Welcome to the JDBC and Hibernate module!

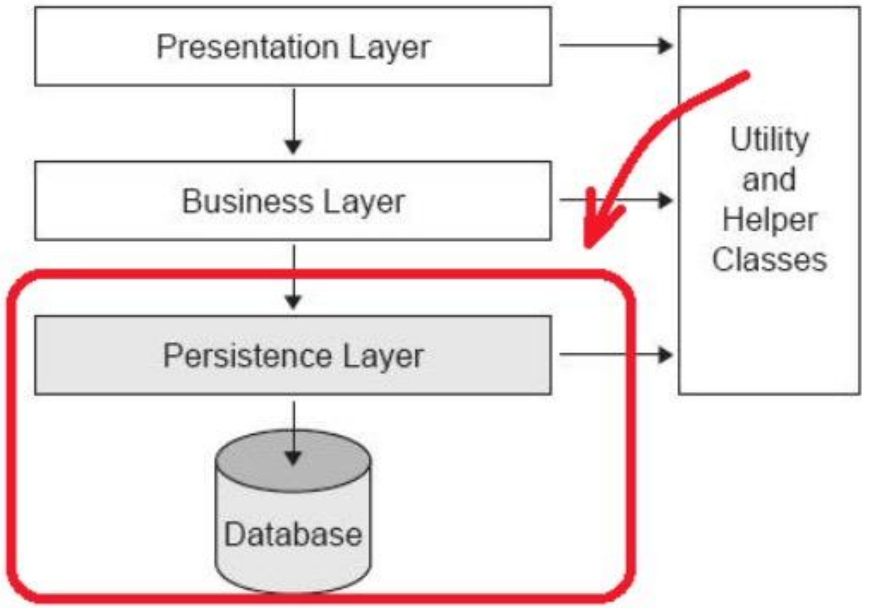
Trainer: Diana Cavalcanti

Setup

- WorkBench
    - Java_Specification_-_Linux_-_en.pdf
    - Java_Specification_-_MacOS_-_en.pdf
    - Java_Specification_-_Windows_-_en.pdf

Persistence layer

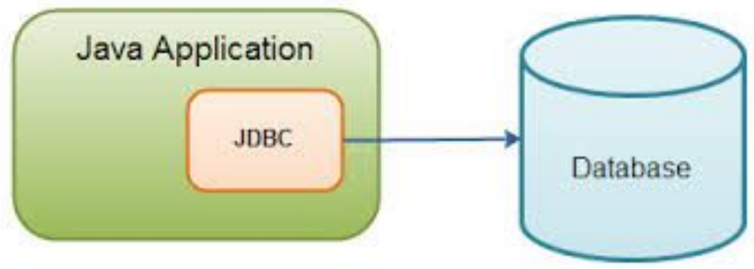
- It is the layer that deal with the database
- It consist of a set of classes that maps the database and all operation on that table
- DAO vs Repository
  - https://www.baeldung.com/java-dao-vs-repository
  - https://www.baeldung.com/java-dao-pattern



JDBC

JDBC

- **Java Database Connectivity (JDBC)**
  - An application programming interface (API) for the programming language Java, which defines how a client may access a database.
  - Basically, the JDBC library will help you interact with a database from your Java applications.



Read more: https://www.tutorialspoint.com/jdbc/jdbc-introduction.htm

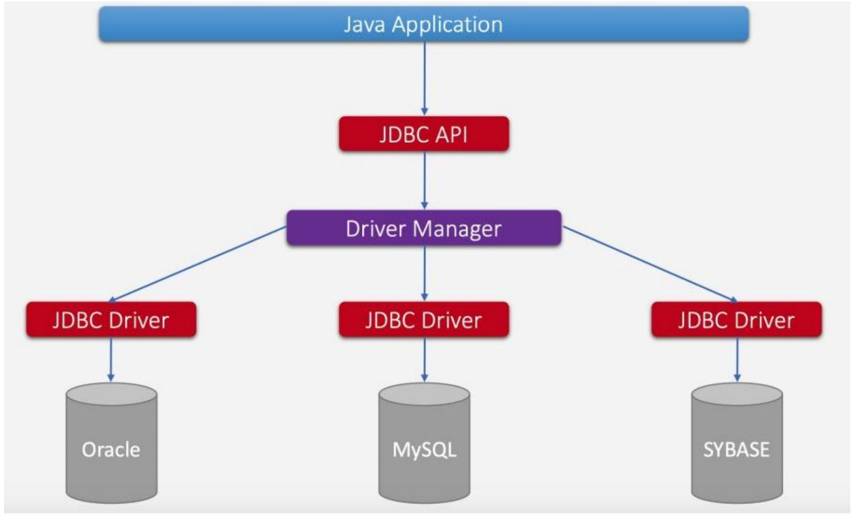
JDBC

- The JDBC library includes APIs for each of the tasks mentioned below that are commonly associated with database usage:
  - making a connection to a database
  - creating queries and update statements
  - executing queries and update statements on the database
  - viewing and modifying the results received from the database

- The Java JDBC API standardizes how to connect to a database, how to execute queries against it, how to navigate the result of such a query, how to execute updates on the database and how to call stored procedures.
- This standardization means that the code looks the same across different database products. Thus, changing to another database will be a lot easier, if your project needs that in the future.

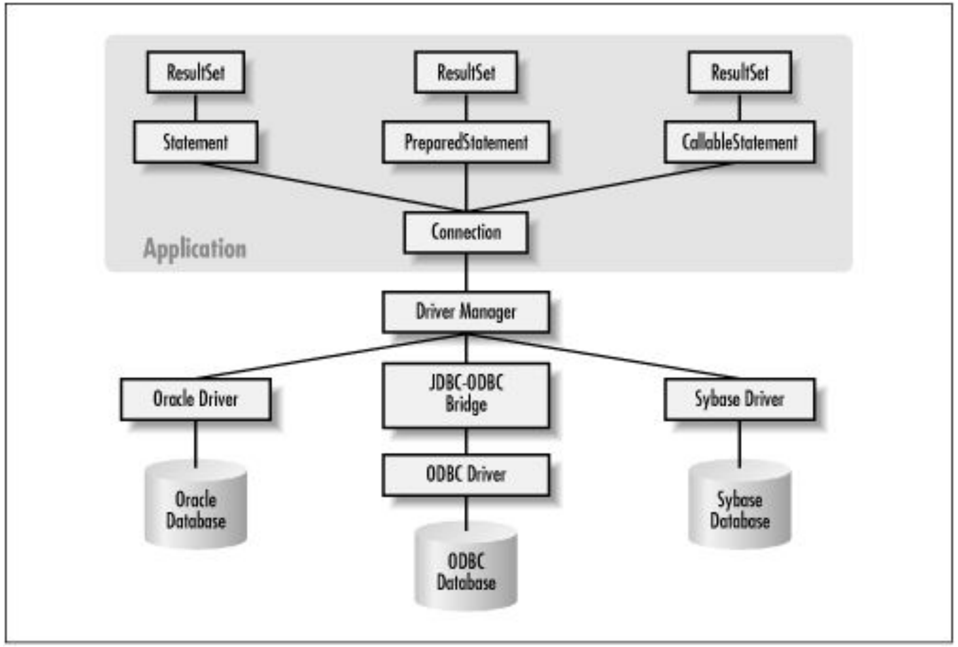
JDBC Architecture

- In order to support this level of standardization across multiple database types, the JDBC architecture consists of two layers:
  - **JDBC API** - provides the application-to-JDBC Manager Connection - the part of the code your application will interact with
  - **JDBC Driver API -** supports the JDBC Manager-to-Driver Connection - what JDBC uses behind the scenes in order to interact with different databases



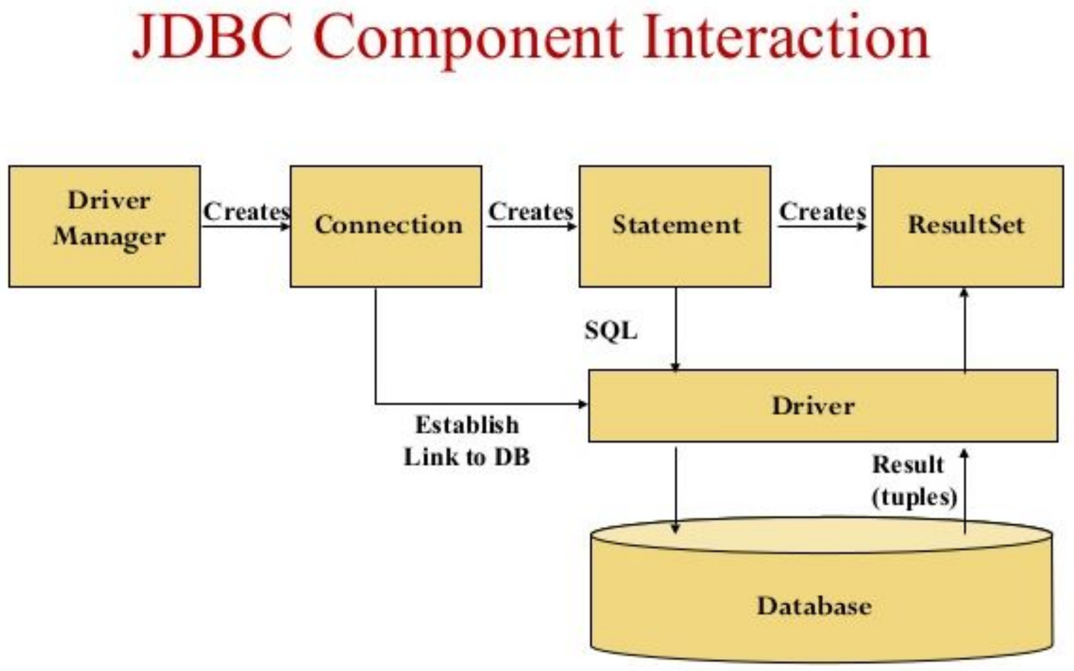
JDBC Architecture

- **The JDBC API** uses a driver manager and database specific drivers to provide transparent connectivity to heterogeneous databases.
- **The JDBC Driver Manager** ensures that the correct driver is used to access each data source. The driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases.



JDBC - Components

- DriverManager
- Driver
- Connection
- Statement
- ResultSet
- *SQLException



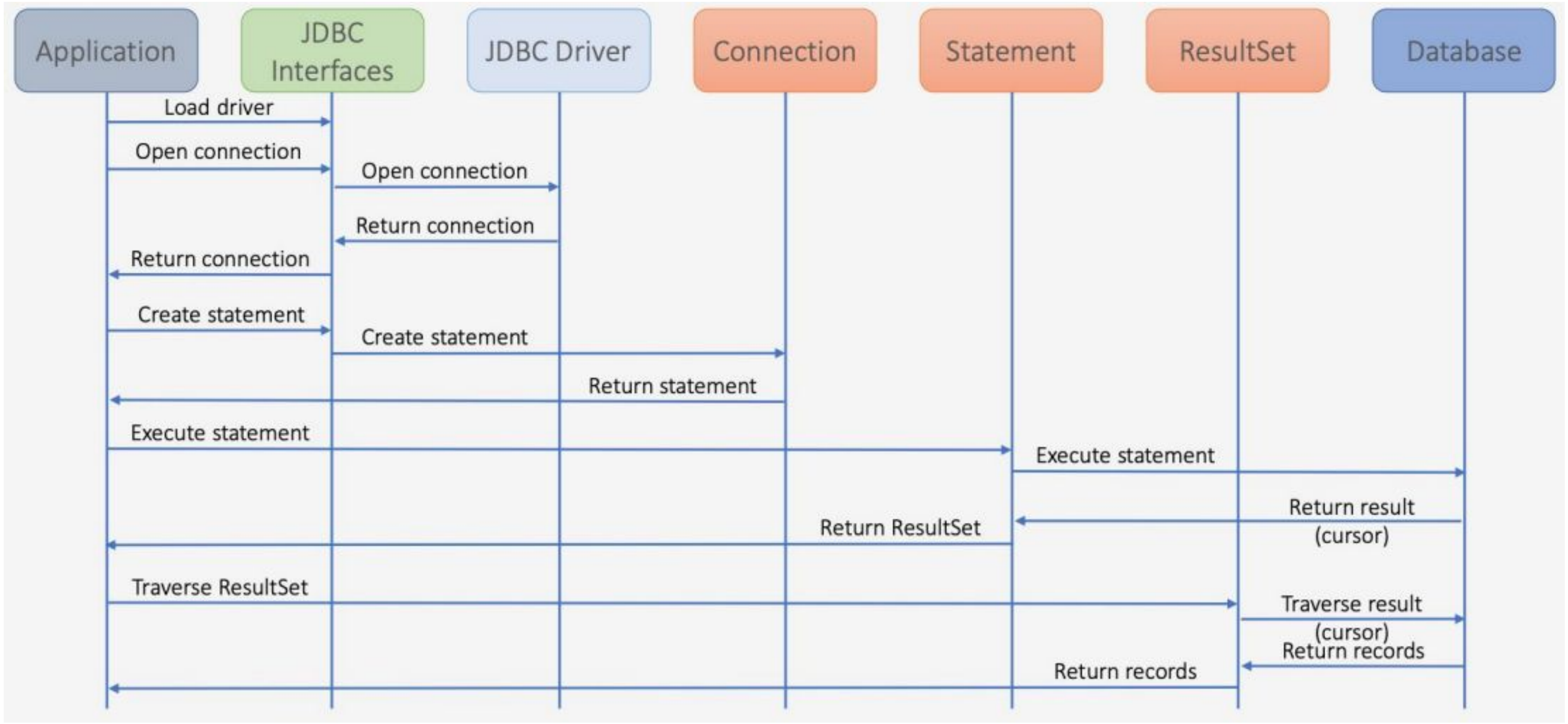
JDBC - Components

- **DriverManager**
  - Manages a list of database drivers. It matches connection requests from the java application with the proper database driver using communication subprotocols. The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database connection. More simply said, the DriverManager selects the proper driver to use in order to communicate with your designated database.
- **Driver**
  - It handles the communication with the database server. You will interact directly with Driver objects very rarely. Instead, you will use the DriverManager object, which manages the Drivers. The DriverManager also abstracts the details associated with working with Driver objects.
- **Connection**
  - It’s an interface declaring all of the methods needed for connecting to a database. Before you can read or write data from and to a database via JDBC, you need to open a connection to the database. All communications with the database will be done through the connection object only.

JDBC - Components

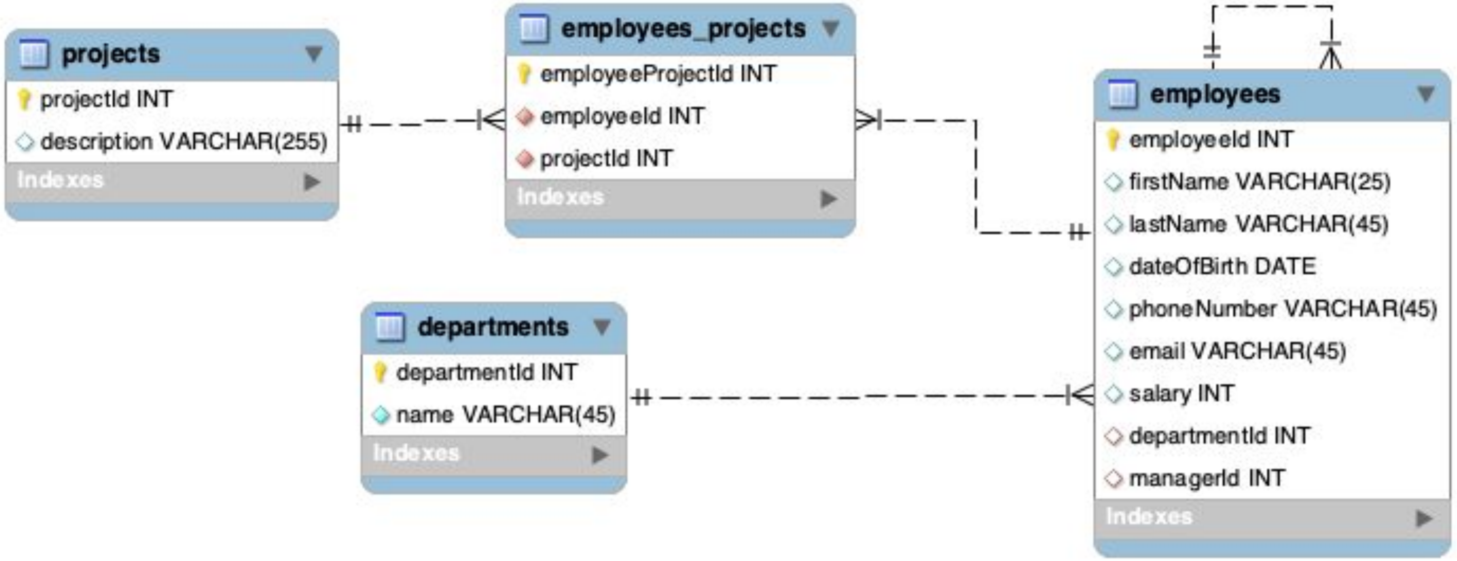
- **Statement**
  - It’s an interface used to execute SQL statements against a relational database. You can obtain a JDBC Statement from a JDBC Connection. Once you have a Java Statement instance, you can execute either a database query or a database update with it.
- **ResultSet**
  - The ResultSet interface represents the result of a database query. These objects store the data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data. A JDBC ResultSet contains records. Each record contains a set of columns.
- **SQLException**
  - It handles any errors that occur in a database application. When interacting with a database you might find yourself trying to execute unsupported operations like deleting a row which is referenced by a foreign key. This operation will throw a SQLException which you can then treat accordingly.

JDBC Component Interaction Diagram



Setup

- Database



Setup

- Database
    - **CREATE DATABASE** sda_database;
    - **USE** sda_database;
    -
    - Execute the script
        - https://gitlab.com/sda-international/program/common/databases/-/blob/master/humanResourcesDatabaseStructure.sql
        - Raw file
            - https://gitlab.com/sda-international/program/common/databases/-/raw/master/humanResourcesDatabaseStructure.sql

Setup

- Add a new Maven project on IntelliJ
- Add the dependency
  - In order for you to be able to use all of the functionality provided by JDBC you need to import the MySQL Connector library. MySQL Connector/J is the official JDBC driver for MySQL.
    - MySQL Connector/J
      - https://mvnrepository.com/artifact/mysql/mysql-connector-java
    - pom.xml

```
<dependencies>
<dependency>
   <groupId>mysql</groupId>
   <artifactId>mysql-connector-java</artifactId>
   <version>8.0.21</version>
</dependency>
</dependencies>
```

JDBC Architecture

- Read more
  - https://www.tutorialspoint.com/jdbc/jdbc-sql-syntax.htm
  - https://www.educba.com/jdbc-architecture/
  - https://www.progress.com/faqs/datadirect-jdbc-faqs/what-are-the-types-of-jdbc-drivers

JDBC - Open a Connection

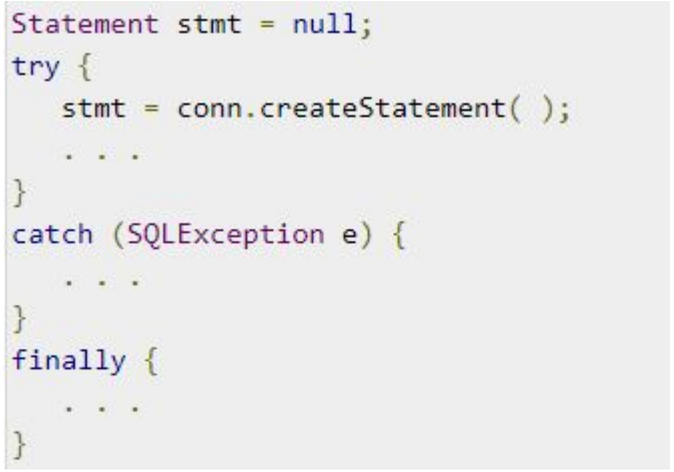
```java
public class DBUtil {

   public static Connection getDBConnection () {
        Connection connection = null;
        // JDBC driver name and database URL
        String dbUrl  = "jdbc:mysql://localhost:3306/sda_database?serverTimezone=UTC"  ;
        String driverJDBC  = "com.mysql.cj.jdbc.Driver" ;
        //  Database credentials
        String userDB  = "root";
        String passwordDB  = "12345678";

        try {
            // Register JDBC driver
            Class.forName(driverJDBC);
            // Connect to database
            if (connection == null || connection.isClosed()) {
                connection =  DriverManager.getConnection(dbUrl, userDB, passwordDB);
            }
        } catch (ClassNotFoundException  e) {
            e.printStackTrace();
        } catch (SQLException  e) {
            e.printStackTrace();
        }
        return connection;
   }
}
```

JDBC - Statement

- Obtaining a Connection object, it is called the **createStatement ()** method to obtain an object of type Statement
  - Statement stmt = con.createStatement()
- You can use methods like **execute ()**, **executeQuery ()**, **executeBatch ()** and **executeUpdate ()** to send instructions SQL to BD
- Subinterfaces:
  - PreparedStatement and CallableStatement
  - PreparedStatement pstmt = con.prepareStatement(...);
  - CallableStatement cstmt = con.prepareCall(...);



```
Statement stmt = null;
try {
   stmt = conn.createStatement( );
   . . .
}
catch (SQLException e) {
   . . .
}
finally {
   . . .
}
```

https://www.tutorialspoint.com/jdbc/jdbc-statements.htm

JDBC - Statement

- **Statement**
  - Use this for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters.
  - String query = "INSERT INTO person(id, name) VALUES(499,'Fernandosaurus')";
  - Statement statement = connection.**createStatement**(query);
- **PreparedStatement**
  - Use this when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.
  - String query = "INSERT INTO person(id, name) VALUES(?, ?)";
  - PreparedStatement preparedStatement = connection.**prepareStatement**(query);
- **CallableStatement**
  - Use this when you want to access the database stored procedures. The CallableStatement interface can also accept runtime input parameters.

JDBC - Statement

Once you've created a Statement object, you can then use it to execute an SQL statement with one of its three execute methods.

- **boolean execute** (String SQL): Returns a boolean value of true if a ResultSet object can be retrieved; otherwise, it returns false. Use this method to execute SQL DDL statements or when you need to use truly dynamic SQL.

- **int executeUpdate (**String SQL): Returns the number of rows affected by the execution of the SQL statement. Use this method to execute SQL statements for which you expect to get a number of rows affected - for example, an INSERT, UPDATE, or DELETE statement.

- **ResultSet executeQuery** (String SQL): Returns a ResultSet object. Use this method when you expect to get a result set, as you would with a SELECT statement.

JDBC - Statement

```
void saveEmployeeWithStatement() {
   String sql = "INSERT INTO employees (firstName, lastName, dateOfBirth,
phoneNumber, email, salary) " +
           "VALUES ('Matt', 'Matthews', '1980-01-01', '0800-800-800',
'm@matthews@gmail.com', '3000')";
   try {
       Statement stmt = DBUtil.getDBConnection().createStatement();
       Integer affectedRows = stmt.executeUpdate(sql);
   } catch (SQLException e) {
       e.printStackTrace();
   }
}
```

JDBC - PrepareStatement

```
public void saveEmployeeWithPreparedStatement(Employee employee) {
   String sql = "INSERT INTO employees (firstName, lastName, dateOfBirth, phoneNumber,
email, salary) " +
           "VALUES (?, ?, ?, ?, ?, ?)";
   try {
       PreparedStatement pstmt = DBUtil.getDBConnection().prepareStatement(sql);
       pstmt.setString(1, employee.getFirstName());
       pstmt.setString(2, employee.getLastName());
       pstmt.setDate(3, Date.valueOf(employee.getDateOfBirth()));
       pstmt.setString(4, employee.getPhoneNumber());
       pstmt.setString(5, employee.getEmail());
       pstmt.setInt(6, employee.getSalary());
       int affectedRows = pstmt.executeUpdate();
       System.out.println("Save with PreparedStatement: " + affectedRows);
   } catch (SQLException e) {
       e.printStackTrace();
   }
}
```

Update employee

```
public void updateEmployee(Employee employee) {
    String sql = "UPDATE employees SET firstName = ?, lastName = ?, dateOfBirth = ?, phoneNumber =
?, email = ?, salary = ? WHERE employeeId = ?" ;
    try {
        pst = DBUtil.getDBConnection().prepareStatement( sql);
        pst.setString(1, employee.getFirstName());
        pst.setString(2, employee.getLastName());
        pst.setDate(3, Date.valueOf(employee.getDateOfBirth()));
        pst.setString(4, employee.getPhoneNumber());
        pst.setString(5, employee.getEmail());
        pst.setInt(6, employee.getSalary());
        pst.setInt(7, employee.getEmployeeId());
        int resultSaved = pst.executeUpdate();
        System.out.println("Row saved: " + resultSaved);
    } catch (SQLException throwables) {
        throwables.printStackTrace();
    }
}
```

Delete employees

```
public void deleteEmployee (int idEmployee) {
   String sql  = "DELETE FROM employees WHERE employeeId = ?" ;
   try {
       pst = DBUtil.getDBConnection().prepareStatement( sql);
       pst.setInt(1, idEmployee);
       int resultSaved = pst.executeUpdate();
       System.out.println("Row saved: " + resultSaved);
   } catch (SQLException throwables) {
       throwables.printStackTrace();
   }
}
```

ResultSet

- In order for you to iterate and extract the data from the ResultSet you need to:
  - Use an iterator and its next() method to go through each of the rows returned by the query
  - 2. Extract the data from each row using the get() method and the type of the expected data

```
public Employee searchEmployeeById(int employeeId) {
    String sql = "SELECT * FROM employees WHERE employeeId = ?" ;
    try {
        PreparedStatement pstmt  = DBUtil.getDBConnection().prepareStatement(sql);
        pstmt.setInt(1, employeeId);
        ResultSet resultSet = pstmt.executeQuery();

        if(resultSet.next()) {
            Employee emp = new Employee()
            emp.setFirstName(resultSet.getString("firstName"));
            emp.setLastName(resultSet.getString("lastName"));
            emp.setDateOfBirth(resultSet.getDate("dateOfBirth").toString());
            emp.setPhoneNumber(resultSet.getString("phoneNumber"));
            emp.setEmail(resultSet.getString("email"));
            emp.setSalary(resultSet.getInt("salary"));
            emp.setEmployeeId(resultSet.getInt("employeeId"));
        }
    } catch (SQLException throwables) {
        throwables.printStackTrace();
    }
    return emp;
```

List All employees

```
public List<Employee> listAllEmployee () {
   String sql = "SELECT * FROM employees" ;
   List<Employee> listE = new ArrayList<Employee>();

   try {
       pst = DBUtil.getDBConnection().prepareStatement( sql);
       ResultSet rs = pst.executeQuery();
       while (rs.next()) {
           Employee employee = new Employee();
           employee.setFirstName(rs.getString("firstName"));
           employee.setLastName(rs.getString("lastName"));
           employee.setPhoneNumber(rs.getString("phoneNumber"));
           employee.setDateOfBirth(rs.getDate("dateOfBirth").toString());
           employee.setEmail(rs.getString("email"));
           employee.setSalary(rs.getInt("salary"));
           employee.setEmployeeId(rs.getInt("employeeId"));
           listE.add(employee);
       }
   } catch (SQLException throwables) {
       throwables.printStackTrace();
   }
   return listE;
}
```

List by firstName

```
public List<Employee> searchEmployeeByFirstName (String firstName) {
   Employee emp = new Employee();
   List<Employee> listEmp = new ArrayList<Employee>();
   String sql = "SELECT * FROM employees WHERE firstName = ?" ;
   try {
       PreparedStatement pstmt = DBUtil.getDBConnection().prepareStatement( sql);
       pstmt.setString(1, firstName);
       ResultSet resultSet = pstmt.executeQuery();

       while(resultSet.next()) {
           emp.setFirstName(resultSet.getString("firstName"));
           emp.setLastName(resultSet.getString("lastName"));
           emp.setDateOfBirth( resultSet.getDate("dateOfBirth").toString());
           emp.setPhoneNumber( resultSet.getString("phoneNumber"));
           emp.setEmail(resultSet.getString("email"));
           emp.setSalary(resultSet.getInt("salary"));
           emp.setEmployeeId(resultSet.getInt("employeeId"));
           listEmp.add(emp);
       }

   } catch (SQLException throwables) {
       throwables.printStackTrace();
   }
   return listEmp;
}
```

JDBC - Exercises 1

1. Display all projects (projectId, description)
2. Display all employees (employeeId, firstName, lastName, dateOfBirth)
3. Display all employees with names starting with the letter J (employeeId, firstName, lastName, dateOfBirth)
4. Display all employees that haven't been assigned to a department
5. Display all employees along with the department they're in (employeeId, firstName, lastName, dateOfBirth, departmentName)