# Grundlagen der Programmkonstruktion
# Übungsblatt 9

## 1 Zusicherungen und Termination (1)

Gegeben ist die folgende Methode `year(int days)`

```
public static boolean isLeapYear(int year) {
    // Gibt true zurueck, wenn year ein Schaltjahr ist,
    // ansonsten false.
}

public static int year (int days) {
    // Gibt die aktuelle Jahreszahl zurueck, wenn
    // seit dem 1.1.1980 "days" Tage vergangen sind.
    assert days >= 0;

    int year = 1980;

    while (days > 365) {
        if (IsLeapYear(year)) {
            if (days > 366) {
                days -= 366;
                year += 1;
            }
        } else {
            days -= 365;
            year += 1;
        }
    }

    return year;
}
```

### 1.1 Zusicherung (0.4)

Finden Sie für alle lokalen Variablen Schleifeninvarianten, Vorbedingungen und Nachbedingungen, welche die möglichen Wertebereiche so weit wie Möglich eingrenzen.

### 1.2 Termination I (0.2)

Die Methode `years(int days)` ist fehlerhaft: Für bestimmte Eingaben terminiert die `while`-Schleife nicht. Finden Sie den Fehlerfall und geben Sie ein Beispiel an.

### 1.3 Termination II (0.4)

Korrigieren Sie die `while`-Schleife, sodass sie immer terminiert und ein korrektes Ergebnis zurückliefert. Achten Sie dabei darauf, dass auch im vorigen Fehlerfall eine korrekte Jahreszahl zurückgegeben werden soll.

## 2 Dateien lesen und schreiben (3)

Achten Sie in den folgenden Aufgaben darauf, dass Sie Exceptions korrekt abfangen.

### 2.1 Anzahl der Zeilen in einer Datei bestimmen (1.2)

Schreiben Sie eine Methode `int countLines(String filename)`, die als Parameter einen Dateinamen erhält, die Datei mit diesem Namen öffnet und anschließend die Zeilen dieser Datei zählt. Benutzen Sie dazu die Methode `BufferedReader.readLine()`. Sollte ein Fehler auftreten, so geben Sie eine Fehlermeldung aus, die auf die Ursache des Fehlers schließen lässt (z.B. "Datei nicht gefunden") und beenden Sie die Methode mit einem Rückgabewert von -1.

### 2.2 Datei einlesen und schreiben (1.8)

Schreiben Sie eine Methode `boolean reverseFile(String inFilename, String outFilename)`. Diese soll die Datei mit dem Dateinamen in der Variable `inFilename` öffnen, einlesen und die Zeilen in dieser Datei umgekehrt in `outFilename` schreiben.

Beenden Sie die Methode mit dem Rückgabewert `true`.

Im Fehlerfall geben Sie eine sinnvolle Fehlermeldung aus, die auf die Fehlerursache schließen lässt und beenden Sie die Methode mit dem Rückgabewert `false`.

**Beispiel:** Enthält `inFilename` den folgenden Text

```
Dies ist die erste Zeile
doch die zweite folgt sogleich
```

so soll `outFilename` danach folgenden Text enthalten:

```
doch die zweite folgt sogleich
Dies ist die erste Zeile
```

## 3 Bonusaufgabe: Philosopher's Problem

In der Programmierung mit mehreren Threads kann es zu vielen Problemen kommen, wenn sich Threads Resourcen miteinander teilen müssen (und ihre Zugriffe auf diese koordinieren müssen). Ein Deadlock tritt z.B. dann auf, wenn ein Thread A auf eine Resource warten, die einem Thread B zugeteilt ist, der

ebenfalls auf eine Resource wartet, die allerdings Thread A zugeteilt ist. In diesem Fall blockieren sich beide Threads.

Ein Beispiel für einen Deadlock ist das Philosophen-Problem (siehe `http://de.wikipedia.org/wiki/Philosophenproblem`). Hier sitzen fünf Philosophen im Kreis. Jeder Philosoph hat einen Teller mit Spaghetti vor sich. Zwischen je zwei Tellern liegt eine Gabel (d.h. es sind fünf Gabeln am Tisch).

Wenn ein Philosoph hungrig ist, greift er zunächst zur Gabel links von seinem Teller (die darauf dem linken Nachbarn nicht mehr zur Verugung steht), danach zur rechten Gabel. Mit beiden Gabeln in der Hand kann der Philosoph nun mit dem Essen beginnen. Danach legt er beide Gabeln zurück. Sollte eine Gabel gerade von einem der Nachbarn benutzt werden, wartet der Philosoph bis der Nachbar die Gabel zurücklegt (ohne jedoch eine bereits genommene Gabel zuuckzulegen).

Das Philosophenproblem ist in den folgenden Klassen und Methoden implementiert:

```
public class Gabel {
    private boolean genommen = false;

    public synchronized void nehmen() { // Atomare Operation
        while(genommen) {
            try {
                System.out.println("Gabel besetzt");
                wait(); // Warte auf Gabel
            } catch(InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }

        genommen = true;
    }

    public synchronized void zuruecklegen() {
        genommen = false;
        // Diejenigen, die auf die Gabel warten, werden nun
        // benachrichtigt.
        notifyAll();
    }
}
```

Jeder Philosoph ist ein eigener Thread:

```
public class Philosoph implements Runnable {

    private int index;
    private Gabel links;
    private Gabel rechts;
```

```
    public Philosoph(int index, Gabel links, Gabel rechts) {
        this.index = index;
        this.links = links;
        this.rechts = rechts;
    }

    public void essen() {
        System.out.println("Ph " + index + " nimmt linke Gabel");
        links.nehmen();

        System.out.println("Ph " + index + " nimmt rechte Gabel");
        rechts.nehmen();

        System.out.println("Ph " + index + " isst.");

        links.zuruecklegen();
        rechts.zuruecklegen();
    }

    public void denken() {
        try {
            System.out.println("Philosoph " + index + " denkt");
            // 1 Millisekunde warten.
            Thread.sleep(1);
        } catch(InterruptedException e) {
            // Wir wurden beim Warten unterbrochen
            Thread.currentThread().interrupt();
        }
    }

    public void run() {
        while(true) {
            denken();
            essen();
        }
    }
}
```

Die main-Methode sieht folgendermaßen aus:

```
public static void main(String[] args) {
    final int anzahl = 5;
    // Tisch decken
    Gabel[] gabeln = new Gabel[anzahl];

    for(int i = 0; i < anzahl; i++) {
```

```
        gabeln[i] = new Gabel();
    }

    // Philosophen anlegen
    Philosoph[] philosophen = new Philosoph[anzahl];

    for(int i = 0; i < anzahl; i++) {
        philosophen[i] =
         new Philosoph(i, gabeln[i], gabeln[(i + 1) % anzahl]);
    }

    // Philosophen "starten"
    for(Philosoph phil : philosophen) {
        new Thread(phil).start();
    }
}
```

Wie sieht ein Deadlock in dieser Aufgabe aus? Geben Sie konkret ein Beispiel an und erläutern Sie es. Finden Sie Möglichkeiten an, in dieser Aufgabe Deadlocks zu vermeiden. Versuchen Sie dabei zu vermeiden, dass die Philosophen miteinander kommunizieren und einzelne Philosophen verhungern.