Module 01 – Piscine Java

OOP/Collections

Summary: Today you will learn how to model the operation of various collections correctly, and create a full-scale money transfer application

Contents

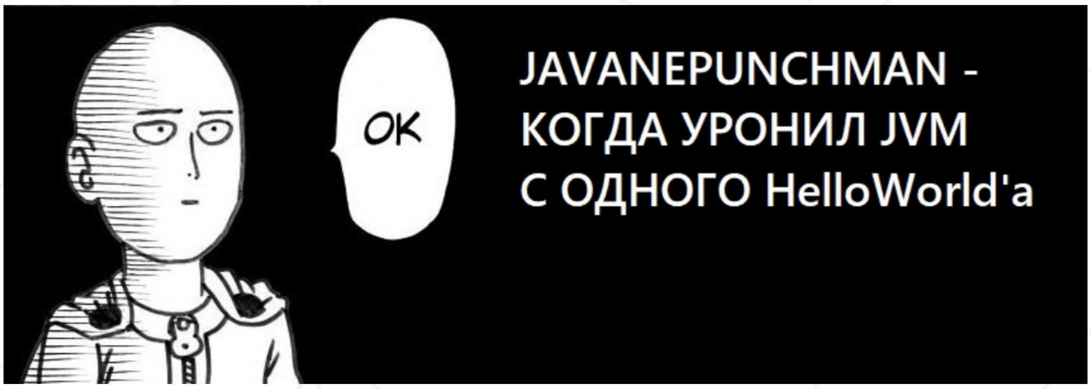
Chapter I

Foreword

Domain modeling is the most challenging task in software development. Solving this task correctly ensures flexibility of the implemented system.
Programming languages supporting the object-oriented programming (OOP) concept enable to effectively divide business processes into logical components called classes.
Each class must comply with SOLID principles:

- Single responsibility principle: a class contains a single logically associated functionality (a coffee machine cannot clean and monitor changes in the call stack; its purpose is to make coffee).
- Open-closed principle: each class can offer an option to extend its functionality. However, such extension should not provide for modifying source class code.
- Liskov substitution principle: derived classes only ADD to the functionality of a source class without modifying it.
- Interface segregation principle: there are many points (interfaces) that describe a logically associated behavior. There is no general-purpose interface.
- Dependency inversion principle: a system must not depend on specific entities; all dependencies are based on abstractions (interfaces).

Today, you should focus on the first SOLID principle.



Chapter II

Instructions

- Use this page as the only reference. Do not listen to any rumors and speculations about how to prepare your solution.
- Now there is only one Java version for you, 1.8. Make sure that compiler and interpreter of this version are installed on your machine.
- You can use IDE to write and debug the source code.
- The code is read more often than written. Read carefully the document where code formatting rules are given. When performing each task, make sure you follow the generally accepted Oracle standards
- Comments are not allowed in the source code of your solution. They make it difficult to read the code.
- Pay attention to the permissions of your files and directories.
- To be assessed, your solution must be in your GIT repository.
- Your solutions will be evaluated by your piscine mates.
- You should not leave in your directory any other file than those explicitly specified by the exercise instructions. It is recommended that you modify your .gitignore to avoid accidents.
- When you need to get precise output in your programs, it is forbidden to display a precalculated output instead of performing the exercise correctly.
- Have a question? Ask your neighbor on the right. Otherwise, try with your neighbor on the left.
- Your reference manual: mates / Internet / Google. And one more thing. There's an answer to any question you may have on Stackoverflow. Learn how to ask questions correctly.
- Read the examples carefully. They may require things that are not otherwise specified in the subject.
- And may the Force be with you!

- Never leave that till tomorrow which you can do today ;)

Chapter III

Introduction to exercises

An internal money transfer system is an integral part of many corporate applications.
Your today's task is to automate a business process associated with transfers of certain amounts between participants of our system.
Each system user can transfer a certain amount to another user. We need to make sure that even if we lose the history of incoming and outgoing transfers for a specific user, we shall still be able to recover this information.
Inside the system, all money transactions are stored in the form of debit/credit pairs. For example, John has transferred \$500 to Mike. System saves the transaction for both users:
John -> Mike, -500, OUTCOME, transaction ID
Mike -> John, +500, INCOME, transaction ID
To recover the connection within such pairs, identifiers of each transaction should be used.
A transfer entry may obviously be lost in such a complex system—it may not be recorded for one of the users (to emulate and debug such a situation, a developer needs to be able to remove the transfer data from one of users individually). Since such situations are realistic, functionality is required for displaying all "unacknowledged transfers" (transactions recorded for one user only) and resolving such issues.
Below is a set of exercises you can do one by one to solve the task.

Chapter IV

Exercise 00 : Models

	Exercise 00
Models	
Turn-in directory : *ex*00/	
Files to turn in : User.java, Transaction.java, Program.java	
Allowed functions : User classes can be employed, along with: Types (+ all methods of these types) : Integer, String, UUID, enumerations	

Your first task is to develop basic domain models—namely, User and Transaction classes. It is quite likely for different users to have the same name in the system. This problem should be solved by adding a special field for a user's unique ID. This ID can be any integer number. Specific ID creation logic is described in the next exercise.
Thus, the following set of states (fields) is typical for User class:

- Identifier
- Name
- Balance

  Transaction class describes a money transfer between two users. Here, a unique identifier should also be defined. Since the number of such transactions can be very large, let us define the identifier as an UUID string. Thus, the following set of states (fields) is typical for Transaction class:

- Identifier
- Recipient (User type)
- Sender (User type)
- Transfer category (debits, credits)
- Transfer amount

It is necessary to check the initial user balance (it cannot be negative), as well as the balance for the outgoing (negative amounts only) and incoming (positive amounts only) transactions (use of get/set methods).
An example of use of such classes shall be contained in Program file (creation, initialization, printing object content on a console). All data for class fields must be hardcoded in Program.

Chapter V

Exercise 01 : ID Generator

	Exercise 01
ID Generator	
Turn-in directory : *ex*01/	
Files to turn in : UserIdsGenerator.java, User.java, Program.java	
Allowed functions : All permissions from the previous exercise can be used	

Make sure that each user ID is unique. To do so, create UserIdsGenerator class. Behavior of the object of this class defines the functionality for generating user IDs.
State-of-the-art database management systems support autoincrement principle where each new ID is the value of the previously generated ID +1.
So, UserIdsGenerator class contains the last generated ID as its state. UserIdsGenerator behavior is defined by int generateId() method that returns a newly generated ID each time it is called.
An example of use of such classes shall be contained in Program file (creation, initialization, printing object content on a console).
Notes:

- Make sure only one UserIdsGenerator object exists (see the Singleton pattern). It is required because existence of several objects of this class cannot guarantee that all user identifiers are unique.
- User identifier must be read-only since it is initialized only once (when the object is created) and cannot be modified later during the program execution.
- Temporary logic for identifier initialization should be added to User class constructor:

```
    public User(...)  {
        this.id = UserIdsGenerator.getInstance().generateId();
}
```

Chapter VI

Exercise 02 : List of Users

	Exercise 02
List of Users	
Turn-in directory : *ex*02/	
Files to turn in : UsersList.java, UsersArrayList.java, User.java,Program.java, etc.	
Allowed functions : All permissions from the previous exercise + throw can be used.	

Now we need to implement a functionality for storing users while the program runs.
At the moment, your application has no persistent storage (such as a file system or a database). However, we want to avoid the dependence of your logic on user storage implementation method. To ensure more flexibility, let us define UsersList interface that describes the following behavior:

- Add a user
- Retrieve a user by ID
- Retrieve a user by index
- Retrieve the number of users

This interface will enable to develop the business logic of your application so that a specific storage implementation does not affect other system components.
We shall also implement UsersArrayList class that implements UsersList interface.
This class shall use an array to store user data. The default array size is 10. If the array is full, its size is increased by half. The user-adding method puts an object of User type in the first empty (vacant) cell of the array.
In case of an attempt to retrieve a user with a non-existent ID, an unchecked UserNotFoundException must be thrown.
An example of use of such classes shall be contained in Program file (creation, initialization, printing object content on a console).
Note:
Nested ArrayList<T> Java class has the same structure. By modeling behavior of this class on your own, you will learn how to use mechanisms of this standard library class.

Chapter VII

Exercise 03 : List of Users

	Exercise 03
List of Users	
Turn-in directory : *ex*03/	
Files to turn in : TransactionsList.java, TransactionsLinkedList.java, User.java, Program.java, etc.	
Allowed functions : All permissions from the previous exercise can be used	

Unlike users, a list of transactions requires a special implementation approach. Since the number of transaction creation operations can be very large, we need a storage method to avoid a costly array size extension.
In this task, we offer you to create TransactionsListinterface describing the following behavior:

- Add a transaction
- Remove a transaction by ID (in this case, UUID string identifier is used)
- Transform into array (ex. Transaction[] toArray())

  A list of transactions shall be implemented as a linked list (LinkedList) in TransactionsLinkedList class. Therefore, each transaction shall contain a field with a link to the next transaction object.
  If an attempt is made to remove a transaction with non-existent ID,
  TransactionNotFoundException runtime exception must be thrown.
  An example of use of such classes shall be contained in Program file (creation, initialization, printing object content on a console).
  Note:

- We need to add transactions field of TransactionsList type to User class so that each user can store the list of their transactions.
- A transaction must be added with a SINGLE operation (O(1))
- LinkedList<T> nested Java class has the same structure, a bidirectional linked list.

Chapter VIII

Exercise 04 : Business Logic

	Exercise 04
Business Logic	
Turn-in directory : *ex*04/	
Files to turn in : TransactionsService.java, Program.java, etc.	
Allowed functions : All permissions from the previous exercise can be used	

The business logic level of the application is located in service classes. Such classes contain basic algorithms of the system, automated processes, etc. These classes are usually designed based on the Facade pattern that can encapsulate behavior of several classes.
In this case, TransactionsService class must contain a field of UserList type for user interactions and provide the following functionality:

- Adding a user
- Retrieving a user's balance
- Performing a transfer transaction (user IDs and transfer amount are specified). In this case, two transactions of DEBIT/CREDIT types are created and added to recipient and sender. IDs of both transactions must be equal
- Retrieving transfers of a specific user (an ARRAY of transfers is returned). Removing a transaction by ID for a specific user (transaction ID and user ID are specified)
- Check validity of transactions (returns an ARRAY of unpaired transactions).

In case of an attempt to make a transfer of the amount exceeding user's residual balance, IllegalTransactionException runtime exception must be thrown.
An example of use of such classes shall be contained in Program file (creation, initialization, printing object content on a console).

Chapter IX

Exercise 05 : Menu

	Exercise 05
Menu	
Turn-in directory : *ex*05/	
Files to turn in : Menu.java, Program.java, etc.	
Allowed functions : All permissions from the previous exercise can be used, as well as try/catch	

- As a result, you should create a functioning application with a console
- menu. Menu functionality must be implemented in the respective class with a link field to TransactionsService.
- Each menu item must be accompanied by the number of the command entered by a user to call an action.
- The application shall support two launch modes—production (standard mode) and dev (where transfer information for a specific user can be removed by user ID, and a function that checks the validity of all transfers can be run).
- If an exception is thrown, a message containing information about the error shall appear, and user shall be provided an ability to enter valid data.
- The application operation scenario is as follows (the program must carefully follow this output example):

```
$ java Program --profile=dev

1. Add a user
2. View user balances
3. Perform a transfer
4. View all transactions for a specific user
5. DEV - remove a transfer by ID
6. DEV - check transfer validity
7. Finish execution
-> 1
Enter a user name and a balance
-> Jonh 777
```

```
User with id = 1 is added
---------------------------------------------------------
1. Add a user
2. View user balances
3. Perform a transfer
4. View all transactions for a specific user
5. DEV - remove a transfer by ID
6. DEV - check transfer validity
7. Finish execution
-> 1
Enter a user name and a balance
-> Mike 100
User with id = 2 is added
---------------------------------------------------------
1. Add a user
2. View user balances
3. Perform a transfer
4. View all transactions for a specific user
5. DEV - remove a transfer by ID
6. DEV - check transfer validity
7. Finish execution
-> 3
Enter a sender ID, a recipient ID, and a transfer amount
-> 1 2 100
The transfer is completed
---------------------------------------------------------
1. Add a user
2. View user balances
3. Perform a transfer
4. View all transactions for a specific user
5. DEV - remove a transfer by ID
6. DEV - check transfer validity
7. Finish execution
-> 3
Enter a sender ID, a recipient ID, and a transfer amount
-> 1 2 150
The transfer is completed
---------------------------------------------------------
1. Add a user
2. View user balances
3. Perform a transfer
4. View all transactions for a specific user
5. DEV - remove a transfer by ID
6. DEV - check transfer validity
7. Finish execution
-> 3
Enter a sender ID, a recipient ID, and a transfer amount
-> 1 2 50
The transfer is completed
---------------------------------------------------------
1. Add a user
2. View user balances
3. Perform a transfer
4. View all transactions for a specific user
5. DEV - remove a transfer by ID
6. DEV - check transfer validity
7. Finish execution
-> 2
Enter a user ID
-> 2
Mike - 400
---------------------------------------------------------
1. Add a user
2. View user balances
3. Perform a transfer
4. View all transactions for a specific user
5. DEV - remove a transfer by ID
6. DEV - check transfer validity
7. Finish execution
-> 4
Enter a user ID
-> 1
To Mike(id = 2) -100 with id = cc128842-2e5c-4cca-a44c-7829f53fc31f
To Mike(id = 2) -150 with id = 1fc852e7-914f-4bfd-913d-0313aab1ed99
TO Mike(id = 2) -50 with id = ce183f49-5be9-4513-bd05-8bd82214eaba
---------------------------------------------------------
1. Add a user
2. View user balances
3. Perform a transfer
4. View all transactions for a specific user
5. DEV - remove a transfer by ID
6. DEV - check transfer validity
7. Finish execution
-> 5
Enter a user ID and a transfer ID
-> 1 1fc852e7-914f-4bfd-913d-0313aab1ed99
Transfer To Mike(id = 2) 150 removed
---------------------------------------------------------
1. Add a user
2. View user balances
3. Perform a transfer
4. View all transactions for a specific user
5. DEV - remove a transfer by ID
6. DEV - check transfer validity
7. Finish execution
-> 6
Check results:
Mike(id = 2) has an unacknowledged transfer id = 1fc852e7-914f-4bfd-913d-0313aab1ed99 from John(id = 1) for 150
```