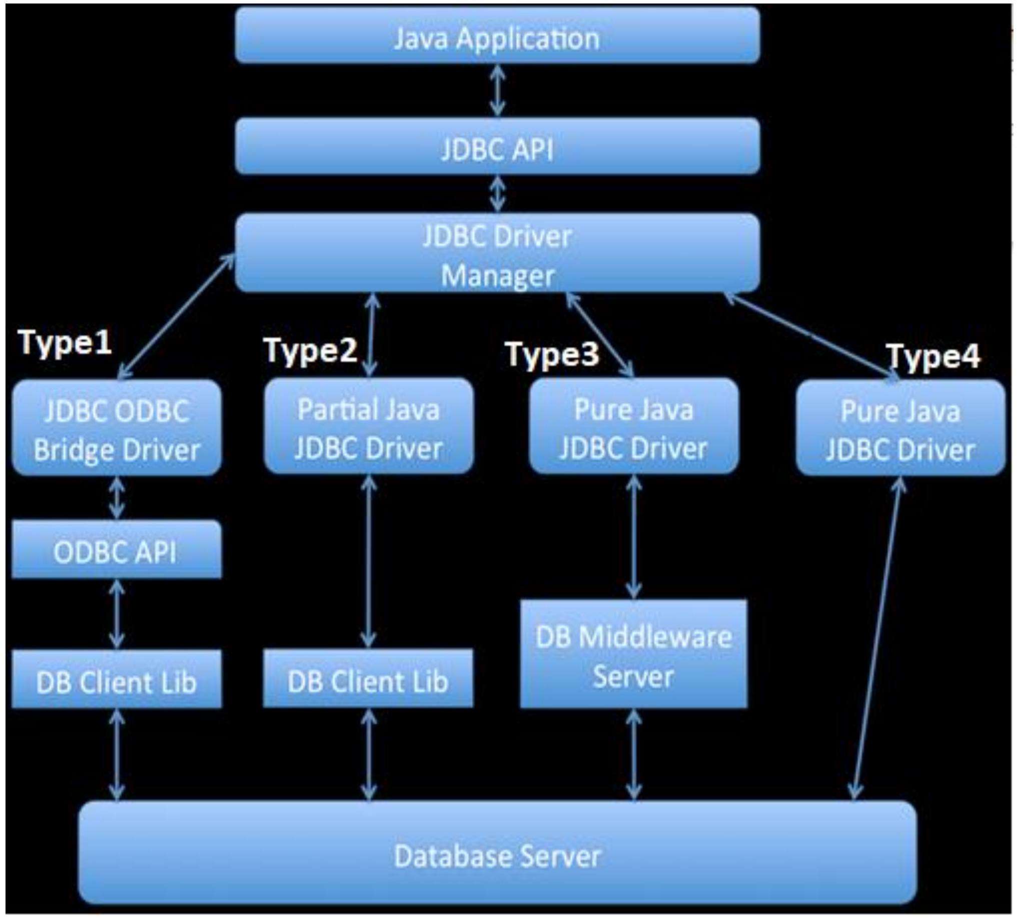
Unit 2
JDBC

LUCÍA SAN MIGUEL LÓPEZ

ODBC & JDBC

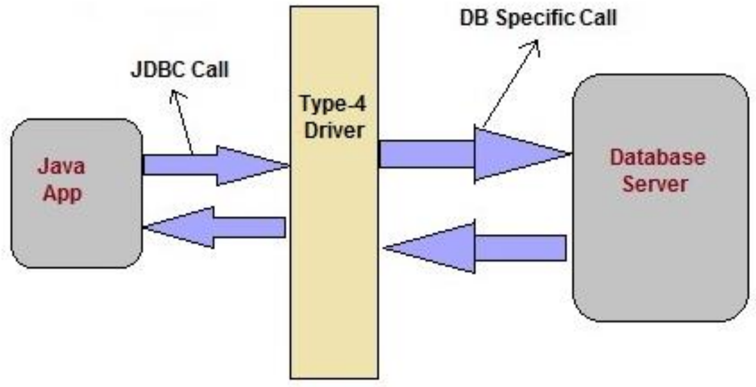
- **ODBC** (Open Database Connectivity) is probably the most widely used API for accessing relational databases.
  - It offers the possibility to connect to a majority of databases on almost every platform.
- **JDBC** (Java Database Connectivity) is a specific Java API that allows us to access relational databases.
- JDBC offers 4 different types of drivers



http://www.studytonight.com/java/introduction-to-jdbc.php

JDBC components. Types of Drivers

- A JDBC driver must be installed for the database management system that will be used. The driver is provided by the seller.
- MySQL provides standards-based drivers for JDBC, ODBC, and .Net enabling developers to build database applications in their language of choice.
- JDBC Driver for MySQL (Connector/J) is a type-4 Database-Protocol driver (Pure Java driver), using a pure Java client to communicate directly with the data origin



Installing JDBC driver

- Installing a JDBC driver consists of copying the driver to the computer and adding its location to the class path. The majority of drivers that are not Type 4 also require installing a client-side API.
  - Download ConnectorJ .jar and add it to the IntelliJ project:
    - Download zip from https://dev.mysql.com/downloads/connector/j/ and extract .jar file
    - Add .jar to File -> Project structure -> Libraries

  - Using Maven:
    - https://mvnrepository.com/artifact/mysql/mysql-connector-java/8.0.26

See JDBCConnectionExample

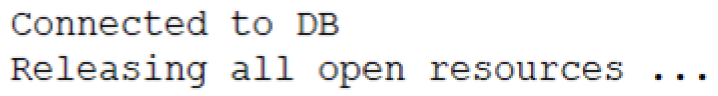
JDBC - Establishing connections

To establish a connection, one of the following classes must be used:

- **DriverManager**: A class that connects an application with a data source specified by a URL. It automatically loads JDBC drivers found in the class path.
- **DataSource**: An interface that hides the low-level data source details of the application. It provides advanced features like connection pools and distributed transactions.
  - Used in Java EE applications

Activity 2_1

- Read the code for the **JDBCConnectionExample** classes. Make sure you understand it.
- Change the values of mysql-properties.xml to the correct values needed for your connection.
- Run **JDBCConnectionExample App.java** and verify if the connection to the database was made properly.



```
Connected to DB
Releasing all open resources ...
```

See JDBCCoffeeExample (CoffeeDAO.java listTable method)

SQL queries

To process any SQL statement with JDBC, the following steps must be followed:

1. Establish a connection (Connection class)
2. Create a statement (Statement class)
3. Execute the query:

   - **executeQuery**: Returns a ResultSet object

   - **executeUpdate**: Returns an integer that represents the number of rows affected by the statement. Use with INSERT, DELETE and UPDATE statements

   - **execute**: Returns *true* if the first object returned by the query is a ResultSet. Use when a query can return more than one ResultSet. By calling Statement.getResultSet repeatedly, we can recover all ResultSet objects.

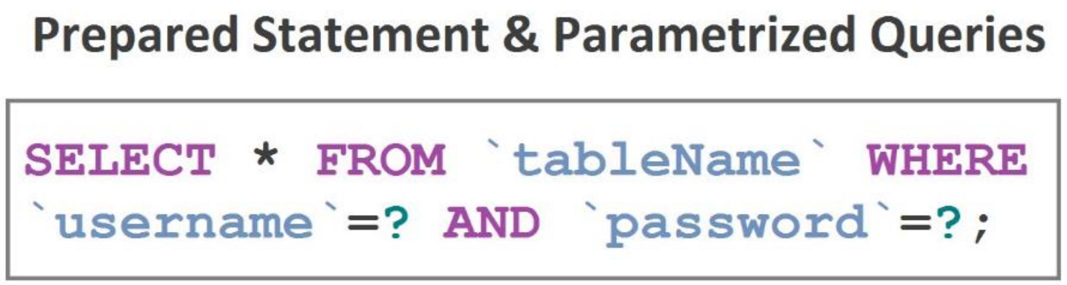
4. Process the result (ResultSet class)
5. Close the connection (Connection class)

See JDBCCoffeeExample (CoffeeDAO.java updateCoffeeSales method)

JDBC - Prepared Statements

A PreparedStatement is a statement specialization with two fundamental advantages:

- ❖ The query is sent precompiled to the database manager ready to be executed. Therefore, it is more efficient and secure (code injection attacks are avoided).
- ❖ Allows for parameterized queries:
  - The construction of queries with parameters is simplified.
  - The same query can be used with different values.



Prepared Statement & Parametrized Queries

```
SELECT * FROM `tableName` WHERE
`username`=? AND `password`=?;
```

JDBC - Working with ResultSet: Recover and modify values

- A *ResultSet* is an object that stores data generated from executing a query.
- To access the data, a cursor is used, with its initial position before the first row.
- With the method *next* () the cursor is moved to the next row.
- The *ResultSet* interface declares *getter* methods to recover values from columns on the current row according to the type of data (getLong, getBoolean…)
  https://docs.oracle.com/javase/7/docs/api/java/sql/ResultSet.html
- Values can be recovered using a numeric index or the alias (name).

```
String coffeeName = rs.getString(1);
String coffeeName = rs.getString("COF_NAME");
```

- Recommendations:
  - Use numeric index.
  - Read columns from left to right.
  - Read each column only once.

JDBC - Working with ResultSet: Recover and modify values

- The ResultSet interface provides methods for retrieving and manipulating the results of executed queries, and ResultSet objects can have different functionality and characteristics.
- These characteristics are:

  - Concurrency
  - Type
  - Cursor *holdability*

JDBC - Working with ResultSet: Recover and modify values

ResultSet concurrency:

The concurrency of a ResultSet object determines what level of update functionality is supported.

There are two concurrency levels:

- CONCUR_READ_ONLY: The ResultSet object cannot be updated using the ResultSet interface.
- CONCUR_UPDATABLE: The ResultSet object can be updated using the ResultSet interface.

The default ResultSet concurrency is CONCUR_READ_ONLY.

JDBC - Working with ResultSet: Recover and modify values

ResultSet type:

The sensitivity of a ResultSet object is determined by one of these ResultSet types:

- TYPE_FORWARD_ONLY: The result set cannot be scrolled; its cursor moves forward only, from before the first row to after the last row.
- TYPE_SCROLL_INSENSITIVE: The result can be scrolled; its cursor can move both forward and backward relative to the current position, and it can move to an absolute position. The result set is insensitive to changes made to the underlying data source while it is open.
- TYPE_SCROLL_SENSITIVE: The result can be scrolled; its cursor can move both forward and backward relative to the current position, and it can move to an absolute position. The result set reflects changes made to the underlying data source while the result set remains open.

The default ResultSet type is TYPE_FORWARD_ONLY.

See JDBCCoffeeExample (CoffeeDAO.java updateCoffeePrices & insertNewRow methods)

JDBC - Working with ResultSet: Recover and modify values

Methods available to move the cursor:

•**next**: Moves the cursor forward one row. Returns true if the cursor is now positioned on a row and false if the cursor is positioned after the last row.
•**previous**: Moves the cursor backward one row. Returns true if the cursor is now positioned on a row and false if the cursor is positioned before the first row.
•**first**: Moves the cursor to the first row in the ResultSet object. Returns true if the cursor is now positioned on the first row and false if the ResultSet object does not contain any rows.
•**last**: Moves the cursor to the last row in the ResultSet object. Returns true if the cursor is now positioned on the last row and false if the ResultSet object does not contain any rows.
•**beforeFirst**: Positions the cursor at the start of the ResultSet object, before the first row. If the ResultSet object does not contain any rows, this method has no effect.
•**afterLast**: Positions the cursor at the end of the ResultSet object, after the last row. If the ResultSet object does not contain any rows, this method has no effect.
•**relative(int rows)**: Moves the cursor relative to its current position.
•**absolute(int row**): Positions the cursor on the row specified by the parameter row.

See JDBCCoffeeExample (CoffeeDAO.java updateSales method)

JDBC - Transactions

A transaction is a set of one or more statements that is executed as a unit, so either all of the statements are executed, or none of the statements are executed.

For example, when the proprietor of The Coffee DB updates the amount of coffee sold each week, the proprietor will also want to update the total amount sold to date. However, the amount sold per week and the total amount sold should be updated at the same time; otherwise, the data will be inconsistent.
The way to be sure that either both actions occur or neither action occurs is to use a transaction.

When a connection is created, it is in auto-commit mode. This means that each individual SQL statement is treated as a transaction and is automatically committed right after it is executed.

The way to allow two or more statements to be grouped into a transaction is:

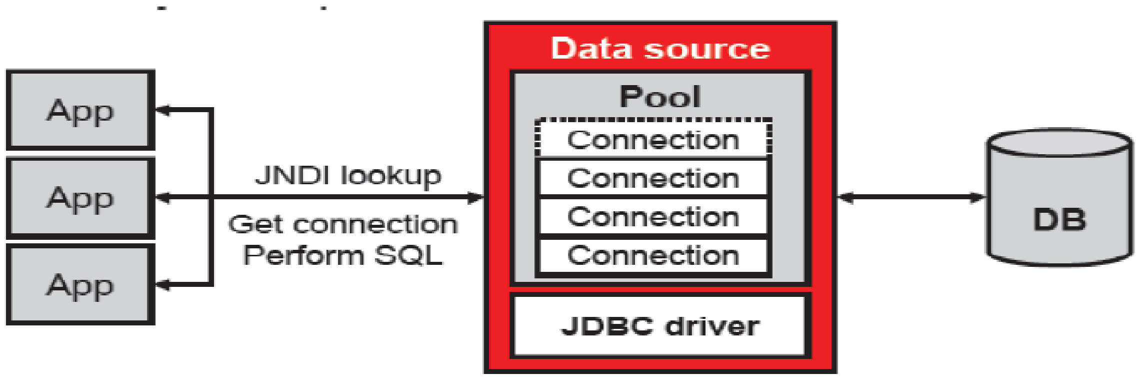
- Disable Auto-Commit Mode
- Commit Transactions: no SQL statements are committed until you call the method commit explicitly
- Set Roll Back calls: undo partial operations if an error occurs in the database
- Enable Auto-Commit Mode

Recovering autoincremental value

```
String sql = "INSERT INTO table (column1, column2) values(?, ?)";
stmt = conn.prepareStatement(sql, Statement.RETURN_GENERATED_KEYS);
stmt.executeUpdate();
ResultSet rs = stmt.getGeneratedKeys();
if(rs.next()) {
		auto_id = rs.getInt(1);
}
```

Connection pooling

- Most of the processes only need to have access to a JDBC connection when they are actively processing a transaction, which usually only takes a few milliseconds.
- When a transaction is not being processed, the connection is inactive. *A connection pool allows for an inactive connection to be used by another thread to complete a task.*



Connection pooling

- When a thread needs to access a DBMS through the JDBC, it requests a connection to the pool.
- A connection is returned from the pool of available connection objects. If there is no available connection, the lookup creates a new connection.
- When finished, the connection is returned to the pool so other threads can use it.
- When a connection is allowed through a pool, it is used exclusively by the thread that requested it, the same as if using DriverManager.getConnection

See JDBCCoffeeExampleWith Pool

Connection pooling

Advantages: Improvement of performance and scalability

- Reduces the number of times new connection objects are created.
- Promotes connection object reuse.
- Quickens the process of getting a connection.
- Reduces the amount of effort required to manually manage connection objects.
- Minimizes the number of stale connections.
- Controls the amount of resources spent on maintaining connections.

Scope:

- Specific for an application.
- Global to every application, reproducing the pool for each application.
- Global to every application, sharing the pool between all the applications.

This two libraries are required for working with Connection pools in Java: commons-dbcp2 y commons-pool

Activity 2_2

Modify the webstore application for using a Connection pool:

1. Create a new class DBConnPool
2. Add the new methods:
    - getPool()
    - closePool()
3. Use the singleton pattern for creating the pool only once
4. Modify the methods of DBConnection:
    - getConnection(): Get a connection from the pool
    - closeConnection(): Release the connection
5. Test the app: No more code needs to be modified