Development on the Java SE platform Part 2

Workbook
(version 1.2, 30.07.2021)

Objectives:	Learn the Java SE platform
The authors:	Cătălin Tudose (f,tudose@dxc.com) Vladimir Sonkin (vladimir.sokin@dxc.com)
Product version:	JavaSE 8 or higher

Table of Contents

Prerequisites
Installing and adjusting runtime environment

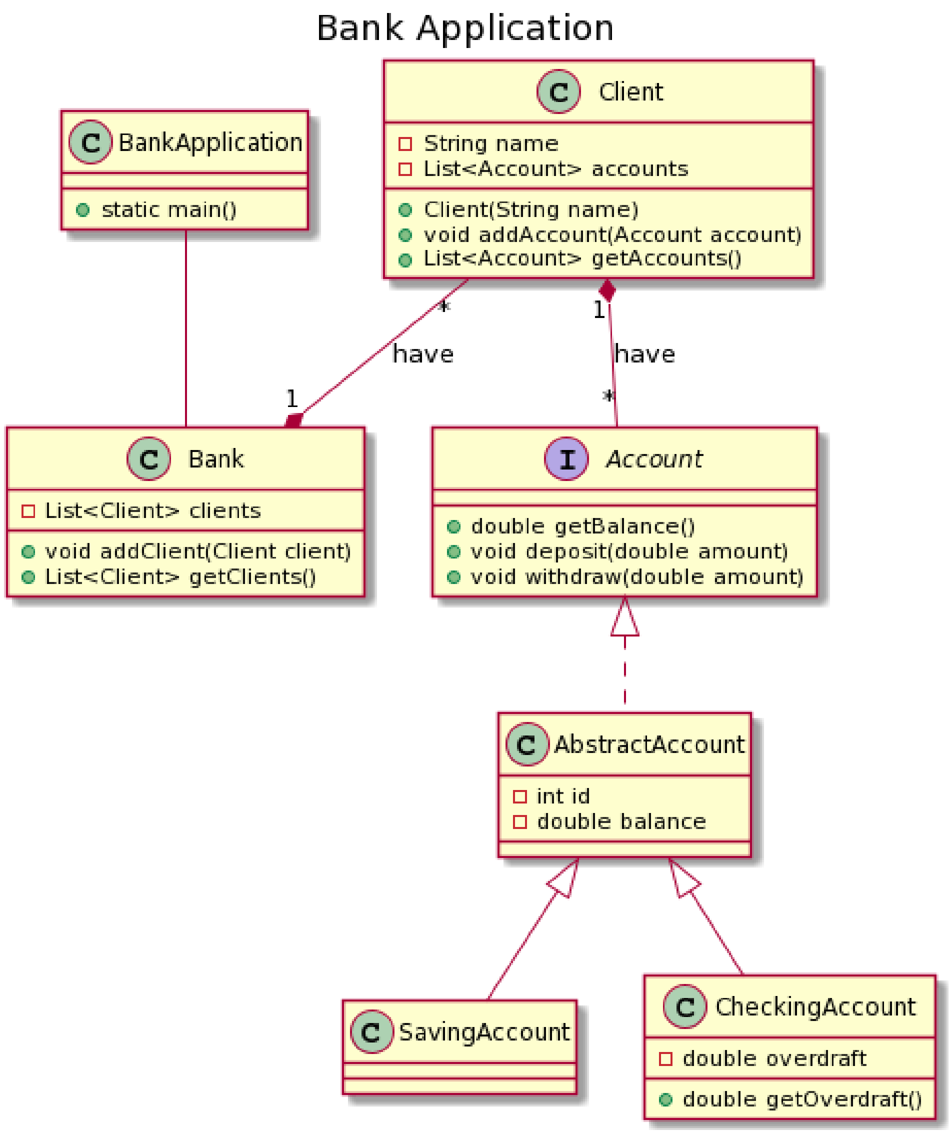
Objectives

Install and configure runtime environment in order to solve the exercises.

Tasks

1. You should download and install JDK 8 or higher from the www.oracle.com website.
2. You should download and install IntelliJ from the https://www.jetbrains.com/idea/download/ website or Eclipse from the www.eclipse.org website.

As the basic task for this part we will take the Bank Application. This application allows to manage a Bank. The Bank has clients, each client has accounts of 2 types: saving account and checking account (with possible overdraft). You can see the UML class diagram below:

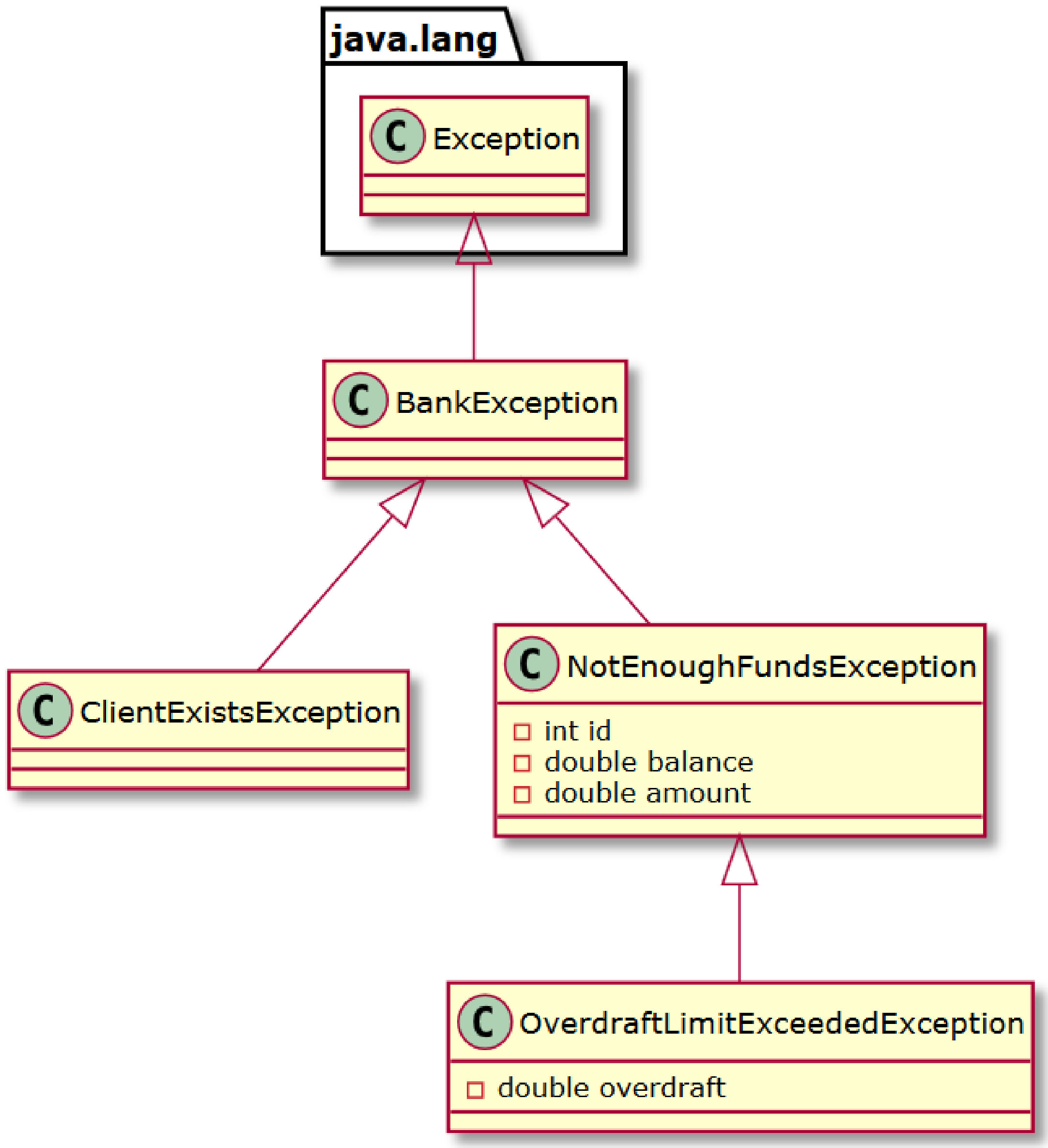


If the **`withdraw`** method requests the amount of money that exceeds the amount that can be given to the client from a **`SavingAcount`**, we throw a checked exception **`NotEnoughFundsException`**.

The **`OverdraftLimitExceededException`** is thrown by an overriden **`withdraw()`** method of the **`CheckingAccount`** class and extends **`NotEnoughFundsException`**.

If a client with the given name already exists in the bank, we throw the checked exception **`ClientExistsException`**.

There is a UML class diagram for the exceptions hierarchy:



Exercise 1
Bank Application. Using collections

Objectives

Develop skills of working with Java Collection Framework, sorting, and Generics.

Description

The current Bank Application implementation implies that a client has exactly one account and that there are maximum ten clients in the bank. Generating reports and adding other helpful features is problematic. Modify the application using Collections Framework.

Tasks

1. Modify the **`Bank`** class; store the client list using the **`java.util.Set;`** select the required interface implementation. Implement access methods **`getClients(), addClient(Client).`** Make sure that collection works properly, that is the **`equals()`** and **`hashCode()`** methods are properly overridden. Properly encapsulate the **`clients`** collections. To do that, the **`getClients()`** method must return unmodifiable client set.
2. Modify the **`Client`** class; store the account list using the **`java.util.Set`**. Perform actions similar to those described above.
3. Implement the **`BankReport`** class that contains the following methods:

`int getNumberOfClients(bank)`	Returns the number of bank clients.
`int getNumberOfAccounts(bank)`	Returns the total number of accounts for all bank clients.
`SortedSet getClientsSorted(bank)`	Displays the set of clients in alphabetical order.
`double getTotalSumInAccounts(bank)`	Returns the total sum (balance) from the accounts of all bank clients.
`SortedSet getAccountsSortedBySum(bank)`	Returns the set of all accounts. The list is ordered by current account balance.
`double getBankCreditSum(bank`)	Returns the total amount of credits granted to the bank clients. That is, the sum of all values above account balance for **`CheckingAccount`**
`Map <Client, Collection<Account>> getCustomerAccounts(bank)`	Returns a map **`Client<=>List_of_His accounts`**. This method is somehow 'artificial', because a client already has a list of his/her accounts. The aim of this step is to learn to declare complex data structures using generics and convert data
`Map<String, List<Client>>`	Add field **`city`** to class **`Client`**. This

`getClientsByCity(bank)`	method needs a table **`Map<String, List<Client>>`**, with cities as the keys and values – the list of clients in each city. Print the resulting table, and order by city name alphabetically.

4. Define all collections used with the help of Generics.
5. Implement a feature of displaying the bank statistics by calling corresponding methods of the **`BankReport`** class from the **`BankApplication`**. Define a command line argument that will launch BankApplication in special mode **`java BankApplication -statistics`** and will read the ‘display statistic’ command from the console.

Exercise 2
Bank Application. Using multithreading

Objectives

Learn the model of asynchronous data processing.

Description

In classic synchronous model, operation 2 executes only after operation 1. In case operation 1 takes a lot of time to execute, the performance decreases. In asynchronous model, data needed for operation 1 to execute is queued (queuing is a fast constant time operation); then a concurrent thread selects the data from the queue and operates with it.

Tasks

1. In Bank Application the **`EmailNotificationListener`** has been added. It sends notification emails. In general, sending a letter is a long operation. Create the **`Email`** class encapsulating the data about the added client. The **`EmailNotificationListener.onClientAdded()`** method places the **`Email`** class instance to the queue and sends mail from a concurrent thread.

Detailed guidelines

1. Create the **`Email`** class with the fields **`Client`**, **`from`**, **`to`** etc. We will not send real emails in this example
2. Create the **`EmailService`** class and define the **`sendNotificationEmail(Email)`** method. This method adds the **`Email`** class instance to a queue.
3. Create the **`Queue`** class implementing the queue on the basis of **`java.util.List`**. The **`queue.add(Email)`** method is called from the **`sendNotificationEmail(Email)`** method from **`EmailService.`**
4. The **`EmailService`** class constructor creates and starts the thread. This additional thread gets the **`Email`** objects from the queue and emulates sending the e-mail.
5. Implement this asynchronous mechanism of sending the email using **`wait-notify`**.
6. Implement the **`emailService.close()`** method that forbids further adding of messages to the queue and terminates the concurrent thread. Call this method **`emailService.close()`** from the main control thread (**`main()`**) and make sure that JVM shuts down properly.

Exercise 3
Bank Application. Using streams

Objectives

Develop skills of working with streams.

Description

The current Bank Application implements the **`BankReport`** class. You will be required to create a **`BankReportStreams`** class with the same methods and functionality, but this time using streams.

Tasks

1. Implement the **`BankReportStreams`** class that contains the following methods, this time implemented using streams:

`int getNumberOfClients(bank)`	Returns the number of bank clients.
`int getNumberOfAccounts(bank)`	Returns the total number of accounts for all bank clients.
`SortedSet getClientsSorted(bank)`	Displays the set of clients in alphabetical order.
`double getTotalSumInAccounts(bank)`	Returns the total sum (balance) from the accounts of all bank clients.
`SortedSet getAccountsSortedBySum(bank)`	Returns the set of all accounts. The list is ordered by current account balance.
`double getBankCreditSum(bank`)	Returns the total amount of credits granted to the bank clients. That is, the sum of all values above account balance for **`CheckingAccount`**
`Map <Client, Collection<Account>> getCustomerAccounts(bank)`	Returns a map **`Client<=>List_of_His accounts`**. This method is somehow ‘artificial’, because a client already has a list of his/her accounts. The aim of this step is to learn to declare complex data structures using generics and convert data
`Map<String, List<Client>> getClientsByCity(bank)`	Add field **`city`** to class **`Client`**. This method needs a table **`Map<String, List<Client>>`**, with cities as the keys and values – the list of clients in each city. Print the resulting table, and order by city name alphabetically.

2. Write tests that will check the functionality of this **`BankReportStreams`** class.