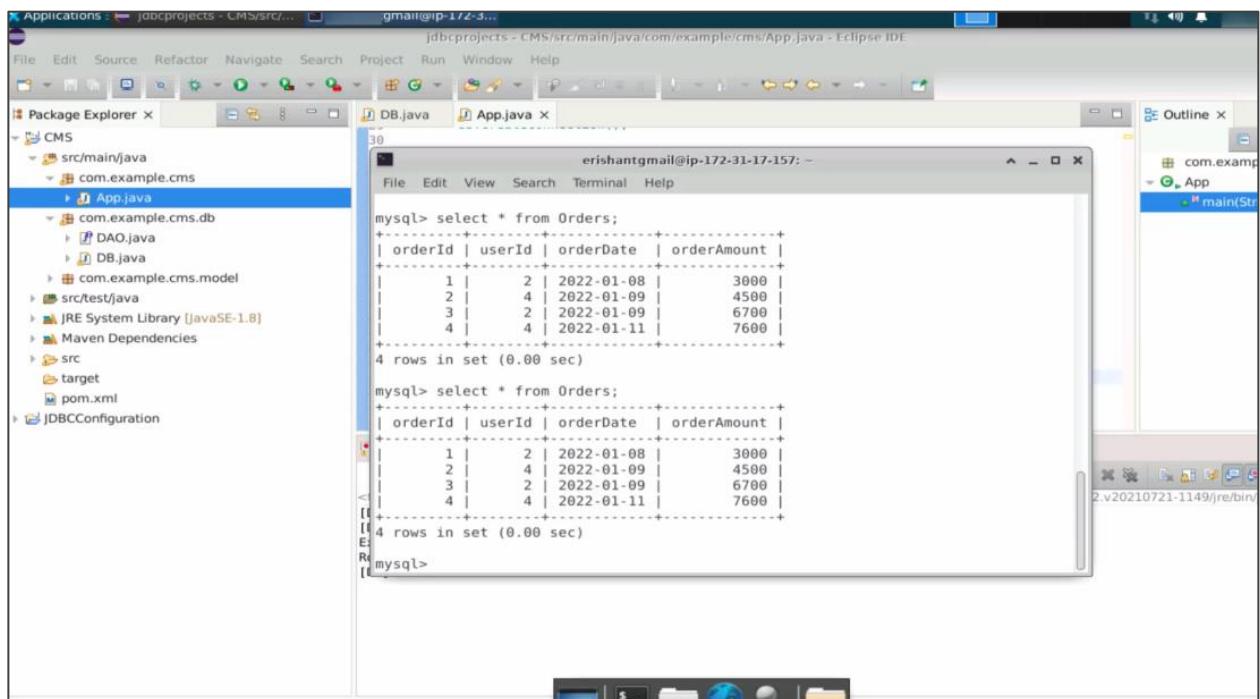
The screenshot shows the Eclipse IDE interface with the MySQL terminal window open. The terminal window displays the following SQL queries and their results:

```
mysql> select * from Orders;
Empty set (0.00 sec)

mysql> select * from Orders;
+-----+-----+-----+-----+
| orderId | userId | orderDate | orderAmount |
+-----+-----+-----+-----+
| 1 | 2 | 2022-01-08 | 3000 |
| 2 | 4 | 2022-01-09 | 4500 |
+-----+-----+-----+-----+
2 rows in set (0.00 sec)

mysql> select * from Orders;
+-----+-----+-----+-----+
| orderId | userId | orderDate | orderAmount |
+-----+-----+-----+-----+
| 1 | 2 | 2022-01-08 | 3000 |
| 2 | 4 | 2022-01-09 | 4500 |
| 3 | 2 | 2022-01-09 | 6700 |
| 4 | 4 | 2022-01-11 | 7600 |
+-----+-----+-----+-----+
4 rows in set (0.00 sec)

mysql> select * from Orders;
```



The screenshot shows the Eclipse IDE interface with the MySQL terminal window open. The terminal window displays the following SQL queries and their results:

```
mysql> select * from Orders;
+-----+-----+-----+-----+
| orderId | userId | orderDate | orderAmount |
+-----+-----+-----+-----+
| 1 | 2 | 2022-01-08 | 3000 |
| 2 | 4 | 2022-01-09 | 4500 |
| 3 | 2 | 2022-01-09 | 6700 |
| 4 | 4 | 2022-01-11 | 7600 |
+-----+-----+-----+-----+
4 rows in set (0.00 sec)

mysql> select * from Orders;
+-----+-----+-----+-----+
| orderId | userId | orderDate | orderAmount |
+-----+-----+-----+-----+
| 1 | 2 | 2022-01-08 | 3000 |
| 2 | 4 | 2022-01-09 | 4500 |
| 3 | 2 | 2022-01-09 | 6700 |
| 4 | 4 | 2022-01-11 | 7600 |
+-----+-----+-----+-----+
4 rows in set (0.00 sec)

mysql>
```

5.4 Change the column name to **orderamount** in the updated SQL code

The screenshot shows the Eclipse IDE interface with the title bar "Applications: jdbcprojects - CMS/src...". The code editor window displays the file "DB.java" with the following content:

```

285     int[] results = preparedStatement.executeBatch();*/
286
287     System.out.println("Batch Executed");
288
289 }
290 } catch (Exception e) {
291     System.out.println("Exception Occurred: "+e);
292 }
293 }
294 }
295 public void executeTransaction() {
296     try {
297         connection.setAutoCommit(false);
298
299         String sql1 = "insert into Orders values(null, 5, '2022-01-09', 7700)";
300         String sql2 = "update Orders set orderamount = 4900 where orderId = 4";
301
302         statement.addBatch(sql1);
303         statement.addBatch(sql2);
304
305         statement.executeBatch();
306         connection.commit(); // We will ourselves do the commit
307         System.out.println("Batch Executed and Transaction Committed :)");
308
309     } catch (Exception e) {
310         System.out.println("Exception Occurred: "+e);
311         try {
312             System.out.println("Rolling Back the Transaction...");
313             connection.rollback();
314         } catch (SQLException el) {
315             el.printStackTrace();
316         }
317     }
318 }
319 }
320 }
321 }
322 }

```

A red box highlights the line "String sql2 = "update Orders set orderamount = 4900 where orderId = 4";" in the code.

5.5 Re-run the code, and you will see the code executed without any errors

The screenshot shows the Eclipse IDE interface with the title bar "jdbcprojects - CMS/src...". The code editor window displays the file "App.java" with the following content:

```

//db.createCustomer(customer);
//db.updateCustomer(customer);

//db.deleteCustomer(3);

//System.out.println();

ArrayList<Customer> customers = db.getAllCustomers();
customers.forEach(cRef -> System.out.println(cRef));

db.closeConnection();/*

//Scanner scanner = new Scanner(System.in);
//System.out.println("Enter Name: ");
//String name = scanner.nextLine();

//System.out.println("Enter Password: ");
//String password = scanner.nextLine();

//System.out.println("Enter Customer ID:");
//int cid = scanner.nextInt();

scanner.close();

DB db = new DB();
db.createConnection();
//db.executeProcedure(name, password);
//db.executeProcedure(cid);

//db.executeSQLStatementsInBatch();

db.executeTransaction();

```

The "Console" tab is selected, showing the output of the application's execution:

```

<terminated> App [Java Application] /usr/eclipse/plugins/org.eclipse.justj.openjdk.hotspot.jdk11
[DB] Driver Loaded
[DB] Connection Created
Batch Executed and Transaction Committed :)
[DB] Connection Closed. Close Status: true

```

A red box highlights the output text in the console.

5.6 Return to the terminal and run the **select * from Orders** command. You will see a new row added and the **orderamount** of order ID 4 gets updated.

The screenshot shows the Eclipse IDE interface with the following details:

- Title Bar:** Applications > jdbcprojects - CMS > ...
- Toolbar:** File, Edit, Source, Refactor, Navigate, Search, Project, Run, Window, Help.
- Left Panel (Project Explorer):** Shows DB.java and App.java X.
- Right Panel (Terminal):** Displays MySQL command-line interface output.

MySQL Terminal Output:

```
erishant@gmail:ip-172-31-17-152: ~
File Edit View Search Terminal Help

mysql> select * from Orders;
+-----+-----+-----+-----+
| orderId | userId | orderDate | orderAmount |
+-----+-----+-----+-----+
| 1 | 2 | 2022-01-08 | 3000 |
| 2 | 4 | 2022-01-09 | 4500 |
| 3 | 2 | 2022-01-09 | 6700 |
| 4 | 4 | 2022-01-11 | 7600 |
+-----+-----+-----+-----+
4 rows in set (0.00 sec)

mysql> select * from Orders;
+-----+-----+-----+-----+
| orderId | userId | orderDate | orderAmount |
+-----+-----+-----+-----+
| 1 | 2 | 2022-01-08 | 3000 |
| 2 | 4 | 2022-01-09 | 4500 |
| 3 | 2 | 2022-01-09 | 6700 |
| 4 | 4 | 2022-01-11 | 7600 |
+-----+-----+-----+-----+
4 rows in set (0.00 sec)

mysql> select * from Orders;
```

The screenshot shows the Eclipse IDE interface with several windows open. The leftmost window displays Java code for a class named DB.java. The code includes methods for creating, updating, and deleting customers, as well as a main method that connects to a MySQL database, creates a customer, and inserts several orders. The rightmost window is a terminal window titled 'erishant@gmail.ip-172-31-17-157: ~' showing MySQL command-line output. The output shows the creation of a customer and the insertion of six orders into the Orders table. One specific row from the MySQL output is highlighted with a red box.

```
File Edit View Search Terminal Help
```

```
File Edit View Search Terminal Help
```

```
erishant@gmail.ip-172-31-17-157: ~
```

	orderId	userId	orderDate	orderAmount
1	1	2	2022-01-08	3000
2	2	4	2022-01-09	4500
3	3	2	2022-01-09	6700
4	4	4	2022-01-11	7600

4 rows in set (0.00 sec)

```
mysql> select * from Orders;
```

```
select * from Orders;
```

```
^C
```

```
mysql> select * from Orders;
```

orderId	userId	orderDate	orderAmount
1	2	2022-01-08	3000
2	2	2022-01-09	4500
3	2	2022-01-09	6700
4	4	2022-01-11	4900
6	5	2022-01-09	7700

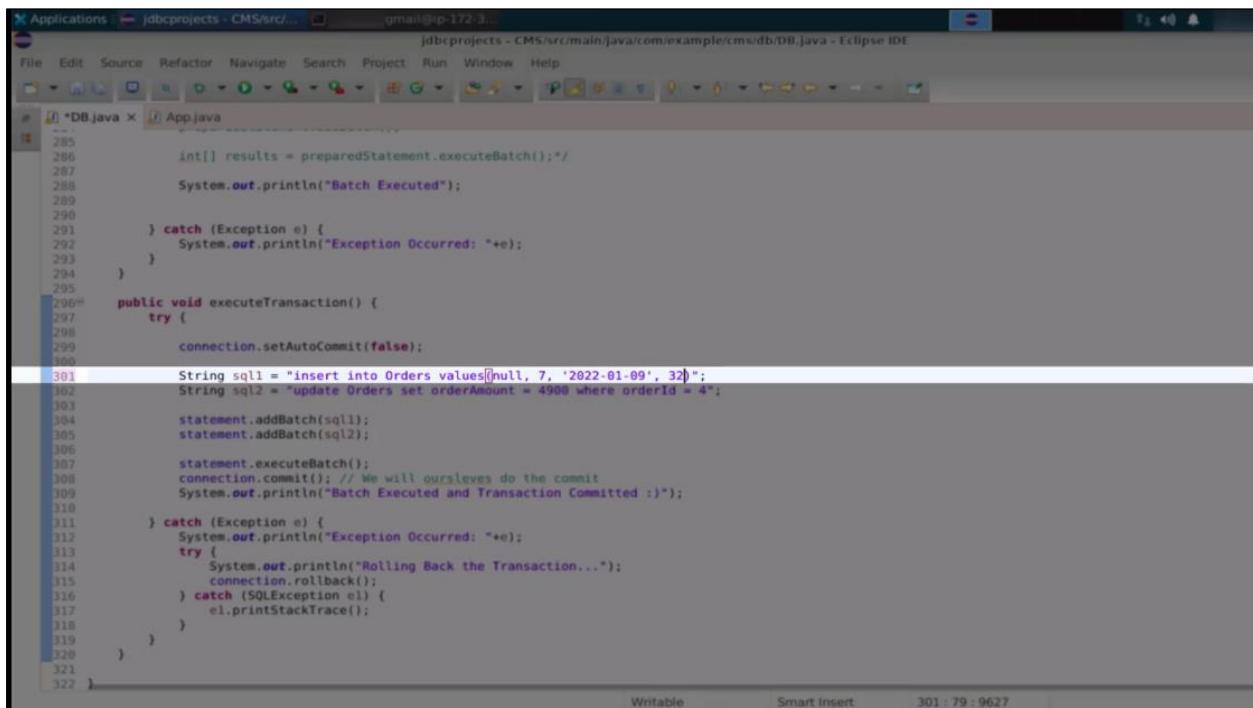
5 rows in set (0.00 sec)

```
mysql>
```

```
DB db = new DB();
db.createConnection();
//db.executeProcedure(name, pa
//db.executeProcedure(cid);
//db.executeSQLStatementsInBatch();
db.beginTransaction();
db.closeConnection();
```

Step 6: Use the delete operation

6.1 Insert another order by changing the values in the insert statement



The screenshot shows the Eclipse IDE interface with the title bar "jdbcprojects - CMS/src/main/java/com/example/cms/db/DB.java - Eclipse IDE". The code editor displays Java code for database operations:

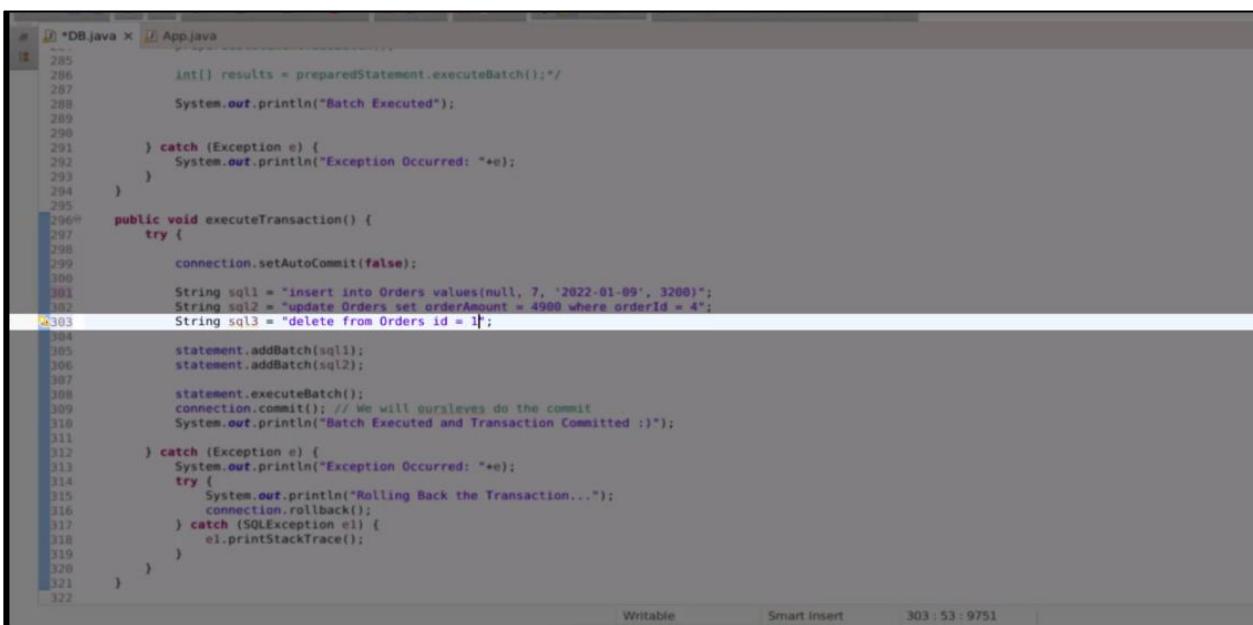
```

285     int[] results = preparedStatement.executeBatch();/*
286
287     System.out.println("Batch Executed");
288
289
290 } catch (Exception e) {
291     System.out.println("Exception Occurred: "+e);
292 }
293
294 }
295
296 public void executeTransaction() {
297     try {
298
299         connection.setAutoCommit(false);
300
301         String sql1 = "insert into Orders values(null, 7, '2022-01-09', 320";
302         String sql2 = "update Orders set orderAmount = 4900 where orderId = 4";
303
304         statement.addBatch(sql1);
305         statement.addBatch(sql2);
306
307         statement.executeBatch();
308         connection.commit(); // We will ourselves do the commit
309         System.out.println("Batch Executed and Transaction Committed :"));
310
311     } catch (Exception e) {
312         System.out.println("Exception Occurred: "+e);
313         try {
314             System.out.println("Rolling Back the Transaction...");
315             connection.rollback();
316         } catch (SQLException el) {
317             el.printStackTrace();
318         }
319     }
320 }
321
322 }

```

The code includes logic for executing a batch of SQL statements (insert and update) and committing the transaction. It also includes exception handling and rollback logic.

6.2 Write the delete statement



The screenshot shows the Eclipse IDE interface with the title bar "jdbcprojects - CMS/src/main/java/com/example/cms/db/DB.java - Eclipse IDE". The code editor displays Java code for database operations, similar to the previous screenshot but with an additional delete statement:

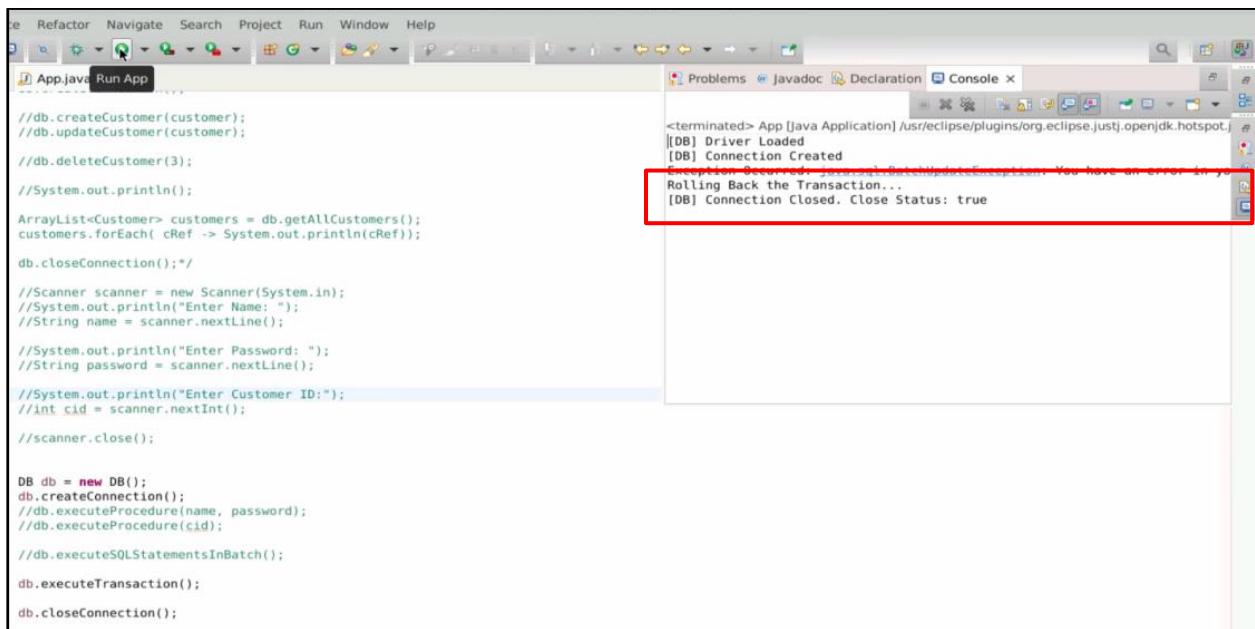
```

285     int[] results = preparedStatement.executeBatch();/*
286
287     System.out.println("Batch Executed");
288
289
290 } catch (Exception e) {
291     System.out.println("Exception Occurred: "+e);
292 }
293
294 }
295
296 public void executeTransaction() {
297     try {
298
299         connection.setAutoCommit(false);
300
301         String sql1 = "insert into Orders values(null, 7, '2022-01-09', 3200)";
302         String sql2 = "update Orders set orderAmount = 4900 where orderId = 4";
303         String sql3 = "delete from Orders id = 1";
304
305         statement.addBatch(sql1);
306         statement.addBatch(sql2);
307
308         statement.executeBatch();
309         connection.commit(); // We will ourselves do the commit
310         System.out.println("Batch Executed and Transaction Committed :"));
311
312     } catch (Exception e) {
313         System.out.println("Exception Occurred: "+e);
314         try {
315             System.out.println("Rolling Back the Transaction...");
316             connection.rollback();
317         } catch (SQLException el) {
318             el.printStackTrace();
319         }
320     }
321
322 }

```

The code now includes a delete statement (line 303) along with the insert and update statements. The rest of the code remains the same, including exception handling and transaction management.

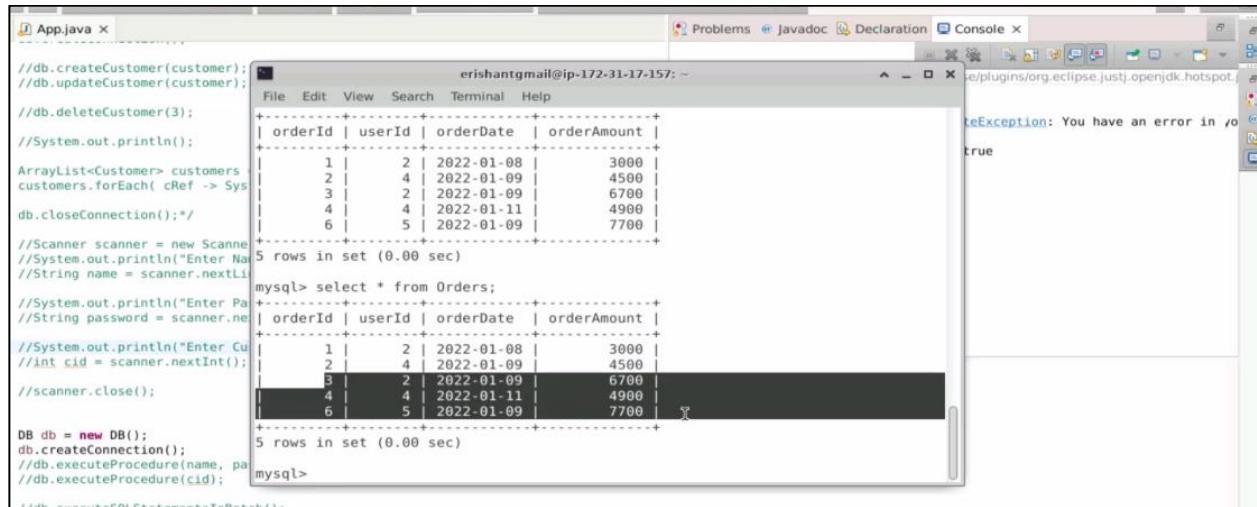
6.3 Re-run the code, and you will see the rollback function is called because one or more batches failed during execution



The screenshot shows the Eclipse IDE interface. On the left, the code editor displays `App.java` with Java code. On the right, the `Console` tab shows the application's output. A red box highlights the error message in the console:

```
<terminated> App [java Application] /usr/eclipse/plugins/org.eclipse.jst.jdk.hotspot.jvm
[DB] Driver Loaded
[DB] Connection Created
Exception occurred in thread "main" java.sql.BatchUpdateException: You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'Delete from customer where id = ?' at line 1
Rolling Back the Transaction...
[DB] Connection Closed. Close Status: true
```

6.4 Run the **select * from Orders;** command in the Terminal Emulator, and you will see that there is no change in the table



The screenshot shows the Eclipse IDE interface. On the left, there is a code editor window titled "App.java" containing Java code. On the right, there is a terminal window titled "erishant@gmail.com@ip-172-31-17-157: ~" showing MySQL command-line interface output.

```
//db.createCustomer(customer);
//db.updateCustomer(customer);
//db.deleteCustomer(3);
//System.out.println();
ArrayList<Customer> customers;
customers.forEach(cRef -> Sys
db.closeConnection();*/
//Scanner scanner = new Scanner
//System.out.println("Enter Na
//String name = scanner.nextLine();
//System.out.println("Enter Pa
//String password = scanner.nextLine();
//System.out.println("Enter Cu
//int cid = scanner.nextInt();
//scanner.close();
DB db = new DB();
db.createConnection();
//db.executeProcedure(name, pa
//db.executeProcedure(cid);
//db.executeProcedure(name, pa
//db.executeProcedure(cid);
```

MySQL terminal output:

```
File Edit View Search Terminal Help
+-----+-----+-----+
| orderId | userId | orderDate | orderAmount |
+-----+-----+-----+
| 1 | 2 | 2022-01-08 | 3000 |
| 2 | 4 | 2022-01-09 | 4500 |
| 3 | 2 | 2022-01-09 | 6700 |
| 4 | 4 | 2022-01-11 | 4900 |
| 6 | 5 | 2022-01-09 | 7700 |
+-----+-----+-----+
5 rows in set (0.00 sec)

mysql> select * from Orders;
+-----+-----+-----+
| orderId | userId | orderDate | orderAmount |
+-----+-----+-----+
| 1 | 2 | 2022-01-08 | 3000 |
| 2 | 4 | 2022-01-09 | 4500 |
| 3 | 2 | 2022-01-09 | 6700 |
| 4 | 4 | 2022-01-11 | 4900 |
| 6 | 5 | 2022-01-09 | 7700 |
+-----+-----+-----+
5 rows in set (0.00 sec)

mysql>
```

By following these steps, you have successfully managed transaction execution to perform all SQL statements together, ensuring data integrity and consistency within the database.