Übung 2: Concurrent Bank

In dieser Übung sollen die Funktionen einer Bank threadsicher implementiert werden. Kunden sollen in der Lage sein, neue Konten bei einer Bank zu eröffnen. Jedes Konto hat eine Kontonummer und einen Kontoinhaber. Auf Konten können Geldbeträge einbezahlt und abgehoben werden, das Konto darf jedoch nicht überzogen werden. Zusätzlich können Geldbeträge von einem Konto auf ein anderes transferiert werden. Ein Konto darf nur gelöscht werden, wenn es saldiert worden ist, d.h. wenn der Saldo 0.00 ist. Wird ein Konto gelöscht, so wird es als passiv markiert (ein Klient könnte ja noch eine Referenz darauf besitzen). Auf passiven Konten dürfen keine Transaktionen mehr ausgeführt werden. Nummer, Name und Saldo kann jedoch auch auf einem passiven Konto abgefragt werden (der Saldo ist dabei immer 0).

Damit Sie sich voll auf den Aspekt Concurrency konzentrieren können haben wir Ihnen ein Benutzerinterface bereitgestellt. Dieses Programm verwendet folgende Interfaces um auf die Funktionalitäten der Bank zuzugreifen. Es ist geplant diesen Programmrahmen in nachfolgenden Übungen ebenfalls einzusetzen, z.B. um die Funktionalität mit Scala STM zu realisieren. Bei allen Methoden ist zusätzlich die Ausnahme *java.io.IOException* deklariert; diese wird geworfen wenn bei der Kommunikation (über Prozessgrenzen hinweg, nicht relevant in dieser Aufgabe) Probleme auftreten.

```
package bank;

public interface Bank {
    public String createAccount(String owner);          // creates a new account and returns the account number
    public boolean closeAccount(String number);         // closes the specified account and returns success of closing
                                                        //          and passivates the account
    public Set<String> getAccountNumbers();             // returns a set of the account numbers of all currently active accounts
    public Account getAccount(String number);           // returns the account with the given number

    public void transfer(Account a, Account b, double amount)
                    throws IllegalArgumentException, InactiveException, OverdrawException;
}

public interface Account {
    public String getNumber();                          // returns the account number
    public String getOwner();                           // returns the account owner
    public boolean isActive();                          // returns the state of the account.

    public void deposit (double amount) throws IllegalArgumentException , InactiveException;
    public void withdraw(double amount) throws IllegalArgumentException , InactiveException, OverdrawException;
    public double getBalance();
}

public class  OverdrawException extends Exception {
    public OverdrawException (String reason){ super(reason); }
}

public class  InactiveException extends Exception {
    public InactiveException (String reason){ super(reason); }
}
```

Um auf die Implementierung der Bank zugreifen zu können wird das Interface *BankDriver* definiert. Das Klientenprogramm verwendet dieses Interface, um die Verbindung zum Server auf- bzw. abzubauen.

```
package bank;

public interface BankDriver {
    public void connect(String[] args);                 // establish connection to the (remote) bank
    public void disconnect();                           // close connection to the (remote) bank
    public Bank getBank();
}
```

Beim Start des Klientenprogramms muss als erster Parameter eine Klasse angegeben werden, welche das Interface *BankDriver* implementiert. Das Klientenprogramm ladet dann diese Klasse und ruft über dieses Objekt die Methode *connect* auf und übergibt als Parameter die restlichen Kommandozeilenargumente als String-Array.

Im Beispiel

```
java bank.Client bank.local.ConprBankDriver
```

lädt das Klientenprogramm die Klasse *bank.local.ConprBankDriver* und ruft in dieser Klasse die Methode *connect* auf und übergibt als Parameter einen leeren String-Array.

Um Ihnen die Situation zu vereinfachen, haben wir Ihnen die die Klasse *bank.local.ConprBankLaucher* bereitgestellt. Sie enthält eine Main Methode die den *bank.local.ConprBankDriver* lädt. Darin finden sie eine **nicht threadsichere** Implementierung der Bank. Verwenden Sie gleich diese als Ausgangspunkt für die folgende Aufgabe.

Aufgabe:

In dieser Übung sollen Sie die Bank mit Java implementieren. Die Bank wird dabei in derselben JVM laufen wie das GUI. Die Klasse welche das Interface *BankDriver* implementiert ist daher trivial. Die Methode *connect* erzeugt eine neue Instanz der Bank-Implementierung, die von der Methode *getBank* zurückgegeben wird. In der Methode *disconnect* wird diese Referenz wieder zurückgesetzt.

```
package bank.local;

public class ConprBankDriver implements bank.BankDriver {
  private ConprBank bank = null;

  @Override
  public void connect(String[] args) {
    bank = new ConprBank();
  }

  @Override
  public void disconnect() {
    bank = null;
  }

  @Override
  public bank.Bank getBank() {
    return bank;
  }
}
```

Ihre Bank soll jedoch bereits so programmiert sein, dass eine Reihe von parallelen Threads gleichzeitig Operationen auf den vorhandenen Kontos durchführen können. Beachten Sie dabei, dass die Threads präemptiv sein können, das heisst, dass Sie jederzeit mit dem Entzug des Prozessors rechnen müssen.

Beim der *transfer*-Operation soll es möglich sein, dass parallel zwei Transfers ausgeführt werden können solange unterschiedliche Konten betroffen sind, d.h. die Methode *transfer* soll nicht als *synchronized* deklariert werden, denn dadurch erzwingt man, dass jeweils nur ein Transfer ausgeführt wird -- das würde einer Bank entsprechen, welche jeweils nur einen Kunden aufs mal in die Schalterhalle einlassen würde.

Im Klientenprogramm steht ein Menu *Test* zur Verfügung. Der Test *Threading Test* führt Transaktionen auf der Bank aus und prüft, ob diese thread-safe sind. Achtung: Sie müssen diesen Test auf mehreren Maschinen ausprobieren, insbesondere auf einer mit mehreren Prozessoren. Der Test *Transfer Test* prüft zudem, ob die transfer-Operation richtig implementiert ist. Falls diese Operation nicht terminiert dann steckt die Applikation in einem Deadlock fest.

Die Dateien zur Bank stehen auf dem AD unter conpr\02_Locks\Assignment\02_AS_Bank.zip zur Verfügung.