M.Sc. (Five Year Integrated) in Computer Science
(Artificial Intelligence & Data Science)

Java Database Connectivity

Submitted by

NANDHU KRISHNAN A

(80522014)

DEPARTMENT OF COMPUTER SCIENCE

COCHIN UNIVERSITY OF SCIENCE AND TECHNOLOGY (CUSAT)
KOCHI - 682022

1 Introduction to Java Database Connectivity (JDBC)

JDBC is a powerful Java-based API that enables applications to interact with relational databases. It provides a standardized way for Java programs to establish connections, execute SQL queries, and manage data within databases. JDBC abstracts the underlying complexities of database communication, allowing developers to focus on application logic.

1.1 Benefits of JDBC

JDBC offers several benefits to developers:

- Simplified Database Communication: JDBC abstracts low-level database communication, providing a consistent API for various database management systems.
- Portability: JDBC code is largely portable across different database vendors, reducing the effort required to switch databases.
- Security: JDBC helps manage database access and authentication securely.
- Dynamic Query Building: Java programs can dynamically build and execute SQL queries based on user input.

1.2 JDBC Architecture

The JDBC architecture comprises key components:

- DriverManager: Manages a list of database drivers and establishes connections.
- Connection: Represents a database connection, providing methods to execute SQL statements.

- Statement: Allows execution of SQL queries and updates.
- ResultSet: Holds query results for processing.

2 Types of JDBC Drivers

2.1 Type 1 Driver (JDBC-ODBC Bridge)

Type 1 driver acts as a bridge between Java and databases supporting ODBC (Open Database Connectivity).

- Bridges Java and databases via ODBC (Open Database Connectivity).
- Requires an ODBC driver for the target database system.
- Often considered outdated and not recommended for modern applications.
- Provides a rudimentary method for Java programs to connect to databases.
- Useful in legacy systems but lacks portability and security features.

```
import java.sql.*;

public class Type1DriverExample {
    public static void main (String [] args) {
        try {
            // Placeholder code for Type 1 Driver example
            String jdbcUrl = "jdbc:odbc:database_dsn";
            String username = "username";
            String password = "password";
            // Load the JDBC-ODBC Bridge driver
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            // Establish connection
            Connection connection = DriverManager.getConnection
               (jdbcUrl, username, password);
            // Execute SQL queries
            Statement statement = connection.createStatement();
            ResultSet resultSet = statement.executeQuery("
               SELECT * FROM employees");
            // Process query results
```

```
            while (resultSet.next()) {
                int employeeId = resultSet.getInt("id");
                String employeeName = resultSet.getString("name
                    ");
                System.out.println("Employee ID: " + employeeId
                    + ", Name: " + employeeName);
            }
            // Close  all the resources
            resultSet.close();
            statement.close();
            connection.close();
        } catch (Exception exception) {
            System.out.println(exception.getMessage());
        }
    }
}
```

2.2 Type 2 Driver (Native-API Driver)

Type 2 driver uses native libraries provided by the database vendor to connect to the database.

- Uses native libraries provided by the database vendor for communication.
- Offers better performance compared to Type 1 due to direct database access.
- Requires the specific database vendor's driver, making it less portable.
- Allows Java applications to establish connections with the database.
- Suitable for scenarios where direct access to the database is preferred.

```
import java.sql.*;

public class Type2DriverExample {
    public static void main (String [] args) {
        try {
            // Placeholder code for Type 2 Driver example
            String jdbcUrl = "jdbc:oracle:thin:@localhost:1521:
                database";
            String username = "username";
            String password = "password";
```

```
            // Load the Oracle JDBC driver (Type 2)
            Class.forName("oracle.jdbc.driver.OracleDriver");
            // Establish connection
            Connection connection = DriverManager.getConnection
               (jdbcUrl, username, password);
            // Execute SQL queries
            Statement statement = connection.createStatement();
            ResultSet resultSet = statement.executeQuery("
               SELECT * FROM employees");
            // Process query results
            while (resultSet.next()) {
                int employeeId = resultSet.getInt("id");
                String employeeName = resultSet.getString("name
                   ");
                System.out.println("Employee ID: " + employeeId
                    + ", Name: " + employeeName);
            }
            // Close all the resources
            resultSet.close();
            statement.close();
            connection.close();
        } catch (Exception exception) {
            System.out.println(exception.getMessage());
        }
    }
}
```

2.3 Type 3 Driver (Network Protocol Driver)

The Type 3 driver typically involves a middleware server to convert JDBC calls into a database-specific protocol.

- Utilizes a middleware server to convert JDBC calls into a database-specific protocol.
- Often used in distributed environments with a middle tier.
- Provides a level of database vendor independence.
- Requires middleware-specific driver classes for communication.

- Suitable for scenarios where middleware servers offer additional features like security and scalability.

```
import java.sql.*;

public class Type3DriverExample {
    public static void main(String[] args) {
        try {
            String jdbcUrl = "jdbc:netprotocol:database_server:
               port/database_name";
            String username = "username";
            String password = "password";

            // Load the Type 3 driver (middleware-specific
               driver)
            Class.forName("com.example.Type3Driver");

            // Establish connection
            Connection connection = DriverManager.getConnection
               (jdbcUrl, username, password);

            // Execute SQL queries
            Statement statement = connection.createStatement();
            ResultSet resultSet = statement.executeQuery("
               SELECT * FROM employees");

            // Process query results
            while (resultSet.next()) {
                int employeeId = resultSet.getInt("id");
                String employeeName = resultSet.getString("name
                   ");
                System.out.println("Employee ID: " + employeeId
                    + ", Name: " + employeeName);
            }

            // Close all resources
            resultSet.close();
            statement.close();
            connection.close();
        } catch (Exception exception) {
            System.out.println(exception.getMessage());
```

```
        }
    }
}
```

2.4 Type 4 Driver (Thin Driver)

The Type 4 driver communicates directly with the database server without the need for a middleware server.

- Communicates directly with the database server without middleware.
- Offers a pure Java solution for efficient database connectivity.
- Suitable for applications where direct, platform-independent access is essential.
- Requires only the inclusion of the specific JDBC driver class for the database.
- Widely used in modern Java applications for fast and reliable database access.

```
import java.sql.*;

public class Type4DriverExample {
    public static void main(String[] args) {
        try {
            String jdbcUrl = "jdbc:oracle:thin:@localhost:1521:
               database";
            String username = "username";
            String password = "password";

            // Load the Oracle Thin driver (Type 4)
            Class.forName("oracle.jdbc.driver.OracleDriver");

            // Establish connection
            Connection connection = DriverManager.getConnection
               (jdbcUrl, username, password);

            // Execute SQL queries
            Statement statement = connection.createStatement();
            ResultSet resultSet = statement.executeQuery("
               SELECT * FROM employees");

            // Process query results
```

```
            while (resultSet.next()) {
                int employeeId = resultSet.getInt("id");
                String employeeName = resultSet.getString("name
                    ");
                System.out.println("Employee ID: " + employeeId
                    + ", Name: " + employeeName);
            }

            // Close all resources
            resultSet.close();
            statement.close();
            connection.close();
        } catch (Exception exception) {
            System.out.println(exception.getMessage());
        }
    }
}
```

3 Using JDBC in Java

3.1 Step-by-Step Approach to Using JDBC

Effectively utilizing JDBC to interact with databases involves a systematic series of steps, ensuring seamless integration and effective data management within Java applications.

1. Load the Driver: Begin by employing the `Class.forName()` method to load the requisite JDBC driver class. This step registers the driver with the DriverManager, enabling its utilization for database communication.

2. Establish Connection: Next, establish a connection to the database using the `DriverManager.getConnection()` method. This facilitates a secure pathway for communication between the Java application and the database, ensuring data exchange.

3. Execute SQL Statements: Utilize either the `Statement` or `PreparedStatement` to execute SQL queries or updates on the connected database. These constructs serve as channels for transmitting queries, making it possible to retrieve, modify, or insert data.

4. Process Results: Upon executing a query, manage the query results with the `ResultSet` object. This object allows you to systematically navigate through the retrieved data, enabling efficient processing and analysis.

5. Close Resources: To ensure efficient resource management, it is crucial to close the acquired resources. Properly close the `ResultSet`, `Statement`, and `Connection` objects after their respective usage. This step safeguards against resource leakage and maintains optimal database connections.

3.2 Example JDBC Code

```
import java.sql.*;

public class JdbcSampleExample {
    public static void main (String [] args) {
        try {
            // Placeholder code for a sample JDBC code
            String jdbcUrl = "jdbc:mysql://localhost:3306/
               database";
            String username = "username";
            String password = "password";
            // Load the MySQL JDBC driver
            Class.forName("com.mysql.cj.jdbc.Driver");
            // Establish connection
            Connection connection = DriverManager.getConnection
               (jdbcUrl, username, password);
            // Execute SQL queries
            Statement statement = connection.createStatement();
            ResultSet resultSet = statement.executeQuery("
               SELECT * FROM customers");
            // Process query results
            while (resultSet.next()) {
                int customerId = resultSet.getInt("id");
                String customerName = resultSet.getString("name
                   ");
                System.out.println("Customer ID: " + customerId
                    + ", Name: " + customerName);
            }
            // Close all the resources
            resultSet.close();
            statement.close();
            connection.close();
        } catch (Exception exception) {
            System.out.println(exception.getMessage());
        }
    }
}
```

4 Conclusion

JDBC revolutionized database connectivity in Java applications. By providing a standardized API and connection management, JDBC made it easier to build powerful, data-driven applications. Understanding different types of JDBC drivers and their characteristics empowers developers to choose the right driver for their application's requirements.