ece-database-2023 LAB 4

Java Database Connectivity (JDBC)

Exercise 1

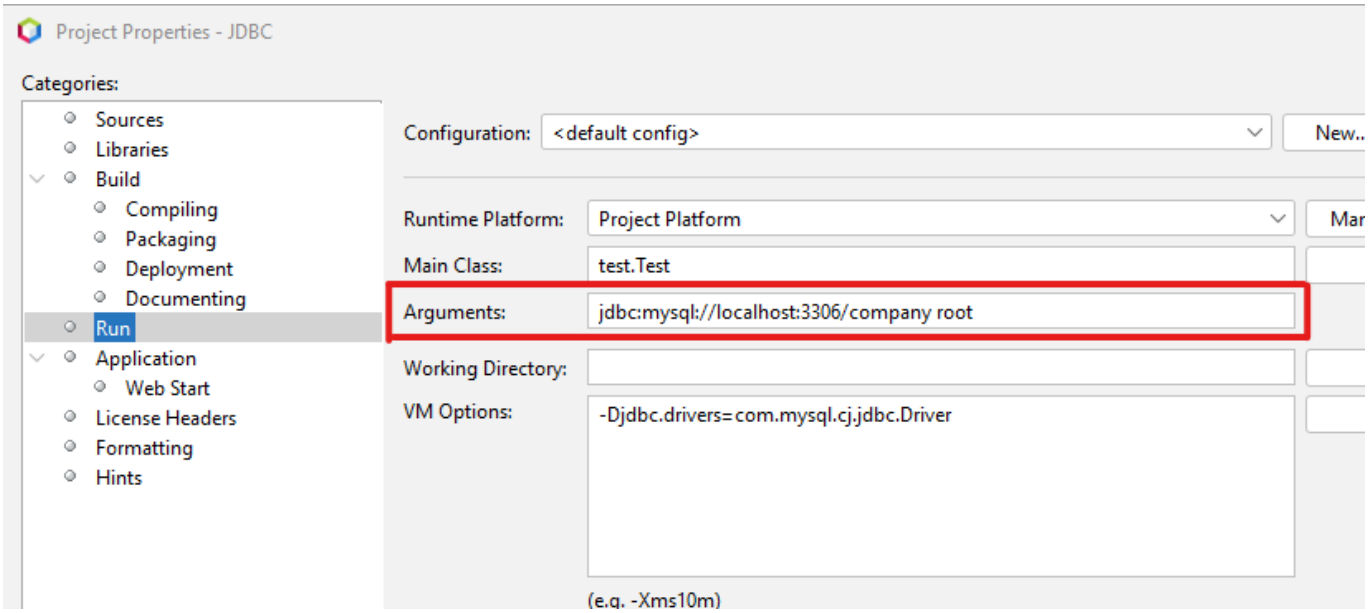
Create Connection with MySQL :

The constructor takes the database's **URL**, **the user's login** and **password** as parameters and establishes a connection to the database. The connection is stored in a (private) instance field, since all the other methods of the class use the connection

`DataAccess.java`

```
package model;

import java.sql.Connection;
import java.sql.SQLException;
import java.sql.DriverManager;

public class DataAccess {

  private Connection connection;

  public DataAccess(String url, String login, String password) throws
      SQLException {
    connection = DriverManager.getConnection(url, login, password);
    System.out.println("connected to " + url);
  }
}
```

When using NetBeans, you can set these arguments in the **"Arguments"** section of the **"Run"** configuration of your project: in the corresponding field, list all the arguments on a single line, separated with space characters.



Workaround for NetBeans Bug

The parameters of the constructor are taken from the ☐ **args** parameter of the main method.

```
public class Test {
  /**
   * @param args the command line arguments
   *
   * @throws java.lang.Exception
   */
  public static void main(String[] args) throws Exception {
    ...
}
```

There is a conditional check at the beginning of the main method :

```
if (args.length == 2) {
  args = Arrays.copyOf(args, 3);
  args[2] = "";
}
```

This code checks if there are exactly two command-line arguments (args.length == 2). If there are only two arguments, it adds an empty string as the third argument. This is done as a workaround for a potential bug in NetBeans, which might not pass the third argument correctly in certain situations.

Creating a Data Access Object

After the workaround, the code proceeds to create an instance of the DataAccess class:

```
// create a data access object
data = new DataAccess(args[0], args[1], args[2]);
```

It uses the command-line arguments (args) as parameters to the DataAccess constructor. The assumption here is that the first argument is the database URL, the second argument is the username, and the third argument is the password for the database connection.

Test.java

```
package test;

import java.util.Arrays;
import model.DataAccess;
/**
 *
 * @author Jean-Michel Busca
 */
public class Test {
  /**
   * @param args the command line arguments
   *
   * @throws java.lang.Exception
   */
  public static void main(String[] args) throws Exception {
        // work around Netbeans bug
        if (args.length == 2) {
          args = Arrays.copyOf(args, 3);
          args[2] = "";
        }

        // create a data access object
        data = new DataAccess(args[0], args[1], args[2]);

        // access the database using high-level Java methods
        // ...
        // close the data access object when done
  }
}
```

Output :

```
run:
connected to jdbc:mysql://localhost:3306/company
BUILD SUCCESSFUL (total time: 0 seconds)
```

Close method :

To close a database connection in Java, you should use the **close( )** method provided by the **java.sql.Connection interface**, which is typically implemented by database connection classes like **java.sql.Connection**, **java.sql.Statement**, and **java.sql.ResultSet**.

Here's how you can close a database connection in your Java code:

DataAccess.java

```
// Method to close the database connection
public void closeConnection() {
    if (connection != null) {
        try {
            connection.close();
            System.out.println("Database connection closed.");
        } catch (SQLException e) {
            // Handle any potential exceptions here
            e.printStackTrace();
        }
    }
}
```

After that we need to call the close method from **"DataAccess"** class, as below :

Test.java

```
public static void main(String[] args) throws Exception {
    DataAccess data = null;

    // work around Netbeans bug
    if (args.length == 2) {
        args = Arrays.copyOf(args, 3);
        args[2] = "";
    }

    try {
        // create a data access object
        data = new DataAccess(args[0], args[1], args[2]);

        // access the database using high-level Java methods
        // ...
    } finally {
        // close the data access object when done
        if (data != null) {
            data.closeConnection();
        }
    }


}
```

Output :

```
run:
connected to jdbc:mysql://localhost:3306/company
Database connection closed.
BUILD SUCCESSFUL (total time: 0 seconds)
```

Exercice 2

Write the method List getEmployees() that returns the number, name and salary of all the employee in the EMP table.

Note: the class EmployeeInfo is already defined in the model package. So we can use those methods from EmployeeInfo class and put them into our new method to retrieve employee information

EmployeeInfo.java

```
package model;

/**
 *
 * @author Jean-Michel Busca
 */
public class EmployeeInfo {

  private final int id;
  private final String name;
  private final float salary;

  public EmployeeInfo(int id, String name, float salary) {
    this.id = id;
    this.name = name;
    this.salary = salary;
  }

  @Override
  public String toString() {
    return "EmployeeInfo{" + "id=" + id + ", name=" + name + ", salary=" + salary
+ "}\n";
  }

  public int getId() {
    return id;
  }

  public String getName() {
```

```
    return name;
  }

  public float getSalary() {
    return salary;
  }

}
```

In DataAccess class we can add a new method to retrieve employee information using the method from class Employeeinfo.

```
// Method to get a list of EmployeeInfo objects
public List<EmployeeInfo> getEmployees() throws SQLException {
    List<EmployeeInfo> employees = new ArrayList<>();

    // SQL query without a prepared statement (be cautious of SQL injection here)
    String sql = "SELECT EID, ENAME, SAL FROM EMP";

    try (Statement statement = connection.createStatement();
        ResultSet resultSet = statement.executeQuery(sql)) {
        while (resultSet.next()) {
            int id = resultSet.getInt("EID");
            String name = resultSet.getString("ENAME");
            float salary = resultSet.getFloat("SAL");

            // Create an EmployeeInfo object and add it to the list
            EmployeeInfo employee = new EmployeeInfo(id, name, salary);
            employees.add(employee);
        }
    }

    return employees;
}
```

Output :

```
run:
connected to jdbc:mysql://localhost:3306/company
Employee ID: 7369
Employee Name: SMITH
Employee Salary: 800.0
-----------------------
Employee ID: 7499
Employee Name: ALLEN
Employee Salary: 1600.0
-----------------------
Employee ID: 7521
Employee Name: WARD
```

```
Employee Salary: 1250.0
----------------------
Employee ID: 7566
Employee Name: JONES
Employee Salary: 2975.0
----------------------
Employee ID: 7654
Employee Name: MARTIN
Employee Salary: 1250.0
----------------------
Employee ID: 7698
Employee Name: BLAKE
Employee Salary: 2850.0
----------------------
Employee ID: 7782
Employee Name: CLARK
Employee Salary: 2450.0
----------------------
Employee ID: 7788
Employee Name: SCOTT
Employee Salary: 3000.0
----------------------
Employee ID: 7839
Employee Name: KING
Employee Salary: 5000.0
----------------------
Employee ID: 7844
Employee Name: TURNER
Employee Salary: 1500.0
----------------------
Employee ID: 7876
Employee Name: ADAMS
Employee Salary: 1100.0
----------------------
Employee ID: 7900
Employee Name: JAMES
Employee Salary: 950.0
----------------------
Employee ID: 7902
Employee Name: FORD
Employee Salary: 3000.0
----------------------
Employee ID: 7934
Employee Name: MILLER
Employee Salary: 1300.0
----------------------
Employee ID: 8000
Employee Name: SMITH
Employee Salary: 3000.0
----------------------
Database connection closed.
BUILD SUCCESSFUL (total time: 2 seconds)
```

Exercice 3

Write the method boolean raiseSalary(String job, float amount) that raises the salary of the employees with the specified job by the specified amount.

```
// Method to raise the salary of employees with a specified job by a specified
amount
public boolean raiseSalary(String job, float amount) throws SQLException {
    // SQL query without a prepared statement (be cautious of SQL injection here)
    String sql = "UPDATE EMP SET SAL = SAL + " + amount + " WHERE JOB = '" + job +
"'";

    try (Statement statement = connection.createStatement()) {
        // Execute the SQL update
        int rowsAffected = statement.executeUpdate(sql);

        // Check if any rows were affected (salary updated)
        return rowsAffected > 0;
    }
}
```

Call this method in our main

```
// Call the raiseSalary method to raise the salary of employees with a specified
job
String jobToRaise = "CLERK"; // Replace with the job you want to target
float raiseAmount = 100;    // Replace with the amount by which to raise the
salary

boolean success = data.raiseSalary(jobToRaise, raiseAmount);

if (success) {
    System.out.println("Salary raised successfully.");
} else {
    System.out.println("No employees with the specified job found.");
}
```

previous Output :

```
run:
connected to jdbc:mysql://localhost:3306/company
Salary raised successfully.
Employee ID: 7369
Employee Name: SMITH
Employee Salary: 800.0
-----------------------
...
```

```
----------------------
Employee ID: 7876
Employee Name: ADAMS
Employee Salary: 1100.0
----------------------
Employee ID: 7900
Employee Name: JAMES
Employee Salary: 950.0
----------------------
...
----------------------
Employee ID: 7934
Employee Name: MILLER
Employee Salary: 1300.0
----------------------
...
----------------------
Database connection closed.
BUILD SUCCESSFUL (total time: 2 seconds)
```

New Output :

```
run:
connected to jdbc:mysql://localhost:3306/company
Salary raised successfully.
Employee ID: 7369
Employee Name: SMITH
Employee Salary: 900.0
----------------------
...
----------------------
Employee ID: 7876
Employee Name: ADAMS
Employee Salary: 1200.0
----------------------
Employee ID: 7900
Employee Name: JAMES
Employee Salary: 1050.0
----------------------
...
----------------------
Employee ID: 7934
Employee Name: MILLER
Employee Salary: 1400.0
----------------------
...
----------------------
Database connection closed.
BUILD SUCCESSFUL (total time: 2 seconds)
```

to perform an SQL injection attack that raises the salary of all employees, we just need to change our SQL query, like below.

```
String sql = "UPDATE EMP SET SAL = SAL + ? WHERE JOB = ? OR 1 = 1 ";
```

the JOB value was "CLERK" and amount of salary to raise was 1.00$. but because of "1=1" statement, it will change all salary values instead of CLERK salary.

Output :

```
run:
connected to jdbc:mysql://localhost:3306/company
Salary raised successfully.
Employee ID: 7369
Employee Name: SMITH
Employee Salary: 901.0
----------------------
Employee ID: 7499
Employee Name: ALLEN
Employee Salary: 1601.0
----------------------
Employee ID: 7521
Employee Name: WARD
Employee Salary: 1251.0
----------------------
Employee ID: 7566
Employee Name: JONES
Employee Salary: 2976.0
----------------------
Employee ID: 7654
Employee Name: MARTIN
Employee Salary: 1251.0
----------------------
Employee ID: 7698
Employee Name: BLAKE
Employee Salary: 2851.0
----------------------
Employee ID: 7782
Employee Name: CLARK
Employee Salary: 2451.0
----------------------
Employee ID: 7788
Employee Name: SCOTT
Employee Salary: 3001.0
----------------------
Employee ID: 7839
Employee Name: KING
Employee Salary: 5001.0
----------------------
Employee ID: 7844
```

```
Employee Name: TURNER
Employee Salary: 1501.0
----------------------
Employee ID: 7876
Employee Name: ADAMS
Employee Salary: 1201.0
----------------------
Employee ID: 7900
Employee Name: JAMES
Employee Salary: 1051.0
----------------------
Employee ID: 7902
Employee Name: FORD
Employee Salary: 3001.0
----------------------
Employee ID: 7934
Employee Name: MILLER
Employee Salary: 1401.0
----------------------
Employee ID: 8000
Employee Name: SMITH
Employee Salary: 3001.0
----------------------
Database connection closed.
BUILD SUCCESSFUL (total time: 0 seconds)
```

Exercice 4

Prepared statements are more efficient and secure than regular statements because they allow you to precompile SQL queries and reuse them with different parameter values. They provide the following benefits:

Security: Prepared statements automatically handle parameterization of user input, making it much harder for malicious users to perform SQL injection attacks.

Performance: Prepared statements are precompiled and can be reused with different parameter values. This reduces the overhead of query compilation, resulting in better performance for frequently executed queries.

Here's how you can create versions of the getEmployees and raiseSalary methods that use prepared statements and ensure that SQL injection is no longer possible:

`DataAccess.java`

1. getEmployeesPS Method using Prepared Statement :

```
// Method to get a list of EmployeeInfo objects using a prepared statement
public List<EmployeeInfo> getEmployeesPS() throws SQLException {
    List<EmployeeInfo> employees = new ArrayList<>();

    // SQL query with a prepared statement
    String sql = "SELECT EID, ENAME, SAL FROM EMP";
```

```
    try (PreparedStatement statement = connection.prepareStatement(sql)) {
        // Set the job parameter for the prepared statement
        //...

        try (ResultSet resultSet = statement.executeQuery()) {
            while (resultSet.next()) {
                int id = resultSet.getInt("EID");
                String name = resultSet.getString("ENAME");
                float salary = resultSet.getFloat("SAL");

                // Create an EmployeeInfo object and add it to the list
                EmployeeInfo employee = new EmployeeInfo(id, name, salary);
                employees.add(employee);
            }
        }
    }

    return employees;
}
```

2. raiseSalaryPS Method using Prepared Statement:

```
// Method to raise the salary of employees with a specified job using a prepared
statement
public boolean raiseSalaryPS(String job, float amount) throws SQLException {
    // SQL query with a prepared statement to update salary
    String sql = "UPDATE EMP SET employee_salary = employee_salary + ? WHERE
employee_job = ?";

    try (PreparedStatement statement = connection.prepareStatement(sql)) {
        // Set parameters for the prepared statement
        statement.setFloat(1, amount);
        statement.setString(2, job);

        // Execute the SQL update
        int rowsAffected = statement.executeUpdate();

        // Check if any rows were affected (salary updated)
        return rowsAffected > 0;
    }
}
```

Exercice 5

The methods that we will create next will use the methods from class DepartmentInfo :

DepartmentInfo.java

```
package model;

/**
 *
 * @author Jean-Michel Busca
 */
public class EmployeeInfo {

  private final int id;
  private final String name;
  private final float salary;

  public EmployeeInfo(int id, String name, float salary) {
    this.id = id;
    this.name = name;
    this.salary = salary;
  }

  @Override
  public String toString() {
    return "EmployeeInfo{" + "id=" + id + ", name=" + name + ", salary=" + salary
+ "}\n";
  }

  public int getId() {
    return id;
  }

  public String getName() {
    return name;
  }

  public float getSalary() {
    return salary;
  }

}
```

here are two implementations of the getDepartments method: one using statements and the other using prepared statements. Both implementations allow us to specify criteria and retrieve departments matching those criteria. If a criterion is null, it is omitted from the query.

DataAccess.java

1. "getDepartments" Method using Statements :

```
// Method to retrieve departments matching the specified criteria using statements
public List<DepartmentInfo> getDepartments(Integer id, String name, String
location) throws SQLException {
    List<DepartmentInfo> departments = new ArrayList<>();
```

```
    StringBuilder sql = new StringBuilder("SELECT * FROM DEPT WHERE 1=1");

    if (id != null) {
        sql.append(" AND DID = ").append(id);
    }

    if (name != null) {
        sql.append(" AND DNAME = '").append(name).append("'");
    }

    if (location != null) {
        sql.append(" AND DLOC = '").append(location).append("'");
    }

    try (Statement statement = connection.createStatement();
            ResultSet resultSet = statement.executeQuery(sql.toString())) {

        while (resultSet.next()) {
            int departmentId = resultSet.getInt("DID");
            String departmentName = resultSet.getString("DNAME");
            String departmentLocation = resultSet.getString("DLOC");

            DepartmentInfo department = new DepartmentInfo(departmentId,
departmentName, departmentLocation);
            departments.add(department);
        }
    }

    return departments;
}
```

Input :

```
List<DepartmentInfo> departments = data.getDepartments(null,null,null);
```

Output :

```
run:
connected to jdbc:mysql://localhost:3306/company
D ID: 10
D Name: ACCOUNTING
D LOC: NEW-YORK
-----------------------
D ID: 20
D Name: RESEARCH
D LOC: DALLAS
-----------------------
D ID: 30
```

```
D Name: SALES
D LOC: CHICAGO
-----------------------
D ID: 40
D Name: OPERATIONS
D LOC: BOSTON
-----------------------
Database connection closed.
BUILD SUCCESSFUL (total time: 0 seconds)
```

Input :

```
List<DepartmentInfo> departments = data.getDepartments(null,null,"NEW-YORK);
```

Output :

```
run:
connected to jdbc:mysql://localhost:3306/company
D ID: 10
D Name: ACCOUNTING
D LOC: NEW-YORK
-----------------------
Database connection closed.
BUILD SUCCESSFUL (total time: 0 seconds)
```

Input :

```
List<DepartmentInfo> departments = data.getDepartments(null,"RESEARCH","DALLAS);
```

Output :

```
run:
connected to jdbc:mysql://localhost:3306/company
D ID: 20
D Name: RESEARCH
D LOC: DALLAS
-----------------------
Database connection closed.
BUILD SUCCESSFUL (total time: 0 seconds)
```

2. "getDepartments" Method using PreparedStatements :

```
// Method to retrieve departments matching the specified criteria using prepared
statements
public List<DepartmentInfo> getDepartments(Integer id, String name, String
location) throws SQLException {
    List<DepartmentInfo> departments = new ArrayList<>();
    String sql = "SELECT * FROM DEPT WHERE ";
    boolean hasCriteria = false;

    if (id != null) {
        sql += "DID = ? ";
        hasCriteria = true;
    }

    if (name != null) {
        if (hasCriteria) {
            sql += "AND ";
        }
        sql += "DNAME = ? ";
        hasCriteria = true;
    }

    if (location != null) {
        if (hasCriteria) {
            sql += "AND ";
        }
        sql += "DLOC = ? ";
    }

    try (PreparedStatement statement = connection.prepareStatement(sql)) {
        int parameterIndex = 1;

        if (id != null) {
            statement.setInt(parameterIndex++, id);
        }

        if (name != null) {
            statement.setString(parameterIndex++, name);
        }

        if (location != null) {
            statement.setString(parameterIndex, location);
        }

        try (ResultSet resultSet = statement.executeQuery()) {
            while (resultSet.next()) {
                int departmentId = resultSet.getInt("DID");
                String departmentName = resultSet.getString("DNAME");
                String departmentLocation = resultSet.getString("DLOC");

                DepartmentInfo department = new DepartmentInfo(departmentId,
departmentName, departmentLocation);
                departments.add(department);
            }
```

```
            }
        }

        return departments;
    }
```

Exercice 6

To execute a SELECT statement on the database and return a list of strings representing the result, we can use JDBC to execute the query and process the result set. Here's a method executeQuery that accomplishes this:

```
// Method to execute a SELECT query and return the result as a list of strings
public List<String> executeQuery(String query) throws SQLException {
    List<String> resultList = new ArrayList<>();

    try (Statement statement = connection.createStatement();
         ResultSet resultSet = statement.executeQuery(query)) {

        ResultSetMetaData metaData = resultSet.getMetaData();
        int columnCount = metaData.getColumnCount();

        // Append the header row to the result
        StringBuilder header = new StringBuilder();
        for (int i = 1; i <= columnCount; i++) {
            header.append(metaData.getColumnName(i));
            if (i < columnCount) {
                header.append(", ");
            }
        }
        resultList.add(header.toString());

        // Append each tuple to the result
        while (resultSet.next()) {
            StringBuilder tuple = new StringBuilder();
            for (int i = 1; i <= columnCount; i++) {
                tuple.append(resultSet.getString(i));
                if (i < columnCount) {
                    tuple.append(", ");
                }
            }
            resultList.add(tuple.toString());
        }
    }

    return resultList;

}
```

Now we create a query and put it into executeQuery method as a parameter :

```
String query = "SELECT EID, ENAME, SAL FROM EMP";
List<String> exQuery = data.executeQuery(query);

for (String line : exQuery) {
    String[] parts = line.split("\t");
    for (String part : parts) {
        System.out.print(part + "\t");
    }
    System.out.println(); // Move to the next line
}
```

Ouput :

```
run:
connected to jdbc:mysql://localhost:3306/company
EID, ENAME, SAL
7369, SMITH, 906.00
7499, ALLEN, 1602.00
7521, WARD, 1252.00
7566, JONES, 2977.00
7654, MARTIN, 1252.00
7698, BLAKE, 2852.00
7782, CLARK, 2452.00
7788, SCOTT, 3002.00
7839, KING, 5002.00
7844, TURNER, 1502.00
7876, ADAMS, 1206.00
7900, JAMES, 1056.00
7902, FORD, 3002.00
7934, MILLER, 1406.00
8000, SMITH, 3002.00
Database connection closed.
BUILD SUCCESSFUL (total time: 0 seconds)
```

2. executeStatement Method using Prepared Statements :

```
// Method to execute any SQL statement and return the result or the update count
public List<String> executeStatement(String statement) throws SQLException {
    List<String> resultList = new ArrayList<>();

    // Check if the statement is a query or an update
    boolean isQuery = statement.trim().toLowerCase().startsWith("select");

    try {
        if (isQuery) {
            // Execute a query using a prepared statement
            try (PreparedStatement preparedStatement =
connection.prepareStatement(statement);
```

```
                ResultSet resultSet = preparedStatement.executeQuery()) {

                ResultSetMetaData metaData = resultSet.getMetaData();
                int columnCount = metaData.getColumnCount();

                // Append the header row to the result
                StringBuilder header = new StringBuilder();
                for (int i = 1; i <= columnCount; i++) {
                    header.append(metaData.getColumnName(i));
                    if (i < columnCount) {
                        header.append("\t");
                    }
                }
                resultList.add(header.toString());

                // Process the result set and add rows to the result list
                while (resultSet.next()) {
                    StringBuilder row = new StringBuilder();
                    for (int i = 1; i <= resultSet.getMetaData().getColumnCount();
i++) {
                        row.append(resultSet.getString(i));
                        if (i < resultSet.getMetaData().getColumnCount()) {
                            row.append("\t");
                        }
                    }
                    resultList.add(row.toString());
                }
            }
        } else {
            // Execute an update statement using a prepared statement
            try (PreparedStatement preparedStatement =
connection.prepareStatement(statement)) {
                int updateCount = preparedStatement.executeUpdate();
                resultList.add(String.valueOf(updateCount)); // Add update count
to the result list
            }
        }
    } catch (SQLException e) {
        // Handle any SQL exception and add the error message to the result list
        resultList.add("Error: " + e.getMessage());
    }

    return resultList;
}
```

The executeStatement method first checks whether the statement is a SELECT query (case-insensitive check). If it's a query, it executes it using a prepared statement, processes the result set, and adds rows to the result list.

If the statement is not a SELECT query, it's assumed to be an update statement (INSERT, UPDATE, DELETE, etc.), and it's executed using a prepared statement. The method then adds the update count to the result list.

Any SQL exceptions are caught and added to the result list with an error message.

By using prepared statements, this method can handle both queries and updates while providing security against SQL injection and better performance.

Input :

```
String query1 = "SELECT EID, ENAME, SAL FROM EMP";
String query2 = "UPDATE EMP SET SAL = SAL + 100";

System.out.println("Before");
List<String> exQuery1 = data.executeStatement(query1);

for (String line : exQuery1) {
    String[] parts = line.split("\t");
    for (String part : parts) {
        System.out.print(part + "\t");
    }
    System.out.println(); // Move to the next line
}

System.out.println("values updated");
List<String> exQuery2 = data.executeStatement(query2);

for (String line : exQuery2) {
    String[] parts = line.split("\t");
    for (String part : parts) {
        System.out.print(part + "\t");
    }
    System.out.println(); // Move to the next line
}

System.out.println("After");
List<String> exQuery3 = data.executeStatement(query1);

for (String line : exQuery3) {
    String[] parts = line.split("\t");
    for (String part : parts) {
        System.out.print(part + "\t");
    }
    System.out.println(); // Move to the next line
}
```

Output :

```
run:
connected to jdbc:mysql://localhost:3306/company
Before
EID ENAME   SAL
7369    SMITH   1206.00
7499    ALLEN   1902.00
```

```
7521    WARD    1552.00
7566    JONES   3277.00
7654    MARTIN  1552.00
7698    BLAKE   3152.00
7782    CLARK   2752.00
7788    SCOTT   3302.00
7839    KING    5302.00
7844    TURNER  1802.00
7876    ADAMS   1506.00
7900    JAMES   1356.00
7902    FORD    3302.00
7934    MILLER  1706.00
8000    SMITH   3302.00
Value updated
15
After
EID ENAME   SAL
7369    SMITH   1306.00
7499    ALLEN   2002.00
7521    WARD    1652.00
7566    JONES   3377.00
7654    MARTIN  1652.00
7698    BLAKE   3252.00
7782    CLARK   2852.00
7788    SCOTT   3402.00
7839    KING    5402.00
7844    TURNER  1902.00
7876    ADAMS   1606.00
7900    JAMES   1456.00
7902    FORD    3402.00
7934    MILLER  1806.00
8000    SMITH   3402.00
Database connection closed.
BUILD SUCCESSFUL (total time: 2 seconds)
```

Prepared statements are used for both methods to ensure SQL safety and improve performance. Prepared statements automatically handle parameterization and SQL injection prevention.

These implementations provide secure and efficient ways to execute both SELECT queries and other SQL statements while leveraging the benefits of prepared statements.