ICSI 410. Database Systems -- Project 0 (30 points)

Goals

- Install Java and Eclipse (if not yet installed)
- Create a Java project in Eclipse
- Import Java source code into an Eclipse Project
- Start using `javadoc`
- Apply OOP concepts to Java code
- Understand the challenges in developing applications without using a database

PLEASE compress the project containing your code into a zip file (in the form of `[your first name]_[your last name].zip`) and then submit that file on Brightspace (see PART X below).

PART I: Installing Java (if not yet installed)

- Start a Terminal (Linux or Mac) or a Command Prompt (Windows)
- Type:

```
java -version
```

- If the version is before version 8 or no such information is shown, install Java after visiting: https://java.com/en/download/

PART II: Installing Eclipse (if not yet installed)

- Visit: https://www.eclipse.org/downloads/
- Download the installer.
- Start `Eclipse installer` and then choose `Eclipse IDE for Java Developers` and press the `INSTALL` button.

PART III: Creating a Java project in Eclipse

- Start Eclipse.
- In the menu bar, choose `File`, `New`, and then `Java Project`.
- In the `Project name` text box, enter a name (e.g., `csi410`).

PART IV: Importing `bank.zip` into Eclipse

- In the `Package`, `Project Explorer`, or `Navigator` window, choose a project (e.g., `csi410`).
- In the menu bar, choose `File`, `Import`, `General`, and then `Archive File`. Next, press the `Next` button.
- Click the `Browse...` button and then choose the `bank.zip` file and press the `Open` button.
- In the `Import` dialog box, press the `Finish` button.
- Expand the project, the `src` folder, and then the `bank.nodb` package. If you cannot see this `bank.nodb` package in the `src` folder, then it means that, for some reason, the java code is imported into a different directory and Eclipse may not compile the needed Java code automatically. In this case, (i) right-click the `src` folder, (ii) choose `New` and then `Package`, (iii) create a package named `bank.nodb` and then (iv) move the imported java files including `UnitTests.java` to the `bank.nodb` package.
- In the `bank.nodb` package, double-click the `UnitTests.java` file. If there are many compile errors in `UnitTests.java`, then it means that the Java project is not configured to run JUnit tests. In this case, (i) right-click the Java project, (ii) choose `Properties` and then `Java Build Path`, (iii) choose the `Libraries` tab, (iv) in `Jars and class folders on the build path:`, choose `Classpath` and then the `Add Library...` button, (v) in the `Add Library` dialog box, choose `JUnit` and then `JUnit5` or `Junit4`, and finally (vi) press the `Finish`button and the `Apply and Close` button. If the Java project is not configured to run JUnit tests, then another possible solution is to (i) right-click the error icon on line 3 (`import static org.junit.Assert.*;`) in UnitTests.java, (ii) in the popup, choose `Fix project setup...`, (iii) in the `Project Setup Fixes` dialog box, choose `Add Unit 4 library to the build path`, and (iv) press the `OK` button. If you still see many compile errors in `UnitTests.java` although you tried the two mentioned above, then check if there is any file named `module-info.java`. If so, please remove that file.
- You can run the unit tests in `UnitTests.java` by pressing the `Run` button which contains a green circle and a white triangle. These unit tests will fail until you complete Parts 2 and 3 below.

PART V: Creating API documents using `javadoc`

- Click the `csi410` project icon in the `Navigator` or `Project Explorer` window.
- Select `Generate Javadoc` from the `Project` menu.
- In the `Generate Javadoc` dialog box, press the `Finish` button.
- See that some new folders such as `doc` and `doc.resources` are created in the project.

- To open the newly created HTML documentation files, just double-click them (you can start with `index.html`).

PART VI: Understanding the code in `Customer.java`

The purpose of the `Customer` class is to represent customers in a banking context. The `Customer` class includes member variables to represent the customer number (`customerNumber`) and ZIP code (`zipCode`) of each customer. The constructor `Customer(String customerNumber, int zipCode)` can create instances of the `Customer` class while setting the `customerNumber` and `zipCode` member variables of each instance to the parameters passed to the constructor. When the `customerNumber()` method is called on a `Customer` instance, the method returns the value of the `customerNumber` member variable of that `Customer` instance. The `zipCode()` and `toString()` methods return the value of the `zipCode` member variable and a string representation of the `Customer` instance, respectively.

PART VII: Understanding the code in `BankAccount.java`

Each instance of the `BankAccount` class has the following three member variables to represent an account in a banking system:

- `accountNumber`: the unique identifier of the bank account.
- `customerNumber`: the customer number of the owner of the account.
- `balance`: the balance of the account

The `BankAccount(String accountNumber, String customerNumber, double balance)` constructor can create `BankAccount` instances while setting the three member variables of each instance to the parameter values passed to the constructor.

Given a `BankAccount` instance, the `accountNumber()`, `customerNumber()`, and `balance()` methods return the values of the `accountNumber`, `customerNumber`, and `balance` member variables of that instance. The `toString()` method returns a string representation of the `BankAccount` instance.

PART VIII: Understanding the code in `Bank.java`

Each instance of the `Bank` class represents a bank. Its `customers` member variable references a map that can quickly access each `Customer` instance given the corresponding `customerNumber`. Similarly, the `accounts` member variable references another map that can access `BankAccount`instances given their `accountNumber`. Given a `Bank` instance, the `register(Customer customer)` and `register(BankAccount bankAccount)` methods can add to the `Bank` instance the specified `Customer` and `BankAccount` instances, respectively.

Consider the following code in `BankAccount#main(String[])`:

```
    var bank = new Bank("Sample");
    addData(bank, 10);
```

The above code constructs a `Bank` which is named `Sample` and has 10 `Customer`s and 21 `BankAccount`s.

Next, the following code in `BankAccount#main(String[])`:

```
    System.out.println("customers:");
    bank.customers.values().stream().forEach(c -> System.out.println(c));
```

shows each `Customer` registered in the `Bank` as follows:

```
customers:
{customerNumber=C00, zipCode=12222}
{customerNumber=C01, zipCode=12223}
{customerNumber=C02, zipCode=12224}
{customerNumber=C03, zipCode=12225}
{customerNumber=C04, zipCode=12222}
{customerNumber=C05, zipCode=12223}
{customerNumber=C06, zipCode=12224}
{customerNumber=C07, zipCode=12225}
{customerNumber=C08, zipCode=12222}
{customerNumber=C09, zipCode=12223}
```

Note that the above code obtains a `Stream` of `Customer`s by calling `bank.customers.values().stream()` and then, for each `c` in that `Stream`, executes `System.out.println(c)`.

Similarly, the following code:

```
    System.out.println("accounts:");
    bank.accounts.values().stream().forEach(a -> System.out.println(a));
```

outputs each `BankAccount` registered in the `Bank` as follows:

```
accounts:
{accountNumber=A00, customerNumber=C00, balance=1000.0}
{accountNumber=A01, customerNumber=C00, balance=10000.0}
{accountNumber=A02, customerNumber=C01, balance=100000.0}
{accountNumber=A03, customerNumber=C01, balance=1000.0}
{accountNumber=A04, customerNumber=C02, balance=10000.0}
{accountNumber=A05, customerNumber=C02, balance=100000.0}
{accountNumber=A06, customerNumber=C03, balance=1000.0}
{accountNumber=A07, customerNumber=C03, balance=10000.0}
```

```
{accountNumber=A08, customerNumber=C04, balance=100000.0}
{accountNumber=A09, customerNumber=C04, balance=1000.0}
{accountNumber=A10, customerNumber=C05, balance=10000.0}
{accountNumber=A11, customerNumber=C05, balance=100000.0}
{accountNumber=A12, customerNumber=C06, balance=1000.0}
{accountNumber=A13, customerNumber=C06, balance=10000.0}
{accountNumber=A14, customerNumber=C07, balance=100000.0}
{accountNumber=A15, customerNumber=C07, balance=1000.0}
{accountNumber=A16, customerNumber=C08, balance=10000.0}
{accountNumber=A17, customerNumber=C08, balance=100000.0}
{accountNumber=A18, customerNumber=C09, balance=1000.0}
{accountNumber=A19, customerNumber=C09, balance=10000.0}
{accountNumber=A20, customerNumber=C09, balance=100000.0}
```

Note also that the following code:

```
    System.out.println("accounts with balance > 10,000:");
    bank.queryBankAccounts(10000).forEach(a -> System.out.println(a));
```

outputs:

```
accounts with balance > 10,000:
{accountNumber=A02, customerNumber=C01, balance=100000.0}
{accountNumber=A05, customerNumber=C02, balance=100000.0}
{accountNumber=A08, customerNumber=C04, balance=100000.0}
{accountNumber=A11, customerNumber=C05, balance=100000.0}
{accountNumber=A14, customerNumber=C07, balance=100000.0}
{accountNumber=A17, customerNumber=C08, balance=100000.0}
{accountNumber=A20, customerNumber=C09, balance=100000.0}
```

The reason for the above output is that the `queryBankAccounts(double amount)` method returns a `Stream` of `BankAccount`s whose `balance` is greater than the specified `amount`:

```
public Stream<BankAccount> queryBankAccounts(double amount) {
    return accounts.values().stream().filter(b -> b.balance() > amount);
}
```

Consider the next block of code:

```
    System.out.println("ZIP code of the owner of account A10: "
            + bank.queryZipCode("A10"));
    System.out.println("ZIP code of the owner of account A11: "
            + bank.queryZipCode("A11"));
    System.out.println("ZIP code of the owner of account A15: "
            + bank.queryZipCode("A15"));
```

, which uses:

```
public Integer queryZipCode(String accountNumber) {
    var account = accounts.get(accountNumber);
    if (account == null)
        return null;
    var customer = customers.get(account.customerNumber());
    return customer == null ? null : customer.zipCode();
}
```

This `queryZipCode(String accountNumber)` method finds, for the given `accountNumber`, the `BankAccount` having that `accountNumber` as well as the `Customer` having the `customerNumber` of that `BankAccount`. For this reason, we can obtain the output below:

```
ZIP code of the owner of account A10: 12223
ZIP code of the owner of account A11: 12223
ZIP code of the owner of account A15: 12225
```

Next, the code below:

```
    System.out.println("account number, ZIP code");
    bank.queryAccountNumberZipCode().forEach(
            e -> System.out.println(e.getKey() + ", " + e.getValue()));
```

produces this output:

```
account number, ZIP code
A00, 12222
A01, 12222
A02, 12223
A03, 12223
A04, 12224
A05, 12224
A06, 12225
A07, 12225
A08, 12222
A09, 12222
A10, 12223
A11, 12223
A12, 12224
A13, 12224
A14, 12225
A15, 12225
A16, 12222
A17, 12222
A18, 12223
A19, 12223
A20, 12223
```

Note that the above output results from the following method which provides, for each pair of `accountNumber` and `customerNumber` from the `accounts` member variable, a pair of `String` and `Integer` representing that `accountNumber` and the `zipCode` of the related `Customer`:

```
public Stream<Map.Entry<String, Integer>> queryAccountNumberZipCode() {
    return accounts.values().stream()
            .map(a -> Map.entry(a.accountNumber(),
customers.get(a.customerNumber()).zipCode()));
}
```

Also, the `queryTotalAccountBalance()` and `queryMaximumAccountBalance()` methods can find the sum and the maximum of the `balance`s of the `BankAccount`s in `accounts`, respectively:

```
public Double queryTotalAccountBalance() {
    return accounts.values().stream().mapToDouble(a -> a.balance()).sum();
}

public Double queryMaximumAccountBalance() {
    return accounts.values().stream().
            mapToDouble(a -> a.balance()).max().getAsDouble();
}
```

Therefore, the following code:

```
    System.out.println("sum of account balances: " +
bank.queryTotalAccountBalance());
    System.out.println("maximum of account balances: " +
bank.queryMaximumAccountBalance());
```

outputs:

```
sum of account balances: 777000.0
maximum of account balances: 100000.0
```

Note that the code below:

```
    System.out.println("ZIP code, number of customers");
    bank.queryZipCodeCustomers().forEach(e ->
        System.out.println(e.getKey() + ", " + e.getValue()));
```

outputs:

```
ZIP code, number of customers
12224, 2
12225, 2
12222, 3
12223, 3
```

using:

```
public Stream<Entry<Integer, Long>> queryZipCodeCustomers() {
    var summary = customers.values().stream()
            .collect(Collectors.groupingBy(c -> c.zipCode(),
                    Collectors.counting()));
    return summary.entrySet().stream();
}
```

The `queryZipCodeCustomers()` method groups the `Customer`s by their `zipCode` and then finds the number of `Customer`s for each `zipCode`.

PART IX: Completing `Bank.java`

The goal of this part is to complete three additional methods in the `Bank` class. Each of the tasks accounts for 10 points.

Task 1 (10 points). In `Bank.java`, implement the `queryCustomers(int zipCode)` method so that it can return a `Stream` of `Customer`s having the specified `zipCode`. Consider retrieving all the `Customer`s in the `customers` member variable and then selecting only the relevant ones from the `Customer`s.

If you implement this method correctly, then the following code in `Bank#main(String[])`:

```
    System.out.println("customers with ZIP code 12222:");
    bank.queryCustomers(12222).forEach(c -> System.out.println(c));
```

will output:

```
customers with ZIP code 12222:
{customerNumber=C00, zipCode=12222}
{customerNumber=C04, zipCode=12222}
{customerNumber=C08, zipCode=12222}
```

Also, the following code:

```
    System.out.println("customers with ZIP code 12225:");
    bank.queryCustomers(12225).forEach(c -> System.out.println(c));
```

will output:

```
customers with ZIP code 12225:
{customerNumber=C03, zipCode=12225}
{customerNumber=C07, zipCode=12225}
```

When you finish implementing the `queryCustomers(int zipCode)` method, please make sure that your code passes the unit test named `task1()` in `UnitTests.java`.

Task 2 (10 points). Implement the `queryZipCodeAccounts()` method so that it can return a `Stream` of `Entry`s each having a `zipCode` and the number of `BankAccount`s having `Customer`s with that `zipCode`.

If you complete this method, the following code in `Bank#main(String[])`:

```
    System.out.println("ZIP code, number of accounts");
    bank.queryZipCodeAccounts().forEach(e -> System.out.println(e.getKey() +
            ", " + e.getValue()));
```

will output:

```
ZIP code, number of accounts
12224, 4
12225, 4
12222, 6
12223, 7
```

Your code needs to pass the unit test named `task2()` in `UnitTests.java`.

Task 3 (10 points). Implement the `queryMaxBalanceBankAccounts()` method so that it can find the `BankAccount`s with the maximum balance (i.e., those whose `balance` is not smaller than the `balance` of any other `BankAccount`).

If you complete this method, the following code in `Bank#main(String[])`:

```
    System.out.println("accounts with the maximum balance");
    bank.queryMaxBalanceBankAccounts().forEach(a -> System.out.println(a));
```

will output:

```
accounts with the maximum balance
{accountNumber=A02, customerNumber=C01, balance=100000.0}
{accountNumber=A05, customerNumber=C02, balance=100000.0}
{accountNumber=A08, customerNumber=C04, balance=100000.0}
{accountNumber=A11, customerNumber=C05, balance=100000.0}
{accountNumber=A14, customerNumber=C07, balance=100000.0}
{accountNumber=A17, customerNumber=C08, balance=100000.0}
{accountNumber=A20, customerNumber=C09, balance=100000.0}
```

Please ensure that your code passes the unit test named `task3()` in `UnitTests.java`.

PART X: Submitting Your Work

- In Eclipse, choose the `src` source folder and then right-click the `bank.nodb` package.
- In the popup menu, choose `Export...`
- In the `Export` dialog box, choose `General` and `Archive File`.
- Make sure all the files in the `bank.nodb` package are selected.
- In the `To archive file` text box, type the name of the zip file (in the form of `[your first name]_[your last name].zip`). Please remember the folder where the zip file will be saved.
- Press the `Finish` button.
- On Brightspace, upload the zip file mentioned above.

- Next, download your submission from Brightspace and make sure that your zip file contains all the needed files (e.g., `Bank.java`, `BankAccount.java` , and `Customer.java`).