JDBC Connection

Steps for JDBC Connection

1. Load the JDBC Driver (com.mysql.cj.jdbc.Driver for MySQL).
2. Establish a connection using DriverManager.
3. Create a Statement or PreparedStatement.(Always prefer PreparedStatement over Statement, especially when handling user input!)
4. Execute a query.
5. Process the results using ResultSet.
6. Close the resources.

Problems with Statement:

- **SQL Injection Risk**: If name contains '; DROP TABLE users; --, it could delete the entire table!
- **Poor Performance**: Each query is compiled and executed separately.

Advantages of PreparedStatement over Statement:

1. **Prevents SQL Injection**: User input is treated as data, not part of the SQL command.
2. **Better Performance**: The query is precompiled and executed multiple times with different values.
3. **Cleaner Code**: Uses ? placeholders instead of manual string concatenation.

1 Difference Between `Statement` and `PreparedStatement`

Feature	`Statement`	`PreparedStatement`
Query Type	Used for static SQL queries.	Used for dynamic SQL queries with parameters.
Parameter Handling	Query must be concatenated manually.	Uses `?` placeholders for parameters.
Performance	Query is compiled every time it's executed.	Query is precompiled and reused.
SQL Injection	Prone to SQL injection (if user input is not sanitized).	Prevents SQL injection automatically.
Batch Execution	Not optimized for batch inserts/updates.	Supports batch execution efficiently.
Code Readability	Messy when using variables inside the query.	Clean and structured syntax.

Here using try with resource concept so no need to close connection manually

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
public class JDBCExample {
  public static void main(String[] args) {
    // Database connection details
    String url = "jdbc:mysql://localhost:3306/testdb"; // Change 'testdb' to your database name
    String user = "root";  // Change to your MySQL username
    String password = "password"; // Change to your MySQL password
    // SQL Query
    String query = "SELECT id, name, email FROM users"; // Change 'users' to your table name
    // Establish Connection
    try (Connection conn = DriverManager.getConnection(url, user, password);
        PreparedStatement stmt = conn.prepareStatement(query);
        ResultSet rs = stmt.executeQuery()) {
      // Process ResultSet
      while (rs.next()) {
        int id = rs.getInt("id");
        String name = rs.getString("name");
        String email = rs.getString("email");
        System.out.println("ID: " + id + ", Name: " + name + ", Email: " + email);
      }
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }
}
```

Explanation:

1. **Database Connection**
    - Uses DriverManager.getConnection(url, user, password); to establish a connection.
2. **SQL Execution**
    - Uses PreparedStatement to execute a SQL SELECT query.
3. **Result Processing**

- Retrieves id, name, and email from the users table and prints them.

4. **Resource Management**

   - Uses **try-with-resources** (try (...) {}) to automatically close Connection, PreparedStatement, and ResultSet.

INSERT Data (Adding a New Record)

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
public class JDBCInsertExample {
  public static void main(String[] args) {
    String url = "jdbc:mysql://localhost:3306/testdb";
    String user = "root";
    String password = "password";
    String insertQuery = "INSERT INTO users (name, email) VALUES (?, ?)";
    try (Connection conn = DriverManager.getConnection(url, user, password);
        PreparedStatement stmt = conn.prepareStatement(insertQuery)) {
      stmt.setString(1, "John Doe");
      stmt.setString(2, "john.doe@example.com");
      int rowsInserted = stmt.executeUpdate();
      if (rowsInserted > 0) {
        System.out.println("✅ A new user was inserted successfully!");
      }
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }
}
```

✅ **Explanation:** Uses PreparedStatement to insert a new record into the users table.

2️⃣ UPDATE Data (Modifying an Existing Record)

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
public class JDBCUpdateExample {
  public static void main(String[] args) {
    String url = "jdbc:mysql://localhost:3306/testdb";
    String user = "root";
    String password = "password";
    String updateQuery = "UPDATE users SET email = ? WHERE name = ?";
```

```
    try (Connection conn = DriverManager.getConnection(url, user, password);
         PreparedStatement stmt = conn.prepareStatement(updateQuery)) {
        stmt.setString(1, "new.email@example.com");
        stmt.setString(2, "John Doe");
        int rowsUpdated = stmt.executeUpdate();
        if (rowsUpdated > 0) {
            System.out.println("✅ User's email updated successfully!");
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
  }
}
```

✅ **Explanation:** Updates the email for a user with the name "John Doe."

3️⃣ DELETE Data (Removing a Record)

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
public class JDBCDeleteExample {
  public static void main(String[] args) {
    String url = "jdbc:mysql://localhost:3306/testdb";
    String user = "root";
    String password = "password";
    String deleteQuery = "DELETE FROM users WHERE name = ?";
    try (Connection conn = DriverManager.getConnection(url, user, password);
         PreparedStatement stmt = conn.prepareStatement(deleteQuery)) {
        stmt.setString(1, "John Doe");
        int rowsDeleted = stmt.executeUpdate();
        if (rowsDeleted > 0) {
            System.out.println("✅ User deleted successfully!");
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
  }
}
```

✅ **Explanation:** Deletes the user with the name "John Doe."

Key Takeaways

1. **Use PreparedStatement** to avoid SQL injection and optimize performance.
2. **Use executeUpdate()** for INSERT, UPDATE, and DELETE operations.
3. **Always close resources** (Connection, Statement, ResultSet) using **try-with-resources**.
4. **Handle exceptions properly** to catch and debug SQL errors.

Question

The use of **Class.forName("com.mysql.cj.jdbc.Driver")** in JDBC

Before Java 6

It **explicitly loads** the MySQL JDBC Driver.
Registers the driver with the DriverManager, allowing JDBC to use it for database connections.
Required in **older versions of JDBC (before Java 6).**

After Java6

- The JDBC Driver is automatically loaded if it's on the classpath.
- JDBC 4.0 introduced Service Provider Mechanism (SPI), which auto-discovers the driver.

```
// No need for Class.forName()
Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/testdb", "root", "password");
```

- The MySQL driver will be automatically loaded from **mysql-connector-java** **(if added in the classpath).**

JDBC Version	Is `Class.forName()` Needed?	Why?
JDBC 3.0 (Java 5 & earlier)	✅ Yes	Drivers were not auto-registered.
JDBC 4.0+ (Java 6 & later)	❌ No	Auto-loading via SPI (Service Provider Interface).

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

public class JdbcExample {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/employees";
        String user = "root";
        String password = "password";

        try {
            // Load MySQL JDBC Driver
            Class.forName("com.mysql.cj.jdbc.Driver");

            // Establish Connection
            Connection conn = DriverManager.getConnection(url, user, password);
```

But coming to spring , no need to write all these code

Spring Boot + JPA Approach

```java
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import jakarta.persistence.*;
@Entity
class Employee {
  @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;
  private String name;
}
@Repository
interface EmployeeRepository extends JpaRepository<Employee, Long> { }
@Service
public class EmployeeService {
  @Autowired private EmployeeRepository repository;

  public List<Employee> getAllEmployees() {
    return repository.findAll();
  }
```

}

Hibernate (ORM Framework)

Hibernate is used to map Java objects to database tables.

Example: Employee Entity with Hibernate

```
@Entity
@Table(name = "employees")
public class Employee {
   @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   private String name;
}
```

Repository for CRUD Operations

```
@Repository
public interface EmployeeRepository extends JpaRepository<Employee, Long> { }
```

How to Resolve Conflicts in Git

- **Step 1:** Pull latest changes → git pull origin main
- **Step 2:** If conflict occurs, manually edit conflicted files.
- **Step 3:** Mark conflicts as resolved → git add .
- **Step 4:** Commit changes → git commit -m "Resolved merge conflict"
- **Step 5:** Push changes → git push origin main

What is Retrospection in Agile?

- A **Sprint Retrospective** is a meeting at the end of a sprint where the team reflects on:
    - What went well?
    - What went wrong?
    - How can we improve?

How to Make an API HTTPS?

- Purchase an **SSL certificate** (or use a free one like Let's Encrypt).

- Install the certificate on the server.
- Configure the web server (e.g., Apache, Nginx) to use HTTPS.
- Update API base URLs to https:// instead of http://.

Example for **Spring Boot:**

```
server.port=8443
server.ssl.key-store=classpath:keystore.p12
server.ssl.key-store-password=yourpassword
server.ssl.key-store-type=PKCS12
```

Use of @Query Annotation in Spring Data JPA

The @Query annotation in Spring Data JPA allows writing **custom JPQL (Java Persistence Query Language) or native SQL queries** inside repository methods.

Example of @Query with JPQL

```
@Repository
public interface EmployeeRepository extends JpaRepository<Employee, Long> {

    // Using JPQL (Java Persistence Query Language)
    @Query("SELECT e FROM Employee e WHERE e.name = :name")
    List<Employee> findByName(@Param("name") String name);
}
```

Example of @Query with Native SQL Query

```
@Repository
public interface EmployeeRepository extends JpaRepository<Employee, Long> {

    // Using Native Query
    @Query(value = "SELECT * FROM employees WHERE name = :name", nativeQuery = true)
    List<Employee> findByNameNative(@Param("name") String name);
}
```

2. Native Query vs JPQL

Feature	JPQL	Native Query
Syntax	Uses **entity names**	Uses **table names**
Database Independence	**Database-independent**	**Database-specific** (depends on SQL dialect)
Flexibility	Cannot use complex SQL features	Can use native SQL functions, joins, etc.
Performance	May be **optimized** by JPA	Executes exactly as written

Example of Difference

JPQL Query:

java Copy Edit

```java
@Query("SELECT e FROM Employee e WHERE e.salary > :salary")
```

Native Query:

java Copy Edit

```java
@Query(value = "SELECT * FROM employees WHERE salary > :salary", nativeQuery = true)
```