Welcome to the JDBC and Hibernate module!

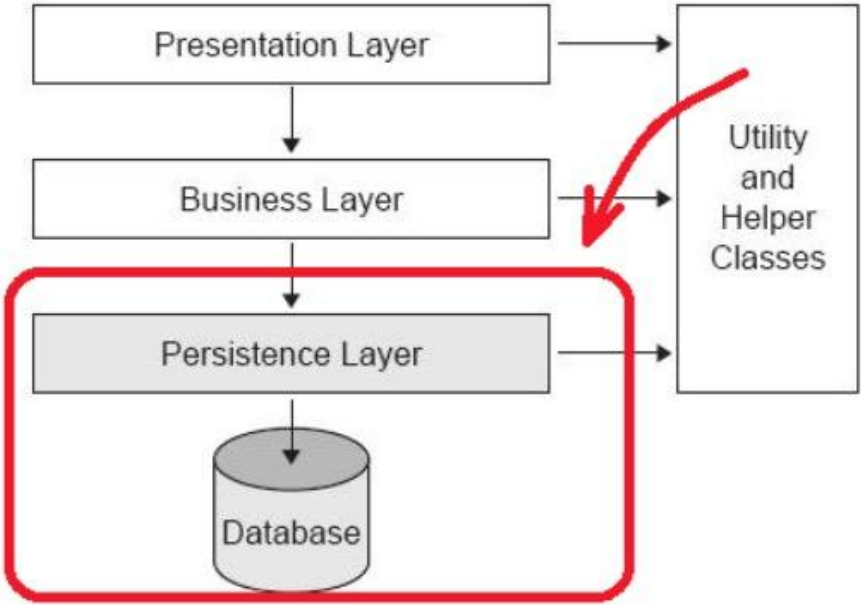
Trainer: Diana Cavalcanti

Setup

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

Persistence layer

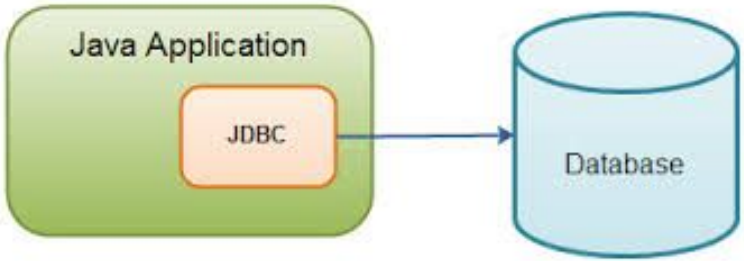
- It is the layer that deal with the database
- It consist of a set of classes that maps the database and all operation on that table
- DAO vs Repository
  - https://www.baeldung.com/java-dao-vs-repository
  - https://www.baeldung.com/java-dao-pattern



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

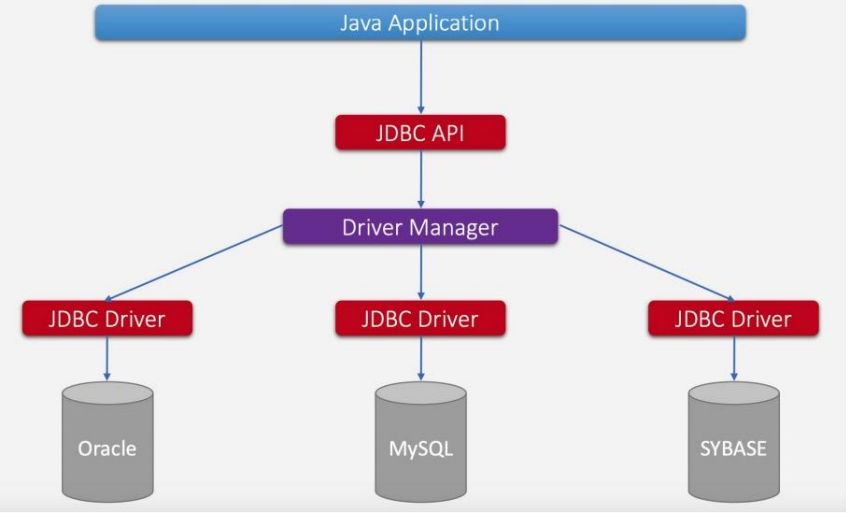
JDBC

- 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.

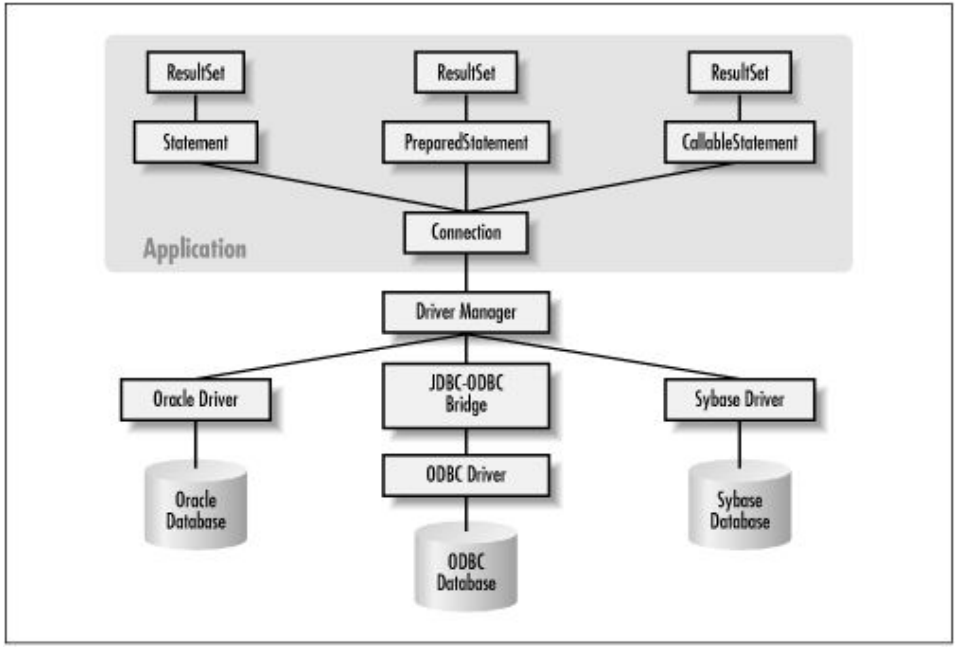
JDBC Architecture

- 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



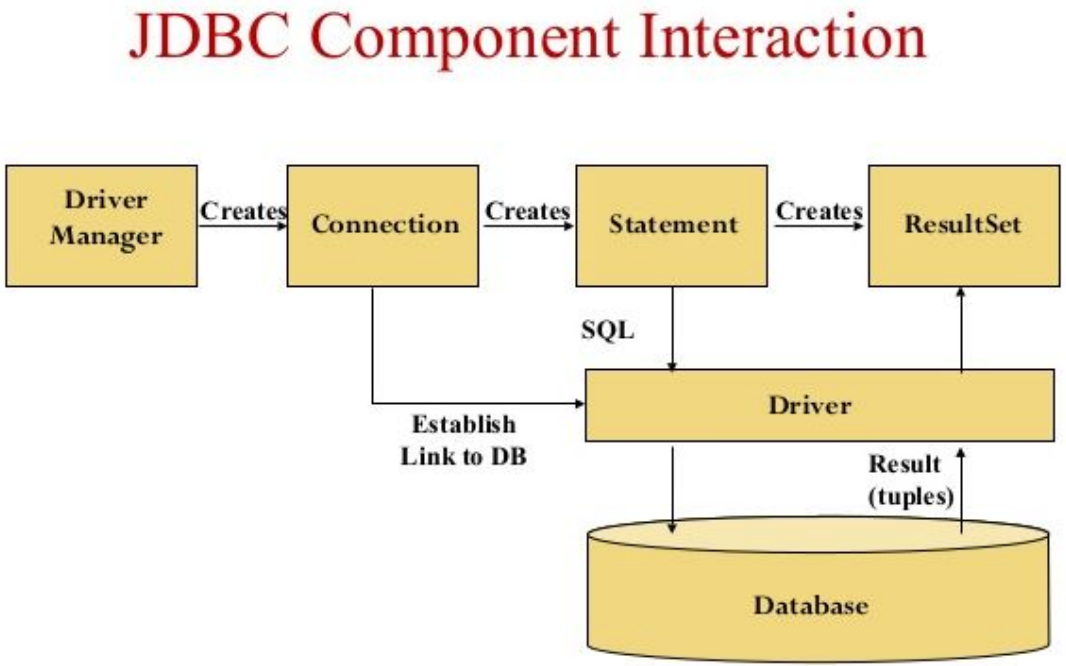
JDBC Architecture

- **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**



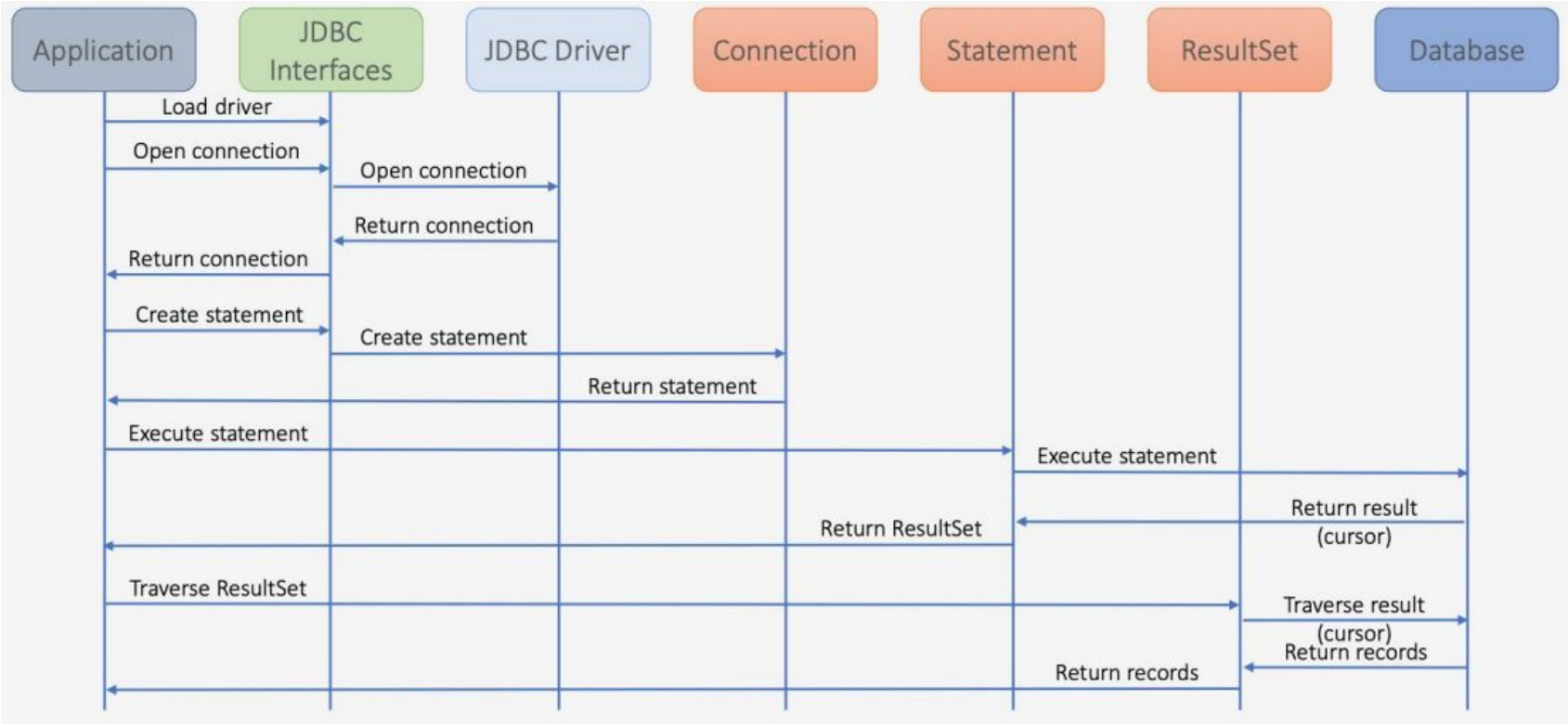
JDBC - Components

- **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

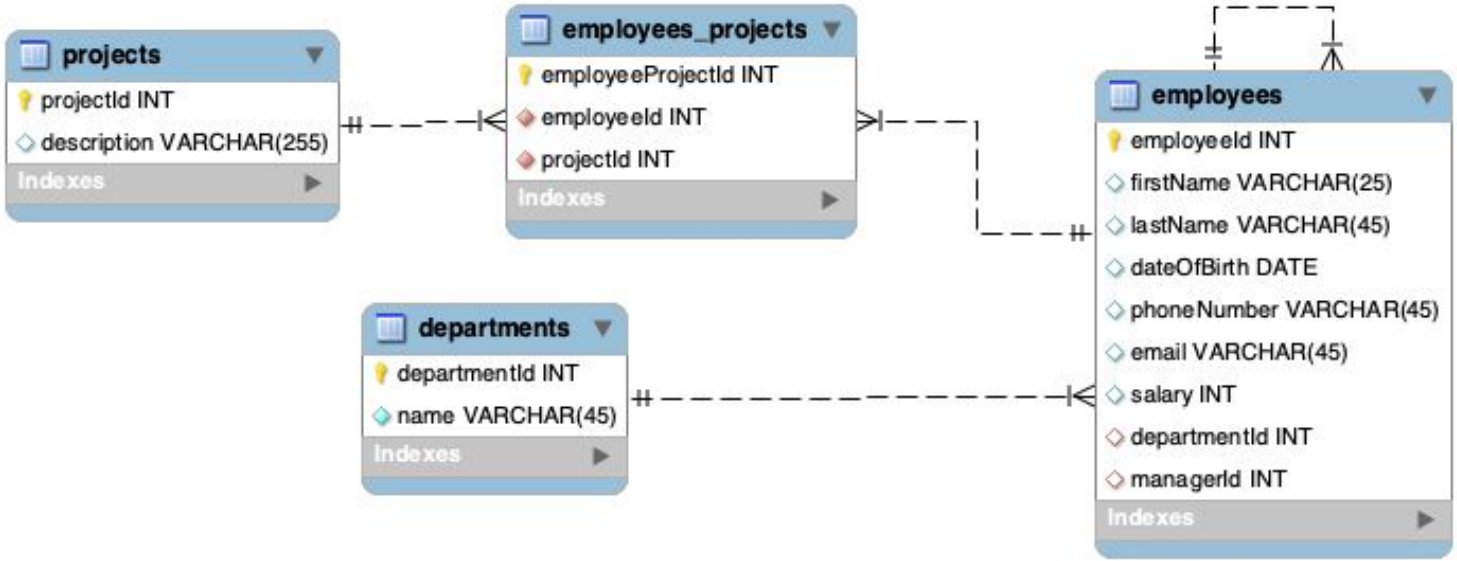
- **Statement**
  - It’s an interface used to execute SQL statements against a relational database. You can obtain a JDBC Statement from a JDBC Connection. Once you have a Java Statement instance, you can execute either a database query or a database update with it.
- **ResultSet**
  - The ResultSet 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

```
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)

Select with JOIN

```
public List<EmployeeDepartment> listEmployeeWithDepartmentName () {
   String sql = "SELECT e.employeeId as empId , e.firstName as fName, d.name as dName "   +
                "FROM employees e "  +
                "INNER JOIN departments d ON e.departmentId = d.departmentId "  ;
   List<EmployeeDepartment> listE = new ArrayList<EmployeeDepartment>();
   EmployeeDepartment  empDepartment = null;

   try {
       pst = connection.prepareStatement(sql);
       rs = pst.executeQuery();
       while(rs.next()) {
           empDepartment =  new EmployeeDepartment();
           empDepartment.setEmployeeId( rs.getInt("empId"));
           empDepartment.setEmployeeFirstName( rs.getString("fName"));
           empDepartment.setDepartmentName( rs.getString("dName"));
           listE.add(empDepartment);
       }
       connection.commit();
   } catch (SQLException throwables) {
       throwables.printStackTrace();
   }
   return listE;
}
```

Best Practices - Code Structure

Regarding the code structure, in order to have a clean and easy to use application you should:

1. Create separate classes that will interact with the database for separate tables:
    a. DepartmentRepository will define the findAll, findById, save, delete, etc methods for the Departments table;
    b. ProjectRepository will define the findAll, findById, save, delete, etc methods for the Projects table;
2. Create a common DatabaseUtils class that will store the database connection details

3. Pick a standard and use it throughout the classes - be consistent
    a. consistent naming: don’t call a method getAll in one repository and findAll in another
    b. consistent returns types: findAll should either return a List or a Set of objects; Don’t make it return a List in the DepartmentRepository and a Set in the ProjectRepository
    c. consistent parameter types: save should either take as parameters the object that it needs to save (Department department) or the list of properties (departmentId, name, etc)

JDBC - executeQuery vs executeUpdate

Depending on the type of query that you want to run on the database, you will need to call different methods on the Statement object:

- *executeQuery()* method executes statements that return a result set by fetching data from the database. It executes only select statements.
- *executeUpdate()* method execute statements that insert / update/ delete data from the database. This method return an int value representing number of records affected. It executes only non-select statements.

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM
departments");

while(rs.next()) {
   Integer deptId = rs.getInt("departmentId");
   String deptName = rs.getString("name");

   System.out.println(deptId + " " + deptName);
}
```

```
Statement stmt = conn.createStatement();
Integer affectedRows = stmt.executeUpdate("UPDATE
departments SET name = 'HumanResources' WHERE
departmentId = 1");

System.out.println(affectedRows);
```

JDBC - Statement vs PreparedStatement

When you have multiple parameters by which you're building your query it would be advisable to use a PreparedStatement instead of a regular Statement.

The provided params need to be represented in the query as **?** and they'll be set on it using the *setString(), setInt(),* etc methods.

```
String insertIntoEmployees = "INSERT INTO employees (firstname,
lastname, dateOfBirth, phoneNumber, email, salary) values ("
        + "'Matt', "
        + "'Matthews', "
        + "'1980-01-01', "
        + "'0800-800-800', "
        + "'m@matthews@gmail.com', "
        + "'3000')";

Statement stmt = conn.createStatement();
Integer affectedRows = stmt.executeUpdate(insertIntoEmployees);
```

```
String insertIntoEmployees = "INSERT INTO employees (firstname,
lastname, dateOfBirth, phoneNumber, email, salary) values (?, ?, ?,
?, ?, ?)";

PreparedStatement stmt =
conn.prepareStatement(insertIntoEmployees);

stmt.setString(1,"Matt");
stmt.setString(2,"Matthews");
stmt.setDate(3, Date.valueOf("1980-01-01"));
stmt.setString(4, "0800-800-800");
stmt.setString(5,"m@matthews.gmail.com");
stmt.setInt(6,3000);
```

DAY 2

Hibernate

Hibernate

Working with both **Object-Oriented** software and **Relational Databases** can be **cumbersome** and time **consuming**.

Hibernate is an Object/Relational Mapping (ORM) solution for Java environments. The term ORM refers to the technique of **mapping data** between an object model representation to a relational data model representation.

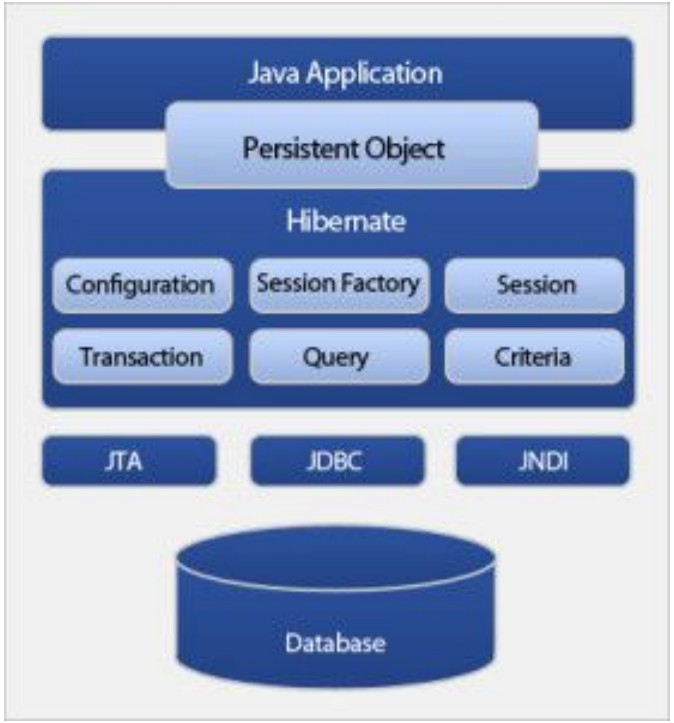
Hibernate

- **Hibernate** takes care of the **mapping** from J**ava classes** to **database tables**, and from Java data types to SQL data types.
- It provides **storing** and **retrieving** Java **objects** directly **to** and **from** the database.
- It can significantly **reduce development time** otherwise spent with manual data handling in SQL and JDBC.

Hibernate Architecture

- **Hibernate**, as an ORM solution, effectively "sits between" the Java application data access layer and the Relational Database.
- The Java application makes use of the Hibernate APIs to **load, store, query** its **domain data**.
- **Hibernate** uses various existing Java APIs like **JDBC** for **connecting to and executing** operations on the **database**.
- It uses a persistence context where **Java objects mirroring database tables** live. Using its various interfaces, Hibernate will then **only operate on the Java objects** and **not on the database tables**.

Hibernate Architecture



Hibernate - Components

- **Configuration**
  - The Configuration object is the first Hibernate object you create in any Hibernate application. It is usually created only once during application initialization.
  - The Configuration object provides two key components: **the database connection details** (configuring which database to use) and **the class mapping setup** (configuring which tables to map to which Java objects).
- **SessionFactory**
  - The SessionFactory is a thread-safe and immutable representation of the mapping of the application domain model to a database.
  - The SessionFactory is a heavyweight object so for **any given database**, the application should have **only one associated SessionFactory**.
  - The SessionFactory maintains services that Hibernate uses across all Session(s)

Hibernate - Components

- **Session**
  - A **Session** is used to get a **physical connection** with a database. The Session object is lightweight and designed to be **instantiated each time an interaction is needed** with the database. Persistent objects are **saved and retrieved** through a Session object.
  - The session objects should not be kept open for a long time because they are not thread safe and they should be created and destroyed them as needed.
- **Transaction**
  - A **Transaction** represents a unit of work with the database and most of the RDBMS supports transaction functionality. Transactions in Hibernate are handled by an underlying transaction manager (from JDBC or JTA).
- **Query**
  - **Query** objects use SQL or Hibernate Query Language (HQL) string to **retrieve** data from the database and **create objects**. A Query instance is used to bind query parameters, limit the number of results returned by the query, and finally to execute the query.

Hibernate Domain Model - How To

- **Hibernate Domain Model**
  - The **Domain model** is the central character in an ORM. It is, simply said, the representation of the **database model** using **Java objects**.
  - Hibernate will therefore **interact and execute queries** on these **Java objects** instead of the database tables. Hibernate works best if the objects follow the Plain Old Java Object structure.
  - When creating the Hibernate Domain Model you need to think about the database tables and the relationships between them. Usually, one table is mapped to **one Java class** and the relationship between tables is mapped through **composition**.

Hibernate Domain Model - How To

- Consider the two tables:
  - Departments (departmentId, name)
  - Employees (employeeId, firstname, lastname) in a one-to-many relationship.
- You would therefore have an Employee class with the three properties and a Department class with the two properties.
- In order to map the relationship you can create a List in the Department class (representing the employees that belong to that particular department).
- The Employee would have a Department property (representing the department that the employee belongs to).

Hibernate Domain Model - How To

- The first thing you need to in order to create the domain model is to create the class hierarchy.
- Start by creating classes that represent specific tables and map table columns to class properties.
- Classes and properties are mapped via annotations.
- Once all the tables have been mapped, proceed to mapping the relationships.
- Add Sets or Object properties where tables are connected.

Hibernate - Application - Setup

- Add a new Maven project on IntelliJ
- Add the dependencies on pom.xml
  - In order for you to be able to use all of the functionality you need to import the MySQL Connector library. MySQL Connector/J is the official JDBC driver for MySQL. And the Hibernate Core library.

    - MySQL Connector/J
      - https://mvnrepository.com/artifact/mysql/mysql-connector-java
    - Hibernate
      - https://mvnrepository.com/artifact/org.hibernate/hibernate-core
    - pom.xml

```
<dependencies>
<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-core -->
<dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate-core</artifactId>
   <version>5.4.24.Final</version>
</dependency>
<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
   <dependency>
       <groupId>mysql</groupId>
       <artifactId>mysql-connector-java</artifactId>
       <version>8.0.21</version>
   </dependency>
</dependencies>
```

Hibernate - Setup

- In resources folder add a new folder:
  - META-INF
- In META-INF folder add a xml file:
  - persistence.xml

Hibernate - Setup

- **The persistence.xml - Copy the content to file persistence.xml**

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
        http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
        version="2.1">

  <persistence-unit name="PERSISTENCE_MYSQL">
    <description> Hibernate JPA Configuration Example</description>
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <properties>
      <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver" />
      <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL8Dialect"/>
      <property name="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/sda_hibernate?createDatabaseIfNotExist=true  &amp;serverTimezone=UTC
"/>
      <property name="javax.persistence.jdbc.user" value="root" />
      <property name="javax.persistence.jdbc.password" value="12345678" />
      <property name="hibernate.show_sql" value="true" />
      <property name="hibernate.format_sql" value="true" />
      <property name="hibernate.hbm2ddl.auto" value="update" />
      <!-- <property name="hibernate.hbm2ddl.auto" value="create" /> /> -->
      <!-- <property name="hibernate.hbm2ddl.auto" value="validate" /> /> -->
    </properties>
  </persistence-unit>
</persistence>
```

Hibernate - Setup

- Add a new java package in src named util
- Add a new Java class DBUtil.java

```
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class DBUtil {

   private static final String PERSISTENCE_UNIT_NAME = "PERSISTENCE_MYSQL";
   private static EntityManagerFactory factory;

   public static EntityManager getEntityManager() {
       if (factory == null) {
           factory = Persistence.createEntityManagerFactory(PERSISTENCE_UNIT_NAME);
       }
       return factory.createEntityManager();
   }

   public static void shutdown() {
       if (factory != null) {
           factory.close();
       }
   }
}
```

Hibernate - Create the domain model

- In order for you to create the domain model you need to create the classes that will map the databases table. T
- he domain model classes are regular POJOs, classes with just properties and getters and setters.
- After you have the class, you need to add the annotations.
- The only required annotations are @Entity, @Id, @GeneratedValue.
- Annotations like @Table or @Column are only necessary if the class/property names are different than the table/column.

Common Annotations

- The most common annotations are:
  - **@Entity** - states that a class is mapped to a table. Each entity must have a no-arg constructor and a primary key
  - **@Table** - is an optional annotation and defines which table the class is mapping. If the table name matches the class name then you don’t need to declare it
  - **@Id** - marks the property of the class that represents the primary key
  - **@GeneratedValue** - is set on the primary key property and defines the way its value will be generated. If you want to use an auto-incremented database column to generate your primary key values, you need to set the strategy to GenerationType.IDENTITY
  - **@Column** - is an optional annotation and specifies which table column the property is mapping. If the property name matches the column name then you don’t need to declare it
  - **@Transient** - sometimes, you may want to make a field non-persistent. It specifies that the field will not be persisted in the database
  - **@Enumerated** - enables you to define how an enum attribute gets persisted in the database

https://thorben-janssen.com/key-jpa-hibernate-annotations/

Hibernate - Create the domain model

```
import java.util.List;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
@Table(name = "employee") //  optional - identical names
public class Employee {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   @Column(name = "employeeId") // optional - identical names
   private int employeeId;
   @Column(name = "firstName") // optional - identical names
   private String firstName;
   private String lastName;
   private String dateOfBirth;
   private String phoneNumber;
   private String email;
   private int salary;

//Gets and Sets
}
```

EmployeeRepository - save employee

```
public class EmployeeRepository {
     private final EntityManager em;

     public EmployeeRepository() {
        this.em = DBUtil.getEntityManager();
     }

     public void saveEmployee(Employee employee) {
        try {
            this.em.getTransaction().begin();
            this.em.persist(employee);
            this.em.getTransaction().commit();
        } catch (Exception e) {
            e.printStackTrace();
            this.em.getTransaction().rollback();
        }
     }

}
```

EmployeeRepository - update employee

```
public void updateEmployeeById(Employee employee) {
    try {
        this.em.getTransaction().begin();
        this.em.merge(employee);
        this.em.getTransaction().commit();
    } catch (Exception e) {
        this.em.getTransaction().rollback();
    }
}
```

EmployeeRepository - delete employee

```
public void updateEmployeeById(Employee employee) {

    try {

        this.em.getTransaction().begin();

        this.em.remove(em.merge(employee));

        this.em.getTransaction().commit();

    } catch (Exception e) {

        this.em.getTransaction().rollback();

    }

}
```

EmployeeRepository

- List all employees

```
public List<Employee> listAllEmployees() {
    String sql = "FROM Employee";
    return em.createQuery(sql).getResultList();
}
```

- Search employee by ID

```
public Employee searchEmployeeById(int employeeId) {
    return em.find(Employee.class, employeeId);
}
```

DAY 3

Hibernate - Relationships

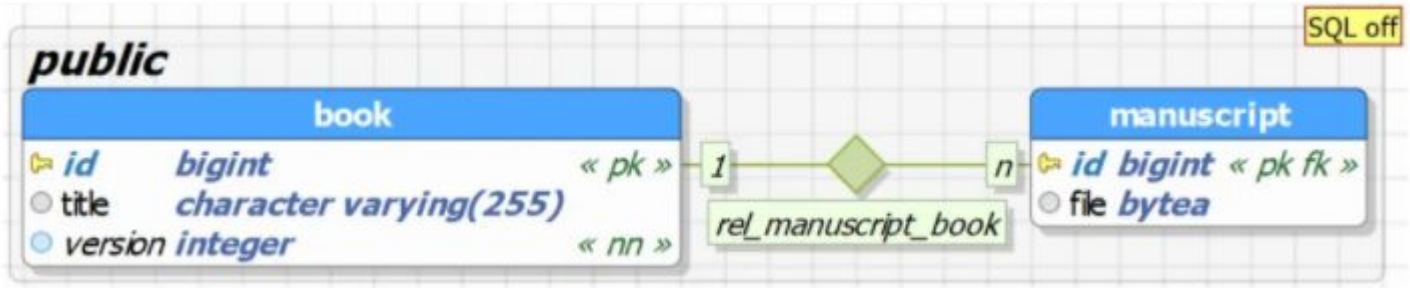
- **Understanding Entities and Their Associations**
  - **Entities can contain references to other entities**, either directly as an embedded property or field, or indirectly via a collection of some sort (arrays, sets, lists, etc.).
  - These associations are represented using **foreign key relationships** in the underlying tables. These foreign keys will rely on the identifiers used by participating tables.
  - When **only one of the pair of entities** contains a reference to the other, the association is **unidirectional**. If the **association is mutual**, then it is referred to as **bidirectional**.
  - You can establish either unidirectional or bidirectional i.e you can either model them as an attribute on only one of the associated entities or on both. It will not impact your database mapping tables, but it defines in which direction you can use the relationship in your model and criteria queries.
  - `@OneToOne`
  - `@OneToMany`
  - `@ManyToOne`
  - `@ManyToMany`
  -

Relationships - One-to-One

- **One-to-One**
  - Defines a single-valued association to another entity that has one-to-one multiplicity.
  - If the relationship is bidirectional, the non-owning side must use the mappedBy element of the OneToOne annotation to specify the relationship field or property of the owning side.
  - The OneToOne annotation may be used within an embeddable class to specify a relationship from the embeddable class to an entity class.

  - https://www.baeldung.com/jpa-one-to-one
  - https://docs.jboss.org/hibernate/jpa/2.1/api/javax/persistence/OneToOne.html
  - https://thorben-janssen.com/hibernate-tips-same-primary-key-one-to-one-association/

Relationships - One-to-One




```java
@Entity
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    @SequenceGenerator(name = "book_seq")
    private Long id;

    private String title;

    @OneToOne(mappedBy = "book")
    private Manuscript manuscript;

    ...
}
```

```java
@Entity
public class Manuscript {

    @Id
    private Long id;

    private byte[] file;

    @OneToOne
    @JoinColumn(name = "id")
    @MapsId
    private Book book;

    ...
}
```

Relationships - Many-to-One/One-to-Many

- In order to map a One-To-Many relationship there are three annotations that you need to consider:
  - **@ManyToOne** - set on the many side (many employees in one department) @JoinColumn - set on the many side ○ marks the column used as a foreign key (the departmentId column from the Employee table)
  - **@OneToMany** - set on the one side (one department has multiple employees) ○ marks the property used as a reference (the department property on the Employee class)
  -

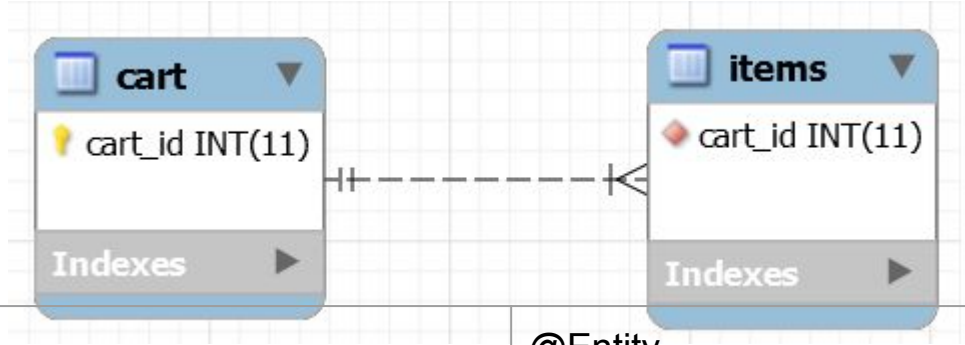
https://www.baeldung.com/hibernate-one-to-many

https://examples.javacodegeeks.com/enterprise-java/hibernate/hibernate-one-many-example/

https://www.javatpoint.com/hibernate-one-to-many-mapping-using-annotation-example

https://medium.com/@rajibrath20/the-best-way-to-map-a-onetomany-relationship-with-jpa-and-hibernate-dbbf6dba00d3

Hibernate - Many-to-One/One-to-Many




```
@Entity
@Table(name="CART")
public class Cart {

    //...

    @OneToMany(mappedBy="cart")
    private Set<Items> items;

    // getters and setters
}
```

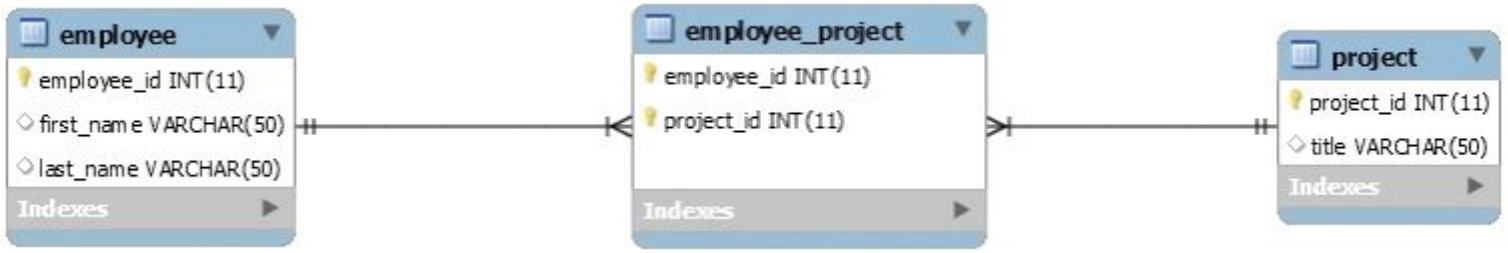
```
@Entity
@Table(name="ITEMS")
public class Items {

    //...
    @ManyToOne
    @JoinColumn(name="cart_id", nullable=false)
    private Cart cart;

    public Items() {}

    // getters and setters
}
```

Hibernate - Many-to-Many

- **Relationships -**
  - A many-to-many relationship means that multiple rows in a table are mapped to multiple rows in another table. Take for example an employee that works on multiple projects while there can be multiple employees working on the same project.
  - The way that this is mapped at the database level is that you have an intermediary table which contains foreign keys referencing the primary keys in the main tables

- https://www.baeldung.com/hibernate-many-to-many
- https://www.baeldung.com/jpa-many-to-many
- https://vladmihalcea.com/the-best-way-to-map-a-many-to-many-association-with-extra-columns-when-using-jpa-and-hibernate/
- https://docs.jboss.org/hibernate/jpa/2.1/api/javax/persistence/ManyToMany.html

Hibernate - Many-to-Many

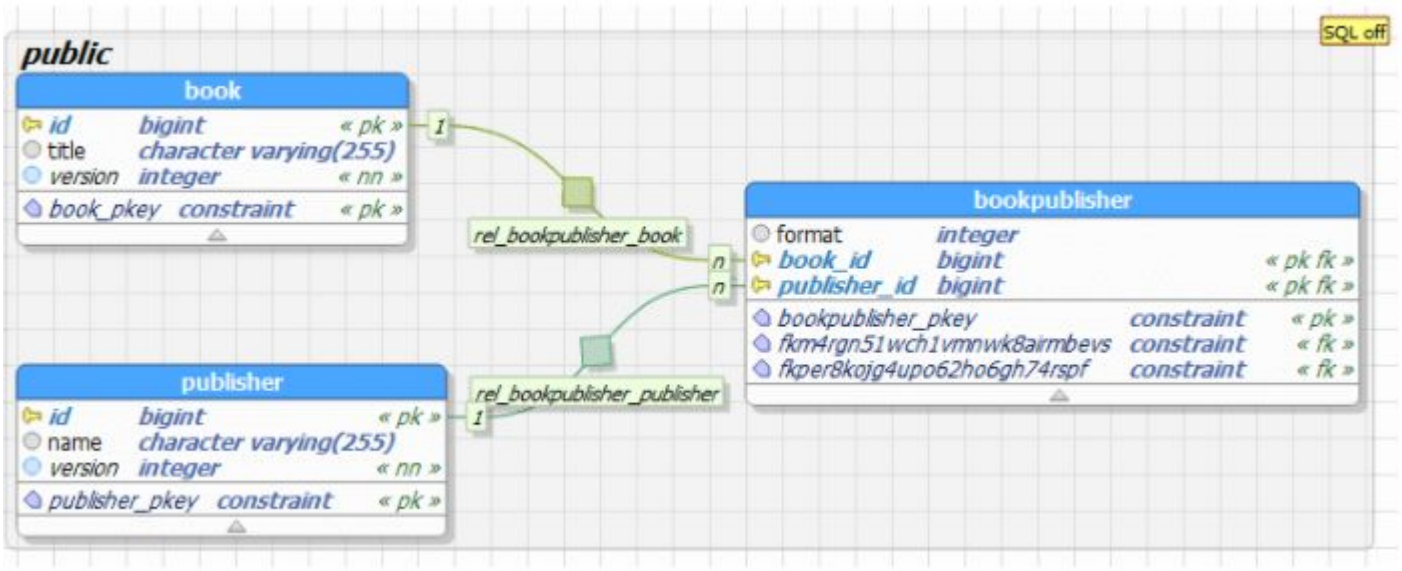



```
@Entity
@Table(name = "Employee")
public class Employee {
    // ...

    @ManyToMany(cascade = { CascadeType.ALL })
    @JoinTable(
        name = "Employee_Project",
        joinColumns = { @JoinColumn(name = "employee_id") },
        inverseJoinColumns = { @JoinColumn(name = "project_id") }
    )
    Set<Project> projects = new HashSet<>();

    // standard constructor/getters/setters
}
```

```
@Entity
@Table(name = "Project")
public class Project {
    // ...

    @ManyToMany(mappedBy = "projects")
    private Set<Employee> employees = new
HashSet<>();

    // standard constructors/getters/setters
}
```

Hibernate - Many-to-Many

- **Many-to-Many Association with additional Attributes**



https://thorben-janssen.com/hibernate-tip-many-to-many-association-with-additional-attributes/

Hibernate - Many-to-Many

- **Many-to-Many Association with additional Attributes**

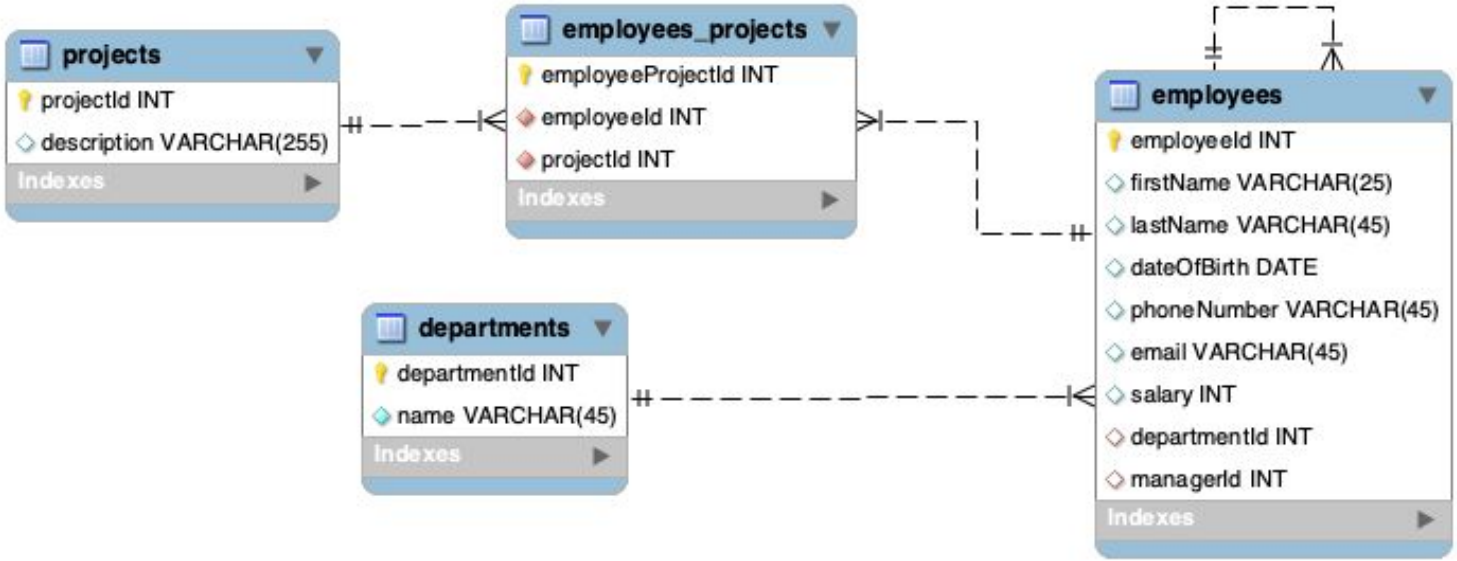
```
@Entity
class BookPublisher {

    @EmbeddedId
    private BookPublisherId id =
new BookPublisherId();

    @ManyToOne
    @MapsId("bookId")
    private Book book;

    @ManyToOne
    @MapsId("publisherId")
    private Publisher publisher;

    private Format format;

    ...
}
```

```
@Embeddable
public static class BookPublisherId
implements Serializable {

    private static final long serialVersionUID
= 1L;

    private Long bookId;
    private Long publisherId;


//Gets and Sets
//hashCode()
//equals
```

```
@Entity
public class Book {

    @Id
    @GeneratedValue
    private Long id;

    @OneToMany(mappedBy = "publisher")
    private Set<BookPublisher>
bookPublishers = new HashSet<>();

    ...
}


@Entity
public class Publisher {

    @Id
    @GeneratedValue
    private Long id;

    @OneToMany(mappedBy =
"publisher")
    private Set<BookPublisher>
bookPublishers = new HashSet<>();

    ...
}
```

Setup

- Database



```
@Entity
public class Departments {


    @OneToMany(mappedBy = "departments")
    private List<Employee> comments = new ArrayList<>();
}
```

```
@Entity
public class Employee {


    @ManyToOne(cascade = {CascadeType.ALL})
    @JoinColumn(name = "managerId")
    private Employee manager;

    @OneToMany(mappedBy = "manager", cascade = {CascadeType.ALL})
    private List<Employee> subordinates;

    @ManyToOne(cascade = {CascadeType.ALL})
    @JoinColumn(name = "departmentId")
}
```

```
@Entity
public class EmployeeProject {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int employeeProjectId;

    @ManyToOne(cascade = {CascadeType.ALL})
    @JoinColumn(name = "employeeId")
    private Employee employee;

    @ManyToOne(cascade = {CascadeType.ALL})
    @JoinColumn(name = "projectId")
    private Projects project;

}
```

Hibernate - Queries

- Hibernate Query Language (HQL) is an object-oriented query language, similar to SQL, but instead of operating on **tables** and **columns**, HQL works with persistent **objects (classes)** and their **properties (classes attributes)**.
- HQL queries are translated by Hibernate into conventional SQL queries, which in turn perform actions on the database.

Hibernate Query Language

- Hibernate Query Language (HQL) is an object-oriented query language, similar to SQL, but instead of operating on tables and columns, HQL works with persistent objects and their properties.
- HQL queries are translated by Hibernate into conventional SQL queries, which in turn perform actions on the database.
- HQL syntax is very similar so SQL but it runs on Java objects instead of database tables.
- Keywords like SELECT, FROM, and WHERE, etc., are not case sensitive, but properties like table and column names are case sensitive in HQL.
- The SELECT keyword is optional in HQL

https://www.tutorialspoint.com/hibernate/hibernate_query_language.htm

Hibernate Query Language

- The query interface provides many methods. There is given commonly used methods:
  1. **public int executeUpdate()** is used to execute the update or delete query.
  2. **public List list()** returns the result of the ralation as a list.
  3. **public Query setFirstResult(int rowno)** specifies the row number from where record will be retrieved.
  4. **public Query setMaxResult(int rowno)** specifies the no. of records to be retrieved from the relation (table).
  5. **public Query setParameter(int position, Object value)** it sets the value to the JDBC style query parameter.
  6. **public Query setParameter(String name, Object value)** it sets the value to a named query parameter.

HQL and JPQL

- The Hibernate Query Language (HQL) and Java Persistence Query Language (JPQL) are both object model focused query languages similar in nature to SQL. JPQL is a heavily-inspired-by subset of HQL. A JPQL query is always a valid HQL query, the reverse is not true however.
- https://docs.jboss.org/hibernate/core/3.3/reference/en/html/queryhql.html
- https://docs.jboss.org/hibernate/orm/4.3/devguide/en-US/html/ch11.html#:~:text=The%20Hibernate%20Query%20Language%20(HQL,reverse%20is%20not%20true%20however.
- https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/chapters/query/hql/HQL.html

HQL - Examples

```
//      public void list() {
//              EntityManager entityManager = EntityManagerUtil.getEntityManager();
                String sql = "from Employee  ep";
//
//              //Form 1
//              TypedQuery<Employee> tp =entityManager.createQuery(sql, Employee.class);
//              List<Employee> x = tp.getResultList();
//
//              //Form 2
//              List<Employee  > t = entityManager.createQuery(sql).getResultList();
//
//              //Form 3
//              Query tt= entityManager.createQuery(sql);
//              List<?> g = tt.getResultList();
//              for (Object obj : g) {
//                              System.out.println(((Employee) obj).getFirstName());
//              }
//
//              List<Employee> ttt = tt.getResultList();
//              for (Employee emp: ttt) {
//              }
```

EmployeeRepository - HQL

- **Search employee by firstName (parameter)**

```
public List<Employee> searchEmployeeByFirstName(String firstName) {
   String sql = "FROM Employee A WHERE A.firstName = :firstName";

   return em.createQuery(sql)
           .setParameter("firstName", firstName).getResultList();
}
```

- **Search employee by firstName and ID (parameter)**

```
public List<Employee> searchEmployeeByFirstName(String firstName, int id) {
   String sql = "FROM Employee A WHERE A.firstName = :firstName AND A.employeeId = :id";

   return em.createQuery(sql)
           .setParameter("firstName", firstName)
           .setParameter("id", id).getResultList();
}
```

EmployeeRepository - HQL

- **Update salary employee by percent**

```
public void updateSalaryByPercent (int salary) {
    em.getTransaction().begin();
    try{
        String sql  = "UPDATE Employee E SET E.salary = E.salary + (( E.salary * 10 ) / 100) "   +
                "  WHERE E.salary < :salary" ;
        int result   = em.createQuery(sql)
                .setParameter( "salary",salary )
                .executeUpdate();
        System.out.println("Changed rows: "  + result);

     em.getTransaction().commit();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Select with JOIN

Select with JOIN - Customized Object

```
public List<EmployeeDepartment> listEmployeeWithDepartmentName () {

  String sql = "SELECT new model.EmployeeDepartment(e.employeeId, e.firstName, d.name) "   +

           " FROM Employee e JOIN e.departments d" ;

   return em.createQuery(sql, EmployeeDepartment.class).getResultList();

}
```

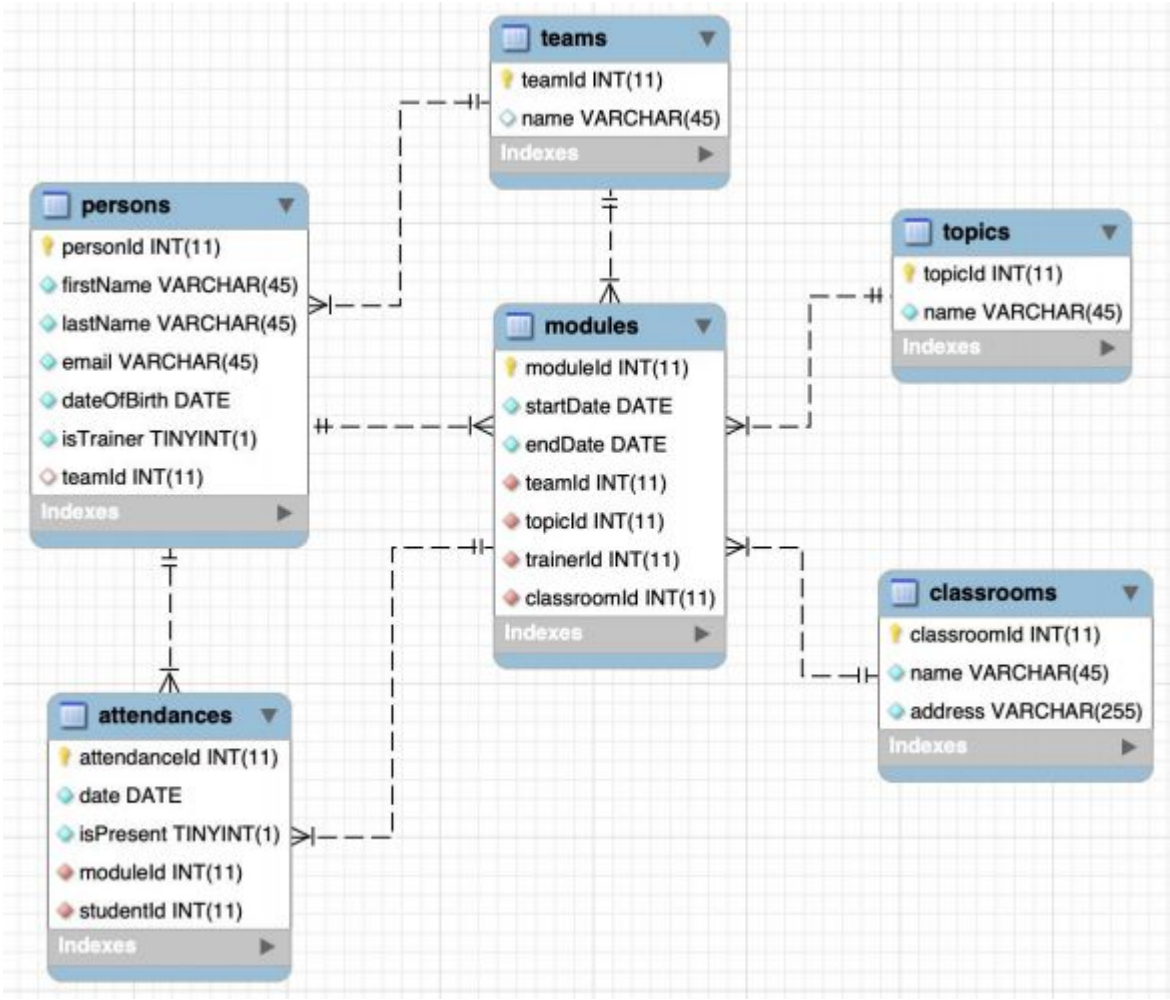
Select with JOIN - Interacting with join attribute

```
public List<Employee> findEmployeeByDept (String department) {

   String sql  = "FROM Employee e "  +

                 " WHERE e.department.departmentName = :deptName" ;



   return  em.createQuery( sql)

            .setParameter( "deptName", department)

            .getResultList();

}
```

Read more

- https://docs.jboss.org/hibernate/orm/5.1/userguide/html_single/chapters/query/hql/HQL.html
- https://thorben-janssen.com/query-complex-jpa-hibernate/
- https://www.baeldung.com/jpa-join-types
- https://thorben-janssen.com/hibernate-best-practices/

Exercises - Hibernate 1



Exercises - Hibernate 1

- The SQL script to import the database to your local MySQL server can be found here:
- https://gitlab.com/sda-international/program/common/databases/-/blob/master/sdaDatabaseStructure.sql

1. List all students.
2. List all students for team Python1Tallin.
3. List all groups that had classes in location BucharestCowork.
4. List all groups that had classes in location Tallin Cozy Space in March 2020.
5. List all students that already finished the SQL module.
6. List all students with 100% attendance rate.
7. List all trainers that teach Java Fundamentals.
8. List all trainers that teach at BucharestCowork location.
9. List all trainers that taught students with 100% attendance rate.
10. List the trainer that taught the highest number of modules.