Working with Databases using JDBC

Student Workbook

Version 1.2

Table of Contents

Module 1

Updating the Maven pom.xml

Section 1–1

The pom.xml

The pom.xml file in IntelliJ

- **The Project Object Model (pom) file is an XML file that contains metadata and configuration details for building the project**

- **Common things to modify in the pom.xml**

  - Changing the target Java version

  Adding or changing dependencies for your project
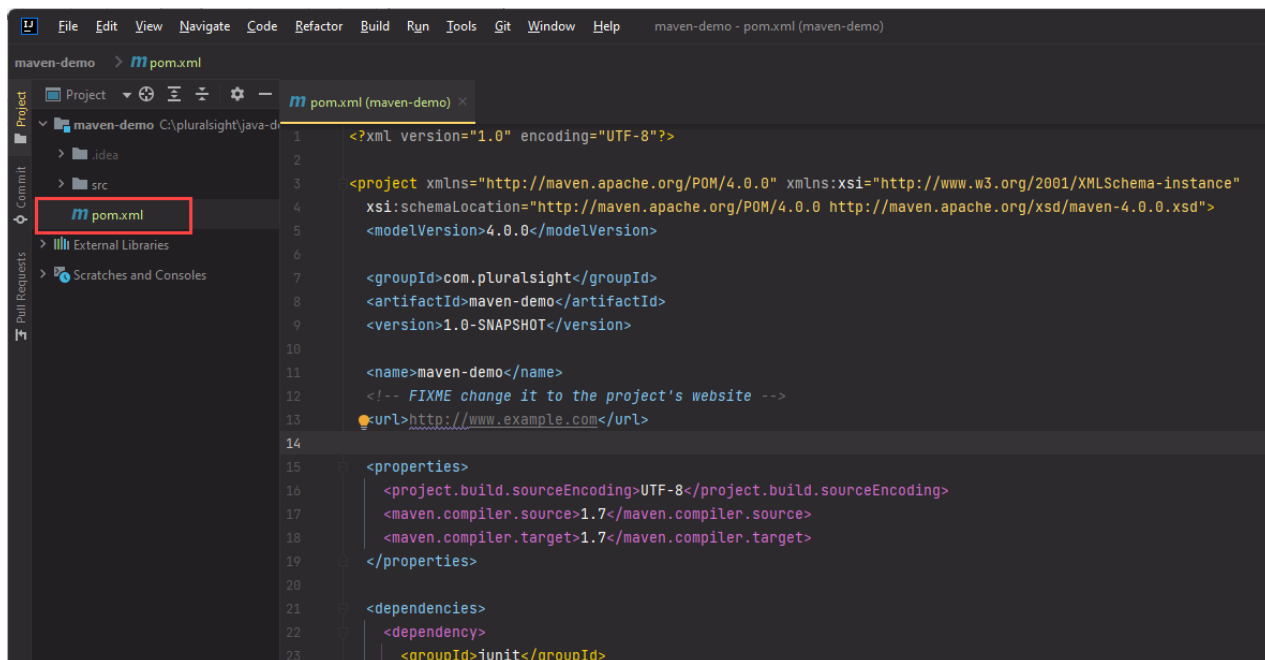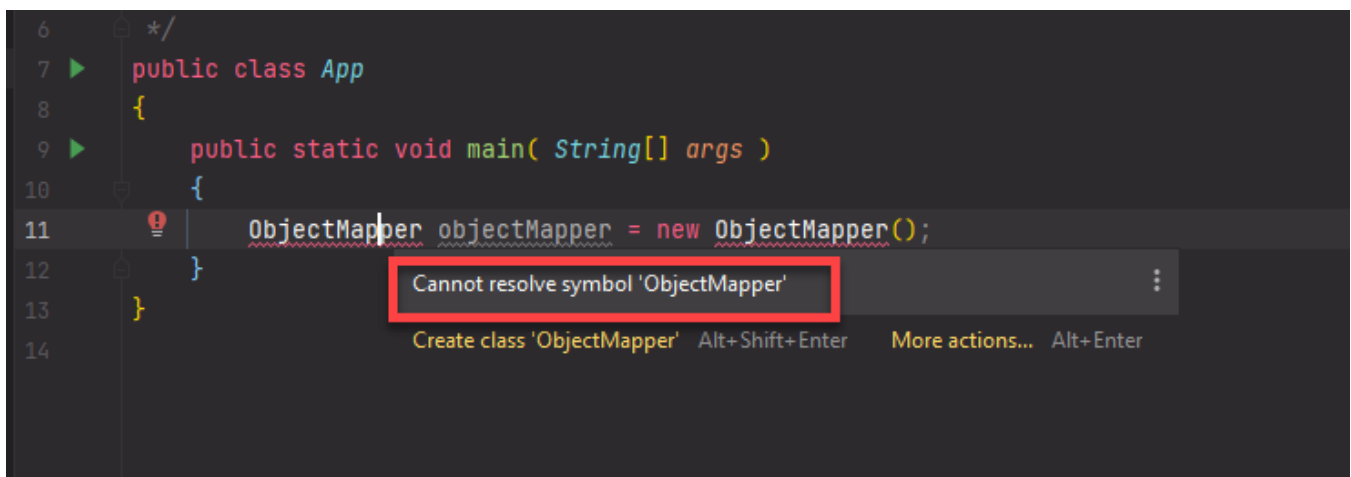
- **When you make a change to your `pom.xml` file you need to reload the maven dependencies**
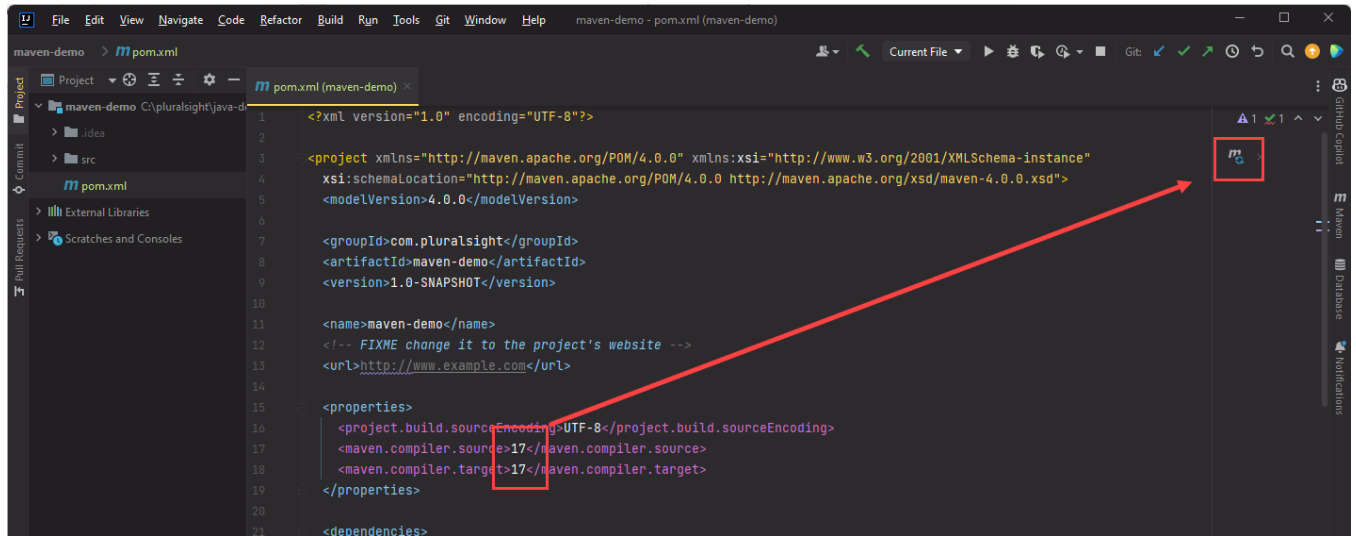
When to add dependencies

- **Whenever you want to use a third-party library in your project, you'll need to make sure this is added to your dependency list.**

- **For example, when we want to use the Jackson ObjectMapper, we need to add the Jackson dependency**

```
 6     */
 7  ▶  public class App
 8     {
 9  ▶      public static void main( String[] args )
10         {
11  ⚡          ObjectMapper objectMapper = new ObjectMapper();
12         }         Cannot resolve symbol 'ObjectMapper'                    ⋮
13     }
14             Create class 'ObjectMapper'  Alt+Shift+Enter    More actions...  Alt+Enter
```

- **It cannot find the package from where to import the library, because it is not added to the project**

How to add dependencies

- **We add a `<dependency>` node to our XML file in the `<dependencies>` node**

Example

NOTE: you need to know which external library contains the packages and objects that your project is dependent on

```
<dependency>
   <groupId>com.pluralsight</groupId>
   <artifactId>example</artifactId>
   <version>1.02</version>
</dependency>
```

- **After adding a dependency make sure that you reload all Maven dependencies**

- After loading dependencies you can expand the **External Libraries** folder and see the imported **jars**
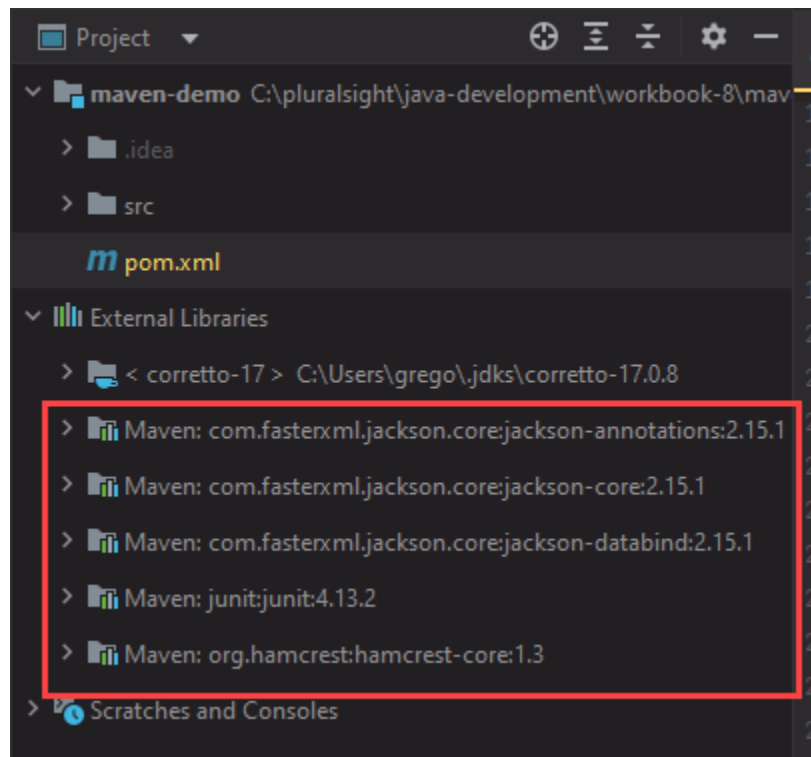
- Now we've added the dependency, the suggested fixes can find the import with our **ObjectMapper** class in it
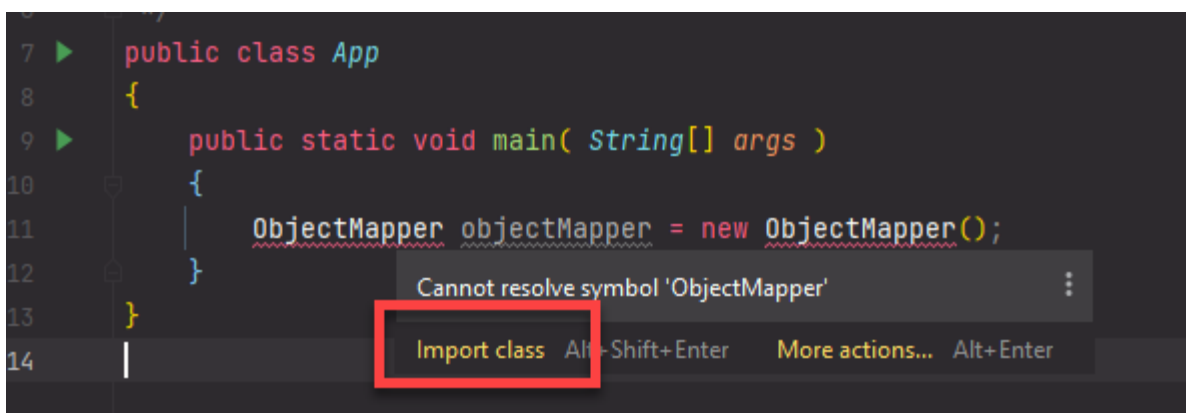
Exercises

Create a new project called LoggerExercise

EXERCISE 1

1. Add the following code snippet in the `App.java` file of your project (replace all other code except for the package statement)

```java
public class App{

    final static Logger logger = Logger.getLogger(App.class);

    public static void main(String[] args) {

            logMeLikeYouDo("☕");

    }

    private static void logMeLikeYouDo(String input){

            if(logger.isDebugEnabled()){
                    logger.debug("This is debug : " + input);
            }

            if(logger.isInfoEnabled()){
                    logger.info("This is info : " + input);
            }

            logger.warn("This is warn : " + input);
            logger.error("This is error : " + input);
            logger.fatal("This is fatal : " + input);

    }

}
```

2. Add the necessary dependency (log4j) to get the imports on top of your file that are needed to get this code to run. This will require some googling. (Do not use the default `java.util.logging` package)

3. Make sure the project is updated and run the code. The code will run, but you may not get the output you'd expect.

4. BONUS: configure the project such that you can use log4j. Hint: the proper way to do this is adding a resources folder with a .properties file.

5. The following `log4j.properties` file that you can give them and have them put in the `src/main/resources` folder.

```
# Root logger option

log4j.rootLogger=TRACE, console, file


# Direct log messages to the console

log4j.appender.console=org.apache.log4j.ConsoleAppender

log4j.appender.console.target=System.out

log4j.appender.console.layout=org.apache.log4j.PatternLayout

log4j.appender.console.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c{
1}:%L - %m%n


# Direct log messages to a log file

log4j.appender.file=org.apache.log4j.DailyRollingFileAppender

log4j.appender.file.File=log/application.log

log4j.appender.file.DatePattern='.'yyyy-MM-dd

log4j.appender.file.layout=org.apache.log4j.PatternLayout

log4j.appender.file.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:
%L - %m%n
```

Section 1–2

CodeWars

CodeWars Kata

- **Length and Two Values**

  – Given 3 inputs, create an array with the correct values

- **Complete this Kata for additional Java practice**

  – https://www.codewars.com/kata/62a611067274990047f431a8/java

Module 2

JDBC Basics

Section 2–1

Overview of JDBC

Java Applications and Databases

- **Java Applications allow users to interact with computers**

  – They can be simple or complex

  ```
  Enter a number (or "done"): 15
  Enter a number (or "done"): 23
  Enter a number (or "done"): -19
  Enter a number (or "done"): 84
  Enter a number (or "done"): 5
  Enter a number (or "done"): done
  -------------------------------------------
                              Total: 108
  ```

- **Databases store data**

  – Information about cities

ID	Name	CountryCode	District	Population
1	Kabul	AFG	Kabol	1780000
2	Qandahar	AFG	Qandahar	237500
3	Herat	AFG	Herat	186800
4	Mazar-e-Sharif	AFG	Balkh	127800
5	Amsterdam	NLD	Noord-Holland	731200
6	Rotterdam	NLD	Zuid-Holland	593321
7	Haag	NLD	Zuid-Holland	440900
8	Utrecht	NLD	Utrecht	234323
9	Eindhoven	NLD	Noord-Brabant	201843
10	Tilburg	NLD	Noord-Brabant	193238
11	Groningen	NLD	Groningen	172701
12	Breda	NLD	Noord-Brabant	160398

- **We want to use that data in our applications**

  – Applications are most useful when they connect users to data

```
What would you like to do?
  1) List cities
  2) Add a city
  0) Exit
Please make a selection: 1

Cities
----------------------
1: Kabul - AFG
2: Qandahar - AFG
3: Herat - AFG
4: Mazar-e-Sharif - AFG
5: Amsterdam - NLD
```

JDBC

- **JDBC (Java database connectivity) is the specification or API that lets Java programs to access database management systems**

- **The JDBC API consists of a set of interfaces and classes you will need as a programmer to do things like:**

  - connect to the database

  - execute a query

  - process the results

- **Since JDBC is a standard specification, a Java program that uses the JDBC can connect to any database management system, as long as they have a *driver* for that DBMS**

- **The JDBC driver's job is to provide a connection to the database**

  - It also implements the protocol for transferring the query and its results between Java application and the DBMS

Driver

- **A *driver* is implemented in an external library that you generally have to include in your *classpath***

  – It can also be imported as a Maven dependency in the pom.xml file

- **For example, to connect to MySQL you will need to find a MySQL library to include into your project**

- **You may have installed the MySQL driver as part of your MySQL installation**

  – If not, the MySQL driver is located here: `https://dev.mysql.com/downloads/connector/j/`

  – **Note:** This is the downloaded local jar package

    * It is also common to install the dependency with Maven in the `pom.xml` file

- **There are two ways you can connection to a database:**

  – `java.sql.DriverManager` — older way

  – `java.sql.DataSource` — newer, and the recommended way

Using Maven to Load the Driver

- **When you create a new Java project, you may need to download and import external dependencies**

- **The MySQL Driver is located in a jar file that is published in the Maven registry**

- **You just need to update the `pom.xml` with the following dependency**

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.33</version>
</dependency>
```
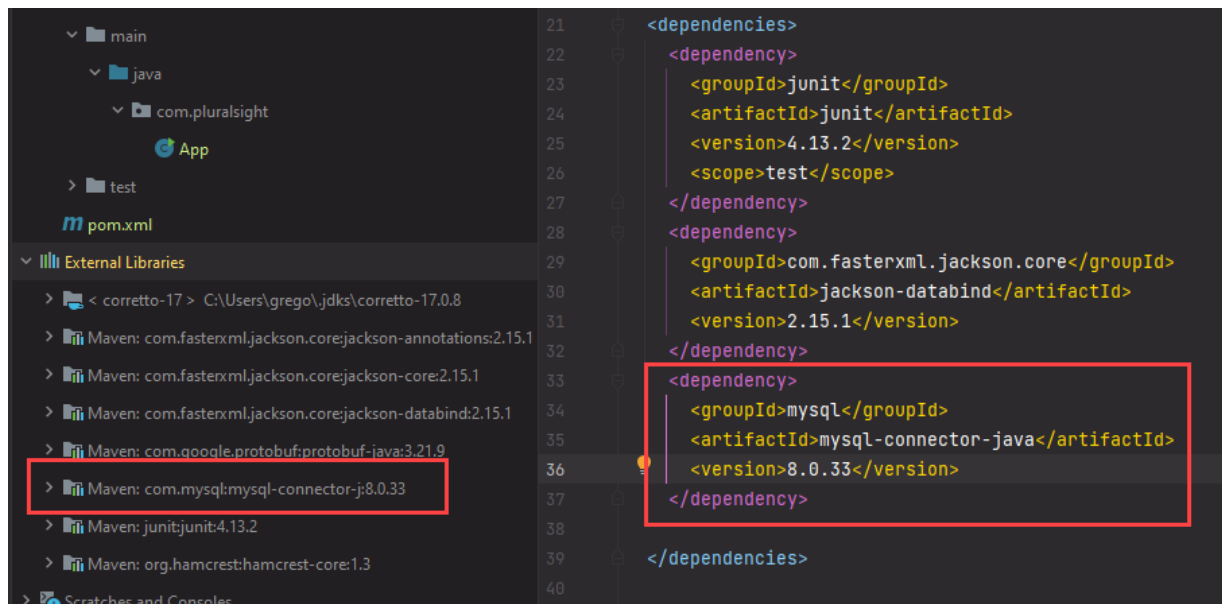
JDBC and MySQL Imports

- **Typically, you will need 3 imports**

  – The MySQL driver will be one of the two below depending on the version

    * you can check it after you add the JAR file to your project

  – See: https://dev.mysql.com/doc/connector-j/8.0/en/connector-j-api-changes.html

  ### Example

  ```
  import com.mysql.cj.jdbc.MysqlDataSource;

  import com.mysql.jdbc.MysqlDataSource;
  ```

  – The MySQL classes are available by importing `java.sql.*` and if you choose to use the new `DataSource` approach, it is available by importing `javax.sql.*`

  ### Example

  ```
  import java.sql.*;
  import javax.sql.DataSource;
  ```

- **But, remember IntelliJ can also auto generate the necessary import statements as you write your code**

The Database URL

- **A database URL has the following format:**

```
jdbc:mysql://[host][:port]/[database][?propertyName1=propertyValue1]
```

- **Most database URLs have similarities:**

  - They start with the protocol **jdbc**

  - They are followed by the database engine

    * mysql, oracle, sqlserver, postgresql

  - Then they have the address of the database

  - Finally, you may provide optional properties

- **For example, to connect to a MySQL database you would use a URL similar to the following**

Example

```
jdbc:mysql://localhost:3306/sakila
```

- **To confirm your MySQL configuration, go to the MySQL Workbench and choose Server -> Server Status**

Working with JDBC Objects

- **Using JDBC typically requires the following objects:**

    – Create either `DriverManager` or `DataSource` to access the MySQL JDBC driver

    * Or another driver if you are working with a different database (Postgres, Oracle, MS SQL Server, etc)

    – Open a `Connection` to your database

    – Use a `Statement` (or `PreparedStatement`) to execute your query on the database

    – If your query returns results, use `ResultSet` to iterate through each row of the result

JDBC Programming Process

- **Database interactions follow these steps:**

  - These steps require the JDBC objects previously mentioned

    * Open a connection to the database

    * Execute your query (and process the result)

    * Close the connection

Example

```java
// load the MySQL Driver
Class.forName("com.mysql.cj.jdbc.Driver");

// 1. open a connection to the database
// use the database URL to point to the correct database
Connection connection;
connection = DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/world",
                username,
                password);

// create statement
// the statement is tied to the open connection
Statement statement = connection.createStatement();

// define your query
String query = "SELECT Name FROM city " +
                "WHERE CountryCode = 'USA'";

// 2. Execute your query
ResultSet results = statement.executeQuery(query);

// process the results
while (results.next()) {
  String city = results.getString("Name");
  System.out.println(city);
}

// 3. Close the connection
connection.close();
```

Exercise

Create a new folder in the `LearnToCode\Workbood8` **folder named** `Mod03`.

EXERCISE 1

Create a new maven project named **NorthwindTraders**. **DO NOT** use a Maven Archetype, but create a regular Java Maven project with a GroupId of`com.pluralsight`.
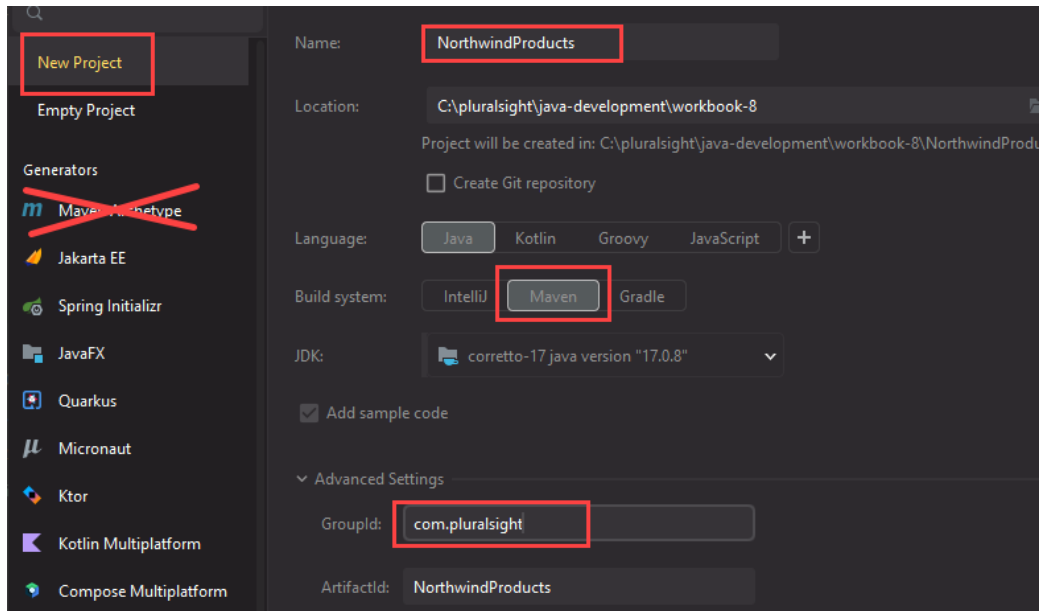
In your `pom.xml` file, add a dependency to the MySQL Driver.

In the `main` method connect to the Northwind database. (jdbc:mysql://localhost:3306/northwind)

Execute a query to select all products.

Display the name of each product sold by Northwind to the screen.

Commit and push your code!

Section 2–2

Coding with JDBC

Connecting with `DriverManager`

- **`DriverManager`** is the *classic (older) way* to connect to a database

- You use a method named **`Class.forName`** to load the JDBC driver into the VM's memory

- Then you use the **`DriverManager`** class to create a **`Connection`** object with a username and password to the database you want to interact with

- The **`Connection`** is a resource and should be closed when you are done with it

Example

```java
// uses eager loading to load the driver into memory
// if the dependency has not been added the application
// will fail here
Class.forName("com.mysql.cj.jdbc.Driver");

// use the driver to create a connection
Connection connection = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/sakila", username, password);
```

Statement vs `PreparedStatement`

- **A JDBC `Statement` object only supports basic SQL queries**

- **Most SQL queries require some form of user input**

  - Dynamically building a query through String concatenation is not considered safe

    * We will discuss SQL Injection attacks shortly

Example

```
String country = "USA";
String query = "SELECT Name FROM city " +
               "WHERE CountryCode = '" + country + "'";

Statement statement = connection.createStatement();
ResultSet results = statement.execute(query);
```

- **JDBC includes a safe alternative - the `PreparedStatement`**

  - The `PreparedStatement` does not require String concatenation

Example

```
String country = "USA";
String query = "SELECT Name FROM city " +
               "WHERE CountryCode = ?";

PreparedStatement statement =
               connection.prepareStatement(query);
statement.setString(1, country);
ResultSet results = statement.execute(sql);
```

Creating a `PreparedStatement` with the SQL Query

- **To submit a SQL query to the database through the `Connection`, create a `PreparedStatement`**

Example

```
// Looking for all customers
PreparedStatement preparedStatement = connection.prepareStatement(
   "SELECT first_name, last_name FROM customer");
```

- **`PreparedStatement` can also be used with parameters**

- **To add a parameter to the SQL statement, use a question mark ?**

  - You can have as many parameters as you need in the `PreparedStatement`

Example

```
// Looking for all customers with the last name
// that begins with a specified value
PreparedStatement preparedStatement = connection.prepareStatement(
   "SELECT first_name, last_name FROM customer " +
   "WHERE last_name LIKE ?");
```

Setting Parameter Values in a
`PreparedStatement`

- Once you've built the prepared statement, you must provide values for your parameters

- To add the value for that parameter, use the **`PreparedStatement`** object, and call **`setXXX()`**, where XXX refers to the type you wish to set, including:

  - `setString()`

  - `setInt()`

  - `setDouble()`

  - `setDate()`

  - `setBlob()`

- IMPORTANT: Parameters are identified using a **`1-based`** counting system

Example

```java
// Look for all customers with the last name
// that begins with "Sa" in order

PreparedStatement preparedStatement = connection.prepareStatement(
    "SELECT first_name, last_name FROM customer " +
    "WHERE last_name LIKE ?");

// the number 1 below specifies the first parameter
//    THIS STATEMENT USES 1-BASED COUNTING!!!
preparedStatement.setString(1, "Sa%");
```

SQL Injection: Always
Use `PreparedStatement`
Over `Statement`

- When you Google, you may see that Java has a `Statement` class

- We shouldn't use it!

- The `PreparedStatement` is much better at avoiding malicious SQL injection attacks

- What is a SQL injection attack? Attackers trick the SQL engine into executing unintended commands by supplying specially crafted string input

Example

```
String userid = /* value from user */

String sql = "SELECT * FROM Users WHERE UserId=" + userid;
```

- **A malicious user might enter: 20213 or 1 = 1**

Example

```
String userid = "20213 or 1=1";

String sql = "SELECT * FROM Users WHERE UserId=" + userid;
```

BECOMES THE QUERY:
```
SELECT * FROM Users WHERE UserId=20213 or 1=1
```

- **It gets worse…**

Example

```
String userName = "Robert');DROP TABLE students;--";

String sql = "SELECT * FROM Users WHERE (UserName='" +
             username + "')";
```

BECOMES THE QUERY:
```
SELECT * FROM Users WHERE (UserName='Robert');
DROP TABLE Students;--'
```


- **A `PreparedStatement` always treat parameter data as data and never as a part of an SQL statement**

Executing the Query

- **To run the SQL statement in the prepared statement, you can use `executeQuery()`**

  - It returns a `ResultSet` that contains the records that SQL selected

- **You can also use `executeUpdate()` if your `PreparedStatement` object contains a SQL statement like `INSERT INTO, UPDATE`, etc.**

  - It will return the number of rows that were updated

- **You can also use `execute()` to run both select and insert/update statements**

  - It returns a `boolean`

  - A true indicates that the execute method returned a result set object which can be retrieved using `getResultSet()` method

  - A false indicates that the query returned an `int` value or `void`

- **More in this soon**

Using a `ResultSet` to Extract the Results

- **A `ResultSet` encapsulates the result of your query**

- **You can use the `getXXX()` methods to get the result and convert it to the type specified. They include:**

  - `getString()`

  - `getInt()`

  - `getDouble()`

  - `getDate()`

  - `getBlob()`

- **You can get the result field *by index* (1-based)**

Example

```java
// Query was:  SELECT first_name, last_name FROM ...
ResultSet resultSet = preparedStatement.executeQuery();

while (resultSet.next()) {

    // Get the 1st and 2nd fields returned from the query
    // based on the SELECT statement
     System.out.printf("first_name = %s, last_name = %s;\n",
        resultSet.getString(1),
        resultSet.getString(2));
}
```

- **You can also get the result by** *field name*

Example

```java
// Query was:  SELECT first_name, last_name FROM ...
ResultSet resultSet = preparedStatement.executeQuery();

while (resultSet.next()) {

    // Get the fields named first_name and last_name in the
    // SELECT statement
    System.out.printf("first_name = %s, last_name = %s;\n",
        resultSet.getString("first_name"),
        resultSet.getString("last_name"));
}
```

Always Close your Resources

- **Database connections are expensive, and you need to close those resources**

  – This means connection objects, prepared statement objects and result sets

- **You should close your resources in the reverse order that you created those resources**

- **For example, if you created a `Connection`, then a `PreparedStatement` and finally a `ResultSet`, close the results in the following order**

  – `ResultSet`

  – `PreparedStatement`

  – `Connection`

Example

```
resultSet.close();
preparedStatement.close();
connection.close();
```

Java and Exceptions

- **When a method in Java potentially throws an exception and the programmer doesn't surround it with a `try/catch` block, the Java compiler complains**

- **A programmer can declare in the method signature that the method might encounter an exception but that it doesn't handle it**

  - To do this, add a `throws` statement to the function declaration

- **We do this when testing a snippet of code and we don't want to worry about handling exceptions**

Example

```java
public static void main(String[] args) throws SQLException,
ClassNotFoundException {

}
```

- **We also do this to propagate errors to a calling method that has the `try/catch` around the call to the method**

Example

```java
public static void main(String[] args) {
   try {
      doQuery();
   }
   catch(Exception e) {
      e.printStackTrace();
   }
}
```

```java
public static void doQuery() throws SQLException {

}
```

Passing Username and Password on the Command Line

- **In the examples that follow, we will pass the username and password to the database in on the command line**

- **In IntelliJ, you can pass input parameters to the application by editing configurations**
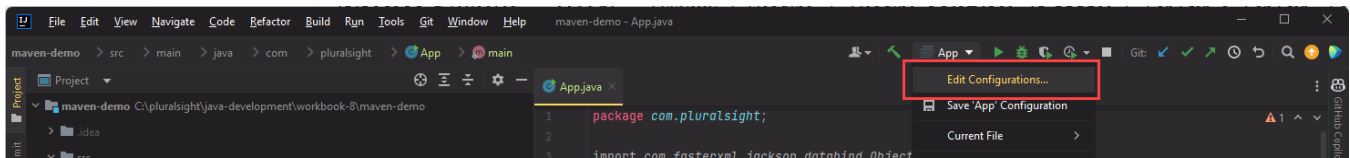
- **You can "hard code" your input parameters in the program arguments textbox**
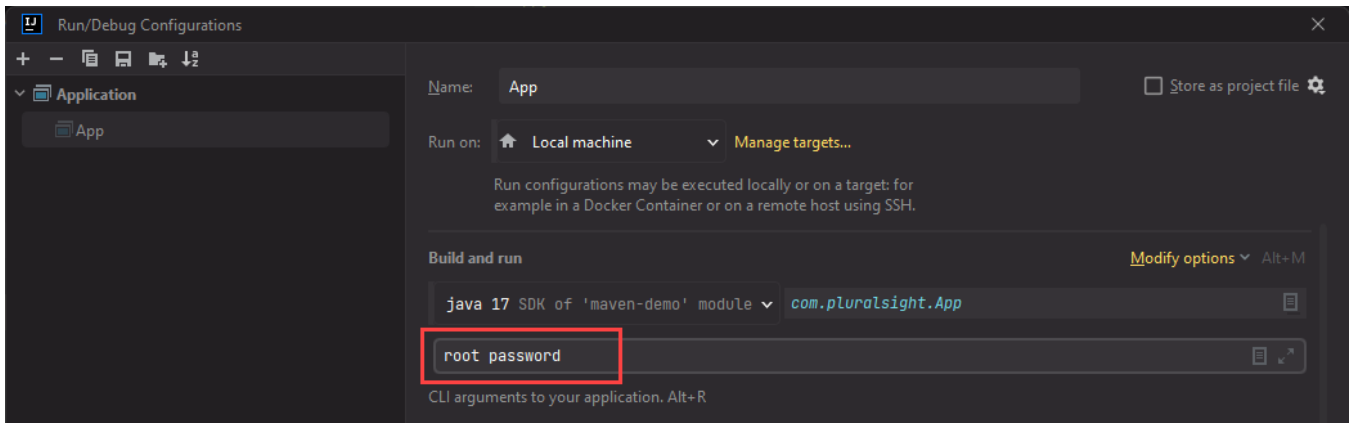
Example: Executing a Query

```java
package com.pluralsight;

import java.sql.*;

public class UsingDriverManager {
  public static void main(String[] args) throws SQLException,
  ClassNotFoundException {

    if (args.length != 2) {
      System.out.println(
        "Application needs two arguments to run: " +
        "java com.pluralsight.UsingDriverManager <username> <password>");
      System.exit(1);
    }

    // get the user name and password from the command line args
    String username = args[0];
    String password = args[1];

    // load the driver
    Class.forName("com.mysql.cj.jdbc.Driver");

    // create the connection and prepared statement
    Connection connection = DriverManager.getConnection(
      "jdbc:mysql://localhost:3306/sakila", username, password);

    PreparedStatement preparedStatement =
      connection.prepareStatement(
      "SELECT first_name, last_name FROM customer " +
      "WHERE last_name LIKE ? ORDER BY first_name");

    // set the parameters for the prepared statement
    preparedStatement.setString(1, "Sa%");

    // execute the query
    ResultSet resultSet = preparedStatement.executeQuery();

    // loop thru the results
    while (resultSet.next()) {

      // process the data
      System.out.printf("first_name = %s, last_name = %s;\n",
        resultSet.getString(1), resultSet.getString(2));
    }

    // close the resources
    resultSet.close();
    preparedStatement.close();
    connection.close();
  }
}
```

Exercises

EXERCISE 2

Continue working on the NorthwindTraders in the previous exercise.

Modify your code to display all Products in the Northwind database. The list must include:
- product id
- product name
- unit price
- units in stock

Consider how you will display each product. Here are a couple of options:

Option 1: Stacked Information

```
Product Id: 1
Name:       Chai
Price:      18.00
Stock:      396

------------------

Id:         2
Name:       Chang
Price:      19.00
Stock:      17

------------------
```

Option 2: Rows of Information

```
Id   Name            Price   Stock
---- -------------- ------- ------
1    Chai            18.00   39
2    Chang           19.00   17
```

Section 2–3

Handling Exceptions

`try / catch / finally`

- **The code in the previous exercise worked, but if an exception occurred, it might not get the resources closed**

- **The `try/catch` that you have seen in the past has a third sections that you can include**

  - `try` is the block that you try to execute

  - `catch` is the block that executes if the try encountered an exception

  - `finally` always executes -- regardless of whether the catch handles an exception

- **`finally` is where you make sure you dispose of, or close, resources**

Example

```
// Establish the variables with null outside the try scope
// so they are available in the catch and finally blocks
Connection connection = null;
PreparedStatement preparedStatement = null;
ResultSet resultSet = null;

try {
    // work with the database
}
catch (SQLException e) {
    e.printStackTrace();
}
finally {
    if (resultSet != null) resultSet.close();
    if (preparedStatement != null) preparedStatement.close();
    if (connection != null) connection.close();
}
```

Example: Executing a Query - Includes `try-catch` Blocks

```java
package org.yearup.jdbc;

import java.sql.*;

public class UsingDriverManager {

  public static void main(String[] args) throws SQLException,
  ClassNotFoundException {

     if (args.length != 2) {
        System.out.println(
           "Application needs two arguments to run: " +
           "java com.hca.jdbc.UsingDriverManager <username> " +
           "<password>");
        System.exit(1);
     }

     // get the user name and password from the command line args
     String username = args[0];
     String password = args[1];

     // load the driver
     Class.forName("com.mysql.cj.jdbc.Driver");

     // Establish the variables with null outside the try scope
     Connection connection = null;
     PreparedStatement preparedStatement = null;
     ResultSet resultSet = null;

     try {
        // create the connection and prepared statement
        connection = DriverManager.getConnection(
           "jdbc:mysql://localhost:3306/sakila",
           username, password);

        preparedStatement = connection.prepareStatement(
           "SELECT first_name, last_name FROM customer " +
           "WHERE last_name LIKE ? ORDER BY first_name");

        // set the parameters for the prepared statement
        preparedStatement.setString(1, "Sa%");

        // execute the query
        resultSet = preparedStatement.executeQuery();
```

```java
            // loop thru the results
        while (resultSet.next()) {

             // process the data
            System.out.printf("first_name = %s, last_name = %s;\n",
                resultSet.getString(1), resultSet.getString(2));
        }
    }
    catch (SQLException e) {
      e.printStackTrace();
    }
    finally {
        // close the resources
        if (resultSet != null) resultSet.close();
        if (preparedStatement != null) preparedStatement.close();
        if (connection != null) connection.close();
    }
  }
}
```

- **The `close()` method can throw an exception**

  - This is one of the reasons we had to add throws to our main method signature

- **A better example would handle the exceptions on the `close()` calls**

Example: Handling Exceptions During Closing Resources

```java
package com.hca.jdbc;

import java.sql.*;

public class UsingDriverManager {

  public static void main(String[] args) throws
  ClassNotFoundException {

     if (args.length != 2) {
        System.out.println(
           "Application needs two arguments to run: " +
           "java com.hca.jdbc.UsingDriverManager <username> " +
           "<password>");
        System.exit(1);
     }

     // get the user name and password from the command line args
     String username = args[0];
     String password = args[1];

     // load the driver
     Class.forName("com.mysql.cj.jdbc.Driver");

     // Establish the variables with null outside the try scope
     Connection connection = null;
     PreparedStatement preparedStatement = null;
     ResultSet resultSet = null;

     try {
        // create the connection and prepared statement
        connection = DriverManager.getConnection(
           "jdbc:mysql://localhost:3306/sakila",
           username, password);

        preparedStatement = connection.prepareStatement(
           "SELECT first_name, last_name FROM customer " +
           "WHERE last_name LIKE ? ORDER BY first_name");

        // set the parameters for the prepared statement
        preparedStatement.setString(1, "Sa%");

        // execute the query
        resultSet = preparedStatement.executeQuery();
```

```java
            // loop thru the results
            while (resultSet.next()) {

                // process the data
                System.out.printf("first_name = %s, last_name = %s;\n",
                    resultSet.getString(1), resultSet.getString(2));
            }
        }
        catch (SQLException e) {
            e.printStackTrace();
        }
        finally {
            // close the resources
            if (resultSet != null) {
                try {
                    resultSet.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
            if (preparedStatement != null) {
                try {
                    preparedStatement.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
            if (connection != null) {
                try {
                    connection.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

Exercises

EXERCISE 3

Continue working with the NorthwindTraders project.

Update the `main` method to create a home screen for the user to choose what they want to do.

```
What do you want to do?
  1) Display all products
  2) Display all customers
  0) Exit
Select an option:
```

Now, add logic to query the Customers table in the Northwind database and display the customer…
  - `contact name`
  - `company name`
  - `city where they are located`
  - `country where they are located`
  - `phone number`

…of all customers. Order the results by country.

In this project, make sure to use a try/catch/finally around your code. You may NOT have a throws statement on your main function.

Commit and push your code!

`try-with-resources`

- **The `try-with-resources` statement is a `try` statement that declares one or more resources**

  - A resource is an object that must be released

- **The `try-with-resources` statement *automatically* ensures that each resource is closed at the end of the block**

  - You do not need a finally block to close the resource

- **The `try` statement is followed by parenthesis that allows you to declare one or more objects**

  - You don't include other executable statements in the parenthesis

- **Any object that implements `java.lang.AutoCloseable`, be used as a resource and included in the parethhesis**

Example

```
try (
   Connection connection = dataSource.getConnection();
   PreparedStatement preparedStatement =
        connection.prepareStatement("...");
) {
  //Place code here that use connection and prepared statement
  //Resources are automatically closed
}
catch (SQLException e) {
    e.printStackTrace();
}
```

Example: `try-with-resources`

```java
package com.hca.jdbc;

import java.sql.*;

public class UsingDriverManager {

  public static void main(String[] args) throws ClassNotFoundException {

    if (args.length != 2) {
       System.out.println(
          "Application needs two arguments to run: " +
          "java com.hca.jdbc.UsingDriverManager <username> " +
          "<password>");
       System.exit(1);
    }

    // get the user name and password from the command line args
    String username = args[0];
    String password = args[1];

    // load the driver
    Class.forName("com.mysql.cj.jdbc.Driver");

    // create the connection and prepared statement in a
    // try-with-resources block
    try (  Connection connection = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/sakila", username, password);

            PreparedStatement preparedStatement =
            connection.prepareStatement(
            "SELECT first_name, last_name FROM customer " +
            " WHERE last_name LIKE ? ORDER BY first_name");
    ) {

        // set the parameters for the prepared statement
        preparedStatement.setString(1, "Sa%");
```

```java
        // execute the query - also declare the ResultSet within
        // a try-with-resources block
        try (
            ResultSet resultSet = preparedStatement.executeQuery()
        ) {

          // loop thru the results
            while (resultSet.next()) {

             // process the data
                System.out.printf(
                 "first_name = %s, last_name = %s;\n",
                 resultSet.getString(1), resultSet.getString(2));
            }
        }
      }
      catch (SQLException e) {
          // This will catch all SQLExceptions occurring in the
          // block including those in nested try statements
          e.printStackTrace();
      }
    }
}
```

Exercises

EXERCISE 4

Continue working on the NorthwindTraders project.

Add a home screen menu option:

```
3) Display all categories
```

Your program should query the Categories table in the Northwind database and lists the…
- category id
- category name

…of all categories. Order the results by category id.

Then, prompt the user for a `categoryId` to display all products in that category.

Now query the `Products` table in the Northwind database and lists the…
- product id
- product name
- unit price
- units in stock

…of the products in the category the user selected.

In the code for this exercise, make sure to use a try-with-resources around your code. This means you won't have to explicitly close your resources in a finally block.

Commit and push your code!

See diagram next page for ideas…

```
try
    Class.forName

    try (
        Connectione connection = ...
    )
    try (
        PreparedStatement statement =
        ResultSet results = statement.execute()
    )

    looped thru results

    ask the user for an emplyoyee name

    try (
        PrepareStatement statement =
    }
    set parameter on statement
        try (ResultSet results = statement.executeQuery()
        )
        looped thru results
```

```
catch(ClassNotFoundException
catch(SQLException
catch(Exception
```

Section 2–4

CodeWars

CodeWars Kata

- **Insert Dashes**

  – Given a number - you must return a string with dashes added in the appropriate positions

- **Complete this Kata for additional Java practice**

  – https://www.codewars.com/kata/55960bbb182094bc4800007b/java

Module 3

JDBC DataSources

Section 3–1

DataSources

`java.sql.DataSource`

- **Using `Class.forName()` over time was proving to be too awkward**

  – Theoretically, you only have to call it once for the application, but finding a good place to put it was difficult in web applications

- **The current convention is to use a `java.sql.DataSource` object as a driver**

- **`DataSource` is an interface, so we don't necessarily *need to explicitly know what type of underlying database we are using***

  – In other words, we don't need to be concerned too deeply whether we use MySQL, SQLServer, Oracle, etc.

- **The `org.apache.commons` package has `BasicDataSource` object that can be used with various databases**

  – You must add a dependency to this package in the `pom.xml` file

  – After editing `pom.xml` you must reload/refresh maven

Example

```
<dependencies>
  <dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-dbcp2</artifactId>
  </dependency>
</dependencies>
```

- **The purpose of a DataSource is to manage a data connection for you**

    - Behind the scenes it uses the `DriverManager` object so that you don't have to

    - It also manages the opening and closing of connections

Defining a DataSource

- **Every driver has it's own way to instantiate a `DataSource`**

  - You will need to consult the documentation for your specific driver

- **In MySQL, we can instantiate and configure a `DataSource` as shown below**

Example

```
import org.apache.commons.dbcp2.BasicDataSource;
…

String username = args[0];
String password = args[1];

BasicDataSource dataSource = new BasicDataSource();

dataSource.setUrl("jdbc:mysql://localhost:3306/sakila");
dataSource.setUser(username);
dataSource.setPassword(password);
```

- **Every specific datasource implements the `java.sql.DataSource` interface, so we often use the interface type to hold the datasource object**

Example

```
DataSource dataSource = new BasicDataSource();
```

Creating a Connection using a `DataSource`

- **Once you have a the `DataSource` instance, you can call the `getConnection` method to obtain a connection**

  - Use the connection much in the same way as you did before

Example

```
Connection connection = dataSource.getConnection();
```

- **Once you create connection, create a prepared statement**

Example

```
try (Connection connection = dataSource.getConnection();
     PreparedStatement preparedStatement =
        connection.prepareStatement(
        "SELECT first_name, last_name FROM customer " +
        "WHERE last_name LIKE ? ORDER BY first_name");
) {

   preparedStatement.setString(1, "Sa%");

   try (ResultSet resultSet = preparedStatement.executeQuery()
   ) {
      while (resultSet.next()) {
         System.out.printf("first_name = %s, last_name = %s;\n",
            resultSet.getString(1), resultSet.getString(2));
      }
   }
}
catch (SQLException e) {
   e.printStackTrace();
}
```

Example: Using a `DataSource`

```java
package com.yearup.jdbc;

import org.apache.commons.dbcp2.BasicDataSource;
import javax.sql.DataSource;
import java.sql.*;

public class UsingDataSource {

  public static void main(String[] args) {

    if (args.length != 2) {
      System.out.println(
        "Application needs two arguments to run: " +
        "java com.hca.jdbc.UsingDriverManager <username> " +
        "<password>");
      System.exit(1);
    }

    // Get the username and password
    String username = args[0];
    String password = args[1];

    // Create the datasource
    BasicDataSource dataSource = new BasicDataSource ();

    // Configure the datasource
    dataSource.setUrl("jdbc:mysql://localhost:3306/sakila");
    dataSource.setUser(username);
    dataSource.setPassword(password);

    // Interact with the database
    doSimpleQuery(dataSource);
  }

  private static void doSimpleQuery (DataSource dataSource) {

    // Create the connection and prepared statement
    try (Connection connection = dataSource.getConnection();
         PreparedStatement preparedStatement =
           connection.prepareStatement(
           "SELECT first_name, last_name FROM customer " +
           "WHERE last_name LIKE ? ORDER BY first_name");
    ) {

       // Set any required parameters
       preparedStatement.setString(1, "Sa%");
```

```java
        // Execute the query
        try (ResultSet resultSet =
            preparedStatement.executeQuery()
        ) {
          // Process the results
          while (resultSet.next()) {
            System.out.printf(
                "first_name = %s, last_name = %s;\n",
                resultSet.getString(1), resultSet.getString(2));
          }
        }
    }
    catch (SQLException e) {
        // This will catch all SQLExceptions occurring
        // in the try block, including those in nested
        // try statements
        e.printStackTrace();
    }
  }
}
```

Querying a Database Using a Join

- **Join queries can be made just like any other queries**

- **In the example below we are using join to join the tables: customer, address, city and country**

  - We are querying all rows whose last names begin with "Ma" and display information from all the tables

Example

```
PreparedStatement preparedStatement =
   connection.prepareStatement(
      "SELECT customer.first_name, customer.last_name, " +
         "address.address, city.city, country.country " +
      "FROM customer " +
      "LEFT JOIN address " +
      "  ON (customer.address_id = address.address_id) " +
      "LEFT JOIN city " +
      "  ON (address.city_id = city.city_id) " +
      "LEFT JOIN country " +
      "  ON (city.country_id = country.country_id) " +
      "WHERE last_name LIKE 'Ma%' " +
      "ORDER BY customer.first_name;");
```

RESULTS

```
first_name = CALVIN, last_name = MARTEL, address = 138 Caracas
Boulevard, city = Maracabo, country = Venezuela
first_name = CLIFTON, last_name = MALCOLM, address = 1489
Kakamigahara Lane, city = Tanshui, country = Taiwan
first_name = DONALD, last_name = MAHON, address = 1774 Yaound
Place, city = Ezhou, country = China
first_name = ERICA, last_name = MATTHEWS, address = 1294 Firozabad
Drive, city = Pingxiang, country = China
first_name = GREGORY, last_name = MAULDIN, address = 507 Smolensk
Loop, city = Sousse, country = Tunisia
first_name = GWENDOLYN, last_name = MAY, address = 446 Kirovo-
Tepetsk Lane, city = Higashiosaka, country = Japan$
```

Example: JOIN

- **Notice in this example that we are passing in the `DataSource` object as a parameter**

Example

```
private static void doJoin(DataSource dataSource) {

  try (Connection connection = dataSource.getConnection();

      PreparedStatement preparedStatement =
          connection.prepareStatement(
          "SELECT customer.first_name, customer.last_name, " +
                  "address.address, city.city, country.country " +
          "FROM customer " +
          "LEFT JOIN address " +
          "  ON (customer.address_id = address.address_id) " +
          "LEFT JOIN city " +
          "  ON (address.city_id = city.city_id) " +
          "LEFT JOIN country " +
          "  ON (city.country_id = country.country_id) " +
          "WHERE last_name LIKE 'Ma%' " +
          "ORDER BY customer.first_name;");
  ) {

    try (ResultSet resultSet = preparedStatement.executeQuery()
    ) {

      while (resultSet.next()) {
        System.out.printf(
          "first_name = %s, last_name = %s, address = %s," +
          " city = %s, country = %s\n",
           resultSet.getString(1), resultSet.getString(2),
          resultSet.getString(3), resultSet.getString(4),
          resultSet.getString(5));
      }
    }
  }
  catch (SQLException e) {
    e.printStackTrace();
  }
}
```

Exercises

EXERCISE 1

Refactor the **NorthwindTraders** project to use the `DataSource` class instead of `DriverManager` to generate connections.

Remember to modify the `pom.xml` file to add the necessary dependency for the `BasicDataSource` class.

```
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-dbcp2</artifactId>
</dependency>
```

Commit and push your code!

EXERCISE 2

Create a new Maven Project named **SakilaMovies** in the `workbook-8` folder.

Your program should use the `DataSource` approach to obtaining a connection.

Begin by asking the user for a last name of an actor they like. Use that to display a list of all actors with that last name.

Now ask the user to enter a first name and a last name of an actor they want to see the movies of. Use that to display a list of the films that actor has been in.

Use a try-with-resources around your code. This means you won't have to explicitly close your resources.

One challenge to this project is handling the situation where a query returned no records. For example, if the user typed in a last name of Wyatt there might be no actors with that name.

One way to do that is the following:

```
// use the first call to next() to see if there are records
if (results.next()) {
   System.out.println("Your matches are: \n");

   // if there are, you are already sitting on the first one so
   // switch your loop to using a do/while
   do {
      // process results
   } while (results.next());
}
else {
   System.out.println("No matches!");
}
```

Commit and push your code!

Section 3-2

CodeWars

CodeWars Kata

- **Perpendicular Lines**

  - You are given a number of lines, you need to determine the maximum number of 90 degree intersections you can make with that number of lines.

- **Complete this Kata for additional Java practice**

  - https://www.codewars.com/kata/6391fe3f322221003db3bad6/java

Section 3–2

Creating a DataManager

DataManager - A Better Strategy

- **Rather than encode all of your database interactions, you should encapsulate it into a "data manager" type class**

  – In other words, you should have a single class that has the responsibility of communicating with the database

- **This means that you will need to use model classes to carry database data**

Example

```java
public class NorthwindDataManager {

  public List<Supplier> getAllSuppliers() {
      // run query
      // loop thru results
      //    - create supplier object
      //    - add to arraylist
      // return arraylist
  }

  public List<Product> getAllProducts() {
      // run query
      // loop thru results
      //    - create product object
      //    - add to arraylist
      // return arraylist
  }

  public List<Product> getProductsInPriceRange(float min,
                                               float max) {
      // run query
      // loop thru results
      //    - create product object
      //    - add to arraylist
      // return arraylist
  }
}
```

Responsibilities

- **Your application code should not communicate directly with the database**

  – Instead we introduce an intermediary object that handles all database communication - a middle-man if you will

- **Your Application**

  – Interacts with the user

  – Displays Product, Category, Supplier and Customer information

  – Gets input from the user

- **The DataManager class**

  – Is the middle-man… a translator that converts

    * Java objects into SQL statements (pushes data TO the database)

    * SQL Rows into Java objects (pulls data FROM the database)

- **The Database**

  – Stores data in tables

  – Handles SQL queries and CRUD operations

Example: Using a `DataManager`

```java
public class NorthwindDataManager {

  private DataSource dataSource;

  public NorthwindDataManager(DataSource dataSource){
    this.dataSource = dataSource;
  }

  public List<Product> getAllProducts() {
    List<Product> products = new ArrayList<>();

    String sql = "SELECT ProductID " +
                 " , ProductName " +
                 " , UnitPrice " +
                 "FROM products;";

    Connection conn = dataSource.getConnection();

    PreparedStatement statement = conn.prepareStatement(sql);

    ResultSet results = statement.executeQuery(sql);

    while (results.next())
    {
      int id = results.getInt("ProductID");
      String name = results.getString("ProductName");
      double price = results.getDouble("UnitPrice");

      Product product = new Product(id, name, price);
      products.add(product);
    }

    return products;
  }

  public List<Product> getProductsInPriceRange(float min,
                                               float max) {
    // find products in price range
  }

}
```

NorthwindApplication.java

```java
public class NorthwindApplication {

  public static void main(String[] args) {

      // Get the username and password
      String username = args[0];
      String password = args[1];

      // Create the datasource
      BasicDataSource dataSource = new BasicDataSource();

      // Configure the datasource
      dataSource.setUrl("jdbc:mysql://localhost:3306/Northwind");
      mysqlDataSource.setUser(username);
      mysqlDataSource.setPassword(password);

      // Create the NorthwindDataManager
      NorthwindDataManager dataManager =
                  new NorthwindDataManager(dataSource);

      // Interact with the database
      List<Product> products = dataManager.getAllProducts();

      // display products
      products.forEach(System.out::println);
  }
}
```

Exercises

EXERCISE 3

Continue working with the SakilaMovies project Refactor the data access logic in your application and move it to a DataManager class.

You will need to create two Model classes. You do not need all of the database fields, but you should included the following fields.

```
Actor
-------
actorId
firstName
lastName

Film
---------
filmId
title
description
releaseYear
length
```

Your DataManager class should include a method to search for actors by name and return a list or Actors. You should also add a method to return a list of Films by actor id.

Prompt the user to search for actor by name and display the results of the search.

Add another prompt to ask the user to enter an actor id, then search for and display a list of movies that the actor has appeared in.

Commit and push your code!

Adding Mutiple DataManagers

- **Databases have many tables**

- **A single DataManager class can get very large where methods have low cohesion**

- **One way to handle this is to create one DataManager class per table**

  – These classes are often referred to as a Data Access Object (DAO)

  – Any logic related to accessing, adding or changing data in a table should be added to the appropriate DAO

Example: DAOs

ProductDao.java

```java
public class ProductDao {
  private DataSource dataSource;

  publid ProductDao (DataSource dataSource) {
    this.dataSource = dataSource;
  }

  public List<Product> getAllProducts(){...}

  public Product getProduct(int productid){...}

  public Product add(Product product){...}

  public void update(Product product){...}

  public void delete(int productid){...}

}
```

CategoryDao.java

```java
public class CategoryDao {
  private DataSource dataSource;

  publid CategoryDao (DataSource dataSource) {
    this.dataSource = dataSource;
  }

  public List<Category> getAllCategories(){...}

  public Category get Category(int categoryId){...}

  public Category add(Category category){...}

  public void update(Category category){...}

  public void delete(int categoryId){...}

}
```

Section 3–3

CodeWars

CodeWars Kata

- **Countries Capitals for Trivia Night**

  – Write a query to answer the trivia question

- **Complete this Kata for additional Java practice**

  – https://www.codewars.com/kata/5e5f09dc0a17be0023920f6f/sql

Module 4

CRUD Operations

Section 4–1

CRUD Operations

Transitioning from Querying

- **JDBC does more than just querying**

- **In this section, we will discuss:**

  - Inserting

  - Inserting with Keys

  - Updating

  - Deleting

- **Also, although not here, you can do other things like execute stored procedures and transactions**

Inserting into the Database

- **Inserting into the database requires a connection and prepared statement just like querying**

- **We call `execute()` or `executeUpdate()` when we want to perform the insert**

  - `executeUpdate()` will return the number of rows that were affected by the insert

Example

```java
public static void insertIntoDirect(DataSource dataSource) {

  // Create the connection and prepared statement
  try (Connection connection = dataSource.getConnection();
       PreparedStatement preparedStatement =
           connection.prepareStatement(
         "insert into country (country) values (?);");
  ) {
    // set the parameter
     preparedStatement.setString(1, "Eritrea");

    // execute the query
     int rows = preparedStatement.executeUpdate();

    // confirm the update
     System.out.printf("Rows updated %d\n", rows);
  }
  catch (SQLException e) {
     e.printStackTrace();
  }
}
```

Inserting into Table and Getting the Auto-Generated Primary Key

---

- **If you want to retrieve the automatic primary keys that were generated, you must pass an additional parameter to `prepareStatement`**

  - `Statement.RETURN_GENERATED_KEYS`

Example

```
PreparedStatement preparedStatement =
    connection.prepareStatement("insert-statement-here",
                            Statement.RETURN_GENERATED_KEYS);
```

- **The keys can then be retrieved from the prepared statement using the method `getGeneratedKeys()`**

  - It returns a `ResultSet` with the keys

- **You can iterate through the `ResultSet` using `next()`**

  - Fetch the keys using `getXXX()` methods

Example: Getting Primary Keys after Insert

Example

```java
public static void insertIntoWithGeneratedKeys(
DataSource dataSource) {

  // Create the connection and prepared statement
  try (Connection connection = dataSource.getConnection();
       PreparedStatement preparedStatement =
           connection.prepareStatement(
           "insert into country (country) values (?);",
             Statement.RETURN_GENERATED_KEYS);

  ) {
     // Set parameters in the preparedStatement
     preparedStatement.setString(1, "Mongolia");

     // Execute the preparedStatement
     int rows = preparedStatement.executeUpdate();

     // Display the number of rows that were updated
     System.out.printf("Rows updated %d\n", rows);

     // Get the result containing primary key(s)
     try (ResultSet keys = preparedStatement.getGeneratedKeys()
     ) {

         // Iterate through the primary keys that were generated
         while (keys.next()) {
             System.out.printf("%d key was added\n",
                 keys.getLong(1));
         }
     }
  }
  catch (SQLException e) {
     e.printStackTrace();
  }
}
```

Updating a Record

- **Update follows the same process:**

  - Create a `DataSource`

  - Create a `PreparedStatement`

  - Set the query parameters

  - Set values for what is required to update

Example

```java
public static void updateRecord(DataSource dataSource) {

  // Create the connection and prepared statement
  try (Connection connection = dataSource.getConnection();
       PreparedStatement preparedStatement =
          connection.prepareStatement(
           "UPDATE film_text SET description = ? " +
          "WHERE film_id = ?");
  ) {

    // Set two parameters in the preparedStatement
    preparedStatement.setString(1, "Apache Devine is an Apache " +
       "Project that delivers messages to different brokers " +
       "without care as to what technology it is");
    preparedStatement.setLong(2, 31);

    // Execute the preparedStatement
    int rows = preparedStatement.executeUpdate();

    // Display the number of rows that were updated
    System.out.printf("Rows updated %d\n", rows);
  }
  catch (SQLException e) {
    e.printStackTrace();
  }
}
```

Delete a Record

- **Delete uses the same process we've just seen**

  - The only difference is you use a DELETE SQL statement

Example

```java
public static void deleteRecord(DataSource dataSource) {

  // Create the connection and prepared statement
  try (Connection connection = dataSource.getConnection();
       PreparedStatement preparedStatement =
         connection.prepareStatement(
         "DELETE FROM country WHERE country = ?");
  ) {
     // Set parameters in the preparedStatement
    preparedStatement.setString(1, "Mongolia");

     // Execute the preparedStatement
    int rows = preparedStatement.executeUpdate();

     // Display the number of rows that were updated
    System.out.printf("Rows deleted %d\n", rows);
  }
  catch (SQLException e) {
    e.printStackTrace();
  }
}
```

How We Ran the Examples

- **The examples you just saw, were all executed from the following `main()` method**

  - We simply uncommented the one method we wanted to test

Example

```java
package com.yearup.jdbc;

import com.mysql.cj.jdbc.MysqlDataSource;

import javax.sql.DataSource;
import java.sql.*;

public class UsingDataSource {

  // NOTE:  The method uses a throws statement to acknowledge
  // we did not add try/catch logic to handle exceptions
  // thrown by some of the methods

  public static void main(String[] args) throws SQLException,
  ClassNotFoundException {

    // we passed the DB user name and password in to the
    // method using command line args
    if (args.length != 2) {
      System.out.println(
          "Application needs two arguments to run: " +
          "java com.hca.jdbc.UsingDriverManager <username> <password>");

      System.exit(1);
    }

    // Get the user name and password
    String username = args[0];
    String password = args[1];

    // Create the data source
    BasicDataSource dataSource = new BasicDataSource();
    dataSource.setUrl("jdbc:mysql://localhost:3306/sakila");

    dataSource.setUser(username);
    dataSource.setPassword(password);
```

```
      // UNCOMMENT ANY OF THE FOLLOWING LINES TO RUN THE TEST
//    doSimpleQuery(dataSource);
//    doJoin(dataSource);
      insertIntoDirect(dataSource);
//    insertIntoWithGeneratedKeys(dataSource);
//    updateRecord(dataSource);
//    deleteRecord(dataSource);


   }
}
```

Exercises

In this exercise you will create an application to allow a user to perform CRUD operations with the Northwind database.

EXERCISE 1

Create a new Java Project named **NorthwindShippers**. Add the appropriate dependencies in the `pom.xml` file.

Use a DataManager (or DAO file for the Shippers table) for all data access. Create the necessary model class to perform the following steps

You will create a program that leads the user on the following journey:

1. Prompt the user for new shipper data (name and phone) and then insert it into the shippers table. Display the new shipper id when the insert is complete.

2. Run a query and display all of the shippers

3. Prompt the user to change the phone number of a shipper. They should enter the id and the phone number.

4. Run a query and display all of the shippers

5. Prompt the user to delete a shipper.   DO NOT ENTER SHIPPERS 1-3. They have related data in other tables. Delete your new shipper.

6. Run a query and display all of the shippers

Commit and push your code!

Section 4–2

CodeWars

CodeWars Kata

- **Subqueries Master**

  - Write a query with subqueries to perform string manipulation of a name.

- **Complete this Kata for additional Java practice**

  - https://www.codewars.com/kata/594323fde53209e94700012a/sql

Module 5

Miscellaneous Topics

Section 5–1

Miscellaneous Topics

Creating a Table

- **You may be sensing a pattern with the queries we've just seen**

  - Create a `DataSource`

  - Create a `PreparedStatement`

  - Perhaps obtaining a `ResultSet`

- **Creating a table in a database is no different than any other SQL command**

Example

```java
public static void createTable(DataSource dataSource) {

  // Create the connection and prepared statement
  try (Connection connection = dataSource.getConnection();
       PreparedStatement preparedStatement =
          connection.prepareStatement(
          "CREATE TABLE film_reviews (" +
            "movie_review_id bigint primary key, " +
            "news_source varchar(500) not null, " +
            "stars int not null, " +
            "reviewer varchar(200), " +
            "film_id smallint unsigned not null, " +
            "FOREIGN KEY (film_id) REFERENCES " +
            "film(film_id)) ENGINE=INNODB;");
  ) {
    // execute the prepared statement
    preparedStatement.execute();
  }
  catch (SQLException e) {
    e.printStackTrace();
  }
}
```

Calling a Stored Procedure

- **Stored procedures are like functions that are written in SQL**

  – They have a name

  – They can accept parameters

  – They can return an output parameter and a result set

    ∗ Or nothing at all

- **However they are written in a language specific to the DBMS system that you are using**

  – MySQL uses the MySQL Stored Program Language

  – Microsoft SQL Server uses Transact-SQL (or T-SQL), although you integrate C# into it

  – Oracle uses PL/SQL

  – IBM DB2 uses the IBM DB2 SQL

Sample Stored Procedure in MySQL

- **The following is a sample stored procedure in the Northwind database**

Example

```
CREATE DEFINER=`root`@`localhost`
       PROCEDURE `CustOrderHist`(in AtCustomerID varchar(5))
BEGIN

SELECT ProductName,
    SUM(Quantity) as TOTAL
FROM Products P,
     `Order Details` OD,
     Orders O,
     Customers C
WHERE C.CustomerID = AtCustomerID
  AND C.CustomerID = O.CustomerID
  AND O.OrderID = OD.OrderID
  AND OD.ProductID = P.ProductID
GROUP BY ProductName;

END
```

Calling a Stored Procedure in MySQL Workbench

- **To call the stored procedure from MySQL Workbench, use the CALL statement**

  - Include any parameters in parenthesis

Example

```
CALL CustOrderHist('EASTC');
```

- **It returns the rows selected by the stored procedure**

ProductName	TOTAL
Gudbrandsdalsost	30
Flotemysost	55
Thringer Rostbratwurst	21
Steeleye Stout	35
Maxilaku	30
Nord-Ost Matjeshering	15
Louisiana Hot Spiced Okra	24
Chef Anton's Cajun Seasoning	25
Queso Cabrales	5
Uncle Bob's Organic Dried Pears	100
Ipoh Coffee	6
Mozzarella di Giovanni	20
Gumbr Gummibrchen	12
Geitost	30
Louisiana Fiery Hot Pepper Sauce	21
Chai	25
Chef Anton's Gumbo Mix	30
Pt chinois	35
Camembert Pierrot	50

Calling a Stored Procedure in Java

- **To call the stored procedure from Java, use the
  `CallableStatement` rather than `PreparedStatement`**

  – Much of the rest of the process is similar

  – NOTE: We are not showing a complete program, nor are we
    cleaning up the resources

Example

```java
// Create a connection
Connection conn = dataSource.getConnection();

// create the query, but surround it with {  } and
// use a ? for parameters
String query = "{CALL CustOrderHist(?)}";

// Create the CallableStatement
CallableStatement stmt = conn.prepareCall(query);

// Set any parameters
stmt.setString(1, "EASTC");

// Execute the query
ResultSet resultSet = stmt.executeQuery();

// Process the returned values
while (resultSet.next()) {

    System.out.printf("%s - %d\n",
        resultSet.getString("ProductName"),
        resultSet.getInt("TOTAL"));
}
```

- **For more information, see:**
  https://docs.oracle.com/cd/E17952_01/connector-j-8.0-
  en/connector-j-usagenotes-statements-callable.html

Exercises

In this exercise you will work with stored procedures rather than writing queries directly in your java DAO classes.

EXERCISE 1

Create a new Maven project named **NorthwindProcedures**. Add all of the appropriate dependencies in the `pom.xml` file.

Use a DataManager or DAO file for all data access. Create the necessary model class to perform these steps

Using MySQL Workbench explore the available stored procedures. Test the stored procedures in Workbench and determine what input each procedure needs, and also what data each procedure returns.

You will create a program that allows the user to perform the following actions:

1. Search for customer order history - use the `CustOrderHistory` stored procedure

2. Search for sales by year - use the `'Sales by Year'` stored procedure

3. Search for sales by category - use the `'SalesByCategory'` stored procedure

Create a basic user interface and prompt the user for the appropriate information.

Don't forget to to commit and push your code frequently!

Section 5–2

CodeWars

CodeWars Kata

- **Divisible Ints**

  – Given an integer, it is your responsibilitydetermine if the number is evenly divisible by each number in the substring of the number

- **Complete this Kata for additional Java practice**

  – https://www.codewars.com/kata/566859a83557837d9700001a/java