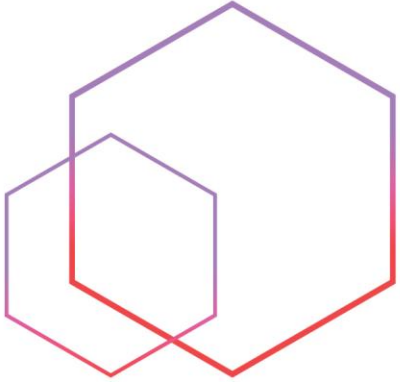


JDBC

JDBC and connecting to SQLite3

What is JDBC?

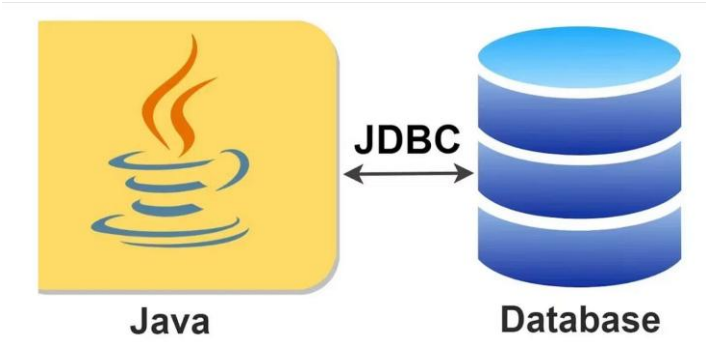
Definition

JDBC (Java Database Connectivity) is a Java API that provides a standard method for accessing and interacting with relational databases. It enables Java programs to execute SQL queries, retrieve results, and update data in a database.

Why JDBC?

- **Standardized API**: Works with various relational databases (e.g., MySQL, PostgreSQL, SQLite, Oracle).
- **Database Independence**: The same JDBC code can interact with multiple databases by simply changing the database driver.
- **Powerful Features**: Includes support for transactions, prepared statements, and callable statements.

Key Components of JDBC



1. **Driver Manager**:
    - Manages database drivers.
    - Establishes a connection to the database.
2. **Connection**:
    - Represents a session with the database.
    - Used to execute SQL queries and manage transactions.
3. **Statement**:
    - Used to execute SQL queries (e.g., `SELECT`, `INSERT`).
    - Types:
        - `Statement`: Basic SQL execution.
        - `PreparedStatement`: Precompiled SQL for parameterized queries.
        - `CallableStatement`: Used for executing stored procedures.
4. **ResultSet**:
    - Represents the results of an SQL query.
    - Provides methods to iterate through rows and access column values.

Steps to Connect to SQLite3 Using JDBC

Step 1: Add SQLite JDBC Driver

SQLite3 requires a JDBC driver to establish a connection.

1. **Download the SQLite JDBC Driver**:
    - Download the `.jar` file from the official website.
2. **Add the Driver to Your Project**:
    - Place the `.jar` file in your project directory.
    - Add the `.jar` file to your classpath in your IDE (e.g., Eclipse or IntelliJ).

Step 2: Establishing a Connection

Concepts:

- **Connection URL**: A JDBC URL specifies the database location and driver. For SQLite, the URL format is:

```
"jdbc:sqlite:<database_file_path>"
```

- **Driver Loading**: The JDBC driver must be loaded using `Class.forName` or automatically by the `DriverManager`.

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

public class SQLiteConnectionExample {
    public static void main(String[] args) {
        String url = "jdbc:sqlite:example.db"; // Database file path

        // Establishing a connection
        try (Connection conn = DriverManager.getConnection(url)) {
            if (conn != null) {
                System.out.println("Connected to the SQLite database.");
            }
        } catch (SQLException e) {
            System.out.println("Connection failed: " + e.getMessage());
        }
    }
}
```

Step 3: Creating a Table

- JDBC uses SQL statements to interact with the database.
- Tables are defined using the `CREATE TABLE` SQL command.

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

public class SQLiteCreateTableExample {
    public static void main(String[] args) {
        String url = "jdbc:sqlite:example.db";

        String sql = "CREATE TABLE IF NOT EXISTS students ("
                   + "id INTEGER PRIMARY KEY AUTOINCREMENT,"
                   + "name TEXT NOT NULL,"
                   + "age INTEGER,"
                   + "grade TEXT"
                   + ");";

        try {
            Connection conn = DriverManager.getConnection(url);
            Statement stmt = conn.createStatement();
            stmt.execute(sql);
            System.out.println("Table created successfully.");
        } catch (SQLException e) {
            System.out.println("Error creating table: " +
e.getMessage());
        }
    }
}
```

Step 4: Performing CRUD Operations

1. Insert Data:

```
String insertSQL = "INSERT INTO students (name, age, grade) VALUES (?, ?, ?)";
try (PreparedStatement pstmt = conn.prepareStatement(insertSQL)){
       pstmt.setString(1, "John Doe");
       pstmt.setInt(2, 20);
       pstmt.setString(3, "A");
       pstmt.executeUpdate();
       System.out.println("Record inserted successfully.");
}
```

2. Read Data - We are using **ResultSet** Class here:

```
String selectSQL = "SELECT id, name, age, grade FROM students WHERE id = ?";
try (PreparedStatement pstmt = conn.prepareStatement(selectSQL)) {
   pstmt.setInt(1, 1); // Assuming ID = 1 exists
   try (ResultSet rs = pstmt.executeQuery()) {
       if (rs.next()) {
           System.out.println("Record found:");
           System.out.println("ID: " + rs.getInt("id"));
           System.out.println("Name: " + rs.getString("name"));
           System.out.println("Age: " + rs.getInt("age"));
           System.out.println("Grade: " + rs.getString("grade"));
       } else {
           System.out.println("No record found with ID = 1.");
       }
   }
}
```

Popular methods:

- **next()**
  - Moves the cursor to the next row in the result set.
  - Returns `true` if a row exists; otherwise, `false`.
  - Always call `next()` before accessing data in the `ResultSet`.
- **Retrieving Column Values**
  - Use methods like `getString()`, `getInt()`, `getDouble()`, etc., to retrieve values from the current row.
  - Column values can be accessed by:
    - **Index**: Starting from 1 (e.g., `rs.getString(1)`).
    - **Name**: Using the column name (e.g., `rs.getString("name")`)
- **getMetaData()**
  - Returns a `ResultSetMetaData` object, which provides information about the columns in the result set.
  - Useful for dynamic queries when you don't know column names in advance.
  - Example:

```
ResultSetMetaData metaData = rs.getMetaData();
int columnCount = metaData.getColumnCount();
for (int i = 1; i <= columnCount; i++) {
    System.out.println("Column Name: " + metaData.getColumnName(i));
}
```

- **Navigating Through Rows()**
  - **`first()`: Moves the cursor to the first row.**
  - **`last()`: Moves the cursor to the last row.**
  - **`previous()`: Moves the cursor to the previous row.**
  - **`absolute(int row)`: Moves the cursor to a specific row.**
  - **`relative(int rows)`: Moves the cursor relative to its current position.**

3. Update Data:

```
String updateSQL = "UPDATE students SET name = ?, age = ?, grade = ? WHERE id =
?";
try (PreparedStatement pstmt = conn.prepareStatement(updateSQL)) {
    pstmt.setString(1, "Jane Smith");
    pstmt.setInt(2, 22);
    pstmt.setString(3, "B");
    pstmt.setInt(4, 1); // Assuming ID = 1 exists
    pstmt.executeUpdate();
    System.out.println("Record updated successfully.");
}
```

4. Delete Data:

```
String deleteSQL = "DELETE FROM students WHERE id = ?";
try (PreparedStatement pstmt = conn.prepareStatement(deleteSQL)) {
    pstmt.setInt(1, 1); // Assuming ID = 1 exists
    pstmt.executeUpdate();
    System.out.println("Record deleted successfully.");
}
```

Important Concepts in JDBC

1. **Prepared Statements**:
    - Prevent SQL injection.
    - Precompile SQL queries for better performance.
2. **Transactions**:
    - Use `conn.setAutoCommit(false)` to manage transactions manually.
    - Commit or roll back transactions using `conn.commit()` or `conn.rollback()`.
3. **Error Handling**:
    - Always close resources (e.g., `Connection`, `Statement`, `ResultSet`) in a `try-with-resources` block.
    - Use `SQLException` for error diagnosis.

Conclusion

This lesson covers the basics of JDBC and how to connect to SQLite3 using it. By understanding JDBC concepts and practicing CRUD operations, you'll gain the foundational skills required to work with relational databases in Java.