Programmierkurs Java

Dr. Dietrich Boles

Aufgaben zu UE17-Arrays
(Stand 28.09.2012)

Aufgabe 1:

Wir simulieren die Abstimmung bei "Deutschland sucht den Superstar". Die Zuschauer können anrufen und ihren Favoriten aus einer bestimmten Anzahl an Sängerinnen und Sängern wählen. Auf dem TV ausgeben wird letztendlich pro Sängerin bzw. Sänger die Anzahl an Zuschauern, die für sie bzw. ihn gestimmt haben, und zwar prozentual.

Schreiben Sie Java-Programm, in dem der Benutzer zunächst die Anzahl an Sängern und anschließend für jeden Sänger die Anzahl an Anrufen eingeben muss. Speichern Sie die Daten in einem geeigneten Array ab. Anschließend soll das Ergebnis der Abstimmung in Form eines Balkendiagramms auf den Bildschirm ausgegeben werden. Und zwar sollen entsprechend der prozentualen Verteilung der Telefonanrufe jeweils Balken aus *-Zeichen (100 % entsprechen dabei 100 *-Zeichen) sowie anschließend der absolute Wert der Telefonanrufe auf den Bildschirm ausgegeben werden.

Beispiel für einen Programmablauf (Eingaben stehen in <>):

```
Anzahl der Saenger (> 0): <4>
Anrufe für Saenger 1 (>= 0): <50>
Anrufe für Saenger 2 (>= 0): <50>
Anrufe für Saenger 3 (>= 0): <40>
Anrufe für Saenger 4 (>= 0): <60>

Abstimmungsergebnis:
************************ 50
************************ 50
**************************** 40
****************************** 60
```

Aufgabe 2:

Schreiben Sie eine Funktion, die Histogramme auf dem Bildschirm darstellt. Die Funktion bekommt ein eindimensionales int-Array (mit nicht-negativen Werten) als Parameter übergeben und soll entsprechend hohe *-Säulen auf den Bildschirm zeichnen.

Beispiel: f([3, 1, 2, 2, 4, 1, 0, 3, 4]) produziert

```
     *    *
*    *   **
* ***   **
****** **
```

Integrieren Sie diese Funktion in ein Java-Programm, mit dessen Hilfe diese Funktion getestet werden kann.

Aufgabe 3:

Schreiben Sie ein Programm zur Matrizenmultiplikation!

Aufgabe 4:

Bei dieser Aufgabe sollen Sie einen einfachen Fahrtroutenplaner implementieren. Stellen Sie sich dazu ein Netz von Städten vor, das durch Straßen miteinander verbunden ist. Schreiben Sie ein Programm, was genau folgendes tut:

- Zunächst wird die Anzahl *anzahl* an Städten eingelesen.
- Anschließend werden die Namen von *anzahl* Städten eingelesen.
- Danach wird die Anzahl *direkt* der Direktverbindungen zwischen einzelnen Städten eingelesen (Verbindungen sind bidirektional)
- Anschließend werden die *direkt* Direktverbindungen eingelesen, und zwar in der Form Ausgangsstadt, Zielstadt.
- Danach soll das Programm in einer Endlosschleife als Auskunftssystem dienen. Für jede Auskunft sollen jeweils zwei Städtenamen eingelesen werden. Das Programm berechnet dann, ob eine Verbindung (auch indirekte!) zwischen den beiden Städten existiert.

Achten Sie bei den Nutzereingaben auf mögliche Fehler!

Beispiel:

```
5
Oldenburg Bremen Hamburg Frankfurt München
3
Oldenburg Bremen
Bremen Hamburg
Bremen Frankfurt
Oldenburg Hamburg (Ausgabe: Verbindung existiert)
Frankfurt Oldenburg (Ausgabe: Verbindung existiert)
Hamburg München (Ausgabe: keine Verbindung)
```

Aufgabe 5:

Vermutlich wurden sie im alten China entdeckt – jene harmonischen Anordnungen von Zahlen in einem quadratischen Schema derart, dass die Summe der Elemente jeder Zeile, jeder Spalte und der beiden Diagonalen gleich einer Konstanten ist. Sie sollen im Folgenden ein Java-Programm entwickelt, das magische Quadrate erzeugt, und zwar mit einem Algorithmus, den schon die alten Chinesen entdeckt haben. Das Programm soll zunächst die Zahl der Zeilen und Spalten des Quadrats einlesen (>= 3, ungerade, <=99), dann das Magische Quadrat berechnen und es anschließend auf den Bildschirm ausgeben.

Der Algorithmus funktioniert so, dass mit der Zahl 1 angefangen wird. Anschließend wird versucht, die jeweils nächst höhere Zahl im Quadrat zu platzieren.

- Die Zahl 1 wird in das Feldelement senkrecht unter der Mitte des Quadrats eingetragen.
- Steht eine Zahl im Feldelement `quadrat[zeile][spalte]`, so kommt ihr Nachfolger in das Feldelement `quadrat[zeile+1][spalte+1]`, sofern dieses Feldelement nicht schon besetzt ist.
- Ist ein solches Feldelement `quadrat[zeile'][spalte']` schon besetzt, versucht man es von dem besetzten Feldelement ausgehend mit dem Feldelement `quadrat[zeile'+1][spalte'-1]`, und wiederholt dies solange, bis man gemäß dieser Regel ein unbesetztes Feldelement findet (siehe bspw. die Platzierung der 6 im unten stehenden Beispiel).
- Läuft der Zeilenindex unten aus dem Feld heraus, beginnt er wieder oben; läuft er rechts aus dem Feld heraus, beginnt er wieder links.

Bei der Eingabe von 5 wird das folgende magische Quadrat erzeugt:

```
11 24  7 20  3
 4 12 25  8 16
17  5 13 21  9
10 18  1 14 22
23  6 19  2 15
```

Aufgabe 6:

Schreiben Sie eine Funktion, die eine quadratische (n x n) - Matrix um 90° nach rechts rotiert.

```
1 2 3          7 4 1
4 5 6    ->    8 5 2
7 8 9          9 6 3
```

Schreiben Sie weiterhin ein Java-Programm, das zunächst eine Zahl n und anschließend n x n Zahlen von der Tastatur einliest und diese in einer n x n-Matrix abspeichert. Zunächst soll die Originalmatrix auf dem Bildschirm ausgegeben werden. Anschließend soll mit dieser Matrix die Rotationsfunktion aufgerufen werden. Zum Schluss soll die rotierte Matrix auf den Bildschirm ausgegeben werden.

Aufgabe 7:

Implementieren Sie folgende Funktionen:

1. Eine Funktion, die zwei eindimensionale int-Arrays als Parameter übergeben bekommt und überprüft, ob die darin gespeicherten Werte jeweils gleich sind

   Beispiel: `f([1, 4, 3], [1, 4, 3]) == true`

2. Eine Funktion, die eine *n* x *n* Matrix mit int-Werten als Parameter übergeben bekommt und die überprüft, ob die Summe aller Reihen und Spalten und der beiden Diagonalen identisch ist.

Beispiel: f([1,4,1],[2,2,2],[3,0,3]) == true

3. Eine Funktion, die drei eindimensionale int-Arrays als Parameter übergeben bekommt und die eine Matrix zurückliefert, bei der die drei Arrays die Zeilen bilden.

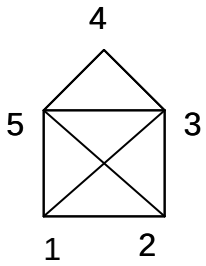
   Beispiel: f([1], [1,3,4], [2, 3]) == [[1], [1, 3, 4], [2, 3]]

4. Eine Funktion, die drei eindimensionale int-Arrays als Parameter übergeben bekommt und die eine Matrix zurückliefert, bei der die einzelnen Werte den Werten der Arrays entsprechen (wo ist der Unterschied zu Teilaufgabe (3)).

   Beispiel: f([1], [1,3,4], [2, 3]) == [[1], [1, 3, 4], [2, 3]]

Aufgabe 8:

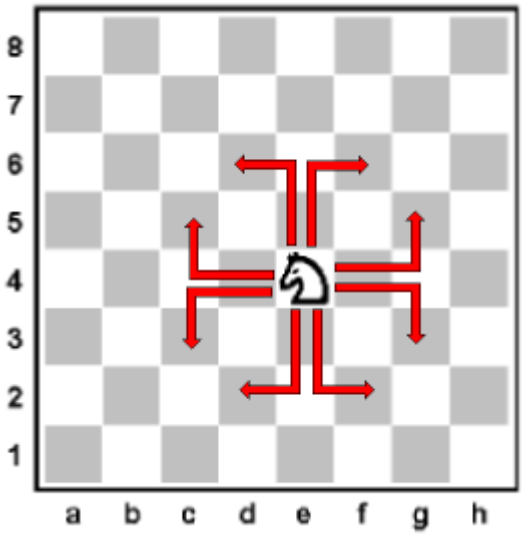
Sie haben in Ihrer Kindheit bestimmt mal das „Haus des Nikolaus“ gezeichnet. Hierbei geht es darum, das folgende Haus so zu zeichnen, dass weder der Stift abgesetzt wird noch eine Linie zweimal gezeichnet wird.



Schreiben Sie ein Programm, das alle Möglichkeiten zum Zeichnen des Haus des Nikolaus ausgibt, wenn mit dem Zeichnen in der linken unteren Ecke begonnen wird. Geben Sie jeweils die entsprechende Nummernfolge aus, z.B. 153125432

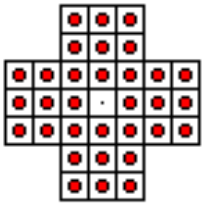
Aufgabe 9:

Stellen Sie sich ein Schachbrett (8x8 Felder) vor. Auf diesem Brett sitzt in der linken unteren Ecke ein einzelner Springer (Pferd). Versuchen Sie eine Folge von Zügen zu finden, so dass der Springer jedes Feld des Brettes genau einmal besucht.



Aufgabe 10:

Schreiben Sie ein Java-Programm, das das Solitaire-Spiel löst. Die Abbildung skizziert das Spielfeld in der Ausgangssituation. Es ist ein Ein-Personen-Spiel. Ein Spielzug sieht so aus, dass der Spieler jeweils eine beliebige Figur auswählt, bei der ein Nachbarfeld besetzt und das dahinter liegende Feld frei ist. Der Spieler nimmt die Figur, platziert sie auf dem freien Feld und entfernt die übersprungene Figur vom Spielfeld. Ziel des Spiels ist es, eine Folge von Spielzügen zu finden, so dass zum Schluss nur noch eine einzige Spielfigur auf dem Spielfeld vorhanden ist. Ihr Programm soll eine solche Folge finden und die Spielzüge der Folge ausgeben.



Aufgabe 11:

Sie kennen sicher das Spiel *Sudoku*: Das Spiel besteht aus einem Gitterfeld mit 3 × 3 Blöcken, die jeweils in 3 × 3 Felder unterteilt sind, insgesamt also 81 Felder in 9 Reihen und 9 Spalten. In einige dieser Felder sind schon zu Beginn Ziffern zwischen 1 und 9 eingetragen. Typischerweise sind 22 bis 36 Felder von 81 möglichen vorgegeben. Ziel des Spiels ist es nun, die leeren Felder des Puzzles so zu vervollständigen, dass in jeder der je neun Zeilen, Spalten und Blöcke jede Ziffer von 1 bis 9 genau einmal auftritt.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

a) Schreiben Sie ein Programm, das ein vorgegebenes Sudoku löst und die Lösung auf den Bildschirm ausgibt. Sie können dabei das gegebene Sudoku als fest kodierte Matrix im Sourcecode repräsentieren.

b) Schreiben Sie ein Programm, das ein neues Sudoku erstellt.

Aufgabe 12:

Implementieren Sie einen interpretativen Hamster-Simulator mit ASCII-Repräsentation. Nutzen Sie folgende Symbole:

>	Hamster (Blickrichtung OST)
^	Hamster (Blickrichtung NORD)
<	Hamster (Blickrichtung WEST)
v	Hamster (Blickrichtung SUED)
o	Kachel mit mindestens einem Korn
#	Kachel mit Mauern

Nutzen Sie zur internen Repräsentation des Territoriums ein Array. Der Anfangszustand des Territoriums kann im Sourcecode festgelegt werden.

Das Programm soll zunächst das Anfangsterritorium auf dem Bildschirm ausgeben. Anschließend kann der Nutzer wiederholt die Hamster-Befehle „vor“, „linksUm“, „nimm“ bzw. „gib“ sowie den Befehl „quit“ eingeben. Nach Eingabe eines Hamster-Befehls wird dabei jeweils der Befehl ausgeführt und das resultierende Territorium auf den Bildschirm ausgegeben. Die Eingabe von „quit“ beendet das Programm.

Beispielhafte Ausgabe des Territoriums:

```
+-+-+-+-+-+-+
| |o| | |#| |
+-+-+-+-+-+-+
| | |>| |o| |
+-+-+-+-+-+-+
| | | | | | |
```

```
+-+-+-+-+-+-+
| |  |#|#|  | |
+-+-+-+-+-+-+
```

Aufgabe 13:
In der heutigen „elektronischen Welt“ wird es immer wichtiger, Daten zu verschlüsseln, damit sie nicht in die Hände Fremder gelangen. Verschlüsselungsalgorithmen gibt es allerdings bereits sehr viel länger als es Computer gibt. Ein solches Verschlüsselungsverfahren nennt sich „Cäsar-Verschlüsselung“, benannt nach Julius Cäsar, der es als erster benutzt haben soll. Um diese Cäsar-Verschlüsselung geht es in dieser Aufgabe.

Die Cäsar-Verschlüsselung eines Textes bestehend aus Buchstaben funktioniert folgendermaßen: Verschiebe jeden Buchstaben um eine bestimmte vorgegebene Verschiebungsdistanz im Alphabet nach hinten. Beträgt die Verschiebungsdistanz beispielsweise 1 wird aus einem ‚a’ ein ‚b’, aus einem ‚b’ ein ‚c’, … und aus einem ‚z’ ein ‚a’ (d.h. am Ende des Alphabets wird wieder vorne begonnen). Beträgt die Verschiebungsdistanz 3 wird aus einem ‚a’ ein ‚d’, aus einem ‚b’ ein ‚e’, … und aus einem ‚z’ ein ‚c’.

Aufgabe (a): Schreiben Sie eine Funktion `verschluesseln` mit folgender Signatur `static char[] verschluesseln(char[] str, int verschiebung`). Die Funktion soll das übergebene char-Array entsprechend der Cäsar-Verschlüsselung verschlüsseln, und zwar mit einer Verschiebungsdistanz (beliebiger int-Wert, u.U. auch negativ!), der im zweiten Parameter übergeben wird. Dabei soll das übergebene Array selbst nicht verändert werden. Stattdessen sollen die verschlüsselten Charakter in einem intern erzeugten Array gespeichert werden, das als Ergebniswert der Funktion geliefert wird. Verschlüsselt werden sollen dabei nur Kleinbuchstaben, alle anderen Zeichen sollen unverändert bleiben. Beispiel:

```
char[] text = {’a’, ’1’, ’z’};
char[] verschluesselterText = verschluesseln(text, 3);
// Ergebnis: verschluesselterText entspricht {’d’, ’1’, ’c’}
```

Aufgabe (b): Schreiben Sie ein Programm, das die Funktion `verschluesseln` ein einziges (!) Mal aufruft, um folgenden Text mit einer Verschiebungsdistanz von 3 zu verschlüsseln (Achten Sie darauf, dass der Text aus 2 Zeilen besteht!):

Mein Login ist "karl";
mein Passwort ist "meier"

Einen String `str` in ein char-Array zu verwandeln, funktioniert folgendermaßen:

char[] zeichen = str.toCharArray();

Ein char-Array `zeichen` in einen String umzuwandeln, funktioniert folgendermaßen:

String str = new String(zeichen);

Folgendes Ergebnis sollte auf dem Bildschirm erscheinen:

```
Mhlq Lrjlq lvw "nduo";
phlq Pdvvzruw lvw "phlhu"
```

Aufgabe 14:
Sie alle kennen aus Ihrer Kindheit das so genannte **Slider-Spiel**. Es ist ein Spiel für eine Person. Das Spielgerät ist eine Tafel mit 4 * 4 Feldern. In dieser Tafel sind auf 15 Feldern Plättchen mit den Ziffern 1 bis 15 platziert. Ein Feld ist leer. Die Plättchen sind verschiebbar. Gegeben eine bestimmte Ausgangsstellung der Plättchen ist es das Ziel, die Plättchen in der Reihenfolge 1 bis 15 anzuordnen.

Schreiben Sie ein Programm, mit dem ein Benutzer das Slider-Spiel spielen kann. Beachten und behandeln Sie falsche Benutzereingaben. Achten Sie auf einen sauberen Programmentwurf (prozedurale Zerlegung)! Wählen Sie aussagekräftige Bezeichner! Die Ausgangsstellung können Sie beliebig vorgeben.

Beispielablauf:

```
+--+--+--+--+
|11|12|13|14|
+--+--+--+--+
| 1| 2| 3| 4|
+--+--+--+--+
| 8| 7| 6| 5|
+--+--+--+--+
|  | 9|10|15|
+--+--+--+--+
Zeile: 3
Spalte: 1
+--+--+--+--+
|11|12|13|14|
+--+--+--+--+
| 1| 2| 3| 4|
+--+--+--+--+
| 8| 7| 6| 5|
+--+--+--+--+
| 9|  |10|15|
+--+--+--+--+
...

+--+--+--+--+
|  | 1| 2| 3|
+--+--+--+--+
| 4| 5| 6| 7|
+--+--+--+--+
| 8| 9|10|11|
+--+--+--+--+
|12|13|14|15|
+--+--+--+--+
```

```
Fertig!
```

Aufgabe 15:

Bei dieser Aufgabe geht es um das Mischen zweier sortierter Arrays. Implementieren Sie dazu folgende Funktion:

```
/**
  Die Funktion mischt die beiden uebergebenen Arrays zu
  einem neuen Array;
  Vorbedingung: Die beiden uebergebenen Arrays sind sortiert.
  Nachbedingung: Das gelieferte Array enthaelt alle Elemente der
    beiden uebergebenen Arrays und ist sortiert. Die beiden
    uebergebenen Arrays bleiben unveraendert.
  @param menge1 sortiertes Array (!= null)
  @param menge2 sortiertes Array (!= null)
  @return sortiertes Array mit allen Elemente der uebergebenen
          Arrays
*/
public static int[] mischen(int[] menge1, int[] menge2);
```

Beispiel:

```
    int[] array1 = {1,3,3,5,6,9};
    int[] array2 = {2,3,5,7,8,9,10};
    int[] ergebnis = mischen(array1, array2);
    // ergebnis == {1,2,3,3,3,5,5,6,7,8,9,9,10}
```

Mögliches Verfahren: Legen Sie für beide Arrays jeweils einen aktuellen Index an. Durchlaufen Sie die beiden Arrays. Vergleichen Sie die beiden Elemente des jeweils aktuellen Index. Fügen Sie das jeweils kleinere Element in das neue Array ein und erhöhen Sie den dazu gehörenden aktuellen Index.

Aufgabe 16:

Bei dieser Aufgabe geht es um das Ordnen einer ungeordneten Matrix (zweidimensionales Array). Ordnen bedeutet: Nach Aufruf der Methode sind die Elemente in der Matrix von oben nach unten und von links nach rechts in aufsteigender Größe sortiert. Implementieren Sie dazu folgende Funktion:

```
public static void matrixOrdnen(int[][] matrix);
```

Beispiel:

```
    int[][] matrix = {{2, 4, 6},
                      {7, 3},
                      {2, 9, 8, 4}};
    matrixOrdnen(matrix);
    // matrix == {{2, 2, 3},
                 {4, 4},
                 {6, 7, 8, 9}};
```

Ein möglicher Algorithmus: Kopieren Sie die Elemente der Matrix in ein genügend großes eindimensionales Array, sortieren Sie dies und kopieren Sie anschließend die Elemente wieder zurück in die Matrix.

Aufgabe 17:

Implementieren Sie eine Funktion, die die transitive Hülle einer übergebenen Matrix berechnet:

```
public static boolean[][] transitiveHuelle(boolean[][] matrix)
```

Sie können voraussetzen, dass die übergebene Objektvariable ungleich null ist und auf ein n x n-Array verweist, mit n >= 1. Die Performance Ihrer Funktion ist unwichtig!

Die **transitive Hülle** th(m) einer Matrix m sei dabei folgendermaßen definiert:

> Besitzt ein Element m[i][j] der Matrix den Wert true, dann bedeutet dies: Es existiert eine direkte Verbindung von i nach j. Besitzt ein Element m[i][j] der Matrix den Wert false, dann bedeutet dies: Es existiert keine direkte Verbindung von i nach j. In th(m), der transitiven Hülle von m, besitzen genau die Elemente th(m)[i][j] den Wert true, die eine direkte oder indirekte Verbindung zwischen i und j kennzeichnen. *Indirekt* bedeutet: es gibt ein m und es gibt einen Pfad (v_1, v_2, ..., v_m) mit i = v_1 und j = v_m und m[v_j][v_{j+1}] = true für alle j = 1, ... m-1.

Beispiel:

```
m = F T F T      th(m) = T T T T
    F F T F              F T T F
    F T F F              F T T F
    T F F F              T T T T
```

Aufgabe 18:

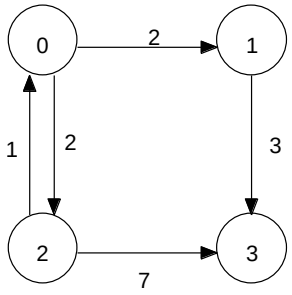
Implementieren Sie eine Funktion zur Lösung des „all pair shortest path problems“, also dem Finden der Länge der jeweils kürzesten Pfade zwischen allen Knoten in einem gerichteten Graphen (Sie können sich vorstellen, dass die Knoten des Graphen Städte und die Kanten des Graphen Einbahnstraßen zwischen den Städten repräsentieren.):

```
public static int[][] kuerzesteEntfernung(int[][] matrix)
```

Übergeben wird eine Matrix m, in der m[i][j] die Länge einer Kante zwischen einem Knoten i und einem Knoten j angibt (also die Länge einer Einbahnstraße zwischen den beiden Städten). Existiert keine Kante zwischen den Knoten i und j, so enthält m[i][j] den Wert –1. Sie können voraussetzen, dass die übergebene Objektvariable m ungleich null ist und auf ein n x n-Array verweist, mit n >= 1. Die Performance Ihrer Funktion ist unwichtig!

Berechnet und zurückgeliefert werden soll eine Matrix k, in der k[i][j] die kürzeste Entfernung zwischen den Knoten i und j bzgl. der Entfernungsangaben in der Matrix m angibt.

Beispiel:




```
m =  0  2  2 -1      k =  0  2  2  5
    -1  0 -1  3           -1  0 -1  3
     1 -1  0  7            1  3  0  6
    -1 -1 -1  0           -1 -1 -1  0
```

Aufgabe 19:

Das „Game-of-Life“ wird auf einem schachbrettartigen Feld gespielt, das eine „Bevölkerung“ von „toten“ und „lebenden“ Zellen darstellt. Jede Zelle kann „überleben“, „sterben“ oder „geboren“ werden. Die schrittweise Entwicklung von einem Stellungsbild zum nächsten erfolgt gemäß einiger Regeln, die berücksichtigen, wie viele lebende Nachbarzellen eine Zelle hat.

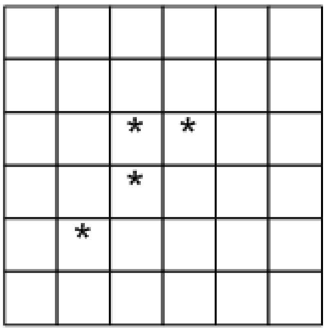
Bei dem Feld handelt es sich um ein Torus-förmiges Feld, bei dem alles, was das Feld nach unten verlässt, oben wieder herauskommt und umgekehrt, und alles, was das Feld nach links verlässt, rechts wieder eintritt und umgekehrt, d.h. alle Zellen haben jeweils genau 8 Nachbarzellen.

Die Regeln, nach denen sich die Population von einer Stellung zur nächsten entwickelt, sind:

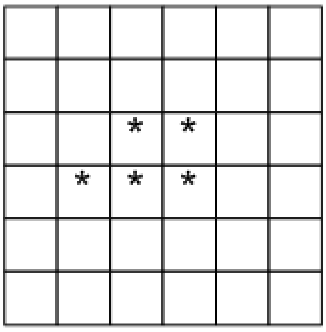
1. Für eine Zelle x, die gerade tot ist, gilt: Wenn x genau 3 lebende Nachbarzellen hat, wird x neu geboren; sonst bleibt x tot.
2. Für eine Zelle x, die gerade lebendig ist, gilt: Wenn x weniger als 2 lebende Nachbarn hat, stirbt x an Vereinsamung; wenn x 2 oder 3 lebende Nachbarzellen hat, bleibt x in der nächsten Stellung lebendig. In allen anderen Fällen stirbt x an Überbevölkerung.

Alle Veränderungen gemäß dieser Regeln geschehen gleichzeitig. Die Simulation beginnt mit einer bestimmten eingelesenen Verteilung von lebenden und toten Zellen.

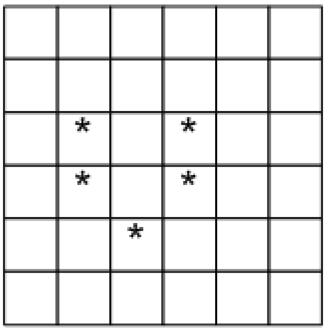
Beispiel:



Population 1



Population 2



Population 3

Aufgabe: Schauen Sie sich die folgende Implementierung des Game-of-Life an. Implementieren Sie die fehlende Funktion *calcNextGeneration* der Klasse *Generations* und testen Sie dann Ihr Programm.

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;

import javax.swing.JFrame;
import javax.swing.JPanel;

public class GameOfLife {

        // Ausgangsfeld
        static int[][] world = {
                        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                        { 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0 },
                        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                        { 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                        { 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                        { 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0 },
                        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0 },
                        { 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0 },
                        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 } };

        public static void main(String[] args) {
                JFrame frame = new JFrame("Game of Life");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setLayout(new BorderLayout());
                WorldPanel panel = new WorldPanel(GameOfLife.world);
                panel.setPreferredSize(new Dimension(GameOfLife.world[0].length * 20,
                                GameOfLife.world.length * 20));
                frame.add(panel, BorderLayout.CENTER);
                frame.pack();
                new Simulation(panel).start();
```

```
                frame.setVisible(true);
        }
}

class WorldPanel extends JPanel {

        private int[][] world;

        public WorldPanel(int[][] world) {
                this.world = world;
        }

        public int[][] getWorld() {
                return this.world;
        }

        public void setWorld(int[][] world) {
                this.world = world;
        }

        public void paintComponent(Graphics g) {
                super.paintComponent(g);
                for (int r = 0; r < this.world.length; r++) {
                        for (int s = 0; s < this.world[r].length; s++) {
                                if (this.world[r][s] == 0) {
                                        g.setColor(Color.WHITE);
                                } else {
                                        g.setColor(Color.DARK_GRAY);
                                }
                                g.fillRect(s * 20, r * 20, 20, 20);
                        }
                }
        }
}

class Simulation extends Thread {

        WorldPanel panel;

        public Simulation(WorldPanel panel) {
                this.panel = panel;
        }

        public void run() {
                try {
                        Thread.sleep(1000);
                        while (true) {
                                this.panel.setWorld(Generations.calcNextGeneration(this.panel
                                                .getWorld()));
                                this.panel.repaint();
                                Thread.sleep(500);
                        }
                } catch (Exception exc) {
                        exc.printStackTrace();
                }
        }
}

class Generations {

        /**
         * Liefert ein Feld mit der nächsten Generation gemäß der
         * Game-of-Life-Regeln.
         *
```

```
         * @param world
         *            das aktuelle Feld; 1 entspricht: Zelle ist lebendig; 0
         *            entspricht: Zelle ist tot
         * @return ein Feld mit der nächsten Generation;1 entspricht: Zelle ist
         *         lebendig; 0 entspricht: Zelle ist tot
         */
        static int[][] calcNextGeneration(int[][] world) {

        }

}
```

Aufgabe 20:

Definieren Sie eine Funktion, die als einzigen Parameter ein Array mit `double`-Werten übergeben bekommt und die den Mittelwert dieser `double`-Werte als `double`-Wert zurück liefert.

Aufgabe 21:

Definieren Sie eine Funktion, die zwei `int`-Arrays als Parameter übergeben bekommt und ein Ergebnis vom Typ `int` liefert. Die Funktion soll die Summe derjenigen Elemente des ersten Arrays berechnen, die durch die Elemente des zweiten Arrays indiziert werden.

Beispiel:

Erstes Array = {23, 4, 12, 14, 5, 9}; Zweites Array = {1, 2, 5}

Berechnung: 4 + 12 + 9 = 25

Lieferung des Wertes 25

Aufgabe 22:

In einem Kreis stehen *n* Kinder (durchnumeriert von 1 bis *n*). Mit Hilfe eines *m*-silbigen Abzählreims wird das jeweils *m*-te unter den noch im Kreis befindlichen Kindern ausgeschieden, bis kein Kind mehr im Kreis steht.

Schreiben Sie ein Java-Programm, das nach Vorgabe von *n* (positive Zahl) und *m* (positive Zahl) die Nummern der Kinder in der Reihenfolge ihres Ausscheidens angibt.

Beispiel*:* Für *n*=6 und *m*=5 ergibt sich die Folge 5, 4, 6, 2, 3, 1.

Aufgabe 23:

Implementieren Sie eine Funktion, die als Parameter ein int-Array übergeben bekommt. Die Funktion soll das

(a) größte,
(b) zweitgrößte

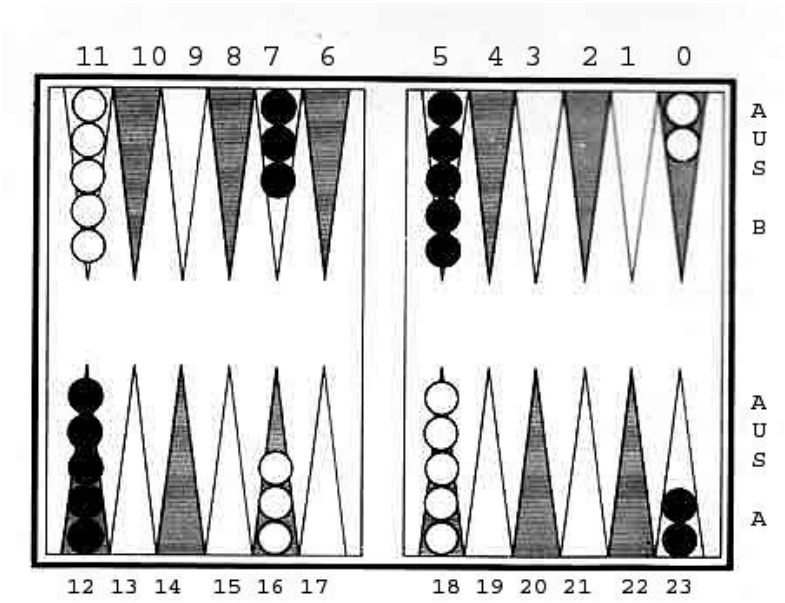
Element des Arrays ermitteln und als Funktionswert liefern. Dabei darf das Array nicht verändert werden.

Aufgabe 24:

Bei dieser Aufgabe sollen Sie ein Programm entwickeln, durch das zwei menschliche Spieler am Computer gegeneinander ein vereinfachtes Backgammon-Spiel spielen können.

Regeln

Das Backgammon-Spielbrett besteht aus 24 Feldern, „Zungen“ genannt, von denen sich jeweils 12 auf einer Seite befinden. Gespielt wird mit 15 weißen Steinen, die Spieler A gehören, und 15 schwarzen Steinen, die Spieler B gehören. Die Startaufstellung der Steine sehen Sie in der folgenden Abbildung.



Spieler A zieht seine Steine gegen, Spieler B mit dem Uhrzeigersinn. Beide versuchen, alle ihre Steine in ihr jeweiliges „Aus“ zu bringen. Wem dies als erstem gelingt, hat das Spiel gewonnen.

Die Spieler ziehen immer abwechselnd. Spieler A beginnt das Spiel. Dazu würfelt er mit einem einzelnen Würfel (Zahlen 1 bis 6). Entsprechend der gewürfelten Zahl, darf er einen seiner Steine entsprechend viele Zungen in seine entsprechende Zugrichtung fortbewegen. Würfelt Spieler A bspw. in der Startaufstellung eine 4, dürfte er bspw. einen Stein von Zunge 0 auf Zunge 4 bewegen. Würfelt Spieler B anschließend eine 5, dürfte er bspw. einen Stein von Zunge 7 auf Zunge 2 verschieben. Gezogen wird dabei immer der oberste Stein einer Zunge. Steine landen im „Aus“, wenn sie über die letzte Zunge des entsprechenden Spielers hinausgezogen werden können.

Beim Ziehen gelten folgende Einschränkungen:

- Auf jeder Zunge dürfen sich zu jedem Zeitpunkt maximal 5 Steine befinden.
- Ein Stein darf nicht auf eine Zunge gezogen werden, die gerade durch einen oder mehrere gegnerische Steine besetzt ist.

Kann ein Spieler, wenn er an der Reihe ist, nicht ziehen, muss er passen. Passen ist darüber hinaus prinzipiell immer möglich.

Alle anderen Regeln des Original-Backgammon-Spiels gelten bei dieser vereinfachten Variante nicht!

Hinweise zur Umsetzung:

- Wählen Sie als Datenstruktur zur Repräsentation des Spielfeldes ein int-Array der Länge 24. Jedes Array-Element repräsentiert dabei eine Zunge. Beachten Sie, dass das Spiel prinzipiell auch funktionieren soll, wenn eine beliebige andere Länge als 24 (die aber durch 2 teilbar sein muss) gewählt wird. Eine Implementierung des Spiels auf der Grundlage einer festgelegten Codierung der Felder und Züge ist nicht erlaubt!
- Steine von Spieler A auf einer Zunge werden durch eine entsprechend hohe positive Zahl, Steine von Spieler B durch eine entsprechend hohe negative Zahl repräsentiert. Die 0 deutet an, dass die Zunge momentan leer ist. In der Darstellung werden Steine von Spieler A durch ein ‚+', Steine von Spieler B durch ein ‚*' repräsentiert.
- Ein Spielzug entspricht der Angabe des Indexes der entsprechenden Zunge, von der ein Stein gezogen werden soll (also die Werte 0 bis 23). Passen wird durch die Eingabe von `-1` signalisiert.
- Geben Sie das Spielbrett analog zu der Darstellung auf der nächsten Seite aus.
- Der Spielablauf lässt sich folgendermaßen skizzieren:

  Ausgabe des Spielbrettes
  Solange das Spiel noch nicht beendet ist, tue folgendes:
  Würfeln;
  Zugeingabe des Spielers, der an der Reihe ist
  Überprüfung des Zuges auf Gültigkeit und ggfls. Wiederholung
  Ausführen des Spielzugs
  Ausgabe des Spielbrettes
  Bekanntgabe des Siegers

Beispiel für einen Programmablauf (in <> stehen Benutzereingaben):

```
 11 10  9  8  7  6  5  4  3  2  1  0
  +           *     *              +
  +           *     *              +
  +           *     *
  +                 *
  +                 *
```

```
  *                 +
  *                 +
  *           +     +
  *           +     +               *
  *           +     +               *
 12 13 14 15 16 17 18 19 20 21 22 23
Würfel = 6; Spieler A, Zungennummer eingeben: <0>

 11 10  9  8  7  6  5  4  3  2  1  0
  +           *  +  *              +
  +           *     *
  +           *     *
  +                 *
  +                 *

  *                 +
  *                 +
  *           +     +
  *           +     +               *
  *           +     +               *
 12 13 14 15 16 17 18 19 20 21 22 23
Würfel = 2; Spieler B, Zungennummer eingeben: <11>
Falscher Zug! Bitte wiederholen!
Würfel = 2; Spieler B, Zungennummer eingeben: <12>

 11 10  9  8  7  6  5  4  3  2  1  0
  +  *        *  +  *              +
  +           *     *
  +           *     *
  +                 *
  +                 *

                    +
  *                 +
  *           +     +
  *           +     +               *
  *           +     +               *
 12 13 14 15 16 17 18 19 20 21 22 23
Würfel = 3; Spieler A, Zungennummer eingeben:

...
```

Aufgabe 25:

Implementieren Sie einen einfachen Taschenrechner, der auf der *Umgekehrten Polnischen Notation* (UPN) (auch Postfix-Notation genannt) basiert. Der Taschenrechner soll die binären Operationen +, -, *, / und % auf int-Werten unterstützen. Benutzer können nur einstellige Zahlen, also Zahlen zwischen 0 und 9, eingeben. Das Programm soll enden, sobald der Benutzer das Zeichen ‚e' (für exit) eingibt.

Bei der UPN werden eingelesene Zahlen jeweils oben auf einen Stapel (maximal N Elemente, N hier 20) gelegt. Wird ein Operator eingegeben, werden die beiden oberen Elemente vom Stapel entfernt, darauf die Operation angewendet, das Ergebnis ausgegeben und das Ergebnis wieder auf den Stapel gelegt.

`9 + (7 - 2) / 4` in der Infix-Notation entspricht bspw. `9 7 2 - 4 / +` in der UPN

Beispiel für einen Programmablauf:

```
9
7
2
-
→ 5
4
/
→ 1
+
→ 10
e
```

Beachten Sie bitte mögliche Fehlerfälle und reagieren Sie darauf durch entsprechende Ausgaben!

Aufgabe 26:

Implementieren Sie in Java eine boolesche Funktion `istElement` mit folgender Signatur: `static boolean istElement(String[] vektor, String element)`. Übergeben wird der Funktion als erster Parameter ein nach Größe sortiertes vollständig gefülltes Array mit Zeichenketten (ungleich `null`) und als zweiter Parameter eine einzelne Zeichenkette (ungleich `null`). Die Funktion soll überprüfen, ob die einzelne Zeichenkette wertmäßig im Array vorhanden ist.

Der Überprüfungsalgorithmus ist dabei vorgegeben, und zwar soll das durch die Übungsaufgaben bekannte Halbierungsverfahren genutzt werden: Es wird das mittlere Array-Element bestimmt (bei einer gerade Anzahl an Elementen eines der beiden mittleren Elementen) und mit dem `element`-Parameter verglichen. Sind die beiden wertgleich, kann `true` geliefert werden. Ist der Wert des `element`-Parameters kleiner, wird derselbe Algorithmus auf die erste Hälfte des Arrays angewendet. Ist der Wert des `element`-Parameters größer, wird derselbe Algorithmus auf die zweite Hälfte des Arrays angewendet.

Implementieren Sie eine vollständig rekursive Lösung, d.h. es dürfen keine Wiederholungsanweisungen benutzt werden

Nutzen Sie zum Vergleichen von `String`-Objekten die Instanzmethode `public int compareTo(String obj)` der Klasse `java.lang.String`. Sie liefert einen Wert kleiner als 0, wenn das `String`-Objekt, für das sie aufgerufen wurde, kleiner als das als Parameter übergebene `String`-Objekt ist. Wenn die beiden `String`-Objekte gleich sind, wird 0 zurückgegeben, ansonsten ein positiver Wert.

Aufgabe 27:

ISBN-Nummern sind eindeutige Kennzeichner von Büchern. Bis 2006 waren die ISBN-Nummern 10stellig, heute werden 13stellige ISBN-Nummern verwendet.

Allerdings gibt es einen Algorithmus, um 13stellige ISBN-Nummern in äquivalente 10stellige ISBN-Nummern umzurechnen. Dieser Algorithmus hat folgende Gestalt:

- Streiche die ersten 3 Ziffern
- Übernehme die nächsten 9 Ziffern
- Ersetze die letzte Ziffer durch ein so genanntes Prüfzeichen `p`.

Das Prüfzeichen `p` ergibt sich aus den 9 Ziffern z_i (i = 1, …, 9) dabei auf folgende Art und Weise:

Sei $S = \left(\sum_{i=1}^{9} (z_i * i) \right) \% 11$, dann ist p = ‚X', falls S = 10, ansonsten ist p = S.

Beispiele:

ISBN13 = 9871234567880 ISBN10 = 1234567881

ISBN13 = 9871234567890 ISBN10 = 123456789X

ISBN13 = 9871200000001 ISBN10 = 1200000005

Aufgabe: Implementieren Sie die folgende Funktion zur Umrechnung von ISBN13 in ISBN10

```
static char[] getISBN10(char[] isbn13)
```

Sie können dabei davon ausgehen, dass das als Parameter übergebene char-Array eine korrekte ISBN13 enthält.

Aufgabe 28:

Schreiben Sie eine Funktion, die beim Schachspiel eingesetzt werden könnte, und zwar soll mit Hilfe der Funktion überprüft werden, ob sich zwei Damen gegenseitig werfen können (zwei Damen können sich genau dann werfen, wenn sie in derselben Spalte, derselben Reihe oder derselben Diagonale stehen).

Die Funktion soll als Parameter eine quadratische Matrix mit boolean-Werten übergeben bekommen, in der genau zwei Felder den Wert true habe. Die beiden Felder sind die von den Damen besetzten Felder. Die Funktion soll genau dann true liefern, wenn die Damen sich werfen können.

Aufgabe 29:

Masken sind char-Arrays, bei denen das Zeichen `?` die Bedeutung eines Jokers beim Vergleich mit einem anderen char-Array hat. Jeder Joker in der Maske steht für

genau ein beliebiges Zeichen im char-Array. Zum Beispiel passt die Maske `a?b?` auf das char-Array `afbh`, dagegen nicht auf `abcd` oder `acb`.

Schreiben Sie eine **rekursive** Funktion `match`, die als Argumente eine Maske und ein char-Array erhält und beide vergleicht. `match` liefert genau dann `true`, wenn die Maske zum char-Array passt, ansonsten `false`.

Schreiben Sie ein kleines Testprogramm für die Funktion `match`.

Wenn Sie eine rekursive Lösung nicht hinbekommen, können Sie auch eine iterative Funktion implementieren.

Aufgabe 30:
Implementieren Sie in Java eine Funktion `plattmachen`, die ein beliebiges 2-dimensionales `int`-Array als Parameter übergeben bekommt und ein 1-dimensionales `int`-Array liefert, in das das als Parameter übergebene Array Zeile für Zeile von links nach rechts kopiert worden ist. Beachten Sie bitte auch Sonderfälle!

Beispiel:

Parameter (2-dimensionales Array):

```
[2, 3, 4, 5]
[6, 7, 8, 9]
[3, 5, 1, 8]
```

Resultat (1-dimensionales Array):

```
[2, 3, 4, 5, 6, 7, 8, 9, 3, 5, 1, 8]
```

Aufgabe 31:

Implementieren Sie das folgende kleine Spiel, bei dem ein Computer gegen einen Menschen spielt:

Das Spiel wird solange gespielt, bis der Mensch verloren oder 10 Punkte erreicht hat. Es besteht aus mehreren Runden. In jeder Runde generiert der Computer zufällig einen Kleinbuchstaben (‚a' … ‚z') und gibt diesen auf den Bildschirm aus. Der Benutzer muss dann innerhalb von jeweils 3 Sekunden überprüfen, ob genau dieser Buchstabe bereits zum zweiten, vierten, sechsten (also einem Vielfachen von 2) Mal auf dem Bildschirm erschienen ist. In diesem Fall (und nur in diesem) muss er eine beliebige Taste drücken. Hat er Recht, bekommt er einen Punkt. Hat er Unrecht oder überschreitet die Zeitgrenze von 3 Sekunden, hat er unmittelbar verloren. Erreicht er 10 Punkte, hat er gewonnen.

Achtung: Implementieren Sie das Spiel auf eine imperative Art und Weise. Sie können dabei folgende Klasse benutzen:

```
public class Util {
```

```
    /**
     * Blockiert das Programm für eine bestimmt Zeit und
     * ueberprueft, ob während dieser Zeit eine Taste
     * gedrückt wurde.
     *
     * @param secs
     *            Anzahl an Sekunden, die gewartet werden soll
     * @return liefert genau dann true, wenn innerhalb von
     *            secs-Sekunden eine Taste gedrückt wurde
     */
    public static boolean keyPressed(int secs)
}
```

Aufgabe 32:

Implementieren Sie eine Java-Funktion, die für ein char-Array überprüft, ob die dort enthaltenen runden Klammern den Regeln einer vollständigen Klammerung wie bei einem Ausdruck entsprechen. Das heißt: Für jede öffnende Klammer '(' muss eine nachfolgende schließende Klammer ')' existieren und die runden Klammern müssen korrekt verschachtelt sein. Zeichen außer den runden Klammern sollen ignoriert werden.

Folgendes sind korrekt geklammerte Zeichenketten: "()", "", "(()(a)(()((c)))". Diese nicht: "(()", "a (()()) a)", "ww)(".

Aufgabe 33:
Implementieren Sie in Java eine boolesche Funktion `istEnthalten`, die zwei `int`-Arrays übergeben bekommt. Die Funktion soll genau dann `true` liefern, wenn das zweite Array elementweise im ersten Array enthalten ist. Dabei gilt: Ein Array *a* ist genau dann elementweise in einem Array *b* enthalten, wenn alle Elemente von *a* in derselben Reihenfolge und ohne Unterbrechungen auch in *b* enthalten sind. Sie können davon ausgehen, dass keine `null`-Werte als Parameter übergeben werden. Die Nutzung von Strings ist nicht erlaubt.

Beispiel:

```
int[] b = {2, 4, 3, 7, 5, 6};
int[] a1 = {3, 7, 5};
int[] a2 = {3, 7, 6};
```

`a1` ist in `b` enthalten, `a2` ist nicht in `b` enthalten.

Aufgabe 34:
Euro-Banknoten haben eine eindeutige Seriennummer, die aus einem führenden Großbuchstaben, einer Zahl mit 10 Ziffern und einer Prüfziffer bestehen.
Beispiel: Z 6016220022 6

Der führende Buchstabe codiert die nationale Zentralbank (NZB), die den Geldschein in Umlauf gebracht hat. Sie wird NZB-Nummer genannt.

Die Prüfziffer berechnet sich wie folgt:

- Der Buchstabe wird durch seine Position im lateinischen Alphabet ersetzt (bei A also 1, bei Z 26)
- Es wird die Quersumme dieser Positionszahl und der 10 Ziffern berechnet (im Beispiel 2+6+6+0+1+6+2+2+0+0+2+2 = 29)
- Von der Quersumme wird der ganzzahlige Rest zum nächst kleineren Vielfachen von 9 bestimmt (Modulo 9) (im Beispiel 2)
- Der Rest wird von 8 subtrahiert. Das Resultat ist die Prüfziffer (im Beispiel 6). Es sei denn das Resultat ist 0, dann ist die Prüfziffer 9.

Implementieren Sie in Java eine Funktion, die als ersten Parameter eine NZB-Nummer als Buchstaben und als zweiten Parameter eine 10-elementiges Array mit int-Werten übergeben bekommt. Die Funktion soll die Prüfziffer der entsprechenden Euro-Banknote berechnen und als int-Wert zurückliefern. Sie können davon ausgehen, dass die übergebenen Parameter korrekt sind.

Aufgabe 35:

Bei dieser Aufgabe sollen Sie ein Programm entwickeln, mit dem Nutzer ihre Erinnerungsfähigkeit trainieren können. Die Spielidee wurde von dem bekannten Spiel *Senso* (http://de.wikipedia.org/wiki/Senso_(Spiel)) übernommen. Der Programmablauf sieht folgendermaßen aus:

- Es werden maximal *n* Runden gespielt (konkret sei n = 50).
- In jeder Runde wird folgendes gemacht:
  - Das Programm ermittelt eine Zufallszahl zwischen 0 und 9.
  - Anschließend werden nacheinander die Zufallszahlen der bisherigen Runden in der entsprechenden Reihenfolge durch ein Leerzeichen getrennt in einer Zeile auf den Bildschirm ausgegeben. Nach der Ausgabe jeder Zahl wird dabei 1 Sekunde gewartet.
  - Nach der Ausgabe aller Zahlen wird der Bildschirm gelöscht.
  - Nun ist der Nutzer an der Reihe. Er muss sich alle ausgegebenen Zahlen merken und sie in der entsprechenden Reihenfolge eingeben.
  - Sobald der Benutzer einen Fehler macht, wird das Programm beendet und die Anzahl der erfolgreich absolvierten Runden auf den Bildschirm ausgegeben.
- Schafft der Nutzer alle n Runden, wird das Programm ebenfalls mit einer entsprechenden Erfolgsmeldung beendet.

Beispiel für einen Programmablauf (Benutzereingaben stehen in Klammern (<>)):

```
1
„Bildschirm wird gelöscht“
<1>
1 5
„Bildschirm wird gelöscht“
<1>
<5>
1 5 4
„Bildschirm wird gelöscht“
<1>
<5>
```

```
<4>
1 5 4 1
„Bildschirm wird gelöscht“
<1>
<5>
<4>
<1>
1 5 4 1 9
„Bildschirm wird gelöscht“
<1>
<5>
<1>
Fehler! Sie haben 4 Runden erfolgreich überstanden
```

Aufgabe 36:

Entwerfen und implementieren Sie ein Programm, das Folgendes leistet:

- Einlesen eines Klartextes
- Verschlüsseln des Klartextes mit Hilfe der Polybios-Tafel
- Ausgabe des Geheimtextes

Polybios-Tafel:

```
  1 2 3 4   5
1 a b c d   e
2 f g h i/j k
3 l m n o   p
4 q r s t   u
5 v w x y   z
```

Aufgabe 37:

Gegeben ist eine Tabelle mit acht Zeilen und zehn Spalten. In der Tabelle ist in den Zeilen und Spalten u. a. das Wort HUND viermal, KATZE zweimal und MAUS dreimal enthalten.

```
R X O J K H C K U H
S K I M A U S U H U
B N H A T N C H U N
R H M U Z D H U N D
M A U S E K A T Z E
A N H E N U F E T T
U D I N O H U N D E
```

D N U H T E R G L Z

Beachten Sie den Hinweis: Die Zeilen sind von links nach rechts und die Spalten von oben nach unten zu lesen. Entwerfen und implementieren Sie ein Programm, das Folgendes leistet:

- Einlesen einer Tabelle mit maximal 20 Zeilen und maximal 20 Spalten,
- Einlesen eines Wortes,
- Ermitteln der Anzahl des Vorkommens dieses Wortes in der Tabelle,
- Ausgabe der Anzahl.

Aufgabe 38:

Die chromatische Tonleiter besteht aus zwölf Tönen. Schreibt man diese Tonleiter zweimal hintereinander auf, so entsteht die aufsteigende Tonfolge:

c cis d dis e f fis g gis a ais h c cis d dis e f fis g gis a ais h

Von einem Ton dieser Tonfolge zum nächsten führt ein Halbtonschritt und zum übernächsten Ton ein Ganztonschritt.

Jede Dur- bzw. Moll-Tonleiter besteht aus acht aufsteigend geordneten Tönen der gegebenen Tonfolge. Der erste Ton ist der Grundton.

Für eine Dur-Tonleiter gilt:

Vom ersten Ton zum zweiten, vom zweiten zum dritten, vom vierten zum fünften, vom fünften zum sechsten und vom sechsten zum siebenten führt je ein Ganztonschritt.

Vom dritten Ton zum vierten und vom siebenten zum achten führt je ein Halbtonschritt.

Für eine Moll-Tonleiter gilt:

Vom ersten Ton zum zweiten, vom dritten zum vierten, vom vierten zum fünften, vom sechsten zum siebenten und vom siebenten zum achten führt je ein Ganztonschritt.

Vom zweiten Ton zum dritten und vom fünften zum sechsten führt je

ein Halbtonschritt.

Beispiele:

Die g-Dur-Tonleiter besteht aus den Tönen g a h c d e fis g.

Die h-Moll-Tonleiter besteht aus den Tönen h cis d e fis g a h.

Entwerfen und implementieren Sie ein Programm in Oberon oder

Turbo Pascal, das für jeden der Grundtöne c, g, d, a, e, h, fis die Dur-

Tonleiter und für jeden der Grundtöne

Aufgabe 39:

Bei dieser Variante des Nim-Spiel sind n Reihen mit n Streichhölzern vorhanden. Zwei Spieler nehmen abwechselnd Streichhölzer aus einer der Reihen weg. Wie viele sie nehmen, spielt keine Rolle; es muss mindestens ein Streichholz sein und es dürfen bei einem Zug nur Streichhölzer einer einzigen Reihe genommen werden. Derjenige Spieler, der den letzten Zug macht, also die letzten Streichhölzer wegnimmt, gewinnt.

Implementieren Sie diese Variante des Nim-Spiels in Java, so dass zwei menschliche Spieler gegeneinander antreten können.

Aufgabe 40:

In dieser Aufgabe geht es um die Verschlüsselung von Texten. Klartexte über einem Klartextalphabet werden durch die Anwendung von Verschlüsselungsalgorithmen in Geheimtexte über einem Geheimtextalphabet überführt. Verschlüsselungsverfahren sollen gewährleisten, dass nur Befugte bestimmte Botschaften lesen können.

Eine der beiden grundlegenden Verschlüsselungsklassen ist die *Transposition*. Dabei werden die Zeichen einer Botschaft (des Klartextes) umsortiert. Jedes Zeichen bleibt zwar unverändert erhalten, jedoch wird die Stelle, an der es steht, geändert. Als sehr einfaches und anschauliches Beispiel einer geregelten Transposition soll hier die *Gartenzaun-Transposition* dienen: Die Buchstaben des Klartextes werden abwechselnd auf zwei Zeilen geschrieben, so dass der erste auf der oberen, der zweite auf der unteren, der dritte Buchstabe wieder auf der oberen Zeile steht und so weiter. Der Geheimtext entsteht, indem abschließend die Zeichenkette der unteren Zeile an die der oberen Zeile angefügt wird. (aus Wikipedia)

Beispiel:

Klartext: `Gartenzaun`
Verfahren:

```
G r e z u
 a t n a n
```

Geheimtext: `Grezuatnan`

Aufgabe:

Definieren und implementieren Sie eine Klasse, die die folgenden zwei Funktionen definiert.

```
class GartenzaunTransposition {

     // liefert den verschluesselten Text
     static char[] verschluesseln(char[] klartext);

     // liefert den entschluesselten Text
     static char[] entschluesseln(char[] geheimtext);
}
```

Aufgabe 41:

Implementieren Sie auf imperative Art und Weise das folgende kleine Taktikspiel für *n* (>=2) Spieler.

Lesen Sie anfangs die Anzahl *n* der teilnehmenden Spieler ein. Alle *n* Spieler besitzen je einen Haufen von 100 Kugeln. Gespielt werden 10 Runden. In jeder Runde wählt jeder Spieler (geheim) eine bestimmte Anzahl (>= 0) an Kugeln aus, die von seinem Haufen entfernt werden. Jeweils der bzw. die Spieler mit der größten Anzahl an gewählten Kugeln gewinnt/gewinnen die Runde und einen Punkt. Sieger des Spiels ist der/sind die Spieler, der/die nach 10 Runden die meisten Punkte hat/haben.

Orientieren Sie sich bei der Implementierung des Spiels an folgendem Beispielablauf (Benutzereingaben in <>). Behandeln Sie fehlerhafte Benutzereingaben adäquat.

```
Anzahl an Spielern: <3>

Runde 1 von 10 Runden
Spieler 1: Wie viele Kugeln von 100? <10>
Spieler 2: Wie viele Kugeln von 100? <9>
Spieler 3: Wie viele Kugeln von 100? <5>
Spieler 1 hat die Runde (mit) gewonnen und bekommt einen Punkt!
Spielstand nach Runde 1:
Spieler 1 hat 1 Punkte und 90 Restkugeln
Spieler 2 hat 0 Punkte und 91 Restkugeln
Spieler 3 hat 0 Punkte und 95 Restkugeln

Runde 2 von 10 Runden
Spieler 1: Wie viele Kugeln von 90? <8>
Spieler 2: Wie viele Kugeln von 91? <8>
Spieler 3: Wie viele Kugeln von 95? <6>
Spieler 1 hat die Runde (mit) gewonnen und bekommt einen Punkt!
Spieler 2 hat die Runde (mit) gewonnen und bekommt einen Punkt!
Spielstand nach Runde 2:
Spieler 1 hat 2 Punkte und 82 Restkugeln
Spieler 2 hat 1 Punkte und 83 Restkugeln
Spieler 3 hat 0 Punkte und 89 Restkugeln
```

```
...

Spielstand nach Runde 10:
Spieler 1 hat 5 Punkte und 0 Restkugeln
Spieler 2 hat 4 Punkte und 0 Restkugeln
Spieler 3 hat 5 Punkte und 0 Restkugeln

Endstand:
Spieler 1 ist der bzw. einer der Sieger des Spiels!
Spieler 3 ist der bzw. einer der Sieger des Spiels!
```

Aufgabe 42:

Gegeben sei ein beliebig großes Array *zahlen* mit beliebigen int-Werten sowie ein positiver int-Wert *divisor*. Berechnen Sie die Anzahl an Teilfolgen von Elementen des Arrays (*n* aufeinanderfolgende Elemente, *n* >= 1), deren Summe durch *divisor* teilbar ist, und geben Sie die Anzahl aus.

Hinweis: Die gegebenen Zahlenwerte brauchen Sie nicht einlesen, sondern können sie im Programm in Form von Konstanten vorgeben.

Beispiel:

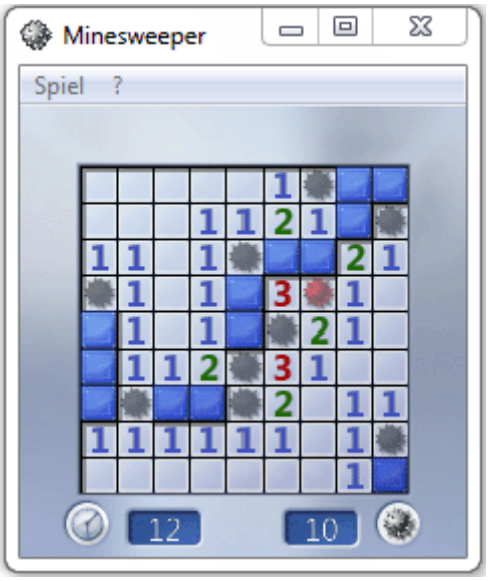
```
final int[] zahlen = { 2, 1, 2, 1, 1, 2, 1, 2 };
final int divisor = 4;

Ausgabe = 6
```

Aufgabe 43:

Sie kennen sicher das Spiel *Minesweeper*. Es ist ein Computerspiel, bei dem ein Spieler durch logisches Denken herausfinden muss, hinter welchen Feldern einer N * M-Matrix Mienen versteckt sind.

Während des Spiels wird beim Anklicken eines Feldes angezeigt, wie viele Mienen auf Nachbarfeldern platziert sind. Jedes Feld hat dabei 8 Nachbarfelder, Felder an den Rändern entsprechend weniger. Ziel des Spiels ist es, alle sicheren Felder zu finden. Sobald man auf ein Mienenfeld klickt, hat man verloren.



Bei dieser Aufgabe geht es nicht um die Implementierung des Spiels selber. Vielmehr wird in einer M*N-Ausgangsmatrix mit char-Zeichen der Anfangszustand eines Minesweeper-Spiels repräsentiert. Mienen werden dabei durch ein ‚*'-Zeichen und sichere Felder durch ein ‚.'-Zeichen repräsentiert. Ihre Aufgabe ist es, für diese gegebene Matrix die Minesweeper-Lösung zu finden und auf den Bildschirm auszugeben. Im folgenden Beispiel wird links eine 4*4-Ausgangsmatrix und rechts die zu berechnende und auszugebende Lösung skizziert.

```
*...    *100
....    2210
.*..    1*10
....    1110
```

Hinweis: Gehen Sie bei Ihrer Implementierung von einer im Quellcode vorgegebenen Ausgangsmatrix aus (die aber prinzipiell beliebig sein kann).

Beispiel 1:

```
public static void main(String[] args) {
     char[][] feld = { { '*', '.', '.', '.' },
                       { '.', '.', '.', '.' },
                       { '.', '*', '.', '.' },
                       { '.', '.', '.', '.' } };
     // Ihre Lösung

}
```

Produziert die folgende Bildschirmausgabe:

```
*100
2210
```

```
1*10
1110
```

Beispiel 2:

```
     public static void main(String[] args) {
          char[][] feld = { { '*', '*', '.', '.', '.' },
                            { '.', '.', '.', '.', '.' },
                            { '.', '*', '.', '.', '.' } };

          // Ihre Lösung

    }
```

Produziert die folgende Bildschirmausgabe:

```
**100
33200
1*100
```

Aufgabe 44:

Diese Aufgabe besteht in der Entwicklung einer Programmentwicklungsumgebung für die esoterische Programmiersprache *Brainfuck*.

Esoterische Programmiersprachen:

Esoterische Programmiersprachen sind Programmiersprachen, die nicht primär für den praktischen Einsatz entwickelt wurden, sondern ungewöhnliche Sprachkonzepte umsetzen. Sie reizen mit spektakulären und zuweilen sogar genialen Ideen, wie dem kleinstmöglichen Compiler, einer Arithmetik ohne Plus, Mal, Durch und Minus, dem kleinsten Symbolvorrat oder sogar der Unsichtbarkeit der Programme. Mit Esoterik im umgangssprachlichen Sinn haben "esoterische Programmiersprachen" allerdings nichts zu tun. Die Beschäftigung mit esoterischen Programmiersprachen kann zu tieferem Verständnis seriöser Programmiersprachen sowie zur Verbesserung strukturellen Denkens führen. Abhängig vom verfolgten Konzept können esoterische Programmiersprachen Konzepte für Sprachdesign und/oder Systemdesign demonstrieren. Link: http://de.wikipedia.org/wiki/Esoterische_Programmiersprache

Brainfuck:

Brainfuck ist eine esoterische Programmiersprache, die vom Schweizer Urban Müller um 1993 entwickelt wurde. Die Sprache wird manchmal auch *Brainf*ck*, *Brainf**** oder *BF* genannt. Link: http://de.wikipedia.org/wiki/Brainfuck

Lexikalik von Brainfuck:
Brainfuck-Quelltext darf beliebige Zeichen enthalten.

Syntax von Brainfuck:
Brainfuck besitzt acht Befehle, jeweils bestehend aus einem einzigen Zeichen:

`> < + - . , [ ]`

Andere im Quelltext vorkommende Zeichen (z. B. Buchstaben, Zahlen, Leerzeichen, Zeilenumbrüche) werden ignoriert und können so als Quelltextkommentar verwendet werden. Folgende EBNF definiert die Syntax von Brainfuck:

```
<BF-Programm> ::= { <Befehl> }
<Befehl>      ::= „>“ | „<“ | „+“ | „-“ | „.“ | „,“ | <Schleife>
<Schleife>    ::= „[“ <BF-Programm> „]“
```

Semantik von Brainfuck:
Grundlage von Brainfuck ist ein Speicher und ein Speicherelementzeiger. Im Falle der PVL wird der Speicher durch ein Array namens `speicher` mit 100 Elementen (Zellen) vom Typ `int` gebildet. Der Speicherelementzeiger namens `zeiger` ist eine Variable vom Typ `int`, in der letztendlich Indizes auf den Speicher verwaltet werden. Bei Start eines Programms werden die einzelnen Speicherelemente sowie der Zeiger mit dem Wert 0 initialisiert.

Die Semantik von Brainfuck ist dann folgendermaßen definiert:

>	`++zeiger;`	Inkrementiert den Zeiger
<	`--zeiger`	Dekrementiert den Zeiger
+	`++(speicher[zeiger]);`	Inkrementiert den Wert des aktuellen Speicherelementes
-	`--(speicher[zeiger]);`	Dekrementiert den Wert des aktuellen Speicherelementes
.	`IO.print((char)speicher[zeiger]);`	Gibt den Wert des aktuellen Speicherelementes als ASCII-Zeichen auf die Standardausgabe aus
;	`speicher[zeiger] = IO.readChar();`	Liest ein Zeichen von der Standardeingabe und speichert dessen ASCII-Wert im aktuellen Speicherelement
[	`while (speicher[zeiger] != 0) {`	Springt hinter den passenden]-Befehl, wenn sich im aktuellen Speicherelement der Wert 0 befindet
]	`}`	Sprint hinter den passenden [-Befehl zurück, wenn sich im aktuellen Speicherelement ein Wert ungleich 0 befindet

Beispiel (aus Wikipedia):
Das folgende Brainfuck-Programm gibt „Hello World!“ und einen Zeilenumbruch aus.

```
 ++++++++++
 [
  >+++++++>++++++++++>+++>+<<<<-
```

```
]                   Schleife zur Vorbereitung der Textausgabe
>++.                  Ausgabe von 'H'
>+.                   Ausgabe von 'e'
+++++++.              'l'
.                     'l'
+++.                  'o'
>++.                  Leerzeichen
<<+++++++++++++++.    'W'
>.                    'o'
+++.                  'r'
------.               'l'
--------.             'd'
>+.                   '!'
>.
```

Zur besseren Lesbarkeit ist dieser Brainfuckcode auf mehrere Zeilen verteilt und kommentiert worden. Brainfuck ignoriert alle Zeichen, die keine Brainfuckbefehle sind.

Zunächst wird die erste (die „nullte“) Zelle des Speichers auf den Wert 10 gesetzt (speicher[0] = 10). Die Schleife am Anfang des Programms errechnet dann mit Hilfe dieser Zelle weitere Werte für die zweite, dritte, vierte und fünfte Zelle. Für die zweite Zelle wird der Wert 70 errechnet, welcher nahe dem ASCII-Wert des Buchstaben 'H' (ASCII-Wert 72) liegt. Die dritte Zelle erhält den Wert 100, nahe dem Buchstaben 'e' (ASCII-Wert 101), die vierte den Wert 30 nahe dem Wert für Leerzeichen (ASCII-Wert 32), die fünfte den Wert 10, welches dem ASCII-Zeichen „Line Feed“ entspricht und als Zeilenumbruch interpretiert wird.

Die Schleife errechnet die Werte, indem einfach auf die zu anfangs mit 0 initialisierten Zellen 10-mal 7, 10, 3 und 1 addiert wird. Nach jedem Schleifendurchlauf wird speicher[0] dabei um eins verringert, bis es den Wert 0 hat und die Schleife dadurch beendet wird.

Am Ende der Schleife hat das Datenfeld dann folgende Werte:
speicher[0] = 0; speicher [1] = 70; speicher [2] = 100;
speicher [3] = 30; speicher [4] = 10;

Als nächstes wird der Zeiger auf die zweite Zelle des Datenfelds (speicher[1]) positioniert und der Wert der Zelle um zwei erhöht. Damit hat es den Wert 72, welches dem ASCII-Wert des Zeichens 'H' entspricht. Dieses wird daraufhin ausgegeben. Nach demselben Schema werden die weiteren auszugebenden Buchstaben mit Hilfe der durch die Schleife initialisierten Werte, sowie der bereits verwendeten Werte, errechnet.

Aufgabe:
Entwickeln Sie ein Java-Programm, das einen Brainfuck-Interpreter implementiert. Konkret soll das Programm folgendes tun:

1. Der Benutzer wird nach dem Namen einer Datei gefragt, die ein Brainfuck-Programm enthält.
2. Die Datei wird eingelesen.

3. Der eingelesene Brainfuck-Quelltext wird zeichenweise interpretiert, d.h. Befehl für Befehl ausgeführt.

Wird im interpretierten Brainfuck-Programm ein Syntaxfehler entdeckt, wird die Ausführung des Programms unmittelbar mit einer entsprechenden Fehlermeldung beendet. Dasselbe trifft für Laufzeitfehler zu. Ein Laufzeitfehler kann auftreten, wenn auf eine nicht vorhandene Speicherzelle (bspw. speicher[-1]) zugegriffen werden soll.

Überlegen Sie sich weiterhin zwei (kleinere) Probleme, zu deren Lösung Sie jeweils ein Brainfuck-Programm schreiben und testen Sie damit Ihren Brainfuck-Interpreter.

Hinweise:
Implementieren Sie den Brainfuck-Interpreter auf eine imperative Art und Weise. Benutzen Sie nur die Konzepte, die in der Vorlesung bereits besprochen wurden. Achten Sie auf eine saubere Strukturierung und aussagekräftige Bezeichner. Verständlichkeit Ihres Quellcodes ist wichtiger als Performance oder möglichst wenig Quellcode.

Aufgabe 45:

1. Schreiben Sie ein Programm, in dem ein int-Array erzeugt wird. Im Array soll das größte Element ermittelt und der entsprechende Wert auf den Bildschirm ausgegeben werden.
2. Schreiben Sie eine Funktion, die ein int-Array als Parameter übergeben bekommt. Die Funktion soll das größte Element im Array ermitteln und liefern. Schreiben Sie ein Testprogramm für die Funktion, in dem die Funktion aufgerufen und der ermittelte Wert auf den Bildschirm ausgegeben wird.
3. Schreiben Sie eine Funktion, die eine int-Matrix als Parameter übergeben bekommt. Die Funktion soll das größte Element der Matrix ermitteln und liefern. Schreiben Sie ein Testprogramm für die Funktion, in dem die Funktion aufgerufen und der ermittelte Wert auf den Bildschirm ausgegeben wird.

Aufgabe 46:

Teilaufgabe 1: Schreiben Sie eine Funktion, die ein int-Array als Parameter übergeben bekommt. Die Funktion soll die Reihenfolge der Werte umdrehen, Beispiel:

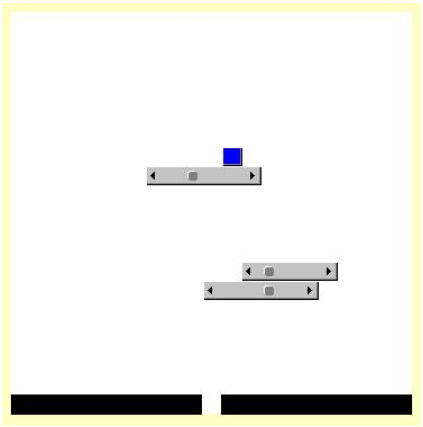
Array anfangs = {2, 4, 8, -3, 5} Array hinterher = {5, -3, 8, 4, 2}

Teilaufgabe 2: Dieselbe Aufgabe. Nur diesmal soll die Funktion das übergebene Array nicht verändern, sondern das entsprechende Array als Funktionswert liefern.

Aufgabe 47:

Das „Balkenspiel“ ist ein Logikspiel für einen Spieler. Es geht darum, durch geschicktes Verschieben von Balken einen Ball in ein Zielfeld fallen zu lassen. Sie können das Spiel gerne mal online ausprobieren:

http://www.kleine-onlinespiele.de/Balken/balken.htm



Entwickeln Sie ein Java-Programm, das es einem Benutzer erlaubt, an einer Konsole das Balkenspiel zu spielen.

In unserem Fall ist die Ausgangssituation in einer Textdatei abgespeichert. Der Ball wird durch ein „*“, die Balken durch Ziffern („0“ bis „9“), der Grund durch ein „+“ und alle anderen Felder durch ein Leerzeichen repräsentiert. Die Größe des Spielfeldes ergibt sich aus der Anzahl an Reihen und Spalten der Datei. Sie können davon ausgehen, dass die Datei immer korrekte Spielfelder enthält und alle Reihen gleich viele Spalten besitzen. Obiges Spielfeld würde bspw. in etwa durch folgende Datei repräsentiert:

```
   *
 1111

   222
  3333

++++ +++++
```

Lesen Sie die Datei mit Hilfe der Funktion `readFileAsCharMatrix` ein, wobei *dateiname* ein String ist, der die Datei kennzeichnet:

```
char[][] spielbrett = IO.readFileAsCharMatrix(dateiname);
```

Der Programmablauf sieht folgendermaßen aus:

Der Benutzer wird zunächst nach dem Namen einer Datei gefragt, die eine Ausgangssituation enthält. Die Datei wird eingelesen. Bis zum Spielende wird dann der Benutzer in einer Schleife aufgefordert, einen Balken und eine Richtung anzugeben und der angegebene Balken wird um ein Feld in die angegebene Richtung verschoben, wobei gegebenenfalls der Ball mitverschoben wird und eventuell runter fällt. Das Spielende ist erreicht, wenn der Ball die Lücke am unteren Rand erreicht.

Hinweise: Balken und der Ball können beliebig weit aber nicht über den Rand hinaus verschoben werden. Sie müssen nicht kontrollieren, ob es in einer bestimmten Situation überhaupt noch möglich ist, zu gewinnen.

Orientieren Sie sich, was den Programmablauf als auch was die Ein- und Ausgaben angeht, an folgendem Beispiel (Eingaben stehen in <>):

```
     *
   1111

      222
    3333

++++ +++++

Balken wählen: <1>
Richtung wählen (l=links, r=rechts): <l>


     *
  1111

      222
    3333

++++ +++++

Balken wählen: <1>
Richtung wählen (l=links, r=rechts): <l>



 1111

     *222
    3333

++++ +++++

Balken wählen: <2>
Richtung wählen (l=links, r=rechts): <l>



 1111

    *222
    3333

++++ +++++

Balken wählen: <3>
Richtung wählen (l=links, r=rechts): <r>



 1111

     222
```

```
     3333

++++*+++++

Hurra! Geschafft!
```

Aufgabe 48:

Bei der in dieser Aufgabe betrachteten Variante des Hangman- bzw. Galgenmännchen-Spiels spielt der Computer gegen einen menschlichen Spieler. Der Computer wählt zunächst ein geheimes Wort aus. Der Spieler muss dieses Wort dann innerhalb von 10 Spielrunden erraten. In jeder Runde teilt der Spieler dem Computer zunächst ein Zeichen mit. Der Computer verrät darauf hin, ob und wenn ja an welcher Stelle bzw. an welchen Stellen das Zeichen im geheimen Wort vorkommt.

Implementieren Sie bitte genau folgenden Algorithmus. Halten Sie sich dabei auch an die Ausgaben im unten stehenden Beispiel:

- Der Computer wählt ein geheimes Wort aus. Stellen Sie sich hierzu einfach vor, es würde folgende Funktion existieren, die ein solches geheimes Wort liefert:

  ```
  static String chooseWord()
  ```

- Es werden 10 Spielrunden gespielt (es sei denn, der Mensch hat vorher das Wort erraten). Der Ablauf jeder Spielrunde sieht so aus:
  - Der Computer gibt das bisher erratene Wort aus. Bisher nicht erratene Zeichen des geheimen Wortes werden dabei durch einen Unterstrich („_“) ersetzt.
  - Der Computer fordert den Mensch auf, ein Zeichen einzugeben.
  - Der Computer überprüft das Zeichen:
    - Hatte der Mensch das Zeichen vorher bereits schon mal eingegeben, wird er darauf hingewiesen („Dummkopf“).
    - Ist das Zeichen nicht im geheimen Wort enthalten, wird der Mensch darauf hingewiesen („Pech gehabt“).
    - Ansonsten wird das Zeichen als geraten vermerkt. Es wird weiterhin überprüft, ob damit das komplette geheime Wort erraten wurde. In diesem Fall wird der Mensch beglückwünscht und das Programm beendet.
- Sind die 10 Spielrunden vorbei und hat der Mensch das geheime Wort nicht erraten, wird eine entsprechende Meldung ausgegeben („Leider verloren“) und das Programm beendet.

Beispiel für einen möglichen Programmablauf. Benutzereingaben stehen in Klammern (<>). Das geheime Wort ist „Java“:

```
Runde 1. Bisher geraten: ____. Was wählst du für ein Zeichen?<a>
Runde 2. Bisher geraten: _a_a. Was wählst du für ein Zeichen?<k>
Pech gehabt! k kommt im Wort nicht vor!
Runde 3. Bisher geraten: _a_a. Was wählst du für ein Zeichen?<v>
Runde 4. Bisher geraten: _ava. Was wählst du für ein Zeichen?<k>
```

```
Dummkopf! k hast du schon mal getippt!
Runde 5. Bisher geraten: _ava. Was wählst du für ein Zeichen?<J>
Gratulation! Du hast das Wort 'Java' erraten!
```