Prof. Dr. Jürgen Giesl
Carsten Fuhs, Peter Schneider-Kamp, Stephan Swiderski

Präsenzübung Programmierung
9. 1. 2008

Vorname: ______________________________

Nachname: ______________________________

Matrikelnummer: ______________________________

Studiengang (bitte ankreuzen):

○ Informatik Bachelor ○ Informatik Diplom ○ Informatik Lehramt
○ Mathematik Bachelor ○ Mathematik Diplom
○ CES Bachelor ○ CES Diplom
○ Werkstoffinformatik ○ Computermathematik

○ Sonstige: ______________________________

- Schreiben Sie bitte auf jedes Blatt **Vorname**, **Name** und **Matrikelnummer**.
- Geben Sie Ihre Antworten bitte in lesbarer und verständlicher Form an. Schreiben Sie bitte nicht mit roten Stiften oder mit Bleistiften.
- Bitte beantworten Sie die Aufgaben auf den **Aufgabenblättern**. Benutzen Sie ggf. auch die Rückseiten der **zur jeweiligen Aufgabe gehörenden** Aufgabenblätter.
- Antworten auf anderen Blättern können nur berücksichtigt werden, wenn **Name, Matrikelnummer und Aufgabennummer** deutlich darauf erkennbar sind.
- Was nicht bewertet werden soll, kennzeichnen Sie bitte durch **Durchstreichen**.
- Werden Täuschungsversuche beobachtet, so wird die Klausur mit **0 Punkten** bewertet.
- Geben Sie bitte am Ende der Klausur **alle Blätter** zusammen mit den Aufgabenblättern ab.

	Anzahl Punkte	Erreichte Punkte
Aufgabe 1	14	
Aufgabe 2	12	
Aufgabe 3	14	
Aufgabe 4	26	
Summe	66	
Prozentzahl		

Vorname	Name	Matr.-Nr.

Aufgabe 1 (Programmanalyse, 8 + 6 Punkte)

a) Geben Sie die Ausgabe des Programms für den Aufruf `java M` an. Schreiben Sie hierzu jeweils die ausgegebenen Zeichen hinter den Kommentar “`OUT:`”.

```
public class A {
    public static double x = 1;
    public A() {
        this(4);
    }
    public A(double x) {
        A.x += x;
    }
    public void f(double x) {
        x += 2*x;
    }
}

public class B extends A {
    public int y = 3;
    public B(int x) {
        super();
        y++;
    }
    public void f(int x) {
        A.x += x;
    }
    public void f(double x) {
        A.x -= x;
        y--;
    }
}

public class M {
    public static void main(String[] args) {
        A a = new A(A.x);
        System.out.println(a.x);           // OUT:
        a.f(10);
        System.out.println(a.x);           // OUT:
        B b = new B(10);
        System.out.println(b.x+" "+b.y); // OUT:
        b.f(10);
        System.out.println(b.x);           // OUT:
        a = b;
        a.f(1.0);
        System.out.println(a.x+" "+b.y); // OUT:
        a.f(10);
        System.out.println(a.x);           // OUT:
    }
}
```

Vorname	Name	Matr.-Nr.

b) Wir schreiben zusätzlich zu `A` und `B` eine neue Klasse `C`. Welche drei Fehler treten beim Compilieren auf? Begründen Sie Ihre Antwort kurz.

```
public class C extends B {
    private double y = 5;
    public C() {
        y++;
    }
    private void f(int x) {
        final int z;
        z = x;
        double x = 0.5*z;
        y += x;
    }
}
```

Vorname	Name	Matr.-Nr.

Aufgabe 2 (Verifikation, 10 + 2 Punkte**)**

Der Algorithmus P berechnet für ein Array a der Länge n die Summe $\sum_{k=0}^{n-1} 2^k * a[k]$, wobei a aus den Zahlen $a[0]$, …, $a[n-1]$ besteht.

a) Vervollständigen Sie die folgende Verifikation des Algorithmus P im Hoare-Kalkül, indem Sie die unterstrichenen Teile ergänzen. Hierbei dürfen zwei Zusicherungen nur dann direkt untereinander stehen, wenn die untere aus der oberen folgt. Hinter einer Programmanweisung darf nur eine Zusicherung stehen, wenn dies aus einer Regel des Hoare-Kalküls folgt.

Algorithmus: P
Eingabe: Ein Array a der Länge n, das die Zahlen $a[0]$, …, $a[n-1]$ enthält
Ausgabe: res
Vorbedingung: $n \geq 0$
Nachbedingung: $res = \sum_{k=0}^{n-1} 2^k * a[k]$

$\langle\ n \geq 0\ \rangle$

$\langle$ ____________________ $\rangle$

$i = 0;$

$\langle$ ____________________ $\rangle$

$res = 0;$

$\langle$ ____________________ $\rangle$

$z = 1;$

$\langle$ ____________________ $\rangle$

$\langle$ ____________________ $\rangle$

`while` $(n > i)$ $\{$

$\langle$ ____________________ $\rangle$

$\langle$ ____________________ $\rangle$

$res = res + z * a[i];$

$\langle$ ____________________ $\rangle$

$z = z * 2;$

$\langle$ ____________________ $\rangle$

$i = i + 1;$

$\langle$ ____________________ $\rangle$

$\}$

$\langle$ ____________________ $\rangle$

$\langle\ res = \sum_{k=0}^{n-1} 2^k * a[k]\ \rangle$

Vorname	Name	Matr.-Nr.

b) Beweisen Sie die Terminierung des Algorithmus P.

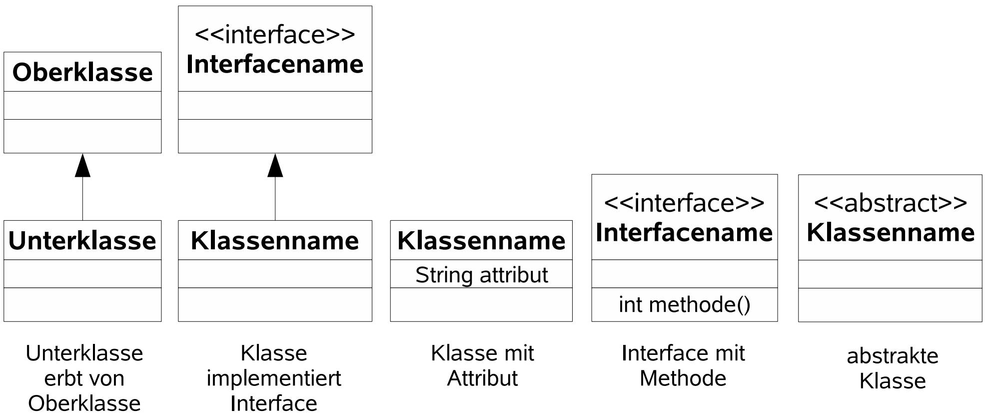
Vorname	Name	Matr.-Nr.

Aufgabe 3 (Datenstrukturen in Java, 6 + 8 Punkte**)**

Ihre Aufgabe ist es, eine objektorientierte Datenstruktur zur Verwaltung von Öfen zu entwerfen. Bei der vorangehenden Analyse wurden folgende Eigenschaften der verschiedenen Öfen ermittelt.

- Jeder Ofen wird durch die maximale Temperatur gekennzeichnet, die er im regulären Betrieb erreichen darf.
- Ein Kohleofen ist ein Ofen, der eine Vorratskammer mit einem bestimmten Volumen hat. Dabei wird unterschieden zwischen solchen Kohleöfen, in denen auch Steinkohle verbrannt werden kann, und Kohleöfen, in denen keine Steinkohle verbrannt werden kann.
- Ein Ölofen ist ebenfalls ein Ofen, der eine Vorratskammer mit einem bestimmten Volumen hat. Für einen Ölofen ist die so genannte Rußzahl eine wichtige Kenngröße.
- Neben den Öfen mit Vorratskammer gibt es auch Öfen mit Anschluss an ein Versorgungsnetz. Für diese Öfen ist die monatliche Grundgebühr für die Sicherstellung der Versorgung ein wichtiges Attribut.
- Ein Elektroofen ist ein Ofen mit Anschluss an ein Versorgungsnetz für elektrischen Strom. Für Elektroöfen ist der Wert der elektrischen Spannung interessant, mit der der jeweilige Elektroofen betrieben wird.
- Gasöfen sind Öfen, die an ein Gasversorgungsnetz angeschlossen werden. Bei solchen Gasöfen ist die Art des Gases anzugeben, das vom jeweiligen Ofen verbrannt werden kann.
- Sowohl Gasöfen als auch alle Öfen mit einer Vorratskammer stoßen Abgase aus. Daher stellen sie eine Methode zur Bestimmung des CO-Anteils in den Abgasen zur Verfügung (CO = Kohlenmonoxid).

a) Entwerfen Sie unter Berücksichtigung der Prinzipien der Datenkapselung eine geeignete Klassenhierarchie für die oben aufgelisteten Arten von Öfen. Achten Sie darauf, dass gemeinsame Merkmale in (evtl. abstrakten) Oberklassen zusammengefasst werden. Notieren Sie Ihren Entwurf graphisch und verwenden Sie dazu die folgende Notation:



Geben Sie für jede Klasse ausschließlich den jeweiligen Namen und die Namen und Datentypen ihrer Attribute an. Methoden von Klassen müssen nicht angegeben werden. Geben Sie für jedes Interface ausschließlich den jeweiligen Namen sowie die Namen und Ein- und Ausgabetypen seiner Methoden an.

Vorname	Name	Matr.-Nr.

Vorname	Name	Matr.-Nr.

b) Implementieren Sie in Java eine Methode `saubereOefen`. Die Methode bekommt als Parameter ein Array von Öfen und einen `double`-Wert `maxCO` übergeben. Sie soll ein Array von denjenigen Öfen aus dem gegebenen Array zurückliefern, die keine Abgase haben oder deren Abgase einen CO-Anteil haben, der höchstens `maxCO` ist. Das übergebene Array soll dabei nicht modifiziert werden. Das Array, das Sie zurückgeben, soll keine `null`-Werte enthalten.

Gehen Sie dabei davon aus, dass das übergebene Array nicht der `null`-Wert ist, dass es keine `null`-Werte enthält und dass für alle Attribute geeignete Selektoren existieren. Verwenden Sie für den Zugriff auf die benötigten Attribute die passenden Selektoren und kennzeichnen Sie die Methode mit dem Schlüsselwort „`static`“, falls angebracht.

Vorname	Name	Matr.-Nr.

Aufgabe 4 (Rekursive Datenstrukturen, 6 + 8 + 12 Punkte)

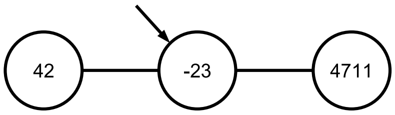
Die Klasse `Knoten` repräsentiert einen Knoten in einem Graph. Das Attribut `nachbarn` enthält die Nachbarknoten als Array. Es ist zu beachten, dass dieses Array `null`-Zeiger enthalten kann. Wenn ein Knoten `A` Nachbar eines Knoten `B` ist, so ist `B` auch Nachbar von `A`, d.h., die Nachbarschaftsbeziehung ist symmetrisch. Jeder Knoten kann einen Wert vom Typ `int` speichern. Außerdem speichert jeder Knoten einen booleschen Wert `besucht`. Es muss davon ausgegangen werden, dass diese Graphen Zyklen enthalten können.

```
public class Knoten {
    public Knoten[] nachbarn;
    public int wert;
    public boolean besucht;
}
```

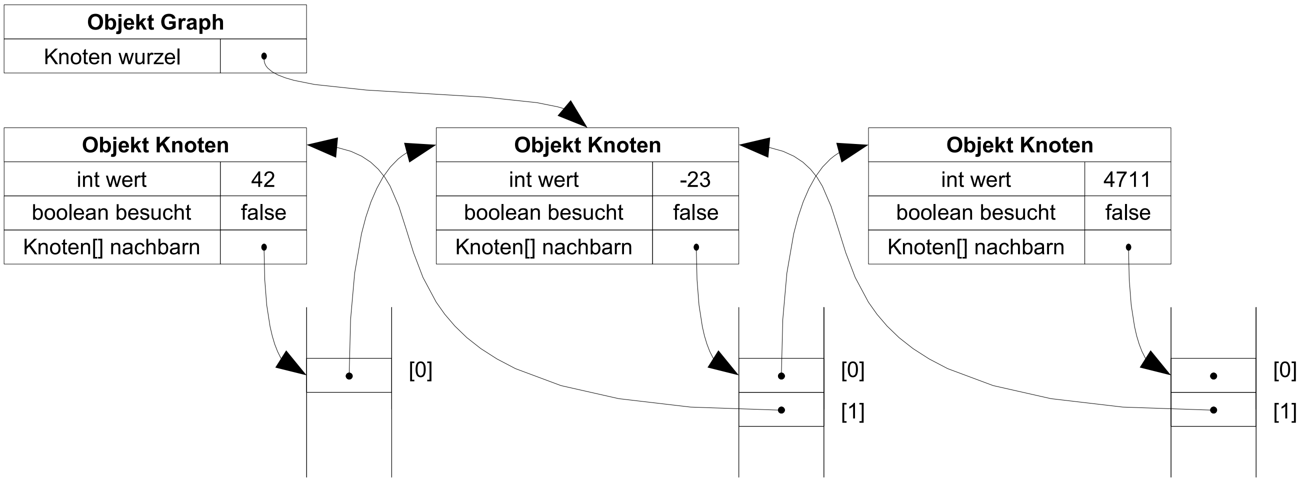
Die Klasse `Graph` repräsentiert einen Graph. Das Attribut `wurzel` der Klasse `Graph` ist der Einstiegspunkt in die Knotenstruktur des Graphen. Ein leerer Graph enthält keine Knoten, so dass das Attribut `wurzel` auf `null` zeigt. Die Methode `besuchtZuruecksetzen` setzt in allen `Knoten` des Graphen das Attribut `besucht` auf `false`. Sie können die Methode `besuchtZuruecksetzen` als gegeben annehmen und auch in Ihren Lösungen verwenden.

```
public class Graph {
    public Knoten wurzel;
    private void besuchtZuruecksetzen(){...}
}
```

Zum Beispiel könnte ein Graph drei Knoten mit den Werten `42`, `-23` und `4711` enthalten, wobei die Wurzel auf den Knoten mit dem Wert `-23` zeigt:



Der Graph könnte dann zum Beispiel folgende Struktur im Speicher besitzen. Hierbei ist beim Knoten mit dem Wert `4711` der erste Eintrag im Array `nachbarn` ein Zeiger auf `null`.



Vorname	Name	Matr.-Nr.

a) Die Methode `loesche` der Klasse `Graph` löscht einen Knoten aus dem Graphen. Danach kann es vorkommen, dass einige Knoten vom Wurzelknoten aus nicht mehr erreichbar sind, so dass diese ebenfalls automatisch gelöscht sind. Sie brauchen diese nicht mehr erreichbaren Knoten nicht manuell zu löschen. Beachten Sie, dass, wenn der Wurzelknoten gelöscht wird, der Graph danach leer sein muss. Nutzen Sie bei der Implementierung aus, dass die Nachbarschaftsbeziehung symmetrisch ist. (Sie müssen daher den zu löschenden Knoten nicht im Graph suchen, sondern Sie können direkt vom zu löschenden Knoten ausgehen und von dort aus seine Nachbarn erreichen.) Implementieren Sie die Methode `loesche` mit der Signatur `public void loesche(Knoten knoten)` in der Klasse `Graph`.

Vorname	Name	Matr.-Nr.

b) Die Methode `anwenden` der Klasse `Graph` wendet einen Besucher auf den Graph an. Ein Besucher ist ein Objekt einer Klasse, welche das Interface `Besucher` implementiert.

```
public interface Besucher {
    void besuche(Knoten knoten);
}
```

Die Methode `besuche` des Besuchers soll durch die Methode `anwenden` auf jeden Knoten *genau einmal* angewendet werden. Sie können hier das Attribut `besucht` der Klasse `Knoten` verwenden, indem Sie es bei einem besuchten Knoten auf `true` setzen. Sie können davon ausgehen, dass das Attribut `besucht` vor dem Aufruf von `anwenden` in jedem Knoten des Graphen auf `false` steht. Sorgen Sie dafür, dass das Attribut `besucht` jedes Knotens nach Verlassen der Methode `anwenden` wieder auf `false` steht. Implementieren Sie die Methode `anwenden` mit der Signatur `public void anwenden(Besucher besucher)` in der Klasse `Graph`. Verwenden Sie dabei ausschließlich Rekursion. Sie dürfen aber eine `for`-Schleife verwenden, um ein Array zu durchlaufen.

Vorname	Name	Matr.-Nr.

c) Die Methode `loescheMaximum` der Klasse `Graph` sucht einen Knoten im Graph mit dem maximalen Attribut `wert` und löscht diesen dann. Sie hat die folgende Signatur:

```
public void loescheMaximum()
```

Implementieren Sie hierzu die Klasse `MaximumFinder`, welche das Interface `Besucher` implementiert.

```
public interface Besucher {
    void besuche(Knoten knoten);
}
```

Nachdem ein `MaximumFinder` auf den Graph angewendet worden ist, soll das Attribut `maximumKnoten` vom `MaximumFinder` einen Knoten mit dem Maximum enthalten. Dieser Knoten soll dann gelöscht werden. Das Attribut `maximumKnoten` von Objekten der Klasse `MaximumFinder` darf dazu die Sichtbarkeit `public` haben. Implementieren Sie die Methode `loescheMaximum` in der Klasse `Graph`, die die von Ihnen implementierte Klasse `MaximumFinder` benutzt.