A picture containing food, drawing

Description automatically generated

project report

Object Oriented Programming 1

Al Sayyed Hadi Ibrahim

# Abstract

In this project, the user can perform CRUD operations on his Contacts list, and search through it using any and as many criteria as needed. The contacts are stored in a MySQL database, and Hibernate Framework was used to link the program to it. In addition to that, the user can import contacts from one of the supported file types that include: XML, CSV, and JSON.

# Data preprocessing and preparations

The contacts inside our projects have a total of 25 fields, that we decided to divide into many classes, and fill each attribute to the corresponding class. For example: *address type, street, city, region* and *country* were moved to a class *Address,* and so on.

Next, we needed some data to make the operations, and <https://www.mockaroo.com/> had almost everything we needed to create some good initial mock data. So, we entered our attributes and desired outcomes and Mockaroo generated and exported our data to a CSV format file that we can download. 1000 records were created this way, having some null values, and a few duplicates. (Location)

The last step we need to make before starting our project is planning and managing our dependencies, since many will be used to read from different file formats like GSON for parsing JSON, and Hibernate’s many dependent libraries like ANTLR, dom4j, slf4j and many more. To make this process easier and automatic, we will convert it to a Maven project, and simply inject any dependency that we need to the pom.xml file, thus letting Maven take care of all the related dependencies for us.

# Parsing from files

To make our program extensible and add as many formats as needed in the future without editing our code, we built an *IParser* interface, that any file reading class must implement. *IParser* is a functional interface that only has one method *readData,* that takes the path of the file as an argument and returns a *Contacts* class, which wraps a *Set<Contact>,* each *Contact* containing 1 record of data. Next, we want to add our 3 parsing classes: *CSVParser,* *JSONParser,* and *XMLParser.*

## CSVParser

First, we start with the CSVParser, which uses a simple *BufferedReader* to read through all its records, each separated by a new line character, and each attribute separated by a comma. The CSV attributes need to follow this format: name,familyName,birthday,gender,location/longitude,location/latitude,occupation,notes,group,email,phoneOne/type,phoneOne/number,phoneTwo/type,phoneTwo/number,address/type,address/street,address/city,address/region,address/country,organization/type,organization/name,organization/location/longitude,organization/location/latitude,organization/jobDescription,website/type,website/url.  
Once the format is set, we split each record with a regular expression “,(?=[^ ])” which matches all records separated by a comma that is not followed by a space, because some of our records contain commas. After that, each value of the generated String [] is mapped to its corresponding attribute, which we then fill in our *Contact.*

## JSONParser

Now that we have our contacts filled in the memory, we can create a JSON format file by serializing our objects, using the [GSON](https://sites.google.com/site/gson/) library. This creates a JSON Array of Contact containing all the information. The same way we serialized our object; we can create a method to deserialize from JSON format to objects and wrap our returned Array in a Contacts class.

## XMLParser

The same way we created the JSON parser, only this time we need to decorate some of our attributes with JAXB Annotations, specifying *Contacts* as ***@XmlRootElement,*** and each of the classes as ***@XmlElement***, and the attributes as ***@XmlAttribute***. Serialization and deserialization were then applied, and the parser was ready to be used.

# Database creation

Next up, we need to create the database to store our contact information. First, we must set the hibernate configurations in our <hibernate.cfg.xml> file. For this project to run on another machine, this file needs to be edited. The ***connection.username*** and ***connection.password*** properties need to be set the user’s login credentials in a MySQL environment. Next we specify our driver, the ***com.mysql.cj.jdbc.Driver,*** and we’ll use the MySQL8Dialect (latest version as of writing this.) We then map the classes from our Data package.

After setting up the configurations, we need to set up our classes, by implementing the Serializable interface as required by Hibernate, and decorating them with the corresponding JPA annotations. Most of the constraints were left empty, but they can easily be modified to satisfy the client’s needs. Briefly, the following annotations were added:

***@Entity***: mark as entity  
***@Id*** and ***@GeneratedValue***: mark as automatically generated Primary Key for the entity.  
***@OnetoOne*** and ***@JoinColumn:*** mark a one-to-one relationship. Ex: each contact has exactly one “Phone” and vice-versa.  
***@OnetoMany:*** mark a one-to-many relationship. Ex: each Organization has many Contacts

Note that we also marked all relationships as ***cascadeType.ALL,*** which will propagate the following operations PERSIST, REMOVE, REFRESH, MERGE, DETACH to all related entities.

At this point, we can create a hibernate session (in our HibernateUtil class), and start adding our entities one after the other, using Hibernate’s saveOrUpdate() function, which will take care of all duplicates, and cascade information to all related entities. All what we need to do now is create a function *batch\_insert,* in which we loop through all the Contacts in memory, imported from a file or created manually, and save them one by one, to serve as checkpoints in case a constraint was violated in one of them.

CRUD operations

To make our CRUD operations extensible, we create a simple interface IDbEntity, that has two simple functions: a getter and a setter for an ID. We implement it in all our classes, then we use simple logic and Hibernate functions like *session.remove* for deletion, *session.merge* for update, and *session.addOrUpdate* for creation.

For our filtered search, it gets a little more complicated. The function *searchBasedOnCriteria* takes a Map<String, String> and creates a *criteriaQuery* based on it, where:

* Key of type String: A String containing the class of the entity and the attribute as String, separated by a space. If the attribute searched is a direct attribute to our root class Contact, the first element is ignored.  
  Ex: “name” will filter based on the Contact’s name, while “Organization name” will filter based on the contact’s organization name.
* String value: The value corresponding to the attribute specified.

Each entry is converted to criteriaQuery of form “WHERE ***key*** = ***value***”

We can add as many criteria to filter on, by looping through all the entry set of the map and

combining the predicates with AND.

Ex: The resulting query of a map containing {name: x , phone number : y} would be: SELECT FROM Contact WHERE name = x AND phone.number = y;

# Command Line Interface and Mapping

Now that everything is set up. All what is left it to make a CLI for the user to interact with, and map to the corresponding function. We do that by using multiple maps, each serving a different goal:

* ***inputFilter*** : filters the important keywords and their aliases out of the input, to be used in functions.
* ***attributeNames*** : filters the attribute names and their aliases.
* ***stringToClassName*** : filters the classes names to the user input. Ex: phone phoneOne
* ***stringToClass*** : same as above, but to Class type directly. Ex: phoneOne Phone.Class

The mapping mechanism works by reading input from the user, then looping through each word and checking if it’s a keyword, an attribute name or a class name. If it’s an attribute, then try to read the next value. If it’s a class name, try to find an attribute and a value next.

After filling our maps, we make a simple switch and map each keyword with its function or functions, depending on the maps or the keyword itself.

# Conclusion

This program reads input from the user and makes queries to search and filter or perform CRUD operations on entities stored in a MySQL database. Many features could be added and improvements to be made, such as using a builder design pattern for the *searchBasedOnCriteria* to create the desired class at runtime without unchecked casting, server hosting of the database can be made for set-up simplicity and unified information, create an API using spring boot, etc.