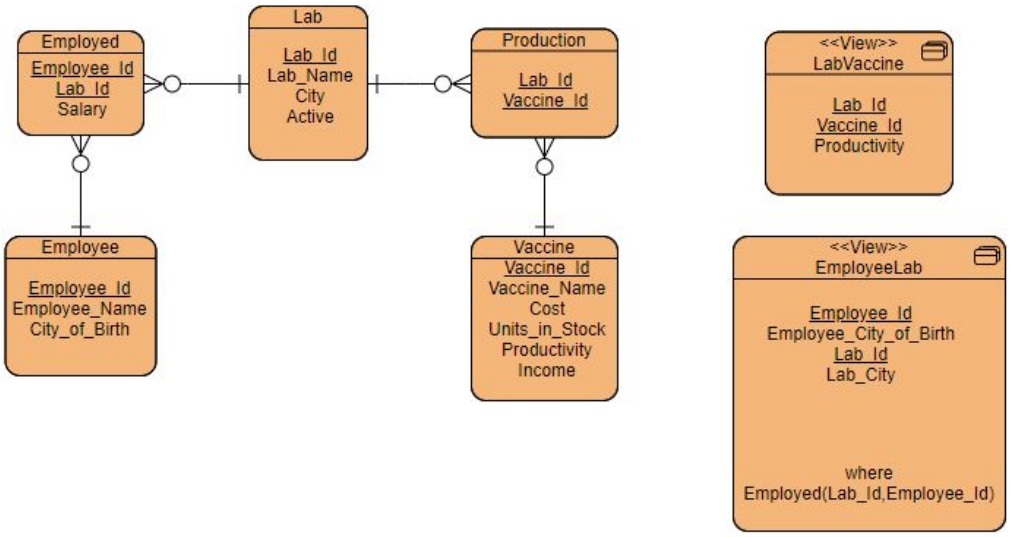
Assignment 2
SQL Programming

Anna

CoronaRecord *is a smart service that gives you statistics about labs and recommends vaccines based on people's rates.*

The mission: design the database and implement the data access layer of the system.

DB design (including views);



a design draw.

DB Overview;

DB Tables Reflecting the Business Object Entities;

Lab

Description	Type	Comments
Lab_Id	Integer	The ID of the lab.
Lab_Name	String	The name of the lab.
City	String	The city where the lab is located.
Active	Boolean	A Boolean attribute indicating whether the lab is active or not.

The Lab table has the exact same attributes as the Lab business object.
Constraints;
Lab_Id is the key attribute, Lab_Id >0, Lab_Name, City and Active are not null.

Employee

Description	Type	Comments
Employee_Id	Integer	The ID of the employee.
Employee_Name	String	The name of the employee.
City_of_Birth	String	The city that the employee was born in.

The Employee table has the exact same attributes as the Employee business object.
Constraints;
Employee_Id is the key attribute, Employee_Id >0, Employee_Name and City_of_Birth are not null.

Vaccine

Description	Type	Comments
Vaccine_Id	Integer	The ID of the vaccine.
Vaccine_Name	String	The name of the vaccine.
Cost	Integer	The cost of the vaccine.
Units_in_Stock	Integer	The amount of vaccines in stock.
Productivity	Integer	The productivity (in %) of the vaccine.
Income	Integer	The total income generated from selling the vaccine.

The Vaccine table has the exact same attributes as the Vaccine business object, and the additional attribute Income.
Constraints;
Vaccine_Id is the key attribute, Vaccine_Id >0, Vaccine_Name, Cost, Units_in_Stock, Productivity and Income are not null, Cost≥0, Units_in_Stock≥0, 0≤Productivity≤100, Income≥0.

DB Tables Reflecting the Relationships b/w the Business Object Entities;

Employed

Description	Type	Comments
Employee_Id	Integer	The ID of the employee.
Lab_Id	Integer	The ID of the lab.
Salary	Integer	The salary of the employee working at the lab.

The Employed table reflects the many to many Relationship b/w the Employee and Lab tables.
Constraints;
Employee_Id and Lab_Id are the key attributes, Employee_Id is a foreign key referencing the key attribute Employee_Id of the Employee table, Lab_Id is a foreign key referencing the key attribute Lab_Id of the Lab table, Salary is not null.

Production

Description	Type	Comments
Lab_Id	Integer	The ID of the lab.
Vaccine_Id	Integer	The ID of the vaccine.

The Production table reflects the many to many Relationship b/w the Lab and Vaccine tables.
Constraints;
Lab_Id and Vaccine_Id are the key attributes, Lab_Id is a foreign key referencing the key attribute Lab_Id of the Lab table, Vaccine_Id is a foreign key referencing the key attribute Vaccine_Id of the Vaccine table.

DB Views;

EmployeeLab

Description	Type	Comments
Employee_Id	Integer	The ID of the employee.
City_of_Birth	String	The city that the employee was born in.
Lab_Id	Integer	The ID of the lab.
City	String	The city where the lab is located.

The EmployeeLab view substitutes a query that returns the attributes Employee_Id, City_of_Birth, Lab_Id and City of the NATURAL JOIN of the Employed, Employee and the Lab tables on Employee_Id, Lab_Id.

LabVaccine

Description	Type	Comments
Lab_Id	Integer	The ID of the lab.
Vaccine_Id	Integer	The ID of the vaccine.
Productivity	Integer	The productivity (in %) of the vaccine.

The LabVaccine view provides access to the attributes Lab_Id, Vaccine_Id of the Production table and the attribute Productivity of the Vaccine table.
This view substitutes a query that returns the attributes Lab_Id, Vaccine_Id and Productivity of the NATURAL JOIN of the Production and the Vaccine tables on Vaccine_Id.

Design Decisions;

The business objects were translated into tables as is, with the exception of adding the Income attribute to the Vaccine table. This additional attribute aggregates the income generated from selling a vaccine throughout the lifecycle of the DB. As will be shown in the API implementation section, storing this additional information will greatly simplify the implementation of the function `public static Integer getIncomeFromVaccine(Integer vaccineID)`.
The additional storage required is negligible relative to the computation improvement gain.

The relationship b/w the Employee and Lab tables was translated into the Employed table, exactly in the same way it was defined in the ERD tutorial, p.12. The Employed table has a key constructed from both attributes Employee_Id and Lab_Id to reflect the many-to-many relationship; each employee can work at several labs and each lab can have several employees.
Employee_Id and Lab_Id are set to be foreign keys of the Employee and Lab tables respectively, to enforce the correctness of the key values in order to maintain DB integrity.
The Salary attribute of the relationship table Employed stores the salary for each employment event.

The relationship b/w the Lab and Vaccine tables was translated into the Production table, exactly in the same way it was defined in the ERD tutorial, p.12. The Production table has a key constructed from both attributes Lab_Id and Vaccine_Id to reflect the many-to-many relationship; each lab can produce several vaccines and each vaccine can be produced at several labs.
Lab_Id and Vaccine_Id are set to be foreign keys of the Lab and Vaccine tables respectively, to enforce the correctness of the key values in order to maintain DB integrity.

The view EmployeeLab was created in order to simplify some of the queries in the implementation and avoid repetitive query code. It allows access to the location information of both employees and labs, naturally joined on the key fields of the Employed table. This becomes useful in the implementation of the functions `public static Integer getBestLab()`, `public static String getMostPopularCity()`.

The view LabVaccine was created in order to simplify a query in the implementation. This becomes useful in the implementation of the function `public static ArrayList<Integer> getPopularLabs()`.

This design yields a DB with minimal redundancy, strictly enforced value constraints and no NULL complications.
The DB integrity is easily maintained by the attribute value constraints enforcement mechanism.

The main driving principles for this design have been consistency, integrity and coherence of the data.

API Implementation;

Constants;

```
private final static String EMPLOYEE_TABLE_NAME = "Employee";
private final static String LAB_TABLE_NAME = "Lab";
private final static String VACCINE_TABLE_NAME = "Vaccine";
private final static String EMPLOYED_TABLE_NAME = "Employed";
private final static String PRODUCTION_TABLE_NAME = "Production";
private final static String LAB_VACCINE_VIEW_NAME = "LabVaccine";
private final static String EMPLOYEE_LAB_VIEW_NAME = "EmployeeLab";
```

These constants store the respective tables' and views' names in order to minimize typographical errors.

Functions;

`public static void createTables()`

Here all the tables and the views of the DB are created. For each table, a typed scheme is created via the object `ArrayList<Pair<String, String>> typedSchema`

And the private function `private static void createTable(String name, ArrayList<Pair<String, String>> typedSchema)` is invoked in order to create the table in the PostgreSQL server DB.

To create the views, the private functions `private static void createLabVaccineView()`, `private static void createEmployeeLabView()` are invoked.

Each attribute in a table corresponds to a `Pair` object in `typedSchema` for that table.

Each pair in `typedSchema` has as its key the name of the attribute and as its value the type (and the `"NOT NULL"` constraint on non-key attributes) of the attribute.

Additional `Pair` objects are added to define key, foreign key and values range constraints.

These constraints are implemented using the corresponding PostgreSQL constraints[1];

`attr is key;` is translated into `"PRIMARY KEY" (attr)"`

`(attr1, attr2) is key;` is translated into `"PRIMARY KEY (attr1, attr2)"`

`attr > 0;` is translated into `"CHECK","(attr > 0)"`

`0 > attr > 100;` is translated into `"CHECK attr BETWEEN 0 AND 100)"`

`attr is foreign key from tbl;` is translated into

`"FOREIGN KEY (attr) REFERENCES tbl(attr) ON DELETE CASCADE ON UPDATE CASCADE"`

Note; the PostgreSQL rule `ON DELETE CASCADE ON UPDATE CASCADE` added to the foreign key attributes allows to maintain the consistency of data, ensuring it always exists in a coherent state.

This means that changes on the referenced primary key are also applied to the referencing foreign key.

`private static void createTable(String name, ArrayList<Pair<String, String>> typedSchema)`

Here a string for the PostgreSQL query to create a new table is constructed and the query is executed.

The string is obtained from the `StringBuilder sb` object which is constructed from the `name` argument storing table's name and the `typedSchema` argument storing all the information needed to create the table; table attributes with corresponding types and table constraints.

Example for the resulting PostgreSQL query;

```
"CREATE TABLE Employee
(Employee_Id integer,
Employee_Name text NOT NULL,
City_of_Birth text NOT NULL,
PRIMARY KEY (Employee_Id),
CHECK","(Employee_Id > 0)
)"
```

[1] https://www.postgresql.org/docs/12/ddl-constraints.html

private static void createLabVaccineView()
Creates the view LabVaccine from the NATURAL JOIN of Production and Vaccine tables.

private static void createEmployeeLabView()
Creates the view EmployeeLab from the NATURAL JOIN of Employed, Employee and Lab tables.

public static void clearTables()
Clears all the tables in the DB.
First a query is executed to obtain the names of all the existing tables. This information is stored in information_schema.tables in the DB.
Next, for each table, a query is executed to clear it using the PostgreSQL command "DELETE FROM tableName CASCADE".

public static void dropTables()
Deletes all the tables and views from the DB.
First, a query is executed to obtain the names of all the existing views. For each view, a query is executed to delete it using the PostgreSQL command "DROP VIEW IF EXISTS viewName CASCADE".
Next, a query is executed to obtain the names of all the existing tables. For each table, a query is executed to delete it using the PostgreSQL command "DROP TABLE IF EXISTS tableName CASCADE".

public static ReturnValue addLab(Lab lab)
Adds a new lab to the DB. The constructed and executed query is "INSERT INTO Lab VALUES (lab.getId(), lab.getName(), lab.getCity(), lab.getIsActive())".
If the query is executed successfully, the returned value is *OK*.
If the DB throws an SQLException, it's caught and analysed to determine a suiting return value.
A *CHECK_VIOLATION* exception means the value constraints weren't met. In this case, lab_Id is not positive.
A *NOT_NULL_VIOLATION* exception means at least one of the values was NULL.
A *FOREIGN_KEY_VIOLATION* exception means the foreign key constraint wasn't met. (impossible in this case).
In all three of these cases the returned value is *BAD_PARAMS*.
A *UNIQUE_VIOLATION* exception means the primary key constraint wasn't met b/c a lab with lab.getId() already exists in the DB.
In this case the returned value is *ALREADY_EXISTS*.
In the case of any other SQLException the returned value is the general *ERROR*.

public static ReturnValue addEmployee(Employee employee)
Adds a new employee to the DB. Very similar to public static ReturnValue addLab(Lab lab).

public static ReturnValue addVaccine(Vaccine vaccine)
Adds a new vaccine to the DB. Very similar to public static ReturnValue addLab(Lab lab). The initial value for the Income attribute is set to 0.

public static ReturnValue employeeJoinLab(Integer employeeID, Integer labID, Integer salary)
Adds a new entry to the Employed table in the DB. Very similar to public static ReturnValue addLab(Lab lab).
On success the returned value is *OK*.
If the DB throws an SQLException, it's caught and analysed to determine a suiting return value.
A *CHECK_VIOLATION* exception means the value constraints weren't met. In this case, Salary is negative.
A *NOT_NULL_VIOLATION* exception means at least one of the values was NULL.
In both these cases the returned value is *BAD_PARAMS*.
A *FOREIGN_KEY_VIOLATION* exception means the foreign key constraint wasn't met. Either there is no employee with employeeID in the Employee table, or there is no lab with labID in the Lab table. (Or an Id value is not positive).
In this case the returned value is *NOT_EXISTS*.
A *UNIQUE_VIOLATION* exception means the primary key constraint wasn't met b/c there already exists an entry with employeeID and labID in the Employed table.
In this case the returned value is *ALREADY_EXISTS*.
In the case of any other SQLException the returned value is the general *ERROR*.

public static ReturnValue labProduceVaccine(Integer vaccineID, Integer labID)
Adds a new entry to the Production table.
Very similar to public static ReturnValue employeeJoinLab(Integer employeeID, Integer labID, Integer salary).

public static Lab getLabProfile(Integer labID)
Retrieves from the DB and returns a lab profile with labID. The constructed and executed query is
"SELECT * FROM Lab WHERE Lab_Id = labID"
If the returned by DB ResultSet is not empty, the first call to the method next() activates the first(and only) row and returns true. Then the methods public int getInt(int columnIndex), public String getString(int columnIndex), public boolean getBoolean(int columnIndex) are called to retrieve the values and set the corresponding fields in the newly created Lab object, that is then returned.
Otherwise, Lab.*badLab*() is returned as there is no lab with labID in the DB.
In case of an SQLException, a Lab.*badLab*() is returned.

public static Employee getEmployeeProfile(Integer employeeID)
Retrieves from the DB and returns an employee profile with employeeID.
Very similar to public static Lab getLabProfile(Integer labID).
If there is no employee with employeeID in the DB or an SQLException is caught, Employee.*badEmployee*() is returned.

public static Vaccine getVaccineProfile(Integer vaccineID)
Retrieves from the DB and returns a vaccine profile with vaccineID.
Very similar to public static Lab getLabProfile(Integer labID).
If there is no vaccine with vaccineID in the DB or an SQLException is caught, Vaccine.*badVaccine*() is returned.

public static ReturnValue deleteLab(Lab lab)
Deletes an entry with lab.getId() from the Lab table in the DB. The constructed and executed query is
"DELETE FROM Lab WHERE Lab_Id = lab.getId()"
The number of affected rows returned from the executed query is stored in affectedRows.
If affectedRows==1, the entry corresponding to lab.getId() was found and deleted. Therefore the returned value is *OK*. If affectedRows==0, no entry corresponding to lab.getId() was found. Therefore the returned value is *NOT_EXISTS*. In any other case, an error occurred and therefore the returned value is *ERROR*.
If an SQLException is caught, the returned value is *ERROR*.

public static ReturnValue deleteEmployee(Employee employee)
Deletes an entry with employee.getId() from the Employee table in the DB.
Very similar to public static ReturnValue deleteLab(Lab lab).

public static ReturnValue deleteVaccine (Vaccine vaccine)
Deletes an entry with vaccine.getId() from the Vaccine table in the DB.
Very similar to public static ReturnValue deleteLab(Lab lab).

public static ReturnValue employeeLeftLab(Integer labID, Integer employeeID)
Deletes an entry with (employeeID, labID) from the Employed table in the DB.
Very similar to public static ReturnValue deleteLab(Lab lab).

public static ReturnValue labStoppedProducingVaccine(Integer labID, Integer vaccineID)
Deletes an entry with (labID, vaccineID) from the Production table in the DB.
Very similar to public static ReturnValue deleteLab(Lab lab).

public static ReturnValue vaccineSold(Integer vaccineID, Integer amount)
Updates the Vaccine table after a vaccine was sold. The constructed and executed query is
"UPDATE Vaccine
SET (Units_in_Stock, Cost, Income, Productivity) =
(Units_in_Stock-amount, Cost*2, Income + Cost*amount, LEAST(Productivity+15,100))
WHERE Vaccine_Id = vaccineID"
The number of affected rows returned from the executed query is stored in affectedRows.
If affectedRows==1, the entry corresponding to vaccineID in the table Vaccine was found and updated successfully. Therefore the returned value is *OK*.
If affectedRows==0, no entry corresponding to vaccineID was found. Therefore the returned value is *NOT_EXISTS*.
In any other case, an error occurred and therefore the returned value is *ERROR*.
If an SQLException is caught, it is analysed to determine the suitable return value.
A *CHECK_VIOLATION* exception means the value constraints weren't met. In this case, Units_in_Stock-amount is negative, meaning the amount is greater than the units in stock.
In this case the returned value is *BAD_PARAMS*.
In case of any other SQLException, an error occurred and therefore the returned value is *ERROR*.

public static ReturnValue vaccineProduced(Integer vaccineID, Integer amount)
Updates the entry with vaccineID in the Vaccine table after an amount of the vaccine was sold.
The first query "SELECT 1 WHERE amount<0" simply checks whether amount is negative, in which case *BAD_PARAMS* is returned.
The second query executes the update;

```
"UPDATE Vaccine SET (Units_in_Stock, Cost, Productivity) =
    (Units_in_Stock+amount, Cost/2, GREATEST(Productivity - 15, 0))
    WHERE Vaccine_Id = vaccineID".
```

PostgreSQL function GREATEST is used in order to ensure the Productivity value is always non-negative.
All the computations in the query are performed in accordance to the specs for this function.
The number of affected rows returned from the executed query is stored in affectedRows.
If affectedRows==1, the entry corresponding to vaccineID in the table Vaccine was found and updated successfully. Therefore the returned value is *OK*.
If affectedRows==0, no entry corresponding to vaccineID was found. Therefore the returned value is *NOT_EXISTS*.
In any other case, an error occurred and therefore the returned value is *ERROR*.
If an SQLException is caught, it is analysed to determine the suitable return value.
A *CHECK_VIOLATION* exception means the value constraints weren't met.
In this case the returned value is *BAD_PARAMS*.
In case of any other SQLException, an error occurred and therefore the returned value is *ERROR*.

public static Boolean isLabPopular(Integer labID)
Returns true if all of the lab's vaccines have a productivity of over 20%, false in any other case.
The first query " SELECT Lab_Id FROM Lab WHERE Lab_Id = labID" checks whether a lab with labID exists in the DB.
If the returned by DB ResultSet is not empty, the first call to the method next() activates the first(and only) row and returns true. Otherwise, it returns false, therefore no lab with labID exists in the DB and the return value is false.
The second query " SELECT Vaccine_Id FROM Lab WHERE Lab_Id = labID AND Productivity <= 20" queries for any vaccines with low Productivity that the lab might produce. If the returned by DB ResultSet is not empty, the returned value is false. Otherwise, the lab with labID exists in the DB and produces no vaccines with low productivity and therefore the returned value is true.
In case of any SQLException, an error occurred and therefore the returned value is false.

public static Integer getIncomeFromVaccine(Integer vaccineID)
Returns the income generated from sales of the vaccine.
The constructed and executed query is " SELECT Income FROM Vaccine WHERE Vaccine_Id = vaccineID".
If the returned by DB ResultSet is not empty, the first call to the method next() activates the first(and only) row and returns true. In this case a vaccine with vaccineID indeed exists in the DB and the corresponding Income value is retrieved by the method public int getInt(int columnIndex) and returned.
Otherwise, the method next() returns false, meaning no vaccine with vaccineID exists in the DB, therefore the returned value is 0.
In case of any SQLException, an error occurred and therefore the returned value is 0.

public static Integer getTotalNumberOfWorkingVaccines()
Returns the total number of vaccines with Productivity over 20%.
The constructed and executed query is " SELECT SUM(Units_in_Stock) FROM Vaccine WHERE Productivity > 20"
PostgreSQL function SUM is used in order to sum up the number of units in stock for each vaccine entry in the Vaccine table with Productivity value over 20%.
If there are no entries in the Vaccine table with Productivity value over 20%, the returned by DB ResultSet is empty.
Therefore, first call to the results.next() method returns false, and the returned value is 0.
Otherwise, the method public int getInt(int columnIndex) is called to retrieve and return the query result.
In case of any SQLException, an error occurred and therefore the returned value is 0.

public static Integer getTotalWages(Integer labID)
Returns the sum of salaries of lab's employees, if there are at least two employees and the lab is active.
Otherwise, returns 0.
The first constructed and executed query is " SELECT Active FROM Lab WHERE Lab_Id = labID".
This checks whether the lab exists and is active.
If the returned by DB ResultSet is not empty, the first call to the method next() activates the first(and only) row and returns true. In this case a lab with labID exists in the database. The corresponding Active value is retrieved by calling the public boolean getBoolean(int columnIndex) method. If the returned value is true, the second query is constructed and executed.
Otherwise, if there is no lab with labID in the DB, or it is not active, the returned value is 0.
The second constructed and executed query is

```
"SELECT total_salary
 FROM
  (SELECT COUNT(*) AS Number_of_Employees, SUM(Salary) AS Total_Salary FROM Employed
   WHERE Lab_Id = labID) AS T1
 WHERE Number_of_Employees>1".
```

The subquery T1 counts the number of employees and sums up the Salary values for all the rows in the table Employed corresponding to labID.
The main query only returns the salaries sum if the number of employees at the lab is over one.
Therefore, after executing the query, the returned ResultSet is either empty and the returned value is 0, or contains one cell storing the sum of the salaries, in which case it is retrieved by the public int getInt(int columnIndex) method and returned.
In case of any SQLException, an error occurred and therefore the returned value is 0.

`public static Integer getBestLab()`
As detailed in the specs.
The constructed and executed query is

```
"SELECT Lab_Id
    FROM
      (SELECT Lab_Id, count(*) AS Local_Employees
       FROM EmployeeLab
       WHERE City_of_Birth = City
       GROUP BY Lab_Id) as T1
    WHERE Local_Employees > 0
    ORDER BY Local_Employees DESC,Lab_Id
    LIMIT 1".
```

The subquery T1 selects Lab_Id and the count of number of rows in each group from all the rows from the EmployeeLab view where the city the lab is located in and the city its employee was born at coincide, grouped by Lab_Id.
The main query having a table T1 of Lab_id and the number of local employees for each lab that has any, orders those by the number of local employees and then by Lab_Id and returns the Lab_Id of the top row.
If the returned by DB ResultSet is not empty, its only cell is retrieved by the `public int getInt(int columnIndex)` method and returned.
Otherwise, the returned value is 0.
In case of any SQLException, an error occurred and therefore the returned value is 0.

`public static String getMostPopularCity()`
As detailed in the specs.
The constructed and executed query is

```
"SELECT City_of_Birth
    FROM
    (SELECT City_of_Birth, COUNT(*) AS Employees_From_City
    FROM EmployeeLab
    GROUP BY City_of_Birth) AS T1
    ORDER BY Employees_From_City DESC,City_of_Birth DESC
    LIMIT 1"
```

The subquery T1 groups the rows of the EmployeeLab view by City_of_birth, and selects the attribute City_of_Birth and the count of rows in each group.
The main query having a table T1 with all the relevant cities and the number of employees working that are from that city for each one, orders it by the number of employees and then by the city names in descending order and selects the City_of_Birth value of the top row.
If the returned by DB ResultSet is not empty, its only cell is retrieved by the `public String getString(int columnIndex)` method and returned.
Otherwise, the returned value is the empty string `""`.
In case of any SQLException, an error occurred and therefore the returned value is `null`.

public static ArrayList<Integer> getPopularLabs()
As detailed in the specs.
The constructed and executed query is
"SELECT DISTINCT Lab_Id
FROM LabVaccine
EXCEPT
(SELECT DISTINCT Lab_Id
FROM LabVaccine
WHERE Productivity<=20)
ORDER BY Lab_Id
LIMIT 3".
The query selects Lab_Ids of all the labs that produce at least one vaccine (and therefore are in the Production table and thus in the LabVaccine view), but subtracts from the result Lab_Ids of all the labs that have an entry in the LabVaccine view with Productivity value <= 20 (and therefore have an entry in the Production table with the corresponding vaccine). The result is ordered by Lab_Id and the top 3 rows are returned.
The object ArrayList<Integer> popularLabs is initialized to be empty. If the returned by DB ResultSet is not empty, its values are iterated upon, retrieved by the public int getInt(int columnIndex) method and added to popularLabs.
The returned value is popularLabs.
In case of any SQLException, an error occurred and therefore the returned value is an empty ArrayList<> object.

public static ArrayList<Integer> getMostRatedVaccines()
As detailed in the specs.
The constructed and executed query is
"SELECT Vaccine_Id
FROM
(SELECT Vaccine_Id, Productivity + Units_in_Stock - Cost as Rating
FROM Vaccine
ORDER BY Rating DESC, Vaccine_Id
LIMIT 10) AS T1".
The subquery selects Vaccine_Id and the calculated Rating from the Vaccine table, ordered by Rating in descending order and then by Vaccine_Id and limited to top 10 rows.
The main query then just selects the Vaccine_Id values from T1.
Note; it was possible to implement this w/o a subquery, but this implementation is more explanatory.
The object ArrayList<Integer> mostRatedVaccines is initialized to be empty. If the returned by DB ResultSet is not empty, its values are iterated upon, retrieved by the public int getInt(int columnIndex) method and added to mostRatedVaccines.
The returned value is mostRatedVaccines.
In case of any SQLException, an error occurred and therefore the returned value is an empty ArrayList<> object.

`public static ArrayList<Integer> getCloseEmployees(Integer employeeID)`
As detailed in the specs.
The first constructed and executed query is

```
"SELECT Employee_Id
    FROM Employee
    WHERE Employee_Id = employeeID".
```

This query only checks whether an entry with `employeeID` exists in the Employee table.
If not, an empty `ArrayList<>` object is returned.

The second constructed and executed query is

```
"SELECT COUNT(Lab_Id)
    FROM Employed
    WHERE Employee_Id = employeeID".
```

This query counts the number of jobs the employee has by selecting the count of the rows of the Employed table with `employeeID`. If the returned by DB `ResultSet` is not empty, its value is retrieved by the `public int getInt(int columnIndex)` method and stored in `employeeJobs`.
If `employeeJobs == 0`, the employee doesn't work and thus all other employees are vacuously close.

In this case The third query is constructed and executed;

```
"SELECT Employee_Id
    FROM Employee
    WHERE Employee_ID != employeeID
    ORDER BY Employee_Id
    LIMIT 10".
```

This query selects the top 10 employees from the Employee table that are not the current employee and are ordered by `employeeID`.
The object `ArrayList<Integer> closeEmployees` is initialised to be empty. If the returned by DB `ResultSet` is not empty, its values are iterated upon, retrieved by the `public int getInt(int columnIndex)` method and added to `closeEmployees`.
The returned value is `closeEmployees`.

Otherwise, `employeeJobs != 0`, the fourth and final query is constructed and executed;

```
"SELECT Employee_ID
    FROM
        (SELECT Employee_ID, Count(Employee_ID) AS Shared_Labs
        FROM Employed
        WHERE Lab_ID IN
                (SELECT Lab_ID
                FROM Employed
                WHERE Employee_ID = employeeID) AND Employee_ID != employeeID
        GROUP BY Employee_ID
        ORDER BY Shared_Labs DESC, Employee_ID
        ) AS T1
    WHERE Shared_Labs >= (SELECT Count(Employee_ID)
                FROM Employed
                WHERE Employee_ID = employeeID)/2.0
    ORDER BY Employee_ID
    LIMIT 10".
```

The subquery T1 from the table Employed, selects all the rows where the Lab_Id value is one of the Lab_Id values of the current employee (labs the employee works at), and the Employee_Id value is different from `employeeID`. Those rows are grouped by Employee_Id and ordered by the number of rows in each group (Shared_Labs), and then by Employee_Id.

The main query having the Employee_Id and the number of shared labs for each employee (except the current one) in T1, selects Employee_Id from top 10 rows, but only from rows where the number of shared labs is equal to or greater than 50% (this number is calculated by counting all the jobs current employee has, from the Employed table and dividing by 2.0 to obtain a float result), ordered by Employee_Id.