Humboldt-Universität zu Berlin — Institut für Informatik
Grundlagen der Programmierung — **WS 15/16**
(Vorlesung von Prof. Bothe)

Übungsblatt 4: Felder und Rekursion

Abgabe: bis **9:00 Uhr** am **14.12.2015** über **Goya**

Die Lösung dieses Übungsblatts soll in Gruppen von je 2 Personen erfolgen. Die Abgabe der Lösungen erfolgt durch Hochladen **einer Datei für jede Aufgabe** im Goya-System. Verwenden Sie exakt den in der Aufgabe angegebenen Dateinamen! **Java-Lösungen** müssen im UTF-8 Format abgegeben werden und auf dem Institutsrechner **`gruenau5`** korrekt kompilierbar sein (ansonsten wird die Aufgabe mit 0 Punkten bewertet).

Aufgabe 1 (Türme von Hanoi → `Hanoi.java`) — 6 Punkte

Das Spiel der Türme von Hanoi wird auf den französischen Mathematiker Edouard Lucas zurückgeführt, der 1883 folgende kleine Geschichte erfand: Im Großen Tempel von Benares, der die Mitte der Welt markiert, ruht eine Messingplatte, in der drei Diamantnadeln befestigt sind. Bei der Erschaffung der Welt hat Gott vierundsechzig Scheiben aus purem Gold auf eine der Nadeln gesteckt, wobei die größte Scheibe auf der Messingplatte ruht, und die übrigen, immer kleiner werdend, eine auf der anderen.
Ziel des Spiels ist es, die Scheiben von einem Ausgangsstapel `A` unter Zuhilfenahme eines Hilfsstapels `B` auf einen Zielstapel `C` zu verschieben (siehe Abbildung 1). Dabei sollen folgende Bedingungen gelten:

- Es darf immer nur eine Scheibe verschoben werden.
- Es darf nie eine größere Scheibe auf eine kleineren Scheibe verschoben werden.

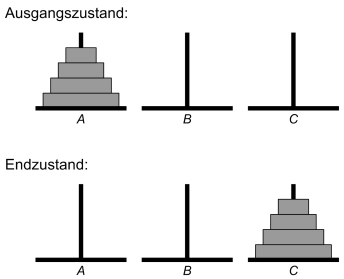

Abbildung 1: Die Türme von Hanoi mit 4 Scheiben: Ausgangs- und Endzustand.
Quelle: http://www.peterloos.de/index.php/m-wpf/m-wpf-userdefined-controls/59-a-wpf-towersofhanoi

Das folgende Java-Programm `Hanoi.java` stammt aus der Vorlesung (II.11, S. 22) und ermittelt unter Berücksichtigung der beiden Bedingungen für n Scheiben rekursiv die notwendigen Verschiebungen und gibt diese auf der Konsole aus.

Hinweis: Um das Java-Programms `Hanoi.java` kompilieren und ausführen zu können, benötigen Sie das Java-Programm `Keyboard.java`. Dieses finden Sie auf der GdP-Webseite im Bereich „Folien“ bzw. unter folgender Adresse:

http://www2.informatik.hu-berlin.de/swt/lehre/GdP-WS-15/java_beispiele/TEIL_II/Keyboard.java

```
public class Hanoi {

    public static void bewege(
            int n,        // Anzahl der Scheiben 'n'
            char start,   // liegen auf 'start'-Platz
            char hilfe,   // (mithilfe des 'hilfe'-Platzes)
            char ziel) {  // muessen auf 'ziel'-Platz

        if (n == 1) {
            System.out.println("Scheibe 1 von " + start + " nach " + ziel);
        } else {
            bewege(n - 1, start, ziel, hilfe);
            System.out.println("Scheibe " + n + " von " + start + " nach "
               + ziel);
            bewege(n - 1, hilfe, start, ziel);
        }
    }

    public static void main(String argv[]) {
        int n;

        System.out.print("Anzahl der Scheiben: ");
        n = Keyboard.readInt();
        if (n > 0) {
            System.out.println("Scheibenbewegungen:");
            bewege(n, 'A', 'B', 'C');
        } else {
            System.out.println("Zahl nicht positiv");
        }
    }
}
```

Erweitern Sie das Programm um eine weitere Bedingung, so dass außerdem nur Verschiebungen zwischen benachbarten Stapeln vorgenommen werden (II.11, S. 39). Folgende Verschiebungen sind dabei zulässig: von `A` nach `B`, von `B` nach `A`, von `B` nach `C` und von `C` nach `B`.

Beispielaufruf und Ausgabe der Implementierung:

```
$ java Hanoi
Anzahl der Scheiben: 2
Scheibenbewegungen:
Scheibe 1 von A nach B
Scheibe 1 von B nach C
Scheibe 2 von A nach B
Scheibe 1 von C nach B
Scheibe 1 von B nach A
Scheibe 2 von B nach C
Scheibe 1 von A nach B
Scheibe 1 von B nach C
```

Aufgabe 2 (Fibonacci-Folge → `Fibonacci.java, FibonacciCache.java`) 6 Punkte

Die Fibonacci-Folge ist eine Folge natürlicher Zahlen, wobei sich das n-te Glied der Folge folgendermaßen berechnen lässt:

$$\text{fib}(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{sonst} \end{cases}$$

Für $n = 0,...,9$ ergibt sich folgende Folge: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34

a) Schreiben Sie ein Java-Programm `Fibonacci.java`, das das n-te Glied der Fibonacci-Folge rekursiv berechnet. Die Zahl n wird dem Programm dabei als Kommandozeilenparameter übergeben. Sie können davon ausgehen, dass n positiv ist. Implementieren Sie die Funktion `fib(int n)`. Die Funktion soll jedes Mal, wenn sie aufgerufen wird, zu Beginn des Aufrufs die Klassenvariable `aufrufe` inkrementieren. Sie können folgendes Code-Fragment für die Implementierung nutzen:

```
public class Fibonacci {

    private static int aufrufe = 0;

    private static int fib(int n) {
        aufrufe++;
        // ToDo: berechne rekursiv das n-te Glied
        // der Fibonacci-Folge und gebe den Wert zurueck
        return 0;
    }

    public static void main(String []args) {
        int n = Integer.parseInt(args[0]);
        System.out.println("fib(" + n + ") = "
            + fib(n) + " (Aufrufe = " + aufrufe + ")");
    }
}
```

Beispielaufruf und Ausgabe der Implementierung:

```
$ java Fibonacci 0
fib(0) = 0 (Aufrufe = 1)

$ java Fibonacci 1
fib(1) = 1 (Aufrufe = 1)

$ java Fibonacci 8
fib(8) = 21 (Aufrufe = 67)

$ java Fibonacci 20
fib(20) = 6765 (Aufrufe = 21891)
```

b) Die einfache Variante der Rekursion benötigt bereits bei kleinen Werten für n sehr viele Aufrufe der Funktion `fib`, um das Ergebnis auszurechnen. Das liegt daran, dass die Funktion für dasselbe n mehrfach aufgerufen werden kann.

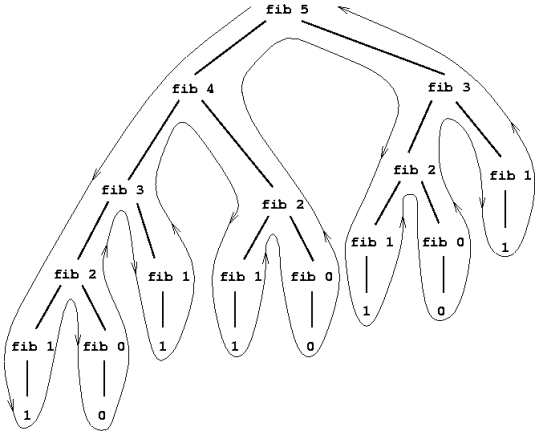

Abbildung 2: Aufrufbaum der Funktion fib mit $n = 5$.
Quelle: https://mitpress.mit.edu/sicp/full-text/sicp/book/node16.html

Wie in Abbildung 2 zu sehen, wird bei der Berechnung des 5-ten Gliedes der Fibonacci-Folge z. B. `fib(2)` insgesamt 3 Mal aufgerufen. Um mehrfache Aufrufe für dasselbe n zu vermeiden und somit die Berechnung zu beschleunigen, ist die Verwendung eines Caches möglich, der bereits berechnete Ergebnisse zwischenspeichert.

Schreiben Sie ein Java-Programm `FibonacciCache.java`, das das n-te Glied der Fibonacci-Folge rekursiv mithilfe eines Caches berechnet. Die Zahl n wird dem Programm dabei als Kommandozeilenparameter übergeben. Sie können davon ausgehen, dass n positiv ist. Implementieren Sie die Funktion `fib(int n, int[] cache)`, die für jedes n höchstens einmal aufgerufen werden soll. Nutzen Sie als Cache ein Array vom Typ int, in dem Sie bereits berechnete Ergebnisse zwischenspeichern und ggf. wieder auslesen (bitte beachten Sie: bei der Initialisierung eines Arrays vom Typ `int` werden alle Einträge initial auf `0` gesetzt). Die Funktion soll jedes Mal, wenn sie aufgerufen wird, zu Beginn des Aufrufs die Klassenvariable `aufrufe` inkrementieren. Sie können folgendes Code-Fragment für die Implementierung nutzen:

```
public class FibonacciCache {

    private static int aufrufe = 0;

    private static int fib(int n, int[] cache) {
        aufrufe++;
        // ToDo: berechne rekursiv das n-te Glied der Fibonacci-Folge,
        // wobei fib fuer jedes n maximal einmal aufgerufen werden
        // soll und gebe den Wert zurueck
        return 0;
    }

    public static void main(String []args) {
        int n = Integer.parseInt(args[0]);
        int[] cache = new int[n]; // initial sind alle Eintraege 0
        System.out.println("fib(" + n + ") = "
            + fib(n, cache) + " (Aufrufe = " + aufrufe + ")");
    }
}
```

Beispielaufruf und Ausgabe der Implementierung:

```
$ java FibonacciCache 0
fib(0) = 0 (Aufrufe = 1)

$ java FibonacciCache 1
fib(1) = 1 (Aufrufe = 1)

$ java FibonacciCache 8
fib(8) = 21 (Aufrufe = 9)

$ java FibonacciCache 20
fib(20) = 6765 (Aufrufe = 21)
```

Aufgabe 3 (Vertauschen von Zahlen → `Blatt4_Aufgabe3.txt`) **8 Punkte**

Betrachten Sie die folgenden vier Java-Methoden. Das Ziel aller Methoden ist es, die Werte der übergebenen Variablen (`a` und `b` im ersten Fall oder die ersten beiden Feldelemente `a[0]` und `a[1]`) **im aufrufenden Programm, also auch außerhalb** von `tausche1()`, ...,`tausche4()` zu vertauschen. Sie dürfen davon ausgehen, dass ein übergebenes Feld stets mindestens zwei Einträge hat. Leider haben nicht alle Methoden den gewünschte Effekt. Ordnen Sie jede Methode in eine der drei folgenden Kategorien ein und begründen Sie die Einordnung **in einem Satz**.

a) Die Methode arbeitet für alle Eingaben korrekt und hat den gewünschten Effekt.

b) Die Methode arbeitet für die meisten Eingaben korrekt (mehr als 50% der möglichen Eingabewerte), aber nicht für alle.

c) Die Methode arbeitet für die meisten Eingaben nicht korrekt.

```
public static void tausche1(int a, int b) {
    int c = a;
    a = b;
    b = c;
}

public static void tausche2(int[] a) {
    int c = a[0];
    a[0] = a[1];
    a[1] = c;
}

public static void tausche3(int[] a) {
    a[0] = a[1] - a[0];
    a[1] = a[1] - a[0];
    a[0] = a[1] + a[0];
}

public static void tausche4(int[] a) {
    int[] c = new int[a.length];
    c[0] = a[1];
    c[1] = a[0];
    for (int i = 2; i < a.length; i++)
        c[i] = a[i];
    a = c;
}
```

Geben Sie die Datei `Blatt4_Aufgabe3.txt` mit folgendem Format ab:

```
tausche1: <Einordnung> - <Begründung>
tausche2: <Einordnung> - <Begründung>
tausche3: <Einordnung> - <Begründung>
tausche4: <Einordnung> - <Begründung>
```