Institut für Informatik der Ludwig-Maximilians-Universität München
Prof. Martin Hofmann, PhD
Dr. Steffen Jost

WiSe 2015/16
16.02.2016

Lösung zur Klausur zur Vorlesung

# Einführung in die Programmierung

Die Bearbeitungszeit beträgt 120 min. Hilfsmittel sind nicht erlaubt, auch der Besitz ausgeschalteter elektronischer Geräte wird als Betrug gewertet. Schreiben Sie Ihren vollständigen Namen und Ihre Matrikelnummer deutlich lesbar auf dieses Deckblatt, sowie Ihren Namen in die Kopfzeile auf **jedem Blatt** der Klausurangabe! Geben Sie alle Blätter ab, lassen Sie diese zusammengeheftet! Verwenden Sie nur dokumentenechte Stifte und weder die Farben rot noch grün.

Kontrollieren Sie, ob Sie alle Aufgabenblätter erhalten haben. Aufgabenstellungen befinden sich auf den Seiten 1–13. Sie dürfen die Rückseiten für Nebenrechnungen nutzen. Falls Sie die Rückseiten für Antworten nutzen, so markieren Sie klar, was zu welcher Aufgabe gehört und schreiben Sie in der entsprechenden Aufgabe, wo alle Teile Ihrer Antwort zu finden sind. Streichen Sie alles durch, was nicht korrigiert werden soll!

Lesen Sie die Aufgabenstellungen vollständig durch, bevor Sie mit der Bearbeitung der Aufgaben beginnen. Es gibt 7 unterschiedlich gewichtete Aufgaben zu insgesamt 51 Punkten. Mit 26 Punkten haben Sie sicherlich bestanden. Die Teilaufgaben können unabhängig voneinander bearbeitet werden.

Mit Ihrer Unterschrift bestätigen Sie, dass Sie zu Beginn der Klausur in ausreichend guter gesundheitlicher Verfassung sind, und diese Klausurprüfung verbindlich annehmen.

## Nachname:

## Vorname:

## Matrikelnummer:

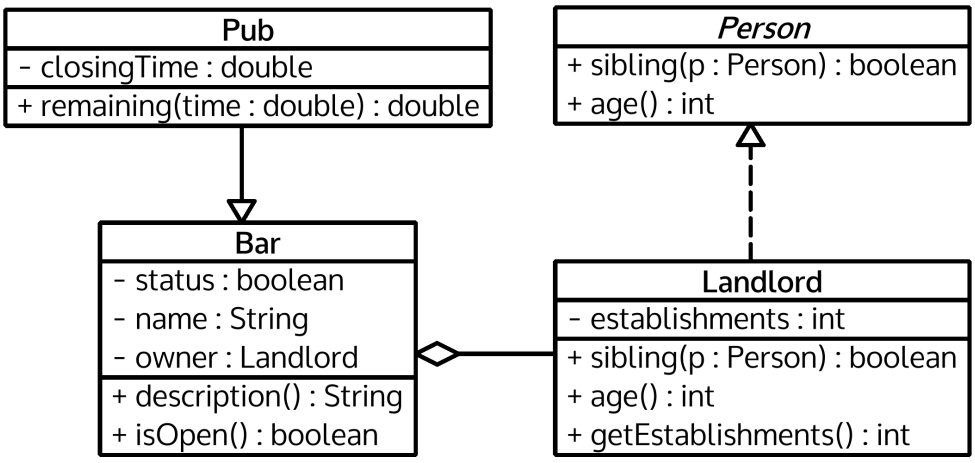
## Studiengang:

Hiermit erkläre ich die Richtigkeit der obigen Angaben: _____
(Unterschrift)

| Aufgabe | 1 | 2 | 3 | 4 | 5 | 6 | 7 | $\sum$ | Bonus | Note |
|---|---|---|---|---|---|---|---|---|---|---|
| Punkte | 8 | 6 | 8 | 8 | 6 | 8 | 7 | **51** | | |
| erreicht | | | | | | | | | | |

**Lösung Aufgabe 1 (UML):** **(8 Punkte)**

**a)** Gegeben ist folgendes UML Klassendiagramm:




Geben Sie eine möglichst vollständige, passende Implementierung an!

*Hinweis:* Geben Sie keine Konstruktoren und keine Methodenrümpfe an (Klassen brauchen aber nicht als `abstract` deklariert werden). Übersetzung: Landlord = Besitzer, Sibling = Geschwister.

**LÖSUNG:** Zum Lesen dieses UML Klassendiagramms musste man lediglich wissen, welcher Pfeil Vererbung und welcher für Interface-Implementierung steht (die Kursivschrift bei `Person` gibt auch noch einen starken Hinweis). Den Aggregationspfeil kann man sich herleiten, da die Instanzvariable `owner` ja mit Typ `Landlord` angegeben ist. Viele verschwendeten hier Zeit durch Hinschreiben von Konstruktoren und Methodenrümpfen, obwohl dies explizit nicht verlangt war!

```
public class Pub extends Bar {
    private double closingTime;

    public double remaining(double time){
      ...}
}

public class Bar {

    private boolean status;
    private String name;
    private Landlord owner;

    public String description() {...}
    public boolean isOpen() {...}
}
```

```
public interface Person {

    public boolean sibling(Person p);

    public int age();

}

public class Landlord implements Person {
    private int establishments;

    @Override
    public boolean sibling(Person p) {...}
    @Override
    public int age() {...}
    public int getEstablishments() {...}
}
```

*Die Aufgabe wird auf dem nächsten Blatt fortgesetzt.*

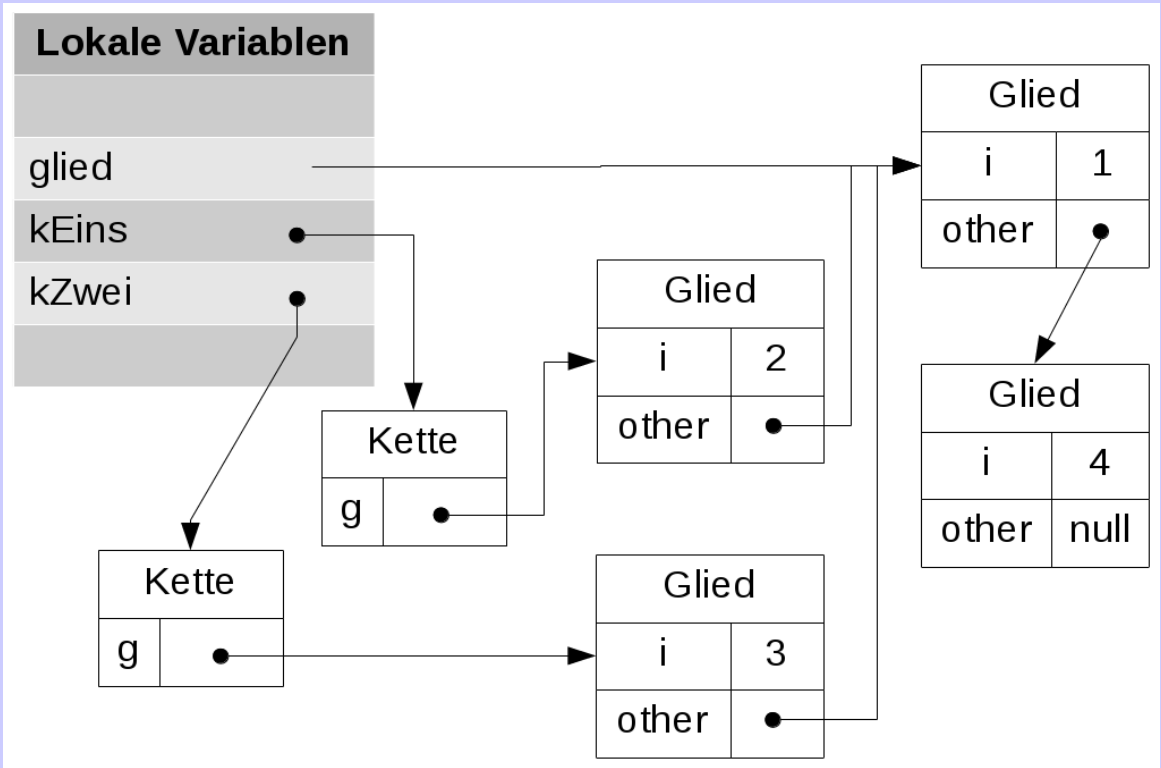
*Fortsetzung von Aufgabe 1:*

**b)** Gegeben sind folgende Definitionen und eine `main`-Methode. Zeichnen Sie das Speicherdiagramm unmittelbar *vor* der letzten Anweisung der `main`-Methode. *Hinweise:* Geben Sie den vollständigen Zustand inklusive aller lokaler Variablen der `main`-Methode an; mögliche Garbage Collection wird ignoriert.

```
public static void
  main(String[] args) {
    Glied glied = new Glied(1,null);
    // Zustand der Vorgabe (unten)
    Kette kEins = new Kette(glied);
    Kette kZwei = new Kette(glied);
    kEins.cons(2);
    kZwei.cons(3);
    kZwei.snoc(4);
    // Zustand hier malen!
    System.out.println("So isses!");
}
```

```
public class Glied {
    int   i;
    Glied other;

    public Glied(int i, Glied other) {
      this.i = i;
      this.other = other;
}   }

public class Kette {
    Glied g;

    public Kette(Glied g) { this.g = g; }

    public void cons(int i) { g=new Glied(i,g); }

    public boolean snoc(int i) {
        Glied p = this.g;
        if (p == null) return false;
        while (p.other != null) { p=p.other; }
        p.other = new Glied(i,null);
        return true;
}   }
```

**LÖSUNG:** Die Themen "einfach verkettete Listen" und "Speicherdiagramme"sollten niemanden überrascht haben. Die `cons`-Methode besteht nur aus einer einzelnen Zuweisung und ist daher so einfach wie nur möglich. Die `snoc`-Methode ist etwas schwieriger, wenn man das Funktionsprinzip eines Listen-Iterators schon vergessen hatte — dann ist hier wohl etwas Fleissarbeit nötig.



**Lösung Aufgabe 2 (Backus-Naur Form):** **(6 Punkte)**

Gegeben sind folgende Produktionen einer BNF-Grammatik mit dem Startsymbol ⟨*Emoticon*⟩:

$$\begin{aligned}
\langle \textit{Emoticon} \rangle &::= \langle \textit{FröhlicherEmoticon} \rangle \mid \langle \textit{TraurigerEmoticon} \rangle \\
\langle \textit{FröhlicherEmoticon} \rangle &::= (\text{“:”} \mid \text{“;”})\ [\langle \textit{Nase} \rangle]\ \langle \textit{FröhlicherMund} \rangle \\
\langle \textit{TraurigerEmoticon} \rangle &::= (\text{“:”} \mid \text{“B”})\ [\langle \textit{Nase} \rangle]\ \langle \textit{TraurigerMund} \rangle \\
\langle \textit{Nase} \rangle &::= \text{“o”} \mid \text{“-”} \\
\langle \textit{FröhlicherMund} \rangle &::= \text{“)”} \mid \text{“))”} \mid \text{“D”} \\
\langle \textit{TraurigerMund} \rangle &::= (\text{“(”})^+ \mid \text{“|”}
\end{aligned}$$

**a)** Entscheiden Sie für die folgenden Wörter jeweils, ob sie in der Sprache dieser Grammatik sind. Begründen Sie Ihre Antwort! Geben Sie weiterhin für Wörter, die in der Sprache sind, eine ausführliche Ableitung an! Führen Sie dabei jeden Schritt *einzeln* aus!

(a) `;-(`

**LÖSUNG:** *Sehr ähnlich zu H4-1a:* “`;`” kommt nur in ⟨*FröhlicherEmoticon*⟩ vor, welches Nonterminal ⟨*FröhlicherMund*⟩ verlangt, doch dort kommt kein “`(`” vor.

(b) `Bo|`

**LÖSUNG:**

$$\begin{aligned}
\langle \textit{Emoticon} \rangle &\to \langle \textit{FröhlicherEmoticon} \rangle \mid \langle \textit{TraurigerEmoticon} \rangle \to \langle \textit{TraurigerEmoticon} \rangle \\
&\to (\text{“:”} \mid \text{“B”})\ [\langle \textit{Nase} \rangle]\ \langle \textit{TraurigerMund} \rangle \to \text{“B”}\ [\langle \textit{Nase} \rangle]\ \langle \textit{TraurigerMund} \rangle \\
&\to \text{“B”}\ \langle \textit{Nase} \rangle\ \langle \textit{TraurigerMund} \rangle \\
&\to \text{“B”}\ (\text{“o”} \mid \text{“-”})\ \langle \textit{TraurigerMund} \rangle \to \text{“B”}\ \text{“o”}\ \langle \textit{TraurigerMund} \rangle \\
&\to \text{“B”}\ \text{“o”}\ ((\text{“(”})^+ \mid \text{“|”}) \to \text{“B”}\ \text{“o”}\ \text{“|”}
\end{aligned}$$

**b)** Geben Sie einen ⟨*Emoticon*⟩ in der Sprache dieser Grammatik an, welcher keine ⟨*Nase*⟩ enthält und aus genau 4 Zeichen (Terminalsymbolen) besteht.

**LÖSUNG:** *Sehr ähnlich zu H4-1b:* `:(((` oder `=(((`. Nicht richtig: `:-))`, `:-((`, `:))`, ...

**c)** Ändern Sie die gegebenen Produktionen so ab, dass Emoticons mit Haaren “`=`” und Bart “`}`” erlaubt sind, aber nur, wenn Haare und Bart exakt gleich lang sind (gleiche Anzahl Zeichen)!
Beispiele, welche ableitbar sein sollen: “`=:-)}`” , “`==;D}}`” , “`===Bo|}}}`” , “`;-)`” , etc.
Gegenbeispiele, welche *nicht* ableitbar sein sollen: “`=:)`” , “`=:)}}`” , “`:o((}`” , etc.
Geben Sie nur die Produktionen für *abgeänderte* Nonterminale an!

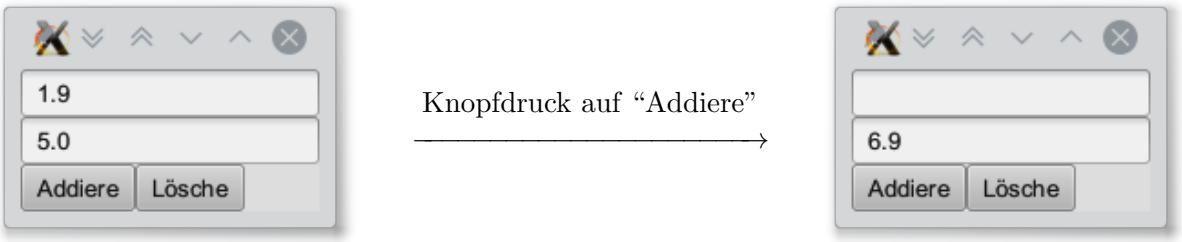
**LÖSUNG:** *Vereinfachte Version von A4-1a: Anstatt mit “≫” und “≪” wird der Ausdruck mit den Terminalsymbolen “`=`” und “`}`” geklammert.*

$$\langle \textit{Emoticon} \rangle ::= \langle \textit{FröhlicherEmoticon} \rangle \mid \langle \textit{TraurigerEmoticon} \rangle \mid \text{“=”}\ \langle \textit{Emoticon} \rangle\ \text{“}\}\text{”}$$

Kein Punktabzug für unnötig kompliziertere indirekte Lösungen (wie in A4-1a notwendig).

**Lösung Aufgabe 3 (Benutzeroberflächen):** **(8 Punkte)**

**a)** Wir wollen einen simplen Taschenrechner schreiben, welcher einfach nur Zahlen aufaddieren kann:




Die GUI besteht aus einem Textfeld zur Eingabe einer Zahl; ein Textfeld zur Anzeige der intern gespeicherten Summe; ein Knopf zum Addieren der im oberen Textfeld befindlichen Zahl zur Summe, welcher anschließend das obere Textfeld leert; und noch ein Knopf zum Rücksetzen der Summe auf 0.

Vervollständigen Sie dementsprechend das nachfolgende Programmgerüst. Folgendes ist noch zu tun:

- Der Knopf zum Addieren fehlt noch komplett. Fügen Sie diesen hinzu!
- Stellen Sie sicher, dass beide Knöpfe auch funktionieren!
  (Der in der Vorgabe enthaltene Knopf wird bis jetzt nur angezeigt, hat aber noch keine Wirkung.)

*Hinweise:* Zur Vereinfachung der Aufgabe wird fast alles in einer Methode implementiert. Zur vollständigen Lösung reicht es also, ein paar Zeilen an der mit `// *** TODO ***` gekennzeichneten Stelle einzufügen. Sie dürfen aber auch neue Methoden definieren, wenn Sie möchten. Die Anwendung der Entwurfsmuster MVC und/oder Observer sind nicht notwendig.

```
public class Main extends Application {
  private  double    summe;
  private  TextField eingabeFeld;
  private  TextField ausgabeFeld;
  private  Button    clrButton;
  private  Button    addButton;

  public static void main(String[] args) { launch(args); }
  public void start(Stage primaryStage) throws Exception{
    summe            = 0;
    eingabeFeld      = new TextField("");
    ausgabeFeld      = new TextField("0");
    clrButton        = new Button("Lösche");
    GridPane scheibe = new GridPane();
    scheibe.add(eingabeFeld,0,0,2,1);      // (Spalte 0, Zeile 0, Breite 2, Höhe 1)
    scheibe.add(ausgabeFeld,0,1,2,1);      // (Spalte 0, Zeile 1, Breite 2, Höhe 1)
    scheibe.add(clrButton,1,2);            // (Spalte 1, Zeile 2)
    // *** TODO ***
    primaryStage.setScene(new Scene(scheibe));
    primaryStage.show();
  }
  public double getEingabe() {  return Double.parseDouble(eingabeFeld.getText());  }
  public void   showSumme()  {  ausgabeFeld.setText(""+summe);  }
}
```

*Die Aufgabe wird auf dem nächsten Blatt fortgesetzt.*

*Fortsetzung von Aufgabe 3:*

Schreiben Sie hier Ihre Lösung zu Aufgabenteil **a)** hin:

**LÖSUNG:** Folgende Zeilen sind bei `// *** TODO ***`

```
addButton = new Button("Addiere");  // Neuer Knopf (speichern in Instanz kein muss)
scheibe.add(addButton,0,2);         // Zur Scheibe bei Spalte 0, Zeile 2 einfügen
addButton.setOnAction(event -> {    // Event-Handler für addButton
    summe += getEingabe();             // Interne Summe erhöhen
    showSumme();                       // Summen-Anzeige aktualisieren
    eingabeFeld.setText("");           // Eingabe-Feld leeren (auch zuerst möglich)
});
clrButton.setOnAction(event -> {    // Event-Handler für clrButton
    summe=0;                           // Summe zurücksetzen
    showSumme();                       // Summen-Anzeige aktualisieren
});
```

Wer die Hausübungen zu dem Thema GUI gemacht hat, sollte hier leicht Punkte verdienen. Aber auch ohne Ahnung von GUIs konnte man zumindest mit Copy-Paste von clrButton zu addButton einen Punkt erzielen.

Es gab keinen Punktabzug, wenn man sich nicht mehr genau an `setOnAction` erinnerte, sondern `button.setIrgendwas` hinschrieb. Der Lambda-Ausdruck war kein Zwang (z.B. Anonyme Klasse).

**b)** Erklären Sie möglichst kurz und knapp das Entwurfsmuster "Beobachter" (engl.: Observer-Pattern). Geben Sie auch an, welche Klassen und/oder Schnittstellen die Java-Standardbibliothek dazu bereitstellt, inklusive angebotenen Methoden. (Es reichen uns die insgesamt 4 in der Vorlesung behandelten Methoden; zur Beschreibung der Methoden reichen jeweils Name und 2–7 Worte, z.B.: `toString`: wandelt Objekt in String um)

**LÖSUNG:** Bei Entwurfsmuster "Beobachter" benachrichtigt ein *beobachtetes* Objekt bei jeder relevanten Zustandsänderung ein (oder mehrere) Beobachter-Objekte, damit diese auf die Zustandsänderung reagieren können, ohne selbst ständig den Zustand des beobachteten Objekts prüfen zu müssen.

Beobachtete Objekte gehören einer Klasse an, welche von der Klasse `Obervable` erbt; Beobachter müssen die Schnittstelle `Oberver` implementieren. Die wichtigsten Methoden sind:

```
class Observable {
  void addOberver(Observer o)           // Beobachter hinzufügen
  void setChanged()                     // Zustandsänderung anzeigen
  void notifyObservers(Object arg)      // Beobachter benachrichtigen
}

interface Observer {
  void update(Observable o, Object arg); // Wird bei Zustandsänderung aufgerufen.
}
```

*Auch hier wurden "ähnliche" Namen akzeptiert, solange die Eckpunkte stimmen und die Funktionsweise anhand des Namens oder durch Kommentare erkennbar blieb.*

**Lösung Aufgabe 4 (Arrays rekursiv sortieren): (8 Punkte)**

Betrachten Sie folgende fehlerhafte Implementierung für das Sortieren durch Mischen (Merge Sort):

```
01    public static void mergeSort(int[] a, int from, int to) {
02      if (from >= to) return;
03      int mid = (from + to) / 2;
04      mergeSort(a, mid + 1, to);
05      mergeSort(a, from, mid);
06      merge(a, from, mid, to);
07    }
08
09    public static void merge(int[] a,  int from, int mid, int to) {
10      int   n = to - from + 1;
11      int[] b = new int[n];   // Kopie des zu sortierenden Bereichs
12      for (int j = 0; j<n; j++) {  b[j] = a[from + j];  }
13      int j  = from;
14      int m  = mid - from;
15      int i1 = 0;
16      int i2 = m;
17      while (i1 < m && i2 < n)
18        { if (b[i1] < b[i2]) {  a[j] = b[i1]; i1++;  }
19          else               {  a[j] = b[i2]; i2++;  }
20          j++;
21        }
22      while (i1 < m) {  a[j] = b[i1]; i1++; j++;  }
23      while (i2 < n) {  a[j] = b[i2]; i2++; j++;  }
24    }
```

Weiterhin betrachten wir folgendes Fragment des Hauptprogramms:

```
30    int[] a1 = new int[]{3,1,2};
31    int[] a2 = new int[]{3,2,1};
```

Der Aufruf `mergeSort(a1,0,2)` sortiert `a1` wie erwartet zu `[1,2,3]`. Dagegen liefert der Aufruf `mergeSort(a2,0,2)` zwar keine Ausnahme, aber jedoch ein falsches Ergebnis!

**LÖSUNG:** *In der Probeklausur hatten wir diese Aufgabe mit SelectionSort; jetzt kam der andere behandelte Sortier-Algorithmus dran. Für a) und b) muss man aber erst mal nur wissen, wie in Java ein simpler Methodenaufruf und Integer-Division funktioniert. Bei c) und d) braucht man wohl eine Nebenrechnung auf Papier – hier droht eine gewisse Gefahr durch verzetteln.*

**a)** Welche rekursiven Aufrufe der Methode `mergeSort` werden durch den Aufruf `mergeSort(a2,0,2)` ausgeführt? *Hinweis: Es sind mehr als 2 rekursive Aufrufe!*

**LÖSUNG:** `mergeSort(a2,2,2)`, `mergeSort(a2,0,1)`, `mergeSort(a2,1,1)` und `mergeSort(a2,0,0)`.

*Die Aufgabe wird auf dem nächsten Blatt fortgesetzt.*

*Fortsetzung von Aufgabe 4:*

**b)** Welche Aufrufe der Methode `merge` werden durch einen Aufruf von `mergeSort(a2,0,2)` ausgeführt? Geben Sie diese in der Reihenfolge an, in der diese auch ausgeführt werden!

**LÖSUNG:** Zuerst `merge(a2,0,0,1)` und danach `merge(a2,0,1,2)`.

**c)** Wie sieht das Array `a2` nach vollständiger Beendigung des Aufruf von `mergeSort(a2,0,2)` aus?

**LÖSUNG:** `a2 = [2,1,3]`

**d)** Korrigieren Sie den Fehler, so dass `int[]`-Arrays korrekt sortiert werden!
*Hinweis: Es reicht aus, eine einzelne Zeile zu verändern!*

**LÖSUNG:**
```
14      int m  = mid - from + 1;
```

**e)** Welche Zeitkomplexität (in $O$-Notation) hat der gewöhnliche Merge-Sort Algorithmus im schlechtesten Fall?

**LÖSUNG:** $O(n \log n)$ *Buchwissen; aber auch herleitbar anhand des gegebenen Codes.*

**Lösung Aufgabe 5 (Vererbung):** **(6 Punkte)**

Betrachten Sie die folgende Klassendefinitionen:

```
public class Gran {
    private int x;
    public Gran() {  this.x = 68;  }
    public int age() {  this.x = this.x+1; return this.x;  }
    @Override
    public String toString() {  return "Gran " + age();  }
}
public class Dad extends Gran {
    private int x;
    public Dad() {  this.x = 41;  }
    @Override
    public String toString() {  return "Dad " + age();   }
}
public class Bro extends Dad {
    private int x;
    public Bro() {  this.x = 21;  }
    @Override
    public int age() {  System.out.print("Bro "); return x;  }
}
public class Sis extends Dad {
    private int x;
    public Sis() {  this.x = 17;  }
    @Override
    public int age() {  System.out.print("Sis "); return super.age() - x;  }
    @Override
    public String toString() {  return "Sis " + super.toString();  }
}
```

Jetzt wird folgender Code ausgeführt.

```
Gran[] family = new Gran[] {new Gran(), new Dad(), new Bro(), new Sis()};
for (Gran member : family) System.out.println(member.toString());
```

Welche Ausgabe wird davon erzeugt?

*Hinweise:* Die zweite Zeile der Ausgabe ist *nicht* `"Dad 41"`;
die vierte Zeile der Ausgabe hat nicht die Form `"Sis` $n$`"` für eine Zahl $n$.

**LÖSUNG:**

```
Gran 69
Dad 69
Bro Dad 21
Sis Sis Dad 52
```

Entgegen Übungsaufgaben fehlte hier die Schwierigkeit, zwischen Überladen und Überschreiben zu unterscheiden: hier gab es *nur* Überschreiben, wie die `@Override`-Annotate direkt anzeigen! Das bei Vererbung alle ererbten Instanzvariablen versteckt vorhanden bleiben, war ja ein Thema in den Wiederholungsvorlesungen gewesen.

Eine von uns unerwartete Schwierigkeit bereitete vielen das mangelnde Grundverständnis des Programmablaufs in Java: Vielen scheint trotz allen Programmieraufgaben immer noch unklar zu sein, welche Ausgaben die Anweisung `System.out.print("B"+foo()+"D");` erzeugt, mit `public String foo(){ System.out.print("A"); return "C"; }`. Das erschwert natürlich viel!

**Lösung Aufgabe 6 (Nebenläufigkeit):** **(8 Punkte)**

Die Mitarbeiter des TCS Lehrstuhls trinken ständig Kaffee. Leider kann jedoch immer nur ein Mitarbeiter den Kaffeeautomaten bedienen: Der Mitarbeiter wählt ein Kaffee-Produkt aus, wartet bis es fertig ist, und entnimmt sein Produkt danach. Erst dann darf der nächste Mitarbeiter seinen nächsten Kaffee anfordern. Dies haben wir wie folgt modelliert:

```
 1  public class CoffeeMachine {
 2    private boolean working;
 3
 4    public CoffeeMachine() {
 5      this.working = false;
 6    }
 7
 8    public void makeCoffee(int nummer) {
 9      while (working) {
10        // warten
11      }
12      System.out.println("Benutzer " + nummer + " bestellt Kaffee.");
13      working = true;
14    }
15
16    public void takeCoffee(int nummer) {
17      System.out.println("Benutzer " + nummer + " entnimmt Tasse.");
18      working = false;
19    }
20  }

31  public class Benutzer implements Runnable {
32    private CoffeeMachine machine;
33    private int nummer;
34
35    public Benutzer(CoffeeMachine machine, int nummer){
36      this.machine = machine;
37      this.nummer   = nummer;
38    }
39
40    public void run() {
41      while (true) {
42        machine.makeCoffee(this.nummer);
43        machine.takeCoffee(this.nummer);
44      }
45    }
46  }
```

Jetzt wird folgender Code ausgeführt:

```
61  CoffeeMachine machine = new CoffeeMachine();
62  new Thread(new Benutzer(machine, 1)).start();
63  new Thread(new Benutzer(machine, 2)).start();
```

*Die Aufgabe wird auf dem nächsten Blatt fortgesetzt.*

*Fortsetzung von Aufgabe 6:*

**a)** Geben Sie den Anfang einer Ausgabe eines Programmablaufs an, der einer illegalen Benutzung des Kaffeeautomaten entspricht. Diese soll so kurz wie möglich sein, d.h. die Verletzung der Bedingung, dass zu jedem Zeitpunkt nur ein Benutzer den Kaffeeautomaten benutzt, soll so früh wie möglich auftreten. Erklären Sie auch kurz, wie es zu diesem Ablauf kam.

**LÖSUNG:**

```
Benutzer 1 bestellt Kaffee.
Benutzer 2 bestellt Kaffee.
```

Benutzer 1 wird innerhalb Methode `makeCoffee` nach dem print-out unterbrochen, jedoch noch bevor `working = true;` ausgeführt werden konnte.

Für die restlichen Teilaufgaben b)–d) betrachten wir eine leichte Veränderung der Klasse `CoffeeMachine`, in der die beiden Methoden `makeCoffee` und `takeCoffee` nun als `synchronized` deklariert sind.

**b)** Gibt es jetzt noch Programmabläufe, die einer illegalen Benutzung des Kaffeeautomaten entsprechen? Begründen Sie Ihre Antwort kurz.

**LÖSUNG:** Nein. Immer nur ein Thread/Benutzer kann eine der beiden Methoden von `CoffeeMachine` ausführen. Dadurch wird sichergestellt, dass die Instanzvariable `working` den Zustand der Maschine immer korrekt wiedergibt. Da ein neuer Kaffee nur dann erfolgreich bestellt werden kann, wenn `working` auf `false` steht, kann es in dieser Modellierung nicht mehr zu einer illegalen Benutzung kommen.

*Anmerkung: Oft wurde bemerkt, dass Benutzer 2 die Tasse von Benutzer 1 klauen könnte. Dies wird durch die Klasse `CoffeeMachine` zwar nicht verhindert, aber die Benutzer sind ja hier so programmiert, dass dies auch nicht versucht wird: ein Benutzer ruft `takeCoffee` nur dann auf, wenn der zuvor getätigte Aufruf von `makeCoffee` erfolgreich beendet wurde (jeder Benutzer ist ja nur 1 Thread). Dies ist hier also unerheblich.*

**c)** Im veränderten Programm kann es nun zu einem Deadlock kommen. Erklären Sie warum!

**LÖSUNG:** Ein Thread beendet den Aufruf von `makeCoffee` und gibt das Lock ab. Der andere Thread betritt nun `makeCoffee`. Da `working` nun `true` ist, verbleibt dieser Thread in der Schleife – doch dabei hält dieser Thread weiterhin das Lock des Kaffeeautomaten und verhindert somit, dass der andere Thread `takeCoffee` ausführt um damit `working` wieder auf `false` zu setzen.

**d)** Geben Sie genau an, wie die beiden Klassen `CoffeeMachine` und `Benutzer` durch Einfügen von Aufrufen der Methoden `wait()` und `notifyAll()` zu verändern sind, so dass jeder Programmablauf einer legalen Benutzung entspricht und keine Deadlocks mehr auftreten können!

**LÖSUNG:**

```
 8    public synchronized void makeCoffee(int nummer) {
 9      while (working) {
10        wait()
11      }
12      System.out.println("Benutzer " + nummer + " bestellt Kaffee.");
13      working = true;
14    }
15
16    public synchronized void takeCoffee(int nummer) {
17      System.out.println("Benutzer " + nummer + " entnimmt Tasse.");
18      working = false;
19      notifyAll();
20    }
21  }
```

*Anmerkung: Interessanterweise kam es öfters vor, dass die letzte Teilaufgabe völlig korrekt gelöst wurde, während in den vorangegangen Teilaufgaben völliges Unverständnis der Materie demonstriert worden war. Vielleicht ist diese alte Klausuraufgabe inzwischen einfach zu bekannt geworden? Natürlich ist der Kommentar in Zeile 10 schon ein starker Hinweis, aber wohin das* `notifyAll` *gehört wurde ja nicht angedeutet.*
*Übrigens: ein* `notifyAll` *in die Zeilen 31-46 ist nutzlos, da es dort nicht in einem synchronisierten Block wäre und selbst wenn, dann würde es das falsche Lock-Objekt betreffen. (Lock-Objekt für* `wait` *ist hier ja dann das* eine *Objekt der Klasse* `CoffeeMachine` *ansonsten hätte man ja mit den* `Benutzer`*-Objekten zwei verschiedene Lock-Objekte! Seltene Lösungen mit eigens kreiertem Lock-Objekt wurden aber auch akzeptiert, wenn diese denn korrekt waren.)*

**Lösung Aufgabe 7 (Hoare Logik):** **(7 Punkte)**

**a)** Erklären Sie kurz: Was bedeutet die Gültigkeit des Hoare-Tripels $\{\text{wahr}\}c\{Q\}$, für irgendeine Anweisung $c$ und irgendeine Aussage $Q$.

**LÖSUNG:** Für *jeden* beliebigen Anfangszustand gilt: *wenn* $c$ terminiert, *dann* gilt danach $Q$.
*Buchwissen aus dem Skript; aber auch leicht herleitbar aus Grundverständnis des Hoare-Kalküls!*

**b)** Gegeben ist die neben abgedruckte komplette Definition der Klasse `Fubar`.
Beweisen Sie mit Hilfe des Hoare Kalküls, dass folgende Invariante für diese Klasse gilt:

`x > 0` oder `y < 0`

Wer möchte, darf diese Invariante zur Abkürzung hier mit $I$ bezeichnen, oder in pseudo-mathematischer Notation als

$\texttt{x} > 0 \vee \texttt{y} < 0$

hinschreiben.
*Hinweise:*

- Schreiben Sie verwendete Schleifeninvarianten explizit hin, um die Korrektur zu erleichtern!
- Generell gilt, das eine Aussage $A$ auch die Aussage ($A$ oder $B$) impliziert; für eine beliebige Aussage $B$.

```
public class Fubar {
   double x;
   double y;

   public Fubar() {
       x = -1.1;
       y = -1.2;
   }
   public Fubar(double d) {
       x = d;
       y = x;
       while (x >= 0.0) {
         x = x - 1.0;
         y = y - 1.0;
       }
   }
   public boolean fing (boolean b) {
       boolean r = x <= 0 && y >= 0;
       if (y >= 0) {  x = x + 1.0;  }
       else        {  x = x - 1.0;  }
       return r;
   }
}
```

**LÖSUNG:** *Praktisch gleiche Aufgabe wie in der Probeklausur, anstatt boolesche Variablen haben wir halt zwei Zahlen $> 0$ bzw. $< 0$, in jedem Fall ist die Struktur gleich: A oder B. Der erste Konstruktor und die Methode waren ebenfalls Vereinfachungen im Vergleich zur Probeklausur. Lediglich der zweite Konstruktor hatte eine gefürchtete while-Schleife, doch zum Glück stand die benötigte Invariante $y = x$ direkt als Zuweisung unmittelbar vor der Schleife. Offenbar ist vielen jedoch noch unbekannt, dass eine Klasse mehr als einen Konstruktor haben kann und/oder was die Konsequenz daraus ist – dies war für viele das Hauptproblem hier! Das Klassen-Invariante und Schleifen-Invariante unabhängige, verschiedene Bedingungen sein dürfen, war für einige ebenfalls ein Problem gewesen.*

Um zu zeigen, dass eine Klassen-Invariante gültig ist, müssen wir:

**a)** Ohne Vorbedingung beweisen, dass nach der Ausführung *jedes einzelnen* Konstruktors *für sich* die Invariante gilt.

**b)** Für jede Methode müssen wir zeigen, dass die Invariante nach Ausführung des Methodenrumpfes gilt, wobei wir die Invariante auch als Vorbedingung annehmen dürfen.

*Es handelt sich also hier also um drei einzelne Teilaufgaben! Wer diese Erkenntnis deutlich zeigte, erhielt dafür schon mal einen Punkt. Wir bemerken noch, dass Konstruktoren eigentlich etwas einfacher als Methoden zu behandeln sind — wegen der simpleren Vorbedingung.*

**1. Konstruktor ohne Argumente**

$$\begin{array}{rll} \{\text{wahr}\} & \texttt{x = -1.1;} & \{\texttt{x} = -1.1\} \\ \{\texttt{x} = -1.1\} & \texttt{y = -1.2;} & \{\texttt{x} = -1.1 \wedge \texttt{y} = -1.2\} \\ & & \{\texttt{y} < 0\} \\ & & \{\texttt{x} > 0 \vee \texttt{y} < 0\} \end{array}$$

Wobei die letzten Beiden Schritte mit der Konsequenz-Regel folgen (hätte man auch in einem Schritt machen dürfen; auch $\texttt{x} = -1.1$ könnte man gleich ignorieren).

**2. Konstruktor mit `double d` Argument** Wer möchte, kann die erste Zuweisung mit der Konsequenz-Regel gleich vergessen, um etwas Schreibarbeit zu sparen. Die zweite Zuweisung ist dagegen wichtig, da diese die gesuchte Schleifeninvariante darstellt:

$$\begin{array}{lll} \{\text{wahr}\} & \texttt{x = d;} & \{\text{wahr}\} \\ \{\text{wahr}\} & \texttt{y = x;} & \{\texttt{y} = \texttt{x}\} \\ \{\texttt{y} = \texttt{x}\} & \texttt{while (x >= 0.0) \{...\}} & \{\texttt{y} = \texttt{x} \wedge \texttt{x} < 0\} \\ & & \{\texttt{y} < 0\} \\ & & \{\texttt{x} > 0 \vee \texttt{y} < 0\} \end{array}$$

wobei die letzten beiden Schritte wieder durch Anwendung der Konsequenz-Regel folgt (zu Verdeutlichung hier in zwei Schritten ausgeführt).

Jetzt müssen wir aber natürlich noch für den Schleifenrumpf nachrechnen, dass die Invariante $\texttt{y} = \texttt{x}$ auch nach einem Durchlauf wieder gilt, wenn dies vorher der Fall war, wobei wir hier zur Verdeutlichung die jeweiligen Lösungsschritte unnötig explizit angeben:

$$\begin{array}{rll} & & \{\texttt{y} = \texttt{x}\} \\ \{\texttt{y} = (\texttt{x} - 1) + 1\} & \texttt{x = x-1;} & \{\texttt{y} = \texttt{x} + 1;\} \\ \{(\texttt{y} - 1) + 1 = \texttt{x} + 1\} & \texttt{y = y-1;} & \{\texttt{y} + 1 = \texttt{x} + 1\} \\ & & \{\texttt{y} = \texttt{x}\} \end{array}$$

**Methode `fing(boolean b)`** Die erste und letzte Anweisung sind für uns hier irrelevant, weshalb wir Aussagen über die Variable `r` gleich mit der Konsequenz-Regel vergessen.

$$\begin{array}{lll} \{I\} & \texttt{boolean r = this.foox || this.fooy;} & \{I\} \\ \{I\} & \texttt{if y>= 0 \{...\} else \{...\}} & \{I\} \end{array}$$

Jetzt müssen wir noch die beiden Nebenrechnungen für die Zweige des Konditionals nachreichen:

$$\begin{array}{rll} \{(\texttt{x} > 0 \vee \texttt{y} < 0) \wedge \texttt{y} >= 0\} & & \\ \{\texttt{x} > 0 \vee \texttt{y} < 0\} & \texttt{x = x + 1;} & \{\texttt{x} - 1 > 0 \vee \texttt{y} < 0\} \\ & & \{\texttt{x} > 0 \vee \texttt{y} < 0\} \end{array}$$

D.h. die If-Bedingung konnten wir im ersten Zweig komplett ignorieren; für den zweiten Fall gilt dies aber nicht mehr:

$$\begin{array}{rll} \{(\texttt{x} > 0 \vee \texttt{y} < 0) \wedge \neg(\texttt{y} >= 0)\} & & \\ \{(\texttt{x} > 0 \vee \texttt{y} < 0) \wedge \texttt{y} < 0\} & & \\ \{\texttt{y} < 0\} & \texttt{x = - 1;} & \{\texttt{y} < 0\} \\ & & \{\texttt{x} > 0 \vee \texttt{y} < 0\} \end{array}$$