FUNDAMENTAL PROGRAMMING TECHNIQUES

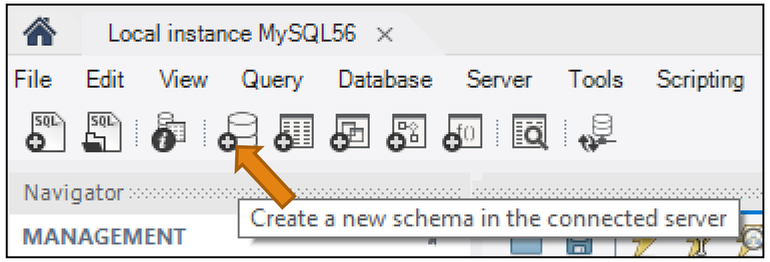
ASSIGNMENT 3 – SUPPORT PRESENTATION

Outline

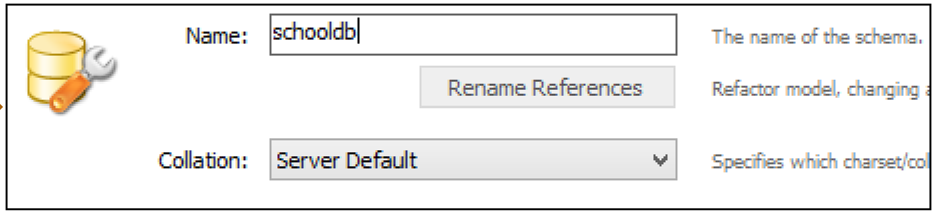
- JDBC Basics
- Layered Architectures
- Reflection Techniques Examples
- Source Code

JDBC Basics - Prerequisites

- **Install MySQL and MySQL Workbench** (see PT2020-2021_Lab_Resources.pdf)
- **Create a database in MySQL Workbench – set “schooldb” as the name of the schema**



Create a new schema in the connected server



Name: schooldb

Rename References

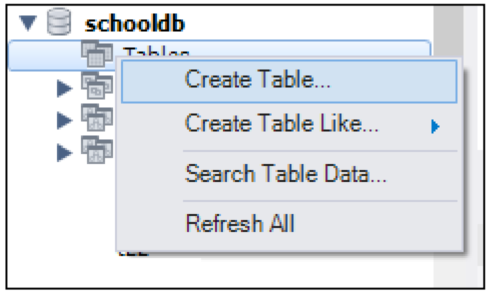
Collation: Server Default

The name of the schema.

Refactor model, changing a

Specifies which charset/col

- **Create a table called “student”**



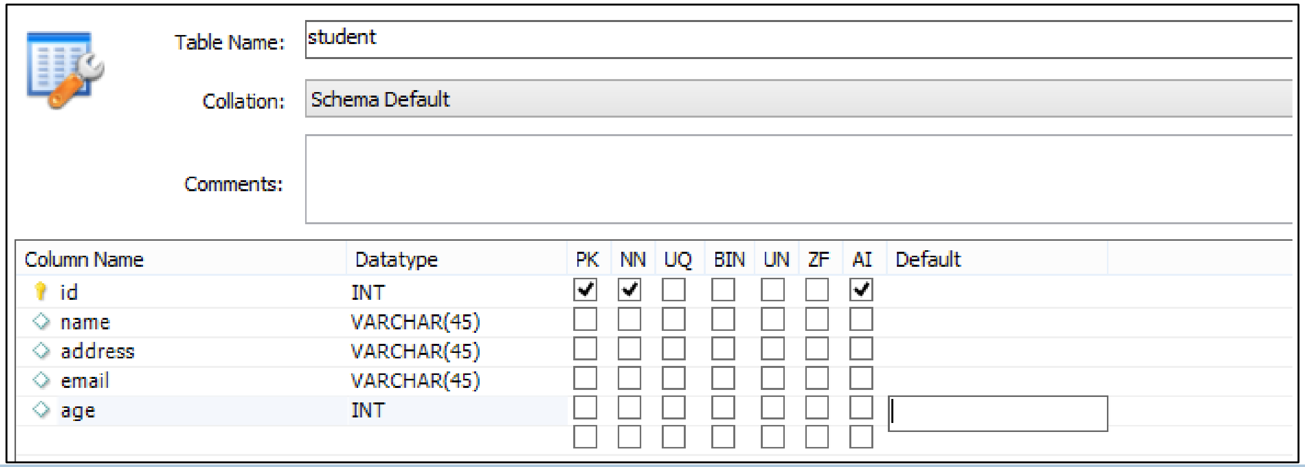


Table Name: student

Collation: Schema Default

Comments:

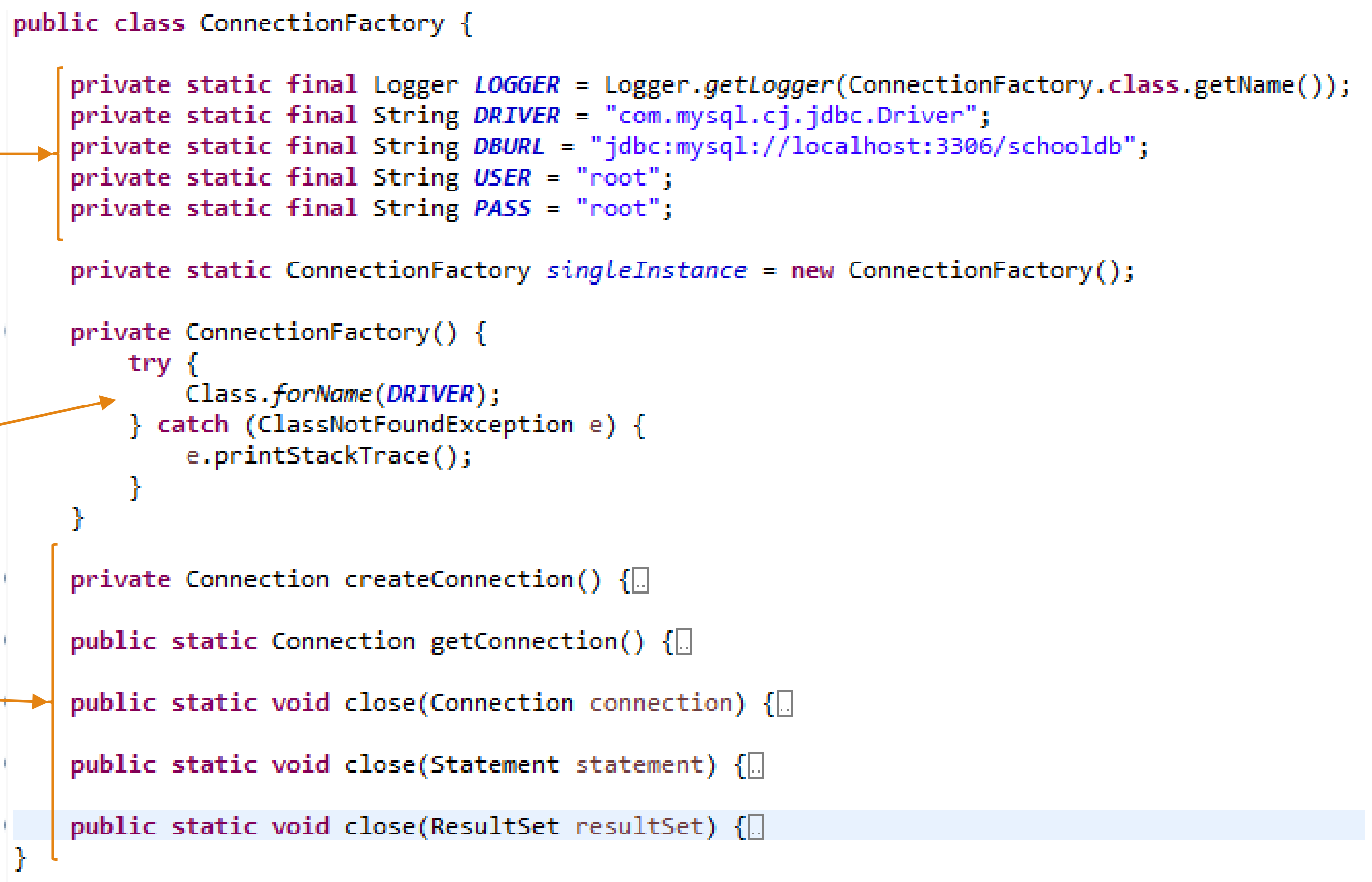
Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI	Default
id	INT	✓	✓					✓	
name	VARCHAR(45)								
address	VARCHAR(45)								
email	VARCHAR(45)								
age	INT								

JDBC Basics – Processing SQL Statements

- Steps
  - Establish a connection with the data source
  - Create a statement
  - Execute the query
  - Process the ResultSet object
  - Close the connection

JDBC Basics – Establishing a Connection

- This class contains the name of the driver (initialized through reflection), the database location (DBURL), and the user and the password for accessing the MySQL Server
- The connection to the DB will be placed in a *Singleton** object
- The class contains methods for creating a connection, getting an active connection and closing a connection, a Statement or a ResultSet

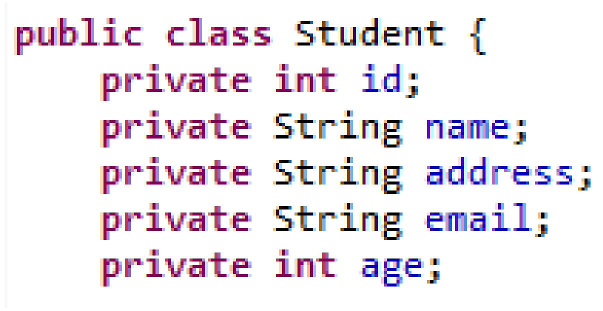


```java
public class ConnectionFactory {

    private static final Logger LOGGER = Logger.getLogger(ConnectionFactory.class.getName());
    private static final String DRIVER = "com.mysql.cj.jdbc.Driver";
    private static final String DBURL = "jdbc:mysql://localhost:3306/schooldb";
    private static final String USER = "root";
    private static final String PASS = "root";

    private static ConnectionFactory singleInstance = new ConnectionFactory();

    private ConnectionFactory() {
        try {
            Class.forName(DRIVER);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }

    private Connection createConnection() {□

    public static Connection getConnection() {□

    public static void close(Connection connection) {□

    public static void close(Statement statement) {□

    public static void close(ResultSet resultSet) {□
}
```

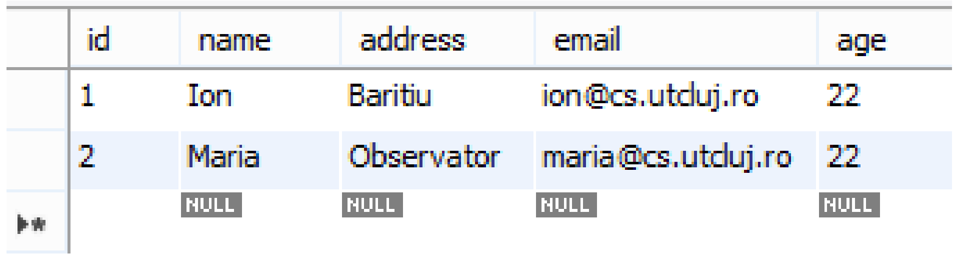
*Singleton Design Pattern: https://en.wikipedia.org/wiki/Singleton_pattern

JDBC Basics – Table Mapping

- In order to extract elements from the DB table, a special class (named entity) must be created.
- This class MUST have the fields exactly the same type as the columns from the corresponding table.
- The class must have also constructors, getters and setters.



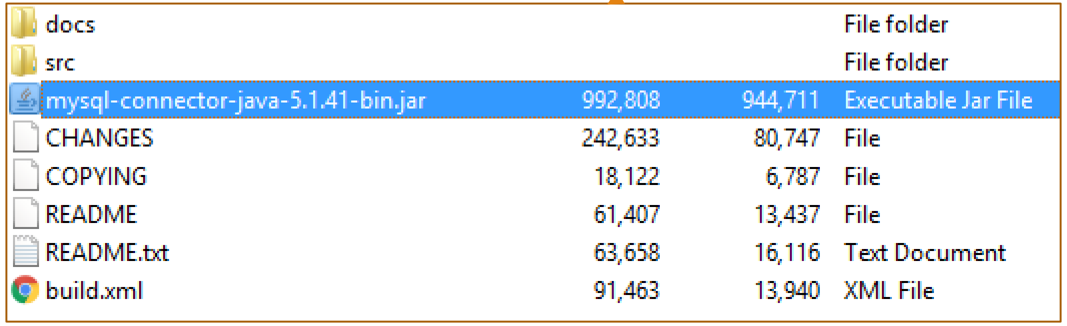
```
public class Student {
    private int id;
    private String name;
    private String address;
    private String email;
    private int age;
```



	id	name	address	email	age
	1	Ion	Baritiu	ion@cs.utcluj.ro	22
	2	Maria	Observator	maria@cs.utcluj.ro	22
▶*		NULL	NULL	NULL	NULL

JDBC Basics - Dependencies

- In order for the Java application to interact with the DB, a special **.jar** library must be added to the application
- It can be added either as an **external jar file dependency** or as a **maven dependency**, in case of a Maven project



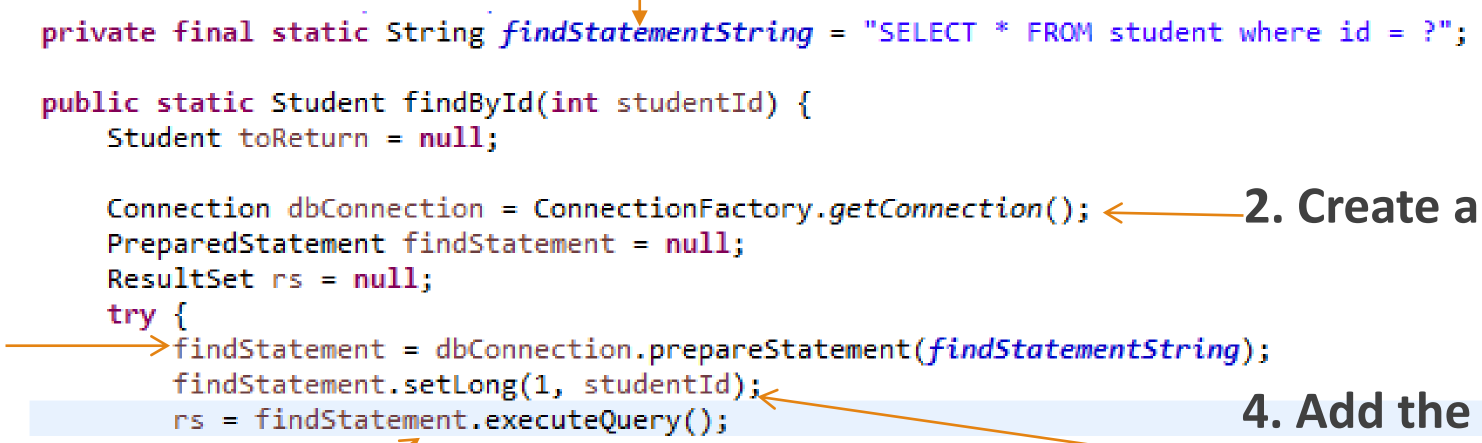
docs			File folder
src			File folder
mysql-connector-java-5.1.41-bin.jar	992,808	944,711	Executable Jar File
CHANGES	242,633	80,747	File
COPYING	18,122	6,787	File
README	61,407	13,437	File
README.txt	63,658	16,116	Text Document
build.xml	91,463	13,940	XML File

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.23</version>
</dependency>
```

- The Java application uses this external library to communicate with the MySQL server
- It sends queries to the server using **Statements** and it receives the results of the queries as **ResultSet**

JDBC Basics – Creating and Executing Statement

1. Define a string with the query



```
private final static String findStatementString = "SELECT * FROM student where id = ?";

public static Student findById(int studentId) {
    Student toReturn = null;

    Connection dbConnection = ConnectionFactory.getConnection();
    PreparedStatement findStatement = null;
    ResultSet rs = null;
    try {
        findStatement = dbConnection.prepareStatement(findStatementString);
        findStatement.setLong(1, studentId);
        rs = findStatement.executeQuery();
```

2. Create a connection to the DB

3. Initialize the query

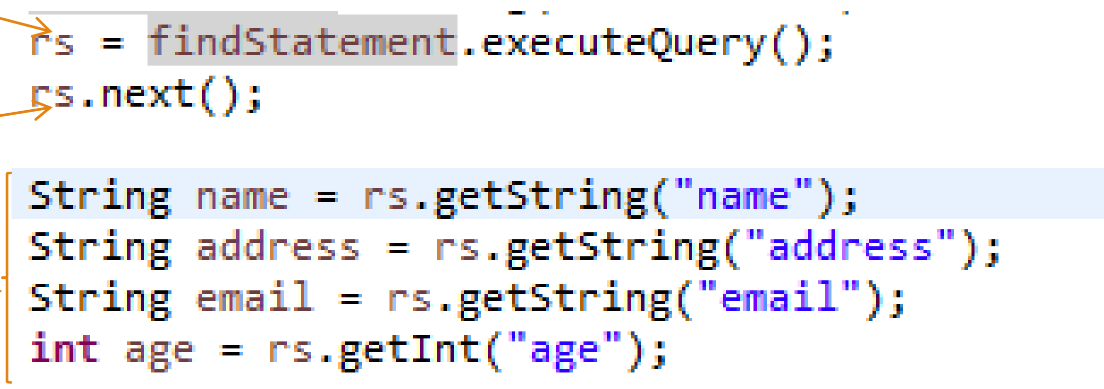
4. Add the parameters to the query (the ? will be replaced with data from the application)

5. Execute the query

JDBC Basics - Process the ResultSet object

The results of the query execution are stored in a result set:

- Each element of the result set corresponds to a row from the table
- The result set can be iterated
- The properties/values from the columns can be extracted if the column name is known

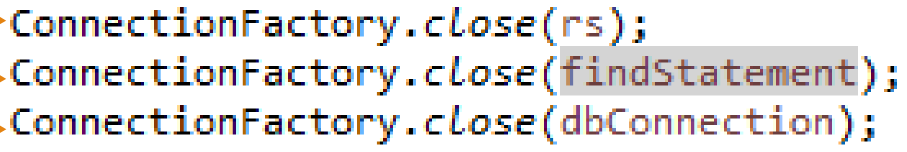


```
rs = findStatement.executeQuery();
rs.next();

String name = rs.getString("name");
String address = rs.getString("address");
String email = rs.getString("email");
int age = rs.getInt("age");
```

JDBC Basics – Closing the connection

After each operation the connection must be closed:

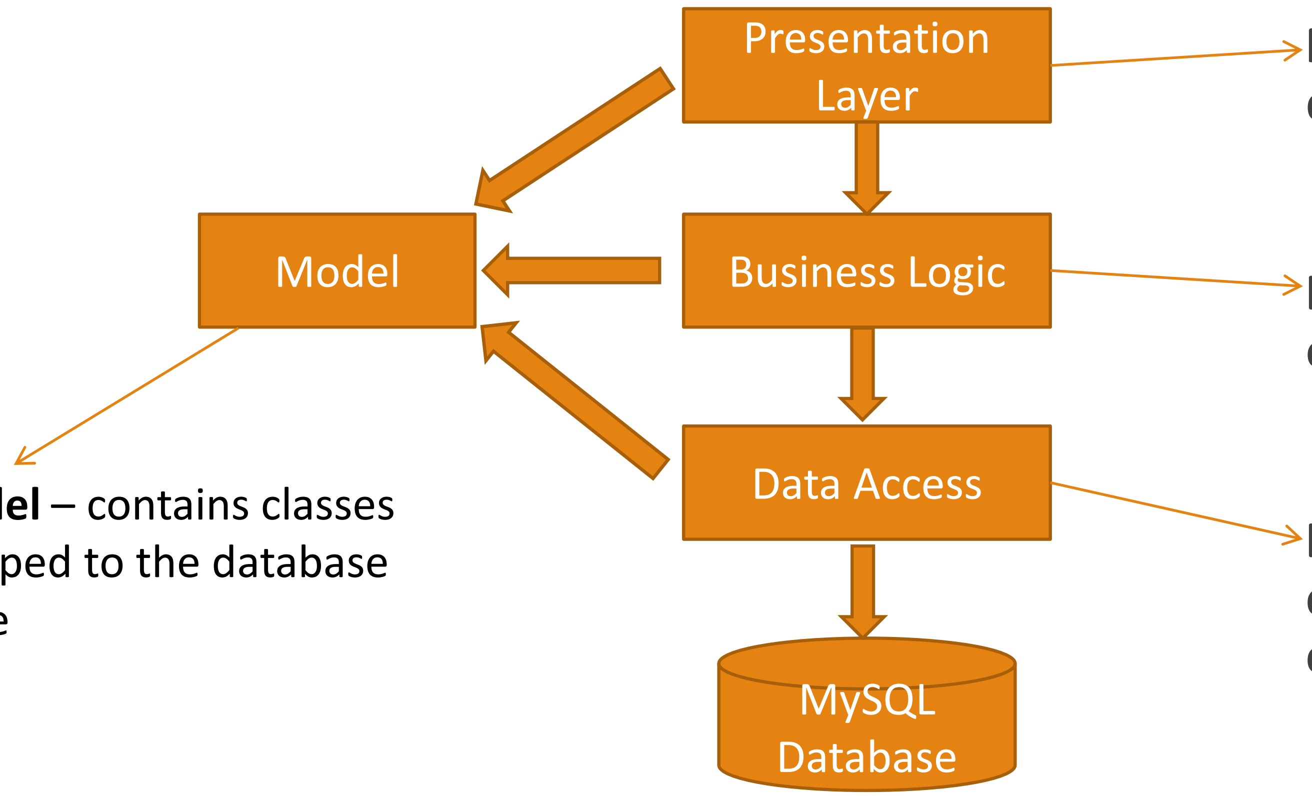
- The result set
- The statement
- The connection



```
ConnectionFactory.close(rs);
ConnectionFactory.close(findStatement);
ConnectionFactory.close(dbConnection);
```

Layered Architectures

- Split the application in different layers
- Each layer has a special purpose and calls functions of the layers below it



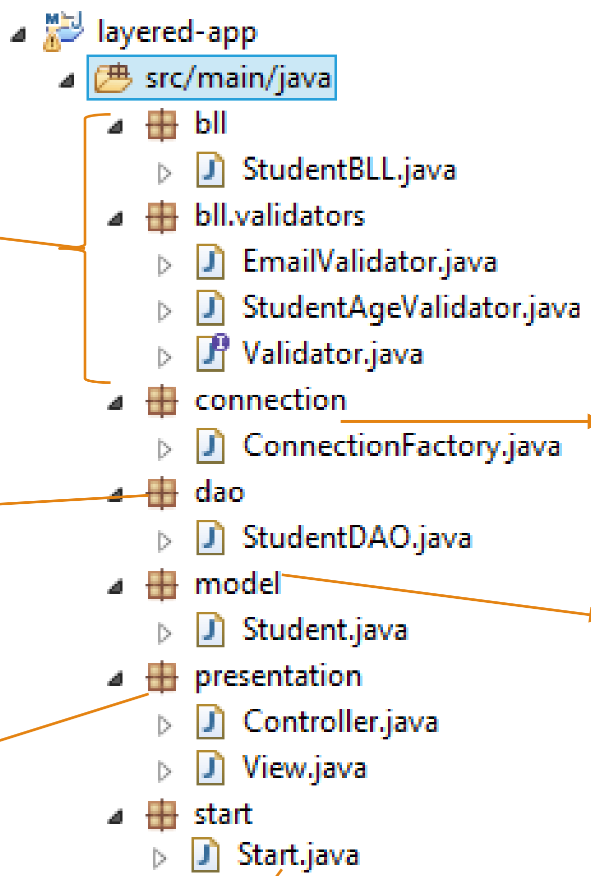
Presentation Layer - contains the classes defining the user interface

Business Layer – contains the classes that encapsulate the application logic

Data Access Layer – contains the classes containing the queries and the database connection

Model – contains classes mapped to the database table

Layered Architectures – 3 Tier Architecture



```
layered-app
  src/main/java
    bll
      StudentBLL.java
    bll.validators
      EmailValidator.java
      StudentAgeValidator.java
      Validator.java
    connection
      ConnectionFactory.java
    dao
      StudentDAO.java
    model
      Student.java
    presentation
      Controller.java
      View.java
    start
      Start.java
```

Business Logic Classes:

- StudentBLL
- Validators - validate the fields

Database connection

Data access classes (also containing queries)

Model Classes

Presentation Layer (UI)Classes

Main Class

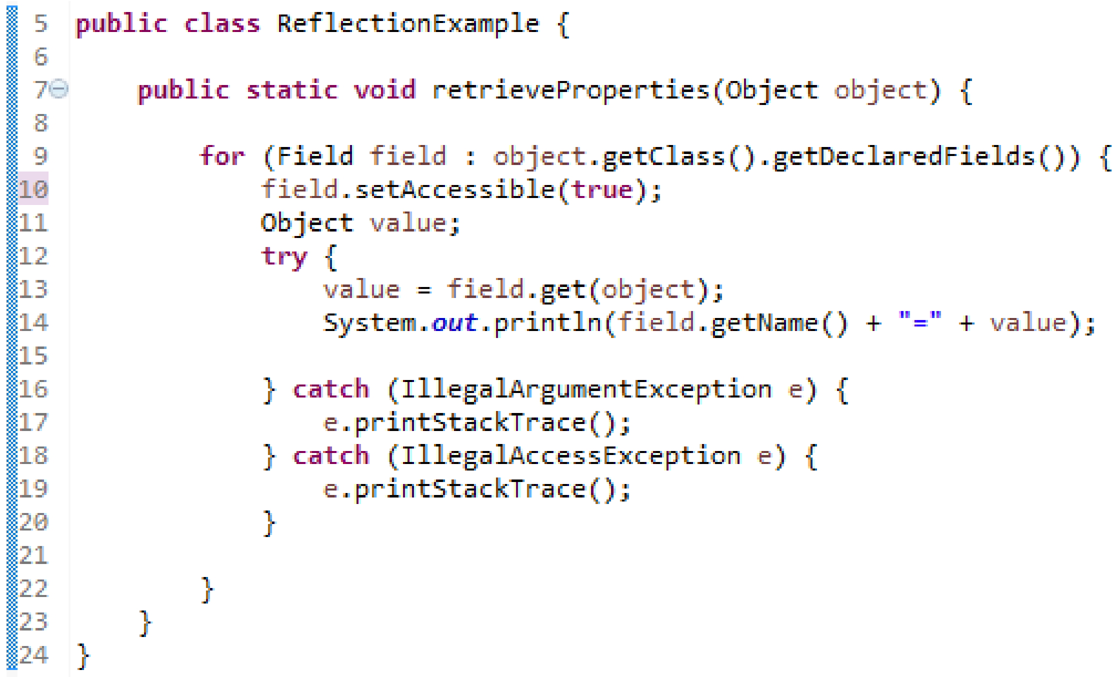
Reflection Techniques Examples

Problem: Given an instance of any object, extract the fields and the corresponding values

(1) Line 9 – obtain the class of the object and request the declared fields of that class

(2) For each declared field :

- Line 10 - Set the field accessible (most of them are private => need to change permissions)
- Line 13 - Return the value of the current field contained in the specified object
- Line 14 - Print the field's name and its value

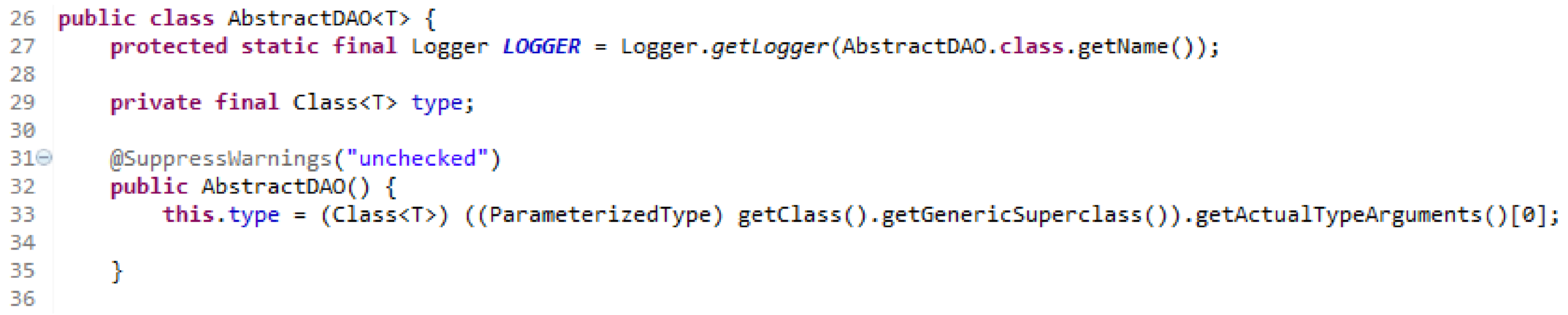


```
 5 public class ReflectionExample {
 6
 7     public static void retrieveProperties(Object object) {
 8
 9         for (Field field : object.getClass().getDeclaredFields()) {
10             field.setAccessible(true);
11             Object value;
12             try {
13                 value = field.get(object);
14                 System.out.println(field.getName() + "=" + value);
15
16             } catch (IllegalArgumentException e) {
17                 e.printStackTrace();
18             } catch (IllegalAccessException e) {
19                 e.printStackTrace();
20             }
21
22         }
23     }
24 }
```

Reflection Techniques Examples

Problem 1: Propose a smart solution for constructing the DAO classes using Generics and Reflection

- Line 26 - Define a class AbstractDAO that defines the common operations for accessing a table: Insert, Update, Delete, FindById, FindAll. Define the operations on the specified generic type ***<T> (! T can be any Java Model Class that is mapped to the Database, and has the same name as the table and the same instance variables and data types as the table fields)***
- Line 29; Line 33 - For each AbstractDAO object obtain the class of the generic type T

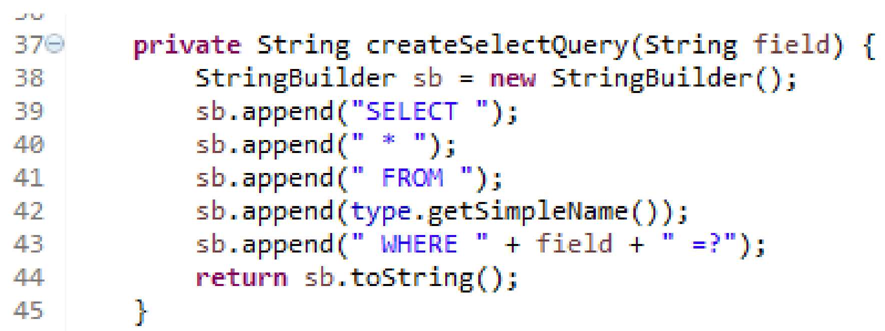


```java
26  public class AbstractDAO<T> {
27      protected static final Logger LOGGER = Logger.getLogger(AbstractDAO.class.getName());
28  
29      private final Class<T> type;
30  
31      @SuppressWarnings("unchecked")
32      public AbstractDAO() {
33          this.type = (Class<T>) ((ParameterizedType) getClass().getGenericSuperclass()).getActualTypeArguments()[0];
34  
35      }
36  
```

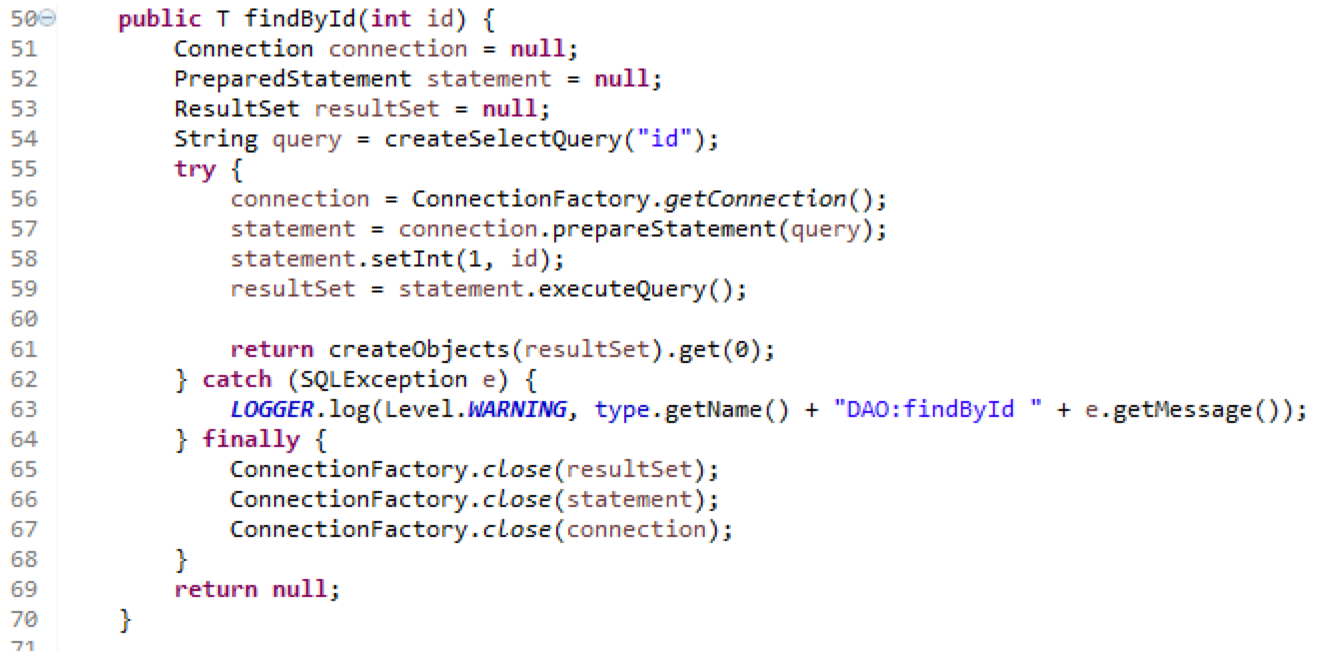
Reflection Techniques Examples

Problem 2: Propose a smart solution for constructing the DAO classes using Generics and Reflection – *findById method*

- Line 54 - A generic query is build using the id field name *(! The value "id" can be replaced in a generic way by defining a custom annotation for PK and obtaining the name of the annotated field)*
- Lines 37-44 - Build the generic select query using the name of the class previously obtained *(! The approach considers that the name of the Java model class is the same as the name of the table in db. For a more generic approach, define a custom annotation for the table name, where the developer can specify a name different from the name of the class)*
- Line 56 - the connection is obtained from the ConnectionFactory previously defined
- Line 58 - The id is considered to be of type int. *(! For a more generic approach, the type of the PK can be specified in a generic way, similarly with the generic Type T)*
- Line 61- Using the ResultSet obtained after executing the statement, we will obtain the list of objects. Only the first element is of interest, since the db should contain only one entry with the specified id



```
37     private String createSelectQuery(String field) {
38         StringBuilder sb = new StringBuilder();
39         sb.append("SELECT ");
40         sb.append(" * ");
41         sb.append(" FROM ");
42         sb.append(type.getSimpleName());
43         sb.append(" WHERE " + field + " =?");
44         return sb.toString();
45     }
```

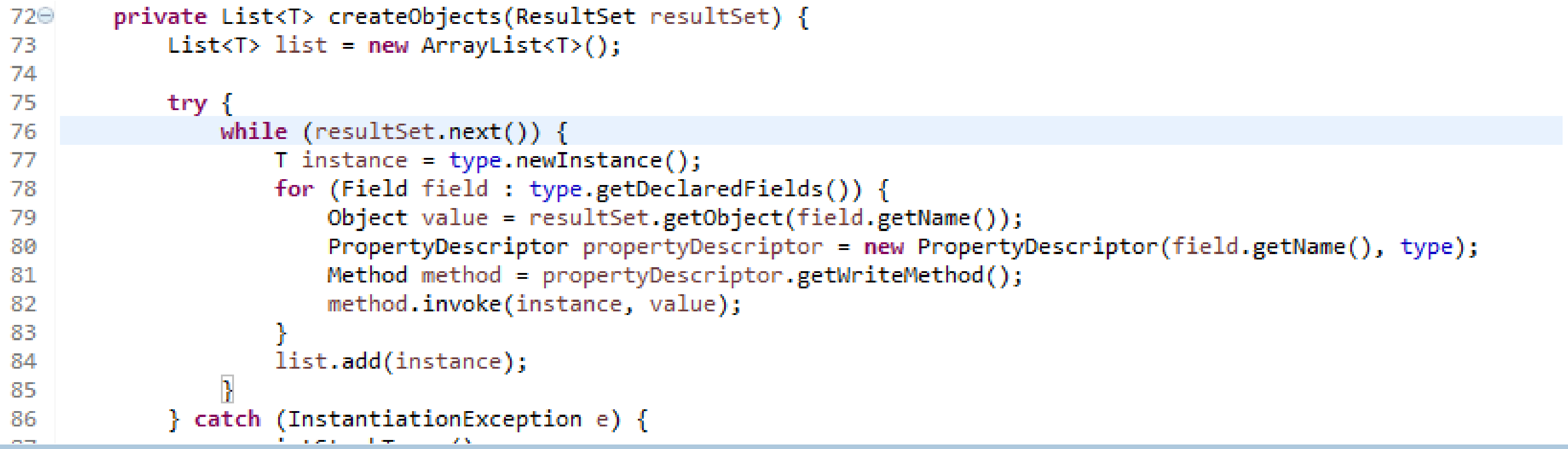


```
50     public T findById(int id) {
51         Connection connection = null;
52         PreparedStatement statement = null;
53         ResultSet resultSet = null;
54         String query = createSelectQuery("id");
55         try {
56             connection = ConnectionFactory.getConnection();
57             statement = connection.prepareStatement(query);
58             statement.setInt(1, id);
59             resultSet = statement.executeQuery();
60 
61             return createObjects(resultSet).get(0);
62         } catch (SQLException e) {
63             LOGGER.log(Level.WARNING, type.getName() + "DAO:findById " + e.getMessage());
64         } finally {
65             ConnectionFactory.close(resultSet);
66             ConnectionFactory.close(statement);
67             ConnectionFactory.close(connection);
68         }
69         return null;
70     }
```

Reflection Techniques Examples

Problem 3: Propose a smart solution for constructing the DAO classes using Generics and Reflection – *findById method (cntd.)*

- Line 72 – Given a result set, obtain the list of model objects of type T
- Line 76 - For each *result* from the ResultSet
  - Line 77 – Create a new instance of type T
  - Line 78 - For each *field* of the class T
    - Line 79 - Retrieve from the current *result* the value of the current *field*
    - Line 80-81 – Obtain the method for setting a value to the *field*
    - Line 82 - using the obtained method set the value to the field

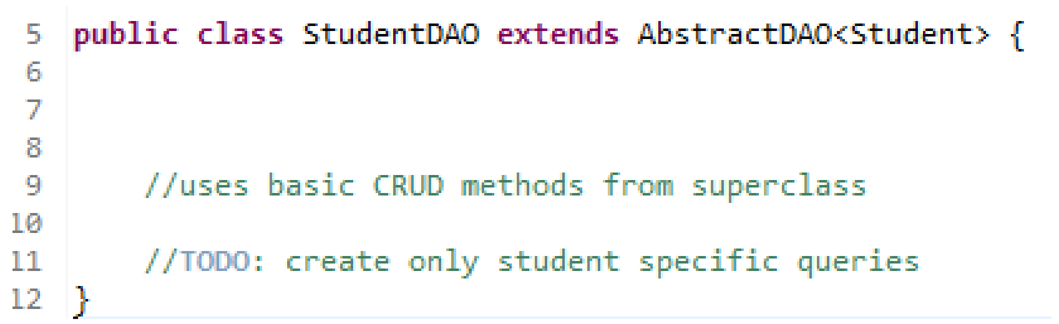


```
72      private List<T> createObjects(ResultSet resultSet) {
73          List<T> list = new ArrayList<T>();
74  
75          try {
76              while (resultSet.next()) {
77                  T instance = type.newInstance();
78                  for (Field field : type.getDeclaredFields()) {
79                      Object value = resultSet.getObject(field.getName());
80                      PropertyDescriptor propertyDescriptor = new PropertyDescriptor(field.getName(), type);
81                      Method method = propertyDescriptor.getWriteMethod();
82                      method.invoke(instance, value);
83                  }
84                  list.add(instance);
85              }
86          } catch (InstantiationException e) {
```

Reflection Techniques Examples

Problem 4: Propose a smart solution for constructing the DAO classes using Generics and Reflection - *Usage*

- Line 5 – Define a class (StudentDAO) that extends the AbstractDAO and specify the used model class: Student
- The generic methods are directly accessed through inheritance
- Other specific methods can be implemented at this level



```
 5  public class StudentDAO extends AbstractDAO<Student> {
 6
 7
 8
 9      //uses basic CRUD methods from superclass
10
11      //TODO: create only student specific queries
12  }
```

Source Code

Download the source code from:

- Simple layered project: https://gitlab.com/utcn_dsrl/pt-layered-architecture
- Reflection example: https://gitlab.com/utcn_dsrl/pt-reflection-example