Exercises: MiniORM

This document defines the **exercise assignments** for the "Databases Advanced – Hibernate" course @ Software University. Please submit your solutions in Judge.

Following guides of this document you will be able to create your custom ORM with basic functionality (insert, update and retrieve single object or set of objects). Our ORM will have options to work with already created tables in a database or create new tables if such are not present yet.

1. Create Annotations

Start with creating new Project called MiniORM. Add package **persistence** to keep our annotations that would be required for our ORM. Create 3 annotations:

- **Id** – has no properties, it will be applied on **fields only**
- **Column** – has property **name**, it will be applied on **fields only**
- **Entity** – has property **name**, it will be applied on **classes only**

Follow the good practices in naming custom annotations.

2. Create Entities

In our project create package called **entities** where we could keep every one of our entities. Now let's **create class User** with fields and properties (**id, username, password, age, registrationDate**). Add appropriate **annotations** to our newly created class and its fields. Create **constructor** that **sets all fields except id**. Order of the parameters in the constructor must be **the same as the sequence of columns in the table in the database**

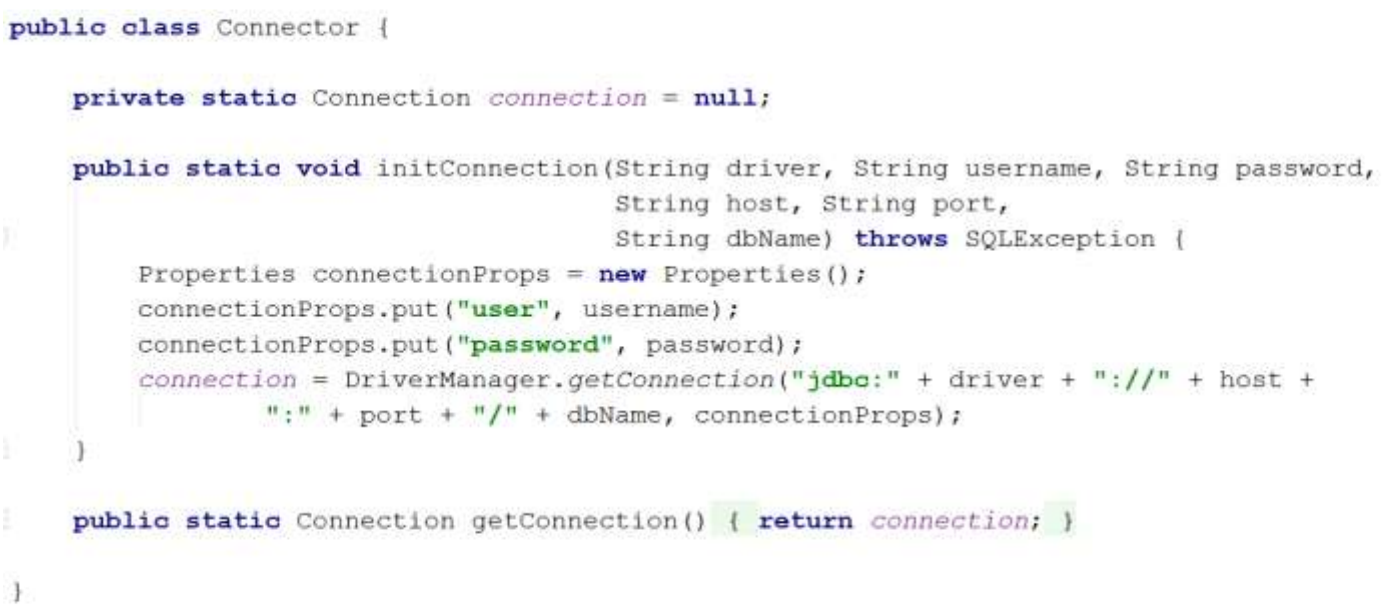
```
@Entity(name = "users")
public class User {

    @Id
    private int id;

    @Column(name = "password")
    private String password;

    @Column(name = "age")
    private int age;

    @Column(name = "registration_date")
    private Date registrationDate;

    public User(String password, int age, Date registrationDate) {
        this.setPassword(password);
        this.setAge(age);
        this.setRegistrationDate(registrationDate);
    }
```

3. Create Database Connection

Now we can **create class that generates connection with our database**. In order to achieve this, we would require the following parameters:

- **Driver** – for MySQL
- **Username** – database username
- **Password** – database password
- **Host** – localhost
- **Port** – 3306 is the default one
- **Database Name** – the current database for the project. We need to create one manually.

The class should generate a **connection** that we would use to connect to the database.



```java
public class Connector {

    private static Connection connection = null;

    public static void initConnection(String driver, String username, String password,
                                      String host, String port,
                                      String dbName) throws SQLException {
        Properties connectionProps = new Properties();
        connectionProps.put("user", username);
        connectionProps.put("password", password);
        connection = DriverManager.getConnection("jdbc:" + driver + "://" + host +
                ":" + port + "/" + dbName, connectionProps);
    }

    public static Connection getConnection() { return connection; }

}
```

4. Create Database Context

It's time to create interface that would define the operations we can perform with the database. Name your interface **`DbContext`** and defined the following methods in it.

- **<E> boolean persist(E entity)** – it will insert or update entity depending if it is attached to the context
- **<E> Iterable<E> find(Class<E> table)** – returns collection of all entity objects of type E
- **<E> Iterable<E> find(Class<E> table, String where)** – returns collection of all entity objects of type T matching the criteria given in "where"
- **<E> E findFirst(Class<E> table)** – returns the first entity object of type E
- **<E> E findFirst(Class<E> table, String where)** – returns the first entity object of type E matching the criteria given in "where"

```java
public interface DbContext {

    <E> boolean persist(E entity) throws IllegalAccessException, SQLException;

    <E> Iterable<E> find(Class<E> table)
            throws ClassNotFoundException, SQLException, InstantiationException, IllegalAccessException;

    <E> Iterable<E> find(Class<E> table, String where)
            throws SQLException, ClassNotFoundException, IllegalAccessException, InstantiationException;

    <E> E findFirst(Class<E> table)
            throws IllegalAccessException, SQLException, InstantiationException;

    <E> E findFirst(Class<E> table, String where)
            throws SQLException, IllegalAccessException, InstantiationException;
}
```

5. Create Entity Manager

Enough with the preparation it's time to write the core of our Mini ORM. That would be **`EntityManager`** class that would be responsible for inserting, updating and retrieving entity objects. That class **would implement methods of the `DbContext` interface**. That class would require a **`Connection`** object that would be initialized with a given connection string.

```java
public class EntityManager  implements DbContext{

    private Connection connection;
    private Set<Object> persistedEntities;

    public EntityManager(Connection connection) {
        this.connection = connection;
        this.persistedEntities = new HashSet<>();
    }

    @Override
    public <E> boolean persist(E entity)
            throws IllegalAccessException, SQLException {
        // TODO: Make the implementation
        return false;
    }

    @Override
    public <E> Iterable<E> find(Class<E> table)
            throws ClassNotFoundException, SQLException, InstantiationException, IllegalAccessException {
        // TODO: Make the implementation
        return null;
    }

    @Override
    public <E> Iterable<E> find(Class<E> table, String where)
            throws SQLException, ClassNotFoundException, IllegalAccessException, InstantiationException {
        // TODO: Make the implementation
        return null;
    }
    @Override
    public <E> E findFirst(Class<E> table)
            throws IllegalAccessException, SQLException, InstantiationException {
        // TODO: Make the implementation
        return null;
    }

    @Override
    public <E> E findFirst(Class<E> table, String where)
            throws SQLException, IllegalAccessException, InstantiationException {
        // TODO: Make the implementation
        return null;
    }
}
```

6. Helper Methods

It’s time for some reflection. We would create 3 methods that would help us

- **private <E> String getTableName(Class<E> entity)** – Returns the value of the **name** type of Entity annotation or if it’s not set returns the name of the entity.
- **private String getFieldName(Field field)** – Returns the value of the **name** type of Column annotation or if it’s not set returns the name of the field.
- **private Field getId(Class c)** – Get the field with Id annotations of the given entity. If there is no field with Id annotations throw exception.

```
private <E> String getTableName(Class<E> entity) {
    // TODO: Make the implementation
    return null;
}


private String getFieldName(Field field) {
    // TODO: Make the implementation
    return null;
}

private Field getId(Class c) {
    // TODO: Make the implementation
    return null;
}
```

7. Create Table

The first thing in our mind should be to create a table if it doesn't exist. We need a method for this as well:

- **private <E> boolean doCreate(E entity, Field primary)**

```
private <E> boolean doCreate(E entity, Field primary) throws IllegalAccessException, SQLException {
    String query = "CREATE TABLE IF NOT EXISTS ";
    // TODO: Make the implementation

    return true;
}
```

We should take care for the corresponding data types in MySQL. Therefore, another help method will appear and it will help us to convert our java types to database ones.

- **private String getDbType(Field field, Field primary)**

```
private String getDbType(Field field, Field primary) {
    field.setAccessible(true);
    if(field.getName().equals(primary.getName())){
        return "BIGINT PRIMARY KEY AUTO_INCREMENT";
    }

    switch (field.getType().getSimpleName()){
        case "int":
            return "INT";
        case "String":
            return "VARCHAR(50)";
        //// TODO: Custom Implementation
    }

    return  null;
}
```

8. Persist Object in the Database

The logic behind the persist method is pretty simple. First the method should **create the table**. Then if the given **object** to be persisted **is not contained** in the database -> **add it**, otherwise **update its properties with the new values**. The method returns whether the object was **successfully persisted** in the database or not.

```java
@Override
public <E> boolean persist(E entity) throws IllegalAccessException, SQLException {

    Field primary = this.getId(entity.getClass());
    primary.setAccessible(true);
    Object value = primary.get(entity);

    this.doCreate(entity,primary);

    if (value == null || (Long)value <= 0) {
        return this.doInsert(entity, primary);
    }

    return this.doUpdate(entity, primary);
}
```

So far we need to implement 2 more methods:

- **`private <E> boolean doInsert(E entity, Field primary)`**
- **`private <E> boolean doUpdate(E entity, Field primary)`**

Both methods would prepare query statements and execute them.

The difference between them is when you insert new entity you should **set its Id**. The Id is generated from the table in the database. Both methods return whether the entity was successfully persisted.

Here are some tips for the Insert method:

```java
private <E> boolean doInsert(E entity, Field primary) throws IllegalAccessException, SQLException {
    String query = "INSERT INTO " + this.getTableName(entity.getClass())+" ";

    return connection.prepareStatement(query).execute();
}
```

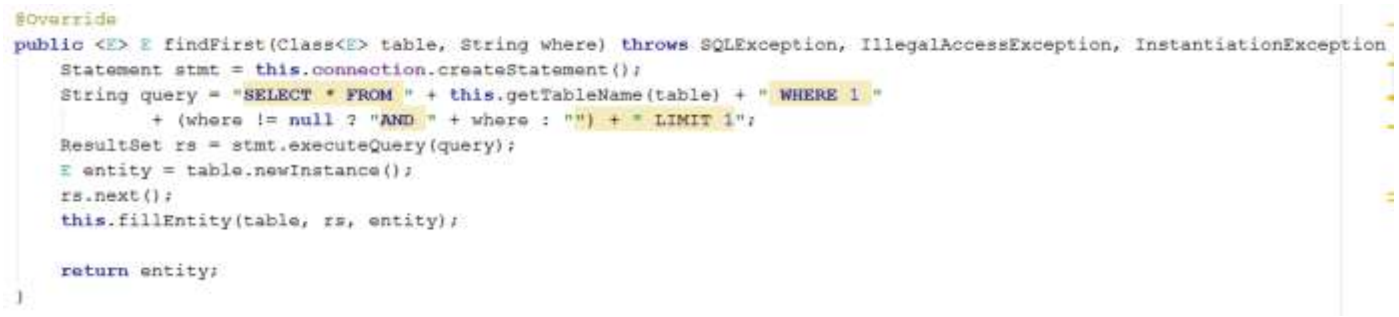
And some tips for the update method

```java
private <E> boolean doUpdate(E entity, Field primary) throws IllegalAccessException, SQLException {

    String query = "UPDATE " + this.getTableName(entity.getClass()) + " SET ";
    String where = "WHERE 1=1 ";

    return connection.prepareStatement(query + where).execute();
}
```

9. Fetching Results

Finally, when we persisted our entities (objects) in the database let 's implement functionality to **get them out of the database and persist them in the operating memory**. We would implement just several methods to get objects from the database. That would be **all Find methods from the DbContext** (check Problem 4. for more information about each one of them).

Here is tip of how to implement **`public <E> E findFirst(Class<E> table, String where)`** the other ones are similar and they would be on you ☺.

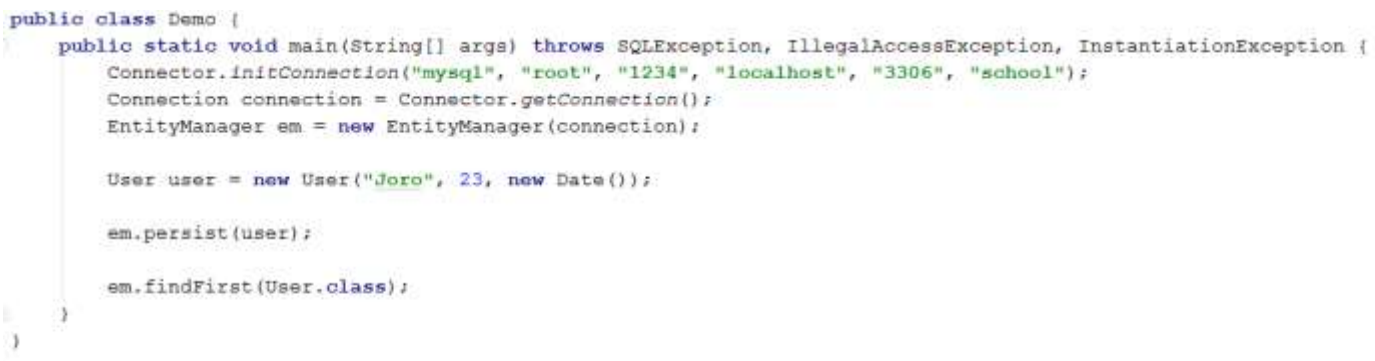


```java
@Override
public <E> E findFirst(Class<E> table, String where) throws SQLException, IllegalAccessException, InstantiationException
    Statement stmt = this.connection.createStatement();
    String query = "SELECT * FROM " + this.getTableName(table) + " WHERE 1 "
            + (where != null ? "AND " + where : "") + " LIMIT 1";
    ResultSet rs = stmt.executeQuery(query);
    E entity = table.newInstance();
    rs.next();
    this.fillEntity(table, rs, entity);

    return entity;
}
```

Here you can see that we used some new method **fillEntity**. That method receives **`ResultSet`** object, **retrieve information from the current row** of the reader fills the data.

10. Test Framework

If you came to this point you are done with building our MiniORM. Now let's test it to make sure it works as expected. Create several users and persist them in the database. Then update some of the properties of the users (e.g., change password or increase age or some other change). Remember you need to use the persist method to commit changes of the object to the database. Make sure the data is always updated in the database. Here is some example of usage:



```java
public class Demo {
    public static void main(String[] args) throws SQLException, IllegalAccessException, InstantiationException {
        Connector.initConnection("mysql", "root", "1234", "localhost", "3306", "school");
        Connection connection = Connector.getConnection();
        EntityManager em = new EntityManager(connection);

        User user = new User("Joro", 23, new Date());

        em.persist(user);

        em.findFirst(User.class);
    }
}
```

11. Fetch Users

Insert several users in the database and **print the usernames and passwords** of those who are **registered after 2010** year and are **at least 18 years old**.

12. Add New Entity

Add new class **Book(id, title, author, publishedOn, language, isHardCovered)**. Add several books to the database. Then check all the books whether their **title is over 30 symbols long** and **have hard cover**. If that is true, then **trim the exceeding length** and make it exactly 30 symbols long. Make sure they are properly persisted in the database after those changes. TODO

Example

Input	Output

13. Update Entity

Add new column Rating (possible values in range from 0 to 10) to the book entity. Find latest 3 books with highest rating and print their titles, authors and rating. Sort them by rating descending then by title ascending.

14. Update Records

Write a program that receives as an input year and then changes all book titles to be uppercase if they are released after that given year and has hard cover. Print their count on the console followed by their titles ordered lexicographically.

Example

Input	Output

`2000`	`Books released after 2000 year: 4`
	`HARRY POTTER 4`
	`HARRY POTTER 5`
	`HARRY POTTER AND HALF-BLOOD PRINCE`
	`HARRY POTTER 7`

15. Delete Records

Extend your ORM to be possible to **delete objects by given Id**. Then find **all books that has rating below 2** and **delete them** from the database. Print on the console the **number of books that has been deleted**.

Example

Output
`3 Books has been deleted from the database.`

16. Delete Inactive Users

Add two columns to the User entity:

- **LastLoginTime** – that will keep the last time our user has been logged in the system
- **IsActive** – that will keep whether the last login time has been for more than a year.

Populate the database with sample data of users.

Write a program that **get as an input username** then check in the database whether the user with **that username is active or not**. If it is active **print his last login time relatively** to the moment of executing the program (check table below for examples). If it is not active **prompt the user to confirm if that user should be deleted**. If the user confirms then **delete that user from the database** otherwise **do nothing**.

Last time user was logged in	Relative time
< 1 second	less than a second
< 1 minute	less than a minute
< 1 hour (x minutes)	{x} minutes ago
< 1 day (x hours)	{x} hours ago
< 1 week (x days)	{x} days ago
< 1 month (x weeks)	{x} weeks ago
< 1 year (x months)	{x} months ago
> 1 year	more than a year

Example

Input	Output
`dragonZ999`	`User dragonZ999 was last online 3 hours ago.`
`bobara2016`	`User bobara2016 was last online less than a minute.`

`pesh123`
`yes`
	`User pesh123 was last online more than a year.`
	`Would you like to delete that user? (yes/no)`
	`User pesh123 was successfully deleted from the database.`

`bigCatTom`
`no`
	`User bigCatTom was last online more than a year.`
	`Would you like to delete that user? (yes/no)`
	`User bigCatTom was not deleted from the database.`