A Partnership for Progress

Introduction to Java Database Connectivity

Introduction to JDBC - Java DataBase Connectivity

Learning Objectives:

In this presentation, we will get to know how to connect MySql Database through a Java Class.

By the end of this session, learners will be able to:

- Demonstrate how to setup JDBC driver.
- Describe the JDBC Connectivity Steps
- Make a MySql Database connectivity through a Java Class.
- Write a SQL query from within a Java class.
- Access records from Database Table through Java Class with user input values.

Agenda/Overview

- Java Database Connectivity/Application Programming Interface.
- Download MySQL JDBC Driver.
- Java Database Connectivity Steps.
  - Load database driver.
  - Open Database Connection.
  - Sending SQL to the Database.
  - Common Methods of Statement class.
  - Response From Database.
  - Close Database Connection.
- Overview of Prepared Statements.
- Adding Placeholders to the Statement.
- Prepared Statement With DML.

Java Database Connectivity Driver/ Application Programming Interface

- A Java Database Connectivity (**JDBC)** Driver Application Programming Interface (API) manages connections to a database issuing queries and commands, and handles result sets obtained from the database.
- An API is Java's solution to the problems. It allows for programmatic interaction with a database:
  - Initial connection.
  - Querying (SQL).
  - Inserting, updating, and deleting of data (DML).
  - Creating, updating, and deleting of tables (DDL).
- JDBC Driver classes are available in the **java.sql package**.

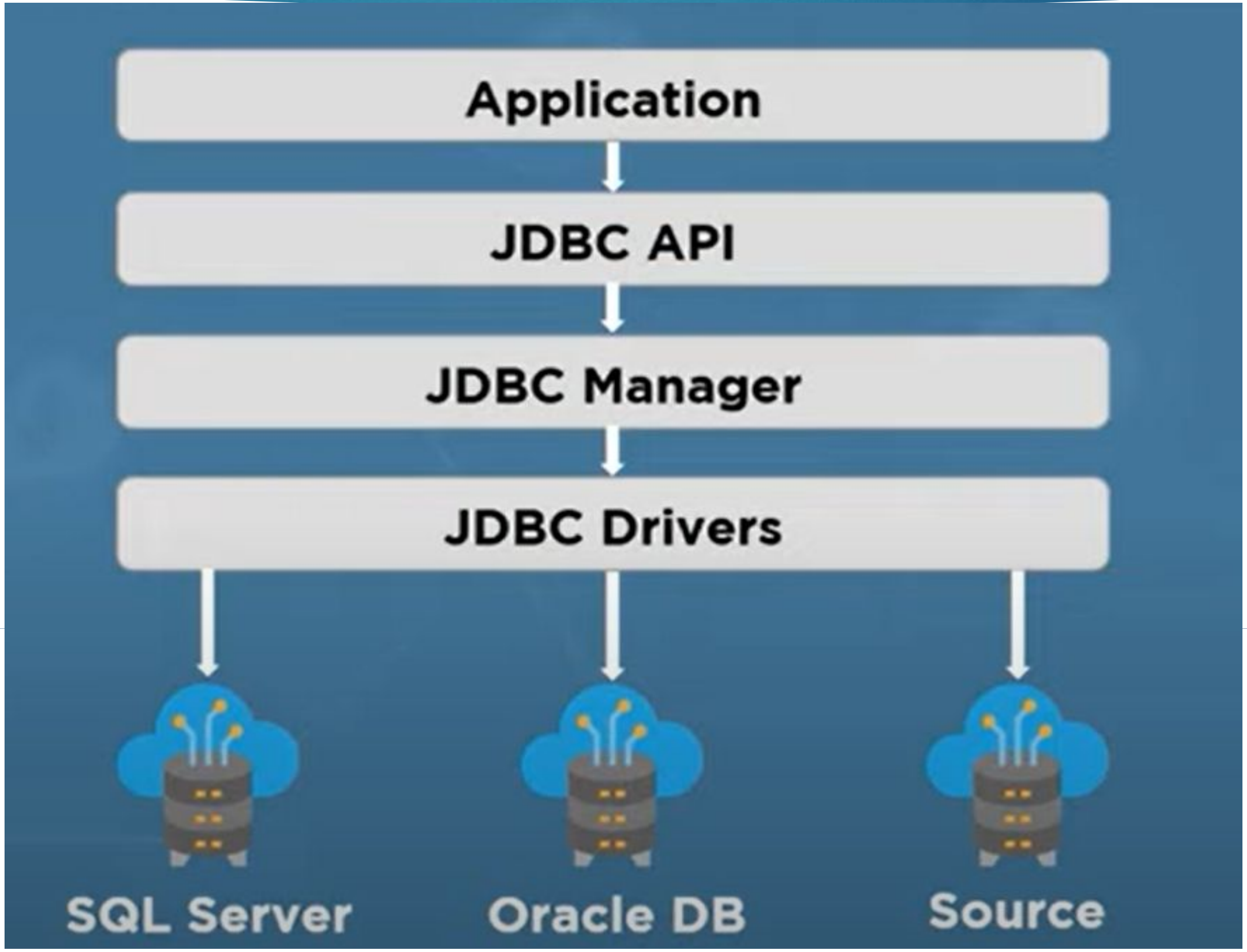
Java Database Connectivity Driver/ Application Programming Interface

(continued)

JDBC/API:

- ❑ A software component enabling a Java application to interact with a database.
- ❑ Implements the protocol for transferring the query and result between client and database.
- ❑ Allows us to develop an application that can send SQL statements to different data sources.

JDBC basic Architecture



Download / Install MySQl JDBC Driver

We can Download and install JDBC driver:

1. By using Maven Dependency
2. By using Jar File - need installation steps

Download MySQl JDBC Driver using Maven

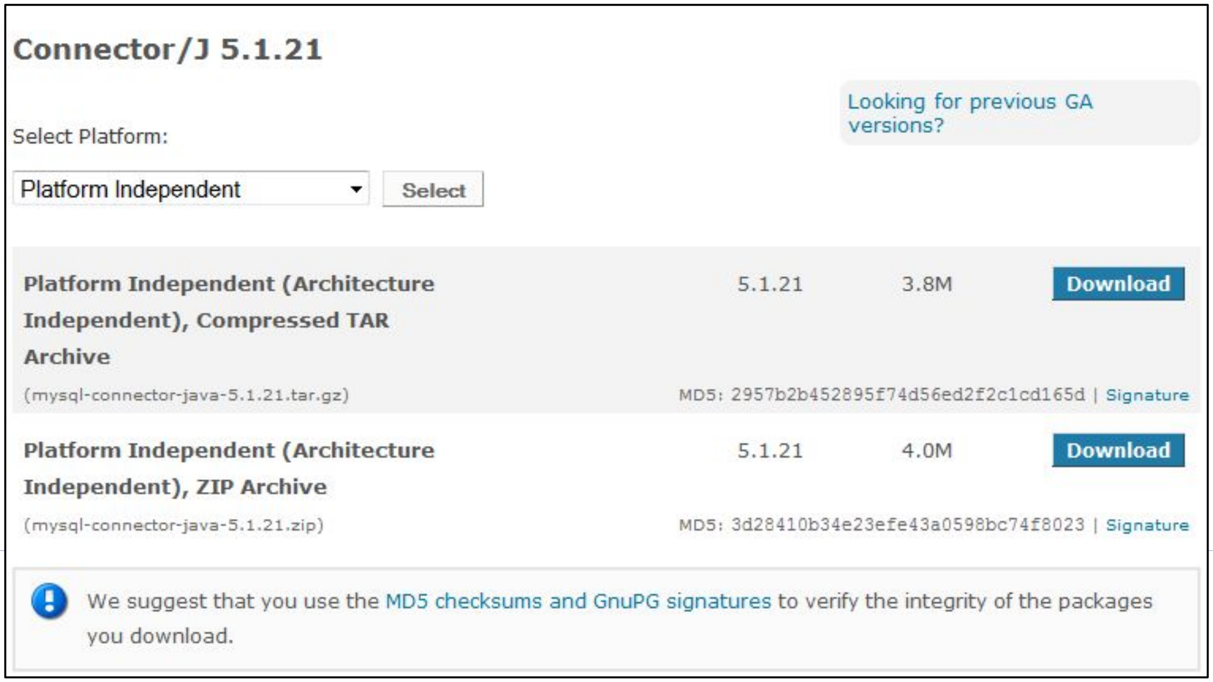
The MySQL Driver is available on Maven Central repository. Copy below MySQL JDBC driver maven dependency and paste in your project pom.xml file:

```
<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->

<dependency>

    <groupId>mysql</groupId>

    <artifactId>mysql-connector-java</artifactId>

    <version>8.0.18</version>
  </dependency>
```

Click here for JDBC Maven dependency official website

MySQL Java Database Connectivity Driver

- **Step 1:** Click here to download MySQL JDBC driver.



Connector/J 5.1.21

Looking for previous GA versions?

Select Platform:

Platform Independent | Select

Platform Independent (Architecture Independent), Compressed TAR Archive	5.1.21	3.8M	Download
(mysql-connector-java-5.1.21.tar.gz)	MD5: 2957b2b452895f74d56ed2f2c1cd165d | Signature		
Platform Independent (Architecture Independent), ZIP Archive	5.1.21	4.0M	Download
(mysql-connector-java-5.1.21.zip)	MD5: 3d28410b34e23efe43a0598bc74f8023 | Signature		

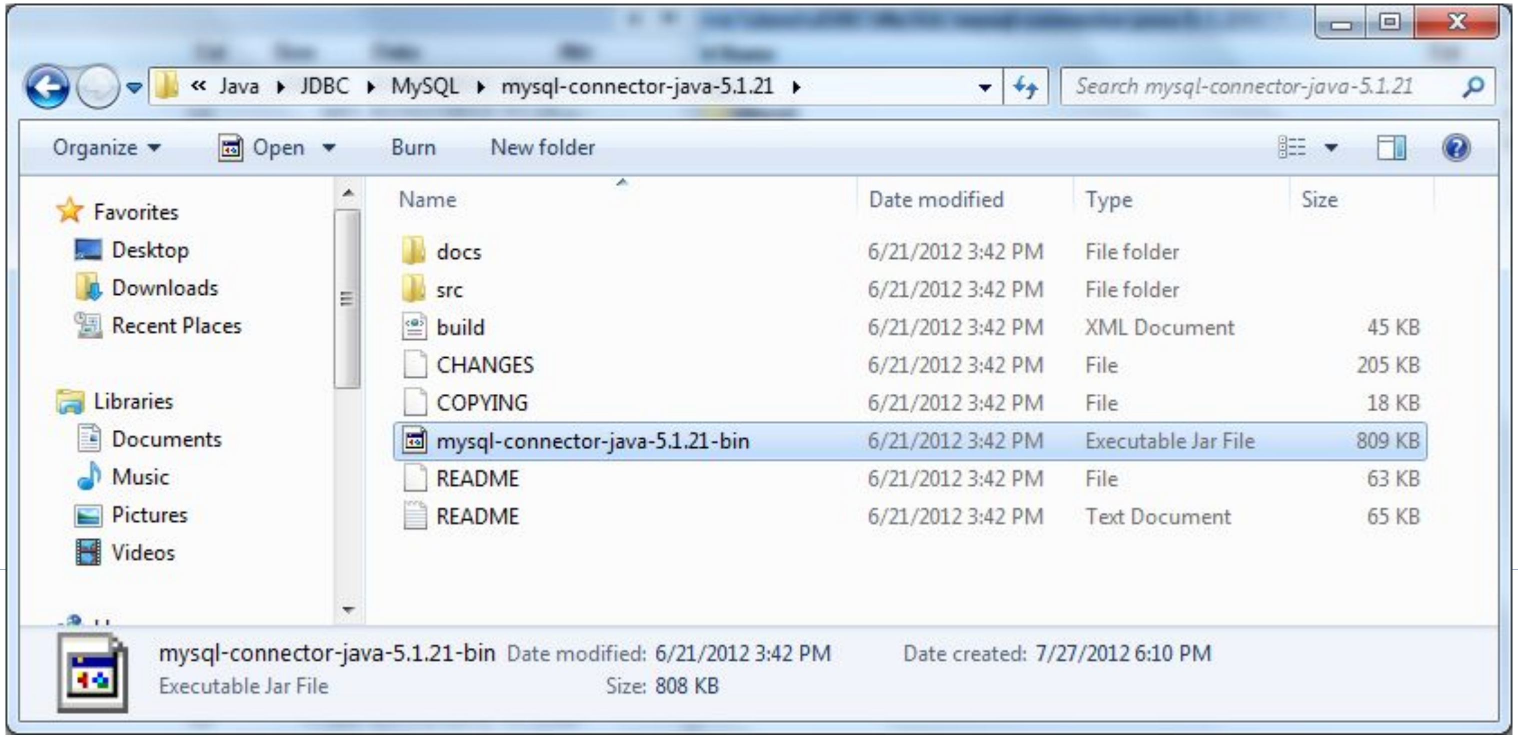
We suggest that you use the MD5 checksums and GnuPG signatures to verify the integrity of the packages you download.

Click on the **Download** button next to ***Platform Independent (Architecture Independent), ZIP Archive*** to download a zip archive.

MySQL Java Database Connectivity Driver

continued

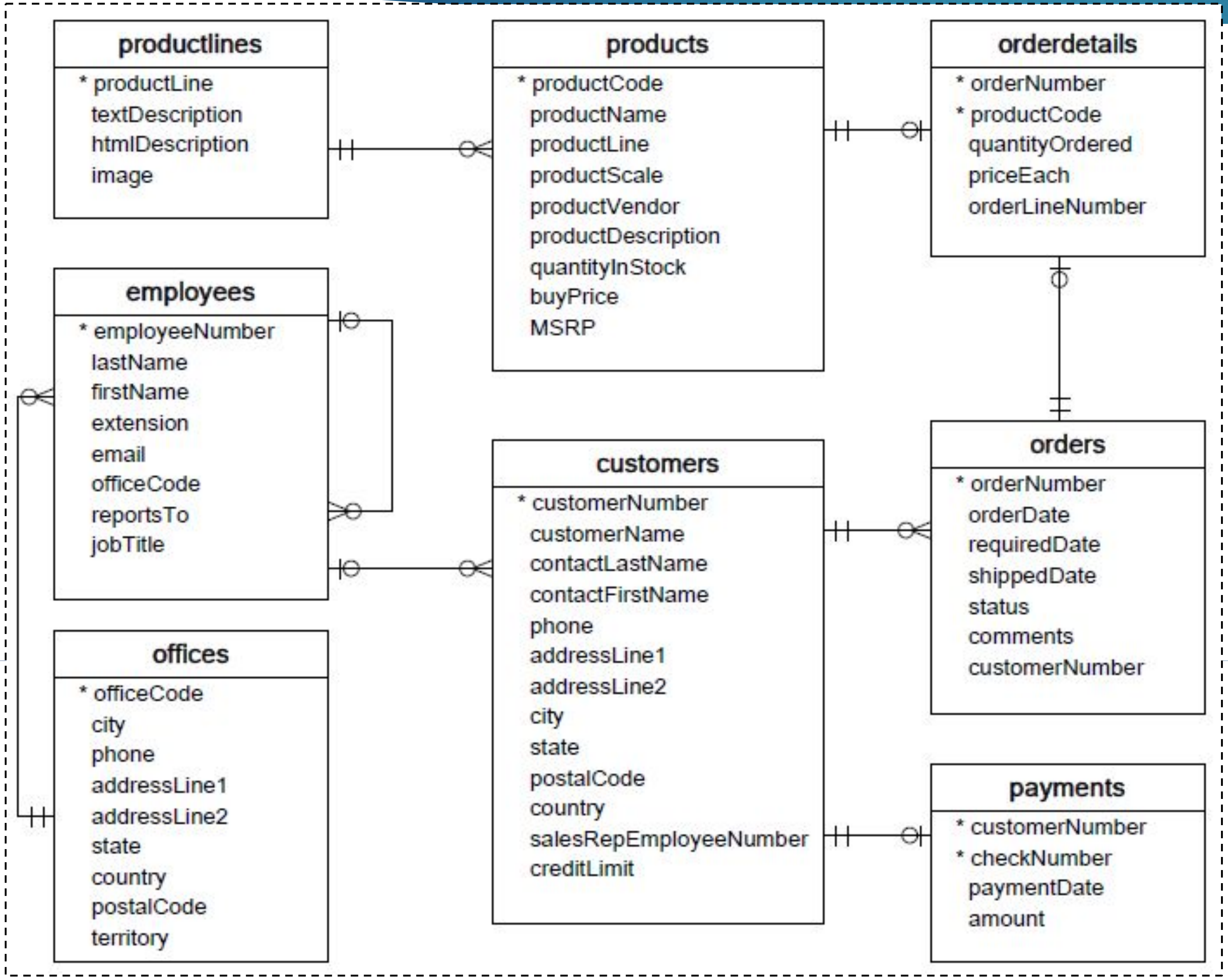
- **Step 2**: Extract the ZIP file to a desired location on your computer.



Name	Date modified	Type	Size
docs	6/21/2012 3:42 PM	File folder	
src	6/21/2012 3:42 PM	File folder	
build	6/21/2012 3:42 PM	XML Document	45 KB
CHANGES	6/21/2012 3:42 PM	File	205 KB
COPYING	6/21/2012 3:42 PM	File	18 KB
mysql-connector-java-5.1.21-bin	6/21/2012 3:42 PM	Executable Jar File	809 KB
README	6/21/2012 3:42 PM	File	63 KB
README	6/21/2012 3:42 PM	Text Document	65 KB

mysql-connector-java-5.1.21-bin Date modified: 6/21/2012 3:42 PM Date created: 7/27/2012 6:10 PM
Executable Jar File Size: 808 KB

Prerequisite



We will use the **classicmodels** database as an SQL sample database for the J**DBC demonstration** to help your work with JDBC - SQL effectively. The **classicmodels** database is a retailer of scale models of classic cars. It contains typical business data such as customers, products, sales orders, and sales order line items, etc. See the schema diagram to the left.

IF you do not have classicmodels database. You can find Lab - 304.1.2 - Download Classicmodels" Database" on Canvas under the Guided Lab section.

Java Database Connectivity Steps

1. Load database driver.
2. Open the database connection.
   a. Database URL.
   b. Username.
   c. Password.
3. Send the statement to the database.
   a. Use **Statement** Class or **Preparedstatement** class.
4. Get the results from the database.
5. Close the Database connection.

Load Database Driver

Step 1-

The easiest way to load the database driver is to use ***Class.forName("com.mysql.jdbc.Driver")*** on the class that implements the java.sql.Driver interface. Before Java 6, we hade to load the driver explicitly by this statement:

```
Class.forName("com.mysql.jdbc.Driver");
```

This statement is no longer needed thanks to the new update in JDBC 4.0, which is included in Java 6. As long as you put the MySQL JDBC driver JAR file into your program's classpath, the driver manager can find and load the database driver.

Open Database Connection

Step 2-

- JDBC driver classes include the ***DriverManager*** class, this class provides ***getConnection() method*** for obtaining a connection instance.
- There are three different signatures of the method getConnection(), which we can use:

```
○   static Connection getConnection(String url)
○   static Connection getConnection(String url, Properties info)
○   static Connection getConnection(String url, String dbuser, String dbpassword)
```

- All three versions have a parameter called *url*, which is the database URL.
- If a connection was made successfully with the database, the getConnection() method returns an instance of ***Connection class,*** which will be used to make queries and perform other database operations.

Open Database Connection

continued

Step 2 (continued)-

JDBC includes a Connection class: **A Java object that represents a unique connection to a database; often opened by using the methods of the DriverManager class before you can read/write data from/to a database using JDBC:**

- ❑ Allows for storing Database Connection as a variable.
- ❑ Creates a connection via *DriverManager*.

Example:

```java
public static void main(String[] args) throws SQLException {
   String dburl = "jdbc:mysql://localhost:3306/classicmodels";
   String user = "root";
   String password = "password";
   try {
      Class.forName("com.mysql.jdbc.Driver"); // optional
      Connection connection = DriverManager.getConnection(dburl, user, password);
   }
   catch(SQLException e) {
       e.printStackTrace();
   }
}
```

Sending SQL to the Database

Step 3-

JDBC Statement:

- The JDBC Statement is an interface used to execute an SQL query within the database.
- **A Java object that can be used to execute either database queries or database updates when using JDBC.**
  - DriverManager -> Connection -> Statement
- Example - Creating a Statement and executing a Query:

```
String SelectSQL = "Select * FROM employees";
Statement stmt = conn.createStatement();
ResultSet result =  stmt.executeQuery(SelectSQL);
```

Common Methods of Statement Interface

Commonly used methods:

- **boolean execute(String sql)**: executes a general SQL statement. It returns true if the query returns a ResultSet, and false if the query returns an update count or nothing. This method can be used with a Statement only.
- **int executeUpdate(String sql)**: executes an INSERT, UPDATE, or DELETE statement and returns an update account indicating the number of rows affected (e.g., 1 row inserted, 2 rows updated, or 0 rows affected).
- **ResultSet executeQuery(String sql)**: executes a SELECT statement and returns a ResultSet object, which contains results returned by the query.

Response From Database

Step 4-

What is a JDBC ResultSet?

- **A Java object that can be created by executing a database query using JDBC that contains records returned by the execution of that query.**
- Use this object to iterate over rows in the ResultSet using **next()** method, and get the value of a column in the current row using **getXXX()** methods (e.g. **getString()**, **getInt()**, **getFloat()** and so on). The column value can be retrieved either by index number (1-based) or by column name.

Response From Database

Step 4 (continued)-

Here is an example of executing a query against a database via JDBC:

```
ResultSet result = stmt.executeQuery(SelectSQL);
while(result.next())
     {
     String name = result.getString("firstName");
     String email = result.getString("email");
     System.out.println(name +" | " + email);
 }
```

- JDBC ResultSet Class object.
  - Returned from Statement.executeQuery().
  - List of output data.
  - Separated into columns.
- Looping through data.
  - ResultSet.next().
- ResultSet.get[Type]().
  - Use column number (starts at 1).
  - Use column header (i.e.,: “student_id”).
  - Replace [Type] with actual type.
    - getString(), getInt(), etc.

Close Database Connection

Step 5-

When you have completed the JDBC database connection, close the connection again. A JDBC connection can use a lot of sources inside your application and inside the database server. Therefore, it is important to close the database connection again after use. You close a JDBC connection via its **`close()`** method. Here is an example of closing a JDBC connection:

```
connection.close();
```

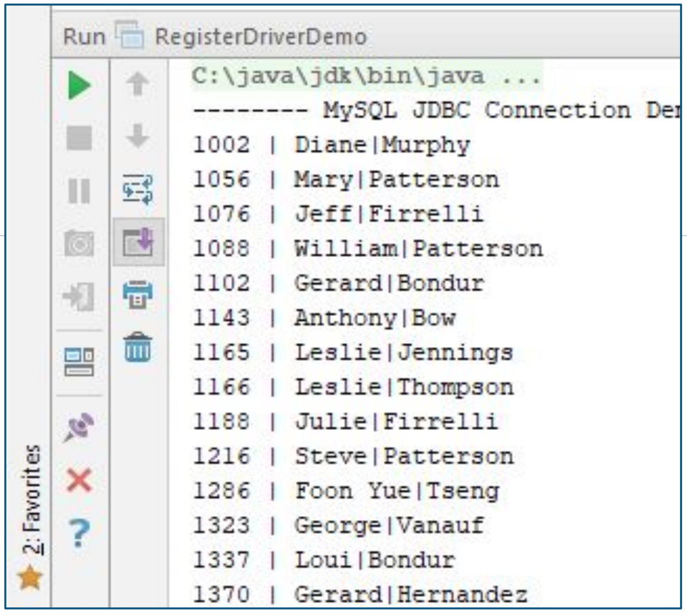
Why close a JDBC connection?

- If not closed, it would lead to memory leaks.
- The connection may remain active even after the user logs out.
- The database server would not be able to provide connections for new requests after it reaches its maximum number of simultaneous connections.
- It would lead to slow performance.
- Eventually, the database server will crash.

Example 1: Java Connection With MySQL

Create a class named ***DemoJDBC***, and write the code in the class.

Result:



```
C:\java\jdk\bin\java ...
-------- MySQL JDBC Connection De
1002 | Diane|Murphy
1056 | Mary|Patterson
1076 | Jeff|Firrelli
1088 | William|Patterson
1102 | Gerard|Bondur
1143 | Anthony|Bow
1165 | Leslie|Jennings
1166 | Leslie|Thompson
1188 | Julie|Firrelli
1216 | Steve|Patterson
1286 | Foon Yue|Tseng
1323 | George|Vanauf
1337 | Loui|Bondur
1370 | Gerard|Hernandez
```

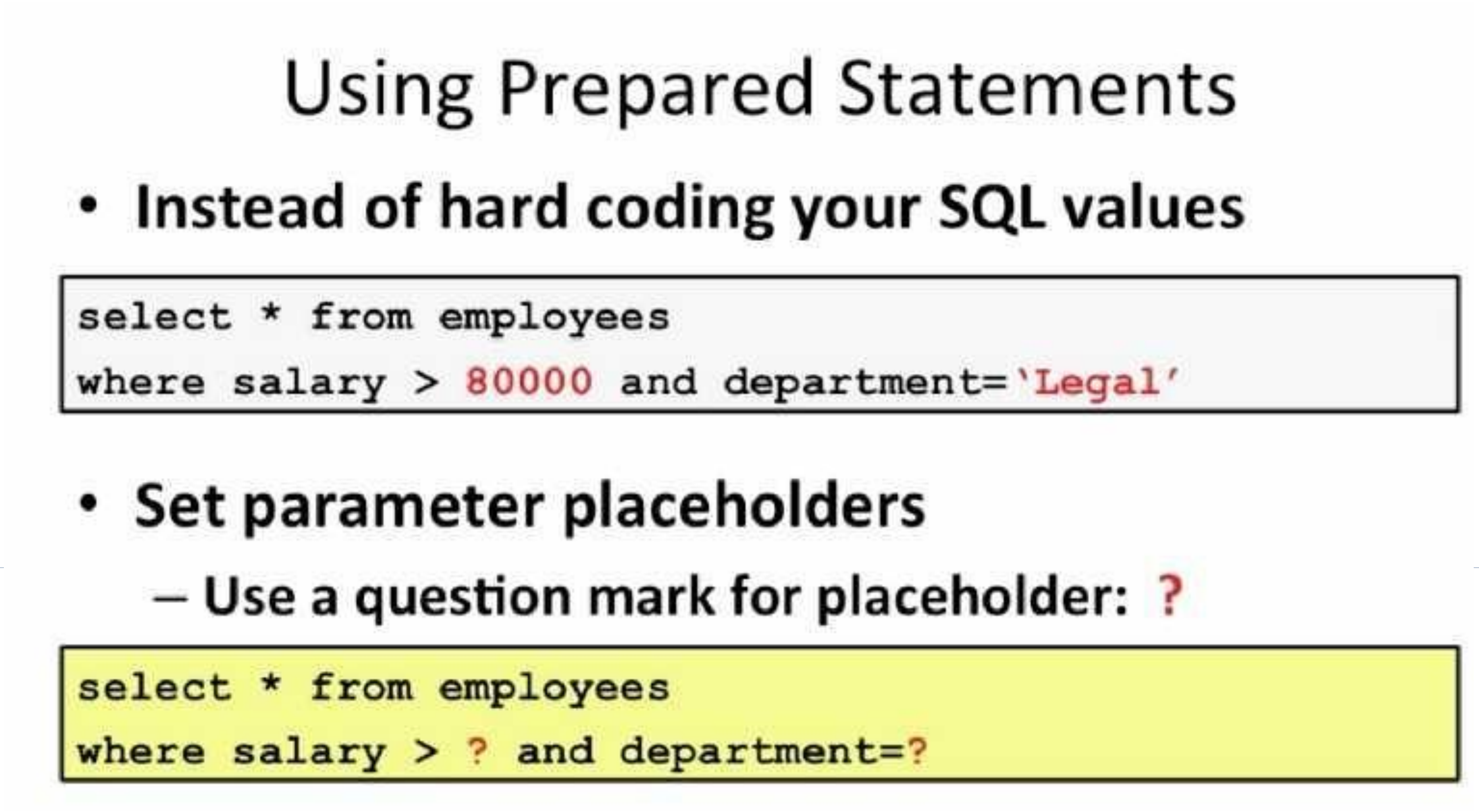
```java
import java.sql.*;
public class DemoJDBC {
   public static void main(String[] args) throws ClassNotFoundException {
       String dburl = "jdbc:mysql://localhost:3306/classicmodels";
       String user = "root";
       String password = "password";
       System.out.println("-------- MySQL JDBC Connection Demo ------------");
       try {
           Class.forName("com.mysql.jdbc.Driver");
           Connection connection = DriverManager.getConnection(dburl, user, password);
           String SelectSQL = "Select * FROM employees";
           Statement stmt = connection.createStatement();
           ResultSet result =  stmt.executeQuery(SelectSQL);
           while(result.next())
           {
              String EmployeeID  = result.getString("employeeNumber");
              String fname = result.getString("firstName");
              String lname  = result.getString("lastName");
              System.out.println(EmployeeID +" | " + fname + "|"+ lname );
           }
           connection.close();
       }
       catch(SQLException e) {
           e.printStackTrace();
       }
   }
}
```

Overview of PreparedStatements

A JDBC PreparedStatement is a special kind of ***Java JDBC Statement*** object with some useful added features. Remember, you need a statement in order to execute either a query or an update. You can use a Java JDBC PreparedStatement instead of a statement, and benefit from its added features:

- Makes it easier to set SQL parameter values.
- Accepts a query in constructor.
- Prevents SQL dependency injection attacks (can safely insert variables into SQL).
- Improves application performance. (when used for multiple executions of the same query, the performance of the PreparedStatement interface is better than the Statement interface).
- Includes precompiled SQL statements (you can pass parameters to your SQL query at run time).

Using Prepared Statements



Using Prepared Statements

- Instead of hard coding your SQL values

```
select * from employees
where salary > 80000 and department='Legal'
```

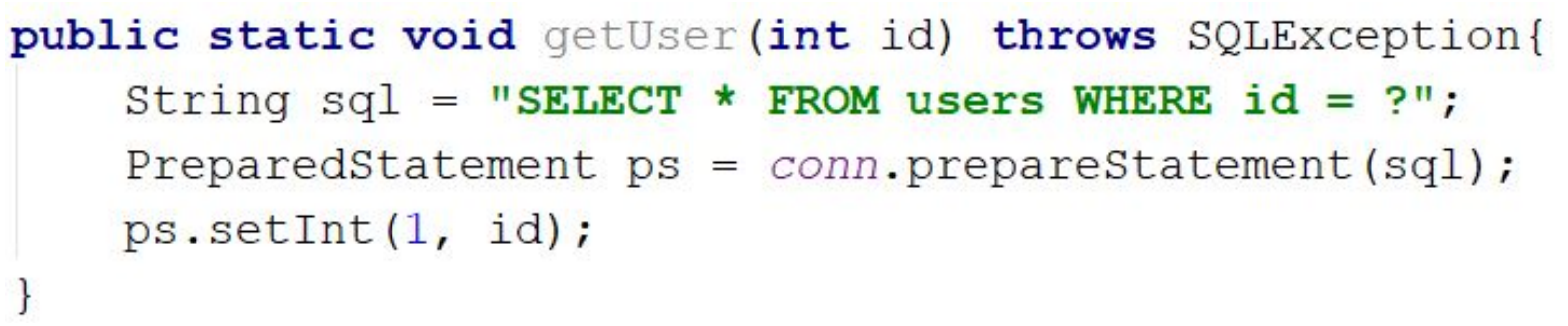
- Set parameter placeholders
  - Use a question mark for placeholder: ?

```
select * from employees
where salary > ? and department=?
```

Adding Placeholders to the Statement

- PreparedStatement set variables:
  - Use PreparedStatement.set[Type].
  - Replaces ? with data.
  - Accepts position and data.

- Creating PreparedStatement:



```
public static void getUser(int id) throws SQLException{
    String sql = "SELECT * FROM users WHERE id = ?";
    PreparedStatement ps = conn.prepareStatement(sql);
    ps.setInt(1, id);
}
```

Methods of PreparedStatement Interface

The important methods of PreparedStatement Interfaces:

Method	Description
public void setInt(int paramIndex, int value)	Sets the integer value to the given parameter index.
public void setString(int paramIndex, String value)	Sets the String value to the given parameter index.
public void setFloat(int paramIndex, float value)	Sets the float value to the given parameter index.
public void setDouble(int paramIndex, double value)	Sets the double value to the given parameter index.
public int executeUpdate()	The executeUpdate(String SQL) method is most often used to execute DML statements (INSERT, UPDATE, and DELETE), and it returns an *int* value, representing the count of the rows affected by its execution.
public ResultSet executeQuery()	The executeQuery(String SQL) method is used to retrieve data from the database by executing a DQL(SELECT) statement and it returns a ResultSet Object containing data requested by executed SQL statement.

Example 1: PreparedStatement

```
import java.sql.*;
import java.sql.PreparedStatement;
public class ExamplePrepareStat{
   public static void main (String[] args)throw Exception {
     Class.forName("com.mysql.jdbc.Driver");
     String url =  "jdbc:mysql://localhost:3306/classicmodels";
     final String USER = "root";
     final String PASS = "password";
     Connection conn = DriverManager.getConnection(url, "root",
     "password");
     String SelectSQL = "select * from employees where employeeNumber = ?";
     PreparedStatement mystmt = conn.prepareStatement(SelectSQL);
     mystmt.setInt(1, 1002);
      ResultSet result = mystmt.executeQuery();

          while(result.next())
                 {
                   String name = result.getString("firstName");
                   String email  = result.getString("email");
                   System.out.println(name +" | " + email);
                 }
     }
}
```

In this example, we will demonstrate how to use Prepared Statements to get employee information by using **employee id.**

PreparedStatement With DML

We can use PreparedStatement for:

- Insert statements.
- Update statements.
- Delete statements.

Example 1: **PreparedStatement with Insert Statement**

```
String sqlQuery = "INSERT INTO EMPLOYEES
(officeCode,firstName,lastName,email,extension,reportsTo,VacationHours,employeeNumber,jobTitle) VALUES (?,?,?,?,?,?,?,?,?)";
prepStmt = con.prepareStatement(sqlQuery);
prepStmt.setInt(1, 6);
prepStmt.setString(2, "Jamil");
prepStmt.setString(3, "fink");
prepStmt.setString(4, "JJ@gmail.com");
prepStmt.setString(5, "2759");
prepStmt.setInt(6, 1143);
prepStmt.setInt(7, 9);
prepStmt.setInt(8, 0003);
prepStmt.setString(9, "Manager");
int affectedRows = prepStmt.executeUpdate();
System.out.println(affectedRows + " row(s) affected !!");
```

Example 2: PreparedStatement with Update Statement

```
String SelectSQL = "update employees set firstName=? , lastName=? where
employeeNumber=?";

PreparedStatement mystmt = conn.prepareStatement(SelectSQL);

mystmt.setString(1, "Gary");

mystmt.setString(2, "Larson");

mystmt.setLong  (3, 1002);

mystmt.executeUpdate();
```

Hands-On Lab

Click here for LAB 305.1.2 - JDBC - PreparedStatement with DML and DDL.

If you have any technical questions while performing the lab activity, ask your instructors for assistance.

Separate Class for Queries

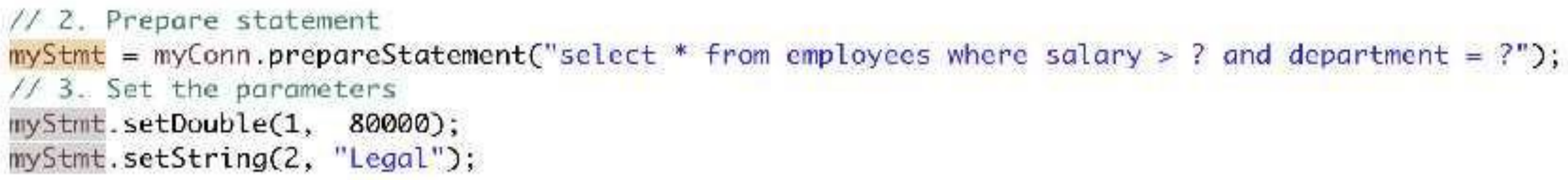
- It is a traditional practice to store all of the SQL queries in a separate class. That class usually contains public, static final string fields that store the queries in a variable.
- That variable is called later in the PreparedStatement in the class that performs the functionality.

Examples:

```
pubic class SqlQuries {
    public final static String GetEmployeByID="select * from employees where
    employeeNumber=? ";
       public final static String GetEmployeBySalalry= "select * from employees
    where salary = ?";
    }
```

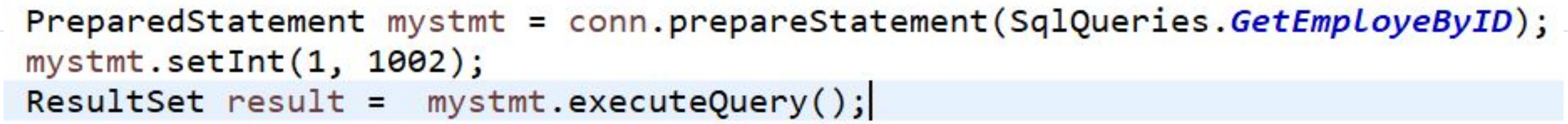
Separate Class for Queries (continued)

- WITHOUT the Separate Class



```
// 2. Prepare statement
myStmt = myConn.prepareStatement("select * from employees where salary > ? and department = ?");
// 3. Set the parameters
myStmt.setDouble(1,  80000);
myStmt.setString(2, "Legal");
```

- WITH the Separate Class



```
PreparedStatement mystmt = conn.prepareStatement(SqlQueries.GetEmployeByID);
mystmt.setInt(1, 1002);
ResultSet result =  mystmt.executeQuery();
```

Hands-On Activity *(optional)*

Create SQL PreparedStatements that would perform the following functionalities. You can use “`classicmodels`” database:

- Select all employees whose **officecode** is 1, 4.
- Select all **orderdetails** whose **orderNumber** is 10100,and 10102.
- Update the extension number of employees whose **officecode** is 2, and new extension number will be “5698.”
- Select all employees whose last name is less than 5 characters in length.

Summary

JDBC (Java Database Connectivity) is the Java API that manages connecting to a database, issuing queries and commands, and handling result sets obtained from the database.

The steps for connecting to a database with JDBC are as follows:

1. Install or locate the database you want to access.
2. Include the JDBC library.
3. Ensure that the JDBC driver you need is on your classpath.
4. Use the JDBC library to obtain a connection to the database.
5. Use the connection to issue SQL commands.

Both *Statement* and *PreparedStatement* can be used to execute SQL queries. These interfaces look very similar; however, they differ significantly from one another in features and performance:

- *Statement* – **Used to execute string-based SQL queries.**
- *PreparedStatement* – **Used to execute parameterized SQL queries.**

References

https://docs.oracle.com/javase/6/docs/api/java/sql/Driver.html

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

End of Section