Funktionale und objektorientierte Programmierkonzepte
Übungsblatt 08

Prof. Karsten Weihe

Übungsblattbetreuer:	Marlon Friedl und Nhan Huynh
Wintersemester 23/24	v2.0.1
Themen:	assert und Exceptions
Relevante Foliensätze:	05 (und natürlich auch weiterhin 01*-04*)
Abgabe der Hausübung:	05.01.2024 bis 23:50 Uhr

Hausübung 08 — Gesamt: 32 Punkte

The Exceptionals - State of Emergency

Beachten Sie die Seite *Verbindliche Anforderungen für alle Abgaben* im Moodle-Kurs.

Verstöße gegen verbindliche Anforderungen führen zu Punktabzügen und können die korrekte Bewertung Ihrer Abgabe beeinflussen. Sofern vorhanden, müssen die in der Vorlage mit `TODO` markierten `crash`-Aufrufe entfernt werden. Andernfalls wird die jeweilige Aufgabe nicht bewertet.

Die für diese Hausübung relevanten Verzeichnisse sind `src/main/java/h08` und ggf. `src/test/java/h08`.

Einleitung

Das Übungsblatt widmet sich dem Thema Fehlerbehandlung, insbesondere im Kontext von Bankanwendungen. In einer Software, die mit sensiblen Finanzdaten arbeitet, ist eine effektive Fehlerbehandlung von entscheidender Bedeutung, um die Integrität der Daten und die Zuverlässigkeit der Anwendung sicherzustellen.

Daher modellieren wir in dieser Hausübung vereinfachte Formen verschiedener Entitäten einer Bank und implementieren die Interaktionen zwischen diesen Entitäten.

- Ein *Kunde* (Record `Customer`) besitzt einen Namen, eine Adresse und ein Geburtsdatum.
- Ein *Account* (Klasse `Account`) repräsentiert ein Kundenkonto bei einer Bank und enthält Informationen wie den Kunden, die IBAN[1], das aktuelle Guthaben, die Bank, bei der das Konto geführt wird, und den Transaktionsverlauf.
- Der *Status* einer Transaktion (Enum `Status`) weist den Status einer Transaktion zu, darunter `OPEN`, `CLOSED` und `CANCELLED`.
- Eine Transaktion (Record `Transaction`) repräsentiert eine Transaktion zwischen zwei Kunden und enthält Informationen wie die beteiligten Kunden, die Transaktionsnummer, den Betrag, eine Beschreibung, das Datum und den Status der Transaktion.
- Der *Transaktionsverlauf* (Klasse `TransactionHistory`) verwaltet die Transaktionen eines Kunden, zeigt die letzten n Transaktionen an, sortiert nach dem Tag, an dem die Transaktion stattgefunden hat.
- Die *Bank* (Klasse `Bank`) verwaltet die Kundenkonten und die Interaktionen zwischen Kunden und Bank.

Für ein besseres Verständnis können Sie sich die Vorlage anschauen.

Die Hausübung konzentriert sich besonders auf die Fehlerbehandlung durch *assert*-Anweisungen, *Exceptions* bzw. *RuntimeExceptions* sowie die Verwendung von `try-catch`- Blöcken.

> **Hinweis:**
>
> 1. Beachten Sie beim Testen Ihrer `assert`-Anweisungen, dass diese standardmäßig beim Ausführen ignoriert werden. Um `assert`-Anweisungen zu aktivieren, müssen Sie die JVM mit dem Parameter `-ea` starten.
>
>   Auf diesem Übungsblatt können Sie die `assert`-Anweisungen aktivieren, indem Sie die Gradle-Aufgabe `application/run` ausführen.
> 2. Die Dokumentation in diesem Übungsblatt ist nicht obligatorisch.

[1] `https://de.wikipedia.org/wiki/Internationale_Bankkontonummer`

H1: Ich möchte ein Kunde sein! 1 Punkt

Wir modellieren einen Kunden als Record `Customer`, der einen Namen, eine Adresse und ein Geburtsdatum besitzt.

Gewährleisten Sie die Daten des Kunden mittels `assert`-Anweisungen, dass diese nicht `null` sind.

H2: Bank und Kunde

Die Klasse `h08.Bank` repräsentiert eine Bank und definiert die Interaktionen zwischen Bank und Kunde. In dieser Aufgabe implementieren wir einige Interaktionen zwischen Bank und Kunde.

Die Bank verwaltet die Kundenaccounts in einem Array `accounts`, wobei nicht alle Felder belegt sein müssen. Die Anzahl der belegten Felder wird in `size` gespeichert.

H2.1: Ist die IBAN schon in Verwendung? 2 Punkte

Bevor wir ein Kundenkonto anlegen können, müssen wir wissen, welche IBAN noch nicht vergeben ist. Dafür implementieren wir die Methode `isIbanAlreadyUsed(long)`, die als Parameter eine IBAN erhält und einen booleschen Wert zurückgibt. Dieser gibt genau dann `true` zurück, falls die IBAN bereits vergeben ist.

H2.2: Ich möchte gerne einen Konto eröffnen! 3 Punkte

Nun können wir mit der Erstellung des Kundenkontos beginnen. Hierfür benötigen wir die Methode `generateIban(Customer, long)`, die einen Kunden und einen Startwert für die Berechnung der IBAN als Parameter erhält und eine IBAN generiert.

Die IBAN wird nach folgender Formel berechnet:

$$\text{IBAN} = |\text{hashcode(Kunde)} \cdot \text{Startwert}|$$

Falls die generierte IBAN bereits vergeben ist, wird eine neue IBAN generiert, wobei die aktuelle generierte IBAN als Startwert verwendet wird. Dieser Vorgang wird solange wiederholt, bis eine IBAN gefunden wird, die noch nicht vergeben ist.

Mit der generierten IBAN können wir nun ein Kundenkonto anlegen, was mittels der Methode `add(Customer)` erfolgt. Diese Methode erstellt für den Kunden ein neues Kundenkonto (`Account`) an der nächstfreien Position im Array `accounts`. Die IBAN des Kundenkontos wird mithilfe der Methode `generateIban(Customer, long)` generiert, wobei der Startwert das Ergebnis von `java.lang.System#nanoTime()` ist. Das neue Konto hat kein Startguthaben und eine leere `TransactionHistory`, die nur die letzten `transactionHistoryCapacity` Transaktionen speichert.

Falls das Array `accounts` voll ist, wird eine `IllegalStateException` mit der Nachricht `"Bank is full"` geworfen.

> **Verbindliche Anforderung:**
>
> Die Anzahl der Kundenkonten wird in `size` gespeichert. Vergessen Sie nicht, die Anzahl zu erhöhen, wenn ein Kundenkonto hinzugefügt wird.

Hinweis:

Jede Klasse, die von `Object` abgeleitet ist, besitzt eine Methode namens `java.lang.Object#hashCode()`, die einen `int` zurückgibt und den zugehörigen Hashwert des Objekts repräsentiert. Zur Berechnung des Betrags können Sie die Methode `java.lang.Math#abs()` verwenden.

H2.3: Kundenkonto entfernen — 2 Punkte

Es sollte möglich sein, ein Kundenkonto zu entfernen. Dafür implementieren wir die Methode `remove(long)`, die als Parameter eine IBAN erhält und das Kundenkonto mit der IBAN aus dem Array `accounts` entfernt. Falls die IBAN nicht gefunden wird, wird eine `NoSuchElementException` mit der IBAN als Nachricht geworfen. Zusätzlich überprüfen wir mittels `assert`, dass die IBAN positiv ist. Die Rückgabe der Methode entspricht dem entfernten Kundenkonto.

Verbindliche Anforderungen:

(i) Das Array soll lückenlos sein, d.h. es dürfen keine leeren Felder zwischen zwei belegten Feldern existieren.

(ii) Die Anzahl der Kundenkonten wird in `size` gespeichert. Vergessen Sie nicht, die Anzahl zu verringern, wenn ein Kundenkonto entfernt wird.

H2.4: Geld abheben und einzahlen — 2 Punkte

Die Bank ermöglicht Kunden Einzahlungen und Auszahlungen. Hierfür dienen die Methoden `withdraw(long, double)` und `deposit(long, double)`, welche eine IBAN und einen Betrag als Parameter erhalten.

Falls der Betrag negativ (inkl. 0) ist, wird eine `IllegalArgumentException` mit dem Betrag als Nachricht geworfen. Sollte die IBAN nicht gefunden werden, erfolgt eine `java.util.NoSuchElementException` mit der entsprechenden IBAN als Nachricht.

Ansonsten wird der Betrag vom Kundenkonto abgezogen oder diesem hinzugefügt. Es ist zu beachten, dass ein Kunde nicht mehr Geld abheben kann, als auf seinem Konto vorhanden ist. Falls der neue Kontostand negativ wäre, führt dies zu einer `IllegalArgumentException` mit dem negativen Betrag als Nachricht.

H3: Eigene Fehlermeldungen — 4 Punkte

Bisher haben wir nur vordefinierte Exceptions in Java verwendet. Um jedoch aussagekräftige Fehlermeldungen zu generieren, können wir eigene Exceptions definieren, die klar und präzise beschreiben, was schief gelaufen ist.

Folgende Exceptions sollen definiert werden:

(a) `BadTimestampException`: Diese Exception leitet von `RuntimeException` ab und signalisiert, dass ein ungültiges Datum angegeben wurde. Der Konstruktor dieser Exception nimmt das Datum als `java.time.LocalDate` als Parameter und wirft eine Exception mit der Nachricht `"Bad timestamp: "` konkateniert mit dem Datum als Nachricht.

(b) `BankException`: Diese Exception leitet von `Exception` ab und gibt an, dass ein Fehler bei der Bank aufgetreten ist. Die Klasse hat zwei Konstruktoren: einen mit der Nachricht als Parameter und einen weiteren mit der BIC (`long`) als Parameter. Beide Parameter dienen als Nachrichten, wobei die BIC mit dem Prefix `"Cannot find Bank with BIC: "` konkateniert wird.

(c) `TransactionException`: Diese Exception leitet von `Exception` ab und hat zwei Konstruktoren. Der erste Konstruktor nimmt die Nachricht und die Transaktionsnummer (`long`) als Parameter. Der zweite Konstruktor hat

ein Array von Transaktionen (`Transaction[]`) als Parameter. Der erste Konstruktor initialisiert die Nachricht durch die Konkatenation der Nachricht und der Transaktionsnummer, wobei beide durch ein Leerzeichen getrennt sind. Der zweite Konstruktor initialisiert die Nachricht durch die Konkatenation des Strings `"Transaction numbers: ["` gefolgt von den Transaktionsnummern, die jeweils durch ein Komma getrennt sind, und endet schließlich mit dem String `"]"`.

> **Hinweis:**
>
> Der Aufruf des Konstruktors aus der Basisklasse muss immer an erster Stelle erfolgen. Da wir beim zweiten Konstruktor nicht die Transaktionen selbst benötigen, sondern nur die Transaktionsnummern, müssen wir das Array auf eine andere Art abbilden Leider können wir keine Anweisungen vor dem Aufruf des Konstruktors aus der Basisklasse schreiben. Welche Möglichkeiten gibt es, das Array auf eine andere Art abzubilden?

H4: Validierung von Daten

Durch die Verwendung eigener Exceptions aus Aufgabe H3 können wir nun Daten validieren und aussagekräftige Fehlermeldungen generieren.

> **Hinweis:**
>
> Für diese Aufgabe benötigen Sie die Java-Klasse `java.time.LocalDate`. Zum besseren Verständnis können Sie sich die Dokumentation ansehen. Insbesondere sind Methoden zur Bestimmung des aktuellen Datums und zum Vergleichen zweier Daten hilfreich.

H4.1: Kundenvalidierung — 1 Punkt

Bisher haben wir kein Mindestalter für Kundenkonten (`Account`) festgelegt. Ein Kunde kann nur dann ein Kundenkonto eröffnen, wenn er mindestens 18 Jahre alt ist. Falls dies nicht der Fall ist, wird eine `BadTimestampException` mit dem Geburtsdatum als Nachricht geworfen.

Stellen Sie außerdem mittels `assert`-Anweisungen sicher, dass der Kunde, die Bank und die Transaktionshistorie nicht `null` sind. Die IBAN des Kundenkontos muss ebenfalls positiv sein.

H4.2: Transaktionsvalidierung — 1 Punkt

Eine Transaktion darf nicht in der Zukunft erstellt werden. Falls dies dennoch geschieht, wird eine `BadTimestampException` mit dem Datum als Nachricht geworfen, sofern der Tag in der Zukunft liegt.

Stellen Sie außerdem mittels `assert`-Anweisungen sicher, dass der Sender, der Empfänger, die Beschreibung, das Datum und der Status nicht `null` sind.

H5: Überweisungen

Zum Schluss implementieren wir die Überweisungen zwischen zwei Kundenkonten.

H5.1: Transaktionen in die Historie aufnehmen — 4 Punkte

Bei einer Überweisung wird eine Transaktion zwischen zwei Kundenkonten erstellt und in die Transaktionshistorie aufgenommen. Die Transaktionshistorie ermöglicht es einem Kunden, die letzten n Transaktionen zu überblicken.

Um dies zu realisieren, implementieren Sie die Methode `add(Transaction)` in der Klasse `TransactionHistory`. Diese Methode erhält eine Transaktion als Parameter und fügt sie der Transaktionshistorie hinzu.

Die Transaktion wird nur aufgenommen, wenn die Transaktionsnummer noch nicht existiert. Andernfalls wird eine `IllegalArgumentException` mit der Nachricht `"This transaction already exists!"` geworfen.

Des Weiteren werden nur die letzten n Transaktionen gespeichert, wobei n die Kapazität der Transaktionshistorie ist. Wenn die Kapazität überschritten wird, erfolgt die Überschreibung der ältesten Transaktion.

Verbindliche Anforderung:

Die Anzahl der Transaktionen in der Historie wird in `size` gespeichert. Vergessen Sie nicht, die Anzahl zu erhöhen, wenn eine Transaktion hinzugefügt wird.

Ebenfalls gibt `nextIndex` die nächstfreie Position im Array `transactions` an. Vergessen Sie hier ebenfalls nicht, die nächstfreie Position zu aktualisieren, wenn eine Transaktion hinzugefügt.

H5.2: Transkation aktualisieren — 2 Punkte

Transaktionen, die im Verlauf der Zeit bearbeitet werden, müssen aktualisiert werden. Hierfür wird die Methode `update(Transaction)` der Klasse `TransactionHistory` verwendet. Diese Methode erhält als Parameter eine Transaktion und aktualisiert die betroffene Transaktion im Array `transactions`. Falls die Transaktion nicht gefunden wird, wird eine `TransactionException` mit der Nachricht `"Transaction does not exist!"` und der Transaktionsnummer als Parameter geworfen.

H5.3: Überweisung tätigen — 5 Punkte

Ein Kunde kann Geld von seinem Kundenkonto auf ein anderes Kundenkonto überweisen. Um dies zu ermöglichen, implementieren wir die Methode `transfer(long, long, int, double, String)` in der Klasse `Bank`, die als Parameter die IBAN des Senders und des Empfängers, die BIC der Bank des Empfängers, den Betrag und die Beschreibung erhält.

Die Methode ist folgendermaßen aufgebaut:

(1) Bestimme den Sender und Empfänger anhand der IBAN und die Bank des Empfängers. Falls der Sender oder Empfänger nicht gefunden werden kann oder die Überweisung von der Bank des Empfängers nicht unterstützt wird, terminiert die Methode mit dem Status `CANCELLED`.

(2) Erstelle eine Transaktion mit den Daten des Senders und Empfängers, dem Betrag und der Beschreibung. Als Datum wird das aktuelle Datum verwendet und der Status wird auf `OPEN` gesetzt. Für die Transaktionsnummer verwenden Sie bitte die Methode `generateTransactionNumber()`

(3) Füge die Transaktion in die Transaktionshistorie des Senders und Empfängers hinzu.

(4) Buche den Betrag vom Sender ab und füge den Betrag dem Empfänger hinzu und aktualisiere den Status der Transkation in der Historie, sofern dies fehlerfrei möglich ist. Falls ein Fehler bei der Buchung auftritt, wird die Transaktion mit dem Status `CANCELLED` abgebrochen, die jeweils auch in der Historie enthalten sein muss.

(5) Falls die Transaktion fehlerfrei durchgeführt werden kann, wird die Transaktion mit dem Status der aktuellen Transaktion abgeschlossen.

Hinweise:

1. Das aktuelle Datum kann mittels der Methode `java.time.LocalDate#now()` erhalten werden.
2. Die Transaktionsnummer können Sie mit der Methode `generateTransactionNumber()` in der Klasse `Bank` generieren.

H5.4: Offene Überweisungen — 5 Punkte

Die offenen Überweisungen müssen von der Bank bearbeitet werden. Daher implementieren wir die Methode `checkOpenTransactions()`, die **alle** offenen Überweisungen überprüft und diese Transaktionen als Array von Transaktionen (`Transcation[]`) zurückgibt. (Alle offenen Transaktionen zu Beginn werden zurückgegeben)

Falls es Transaktionen gibt, die älter als zwei Wochen sind, wird die alte Transaktion mit dem Status `CANCELLED` abgebrochen, und die Transaktion wird erneut durchgeführt (neue Überweisung). Das bedeutet, dass sowohl die alte als auch die neue Transaktion vorhanden sind, wenn die Überweisung erfolgreich ist.

Wenn es Transaktionen gibt, die älter als vier Wochen sind, wird die alte Transaktion mit dem Status `CANCELLED` abgebrochen, und die Transaktion wird nicht erneut durchgeführt. Stattdessen werden die abgebrochenen Transkationen in einem Array gespeichert und später als `TransactionException` geworfen, sofern es Transaktionen gibt, die älter als vier Wochen sind.

Hinweis:

Bestimmen Sie zuerst die Größe des Arrays, bevor Sie diese befüllen.

H6: Testen

Wir wollen nun einen Teil unserer Bank auf die Probe stellen. Dazu gehen wir die Schritte zu einer erfolgreichen und einer fehlerhaften Transaktion durch. Gerade bei dieser Hausübung ist es sehr wichtig, dass Sie sich eigene Testfälle ausdenken und so herausfinden, ob Programmabbrüche richtig behandelt werden. Gehen Sie zu Beginn in `Main`-Klasse:

(1) Erstellen Sie zwei Kunden(`Customer`). Keines der Felder soll `null` sein! Verwenden sie zudem gültige Geburtsdaten. Diese können Sie mithilfe von `java.time.LocalDate#of()` verwenden.

(2) Erstellen Sie als nächstes zwei verschiedene Banken (`Bank`). Die erste Bank erhält als BIC die 42, die zweite 123.

(3) Fügen Sie nun mittels `add()`-Methode von der Klasse `Bank` sowohl die Kunden als auch die jeweils andere Bank in das Bankensystem ein (es gibt zwei verschiedene `add()` Methoden). Achten Sie auch darauf, die Kunden in verschiedene Banken einzufügen, damit der restliche Test problemlos funktioniert.

(4) Zur besseren Übersicht erstellen wir nun eine Referenz vom Typ `Account`, die auf `getAccounts()[0]` der jeweiligen Banken verweisen. Das Endergebnis soll also eine Referenz zu den Accounts der beiden Kunden sein.

(5) Um die Überweisungen tätigen zu können, brauchen die Accounts Guthaben. Fügen Sie den Accounts nun mit `deposit()` ausreichend Guthaben hinzu. Für den weiteren Verlauf sind 20000 ausreichend.

(6) Nun möchten wir einen Aufruf von `transfer()` tätigen, der eine erfolgreiche Transaktion simuliert. Dazu ruft die erste Bank `transfer()` auf und transferiert 1000 vom ersten zum zweiten Account.

(7) Nun überprüfen wir, ob die Kontostände der jeweiligen Accounts. Diese sollten nun 21000 und 19000 betragen. Hierfür können Sie die Klassenmethode `testWithinRange()` der Klasse `StudentTestUtils` verwenden. Als Grenzen verwenden Sie [erwarteter Wert - ϵ, erwarteter Wert + ϵ]. Das epsilon ist bereits in der Klasse `Main` definiert. Überprüfen Sie auch die `TransactionHistory` der beiden Accounts mittels der Klassenmethode `testEquals()` der Klasse `StudentTestUtils`.

(8) Nun wollen wir eine fehlerhafte Transaktion simulieren. Übergeben Sie dazu eine nicht vorhandene BIC. Nun sollte die Methode `CANCELLED` zurückgeben, und es sollte keine Transaktion angelegt worden sein. Auch hier können Sie wieder `testEquals()` verwenden.

(9) Zuletzt wollen wir noch überprüfen, ob eine Exception geworfen wird, wenn ein Kontostand negativ wird. Wir verwenden hier `testThrows()` aus `StudentTestUtils`. Als erstes Argument übergeben wir die erwartete Exceptionklasse, als zweites Argument rufen wir `withdraw(long, double)` mit einem Betrag auf, der größer als der Kontostand ist. Sie können hier einen beliebigen Konto und einen beliebigen Betrag verwenden. Beachten Sie, dass das zweite Argument ein Lambda-Ausdruck ist. Die funktionale parameterlose und rückgabewertlose Methode `execute()` führt einen Block von Anweisungen aus und wirft potenziell eine Exception.