Studiengänge: Elektro- und Informationstechnik, Flug- und Fahrzeuginformatik, Informatik, Mechatronik

Prüfung
Grundlagen der Programmierung 2
Objektorientierte Programmierung

Prüfer: B. Glavina, S. Hahndel, F. Regensburger, U. Schmidt

Prüfungsdauer: 90 Minuten

Hilfsmittel: keine

Studiengang	Dozent	Matrikelnummer	Semester	Raum	Platz

Aufgabe	1	2	3	4	Σ	Note
Punkte						

<u>Bitte beachten:</u>

Tragen Sie Ihre persönlichen Angaben auf dieses Deckblatt ein.

Schreiben Sie Ihre Antworten direkt in die dafür vorgesehenen freien Stellen des Angabentextes.

Geben Sie alle Blätter wieder ab, auch wenn einzelne Seiten nicht beschrieben sein sollten.

Alle Blätter der Angabe müssen bei der Abgabe wieder richtig sortiert und geheftet sein!

Viel Erfolg!

Aufgabe 1 (Verständnisfragen, ca. 20%)

a) Warum muss die Methode `main` eine statische Methode sein, und was bedeutet das?

b) Welchen Zweck hat der Bytecode in Java? Wäre es möglich, einen Java-Compiler zu schreiben, der Maschinencode erzeugt?

c) Mit welchem Operator wird eine Klasse instantiiert, und was bedeutet das?

d) Wodurch unterscheidet sich ein Interface von einer abstrakten Klasse?

e) Der folgende Java-Code enthält acht syntaktische und semantische Fehler. Führen Sie diese Fehler auf unter Angabe der Zeilennummer und des Fehlergrundes.

```
 1   class Klasse {
 2       private int x;
 3
 4       klasse(int x) {
 5           x = this.x;
 6       }
 7   }
 8
 9   class Unterklasse implements Klasse {
10       private int y;
11
12       Unterklasse(int i) {
13           y = i;
14           super(i);
15       }
16
17       Unterklasse(int i, int j) {
18           x = j;
19           this(i);
20       }
21
22       @Override String toString() {
23           return y;
24       }
25   }
```

Aufgabe 2 (Klassenhierarchie, ca. 25%)

Gegeben sei ein Computerspiel, dessen `Player` sich durch ihren `level` und ihren `value` voneinander unterscheiden:

```
enum Level {NOVICE, EXPERIENCED, EXPERT, MASTER, GRANDMASTER}

abstract class Player implements Comparable<Player> {
    private int value;
    private Level level;

    Player(int value, Level level) {
        this.value = value;
        this.level = level;
    }

    int   getValue() { return value; }
    Level getLevel() { return level; }









}
```

a) Ergänzen Sie die obenstehende Klasse um die noch fehlende `Comparable`-Implementierung, welche die `Player` hinsichtlich ihres `levels` vergleicht; bei gleichem `level` entscheidet der `value`.

b) Ergänzen Sie die nachstehenden Klassen um Konstruktoren, denen ein `Level` als Parameter übergeben wird. Ein `Knight` hat immer den `value` 2, ein `Wizard` hat immer den `value` 4.

```
   class Knight extends Player {




   }

   class Wizard extends Player {




   }
```

c) Überschreiben Sie in nachstehender Klasse die Methode `toString`, so dass bei deren Aufruf folgender `String` zurückgegeben wird:

`Total value of all players: xxx`

Für `xxx` ist die Summe aller `values` einzusetzen, die in diesem `Game` vorkommen.

```
class Game {
    private Player[][] game;

    Game(Player[][] game) { this.game = game; }









}
```

d) Schreiben Sie eine Methode `main`, die ein (8 x 8) `Game` erzeugt, mit folgender Vorbesetzung:

- `Knight` mit `Level` *`NOVICE`* auf Position (0,1)
- `Knight` mit `Level` *`EXPERT`* auf Position (2,4)
- `Wizard` mit `Level` *`MASTER`* auf Position (6,7)

Geben Sie das Ergebnis der `toString`-Methode dieses `Games` auf der Konsole aus.

Aufgabe 3 (Polymorphie, ca. 25%)

Gegeben sei folgendes Programm zur Berechnung ebener geometrischer Figuren mit drei und vier Ecken:

```
import java.util.*;

class Figuren {
    public static void main(String[] args) {
        List<Figur> figuren = new LinkedList<Figur>();
        figuren.add(new Viereck(2,2, 10,4, 9,8, 4,9));
        figuren.add(new Dreieck(1,2, 4,1, 3,5      ));
        Collections.sort(figuren);
        for (Figur f : figuren)
            System.out.println(f.toString());
    }
}
```

Dieses Programm gibt Folgendes auf der Konsole aus:

```
Dreieck: Fläche = 5.5
Viereck: Fläche = 35.5
```

Dreiecke werden durch drei Eckpunkte $P_1(x_1, y_1)$, $P_2(x_2, y_2)$, $P_3(x_3, y_3)$, Vierecke durch vier Eckpunkte $P_1(x_1, y_1)$, $P_2(x_2, y_2)$, $P_3(x_3, y_3)$, $P_4(x_4, y_4)$ definiert; die Punktkoordinaten werden dem Konstruktor der jeweiligen Klasse übergeben.

Die Fläche F eines Dreiecks bzw. eines Vierecks lässt sich wie folgt berechnen:

Dreieck: $F = ½ \mid (x_2 – x_1)\,(y_3 – y_1) – (x_3 – x_1)\,(y_2 – y_1) \mid$
Viereck: $F = ½ \mid (x_3 – x_1)\,(y_4 – y_2) + (x_4 – x_2)\,(y_1 – y_3) \mid$

Implementieren Sie die fehlenden Klassen so, dass für das oben gegebene Beispiel die oben gezeigte Ausgabe erzeugt wird.

Hinweise:

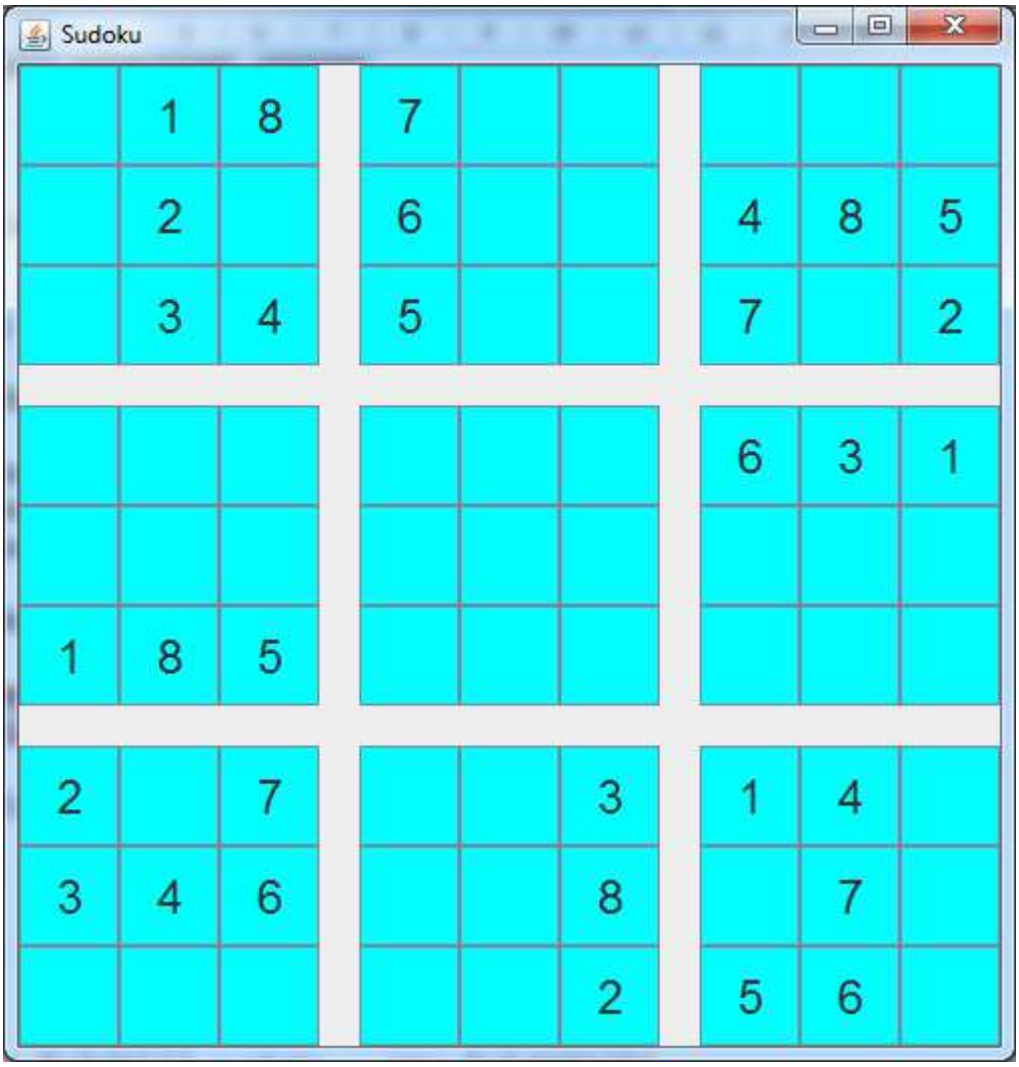
- Damit der Aufruf von `Collections.sort(figuren)` erlaubt ist, müssen alle Figuren miteinander vergleichbar sein. Der Vergleich soll anhand des Flächeninhalts erfolgen.
- Der Absolutbetrag kann mittels `double Math.abs(double x)` berechnet werden.

Aufgabe 4 (GUI, ca. 30 %)

Sudoku ist ein Spiel mit einem quadratischen Spielbrett, das in 3 x 3 Blöcke gegliedert ist, wobei jeder Block aus 3 x 3 Feldern besteht. Bei diesem Spiel besteht die Aufgabe darin, die Ziffern 1 bis 9 so in den 81 Feldern zu platzieren, dass in jeder Reihe, in jeder Spalte und in jedem Block die Ziffern 1 bis 9 jeweils genau einmal vorkommen.

Um eine eindeutige Lösung zu erhalten, wird eine Anfangsbelegung vorgegeben.

Beispiel:



In dieser Aufgabe sollen Sie nur die grafische Oberfläche, wie oben gezeigt, implementieren. Die Prüfung auf erlaubte Feldbelegungen gemäß den Sudoku-Regeln ist nicht Gegenstand dieser Aufgabe!

a) Ergänzen Sie die Klasse `Sudoku` um einen Konstruktor, der das Attribut `panels` gemäß der übergebenen Anfangsbelegung initialisiert und das Spielbrett wie oben gezeigt darstellt; dabei wird die Ziffer 0 als nicht belegtes `Feld` interpretiert.

Hinweis:
Im Konstruktor `GridLayout(int rows, int cols, int hgap, int vgap)` können mittels der Parameter `hgap` und `vgap` Zwischenräume (Einheit Punkte) zwischen den Zellen erzeugt werden.

b) Erweitern Sie das Programm um die Fähigkeit, durch Anklicken von Feldern Ziffern setzen bzw. ändern zu können. Jeder Klick soll die aktuelle Ziffer um 1 inkrementieren. Ein Klick auf ein unbesetztes Feld erzeugt die 1, ein Klick auf die 1 die 2 usw.; ist die 9 erreicht, so führt der nächste Klick wieder zu einem unbesetzten Feld (intern durch 0 dargestellt). Anfangs belegte Felder sollen allerdings unveränderbar sein. Bei Ihrer Lösung muss erkennbar sein, wo neue Anweisungen oder Methoden hinzugefügt werden.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class Feld extends JButton {
    private int zahl;

    Feld(int zahl) {
        setPreferredSize(new Dimension(50,50));
        setBackground(Color.CYAN);
        this.zahl = zahl;
        if (zahl != 0) setText("" + zahl);
    }
}

class Sudoku extends JFrame {
    private JPanel[][] panels = new JPanel[3][3];

    public static void main(String[] args) {
        new Sudoku(new int[][] {
            {0,1,8,  7,0,0,  0,0,0},
            {0,2,0,  6,0,0,  4,8,5},
            {0,3,4,  5,0,0,  7,0,2},

            {0,0,0,  0,0,0,  6,3,1},
            {0,0,0,  0,0,0,  0,0,0},
            {1,8,5,  0,0,0,  0,0,0},

            {2,0,7,  0,0,3,  1,4,0},
            {3,4,6,  0,0,8,  0,7,0},
            {0,0,0,  0,0,2,  5,6,0},
        });
    }
}
```