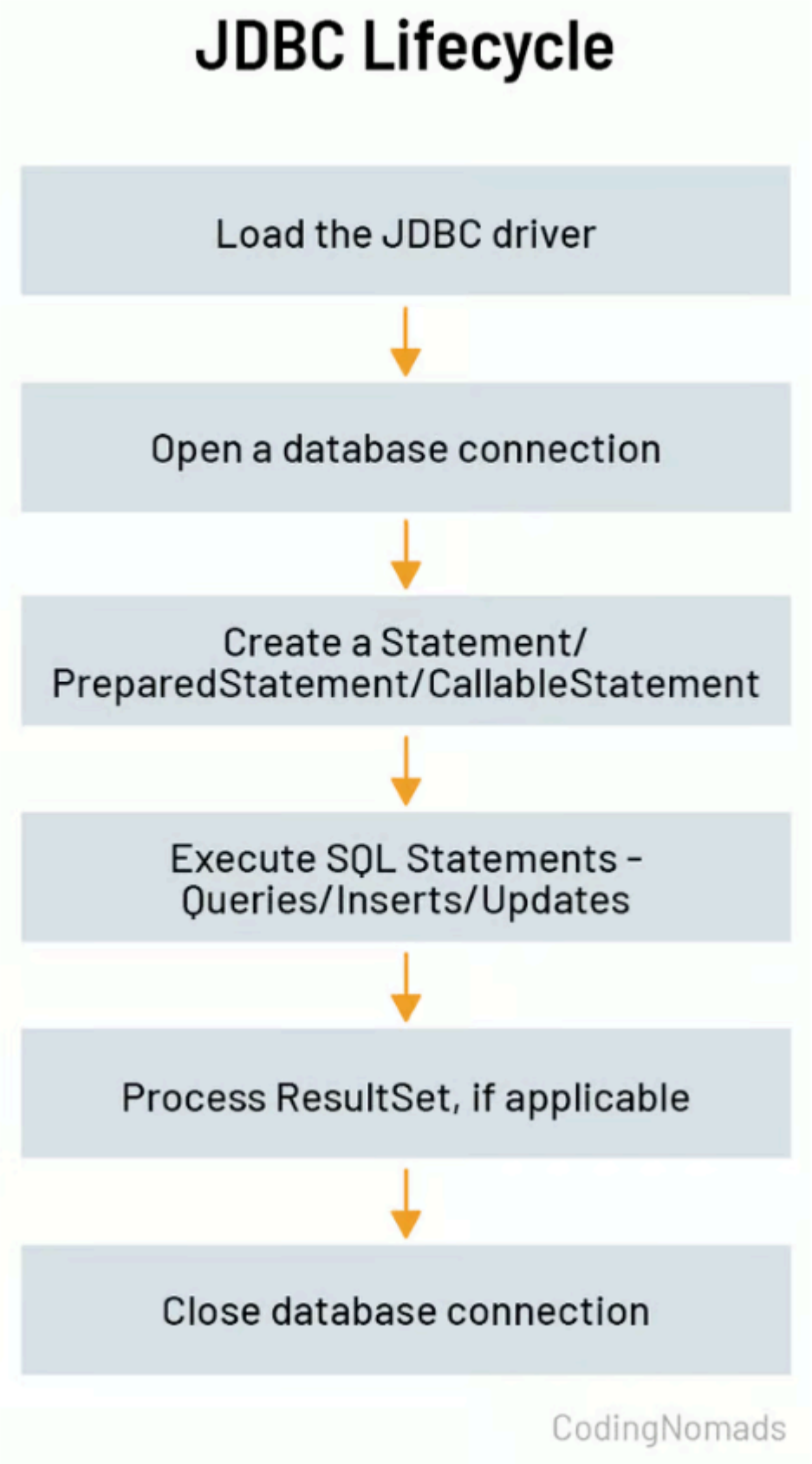
# JDBC

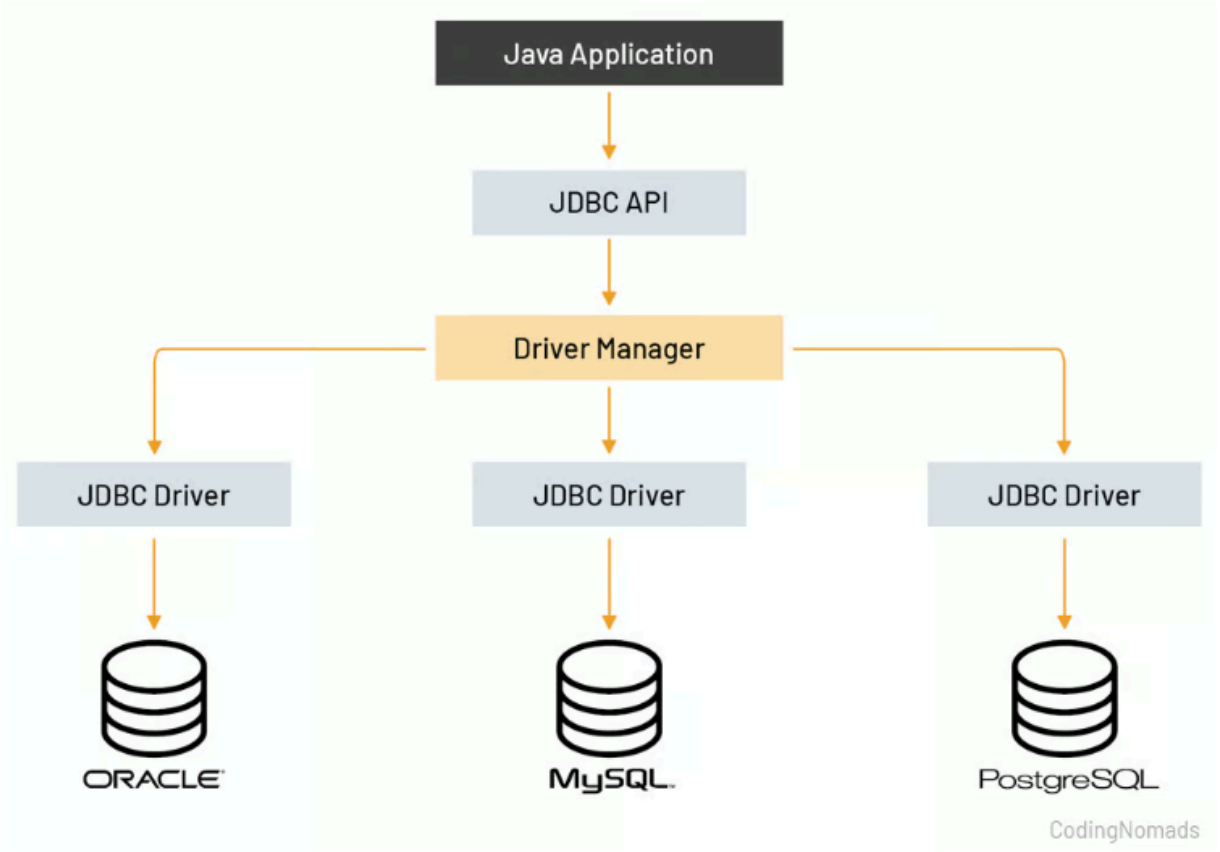
**Table of Contents**



# JDBC life cycle

1. Load the JDBC driver
2. Open a database connection
3. Create a `Statement` / `PreparedStatement` / `CallableStatement`
4. Execute SQL statements
5. Retrieve and potentially process the results of any SQL query
6. Close database connection


JDBC Lifecycle
Load the JDBC driver
Open a database connection
Create a Statement/
PreparedStatement/CallableStatement
Execute SQL Statements -
Queries/Inserts/Updates
Process ResultSet, if applicable
Close database connection
CodingNomads




# JDBC Lifecycle Guide

This guide explains the JDBC (Java Database Connectivity) lifecycle with code examples for each step.

## 1. Load the JDBC Driver

Before Java 8, you needed to explicitly load the JDBC driver using `Class.forName()`. In modern Java applications, this step is automatic when the driver is in the classpath.

```
// Legacy way (before Java 8)
Class.forName("com.mysql.cj.jdbc.Driver");

// Modern way - no explicit loading required
// Just ensure the driver is in your project dependencies
```

Add the appropriate dependency to your project:

```
<!-- For Maven -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
```

```
    <version>8.0.27</version>
</dependency>
```

# 2. Open a Database Connection

Use `DriverManager.getConnection()` to establish a connection to your database:

```
String url = "jdbc:mysql://localhost:3306/your_database";
String username = "your_username";
String password = "your_password";

try (Connection connection = DriverManager.getConnection(url, username, password)) {
    // Use the connection here
} catch (SQLException e) {
    e.printStackTrace();
}
```

# 3. Create SQL Statements

JDBC provides three types of statements:

## Statement

Used for simple SQL queries without parameters:

```
Statement statement = connection.createStatement();
```

## PreparedStatement

Used for SQL queries with parameters (recommended for preventing SQL injection):

```
String sql = "INSERT INTO users (name, email) VALUES (?, ?)";
PreparedStatement pstmt = connection.prepareStatement(sql);
pstmt.setString(1, "John Doe");
pstmt.setString(2, "john@example.com");
```

## CallableStatement

Used for calling stored procedures:

```
String sql = "{call GetUserDetails(?)}";
CallableStatement cstmt = connection.prepareCall(sql);
cstmt.setInt(1, userId);
```

# 4. Execute SQL Statements

Different methods for executing SQL statements:

## For SELECT queries:

```
// Using Statement
ResultSet rs = statement.executeQuery("SELECT * FROM users");

// Using PreparedStatement
ResultSet rs = pstmt.executeQuery();
```

## For INSERT, UPDATE, DELETE:

```
// Using Statement
int rowsAffected = statement.executeUpdate("UPDATE users SET name =
'Jane' WHERE id = 1");

// Using PreparedStatement
int rowsAffected = pstmt.executeUpdate();
```

## For Stored Procedures:

```
boolean hasResults = cstmt.execute();
```

# 5. Process Results

Process the results from a SELECT query:

```
try (ResultSet rs = statement.executeQuery("SELECT * FROM users")) {
    while (rs.next()) {
        String name = rs.getString("name");
        String email = rs.getString("email");
        int age = rs.getInt("age");

        // Process the data
```

```
            System.out.println("Name: " + name + ", Email: " + email + ",
    Age: " + age);
        }
    }
```

## 6. Close Database Connection

Always close your resources in reverse order of creation. Use try-with-resources for automatic resource management:

```
    try (
        Connection connection = DriverManager.getConnection(url, username,
    password);
        PreparedStatement pstmt = connection.prepareStatement(sql);
        ResultSet rs = pstmt.executeQuery()
    ) {
        // Process results
    } catch (SQLException e) {
        e.printStackTrace();
    }
```

## Complete Example

Here's a complete example putting all the steps together:

```
    import java.sql.*;

    public class JDBCExample {
        public static void main(String[] args) {
            String url = "jdbc:mysql://localhost:3306/your_database";
            String username = "your_username";
            String password = "your_password";

            try (Connection connection = DriverManager.getConnection(url,
    username, password)) {
                // Insert a new user
                String insertSql = "INSERT INTO users (name, email) VALUES
    (?, ?)";
                try (PreparedStatement pstmt =
    connection.prepareStatement(insertSql)) {
                    pstmt.setString(1, "John Doe");
                    pstmt.setString(2, "john@example.com");
                    int rowsInserted = pstmt.executeUpdate();
                    System.out.println("Rows inserted: " + rowsInserted);
                }
```

```
18
19              // Query users
20              String selectSql = "SELECT * FROM users";
21              try (
22                  Statement stmt = connection.createStatement();
23                  ResultSet rs = stmt.executeQuery(selectSql)
24              ) {
25                  while (rs.next()) {
26                      System.out.printf("User: %s, Email: %s%n",
27                          rs.getString("name"),
28                          rs.getString("email")
29                      );
30                  }
31              }
32          } catch (SQLException e) {
33              e.printStackTrace();
34          }
35      }
36  }
```

## Best Practices:

1. Always use try-with-resources for automatic resource cleanup
2. Use **PreparedStatement** instead of **Statement** when dealing with user input
3. Handle **SQLExceptions** appropriately in your application
4. Use connection pooling in production applications
5. Don't store credentials in your code; use configuration files or environment variables
6. Close resources in the reverse order of creation when not using try-with-resources

Remember to replace the database URL, username, and password with your actual database credentials when using this code.