|  |  |
| --- | --- |
| **Web programming and advanced development techniques** | 2. JAVA exercise |

Create a package named „bank”, and then make the following classes!

Create a „Person” class, in which there are:

* *name* string field
* *name* read-only property
* constructor, which sets the name variable’s value

Create an „Account” class, in which there are:

* *owner* string field
* *accountNumber* string field
* *balance* signed integer field
* *accountNumber* and *owner* are read-only properties
* *accountNumbercounter* is an integer field, which is static and starts from 0.
* constructor, which gets a Person as the only parameter. This object is the owner of the Account. The balance is 0, and the accountNumber is a unique number starting from 1 (use the static field). The format of the string accountNumber should be „00001”, „00002”, etc.
* *deposit* method, which increases the balance with the given parameter number.
* *withdrawal* method, which decreases the balance with the given parameter. In case the amount to withdraw is greater than what is on the balance, do not go on with the method logic, but throw an *InsufficientFoundsException*.
  + This exception class should be in the bank.exceptions package
  + This exception must be a checked exception

Create a Unit Test called *AccountTest*, which

* Tests whether the deposit function works,
* Tests whether the withdrawal function works in case of sufficient and insufficient funds.

Create the *Bank* class, which should contain:

* *Accounts* field which assigns „accountNumber”s to real Account objects. (You are supposed to figure out what collection or data structure to use for this)
* *createAccount* method, which takes a Person object as a parameter, and creates a new Account with the given Person, and stores it in the „Accounts” field, and then returns the accountNumber of the newly created account.
* *transfer* method, with the parameters: Person who starts the transfer, source accountNumber in a string, destination accountNumber in a string, amount in an integer.  
  The method should verify the following:
  + source accountNumber exists
  + destination accountNumber exists
  + the owner of the source accountNumber is the one who wants to transfer funds
  + Sufficient funds are available on the source account (we only know this if the withdrawal method of the source account has not thrown an exception)
* In case of any verification in the transfer function fails, you should throw a TransferException. For this, create a BankException class, from which you can read out the Bank object (it is given to it in its constructor). The TransferException should be a descendant for this, with the added extra that you can also give an exception message string in its constructor, in which you can describe which one of the 4 verifications failed.
* Create a unit test for testing the success and failure of all 4 verifications.

Create a *DirectDebit* (Átutalási megbízás in hungarian) class, in which

* *client* field, which holds a *Person* object.
* *sourceAccountNumber* string field.
* *destinationAccountNumber* string field.
* *amount* integer field
* constructor setting these
* properties for reading and writing these fields

Extend the *Bank* class with the following:

* *executeDirectDebit* method, which takes in a DirectDebit as a parameter, and attempts to execute it with the *transfer* method.
* *unsuccessfulDirectDebit* dinamic array (ArrayList). The *executeDirectDebit* method stores those DirectDebits here which could not be executed at the time of calling (an exception was thrown at th time of calling).
* *adHocAgreements* fields with a queue type (Queue/LinkedList) which can store DirectDebits.
* *createAdHocAgreement* method to enqueue a new DirectDebit. Parameters should be: client as a Person object, source and destination account numbers, and the amount. There should be no verifications when enqueuing!)
* *dailyBusiness* method, which takes out a DirectDebit a time from the queue one by one, and attempts to execute them.

Create a *FixedDirectDebit* class, which

* should be the descendant of the DirectDebit class,
* and extends it with a „priority” field which is a floating point number, and gets its value in the constructor
* implement the Comparable interface (built-in, have to import it), and implement its *compareTo* method so that, in case of ordering, the FixedDirectDebits should be sorted according to priority in a descending order (highest priority first).

Extend the *Bank* class with the following:

* *fixedDirectDebits* named ordered set (TreeSet), in which you can store the fixedDirectDebits.
* *addFixedDirectDebit* method, which adds to the ordered set a new FixedDirectDebit with the given parameters: client, source, destination, amount, priority.
* the *dailyBusiness* method executes the FixedDirectDebits, too. (in the order of priority)
* *listUnsuccessfulDirectDebit* which writes out all unsuccessfuls in the ArrayList into a file, and then clears the ArrayList:

F/A (fixed or adhoc) \t client name \t source accNum \t destination accNum \t amount \n

* modify the *dailyBusiness* method so that it calls this listUnsuccessful…, and it returns the exact same string written into the file.
* Create a unit test to test these improvements.