Programmierkurs Java

Dr.-Ing. Dietrich Boles

Aufgaben zu UE 11 – Klassen und Objekte III

(Stand 22.03.2020)

Aufgabe 1:

Implementieren Sie eine Klasse *Student*. Die Klasse soll eine Methode *immatrikulieren* definieren, der das Alter eines Studenten als Parameter übergeben wird, sowie eine Methode *exmatrikulieren*, sowie eine dritte Methode, die bei ihrem Aufruf den Altersdurchschnitt aller gerade immatrikulierten Studenten liefert.

Aufgabe 2:

Schauen Sie sich folgendes Programm an:

```
public class UE11Aufgabe2 {

   public static void main(String[] args) {
      IO.println(new Int(8).add(IO.readInt()).add(-2).makeString());
   }
}
```

Aufgabe: Entwickeln Sie eine Klasse `Int`, so dass sich das Programm kompilieren lässt. Die Klasse `Int` soll dabei folgende Semantik haben:

- Dem Konstruktor wird ein int-Wert übergeben, der intern gespeichert werden soll.
- Bei Aufruf der Methode `add` für ein Int-Objekt soll der übergebene int-Wert zum internen Wert addiert und anschließend eine Referenz auf das Objekt selbst geliefert werden.
- Die Methode `makeString` soll so implementiert werden, dass der interne int-Wert in seiner String-Repräsentation geliefert wird.

Bei einer Benutzereingabe von 3 soll das Programm bspw. „9“ auf den Bildschirm ausgeben, bei einer Eingabe von -7 „-1“.

Aufgabe 3:

Definieren Sie eine Klasse zur Repräsentation eines UKW-Radios. Der Zustand des Radios ist gegeben durch eine Frequenz zwischen 87.5 und 108.0 MHz. Nach dem Erzeugen des Radio-Objektes beträgt die Frequenz 87.5 MHz. Die Frequenz kann in den angegebenen Frequenzen schrittweise um 0.5 MHz nach oben bzw. unten

verändert werden. Weiterhin stehen *n* Stationstasten zur Verfügung, auf denen man Frequenzen speichern bzw. gespeicherte Frequenzen wieder abrufen kann.

Insgesamt soll Ihre Klasse also die folgenden Methoden definieren:

- Einen Konstruktor, dem die Anzahl *n* an Stationstasten als Parameter übergeben wird
- Eine Methode, die die aktuelle Frequenz liefert
- Eine Methode zum Verringern der Frequenz um 0.5 MHZ
- Eine Methode zum Erhöhen der Frequenz um 0.5 MHZ
- Eine Methode, um die aktuelle Frequenz auf einer Stationstaste zu speichern
- Eine Methode, um die aktuelle Frequenz auf die Frequenz einer angegebenen Stationstaste einzustellen

Es steht folgendes Testprogramm *UKWRadioGUI* mit einer graphischen Oberfläche für Ihre Klasse zur Verfügung. Wenn Ihre Klasse korrekt implementiert ist und sich in das Testprogramm ohne Compilierfehler einbinden lässt, sollte beim Start des Testprogramms folgende interaktive GUI auf dem Bildschirm erscheinen.


```
import java.awt.Button;
import java.awt.Frame;
import java.awt.GridLayout;
import java.awt.Label;
import java.awt.Panel;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

public class UKWRadioGUI extends Frame {

     UKWRadioGUI() {
           super("UKWRadio");
           this.addWindowListener(new WindowCloseAction());
           UKWRadio radio = new UKWRadio(2);
           this.setLayout(new GridLayout(4, 1));
           Label anzeige = new Label("" + radio.getAktuelleFrequenz());
           Panel einstellenPanel = new Panel();
           Panel stationstaste1Panel = new Panel();
           Panel stationstaste2Panel = new Panel();
           this.add(anzeige);
           this.add(einstellenPanel);
           this.add(stationstaste1Panel);
           this.add(stationstaste2Panel);
```

```
            einstellenPanel.setLayout(new GridLayout(1, 2));
            Button verringern = new Button("Frequenz verringern");
            verringern.addActionListener(
                  new VerringernAction(anzeige, radio));
            Button erhoehen = new Button("Frequenz erhöhen");
            erhoehen.addActionListener(new ErhoehenAction(anzeige, radio));
            einstellenPanel.add(verringern);
            einstellenPanel.add(erhoehen);

            stationstaste1Panel.setLayout(new GridLayout(1, 3));
            Label name1 = new Label("Stationstaste 1");
            Button speichern1 = new Button("Frequenz speichern");
            speichern1.addActionListener(
                  new FrequenzSpeichernAction(anzeige,
                        radio, 0));
            Button einstellen1 = new Button("Frequenz einstellen");
            einstellen1.addActionListener(
                  new FrequenzEinstellenAction(anzeige,
                        radio, 0));
            stationstaste1Panel.add(name1);
            stationstaste1Panel.add(speichern1);
            stationstaste1Panel.add(einstellen1);

            stationstaste2Panel.setLayout(new GridLayout(1, 3));
            Label name2 = new Label("Stationstaste 2");
            Button speichern2 = new Button("Frequenz speichern");
            speichern2.addActionListener(
                  new FrequenzSpeichernAction(anzeige,
                        radio, 1));
            Button einstellen2 = new Button("Frequenz einstellen");
            einstellen2.addActionListener(
                  new FrequenzEinstellenAction(anzeige,
                        radio, 1));
            stationstaste2Panel.add(name2);
            stationstaste2Panel.add(speichern2);
            stationstaste2Panel.add(einstellen2);
            this.pack();
            this.setVisible(true);
      }

      public static void main(String[] args) {
            new UKWRadioGUI();
      }
}

class VerringernAction implements ActionListener {

      Label anzeige;
      UKWRadio radio;

      VerringernAction(Label anzeige, UKWRadio radio) {
            this.anzeige = anzeige;
            this.radio = radio;
      }

      @Override
      public void actionPerformed(ActionEvent ev) {
            this.radio.frequenzVerringern();
            this.anzeige.setText("" + this.radio.getAktuelleFrequenz());
      }

}
```

```
class ErhoehenAction implements ActionListener {

	Label anzeige;
	UKWRadio radio;

	ErhoehenAction(Label anzeige, UKWRadio radio) {
		this.anzeige = anzeige;
		this.radio = radio;
	}

	@Override
	public void actionPerformed(ActionEvent ev) {
		this.radio.frequenzErhoehen();
		this.anzeige.setText("" + this.radio.getAktuelleFrequenz());
	}
}

class FrequenzSpeichernAction implements ActionListener {

	Label anzeige;
	UKWRadio radio;
	int taste;

	FrequenzSpeichernAction(Label anzeige, UKWRadio radio, int taste) {
		this.anzeige = anzeige;
		this.radio = radio;
		this.taste = taste;
	}

	@Override
	public void actionPerformed(ActionEvent ev) {
		this.radio.frequenzSpeichern(this.taste);
		this.anzeige.setText("" + this.radio.getAktuelleFrequenz());
	}
}

class FrequenzEinstellenAction implements ActionListener {

	Label anzeige;
	UKWRadio radio;
	int taste;

	FrequenzEinstellenAction(Label anzeige, UKWRadio radio, int taste) {
		this.anzeige = anzeige;
		this.radio = radio;
		this.taste = taste;
	}

	@Override
	public void actionPerformed(ActionEvent ev) {
		this.radio.frequenzEinstellen(this.taste);
		this.anzeige.setText("" + this.radio.getAktuelleFrequenz());
	}
}

class WindowCloseAction extends WindowAdapter {
	@Override
	public void windowClosing(WindowEvent e) {
		System.exit(0);
	}

}
```

Aufgabe 4:

Gegeben seien folgende Java-Klassen:

```
class Autor {

     // liefert den Namen des Autors
     String getName()
}

class Buch {

     // liefert die ISBN-Nummer des Buches
     String getISBN()

     // liefert den Titel des Buches
     String getTitel()

     // liefert die Autoren des Buches
     Autor[] getAutoren()

     // liefert den Verkaufspreis des Buches
     double getPreis()
}

class Buchhandlung {

     // Klassenmethode, die zum Namen einer Buchhandlung das
     // entsprechende Buchhandlung-Objekt liefert;
     // bei ungültigem Namen wird null geliefert
     static Buchhandlung getBuchhandlung(String name)

     // liefert die Anzahl an verkauften Exemplaren in dieser
     // Buchhandlung für das angegebene Buch
     int getAnzahlVerkaufteExemplare(Buch buch

     // liefert die Bücher, die die Buchhandlung im Sortiment hat
     Buch[] getBuecherImSortiment()
}
```

Schreiben Sie auf der Basis dieser Klassen ein Programm, das folgendes tut:

Zunächst wird der Nutzer nach dem Namen einer Buchhandlung und dem Namen eines Autors gefragt. Anschließend berechnet das Programm, wie viele Tantiemen der angegebene Autor für seine Bücher bekommt, die in der angegebenen Buchhandlung verkauft worden sind, und gibt die Summe auf den Bildschirm aus. Als Tantiemen bekommen die Autoren eines Buches dabei 10 Prozent des Buchpreises jedes verkauften Exemplars, die Sie evtl. (gleichmäßig) miteinander teilen müssen.

Aufgabe 5:

Gegeben seien folgende Java-Klassen:

```
class Professor {
     // liefert den Namen des Professors
     String getName()
}
```

```
class Student {

     // liefert den Namen des Studenten
     String getName()

     // liefert den Studenten mit der angegebenen Matrikelnummer
     static Student getStudent(String matrikelnummer)
}

class PruefungsErgebnis {
     // liefert den Namen des Moduls, in dessen Rahmen die Pruefung
stattgefunden
     // hat
     String getModulname()

     // liefert alle Professoren, die fuer die Pruefung zustaendig waren
     Professor[] getProfessoren()

     // liefert den Studenten mit diesem Pruefungsergebnis
     Student getStudent()

     // liefert die Note der Pruefung
     int getNote()
}

class PruefungsDB {

     // liefert die Pruefungsdatenbank
     static PruefungsDB getPruefungsDB()

     // liefert alle Pruefungsergebnisse des angegebenen Studenten
     PruefungsErgebnis[] getNoten(Student student)
}
```

Schreiben Sie auf der Basis dieser Klassen ein Programm, das folgendes tut:

Zunächst wird der Nutzer nach dem Namen eines Professors und der Matrikelnummer eines Studierenden gefragt. Anschließend berechnet das Programm den Notendurchschnitt des entsprechenden Studierenden bei Prüfungen mit dem entsprechenden Professor und gibt den Notendurchschnitt auf den Bildschirm aus.

Aufgabe 6:

Ein ASCII-Bildschirm im Sinne dieser Aufgabe ist ein rechteckiges Gebilde, das aus einer Menge an Kacheln besteht. In diese Kacheln können char-Zeichen ausgegeben werden.

Die folgende Klasse `Kachel` repräsentiert entsprechende Bildschirm-Kacheln.

```
// Repaesentiert eine Zelle eines ASCII-Bildschirms
class Kachel {

     int reihe;
     int spalte;

     // initialisiert die angegebene Kachel mit einem Leerzeichen
     Kachel(int reihe, int spalte) {
```

```
            this.reihe = reihe;
            this.spalte = spalte;
           // Implementierung hier unwichtig
      }

      // Zeichnet das uebergebene Zeichen in die entsprechende Kachel
      void zeichne(char zeichen) {
            // Implementierung hier unwichtig
      }
}
```

Die folgende Klasse `Bildschirm` repräsentiert entsprechende Bildschirme.

```
// Repraesentiert einen rechteckigen ASCII-Bildschirm, der aus
// einzelnen Kacheln besteht
class Bildschirm {

     // erzeugt/initialisiert einen Bildschirm der angegebenen
     // Groesse
     Bildschirm(int anzahlReihen, int anzahlSpalten) {     }

     // liefert die Kacheln der angegebenen Reihe
     // Hinweis: Sie muessen sich nicht um Index-Fehler kuemmern
     Kachel[] getReihe(int reihenIndex) {  }

     // liefert die Kacheln der angegebenen Spalte
     // Hinweis: Sie muessen sich nicht um Index-Fehler kuemmern
     Kachel[] getSpalte(int spaltenIndex) {     }
```

Teilaufgabe (a):

Implementieren Sie die Klasse *Bildschirm*, d.h. definieren Sie geeignete interne Datenstrukturen sowie implementieren Sie die aufgeführten Konstruktoren und Methoden.

Teilaufgabe (b):

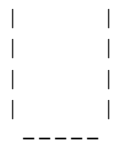
Entwickeln Sie auf der Grundlage der Klassen *Kachel* und Bildschirm ein *Programm*, das ein Rechteck der Größe *n* (*n* ist eine int-Konstante mit Werten größer 0) auf einen Bildschirm mit 24 Zeilen und 80 Spalten zeichnet (beginnend in der linken oberen Ecke des Bildschirms). Dabei dürfen nur die Konstruktoren und Methoden der Klassen genutzt werden. Der Zugriff auf in den Klassen definierte Attribute ist nicht erlaubt.

Beispiel (n = 2):

```
  --
 |  |
 |  |
  --
```

Beispiel (n = 5):

```
  -----
 |     |
```



Aufgabe 7:

Gegeben seien folgende Klassen:

```
class FachNote {
	private String fach; // das Fach (Deutsch, Mathe, ...)
	private int note; // die Note in dem Fach (1 .. 6)

	FachNote(String fach, int note) {
		this.fach = fach;
		this.note = note;
	}

	String getFach() {
		return this.fach;
	}

	int getNote() {
		return this.note;
	}
}

class Schueler {
	String name; // Name des Schuelers
	FachNote[] noten; // Noten in den einzelnen Faechern

	Schueler(String name, FachNote[] noten) {
		this.name = name;
		this.noten = noten;
	}

	String getName() {
		return this.name;
	}

	FachNote[] getNoten() {
		return this.noten;
	}
}
```

Implementieren Sie auf der Grundlage dieser Klassen die folgende Funktion:

```
	/**
	 * Berechnet die Durchschnittsnote der uebergebenen Schueler in dem
	 * uebergebenen Fach. Hinweise: nicht alle Schueler muessen dieselben
	 * Faecher belegen. Sie koennen davon ausgehen, dass alle Faecher
	 * auch durch mindestens einen Schueler belegt werden.
	 *
	 * @param schueler Menge an Schuelern (!= null)
	 * @param fach Name des Faches (!= null)
	 * @return die Durchschnittsnote der uebergebenen Schueler in dem
	 *         uebergebenen Fach
	 */
	static double durchschnittsnote(Schueler[] schueler, String fach)
```

Aufgabe 8:

Beim Online-Banking muss sich der Bankkunde zunächst mit seiner Kontonummer und einer persönlichen Identifikationsnummer (PIN) am Bankrechner authentifizieren. Für das Auslösen sicherungspflichtiger Aufträge (z. B. einer Überweisung) muss außerdem eine Transaktionsnummer (TAN) übermittelt werden. Diese wird aus einer Liste von 100 TANs entnommen, die dem Kunden vorher zugeschickt und durch ihn freigegeben wurde.

An die Transaktionsnummern werden folgende Bedingungen gestellt:

- Eine TAN ist eine zufällige 6-ziffrige Zahl zwischen 100000 und 999999.
- Jede TAN kommt nur einmal vor.
- Jede TAN darf nur einmal verwendet werden.

Folgendes ist eine gültige TAN-Liste:

```
963002 701837 543423 239150 526763 924103 396054 411076 977108 857169
255549 921595 447899 626258 781718 527762 802132 157715 797227 781895
821065 172984 527924 801503 820360 993706 631421 520026 454031 869184
914790 647546 166143 805788 514630 531489 548688 699481 397882 154254
138872 724491 488188 712325 415943 665750 728851 511410 248348 253840
626526 370290 910463 951473 554520 500185 608195 761195 107366 297942
100456 958189 774141 589685 361569 620076 901807 299068 598361 838014
378822 368426 238388 835745 121001 578457 852160 497733 155579 560223
219681 835335 844747 808295 524834 683175 212941 301816 292159 620207
297252 120304 248315 410165 332899 947068 713189 278126 422161 983132
```

Aufgabe: Implementieren Sie eine Klasse `TANListe` zur Verwaltung entsprechender Transaktionsnummern.

Überlegen Sie sich zunächst eine Datenstruktur zur Speicherung von 100 TANs. Dabei sollen sowohl der Wert jeder TAN als auch ihr Zustand (verbraucht/nicht verbraucht) erfasst werden.

Implementieren Sie dann die folgenden Methoden:

1. Einen *Konstruktor*, in dem die 100 nicht-verbrauchten TANs der TAN-Liste erzeugt und gespeichert werden.
2. Eine boolesche Methode `verbrauchen` mit einem int-Wert als Parameter. Die Methode soll prüfen, ob (im Parameter) eine gültige, nicht verbrauchte TAN der TAN-Liste übermittelt wurde. In diesem Fall soll die TAN in der Liste als verbraucht markiert sowie der Rückgabewert `true` geliefert werden. In allen anderen Fällen soll `false` zurückgegeben werden.
3. Eine Methode `print`, durch die die noch nicht verbrauchten TANs der TAN-Liste auf den Bildschirm ausgegeben werden.

Aufgabe 9:

Langtons Ameise ist eine Turingmaschine mit einem zweidimensionalen Speicher und wurde 1986 von Christopher Langton entwickelt. Sie ist ein Beispiel dafür, dass ein deterministisches (das heißt nicht zufallsbedingtes) System aus einfachen Regeln

sowohl für den Menschen visuell überraschend ungeordnet erscheinende als auch regelmäßig erscheinende Zustände annehmen kann. (Quelle: Wikipedia)

Ausgangspunkt:

Gegeben sei eine quadratische 2-dimensionale Welt mit SIZE * SIZE Zellen. SIZE (hier SIZE = 200) sei die Anzahl der Reihen und Spalten. Die Zellen können zwei Zustände einnehmen (weiß und schwarz). Anfangs sind alle Zellen weiß. In dieser Welt lebt genau eine Ameise. Anfangs befindet sich die Ameise auf der Zelle in der Mitte der Welt und schaut nach Norden. Nun wird STEPS (hier STEPS = 40000) mal folgender „Ameisenalgorithmus“ ausgeführt:

- Befindet sich die Ameise auf einer weißen Zelle, so färbt sie sie schwarz, dreht sich um 90 Grad nach rechts und begibt sich auf die nächste Zelle in der neuen Blickrichtung.
- Befindet sich die Ameise auf einer schwarzen Zelle, so färbt sie sie weiß, dreht sich um 90 Grad nach links und begibt sich auf die nächste Zelle in der neuen Blickrichtung.

Die Welt sei dabei ein Torus, d.h. wenn die Ameise die Welt nach oben verlässt, erscheint sie wieder unten und umgekehrt; wenn sie die Welt nach links verlässt, erscheint sie wieder rechts und umgekehrt.

Vorgabe:

Gegeben Sei folgendes Programm:

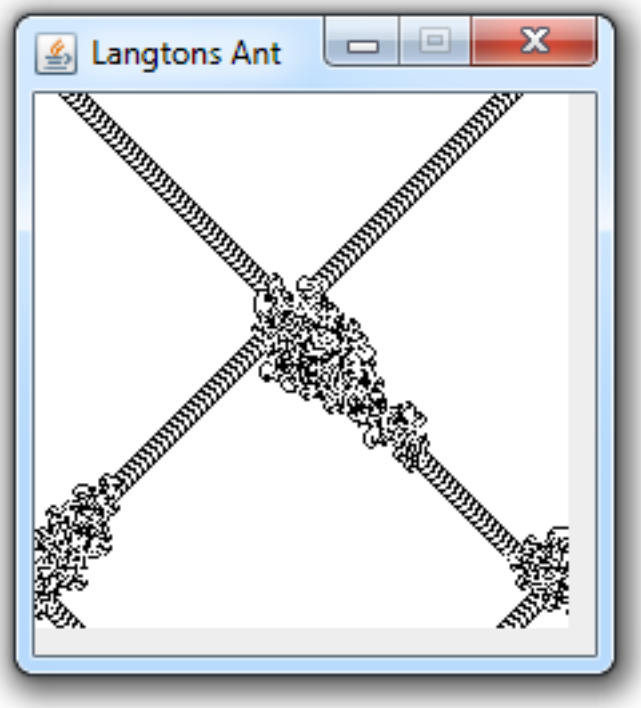
```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.image.BufferedImage;

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

public class LangtonAnt {

    private static final int SIZE = 200;
    private static final int STEPS = 40000;

    public static void main(String[] args) {
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            @Override
            public void run() {
                // erzeugt eine Welt der Groesse SIZE * SIZE
                World world = new World(LangtonAnt.SIZE);

                AntWorldPanel worldPanel =
                    new AntWorldPanel(world);
                AntFrame frame = new AntFrame(worldPanel);
                frame.setLocation(100, 100);
                frame.setResizable(false);
                frame.setVisible(true);
                AntSimulation simulation = new AntSimulation(world,
                        LangtonAnt.STEPS, worldPanel);
                simulation.start();
            }
        });
```

```
	}
}

class AntSimulation extends Thread {

	private World world;
	private int steps;
	private AntWorldPanel panel;

	AntSimulation(World world, int steps, AntWorldPanel panel) {
		this.world = world;
		this.steps = steps;
		this.panel = panel;
	}

	@Override
	public void run() {
		try {
			for (int i = 0; i < this.steps; i++) {

				// holt sich die Ameise aus der Welt
				Ant ant = this.world.getAnt();

				// fuehrt einmal den Ameisenalgorithmus aus
				ant.nextStep();

				if (i % 30 == 0) {
					javax.swing.SwingUtilities.invokeAndWait(
						new Runnable() {
						@Override
						public void run() {
						AntSimulation.this.panel.repaint();
						}
					});
				}
			}
		} catch (Exception exc) {
			exc.printStackTrace();
		}

	}
}

class AntFrame extends JFrame {

	AntFrame(AntWorldPanel world) {
		super("Langtons Ant");
		this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		this.setLayout(new BorderLayout());
		this.add(world);
		this.pack();
	}
}

class AntWorldPanel extends JPanel {

	private World world;
	private BufferedImage image;

	AntWorldPanel(World world) {
		this.world = world;
		this.setPreferredSize(new Dimension(world.getSize(),
							world.getSize()));
```

```
            this.image = new BufferedImage(world.getSize(),
                                           world.getSize(),
                                           BufferedImage.TYPE_INT_ARGB);
      }

      @Override
      protected void paintComponent(Graphics g) {
            super.paintComponent(g);

            Graphics buffer = this.image.getGraphics();
            buffer.setColor(Color.WHITE);
            buffer.fillRect(0, 0, this.image.getWidth(),
                            this.image.getHeight());

            for (int row = 0; row < this.world.getSize(); row++) {
                  for (int column = 0;
                        column < this.world.getSize(); column++) {

                        // liefert true, wenn die Zelle in Zeile row und
                        // Spalte column sich im Zustand schwarz befindet
                        if (this.world.isCellBlack(row, column)) {
                              buffer.setColor(Color.BLACK);
                        } else {
                              buffer.setColor(Color.WHITE);
                        }
                        buffer.fillRect(column, row, 1, 1);
                  }
            }
            g.drawImage(this.image, 0, 0, null);

      }
}
```

Aufgabe:

Implementieren Sie die fehlenden Klassen `World` und `Ant`, die die Welt bzw. die Ameise repräsentieren. Wenn Sie alles korrekt gemacht haben, sollte innerhalb einiger Sekunden nach und nach das folgende Fenster auf dem Bildschirm erscheinen.



Hinweise:

- Sie brauchen den vorgegebenen Code nicht komplett zu verstehen; schauen Sie sich nur die Stellen an, an denen die Klassen `World` bzw. `Ant` genutzt werden. Hier stehen entsprechende Kommentare. Insbesondere brauchen Sie aus den Klassen `World` und `Ant` nicht auf die gegebenen Klassen zugreifen!
- Sie dürfen die vorgegebenen Klassen nicht ändern!

Aufgabe 10:

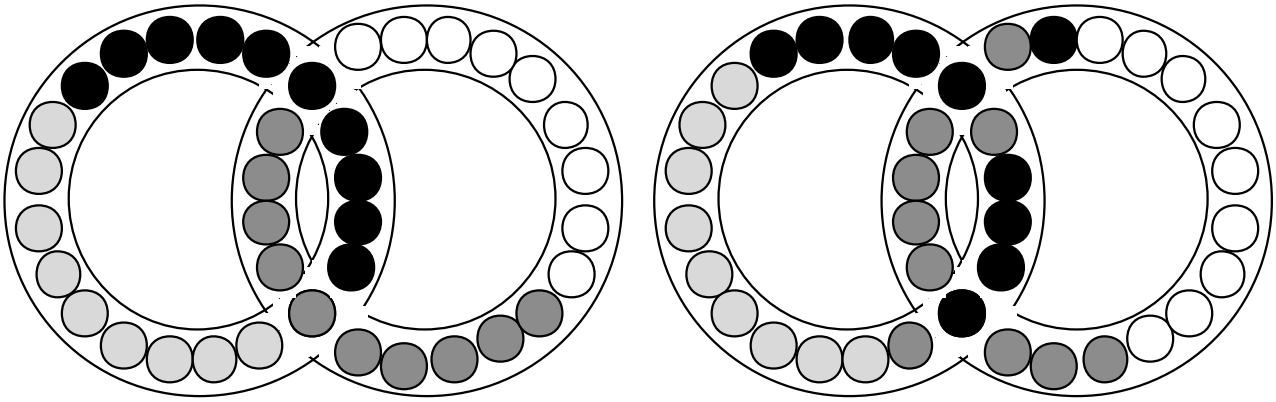
Rubik`s Ringe enthalten 10 schwarze, 9 gelbe, 10 rote und 9 graue Kugeln. Die Kugeln eines Ringes können linksdrehend oder rechtsdrehend verschoben werden. Man kann jeden Ring einzeln drehen. Durch die beiden Schnittpositionen können Kugeln von einem Ring in den anderen übergehen.

Beispiel:

Der Zustand in der Abbildung rechts ist aus dem Zustand links entstanden durch:

- Drehung des rechten Ringes um zwei Kugeln nach rechts (im Uhrzeigersinn),
- Drehung des linken Ringes um eine Kugel ebenfalls nach rechts.

Ziel des Spiels ist, durch Drehen der Ringe aus einer bestimmten Anordnung der Kugeln (Anfangszustand) eine andere Anordnung der Kugeln (Endzustand) herzustellen.



Schreiben Sie ein Java-Programm, das zunächst einen Anfangszustand einliest. Anschließend wird (in einer Endlosschleife) jeweils ein Spielzug eingelesen und der erreichte Zustand auf dem Bildschirm dargestellt (kreisförmige Darstellung der Ringe ist **nicht** erforderlich!). Eine Überprüfung der aktuellen Anordnung der Kugeln mit einem Endzustand ist **nicht** erforderlich!

Sie sollen das Rubik-Spiel auf eine objektorientierte Art und Weise implementieren. Entwerfen Sie daher geeignete Klassen **RubikSpiel**, **Ring**, **Kugel** und **Spielzug!**

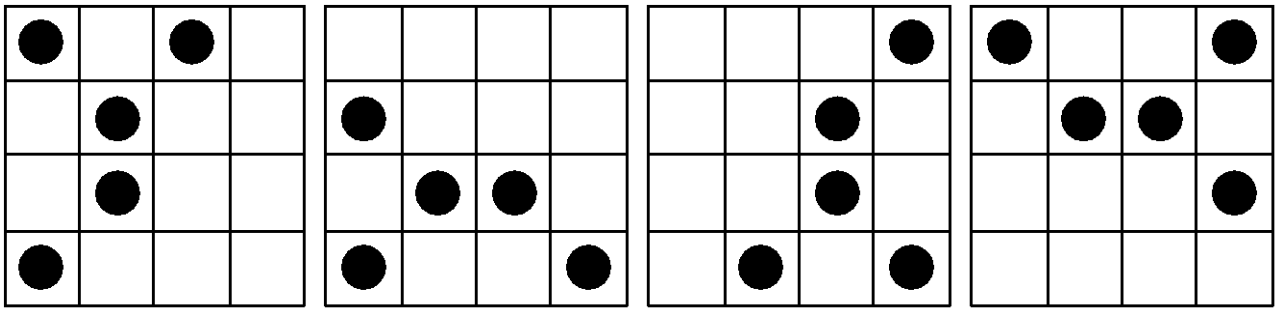
Mögliche Vorgehensweise:

- Entwerfen Sie zunächst die Klassen **Ring** und **Kugel**. Überlegen Sie sich dazu eine geeignete Datenstruktur zur Repräsentation der Ringe. Bedenken Sie dabei: Es gibt zwei kreisförmig zu organisierende Mengen (Ringe) mit jeweils 20 Kugeln, wobei sich die beiden Ringe jedoch zwei Kugeln teilen.
- Entwerfen Sie dann die Klasse **Spielzug**. Überlegen Sie sich dazu eine geeignete Datenstruktur zur Repräsentation von Spielzügen.
- Implementieren Sie die Ein- und Ausgabemethoden.
- Implementieren Sie den Drehungsalgorithmus.
- Implementieren Sie das Rahmenprogramm (Klasse **RubikSpiel**).

Aufgabe 11:

Puan-Puan ist ein Spiel, das von zwei Personen auf einem Spielbrett von n mal n Feldern (n = 2, 3, 4, ..., 9) gespielt wird. Als Spielfiguren stehen n mal n gleichfarbige Steine zur Verfügung. Die Spieler ziehen abwechselnd. Bei einem Zug wird entweder ein Stein gesetzt oder ein bereits gesetzter Stein weggenommen.

Wer eine Steinstellung erzeugt, die schon einmal im Verlauf des aktuellen Spiels aufgetreten ist oder die bis auf eine Drehung des gesamten Spielfeldes um 90, 180 oder 270 Grad mit einer bereits aufgetretenen Stellung übereinstimmt, verliert das Spiel. Das Spiel endet, wenn einer der beiden Spieler verliert bzw. nach 100 Zügen. In letzterem Falle endet das Spiel mit einem Unentschieden.



Vier (bis auf Drehung) gleiche Stellungen („Puan-Gleichheit“)

Schreiben Sie ein Java-Programm, das zuerst die Größe des Spielbrettes einliest und dann abwechselnd die Spielzüge beider Spieler entgegennimmt, bis das Spiel endet.

Implementieren Sie das Spiel auf objektorientierte Art und Weise, d.h. definieren Sie geeignete Klassen insbesondere zur Repräsentation des Spielbrettes, der Spielfiguren, der Spielzüge, der Spieler, der Spielregeln und des Spielablaufes.

Aufgabe 12:

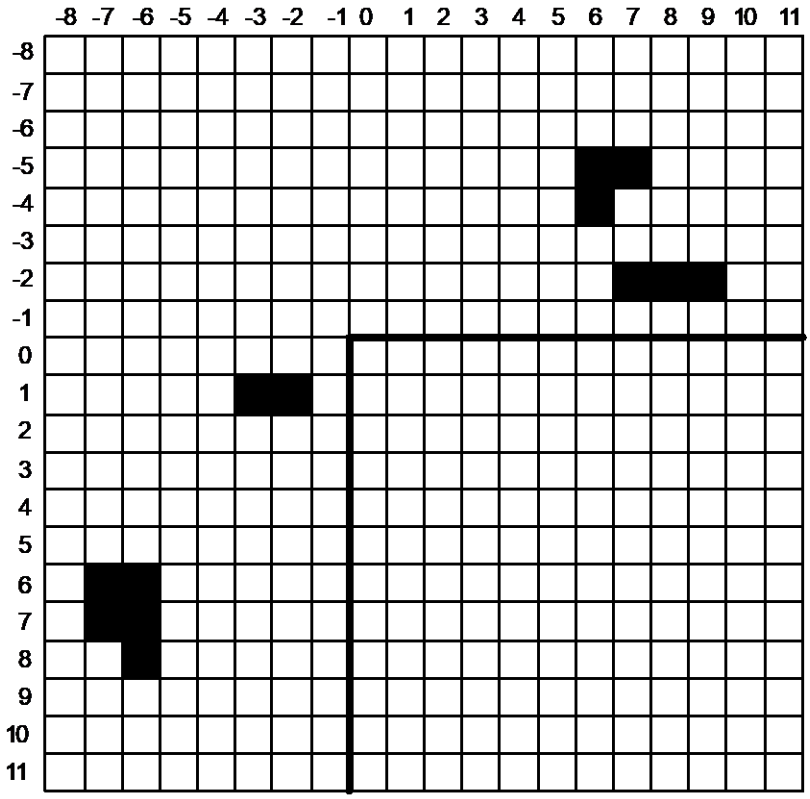
Quadratien ist ein quadratisches Gebiet aus quadratischen Feldern. Das Feld in der Nordwest-Ecke hat die Zeilennummer 0 und die Spaltennummer 0.

Das Wetter in Quadratien wird durch quadratische Wolken bestimmt, die genau ein Feld groß sind. Solche Wolken rücken getaktet über Quadratien vor, und zwar von Norden nach Süden 2 Felder pro Takt und - in einer anderen Höhe - von Westen nach Osten 3 Felder pro Takt. Es regnet überall dort, wo sich nach einem Vorrücken sowohl eine Nord-Süd- als auch eine West-Ost-Wolke befinden. Wolken, aus denen es regnet, lösen sich auf.

Die Wolkenvorhersage gibt an, an welchen Stellen (in der Form: Zeilennummer/Spaltennummer) sich zum aktuellen Zeitpunkt Wolken befinden. Daraus läßt sich dann ermitteln, wo es in Quadratien regnen wird; denn es werden nur solche Wolken angegeben, die über Quadratien hinwegziehen werden.

Beispiel: Wolkenvorhersage:

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




```
Es wird an folgenden Stellen regnen: 1/6, 1/7, 8/9
```

Aufgabe: Entwickeln Sie ein Java-Programm, welches folgendes leistet:

- Einlesen der Größe von Quadratien (Anzahl Zeilen bzw. Spalten).
- Einlesen einer Wolkenvorhersage.
- Ausgabe, wo in Quadratien Regen fällt.

Besondere Anforderungen: Bei der Programmentwicklung sind folgende Anforderungen zu berücksichtigen:

- Wolken dürfen in der Wettervorhersage nur links bzw. oberhalb von Quadratien erzeugt werden. Unter Einhaltung dieser Restriktion dürfen (prinzipiell) beliebig positionierte Wolken eingegeben werden. Eine Beschränkung der Größe des Umlandes von Quadratien ist nicht erlaubt.
- Die Menge der Wolken in der Wettervorhersage ist (prinzipiell) beliebig groß. Es darf vorweg keine Größenbeschränkung angegeben werden.
- Entwickeln Sie eine Klasse Wolke, die Wolken realisiert. Überlegen Sie sich notwendige Attribute (Zeile, Spalte, ...) und Methoden (Überprüfung auf gleichePosition, vorrücken, ...) der Klasse und implementieren Sie diese.
- Entwickeln Sie eine Klasse Wolkenmenge, die eine prinzipiell beliebig große Menge von Wolken verwalten kann und auf der Wettersimulationen möglich sind. Überlegen Sie sich notwendige Attribute (Realisierung als verkettete Liste, ...) und Methoden (Einfügen von Wolken, Entfernen von Wolken, Wettersimulation, ...) der Klasse und implementieren Sie diese.

- Entwickeln Sie das Hauptprogramm. Sie können dabei bspw. folgendermaßen vorgehen: Erzeugen Sie ein Wolkenmenge-Objekt. Erzeugen Sie während des Einlesens der Wolkenvorhersage jeweils ein neues Wolke-Objekt für jede Wolke und fügen Sie dieses in das Wolkenmenge-Objekt ein. Lassen Sie nach Beendigung der Wolkenvorhersage auf dem Wolkenmenge-Objekt eine Wettersimulation ausführen bis keine Wolken mehr da sind (abgeregnet oder über Quadratien hinweg getrieben). Speichern Sie während der Simulation die Regenwolken in einem zweiten Wolkenmenge-Objekt ab. Geben Sie nach der Simulation die Wolke-Objekte, die sich im zweiten Wolkenmenge-Objekt befinden, auf den Bildschirm aus**.**

Beispiel für die Benutzerschnittstelle des Programms:

```
> java Quadratien
Geben Sie bitte die Größe von Quadratien ein:
> 12
Geben Sie bitte die Wettervorhersage ein:
weitere Wolke (y/n)
> y
Zeile angeben:
> 0
Spalte angeben:
> -1
weitere Wolke (y/n)
> y
Zeile angeben:
> -2
Spalte angeben:
> 2
weitere Wolke (y/n)
> n
Es wird an folgenden Stellen regnen: 0/2.
```

Aufgabe 13:

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. Eine Zelle x, die nicht am Spielfeldrand liegt, hat 8 Nachbarzellen, Zellen am Spielfeldrand entsprechend weniger.

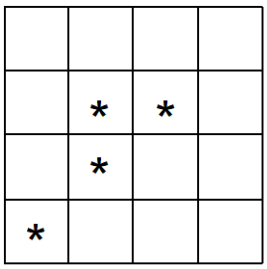
Die Regeln, nach denen sich die Population von einer Stellung zur nächsten entwickelt, sind:

- Für eine Zelle x, die gerade tot ist, gilt: Wenn x genau 3 lebende Nachbarzellen hat, wird x neu geboren; sonst bleibt x tot.
- 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.

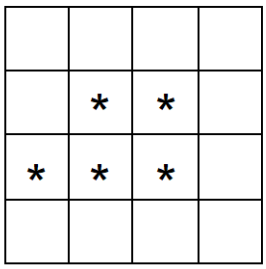
Alle Veränderungen gemäß dieser Regeln geschehen gleichzeitig. Die Simulation beginnt mit einer bestimmten eingelesenen Verteilung von lebenden und toten Zellen. Sie endet nach einer bestimmten eingelesenen Anzahl von

Entwicklungsschritten, jedoch früher, wenn sich die Population nicht mehr ändert. Das Spielfeld sollte aus mindestens 15x15 Feldern bestehen.

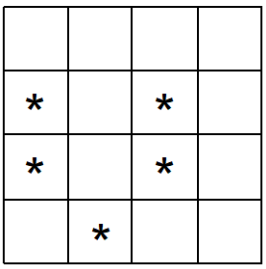
Beispiel:



Population 1



Population 2



Population 3

Aufgabe: Implementieren Sie das „Game-of-Life“ mit objektorientierten Mitteln, d.h. definieren Sie geeignete Klassen zur Repräsentation der Populationen und der Simulation.

Aufgabe 14:

In dieser Aufgabe sollen Sie Umfragen simulieren. Zum Hintergrund der Simulation: Eine Umfrage besteht aus mehreren Fragen. Bei den Fragen soll es sich ausschließlich um Fragen handeln, die mit Ja oder Nein beantwortet werden können. An einer Umfrage können beliebig viele Personen teilnehmen. Sie bekommen die Fragen gestellt und müssen mit Ja oder Nein antworten.

Die Simulation soll folgendermaßen ablaufen:

- Zunächst werden die einzelnen Fragen eingegeben.
- Anschließend wird die Umfrage durchgeführt, d.h. mehreren Umfrageteilnehmern werden die einzelnen Fragen gestellt, die diese beantworten müssen.
- Abschließend werden die Umfrageergebnisse auf den Bildschirm ausgegeben. Konkret wird für jede Frage die absolute und prozentuale Anzahl an Ja- und Nein-Antworten ausgegeben.

Aufgabe: Führen Sie eine objektorientierte Entwicklung eines entsprechenden Java-Programms durch. Überlegen Sie zunächst: Was für Objekte bzw. Klassen lassen sich identifizieren, welche Beziehungen existieren zwischen den Objekten, was für Eigenschaften und Funktionen besitzen die Objekte. Implementieren Sie anschließend die Klassen sowie die eigentliche Simulation.

Ablaufbeispiel:

```
Fragen eingeben
---------------
Titel der Umfrage: PK-Java-Umfrage
Anzahl Fragen: 2
Frage 1: Sind die Vorlesungen
verstaendlich?
Frage 2: Sind die Uebungsaufgaben
zu schwer?

Weiterer Teilnehmer (j/n)?j
Sind die Vorlesungen verstaendlich?
ja/nein (j/n)?n
Sind die Uebungsaufgaben zu schwer?
ja/nein (j/n)?n
Weiterer Teilnehmer (j/n)?n

Umfrageergebnisse
```

```
Umfrage
--------
Weiterer Teilnehmer (j/n)?j
Sind die Vorlesungen verstaendlich?
ja/nein (j/n)?j
Sind die Uebungsaufgaben zu schwer?
ja/nein (j/n)?n
Weiterer Teilnehmer (j/n)?j
Sind die Vorlesungen verstaendlich?
ja/nein (j/n)?j
Sind die Uebungsaufgaben zu schwer?
ja/nein (j/n)?n
```

```
-----------------
Umfrage: PK-Java-Umfrage
Frage:    Sind    die    Vorlesungen
verstaendlich?
ja-Antworten: 2 = 66.66666666666667
Prozent
nein-Antworten:         1         =
33.333333333333336 Prozent
Frage: Sind die Uebungsaufgaben zu
schwer?
ja-Antworten: 0 = 0.0 Prozent
nein-Antworten: 3 = 100.0 Prozent
```

Aufgabe 15:

In dieser Aufgabe sollen Sie EMail-Verkehr simulieren. Zum Hintergrund der Simulation: Personen können sich gegenseitig EMails zuschicken. Jede Person hat einen Namen, eine EMail-Adresse und jeder Person ist eine Mailbox zugeordnet, in der empfangene EMails gespeichert werden. Jede EMail besitzt neben dem Sender und Empfänger (nur einer, keine Gruppen!) ein Subject und den eigentlichen Inhalt.

Die Simulation soll folgendermaßen ablaufen:

- Zunächst melden sich eine Menge von Personen mit Name und EMail-Adresse an.
- Anschließend wird der EMail-Verkehr simuliert, d.h. es werden eine Menge von EMails geschrieben und verschickt. Dabei gilt: Bekannt sind nur jeweils die Namen der Personen, an die eine EMail verschickt werden soll, nicht die EMail-Adresse selbst!
- Abschließend soll es (in einer Schleife) möglich sein, für eine anzugebende Person (Name) die erhaltenen EMails auf den Bildschirm auszugeben.

Aufgabe: Führen Sie eine objektorientierte Entwicklung eines entsprechenden Java-Programms durch. Überlegen Sie zunächst: Was für Objekte bzw. Klassen lassen sich identifizieren, welche Beziehungen existieren zwischen den Objekten, was für Eigenschaften und Funktionen besitzen die Objekte. Implementieren Sie anschließend die Klassen sowie die eigentliche Simulation.

Ablaufbeispiel:

```
Anmeldung
---------
Personenanzahl: 2
Name: dibo
EMail:         boles@informatik.uni-
oldenburg.de
Name: Klaus
EMail: klaus@uni-oldenburg.de

EMailverkehr
------------
Weitere EMail verschicken (j/n)?j
Sendername: dibo
Empfaengername: Klaus
Subject: Hallo
Text: nur ein Gruss von dibo
```

```
Mailboxabfrage
--------------
Weitere Mailboxabfrage (j/n)?j
Name: dibo
Mailbox ist leer
Weitere Mailboxabfrage (j/n)?j
Name: Klaus
****** Begin EMail *****
TO: klaus@uni-oldenburg.de(Klaus)
FROM:          boles@informatik.uni-
oldenburg.de(dibo)
SUBJECT: Hallo

nur ein Gruss von dibo
****** End EMail *****
****** Begin EMail *****
```

```
Weitere EMail verschicken (j/n)?j
Sendername: dibo
Empfaengername: Klaus
Subject: Nochmal hallo
Text: ein weiterer Gruss von dibo
Weitere EMail verschicken (j/n)?n
```

```
TO: klaus@uni-oldenburg.de(Klaus)
FROM:          boles@informatik.uni-
oldenburg.de(dibo)
SUBJECT: Nochmal hallo

ein weiterer Gruss von dibo
****** End EMail *****
Weitere Mailboxabfrage (j/n)?n
```

Aufgabe 16: Opala

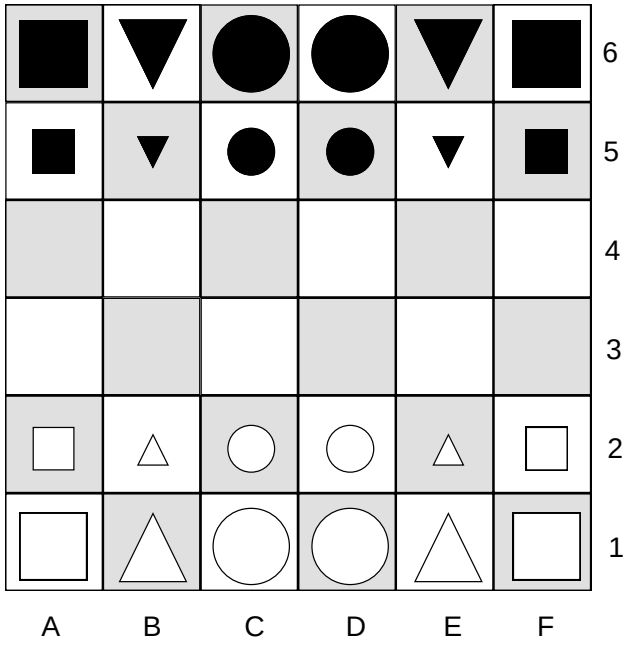
Opala - Spielregeln

Spielbrett

Gespielt wird auf einem Schachbrett mit allerdings nur 6x6 Feldern.

Spielfiguren

Spielfiguren sind geometrische Gebilde. Es gibt große und kleine Quadrate, große und kleine Kreise sowie große und kleine Dreiecke. In der Abbildung ist die Anfangsaufstellung zu sehen.



Spieler

OPALA ist ein Spiel für zwei Spieler. Spieler A (Weiß) spielt mit weißen Spielfiguren von unten nach oben. Spieler B (Schwarz) spielt mit schwarzen Spielfiguren von oben nach unten.

Spielablauf

Es wird immer abwechselnd gezogen, wobei Spieler Weiß beginnt. Es besteht Zugzwang.

Spielzüge

Für die einzelnen Spielfiguren gelten folgende Zugregeln:

- Große Quadrate dürfen wie Türme beim Schach gezogen werden, d.h. beliebig (>0) viele Felder horizontal oder vertikal.

- Kleine Quadrate dürfen wie große Quadrate gezogen werden, allerdings nur ein einzelnes Feld.
- Große Dreiecke dürfen wie Läufer beim Schach gezogen werden, d.h. beliebig (>0) viele Felder in einer Diagonalen.
- Kleine Dreiecke dürfen wie große Dreiecke gezogen werden, allerdings nur ein einzelnes Feld.
- Große Kreise dürfen wie Damen beim Schach gezogen werden, d.h. beliebig (>0) viele Felder in einer Horizontalen, Vertikalen oder Diagonalen.
- Kleine Kreise dürfen wie große Kreise gezogen werden, allerdings nur ein einzelnes Feld

Dabei gelten grundsätzlich folgende Resultate bzw. Einschränkungen:

- Bei einem Spielzug darf nur eine einzelne eigene Spielfigur gezogen werden.
- Bei einem Spielzug dürfen keine Spielfiguren übersprungen werden.
- Es darf nicht auf ein Feld gezogen werden, auf dem bereits eine eigene Spielfigur steht.
- Wird eine Spielfigur auf ein Feld gezogen, auf dem eine Spielfigur des Gegners steht, so wird diese geschlagen, d.h. vom Spielbrett entfernt.

Ziel des Spiels

Ziel des Spiels ist es, mit einer eigenen Spielfigur die gegenüber liegende Grundlinie zu erreichen, und zwar ohne dass diese im nächsten Zug wieder geschlagen werden könnte.

Ende des Spiels und Sieger

Ein Spiel ist beendet

- wenn ein Spieler X mit einer eigenen Spielfigur die gegenüber liegende Grundlinie erreicht hat und diese Spielfigur im nächsten Zug nicht direkt geschlagen werden kann. In diesem Fall hat Spieler X unmittelbar gewonnen.
- wenn ein Spieler X mit einer eigenen Spielfigur die gegenüber liegende Grundlinie erreicht hat und diese Spielfigur zwar im nächsten Zug direkt geschlagen werden kann, sein Gegner dies jedoch nicht tut. In diesem Fall hat Spieler X gewonnen, nachdem der Gegner seinen Zug ausgeführt hat.
- wenn ein Spieler, wenn er am Zug ist, keinen legalen Spielzug mehr ausführen kann. In diesem Fall hat sein Gegner gewonnen.
- sobald ein Spieler keine eigenen Spielfiguren mehr besitzt. In diesem Fall hat der Gegner gewonnen.

Aufgabe: Implementieren Sie das Spiel Opala, so dass es 2 Menschen gegeneinander spielen können. Implementieren Sie das Spiel auf objektorientierte Art und Weise, d.h. definieren Sie geeignete Klassen insbesondere zur Repräsentation des Spielbrettes, der Spielfiguren, der Spielzüge, der Spieler, der Spielregeln und des Spielablaufes.

Aufgabe 17:

Bei dieser Aufgabe sollen Sie ein Programm entwickeln, durch das zwei menschliche Spieler am Computer gegeneinander ein kleines Spiel namens *BreakThrough* spielen können.

Regeln

BreakThrough ist ein Spiel, das von zwei Personen auf einem Spielbrett von n Reihen und m Spalten (n und m = 5, 6, ... oder 9) gespielt wird.

Die Spieler (A und B) haben jeweils 2 * m gleichartige Figuren in ihrer Spielerfarbe (auf dem Brett). Sie werden zu Beginn auf den beiden Reihen aufgestellt, die dem jeweiligen Spieler am nächsten sind (wie bei der Schach-Grundstellung).

Die Spieler ziehen abwechselnd, A beginnt. Wer am Zug ist, muss ziehen, Passen ist nicht erlaubt.

Ein Zug wird gemacht, indem man eine seiner Figuren auf das Feld direkt vor der Figur zieht oder auf ein Feld diagonal vor der Figur. Das Feld, auf das gezogen wird, muss leer oder vom Gegner besetzt sein. Im letzten Fall wird die gegnerische Figur durch den Zug geschlagen, d. h. vom Brett genommen. Ziehen und Schlagen geht immer nur ein Feld weit, man kann keine Felder überspringen.

Man gewinnt, indem man eine seiner Figuren auf die hinterste Reihe auf der gegnerischen Seite bringt, oder indem man alle gegnerischen Figuren schlägt.

Aufgabe

Schreiben Sie ein Java-Programm, mit dessen Hilfe zwei menschliche Spieler das Spiel BreakThrough spielen können. Implementieren Sie das Spiel auf objektorientierte Art und Weise, d.h. definieren Sie geeignete Klassen insbesondere zur Repräsentation des Spielbrettes, der Spielfiguren, der Spielzüge, der Spieler, der Spielregeln und des Spielablaufes.

Hinweise:

Der generelle Spielablauf lässt sich folgendermaßen skizzieren:

- Aufbau des Ausgangsspielbretts
- Ausgabe des Spielbretts
- Spieler A beginnt
- Solange das Spiel nicht beendet ist, tue folgendes
    - Spielzug einlesen
    - Spielzug überprüfen
    - Falls Spielzug nicht korrekt ist, Spiel beenden und Sieger verkünden
    - Falls Spielzug korrekt ist,
        - Spielzug ausführen
        - Ausgabe des Spielbretts
        - Spielerwechsel
- Sieger verkünden

Beispiel für einen Programmablauf mit n = 7 und m = 8 (Benutzereingaben stehen in Klammern (<>)):

```
  Spieler A
  0 1 2 3 4 5 6 7
0 + + + + + + + +
1 + + + + + + + +
2 . . . . . . . .
3 . . . . . . . .
4 . . . . . . . .
5 o o o o o o o o
6 o o o o o o o o
  Spieler B

Spieler A ist am Zug!
von Reihe: 1
von Spalte: 0
nach Reihe: 2
nach Spalte: 0

  Spieler A
  0 1 2 3 4 5 6 7
0 + + + + + + + +
1 . + + + + + + +
2 + . . . . . . .
3 . . . . . . . .
4 . . . . . . . .
5 o o o o o o o o
6 o o o o o o o o
  Spieler B

Spieler B ist am Zug!
von Reihe: 5
von Spalte: 0
nach Reihe: 4
nach Spalte: 1

  Spieler A
  0 1 2 3 4 5 6 7
0 + + + + + + + +
1 . + + + + + + +
2 + . . . . . . .
3 . . . . . . . .
4 . o . . . . . .
5 . o o o o o o o
6 o o o o o o o o
  Spieler B

Spieler A ist am Zug!
von Reihe: 2
von Spalte: 0
nach Reihe: 3
nach Spalte: 1

  Spieler A
  0 1 2 3 4 5 6 7
0 + + + + + + + +
1 . + + + + + + +
2 . . . . . . . .
3 . + . . . . . .
4 . o . . . . . .
5 . o o o o o o o
6 o o o o o o o o
  Spieler B
```

```
Spieler B ist am Zug!
von Reihe: 4
von Spalte: 1
nach Reihe: 3
nach Spalte: 1

  Spieler A
  0 1 2 3 4 5 6 7
0 + + + + + + + +
1 . + + + + + + +
2 . . . . . . . .
3 . o . . . . . .
4 . . . . . . . .
5 . o o o o o o o
6 o o o o o o o o
  Spieler B

...
```

Aufgabe 18: Smess

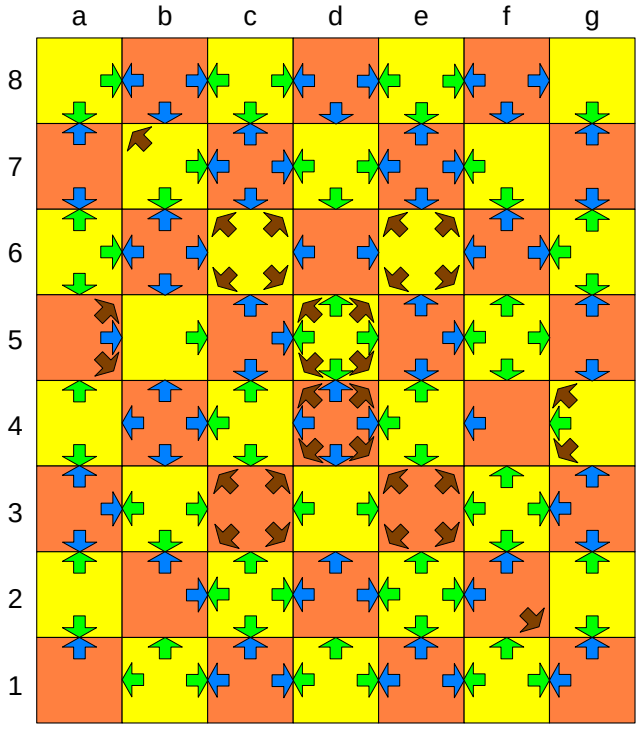
Bei dieser Aufgabe geht es um die Implementierung des Zwei-Personen-Strategiespiels *Smess*. Zunächst die Regeln:

Spieler:

Smess ist ein Spiel für zwei Spieler: Spieler A und Spieler B.

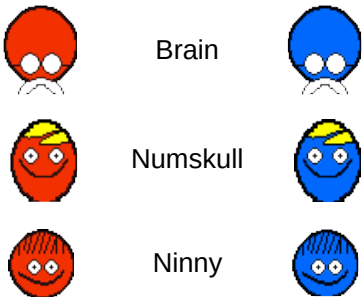
Spielbrett:

Smess wird auf einem Schachbrett-artigen Brett mit 8 x 7 Feldern gespielt (8 Reihen, 7 Spalten). Auf jedem Feld befinden sich Symbole, die Richtungen andeuten. Um einzelne Felder identifizieren zu können, werden Ihnen Koordinaten zugeordnet; die Reihen werden von unten nach oben von 1 bis 8 durchnummeriert, die Spalten von links nach rechts von a bis g.



Spielfiguren:

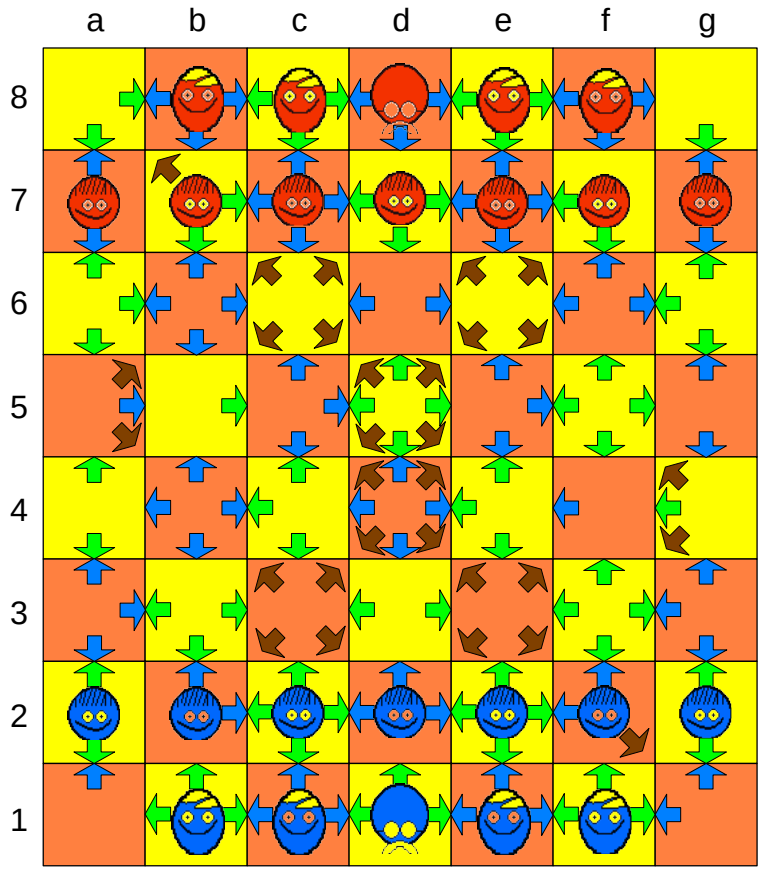
Spieler A besitzt 12 blaue Spielfiguren, Spieler B 12 rote Spielfiguren. Jeder Spieler besitzt dabei 1 „Brain“, 4 „Numskulls und 7 „Ninnys“.



Startaufstellung:

Die blauen Spielfiguren von Spieler A werden in den Reihen 1 und 2 platziert, die roten Spielfiguren von Spieler B in den Reihen 7 und 8. Im Einzelnen werden die Figuren auf folgenden Feldern platziert:

- Blaues Brain: 1d
- Blaue Numskulls: 1b, 1c, 1e, 1f
- Blaue Ninnys: 2a, 2b, 2c, 2d, 2e, 2f, 2g
- Rotes Brain: 8d
- Rote Numskulls: 8b, 8c, 8e, 8f
- Rote Ninnys: 7a, 7b, 7c, 7d, 7e, 7f, 7g



Spielzüge:

Ein Spielzug besteht aus dem Verschieben einer Spielfigur auf dem Spielbrett. Jeder Spieler kann dabei nur seine eigenen Spielfiguren ziehen. Dabei gelten folgende Zugregeln:

- Ninny: Darf in einem Spielzug auf ein Nachbarfeld verschoben werden, und zwar in eine der Richtungen, die auf dem Feld angezeigt wird, auf der die Figur gerade steht.
- Numskull: Darf eine beliebige Anzahl an Felder verschoben werden, und zwar in eine der Richtungen, die auf dem Feld angezeigt wird, auf der die Figur gerade steht. Dabei dürfen keine anderen Spielfiguren übersprungen werden (auch keine eigenen).
- Brain: Darf in einem Spielzug auf ein Nachbarfeld verschoben werden, und zwar in eine der Richtungen, die auf dem Feld angezeigt wird, auf der die Figur gerade steht.

Dabei gilt:

- Spielfiguren dürfen nicht auf Felder verschoben werden, auf denen bereits andere eigene Figuren stehen.
- Wird eine Spielfigur auf ein Feld verschoben, auf der eine Spielfigur des Gegners steht, so wird diese geschlagen, d.h. vom Spielbrett entfernt.
- Wird ein Ninny auf ein Startfeld eines Numskulls des Gegners gezogen, wird das Ninny zu einem Numskull umgewandelt.

Spielablauf:

Es wird abwechselnd gezogen. Spieler A beginnt. Das Spiel ist beendet,

- wenn ein Spieler am Zug ist und nicht ziehen kann oder
- wenn ein Brain geschlagen wird oder
- wenn sich nur noch die beiden Brains auf dem Spielbrett befinden.

Sieger, Verlierer:

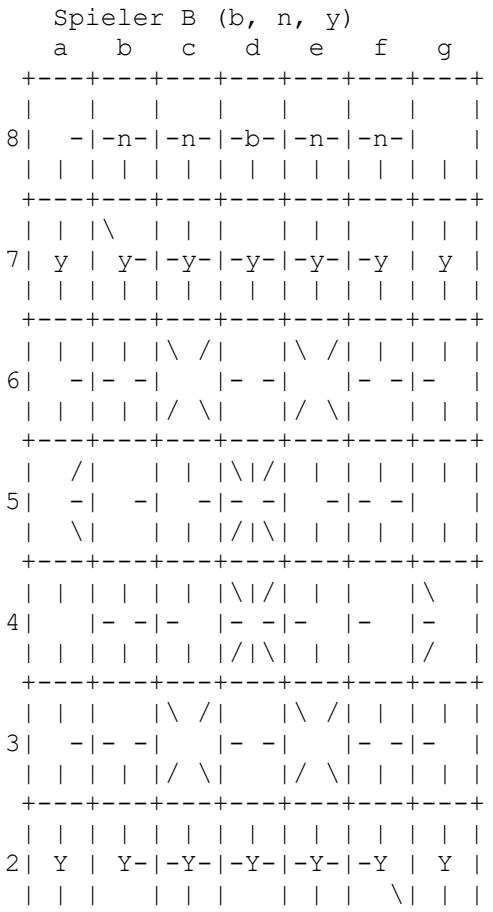
Ist ein Spieler am Zug und kann nicht ziehen, so hat er verloren. Wird das Brain eines Spielers geschlagen, so hat er verloren. In beiden Fällen ist der andere Spieler Sieger. Ein Spiel endet Unentschieden, wenn sich nur noch die beiden Brains auf dem Spielbrett befinden.

Online: Sie können das Spiel über folgenden URL online spielen:

http://www-is.informatik.uni-oldenburg.de/~dibo/teaching/pkjava/objekttheater/plays/smess/performance.html

Aufgabe: Schreiben Sie ein Java-Programm, mit dessen Hilfe zwei menschliche Spieler das Spiel Smess spielen können. Implementieren Sie das Spiel auf objektorientierte Art und Weise, d.h. definieren Sie geeignete Klassen insbesondere zur Repräsentation des Spielbrettes, der Spielfiguren, der Spielzüge, der Spieler, der Spielregeln und des Spielablaufes.

Skizze eines möglichen Programmablaufs:



```
  +---+---+---+---+---+---+---+---+
  | | | | | | | | | | | | | | | |
1|   |-N-|-N-|-B-|-N-|-N-|-  |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
    Spieler A (B, N, Y)

Spieler A ist am Zug!
Von Reihe (1..8): 2
Von Spalte (a..g):d
Nach Reihe (1..8): 3
Nach Spalte (a..g):d
    Spieler B (b, n, y)
    a   b   c   d   e   f   g
  +---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |
8|  -|-n-|-n-|-b-|-n-|-n-|   |
  | | | | | | | | | | | | | | |
  +---+---+---+---+---+---+---+
  | | |\  | | |   | | |   | | |
7| y | y-|-y-|-y-|-y-|-y | y |
  | | | | | | | | | | | | | | |
  +---+---+---+---+---+---+---+
  | | | | |\ /|   |\ /| | | | |
6|  -|- -|   |- -|   |- -|-  |
  | | | | |/ \|   |/ \|   | | |
  +---+---+---+---+---+---+---+
  |  /|   | | |\|/| | | | | | |
5|  -|  -|  -|- -|  -|- -|   |
  |  \|   | | |/|\| | | | | | |
  +---+---+---+---+---+---+---+
  | | | | | | |\|/| | |   |\  |
4|   |- -|-  |- -|-  |-  |-  |
  | | | | | | |/|\| | |   |/  |
  +---+---+---+---+---+---+---+
  | | |   |\ /|   |\ /| | | | |
3|  -|- -|   |-Y-|   |- -|-  |
  | | | | |/ \|   |/ \| | | | |
  +---+---+---+---+---+---+---+
  | | | | | | | | | | | | | | |
2| Y | Y-|-Y-|- -|-Y-|-Y | Y |
  | | |   | | |   | | |  \| | |
  +---+---+---+---+---+---+---+
  | | | | | | | | | | | | | | |
1|   |-N-|-N-|-B-|-N-|-N-|-  |
  |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+
    Spieler A (B, N, Y)

Spieler B ist am Zug!
Von Reihe (1..8):

...
```

Aufgabe 19:

Bei dieser Aufgabe sollen Sie ein Programm entwickeln, durch das ein menschlicher Spieler am Computer das Knobelspiel *Lampen* spielen kann.

Regeln

Das Spiel besteht aus einem quadratischen Spielfeld der Größe *n* (2 < *n* < 10), das aus einzelnen Zellen besteht, die Lampen repräsentieren. Lampen können sich im Zustand *an* oder *aus* befinden. Anfangs sind alle Lampen im Zustand *aus*. In jedem Spielzug wählt der Spieler eine Zelle aus, deren Zustand umgeschaltet werden soll (von *aus* in *an* oder umgekehrt). Gleichzeitig werden aber auch die Nachbarlampen oberhalb, unterhalb, links und rechts von der entsprechenden Lampe umgeschaltet (insofern sie existieren). Ziel (und Ende) des Knobelspiels ist es, den Zustand zu erreichen, dass alle Lampen angeschaltet sind.

Aufgabe

Schreiben Sie ein Java-Programm, mit dessen Hilfe ein menschlicher Spieler das Spiel *Lampen* spielen kann. Gehen Sie bei der Entwicklung nach den Methoden der objektorientierten Softwareentwicklung vor. Finden Sie geeignete Klassen und Methoden.

Hinweise:

Der generelle Spielablauf lässt sich folgendermaßen skizzieren:

- Abfrage der Spielfeldgröße
- Erzeugung/Initialisierung des Spielfeldes und des Spielers
- Spielfeld: Ausgabe
- Solange das Spiel nicht beendet ist (Spielfeld: !alleLampenAn), tue folgendes:
    - Spieler: Korrekten Spielzug einlesen
    - Spielfeld: Spielzug ausführen
    - Spielfeld: Ausgabe

Beispiel für einen Programmablauf (Benutzereingaben stehen in Klammern (<>), Lampen im Zustand *aus* werden durch ein ‚.', Lampen im Zustand *an* durch ein ‚+' gekennzeichnet):

```
Feldgroesse (2 < n < 10): <5>

 01234
0.....
1.....
2.....
3.....
4.....

Reihe der Lampe, die umgeschaltet werden soll (0 <= r < 5): <3>
Spalte der Lampe, die umgeschaltet werden soll (0 <= r < 5): <2>
 01234
0.....
1.....
2..+..
3.+++.
4..+..

Reihe der Lampe, die umgeschaltet werden soll (0 <= r < 5): <0>
Spalte der Lampe, die umgeschaltet werden soll (0 <= r < 5): <0>

 01234
0++...
1+....
```

```
2..+..
3.+++.
4..+..

Reihe der Lampe, die umgeschaltet werden soll (0 <= r < 5): <3>
Spalte der Lampe, die umgeschaltet werden soll (0 <= r < 5): <1>

 01234
0++...
1+....
2.++..
3+..+.
4.++..

...
```

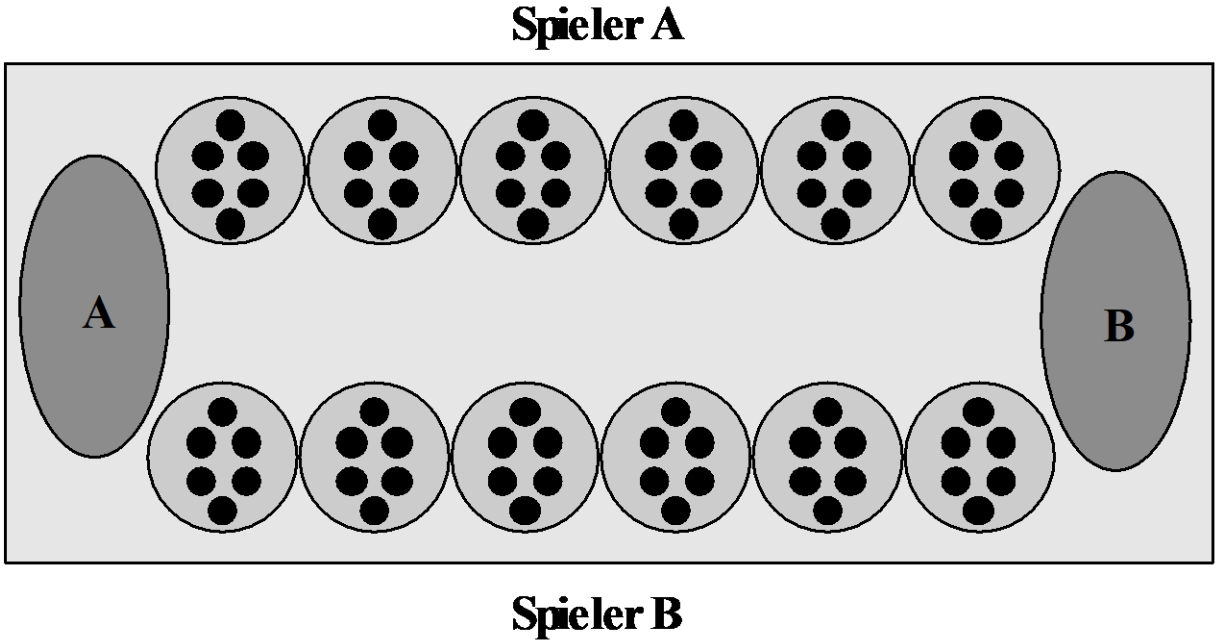
Aufgabe 20:

Kalah ist ein Spiel für zwei Personen. Es besteht aus einem Brett mit zwei Reihen von je sechs Löchern, in denen anfangs je sechs Kugeln liegen, und zwei weiteren Löchern, die Kalah heißen und anfangs leer sind (siehe Abbildung). Spieler B gehören die unteren Löcher und das rechte Kalah (B), Spieler A die oberen Löcher und das linke Kalah (A).

Ein Zug läuft wie folgt ab:

- Der Spieler entleert eines seiner nicht leeren Löcher (nicht sein Kalah) und verteilt auf die entgegen dem Uhrzeigersinn folgenden Löcher je eine Kugel, wobei er das Kalah des Gegners auslässt.
- Landet die letzte Kugel im eigenen Kalah, so darf und muss der Spieler noch einmal ziehen.
- Landet die letzte Kugel in einem eigenen leeren Loch und ist das gegenüberliegende Loch des Gegners nicht leer, so kommt die Kugel mit den gegenüberliegenden Kugeln ins eigene Kalah.

Beide Spieler ziehen abwechselnd, bis alle Löcher eines Spielers leer sind. Dann entleert der andere Spieler seine Löcher in sein Kalah. Gewonnen hat der Spieler, dessen Kalah am Ende mehr Kugeln enthält.



Aufgabe: Schreiben Sie ein Java-Programm, mit dessen Hilfe zwei menschliche Spieler das Spiel Kalah spielen können. Implementieren Sie das Spiel auf objektorientierte Art und Weise, d.h. definieren Sie geeignete Klassen insbesondere zur Repräsentation des Spielbrettes, der Spielfiguren, der Spielzüge, der Spieler, der Spielregeln und des Spielablaufes.

Aufgabe 21:

Reversi ist ein Spiel für zwei Personen.

Spielbrett

Reversi wird auf einem Spielbrett gespielt, das sich aus 8*8 einzelnen Feldern zusammensetzt. Es hat also dieselben Ausmaße wie ein Schachbrett, nur das die Felder nicht abwechselnd schwarz und weiß sind, sondern eine einheitliche Farbe haben. Es ist zweckmäßig, das Brett wie beim Schach mit Koordinaten zu versehen, um Spielzüge eindeutig angeben zu können.

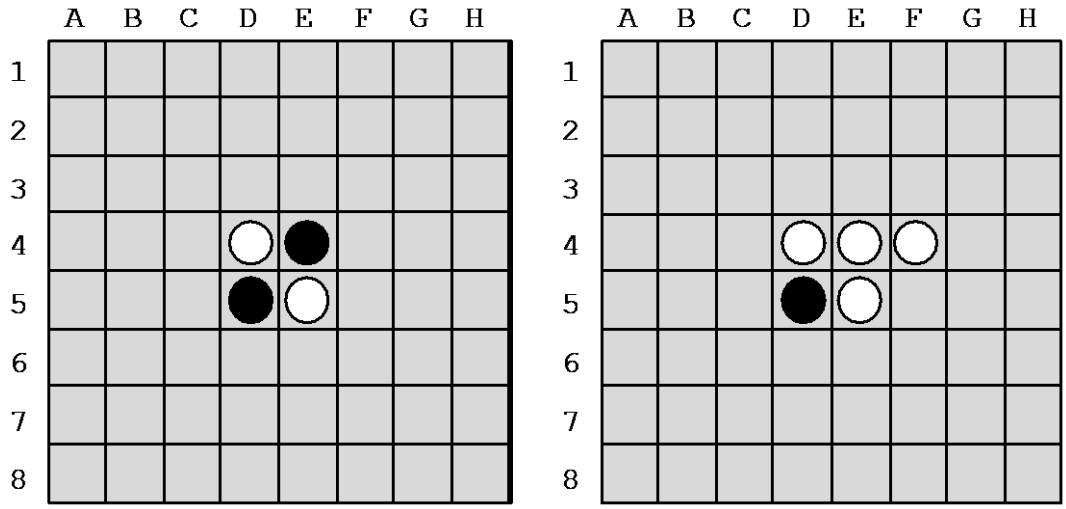
Spielfiguren

Als Spielfiguren werden runde Plättchen (Steine) verwendet. Diese haben je eine schwarze und eine weiße Seite. Es gibt insgesamt 64 Plättchen, also genausoviele, wie das Spielbrett Felder hat. Die Spielplättchen werden im Laufe des Spiels umgedreht (daher der Name Reversi), so dass schwarze zu weißen Steinen werden können und umgekehrt. Im folgenden ist immer von *schwarzen* und *weißen Steinen* die Rede, was sich jeweils auf die Oberseite der Steine - also die sichtbare Farbe - bezieht.

Spielverlauf und Ziel des Spiels

Der Spieler, der das Spiel beginnt, wird im folgenden *Spieler A* genannt. Der andere ist dementsprechend *Spieler B*. Spieler A bekommt die Farbe „Weiß“, Spieler B die Farbe „Schwarz“ zugeteilt. Zu Anfang des Spiels befinden sich vier Steine auf dem Spielbrett, zwei mit ihrer weißen (D4 und E5), zwei mit ihrer schwarzen Seite (E4 und

D5) nach oben (siehe Abbildung links). Die restlichen 60 Steine liegen in einem Sammelpool neben dem Spielbrett.



Spieler A führt dann seinen ersten Spielzug aus: Er nimmt einen Stein aus dem Sammelpool und legt ihn auf ein leeres Feld (selbstverständlich mit seiner Farbe „Weiß“ nach oben). Das leere Feld muss dabei an ein belegtes Feld horizontal, vertikal oder diagonal angrenzen. Außerdem muss der Spielzug zum „Schlagen“ von mindestens einem gegnerischen Stein führen. „Schlagen“ bedeutet für Spieler A: Drehe alle schwarzen Steine um, die sich horizontal, vertikal oder diagonal zwischen bereits gesetzten weißen Steinen und dem neu gesetzten (ebenfalls weißen) Stein befinden. In der Ausgangsituation kann sich Spieler A also eines der Felder E3, F4, D6 und C5 aussuchen. Setzt er den Stein bspw. auf Feld F4, muss er den schwarzen Stein auf Feld E4 umdrehen (siehe Abbildung rechts).

Damit ist der erste Spielzug von Spieler A beendet und Spieler B kommt zum Zug. Dieser führt nun eine identische Aktion durch, nur dass er jetzt natürlich weiße Steine schlagen muss.

Die beiden Spieler führen abwechselnd ihre Spielzüge durch. Ist es einem Spieler nicht möglich, ein leeres Feld mit einem Stein zu besetzen, so muss er passen und der andere Spieler ist wieder am Zug.

Das Spiel ist beendet, wenn kein Spieler mehr einen Stein setzen kann. Dies ist in der Regel der Fall, wenn alle Felder besetzt sind, kann aber auch schon vorher passieren. Die schwarzen und weißen Steine auf dem Spielbrett werden gezählt. Sieger des Spiels ist der Spieler, der die größere Anzahl an Steinen auf dem Spielbrett hat.

Achtung

- Ein Stein, der einmal auf dem Spielbrett liegt, wird nie mehr vom Brett genommen oder verschoben. Er wird höchstens umgedreht.
- Jeder Spielzug muss vollständig ausgeführt werden, d.h. alle Steine, die aufgrund eines neu gesetzten Steines geschlagen werden können, müssen auch umgedreht werden. Die Spieler dürfen sich nicht aussuchen, welche Steine sie umdrehen und welche nicht.
- Solange ein Spieler gegnerische Steine schlagen kann, muss er ziehen. Er darf nicht freiwillig passen.

- Der entscheidende Stein beim Umdrehen ist der neu gesetzte Stein. Zum Beispiel werden weiße Steine nicht umgedreht, die zwischen zwei schwarzen Steinen liegen, die beide schon vorher auf dem Spielbrett lagen. Es findet also keine transitive Fortsetzung beim Umdrehen von Steinen statt. Wird bspw. bei der Ausgangsposition in der Abbildung unten links ein schwarzer Stein auf Feld F6 gesetzt, dann werden die beiden weißen Steine auf den Feldern D4 und E5 umgedreht, und es ergibt sich die Situation in der Abbildung unten rechts. Obwohl durch die Umdreh-Aktion nun auch der weiße Stein auf Feld D6 zwischen zwei schwarzen Steinen (E5 und C7) liegt, wird dieser nicht umgedreht, weil sich die beiden schwarzen Steine schon vorher auf dem Spielbrett befanden.

Aufgabe: Schreiben Sie ein Java-Programm, mit dessen Hilfe zwei menschliche Spieler das Spiel Reversi spielen können. Implementieren Sie das Spiel auf objektorientierte Art und Weise, d.h. definieren Sie geeignete Klassen insbesondere zur Repräsentation des Spielbrettes, der Spielfiguren, der Spielzüge, der Spieler, der Spielregeln und des Spielablaufes.

Aufgabe 22:

Metamorphose ist ein Spiel für zwei Personen (Spieler A und Spieler B).

Spielbrett

Metamorphose wird auf einem Spielbrett gespielt, das sich aus 7*7 einzelnen Feldern zusammensetzt. Bestimmte Felder sind durch Mauern besetzt (siehe Abbildung). Es ist zweckmäßig, das Brett wie beim Schach mit Koordinaten zu versehen, um Spielzüge eindeutig angeben zu können.

Spielfiguren

Als Spielfiguren werden Quadrate und Kreise verwendet. Es gibt weiße Spielfiguren, die Spieler A gehören, und schwarze Spielfiguren, die Spieler B gehören. Anfangs besitzt jeder Spieler vier Quadrate und drei Kreise.

Spielzug

In jedem Spielzug verschiebt ein Spieler eine seiner Spielfiguren um ein oder mehrere Felder auf dem Spielbrett. Dabei gilt: Quadrate können horizontal und vertikal verschoben werden. Kreise können diagonal verschoben werden. Nachdem eine Figur verschoben wurde, wechselt sie ihren Typ (aus einem Kreis wird ein Quadrat und umgekehrt).

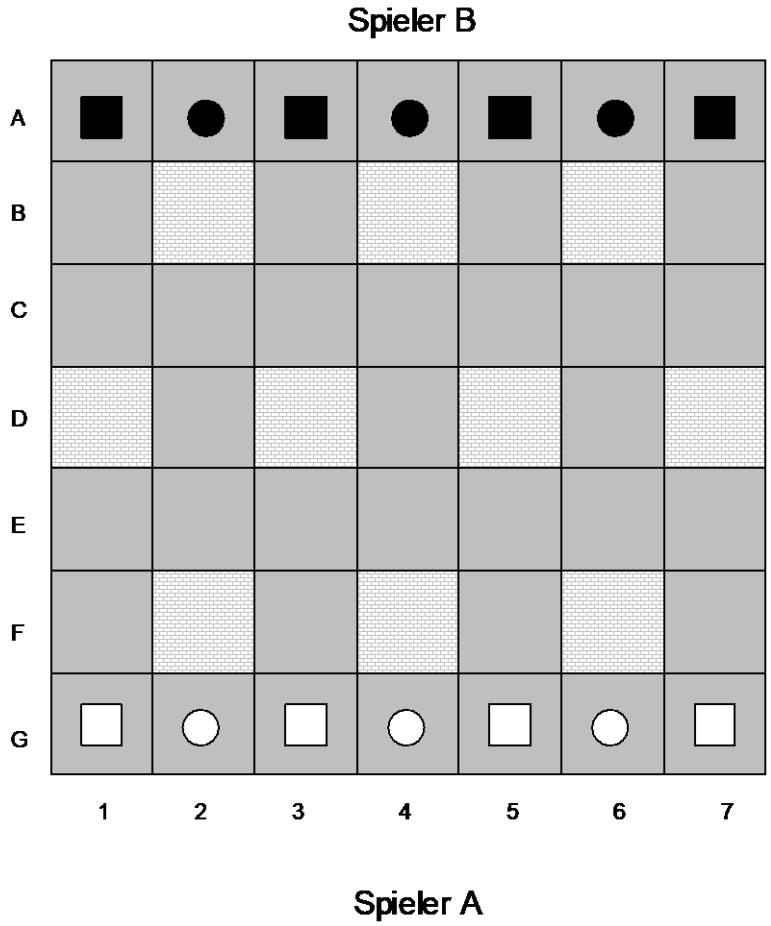
Spielregeln

- Die Anfangsaufstellung ist in der Abbildung dargestellt.
- Spieler A (Weiß) beginnt.
- Es besteht Zugzwang. Kann ein Spieler nicht mehr ziehen kann, hat er verloren!
- Auf einem Feld darf maximal eine Figur stehen.
- Auf die durch Mauern besetzten Felder dürfen keine Figuren gezogen werden.

- Bei einem Spielzug dürfen keine Mauern und andere Figuren (der eigenen oder fremden Farbe) übersprungen werden.
- Endet ein Spielzug auf einem Feld, auf dem eine fremde Figur steht, so gilt diese als geschlagen und wird vom Spielbrett entfernt.
- Eigene Figuren dürfen nicht geschlagen werden.

Ziel des Spiels

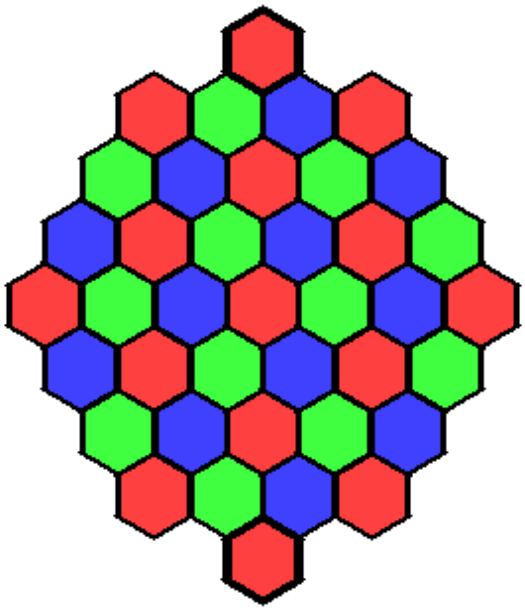
Am Anfang des Spiels wählt jeder Spieler (geheim) eine seiner Spielfiguren aus und merkt sie sich als besondere Spielfigur, den König. Wer als erster den König des Gegners schlägt, hat gewonnen. Das Spiel endet unentschieden, wenn beide Spieler nur noch ihren König besitzen.



Aufgabe: Schreiben Sie ein Java-Programm, mit dessen Hilfe zwei menschliche Spieler das Spiel Metamorphose spielen können. Implementieren Sie das Spiel auf objektorientierte Art und Weise, d.h. definieren Sie geeignete Klassen insbesondere zur Repräsentation des Spielbrettes, der Spielfiguren, der Spielzüge, der Spieler, der Spielregeln und des Spielablaufes.

Aufgabe 23:

PAROB ist eine Schachvariante, die auf einem Spielbrett mit 39 Feldern gespielt wird, deren Anordnung in der folgenden Abbildung gezeigt wird. Bis auf die Ausnahmen, die im Folgenden aufgelistet werden, sind die Spielregeln dieselben wie



beim Schach.

Ziel des Spiels

Die roten Felder mit den dicken Linien ganz unten und oben sind die „Burgen“ der beiden Spieler. Ziel des Spiel ist es, eine eigene Spielfigur in der gegnerischen Burg zu platzieren.

Spieler

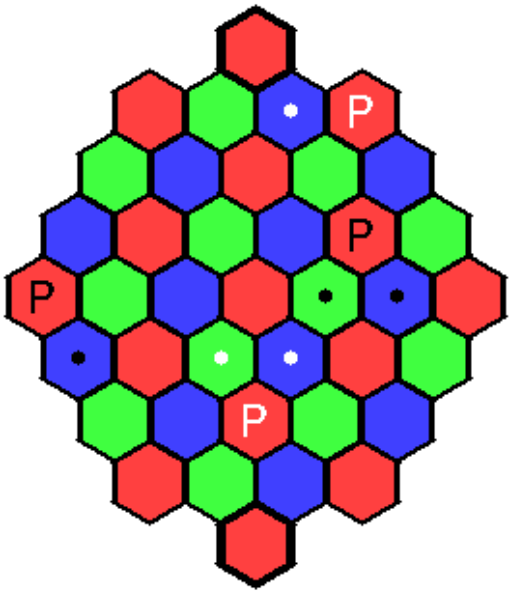
PAROB ist ein Spiel für zwei Spieler. Spieler Weiß spielt mit weißen Spielfiguren von unten nach oben. Spieler Schwarz spielt mit schwarzen Spielfiguren von oben nach unten. Es wird immer abwechselnd gezogen, wobei Spieler Weiß beginnt.

Spielfiguren

Es gibt drei Typen von Spielfiguren: Bauern, Türme und Läufer. Eigene Spielfiguren können nicht geschlagen werden. Auf jedem Feld des Spielbretts kann höchstens eine Spielfigur stehen.

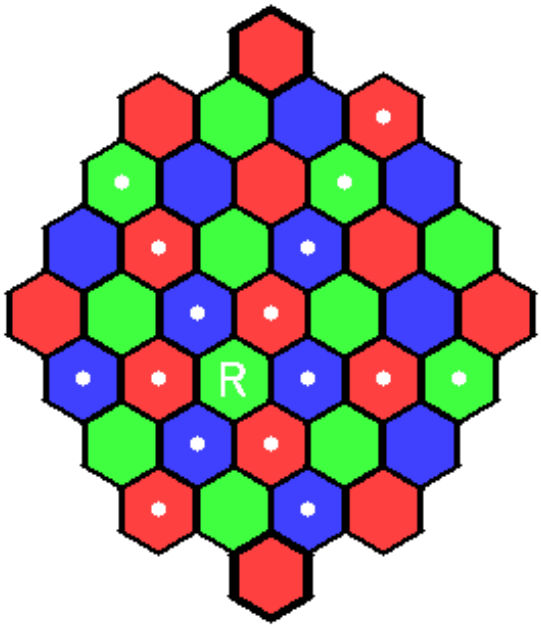
Bauern

Ein Bauer (P) kann in einem Spielzug verschoben werden auf das Feld oberhalb von ihm links und oberhalb von ihm rechts (mit oberhalb ist damit gemeint: weg von der eigenen Burg). Falls ein Bauer auf einem der beiden roten Felder in der vorletzten Reihe steht, kann er auch auf das Feld links bzw. rechts daneben verschoben werden. Steht auf dem Feld, auf das ein Bauer verschoben wird, eine gegnerische Figur, so gilt diese als geschlagen und wird vom Spielbrett entfernt. Anders als beim Schachspiel schlagen Bauern genauso wie sie bewegt werden. Sie dürfen auch nicht wie beim Schach anfangs zwei Felder vorwärts bewegt werden. Die folgende Abbildung enthält Beispiele für gültige Bauernzüge.



Türme

Ein Turm (R) kann in einem Spielzug eine beliebige Anzahl von Feldern in einer geraden Linie in alle sechs Richtungen (links, rechts, oben-links, oben-rechts, unten-links, unten-rechts) verschoben werden. Er darf dabei weder eigene noch fremde Spielfiguren überspringen. Steht auf dem Feld, auf das ein Turm verschoben wird, eine gegnerische Figur, so gilt diese als geschlagen und wird vom Spielbrett entfernt.

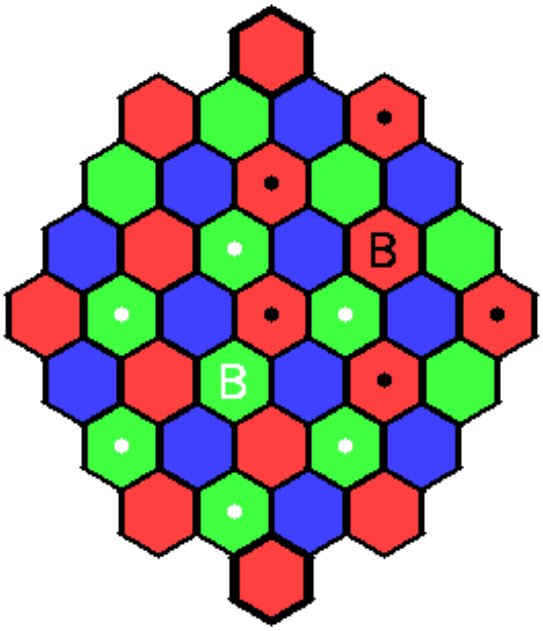


Die obige Abbildung enthält Beispiele für gültige Turmzüge.

Läufer

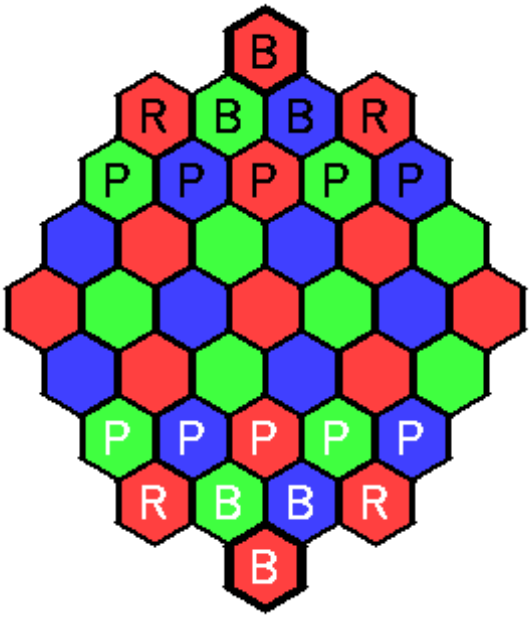
Ein Läufer (B) kann in einem Spielzug verschoben werden auf ein beliebiges der sechs Felder, die zwei Felder entfernt sind und dieselbe Farbe haben (Läufer bleiben

immer auf Feldern derselben Farbe). Läufer können dabei auch eigene und fremde Spielfiguren überspringen (sie sind also quasi eine Mischung von Läufern und Springern beim Schach). Die folgende Abbildung enthält Beispiele für gültige



Läuferzüge.

Spielbeginn



Die obige Abbildung zeigt die initiale Aufstellung der Spielfiguren.

Zusatzregeln

- Sobald ein Spieler keine Spielfiguren mehr hat, die die gegnerische Burg besetzen können, hat er verloren.
- Sobald ein Spieler keine legalen Spielzüge mehr ausführen kann, hat er verloren.

Aufgabe: Schreiben Sie ein Java-Programm, mit dessen Hilfe zwei menschliche Spieler das Spiel PAROB spielen können. Implementieren Sie das Spiel auf objektorientierte Art und Weise, d.h. definieren Sie geeignete Klassen insbesondere zur Repräsentation des Spielbrettes, der Spielfiguren, der Spielzüge, der Spieler, der Spielregeln und des Spielablaufes.

Aufgabe 24:

Rushhour ist ein Spiel für einen einzelnen Spieler. Auf einem rechteckigen Spielbrett, das aus einzelnen Kacheln besteht, wird initial eine Menge an Autos platziert. Jedes Auto überdeckt mindestens zwei Kacheln. Ein Auto mit mehr als zwei Kacheln darf dabei keine Ecken aufweisen. Liegen die Kacheln, die ein Auto überdeckt, nebeneinander, ist es ein horizontal verschiebbares Auto. Liegen die Kacheln übereinander, ist es ein vertikal verschiebbares Auto. Ein Auto wird als Hauptauto gekennzeichnet. Ziel des Spiels ist es nun, die Autos so zu verschieben, bis irgendwann das Hauptauto (im Bild bspw. das rote Auto) den rechten Rand berührt.

Horizontale Autos dürfen dabei nur nach links und rechts, vertikale Autos nur nach oben und unten verschoben werden. Ein Auto darf nicht über den Rand hinweg oder wenn ein anderes Auto im Wege steht, verschoben werden.

Entwickeln Sie ein Java-Programm, mit dem ein Benutzer Rushhour spielen kann. Die Ausgangsstellung der Autos sei dabei in einer Textdatei festgelegt, die mittels einer der readFile-Methoden der Klasse IO gelesen werden kann. Die Autoteile werden dabei durch jeweils gleiche Zeichen repräsentiert; das Hauptauto durch das Zeichen ‚*'. Die Stellung der Autos in der oberen Abbildung entspräche bspw. folgendem Dateiinhalt:

```
122 3
145 36
145**6
7778 6
  98aa
bb9cc
```

Der Benutzer soll nun solange Autos verschieben können (durch Angabe von Auto und Richtung) bis das Hauptauto den rechten Rand erreicht. Im Folgenden wird ein beispielhafter Programmablauf skizziert (Benutzereingaben in <>):

```
122 3
145 36
145**6
7778 6
  98aa
bb9cc

Wahl eines Autos (Buchstabe angeben): <c>
Richtung (l=links, r=rechts, u=hoch, d=runter, q=Autowechsel): <>r
122 3
145 36
145**6
7778 6
  98aa
bb9 cc

Richtung (l=links, r=rechts, u=hoch, d=runter, q=Autowechsel): <q>
Wahl eines Autos (Buchstabe angeben): 8
Richtung (l=links, r=rechts, u=hoch, d=runter, q=Autowechsel): <d>
122 3
145 36
145**6
777  6
  98aa
bb98cc

Richtung (l=links, r=rechts, u=hoch, d=runter, q=Autowechsel): <q>
Wahl eines Autos (Buchstabe angeben): 7
Richtung (l=links, r=rechts, u=hoch, d=runter, q=Autowechsel): <r>
122 3
145 36
145**6
 777 6
  98aa
bb98cc

Richtung (l=links, r=rechts, u=hoch, d=runter, q=Autowechsel): <r>
122 3
145 36
145**6
  7776
  98aa
bb98cc

...
```

Gehen Sie bei der Entwicklung nach den Methoden der objektorientierten Softwareentwicklung vor. Definieren Sie bspw. die Klassen `Spiel`, `Spielbrett` und `Auto`, jeweils mit geeigneten Methoden.

Aufgabe 25:

Gegeben seien die folgenden Klassen `Element` und `List`. Sie implementieren die Datenstruktur einer „verketteten Liste“. Eine verkettete Liste ist eine dynamische Datenstruktur, die eine Speicherung von miteinander in Beziehung stehenden Objekten (hier `int`-Werte) erlaubt. Die Anzahl der Objekte ist im Vorhinein nicht bestimmt. Die Liste wird durch Referenzen auf die jeweils folgende(n) Elemente realisiert.

```
class Element {
     int value; // Speicher fuer einen Wert
     Element next; // Referenz auf das folgende Element
                // (oder null fuer Ende)

     Element(int v, Element n) {
           this.value = v;
           this.next = n;
     }
}

class List {
     Element first; // erstes Element der Liste
     Element last; // letztes Element der Liste

     List() {
           this.first = null;
           this.last = null;
     }

     void append(int i) {
           Element elem = new Element(i, null);
           if (this.first == null) { // erstes Element in der Liste
                 this.first = elem;
                 this.last = elem;
           } else { // Anhaengen des Elementes am Ende
                 this.last.next = elem;
                 this.last = elem;
           }
     }

     void print() {
           Element elem = this.first;
           while (elem != null) {
                 System.out.print(elem.value);
                 if (elem.next != null) {
                       System.out.print(" -> ");
                 }
                 elem = elem.next;
           }
           System.out.println();
     }
}
```

Ergänzen Sie die Klasse `List` um folgende zwei Methoden:

- Einen Copy-Konstruktor, d.h. einen Konstruktor, dem als Parameter ein existierendes Objekt der Klasse `List` übergeben wird. Die neue Liste soll wertegleich der als Parameter übergebenen Liste sein, d.h. die neue Liste soll dieselben Werte in derselben Reihenfolge speichern wie die übergebene Liste (die neue Liste soll nicht lediglich dieselben Element-Objekte enthalten).
- Eine Methode `equals`, als Parameter ein existierendes Objekt der Klasse `List` übergeben wird. Die Methode soll das Objekt, für das die Methode aufgerufen wird, und das übergebene Objekt auf Gleichheit überprüfen und einen entsprechenden booleschen Wert liefern. Zwei Listen sollen dabei genau dann gleich sein, wenn Sie dieselben Werte in derselben Reihenfolge speichern (d.h. wenn ein Aufruf der Methode `print` für beide Listen dieselbe Ausgabe erzeugen würde).

Aufgabe 26:

In dieser Aufgabe geht es um die Ermittlung von Aktiengewinnen bzw. -verlusten. Dazu werden folgende Klassen zur Verfügung gestellt:

- eine Klasse *Bank*, die eine Bank repräsentiert
- eine Klasse *Kunde*, die Kunden einer Bank repräsentiert
- eine Klasse *Depot*, zur Aufbewahrung von Aktien der Bankkunden
- eine Klasse *Aktie*, die Aktien repräsentiert
- eine Klasse *Boerse*, über die aktuelle Aktienkurse ermittelt werden können

Die Klassen haben dabei folgendes Protokoll:

```
// in dem Szenario gibt es genau eine Bank, die eine Menge an Kunden hat
class Bank {

     // liefert den Kunden mit dem als Parameter übergebenen Namen;
     // liefert null, wenn es keinen Kunden mit dem übergebenen
      // Namen gibt
     public Kunde getKunde(String name)
}

// ein Kunde hat einen Namen und ein Depot
class Kunde {

     // liefert den Namen des Kunden (!= null)
     public String getName()

     // liefert das Depot des Kunden oder null, wenn der Kunde kein Depot
    // besitzt
     public Depot getDepot()
}

// in einem Depot werden Aktien aufbewahrt
class Depot {

     // liefert alle Aktien des Depots (!= null)
     public Aktie[] getAktien()
}
```

```
// repräsentiert eine einzelne Aktie; Aktien haben als Identifier eine
// eindeutige ISIN (International Securities Identification Number)
class Aktie {

     // liefert die ISIN der Aktie (!= null)
     public String getISIN()

     // liefert den Kurs, zu dem die Aktie gekauft wurde
     public double getKaufKurs()
}

// repräsentiert eine Möglichkeit zur Ermittlung von Aktienkursen
class Boerse {
     private Boerse()

     // liefert den aktuellen Kurs der Aktie mit der als Parameter
     // übergebenen ISIN
     public static double getAktuellerKurs(String isin)
}
```

Aufgabe:

Schreiben Sie eine Methode, die durch Nutzung der Klassen folgendes tut:

Beim Aufruf können der Methode beliebig viele Namen als Parameter übergeben werden, die Namen von Kunden der Bank entsprechen sollen.

Für jeden angegebenen Kunden soll ermittelt und entsprechend ausgegeben werden, ob und wie viel Gewinn oder Verlust sein Aktienbestand aktuell aufweist, und zwar in folgender Form:

- Gewinn: `<Kundenname>: Aktueller Gewinn = <Gewinn>`
- Verlust: `<Kundenname>: Aktueller Verlust = <Verlust>`
- 0: `<Kundenname>: Weder Gewinn noch Verlust.`

Wird ein unbekannter Kunde als Parameter übergeben, soll ausgegeben werden:

```
<Angegebener Name> ist kein Kunde der Bank.
```

Beispiel:

Eine exemplarische Ausgabe für folgenden Aufruf der Methode

```
gewinnVerlust("Mueller", "Meier", "Schulze", "Schmidt");
```

könnte dann folgende Gestalt haben:

```
Mueller ist kein Kunde der Bank.
Meier: Aktueller Gewinn = 1023.24
Schulze: Aktueller Verlust = 864.32
Schmidt: Weder Gewinn noch Verlust.
```

Aufgabe 27:

In dieser Aufgabe wird eine stark abgespeckte Version des sozialen Netzwerkes *Facebook* simuliert. Facebook besteht hierbei aus einer Menge an Nutzern. Jeder

Nutzer hat eine Menge an Freunden sowie eine Pinnwand. An dieser Pinnwand können Nutzer Nachrichten hinterlassen. Nachrichten können Nutzer mit „Gefällt mir“ markieren.

Gegeben sei folgender Ausschnitt aus einer API, die die Abfrage bestimmter Facebook-Daten ermöglicht.

```
// Facebook enthaelt eine Menge von Nutzern
class Facebook {
	static Nutzer[] nutzer;

	// liefert den Nutzer mit dem angegebenen Namen; liefert null, wenn
	// ein Nutzer mit dem Namen nicht in Facebook existiert
	static Nutzer getNutzer(String name) {
		return nutzer[0]; // dummy-Implementierung
	}
}

// jeder Nutzer hat einen eindeutigen Namen; jeder Nutzer hat eine Menge an
// Freunden; jedem Nutzer gehoert eine Pinnwand, auf der er oder andere
// Nutzer Nachrichten hinterlassen koennen
class Nutzer {
	String name;
	Nutzer[] freunde;
	Pinnwand pinnwand;

	boolean equals(Nutzer n) {
		return this.name.equals(n.name);
	}

	// liefert den Namen des Nutzers
	String getName() {
		return this.name;
	}

	// liefert alle Freunde des Nutzers
	Nutzer[] getFreunde() {
		return this.freunde;
	}

	// liefert die Pinnwand des Nutzers
	Pinnwand getPinnwand() {
		return this.pinnwand;
	}
}

// jede Pinnwand gehoert einem Facebook-Nutzer; auf einer Pinnwand koennen
// Nachrichten hinterlassen werden
class Pinnwand {
	Nutzer eigentuemer;
	Nachricht[] nachrichten;

	// liefert den Eigentuemer der Pinnwand
	Nutzer getEigentuemer() {
		return this.eigentuemer;
	}

	// liefert die Nachrichten, die an der Pinnwand hinterlassen worden
	// sind
```

```
	Nachricht[] getNachrichten() {
		return this.nachrichten;
	}
}

// eine Nachricht besteht aus einem Text; jede Nachricht steht an genau
// einer Pinnwand; Nutzer koennen durch Druecken eines "Gefaellt mir"-
// Buttons signalisieren, dass ihnen die Nachricht gefaellt
class Nachricht {
	String text;
	Pinnwand pinnwand;
	Nutzer[] gefaelltDasNutzer;

	// liefert den Text der Nachricht
	String getText() {
		return this.text;
	}

	// liefert die Pinnwand, an der sich die Nachricht befindet
	Pinnwand getPinnwand() {
		return this.pinnwand;
	}

	// liefert die Nutzer, die denen die Nachricht gefaellt
	Nutzer[] getGefaelltDasNutzer() {
		return this.gefaelltDasNutzer;
	}
}
```

Aufgabe:

Schreiben Sie auf der Grundlage dieser API ein Programm, bei dem der Benutzer zunächst aufgefordert wird, einen Namen einzugeben. Das Programm soll darauf die Anzahl bestimmen und ausgeben, wie oft der Nutzer mit dem eingegebenen Namen auf den Pinnwänden seiner Freunde zu Nachrichten den „Gefällt mir“-Button angeklickt hat. Existiert kein Nutzer mit dem eingegebenen Namen, soll eine Fehlermeldung ausgegeben werden. Die Ausgaben sollen die folgende Form haben:

```
Der Nutzer mit dem Namen dibo hat 13 mal den "Gefaellt mir"-
Button auf den Pinnwaenden seiner Freunde gedrueckt!

Ein Nutzer mit dem Namen Otto ist unbekannt!
```

Aufgabe 28:

In dieser Aufgabe geht es um die objektorientierte Modellierung eines Kopierers. In diesen kann man ein Blatt einlegen, die Anzahl an Kopien einstellen und den Kopierprozess starten.

Implementieren Sie zunächst eine Klasse **`Blatt`**. Diese soll den Inhalt eines Blattes in Form eines Attributs vom Typ **`String`** kapseln und folgende Methoden definieren:

- einen geeigneten Default-Konstruktor
- einen geeigneten Copy-Konstruktor
- eine Methode **`beschreiben`** mit einem Parameter vom Typ **`String`**. Der aktuelle Parameterwert soll dabei den bisherigen Inhalt des Blattes ersetzen.
- eine Methode **`lesen`**, die den aktuellen Inhalt des Blattes als **`String`**-Wert liefert.

Implementieren Sie dann eine Klasse **`Kopierer`**. Diese definiert zwei Attribute

- **`int anzahlKopien`**: Speicherung der eingegebenen Anzahl an Kopien
- **`Blatt blatt`**: Speicherung des eingelegten Blattes

sowie drei Methoden

- eine Methode **`blattEinlegen`**, der das einzulegende Blatt als Parameter übergeben wird, das im entsprechenden Attribut gespeichert wird
- eine Methode **`kopienEinstellen`**, der als Parameter die Anzahl an Kopien übergeben wird, die im entsprechenden Attribut gespeichert wird
- eine Methode **`kopieren`**, die ein 1-dimensionales Array mit Referenzen auf Objekte der Klasse **`Blatt`** zurückliefert. Die Größe des Arrays entspricht dabei der aktuell eingestellten Anzahl an Kopien und die Elemente des Arrays sind Referenzen auf Werte-gleiche Kopien des aktuell eingelegten Blattes.

Behandeln Sie potentiell fehlerhafte Parameterwerte adäquat.

Ein Testprogramm für Ihre Klassen könnte folgendermaßen aussehen:

```
Kopierer k = new Kopierer();
Blatt klausur = new Blatt();
klausur.beschreiben("Klausur A");
k.kopienEinstellen(5);
k.blattEinlegen(klausur);
Blatt[] kopien = k.kopieren();
for (Blatt b : kopien) {
  System.out.println(b.lesen());
}
```

Aufgabe 29:

Gegeben seien folgende Klassen (die internen Datenstrukturen und die Implementierung der Methoden sind hier unwichtig):

```
class Krankenhaus {
  // laedt alle Daten des Krankenhauses aus einer Datenbank
  Krankenhaus()

  // liefert alle Aerzte des Krankenhauses
  Arzt[] getAerzte()
}

class Person {
  // liefert den Namen der Person
  String getName()
}

class Patient extends Person {
  // liefert die Krankheit der Person als String
  String getKrankheit()
}

class Arzt extends Person {
```

```
  // liefert den Titel des Arztes (Dr., Prof. , ...)
  String getTitel()

  // liefert die durch den Arzt behandelten Patienten (!= null)
  Patient[] getBehandeltePatienten()
}
```

Gesucht ist ein Programm, das die Klassen bzw. deren Methoden nutzt, um folgendes zu bewirken: Für alle Ärzte eines Krankenhauses sollen für jeden Arzt des Krankenhauses alle von diesem behandelten Patienten ausgegeben werden und zwar in folgender Form:

```
Arzt <Arzttitel> <Arztname> behandelt aktuell folgende <Anzahl>
Patienten:
Patient 1: <Patientenname> mit der Krankheit <Krankheit>
Patient 2: <Patientenname> mit der Krankheit <Krankheit>
...
Patient <Anzahl>: <Patientenname> mit der Krankheit <Krankheit>
```

Hat ein Arzt aktuell 0 Patienten soll ausgegeben werden:

```
Arzt <Titel> <Arztname> hat aktuell keine Patienten
```

Die in Klammern (**<>**) gesetzten Angaben müssen dabei über entsprechende Methodenaufrufe ermittelt werden.

Aufgabe 30:

Gegeben seien folgende Klassen (die Implementierung der Methoden ist hier unwichtig):

```
class Bibliothek {
    // laedt alle Daten der Bibliothek aus einer Datenbank
    Bibliothek()

    // liefert alle Buecher der Bibliothek (!= null)
    Buch[] getBuecher()
}

class Buch {
    // liefert den Titel des Buches (!= null)
    String getTitle()

    // liefert diejenige Person, die das Buch aktuell ausgeliehen
    // hat; liefert "null", wenn das Buch aktuell nicht
    // ausgeliehen ist
    Person getLeser()
}

class Person {
    // liefert den Namen der Person (!= null)
    String getName()
}
```

Gesucht ist ein Programm, das die Klassen bzw. deren Methoden nutzt, um folgendes zu bewirken: Für alle Bücher der Bibliothek soll überprüft werden, ob sie aktuell ausgeliehen sind. Für jedes Buch, das ausgeliehen ist, soll auf die Konsole ausgegeben werden: **„<Name der ausleihenden Person> hat folgendes Buch ausgeliehen: <Titel des Buches>“**. Ist ein Buch nicht ausgeliehen, soll ausgegeben werden „**Das Buch mit dem Titel " + <Titel des Buches> ist nicht ausgeliehen!**“ Die Angaben in <> müssen natürlich entsprechend ermittelt werden. Die gegebenen Klassen dürfen nicht verändert oder erweitert werden!

Aufgabe 31:

Geben seinen folgende Klassen. Von Interesse sind dabei nur die Methoden (API), nicht deren Implementierung.

```
class Mannschaft {
     String getName()
}

class Spiel {

     // liefert die Heim-Mannschaft
     Mannschaft getHeim()

     // liefert die Auswärts-Mannschaft
     Mannschaft getAuswaerts()

     Ergebnis getErgebnis()
}

class Ergebnis {

     // liefert die geschossenen Tore der Heim-Mannschaft
     int getToreHeim()

// liefert die geschoss. Tore der Auswärts-Mannschaft
     int getToreAuswaerts()
}

class Saison {
     // liefert alle Spiele einer Liga mit unbekannt
// vielen Mannschaften von einer Saison
     static Spiel[] getSpiele(String jahr)
}
```

Implementieren Sie auf der Grundlage dieser Klassen bzw. deren Methoden ein Programm, das folgendes tut: Zu einer gegebenen Saison und einem gegebenen Mannschaftsnamen soll das Programm berechnen und auf die Konsole ausgeben, wie viele Punkte die Mannschaft in der Saison erreicht hat. Bsp.:

```
     final String saison = "2012";
```

```
final String mannschaftsname = "Bayern";
...
Bayern hat in der Saison 2012 54 Punkte erreicht
```

Dabei gilt: Bei einem Sieg, d.h. mehr geschossene Tore, erhält eine Mannschaft 3 Punkte, bei einem Unentschieden (gleich viel geschossene Tore) 1 Punkt.