**Q.What is JDBC?**

**JDBC (Java Database Connectivity)** is a Java API that enables Java applications to interact with a wide range of databases. It provides a standard interface for connecting to relational databases, executing SQL queries, and retrieving results. JDBC is part of the Java Standard Edition platform and allows developers to write database-independent code, meaning that the same code can work with different database systems (e.g., MySQL, Oracle, PostgreSQL) with minimal changes.

**Why Use JDBC?**

1. **Database Access**: JDBC provides a mechanism for Java applications to interact with databases, allowing them to execute SQL queries, retrieve data, and update records in the database.
2. **Database Independence**: JDBC abstracts the database-specific details and provides a uniform interface, making it easier to switch between different databases without significant code changes.
3. **Integration with Java Applications**: JDBC seamlessly integrates with Java applications, enabling developers to incorporate database functionality within their Java programs.
4. **Flexibility**: JDBC allows for executing both static SQL statements and dynamic SQL, providing flexibility in how queries are constructed and executed.
5. **Control Over Database Operations**: JDBC provides fine-grained control over database transactions, connections, and result sets, allowing developers to optimize and manage database interactions effectively.

**Advantages of JDBC**

1. **Platform Independence**: JDBC is part of the Java platform, making it platform-independent. Applications written using JDBC can run on any platform that supports Java.
2. **Database Independence**: JDBC provides a consistent API for connecting to different types of databases, reducing the dependency on a specific database vendor.
3. **Robust Error Handling**: JDBC provides detailed error information, making it easier to diagnose and handle database-related issues.
4. **Integration with Java Ecosystem**: JDBC can easily integrate with other Java technologies like Java EE (Enterprise Edition), Spring Framework, and Hibernate, enhancing the functionality of enterprise applications.
5. **Support for Transactions**: JDBC supports database transactions, allowing developers to ensure data integrity and consistency.
6. **Extensibility**: JDBC drivers are available for a wide range of databases, and custom drivers can be developed if needed.
7. **Mature and Well-Supported**: JDBC has been around for a long time and is well-documented, with a large community of developers and a wealth of resources available.

**Disadvantages of JDBC**

1. **Verbose Code**: JDBC often requires a lot of boilerplate code for tasks like opening connections, executing queries, and closing resources. This can make the code more verbose and harder to maintain.
2. **Manual Resource Management**: Developers need to explicitly manage database resources like connections, statements, and result sets, which can lead to potential resource leaks if not handled properly.
3. **Error-Prone**: The need for manual SQL string construction and the use of raw SQL queries can lead to syntax errors, SQL injection vulnerabilities, and other issues.
4. **Limited Object-Relational Mapping (ORM) Support**: JDBC operates at a lower level compared to ORM frameworks like Hibernate. It does not provide direct support for mapping Java objects to database tables, making it less suitable for complex object-relational mapping.
5. **Performance Overhead**: While JDBC is generally efficient, the abstraction layer it provides can introduce some performance overhead compared to using native database APIs directly.
6. **Complexity in Large Applications**: In large-scale applications with complex database interactions, managing JDBC code can become cumbersome and complex, especially when dealing with transactions, connection pooling, and error handling.

**Conclusion**

JDBC is a powerful and flexible API for database interaction in Java, offering many advantages such as platform and database independence, robust error handling, and transaction support. However, it also comes with some challenges, including verbose code, manual resource management, and complexity in large applications. Understanding these trade-offs is essential when deciding whether to use JDBC directly or opt for higher-level frameworks like Hibernate or Spring Data that build on top of JDBC.

### Q. What is a JDBC Driver?

A **JDBC driver** is a software component that allows Java applications to connect to and interact with databases. It acts as an intermediary, translating Java calls into database-specific commands and vice versa, enabling seamless communication between a Java application and a database.

### Types of JDBC Drivers

JDBC drivers are classified into four types based on their architecture and mode of communication with the database:

#### 1. Type 1: JDBC-ODBC Bridge Driver

* **Description**:
  + The Type 1 driver uses the ODBC (Open Database Connectivity) driver to connect to the database. The JDBC-ODBC bridge translates JDBC calls into ODBC calls, which are then processed by the ODBC driver to interact with the database.
* **Advantages**:
  + **Simplicity**: Easy to use and set up, especially in environments where ODBC drivers are already available.
  + **Wide Database Support**: Can connect to any database with an ODBC driver, offering broad compatibility.
* **Disadvantages**:
  + **Performance Overhead**: The bridge adds an extra layer of translation, leading to slower performance compared to other driver types.
  + **Platform Dependency**: Requires ODBC to be installed on the client machine, making it platform-dependent.
  + **Deprecated**: The JDBC-ODBC bridge is no longer supported in newer versions of Java, making it unsuitable for modern applications.
* **Use Case**:
  + Best suited for legacy systems where ODBC is the primary method of database access. Not recommended for modern Java applications.

**Example**: sun.jdbc.odbc.JdbcOdbcDriver (No longer used in modern Java versions).

#### 2. Type 2: Native-API Driver (Partially Java Driver)

* **Description**:
  + The Type 2 driver converts JDBC calls into database-specific native API calls. It uses native libraries provided by the database vendor, which are typically written in C or C++.
* **Advantages**:
  + **Better Performance**: Offers better performance than Type 1 due to direct interaction with the database’s native API.
  + **Database-Specific Features**: Can leverage the full functionality of the database’s native API, providing access to vendor-specific features.
* **Disadvantages**:
  + **Platform Dependency**: Requires native database libraries, which are platform-specific, making the application less portable.
  + **Complex Deployment**: The need to install native libraries on the client machine can complicate deployment.
* **Use Case**:
  + Ideal for high-performance applications where the database platform is fixed and the native API offers significant advantages.

**Example**: Oracle's OCI (Oracle Call Interface) driver.

#### 3. Type 3: Network Protocol Driver (Middleware Driver)

* **Description**:
  + The Type 3 driver uses a middleware server to convert JDBC calls into a database-independent network protocol. The middleware then translates these into database-specific calls.
* **Advantages**:
  + **Database and Platform Independence**: Since the driver communicates with a middleware server, it can connect to multiple types of databases without needing database-specific drivers.
  + **Flexibility**: Suitable for multi-database and multi-platform environments.
* **Disadvantages**:
  + **Middleware Requirement**: Requires an additional middleware server, which adds complexity and potential points of failure.
  + **Network Overhead**: The extra network communication between the application, middleware, and database can impact performance.
* **Use Case**:
  + Commonly used in enterprise applications that need to connect to multiple databases or in environments where database independence is crucial.

**Example**: IBM DB2 net driver.

#### 4. Type 4: Thin Driver (Pure Java Driver)

* **Description**:
  + The Type 4 driver is written entirely in Java and converts JDBC calls directly into the database-specific protocol. It communicates directly with the database over the network, eliminating the need for any native libraries or middleware.
* **Advantages**:
  + **Platform Independence**: Being pure Java, it is platform-independent and can run on any system with a Java Virtual Machine (JVM).
  + **Performance**: Typically the fastest JDBC driver since it communicates directly with the database.
  + **Simplified Deployment**: No need for native libraries or middleware, making deployment easier.
* **Disadvantages**:
  + **Database-Specific**: Requires a separate driver for each database type, which might limit flexibility if the application needs to support multiple databases.
  + **Limited Vendor Features**: May not support all database-specific features available through native APIs.
* **Use Case**:
  + The most common and recommended choice for modern applications, especially when targeting a specific database. Ideal for web and enterprise applications where performance and portability are key concerns.

**Example**: MySQL Connector/J (com.mysql.cj.jdbc.Driver), Oracle JDBC Thin driver (oracle.jdbc.driver.OracleDriver).

### Summary

* **Driver**: A software component that translates Java JDBC calls into database-specific calls, enabling Java applications to interact with different databases.
* **MySQL Connector/J**: An example of a JDBC driver (specifically a Type 4 driver) that allows Java applications to connect to a MySQL database.
* **Driver's Role**: The driver manages the connection, sends SQL statements to the database, and retrieves results.

The MySQL driver, or any other JDBC driver, is essential for enabling Java applications to interact with databases in a standardized and database-independent manner.

**Q. Steps to connect with database using JDBC?**

**1. Load the JDBC Driver**

Loading the JDBC driver is often not required in newer versions of Java (JDBC 4.0 and later), but it’s good practice to know how to do it.

try {

Class.forName("com.mysql.cj.jdbc.Driver"); // For MySQL

} catch (ClassNotFoundException e) {

e.printStackTrace();

}

**2. Establish a Connection**

Use the DriverManager class to establish a connection to the database.

String url = "jdbc:mysql://localhost:3306/mydatabase"; // Replace with your database URL

String user = "root"; // Replace with your database username

String password = "password"; // Replace with your database password

Connection connection = null;

try {

connection = DriverManager.getConnection(url, user, password);

System.out.println("Connected to the database!");

} catch (SQLException e) {

e.printStackTrace();

}

**3. Create a Statement**

Create a Statement object to execute SQL queries.

Statement statement = null;

try {

statement = connection.createStatement();

} catch (SQLException e) {

e.printStackTrace();

}

**4. Execute a SQL Query**

Execute a SQL query using the Statement object. For example, retrieving all records from an employees table:

String query = "SELECT \* FROM employees";

ResultSet resultSet = null;

try {

resultSet = statement.executeQuery(query);

} catch (SQLException e) {

e.printStackTrace();

}

**5. Process the ResultSet**

Process the results returned by the query.

try {

while (resultSet.next()) {

int id = resultSet.getInt("id");

String name = resultSet.getString("name");

String department = resultSet.getString("department");

System.out.println("ID: " + id + ", Name: " + name + ", Department: " + department);

}

} catch (SQLException e) {

e.printStackTrace();

}

**6. Close the Resources**

Always close the ResultSet, Statement, and Connection objects to free up resources.

try {

if (resultSet != null) resultSet.close();

if (statement != null) statement.close();

if (connection != null) connection.close();

System.out.println("Resources closed!");

} catch (SQLException e) {

e.printStackTrace();

}

**Full Simple Example**

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

public class SimpleJDBCExample {

public static void main(String[] args) {

String url = "jdbc:mysql://localhost:3306/mydatabase"; // Database URL

String user = "root"; // Database username

String password = "password"; // Database password

Connection connection = null;

Statement statement = null;

ResultSet resultSet = null;

try {

// Establish connection

connection = DriverManager.getConnection(url, user, password);

System.out.println("Connected to the database!");

// Create a statement

statement = connection.createStatement();

// Execute a query

String query = "SELECT \* FROM employees";

resultSet = statement.executeQuery(query);

// Process the result set

while (resultSet.next()) {

int id = resultSet.getInt("id");

String name = resultSet.getString("name");

String department = resultSet.getString("department");

System.out.println("ID: " + id + ", Name: " + name + ", Department: " + department);

}

} catch (SQLException e) {

e.printStackTrace();

} finally {

// Close resources

try {

if (resultSet != null) resultSet.close();

if (statement != null) statement.close();

if (connection != null) connection.close();

System.out.println("Resources closed!");

} catch (SQLException e) {

e.printStackTrace();

}

}

}

}

**Steps Summary:**

1. **Load the JDBC Driver** (Optional for JDBC 4.0 and later).
2. **Establish a Connection** using DriverManager.
3. **Create a Statement** object.
4. **Execute a Query** using the Statement.
5. **Process the ResultSet** to retrieve data.
6. **Close the Resources** to prevent resource leaks.

**Q. What is DriverManager Class and its method?**

The DriverManager class in Java is part of the java.sql package and plays a critical role in managing a list of database drivers and establishing connections to databases. It is a crucial part of the JDBC API, which allows Java applications to interact with various databases in a platform-independent manner.

### Overview of DriverManager

* **Role**: The DriverManager class acts as an interface between a user and the drivers for a database. When a connection to a database is requested, DriverManager tries to find an appropriate driver from the set of registered drivers and uses it to establish a connection.

### Key Responsibilities:

1. **Loading JDBC Drivers**: The DriverManager class is responsible for loading and registering JDBC drivers. This is typically done automatically when a JDBC driver class is loaded, but it can also be done manually using the DriverManager.registerDriver() method.
2. **Establishing Connections**: The DriverManager class is used to establish a connection to a specified database URL. It does this by going through the list of registered drivers and finding one that can handle the connection request.
3. **Managing Drivers**: DriverManager maintains a list of all the drivers that have been loaded. These drivers are consulted whenever a new connection is requested.
4. **Logging**: DriverManager also provides simple mechanisms for tracing and logging. It allows you to set up a logging stream to see the details of the JDBC operations being performed.

**Key Methods in DriverManager Class:**

1. **public static synchronized void registerDriver(Driver driver):**

is used to register the given driver with DriverManager. No action is performed by the method when the given driver is already registered.

1. **public static synchronized void deregisterDriver(Driver driver):**

is used to deregister the given driver (drop the driver from the list) with DriverManager. If the given driver has been removed from the list, then no action is performed by the method.

1. **public static Connection getConnection(String url) throws SQLException:**

is used to establish the connection with the specified url. The SQLException is thrown when the corresponding Driver class of the given database is not registered with the DriverManager.

1. **public static Connection getConnection(String url,String userName,String password) throws SQLException:**

is used to establish the connection with the specified url, username, and password. The SQLException is thrown when the corresponding Driver class of the given database is not registered with the DriverManager.

1. **public static Driver getDriver(String url):**

Those drivers that understand the mentioned URL (present in the parameter of the method) are returned by this method provided those drivers are mentioned in the list of registered drivers.

1. **pubic static int getLoginTimeout():**

The duration of time a driver is allowed to wait in order to establish a connection with the database is returned by this method.

1. **pubic static void setLoginTimeout(int sec):**

The method provides the time in seconds. sec mentioned in the parameter is the maximum time that a driver is allowed to wait in order to establish a connection with the database. If 0 is passed in the parameter of this method, the driver will have to wait infinitely while trying to establish the connection with the database.

1. **public static Connection getConnection(String URL, Properties prop) throws SQLException:**

A connection object is returned by this method after creating a connection to the database present at the mentioned URL, which is the first parameter of this method. The second parameter, which is "prop", fetches the authentication details of the database (username and password.). Similar to the other variation of the getConnection() method, this method also throws the SQLException, when the corresponding Driver class of the given database is not registered with the DriverManager.

**Q. What is Connection Interface and its methods?**

The Connection interface in Java is part of the java.sql package and is a core component of the JDBC API. It represents an active connection to a database, allowing Java applications to interact with the database by executing SQL queries, updating data, and managing transactions.

### Key Responsibilities of the Connection Interface

1. **Managing Database Sessions**: The Connection interface represents a session with a specific database. This session allows the application to send SQL commands and receive results.
2. **Executing SQL Statements**: The Connection interface provides methods for creating objects that execute SQL queries, such as Statement, PreparedStatement, and CallableStatement.
3. **Handling Transactions**: The Connection interface allows for transaction management, enabling the application to commit or rollback changes made during a session.
4. **Managing Database Metadata**: Through a Connection object, you can retrieve metadata about the database, such as its structure, supported SQL types, and more.
5. **Managing Connection Properties**: The Connection interface allows you to set various connection properties, such as auto-commit mode, transaction isolation level, and more.

**Commonly used methods of Connection Interface:**

**1) public Statement createStatement():** creates a statement object that can be used to execute SQL queries.

**2) public Statement createStatement(int resultSetType,int resultSetConcurrency):** Creates a Statement object that will generate ResultSet objects with the given type and concurrency.

**3) public void setAutoCommit(boolean status):** is used to set the commit status. By default, it is true.

**4) public void commit():** saves the changes made since the previous commit/rollback is permanent.

**5) public void rollback():** Drops all changes made since the previous commit/rollback.

**6) public void close():** closes the connection and Releases a JDBC resources immediately.

**Q. What is Statement Interface and its methods?**

The Statement interface in Java is part of the java.sql package and is used to execute SQL queries against a database. It provides methods to send SQL commands (such as SELECT, INSERT, UPDATE, DELETE) and retrieve results from the database.

### Key Responsibilities of the Statement Interface

1. **Executing SQL Queries**: The primary purpose of the Statement interface is to send SQL queries to the database.
2. **Retrieving Results**: It allows retrieval of the results from SELECT queries and handles the update counts for INSERT, UPDATE, and DELETE statements.
3. **Controlling Execution**: It provides control over the execution of queries, such as setting timeouts, handling large data sets, and managing cursor movement.

**Methods of Statement Interface:**

### 1. execute(String sql)

* **Purpose**:
  + This method can execute any type of SQL statement: SELECT, INSERT, UPDATE, DELETE, and even DDL (Data Definition Language) statements like CREATE TABLE and DROP TABLE.
  + It is a more general-purpose method compared to executeUpdate() and executeQuery().
* **Return Type**:
  + Returns a boolean value:
    - true if the result is a ResultSet (typically from a SELECT query).
    - false if the result is an update count or if there is no result (for INSERT, UPDATE, DELETE, or DDL statements).
* **Use Cases**:
  + When you do not know the type of SQL statement being executed or when you need to execute multiple types of SQL statements using the same method.
  + Example:

Statement stmt = connection.createStatement();

boolean hasResultSet = stmt.execute("SELECT \* FROM employees");

if (hasResultSet) {

ResultSet rs = stmt.getResultSet();

// Process the result set

} else {

int updateCount = stmt.getUpdateCount();

// Handle update count

}

### 2. executeUpdate(String sql)

* **Purpose**:
  + This method is used to execute SQL statements that modify the database (i.e., INSERT, UPDATE, DELETE) or DDL statements (like CREATE TABLE or DROP TABLE).
* **Return Type**:
  + Returns an integer representing the number of rows affected by the SQL statement.
  + If the SQL statement is a DDL command (like CREATE TABLE), it may return 0 as no rows are directly affected.
* **Use Cases**:
  + When you need to execute a SQL statement that changes the data in the database or modifies the database schema.
  + Example:

Statement stmt = connection.createStatement();

int rowsAffected = stmt.executeUpdate("UPDATE employees SET salary = 50000 WHERE id = 101");

System.out.println("Rows affected: " + rowsAffected);

### 3. executeQuery(String sql)

* **Purpose**:
  + This method is specifically used to execute SQL SELECT queries, which retrieve data from the database.
* **Return Type**:
  + Returns a ResultSet object containing the data produced by the query.
* **Use Cases**:
  + When you need to execute a SQL SELECT query and process the result set returned by the query.
  + Example:

Statement stmt = connection.createStatement();

ResultSet rs = stmt.executeQuery("SELECT \* FROM employees");

while (rs.next()) {

System.out.println(rs.getInt("id") + ", " + rs.getString("name"));

}

### Choosing the Right Method:

* **Use executeQuery()** when you expect a ResultSet, typically from a SELECT statement.
* **Use executeUpdate()** when you perform database modifications and want to know the number of affected rows.
* **Use execute()** when you need flexibility to handle both query and update operations, or when executing SQL commands whose result type you don't know in advance.

**Q. ResultSet Interface and its methods?**

The ResultSet interface in Java is part of the java.sql package and is used to store and manipulate the results returned by executing SQL queries, specifically SELECT queries. When a SELECT query is executed using the Statement interface, a ResultSet object is returned, which contains the data retrieved from the database.

### Key Responsibilities of the ResultSet Interface

1. **Navigating the Result Set**: The ResultSet interface allows you to navigate through the rows of data retrieved from the database.
2. **Accessing Data**: It provides methods to access and retrieve data from the columns of the current row in the result set.
3. **Updating Data**: Depending on the ResultSet type, it may also support updates to the data in the database.

### Types of ResultSet

There are different types of ResultSet objects, depending on their ability to scroll through data and whether they are updatable:

1. **TYPE\_FORWARD\_ONLY**: The cursor can only move forward.
2. **TYPE\_SCROLL\_INSENSITIVE**: The cursor can move forward and backward, but the ResultSet is insensitive to changes made to the underlying data by others.
3. **TYPE\_SCROLL\_SENSITIVE**: The cursor can move forward and backward, and the ResultSet is sensitive to changes made to the underlying data by others.

### Commonly Used Methods in the ResultSet Interface

Here are some of the most commonly used methods provided by the ResultSet interface:

#### 1. **Navigating the Result Set**

* **next()**
  + Moves the cursor forward one row from its current position.
  + Returns true if the new current row is valid, otherwise false if there are no more rows.

boolean next() throws SQLException;

**Example**:

ResultSet rs = stmt.executeQuery("SELECT \* FROM employees");

while (rs.next()) {

System.out.println(rs.getInt("id") + ", " + rs.getString("name"));

}

* **previous()**
  + Moves the cursor to the previous row.
  + Returns true if the new current row is valid; false if it is off the result set.

boolean previous() throws SQLException;

**Example**:

ResultSet rs = stmt.executeQuery("SELECT \* FROM employees");

while (rs.previous()) {

System.out.println(rs.getInt("id") + ", " + rs.getString("name"));

}

* **first()**
  + Moves the cursor to the first row in the ResultSet.

boolean first() throws SQLException;

* **last()**
  + Moves the cursor to the last row in the ResultSet.

boolean last() throws SQLException;

* **absolute(int row)**
  + Moves the cursor to the specified row number in the ResultSet. The first row is 1, the second is 2, and so on. If the specified row is negative, the cursor moves to the row number relative to the end of the result set.

boolean absolute(int row) throws SQLException;

* **beforeFirst()**
  + Moves the cursor to a position before the first row. After calling this method, you must call next() to move to the first row.

void beforeFirst() throws SQLException;

* **afterLast()**
  + Moves the cursor to a position after the last row. After calling this method, you must call previous() to move to the last row.

void afterLast() throws SQLException;

#### 2. **Accessing Data**

* **getInt(int columnIndex) / getInt(String columnLabel)**
  + Retrieves the value of the specified column as an int.

int getInt(int columnIndex) throws SQLException;

int getInt(String columnLabel) throws SQLException;

**Example**:

int id = rs.getInt("id");

* **getString(int columnIndex) / getString(String columnLabel)**
  + Retrieves the value of the specified column as a String.

String getString(int columnIndex) throws SQLException;

String getString(String columnLabel) throws SQLException;

**Example**:

String name = rs.getString("name");

* **getDouble(int columnIndex) / getDouble(String columnLabel)**
  + Retrieves the value of the specified column as a double.

double getDouble(int columnIndex) throws SQLException;

double getDouble(String columnLabel) throws SQLException;

**Example**:

double salary = rs.getDouble("salary");

* **getDate(int columnIndex) / getDate(String columnLabel)**
  + Retrieves the value of the specified column as a java.sql.Date.

java.sql.Date getDate(int columnIndex) throws SQLException;

java.sql.Date getDate(String columnLabel) throws SQLException;

* **getObject(int columnIndex) / getObject(String columnLabel)**
  + Retrieves the value of the specified column as an Object, which can be cast to the appropriate data type.

Object getObject(int columnIndex) throws SQLException;

Object getObject(String columnLabel) throws SQLException;

**Example**:

Object obj = rs.getObject("column\_name");

#### 3. **Updating Data**

If the ResultSet is updatable, you can use these methods to update data:

* **updateInt(int columnIndex, int x) / updateInt(String columnLabel, int x)**
  + Updates the specified column with an int value.

void updateInt(int columnIndex, int x) throws SQLException;

void updateInt(String columnLabel, int x) throws SQLException;

* **updateString(int columnIndex, String x) / updateString(String columnLabel, String x)**
  + Updates the specified column with a String value.

void updateString(int columnIndex, String x) throws SQLException;

void updateString(String columnLabel, String x) throws SQLException;

* **updateRow()**
  + Updates the current row in the ResultSet to reflect the modifications made using updateXXX methods.

void updateRow() throws SQLException;

* **insertRow()**
  + Inserts a new row into the database. This method must be used after calling moveToInsertRow() to move the cursor to the insert row.

void insertRow() throws SQLException;

* **deleteRow()**
  + Deletes the current row from the ResultSet and the underlying database.

void deleteRow() throws SQLException;

#### 4. **Other Useful Methods**

* **close()**
  + Closes the ResultSet object and releases its resources.

void close() throws SQLException;

* **wasNull()**
  + Checks if the last column read had a SQL NULL value.

boolean wasNull() throws SQLException;

* **getMetaData()**
  + Retrieves the ResultSetMetaData object that contains information about the ResultSetMetaData getMetaData() throws SQLException;

**Example**:

ResultSetMetaData metaData = rs.getMetaData();

int columnCount = metaData.getColumnCount();

### Q. PreparedStatement in Java

PreparedStatement is a powerful feature of JDBC (Java Database Connectivity) that helps in executing parameterized SQL queries. It provides better security, performance, and flexibility compared to regular Statement objects. Here's a detailed guide on PreparedStatement, including its usage, benefits, and code examples.

#### **Benefits of PreparedStatement**

1. **Security**: Helps prevent SQL injection attacks by using parameterized queries.
2. **Performance**: Improves performance by allowing the database to cache the SQL query execution plan.
3. **Readability**: Makes the code cleaner and easier to maintain.

#### **Key Methods of PreparedStatement**

* void setInt(int parameterIndex, int x): Sets the value of the specified parameter with an integer.
* void setString(int parameterIndex, String x): Sets the value of the specified parameter with a string.
* void setDouble(int parameterIndex, double x): Sets the value of the specified parameter with a double.
* ResultSet executeQuery(): Executes the query and returns the result set.
* int executeUpdate(): Executes the update and returns the number of rows affected.

#### **Basic Syntax**

String sql = "SELECT \* FROM employees WHERE department = ? AND salary > ?";

PreparedStatement pstmt = connection.prepareStatement(sql);

pstmt.setString(1, "Engineering");

pstmt.setDouble(2, 50000.00);

ResultSet rs = pstmt.executeQuery();

### ****Detailed Example****

#### **1. Setup the Database**

Ensure you have a database and a table to work with. For this example, let's assume you have an employees table:

CREATE TABLE employees (

id INT PRIMARY KEY,

name VARCHAR(100),

department VARCHAR(50),

salary DOUBLE

);

#### **2. JDBC Connection Setup**

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

public class DatabaseConnection {

private static final String URL = "jdbc:mysql://localhost:3306/your\_database";

private static final String USER = "your\_username";

private static final String PASSWORD = "your\_password";

public static Connection getConnection() throws SQLException {

return DriverManager.getConnection(URL, USER, PASSWORD);

}

}

#### **3. Using PreparedStatement for Querying Data**

import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

public class EmployeeDAO {

public void getEmployeesByDepartmentAndSalary(String department, double salary) {

String sql = "SELECT \* FROM employees WHERE department = ? AND salary > ?";

try (Connection conn = DatabaseConnection.getConnection();

PreparedStatement pstmt = conn.prepareStatement(sql)) {

pstmt.setString(1, department);

pstmt.setDouble(2, salary);

try (ResultSet rs = pstmt.executeQuery()) {

while (rs.next()) {

int id = rs.getInt("id");

String name = rs.getString("name");

double empSalary = rs.getDouble("salary");

System.out.println("ID: " + id + ", Name: " + name + ", Salary: " + empSalary);

}

}

} catch (SQLException e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

EmployeeDAO dao = new EmployeeDAO();

dao.getEmployeesByDepartmentAndSalary("Engineering", 50000.00);

}

}

#### **4. Using PreparedStatement for Inserting Data**

import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.SQLException;

public class EmployeeDAO {

public void insertEmployee(int id, String name, String department, double salary) {

String sql = "INSERT INTO employees (id, name, department, salary) VALUES (?, ?, ?, ?)";

try (Connection conn = DatabaseConnection.getConnection();

PreparedStatement pstmt = conn.prepareStatement(sql)) {

pstmt.setInt(1, id);

pstmt.setString(2, name);

pstmt.setString(3, department);

pstmt.setDouble(4, salary);

int rowsAffected = pstmt.executeUpdate();

System.out.println("Rows affected: " + rowsAffected);

} catch (SQLException e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

EmployeeDAO dao = new EmployeeDAO();

dao.insertEmployee(1, "John Doe", "Engineering", 60000.00);

}

}

#### **5. Using PreparedStatement for Updating Data**

import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.SQLException;

public class EmployeeDAO {

public void updateEmployeeSalary(int id, double newSalary) {

String sql = "UPDATE employees SET salary = ? WHERE id = ?";

try (Connection conn = DatabaseConnection.getConnection();

PreparedStatement pstmt = conn.prepareStatement(sql)) {

pstmt.setDouble(1, newSalary);

pstmt.setInt(2, id);

int rowsAffected = pstmt.executeUpdate();

System.out.println("Rows affected: " + rowsAffected);

} catch (SQLException e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

EmployeeDAO dao = new EmployeeDAO();

dao.updateEmployeeSalary(1, 65000.00);

}

}

#### **6. Using PreparedStatement for Deleting Data**

import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.SQLException;

public class EmployeeDAO {

public void deleteEmployee(int id) {

String sql = "DELETE FROM employees WHERE id = ?";

try (Connection conn = DatabaseConnection.getConnection();

PreparedStatement pstmt = conn.prepareStatement(sql)) {

pstmt.setInt(1, id);

int rowsAffected = pstmt.executeUpdate();

System.out.println("Rows affected: " + rowsAffected);

} catch (SQLException e) {

e.printStackTrace();

}

}

public static void main(String[] args) {

EmployeeDAO dao = new EmployeeDAO();

dao.deleteEmployee(1);

}

}

### ****Conclusion****

PreparedStatement provides a safe and efficient way to interact with databases. It helps prevent SQL injection, improves performance, and makes your code more readable. By using PreparedStatement, you can execute various types of SQL queries (SELECT, INSERT, UPDATE, DELETE) with parameterized inputs, ensuring robust and secure database operations.

**Q. CallableInterface complete notes with code below**

The CallableStatement interface in JDBC is used to execute stored procedures in a database. Stored procedures are precompiled SQL statements that can be executed with parameters, and CallableStatement provides a way to call these procedures from Java code.

### ****1. Overview****

* **Definition**: CallableStatement is a sub-interface of PreparedStatement and is used specifically for calling stored procedures in a database.
* **Purpose**: Allows you to execute stored procedures that may have input parameters, output parameters, or both.

### ****2. Key Methods of**** CallableStatement

1. **Setting Input Parameters:**
   * void setBoolean(int parameterIndex, boolean x)
   * void setByte(int parameterIndex, byte x)
   * void setShort(int parameterIndex, short x)
   * void setInt(int parameterIndex, int x)
   * void setLong(int parameterIndex, long x)
   * void setFloat(int parameterIndex, float x)
   * void setDouble(int parameterIndex, double x)
   * void setString(int parameterIndex, String x)
   * void setDate(int parameterIndex, Date x)
   * void setTime(int parameterIndex, Time x)
   * void setTimestamp(int parameterIndex, Timestamp x)
   * void setObject(int parameterIndex, Object x)
2. **Registering Output Parameters:**
   * void registerOutParameter(int parameterIndex, int sqlType)
   * void registerOutParameter(int parameterIndex, int sqlType, String typeName)
3. **Retrieving Output Parameters:**
   * boolean getBoolean(int parameterIndex)
   * byte getByte(int parameterIndex)
   * short getShort(int parameterIndex)
   * int getInt(int parameterIndex)
   * long getLong(int parameterIndex)
   * float getFloat(int parameterIndex)
   * double getDouble(int parameterIndex)
   * String getString(int parameterIndex)
   * Date getDate(int parameterIndex)
   * Time getTime(int parameterIndex)
   * Timestamp getTimestamp(int parameterIndex)
   * Object getObject(int parameterIndex)
4. **Executing the Stored Procedure:**
   * boolean execute()
   * ResultSet executeQuery()
   * int executeUpdate()

### ****3. Steps to Use**** CallableStatement

1. **Establish a Connection:**
   * Use DriverManager.getConnection() to connect to the database.
2. **Create a CallableStatement Object:**
   * Use Connection.prepareCall(String sql) to create a CallableStatement.
3. **Set Input Parameters:**
   * Use setter methods to set input parameters for the stored procedure.
4. **Register Output Parameters:**
   * Register output parameters using registerOutParameter().
5. **Execute the Stored Procedure:**
   * Use execute(), executeQuery(), or executeUpdate() to run the procedure.
6. **Retrieve Output Parameters:**
   * Use getter methods to retrieve the values of output parameters.
7. **Handle Exceptions and Cleanup:**
   * Use try-catch blocks to handle SQL exceptions and ensure proper resource cleanup.
8. **Stored Procedure for Inserting Data**

**DELIMITER //**

CREATE PROCEDURE insertData(

IN p\_id INT,

IN p\_name VARCHAR(255),

IN p\_city VARCHAR(255),

IN p\_college VARCHAR(255),

IN p\_number BIGINT,

IN p\_country VARCHAR(255)

)

BEGIN

INSERT INTO users (id, name, city, college, number, country)

VALUES (p\_id, p\_name, p\_city, p\_college, p\_number, p\_country);

END //

DELIMITER ;

1. **Stored Procedure for Fetching Data**

DELIMITER //

CREATE PROCEDURE InsertUsersWithPrimaryKey()

BEGIN

SELECT \* FROM users;

END //

1. **Stored Procedure for Updating Data**

DELIMITER //

CREATE PROCEDURE UpdateUser(

IN p\_id INT,

IN p\_name VARCHAR(255),

IN p\_city VARCHAR(255),

IN p\_college VARCHAR(255),

IN p\_number BIGINT,

IN p\_country VARCHAR(255)

)

BEGIN

UPDATE users

SET name = p\_name,

city = p\_city,

college = p\_college,

number = p\_number,

country = p\_country

WHERE id = p\_id;

END //

1. **Stored Procedure for Deleting All Records**

DELIMITER //

CREATE PROCEDURE DeleteAllUsers()

BEGIN

DELETE FROM users;

END //

### ****4. Example Code****

**import** java.sql.\*;

**public** **class** WithStoredProcedure {

// Database URL, username, and password

**private** **static** **final** String ***URL*** = "jdbc:mysql://localhost:3306/completejdbc";

**private** **static** **final** String ***USER*** = "root";

**private** **static** **final** String ***PASSWORD*** = "root";

**public** **static** **void** main(String[] args) {

// Example calls to methods

//insertData(108, "amarjeet singh", "bangalore", "ies college bhpal", 6261535069l, "india");

//fetchData();

//*updateData*(108, "Jane Doe", "Los Angeles", "XYZ University", 9876543210L, "india");

//deleteAllRecords();

}

**public** **static** **void** insertData(**int** id, String name, String city, String college, **long** number, String country) {

String insertProcedure = "{CALL insertData(?, ?, ?, ?, ?,?)}";

**try** (Connection connection = DriverManager.*getConnection*(***URL***, ***USER***, ***PASSWORD***);

CallableStatement insertStatement = connection.prepareCall(insertProcedure)) {

insertStatement.setInt(1,id);

insertStatement.setString(2, name);

insertStatement.setString(3, city);

insertStatement.setString(4, college);

insertStatement.setLong(5, number);

insertStatement.setString(6, country);

insertStatement.executeUpdate();

System.***out***.println("Data inserted successfully.");

} **catch** (SQLException e) {

e.printStackTrace();

}

}

**public** **static** **void** fetchData() {

String fetchProcedure = "{CALL InsertUsersWithPrimaryKey()}";

**try** (Connection connection = DriverManager.*getConnection*(***URL***, ***USER***, ***PASSWORD***);

CallableStatement fetchStatement = connection.prepareCall(fetchProcedure);

ResultSet resultSet = fetchStatement.executeQuery()) {

**while** (resultSet.next()) {

**int** id = resultSet.getInt("id");

String name = resultSet.getString("name");

String city = resultSet.getString("city");

String college = resultSet.getString("college");

**long** number = resultSet.getLong("number");

String country = resultSet.getString("country");

System.***out***.println("ID: " + id + ", Name: " + name + ", City: " + city + ", College: " + college

+ ", Number: " + number + ", Country: " + country);

}

} **catch** (SQLException e) {

e.printStackTrace();

}

}

**public** **static** **void** updateData(**int** id, String name, String city, String college, **long** number, String country) {

String updateProcedure = "{CALL UpdateUser(?, ?, ?, ?, ?, ?)}";

**try** (Connection connection = DriverManager.*getConnection*(***URL***, ***USER***, ***PASSWORD***);

CallableStatement updateStatement = connection.prepareCall(updateProcedure)) {

updateStatement.setInt(1, id);

updateStatement.setString(2, name);

updateStatement.setString(3, city);

updateStatement.setString(4, college);

updateStatement.setLong(5, number);

updateStatement.setString(6, country);

updateStatement.executeUpdate();

System.***out***.println("Data updated successfully.");

} **catch** (SQLException e) {

e.printStackTrace();

}

}

**public** **static** **void** deleteAllRecords() {

String deleteProcedure = "{CALL DeleteAllUsers()}";

**try** (Connection connection = DriverManager.*getConnection*(***URL***, ***USER***, ***PASSWORD***);

CallableStatement deleteStatement = connection.prepareCall(deleteProcedure)) {

deleteStatement.executeUpdate();

System.***out***.println("All records deleted successfully.");

} **catch** (SQLException e) {

e.printStackTrace();

}

}

}

### Q. Transaction Management Notes

**1. What is a Transaction?** A transaction is a sequence of one or more SQL operations executed as a single unit of work. It ensures that all operations within the transaction are completed successfully or none at all.

**2. ACID Properties** Transactions must adhere to the following properties to ensure data integrity:

* **Atomicity**: Ensures that all operations within the transaction are completed successfully or none are applied.
* **Consistency**: Ensures that the database transitions from one valid state to another, maintaining data integrity.
* **Isolation**: Ensures that transactions operate independently, and the intermediate state of a transaction is invisible to others.
* **Durability**: Ensures that once a transaction is committed, the changes are permanent, even in the event of a system failure.

**3. Transaction Management in JDBC**

* **setAutoCommit(boolean autoCommit)**:
  + By default, JDBC commits transactions automatically. Setting autoCommit to false allows for manual control of transactions.
* **commit()**:
  + Saves all changes made during the transaction. This method is called to apply changes permanently.
* **rollback()**:
  + Reverts all changes made during the transaction. This method is called in case of an error to undo any changes.

**4. Transaction Management in Spring Framework**

* **@Transactional Annotation**:
  + Marks a method or class to be executed within a transaction. Spring handles the transaction boundaries, committing or rolling back the transaction based on method success or failure.

**5. Transaction Propagation Types**

* **REQUIRED**:
  + Joins the existing transaction if one exists; otherwise, it creates a new transaction.
* **REQUIRES\_NEW**:
  + Always creates a new transaction, suspending any existing transaction.
* **NESTED**:
  + Executes within a nested transaction, which requires a savepoint. Useful for partial commits.
* **MANDATORY**:
  + Requires an existing transaction. Throws an exception if no transaction is present.
* **SUPPORTS**:
  + Executes within a transaction if one exists; otherwise, it runs non-transactionally.
* **NOT\_SUPPORTED**:
  + Executes without a transaction, suspending any existing transaction.
* **NEVER**:
  + Executes without a transaction. Throws an exception if a transaction exists.

**6. Isolation Levels**

* **READ\_UNCOMMITTED**:
  + Allows reading of uncommitted changes made by other transactions. This is the lowest isolation level and can lead to dirty reads.
* **READ\_COMMITTED**:
  + Ensures that only committed changes are visible to the transaction, preventing dirty reads but allowing non-repeatable reads.
* **REPEATABLE\_READ**:
  + Ensures that if a row is read twice in the same transaction, it will have the same value, preventing dirty and non-repeatable reads.
* **SERIALIZABLE**:
  + Provides the highest isolation level by ensuring that transactions are executed in a serial order, preventing all types of concurrency issues.

**7. Best Practices**

* **Keep Transactions Short**:
  + Minimize the duration of transactions to reduce locking and contention on database resources.
* **Handle Exceptions**:
  + Ensure that exceptions are properly handled to rollback transactions in case of errors.
* **Choose Appropriate Isolation Levels**:
  + Select isolation levels based on the concurrency needs and performance requirements of the application.

**8. Conclusion**

Effective transaction management is essential for maintaining the integrity and consistency of database operations. By leveraging transaction management features in JDBC and Spring, you can ensure reliable and predictable database interactions in your Java applications.

**Run the the application using cmd steps below:**

javac -cp .;C:\mysql-connector-j-9.0.0.jar Connect.java -> compile

java -cp .;C:\mysql-connector-j-9.0.0.jar Connect.java -> run