

# Large-Scale and Multi-Structured Databases

## ***JDBC and Maven***

Prof Pietro Ducange

Eng. Alessio Schiavo

# Objective of this Class

- Let the student to think and remember his/her skills about programming in JAVA and using API for manipulating a relational database.
- We will recall some basics elements of JDBC.
- We will briefly introduce MAVEN
- Students will be asked to solve some programming exercises using an IDE (Eclipse or IntelliJ).

# Motivations

Let suppose that our JAVA application needs to access and manipulate data stored in a ***Database Management System*** (DBMS).

A number of difficulties may appear (***Impedance Mismatch***), depending on:

- ***Different access modes:***

SQL is a set-oriented language (operates on sets of tuples), while applications operate on a single tuple at a time

- ***Different types of data:***

Each programming language defines its own types of data, which may differ from those of the SQL (and DBMS)

Es: Java defines only one type to represent the strings

(`java.lang.String`), SQL defines several: CHAR, VARCHAR, TEXT, LONGTEXT,

...

In case we try to map the concepts of the DB within object-oriented languages there are also other difficulties called ***Object-Relational Impedance Mismatch***.

# Connectors

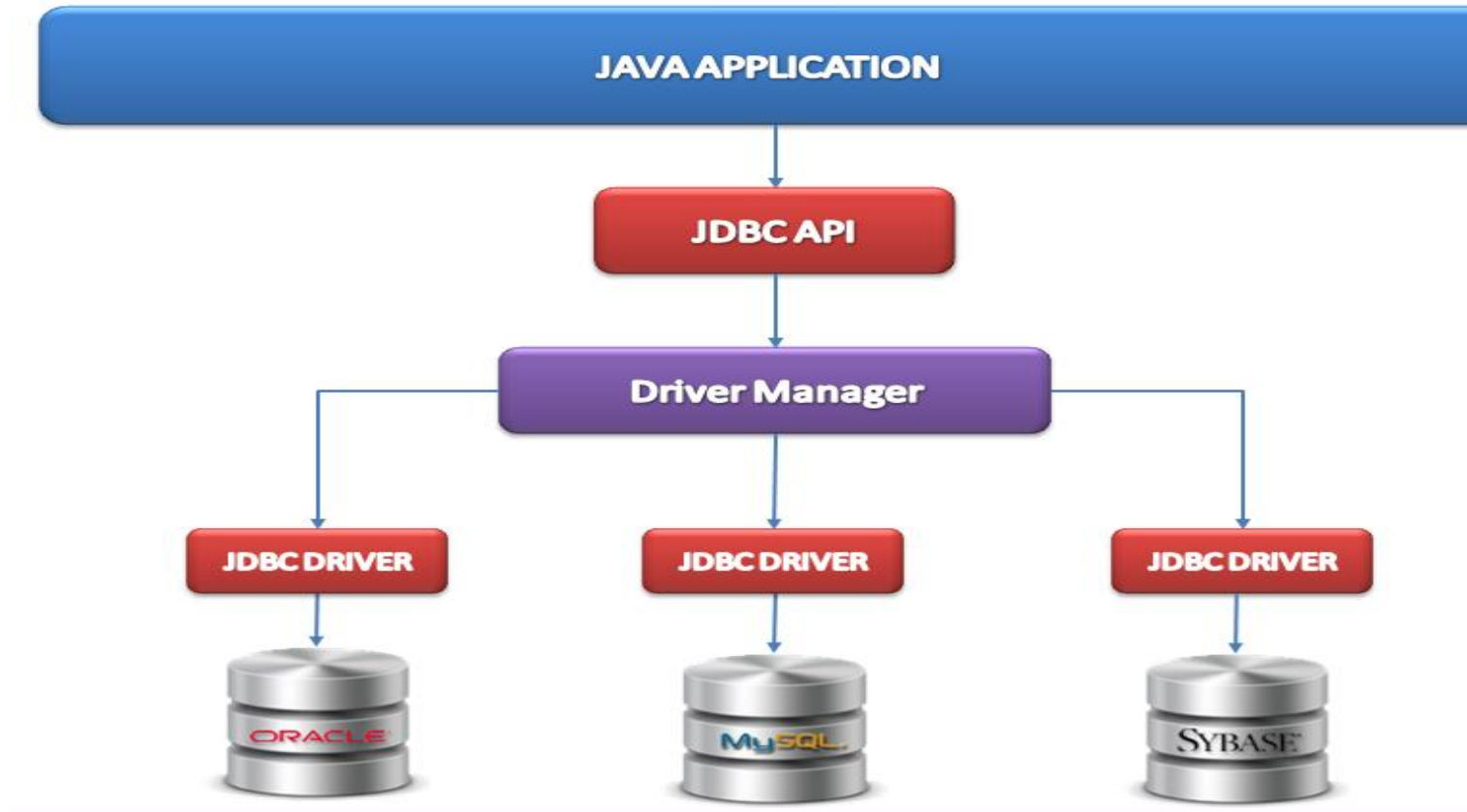
DBMSs provide specific **connectors** (drivers) for certain programming languages.

Connectors are characterized by a **low portability**: each DBMS has its own API, so the **application** becomes **dependent** on the specific DBMS.

In 1992 Microsoft introduced **ODBC** (Open DataBase Connectivity), which defines a unified API for accessing databases in C.

Java introduces **JDBC** (Java DataBase Connectivity): it is almost equivalent to ODBC, but provides a Java-specific API (java.sql.\*)

# JDBC Architecture



*Image extracted from:*

# Basic Workflow of JDBC

1. Load the JDBC Driver
2. Establish a Connection: Create a connection to the database using **DriverManager**
3. Create a Statement: After establishing the connection, you can create a **Statement** or **PreparedStatement** to execute queries
4. Execute SQL Queries: Queries are executed using the **executeQuery** or **executeUpdate** methods
5. Process Results: The **ResultSet** object contains the results of the query, and you can iterate through it to retrieve data
6. **Close the Connection:** It's important to close the Connection, Statement, and ResultSet to free up database resources

# JDBC Driver Manager

The class ***DriverManager*** (java.sql.DriverManager) manages the JDBC drivers and ***creates connections*** to the DBMS.

A ***connection string*** is required to obtain a connection, in form of URN (***Uniform Resource Name***). The string is specific for DBMS and indicates:

- The type of driver
- The type of DBMS
- The host of the DBMS server
- Access credentials
- The database to which you can connect

# Accessing the DB

```
2
3 import java.sql.*;
4
5 public class ExampleConnect {
6
7     public static void main(String[] args) {
8         Connection connection = null;
9         try{
10             connection = DriverManager.getConnection( url: "jdbc:mysql://localhost:3306/unipi", user: "alessio", password: "root");
11             // Do something...
12         } catch (SQLException e) {
13             throw new RuntimeException(e);
14         } finally{
15             if (connection != null) {
16                 try { connection.close(); } catch (SQLException e) {}
17             }
18         }
19     }
20 }
```

Returns from the DriverManager a connection to the DB specified by the connection string

At the end of all the operations we have to close the connection



# Performing Queries

```
// Create a SQL statement
Statement stmt = conn.createStatement();
stmt.execute("SELECT * FROM user");
```

The interface Statement creates an object used for executing a static SQL statement and returning the results it produces.

```
// Fetch results
ResultSet rs = stmt.getResultSet();
while (rs.next()){
    System.out.print(rs.getInt("id"));
    System.out.print(" ");
    System.out.print(rs.getString("first_name"));
    System.out.print(" ");
    System.out.println(rs.getString("last_name"));
}
rs.close();
```

The ResultSet object allows to iterate on each tuple of the table returned by a query.

To obtain the values of the individual columns, specific methods are used for each data type

```
// Close the statement and the connection
stmt.close();
conn.close();
```

Also the ResultSet and the statement objects must be closed.

# Prepared Statements

Using the interface ***java.sql.Statement*** the SQL statement is compiled at the execution time.

Using prepared statements allows us to ***pre-fill*** a statement and improves performance when the same query is ***executed multiple times***, even with different parameters.

The parameters are indicated by question marks and set with appropriate setters.

```
String sql = "SELECT * FROM user WHERE username = ?";  
PreparedStatement pstmt = conn.prepareStatement(sql);  
pstmt.setString(1, "mario.rossi");  
pstmt.execute();
```

Placeholder for  
the parameters

Parameters setup

# When to use **Statement** vs **PreparedStatement**

Use **Statement** when:

- You are executing **simple, static SQL queries** without parameters.
- The query is only executed once and doesn't need to be reused
- Performance and security (SQL injection) are not major concerns.

Use **PreparedStatement** when:

- You are executing **parameterized queries**, especially when the input comes from user data.
- You need to execute the same query **multiple times** with different parameters.
- You want to **prevent SQL injection**.
- You want to improve performance for queries that are executed frequently.
- You are handling **large objects** (BLOBs/CLOBs) or doing batch processing.

# Updating Tables

**INSERT**, **UPDATE** and **DELETE** statements do not return a ResultSet object.

The ***executeUpdate()*** method executes the statement and returns the number of rows actually inserted/modified/deleted.

```
String sql = "INSERT INTO user SET username = ?, first_name = ?, last_name = ?, "  
            + "email = ?, password = SHA1(?)";  
PreparedStatement pstmt = conn.prepareStatement(sql);  
pstmt.setString(1, "alice.ferrari");  
pstmt.setString(2, "Alice");  
pstmt.setString(3, "Ferrari");  
pstmt.setString(4, "alice.ferrari@example.com");  
pstmt.setString(5, "1234");  
  
int insertedRows = pstmt.executeUpdate();  
System.out.println("Rows inserted: " + insertedRows);
```

Executes the statement  
and returns the number  
of inserted rows

# Maven

“Apache Maven is a **software project management** and comprehension tool. Based on the concept of a project object model (**POM**).

Maven can manage a project's **build, reporting** and **documentation** from a central piece of information.” extracted from <https://maven.apache.org/>

We will use Maven for managing the dependencies in our JAVA programs.

Roughly speaking, we will avoid to manually add libraries and API.

# POM file

```
<?xml version="1.0" encoding="UTF-8" ?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>exercises</groupId>
  <artifactId>exercises</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>8.0.17</version>
    </dependency>
  </dependencies>
  <build>
    <sourceDirectory>src</sourceDirectory>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.0</version>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

It is an XML file that contains information about the project and configuration details used by Maven to build the project.

It contains default values for most projects.

Whenever we need a new dependency, we add new fields to the ***dependencies list***.

***groupId*** uniquely identifies a project across all projects.

***artifactId*** is the name of the jar without version.

By default, Maven will download from the central repository.

# Creating a Maven Project (Using Eclipse)

1. Create a New Java Project
2. Create your first Package
3. Create your first Class (main class)
4. Right Click on your Project
5. Select Configure-> Convert to Maven Project
6. Add your dependencies
7. Right Click on your Project
8. Select Maven->Update Project

# Creating a Maven Project (Using IntelliJ)

1. File -> New Project
2. Enter/choose the values displayed in the next screenshot

The screenshot shows the 'New Project' dialog in IntelliJ IDEA. On the left, the 'Generators' list has 'Maven Archetype' selected. The main panel shows the following configuration:

- Name:** lab01\_maven\_jdbc
- Location:** ~/repository/UNIFI/LSMSD/2022\_2023
- JDK:** 1.8 java version "1.8.0\_265"
- Catalog:** Internal
- Archetype:** org.apache.maven.archetypes:maven-archetype-quickstart
- Version:** RELEASE

Below these fields is the 'Additional Properties' section, which is currently empty. At the bottom, the 'Advanced Settings' section is expanded, showing:

- GroupId:** it.unipi.lsmsd
- ArtifactId:** lab01\_maven\_jdbc
- Version:** 1.0.0-SNAPSHOT

A red rectangle highlights the 'Advanced Settings' section, and a red arrow points from the text on the right towards it. 'Create' and 'Cancel' buttons are at the bottom right.

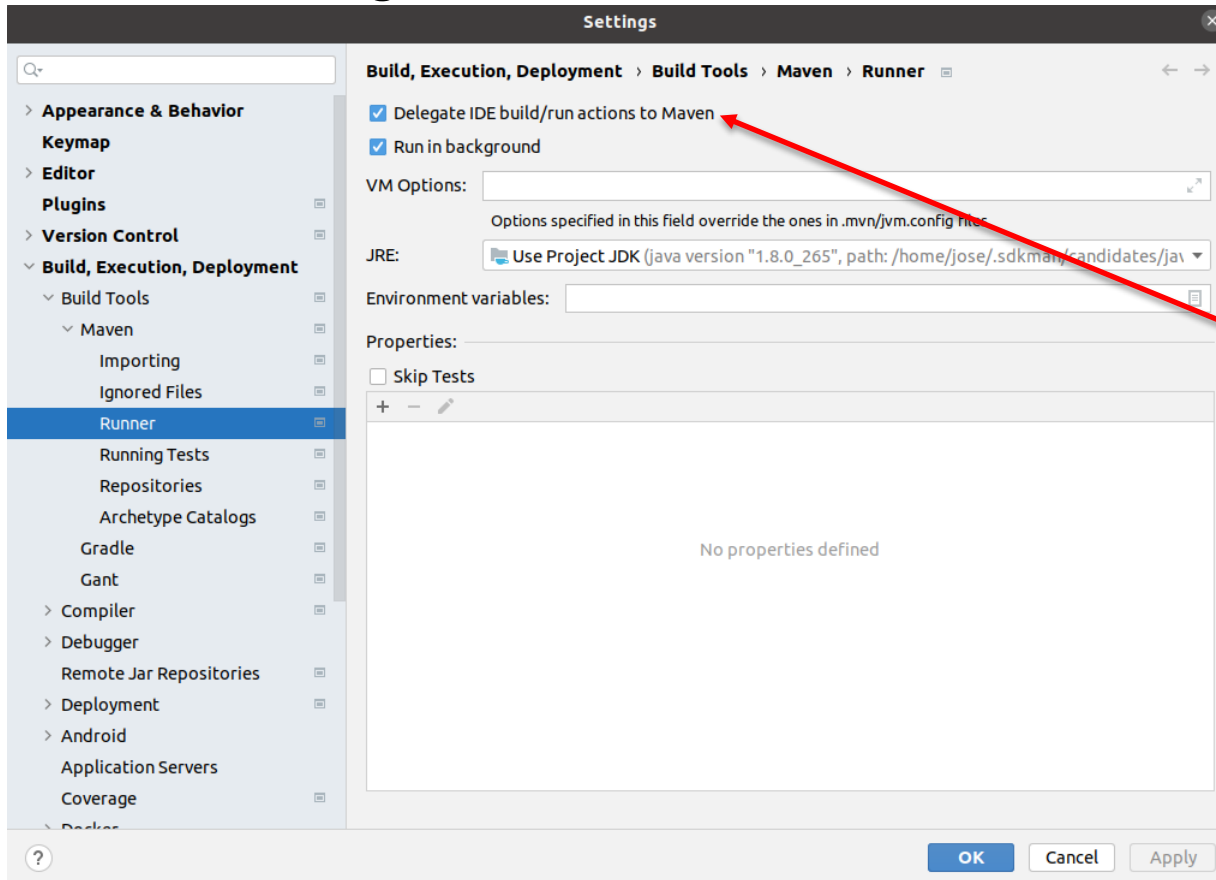
- Coordinates are made up of GroupId, ArtifactId and Version (Maven trinity!).
- Coordinates define the unique place of the project in the Maven universe.



# Creating a Maven Project (Using IntelliJ)

Remember to delegate the build/run actions to Maven:

1. File -> Settings:



Check on this option.

# Setting up the pom.xml file

For this lab session, we would like to interact with a MySQL database so it is required to add the MySQL JDBC Driver.

```
m pom.xml (lab01_maven_jdbc) x
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
4   <modelVersion>4.0.0</modelVersion>
5
6   <groupId>it.unipi.lsmsd</groupId>
7   <artifactId>lab01_maven_jdbc</artifactId>
8   <version>1.0.0-SNAPSHOT</version>
9
10  <name>lab01_maven_jdbc</name>
11
12  <properties>
13    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
14    <maven.compiler.source>1.8</maven.compiler.source>
15    <maven.compiler.target>1.8</maven.compiler.target>
16  </properties>
17
18  <dependencies>
19    <dependency>
20      <groupId>mysql</groupId>
21      <artifactId>mysql-connector-java</artifactId>
22      <version>8.0.30</version>
23    </dependency>
24  </dependencies>
25
26  <build>
27    <finalName>${artifactId}</finalName>
28  </build>
29 </project>
30
```

# First program with JDBC – Example 01

```
1 package it.unipi.lsmsd;
2
3 import java.sql.*;
4
5 public class Example01 {
6
7     public static void main(String[] args) {
8         Connection connection = null;
9         Statement statement = null;
10        ResultSet resultSet = null;
11        try{
12            connection = DriverManager.getConnection( url: "jdbc:mysql://localhost:3306/unipi?" +
13                "zeroDateTimeBehavior=CONVERT_TO_NULL&serverTimezone=CET", user: "jose", password: "jose");
14            statement = connection.createStatement();
15            resultSet = statement.executeQuery( sql: "select id, name from employee order by name asc");
16            while (resultSet.next()) {
17                Integer id = resultSet.getInt( columnLabel: "id");
18                String name = resultSet.getString( columnLabel: "name");
19                System.out.format("%d) %s \n", id, name);
20            }
21        } catch (SQLException e) {
22            throw new RuntimeException(e);
23        } finally{
24            if (resultSet != null) {
25                try { resultSet.close();} catch (SQLException e) {}
26            }
27            if (statement != null) {
28                try { statement.close();} catch (SQLException e) {}
29            }
30            if (connection != null) {
31                try { connection.close();} catch (SQLException e) {}
32            }
33        }
34    }
35 }
```

Too many lines of code to close these objects!!!

Let's organize the code.

# First program with JDBC – Example 01

```
1 package it.unipi.lmsd;
2
3 import java.sql.*;
4 import java.util.Properties;
5
6 public class Example02 {
7     private static final String MYSQL_HOST = "localhost";
8     private static final Integer MYSQL_PORT = 3306;
9     private static final String MYSQL_DATABASE = "unipi";
10    private static final String MYSQL_USERNAME = "jose";
11    private static final String MYSQL_PASSWORD = "jose";
12    // format: mysql://<username>:<password>@<host>:<port>/<db_name>
13    private static final String JDBC_URL = "jdbc:mysql://%s:%s@%s:%d/%s";
14
15    public static void main(String[] args) {
16        String jdbcUrl = String.format(JDBC_URL, MYSQL_USERNAME, MYSQL_PASSWORD, MYSQL_HOST, MYSQL_PORT, MYSQL_DATABASE);
17        Properties properties = new Properties();
18        properties.put("zeroDateTimeBehavior", "CONVERT_TO_NULL");
19        properties.put("serverTimezone", "CET");
20
21        try(
22            Connection connection = DriverManager.getConnection(jdbcUrl, properties);
23            Statement statement = connection.createStatement();
24            ResultSet resultSet = statement.executeQuery( sql: "select id, name from employee order by name asc");
25        ){
26            while (resultSet.next()) {
27                Integer id = resultSet.getInt( columnLabel: "id");
28                String name = resultSet.getString( columnLabel: "name");
29                System.out.format("%d %s \n", id, name);
30            }
31        } catch (SQLException e) {
32            throw new RuntimeException(e);
33        }
34    }
```

Try-with-resources statement.

For more information: <https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html>

# Suggested Readings

<https://dev.mysql.com/doc/connector-j/5.1/en/connector-j-usagenotes-basic.html>

<https://docs.oracle.com/javase/8/docs/api/java/sql/package-summary.html>

From <https://mvnrepository.com/> the dependencies for several tools maybe retrieved and simply added to our JAVA project.

# Exercise 1

***From the command line:***

1. Run the MySQL Server
2. Create a database
3. Create a table “company” with the following Attributes: name, address, employee\_count and website.

***From the Java Application Side,*** write a program that:

1. Makes an access to the database
2. Inserts 3 tuples , such as <Unipi, Lungarno Galilei, 5000, [www.unipi.it](http://www.unipi.it)>.
3. Retrieves all the tuples of the table “company” and prints them on the screen
4. Deletes all the inserted tuples.