



**LÖSUNG:**

**A1:** UML-Klassendiagramm und ein Speicherbild hatte wenige überrascht — die zweit-beliebteste Aufgabe der Klausur. Bei Aufgabenteil b) hatten erschreckend viele Probleme damit, einen einfachen Methodenaufruf in Java hinzuschreiben: weder `insert(Node 32);` noch `taa.insert(32).insert(64);` sind korrekt! Verbotene Aufrufe der *privaten* Methoden `place` oder der direkte Zugriff auf die *privaten* Instanzvariablen waren häufige Fehler.

Das es sich bei der Datenstruktur um einen missgebildeten Binär-Baum handelte, wie diese gelegentlich in Hausaufgaben zu finden sind, war dagegen zur Lösung der Aufgabe weitgehend irrelevant.

**A2:** Diese Aufgabe behandelte nacheinander die Themen: Methodenaufruf und Seiteneffekte, Überladen, Arrays und Exceptions. Jedes richtige Kreuz bringt 3 Punkte, jedes falsche Kreuz oder ungültige Mehrfach-Kreuze -1. Für das Erkennen der vier “schwergewichtigen” Bands innerhalb der Antwortmöglichkeiten gab es leider keine Bonuspunkte.

**A3:** In dieser beliebten Aufgabe wurde sehr häufig die volle Punktzahl erreicht. Überraschenderweise wurden die meisten Punkte in den ersten beiden Aufgabenteilen verschenkt. Die von uns als “schwerer” eingestufte dritte Frage wurde dagegen fast immer richtig gelöst. Die gelegentliche falsche Antwort “nicht-ableitbar” zur zweiten Teilaufgabe kann man immerhin noch damit erklären, dass diese Frage in einer älteren Klausur bereits einmal vorkam — die alte Aufgabe allerdings eine andere Grammatik verwendete!

**A4:** Kommentar zur Aufgabe befindet sich innerhalb der Lösung dort.

**A5:** Der schwierigste Teil der ersten Teilaufgabe war offenbar (ii), da hier die Verzahnung recht verwickelt ist: Erst Ausgabe von A, dann Ausgabe von B, dann führt A Aufruf von `einpacken` aus, Koffer-Thread macht danach seine Ausgabe, B führt Methode `einpacken` aus und erst danach läuft Thread C.

(iv) ist offensichtlich unmöglich, da im gegebenen Code eine Zustandsänderung immer erst *nach* irgendeiner Textausgabe erfolgen kann.

(v) wegen der Synchronisierung von `umpacken` ist es unmöglich, zweimal den gleichen Gegenstand durch Aufruf von `umpacken` zu erhalten, ohne dass jemand anderes diesen vorher wieder hineinsteckt.

(vi) Erst Ausgabe A, dann Ausgabe C, Aufruf `einpacken` von A, danach C, dann nur noch A. Enttäuschend war die geringe Anzahl an richtigen Lösungen zum zweiten und vierten Aufgabenteil. Vielleicht sollte man in der Vorlesung zuerst synchronisierte Blöcke besprechen, bevor man die eigentlich häufiger auftretenden synchronisierten Methoden behandelt.

Die vielen Antworten zur letzten Teilaufgabe, wo man im Code `notifyAll()` einbauen sollte (auch wenn vorher “Nein” angekreuzt war) und warum `sleep` doch besser wäre als `wait` (!?!?!), gingen völlig am Thema vorbei.

Insgesamt wurden in der Aufgabe zur Nebenläufigkeit die meisten Punkte verschenkt.

**A6:** Ankreuzen gültiger Hoare-Tripel kam gut an (2 Punkte für jede richtige Antwort und dank der unsinnigen dritten Antwortmöglichkeit nur -1 für falsche Kreuze); die verschachtelte `while`-Schleife eher weniger. Allerdings sieht dieser letzte Aufgabenteil nur schwer aus, denn wer eine `while`-Schleife behandeln kann, schafft auch zwei — es ist ja schließlich genau das Gleiche zu tun! Die gesuchten Invarianten waren praktisch vorgegeben: man musste nur die beiden 0 in der Vorbedingung durch  $i$  und  $k$  ersetzen, sobald diese im Scope waren.

Die Bedingung  $z = 42$  musste man einfach nur überall mit hinschreiben: Der Code berührt die Variable  $z$  ja nirgends, d.h. man muss den Zustand von  $z$  einfach nur durchschleifen. Für diese elementare Erkenntnis über den Hoare-Kalkül gab es 1 Punkt.

**Lösung Aufgabe 1 (UML & Speicherdiagramm):****(12 Punkte)**

a) Zeichnen Sie das UML Klassendiagramm zu folgenden 4 Klassen:

```
public interface Collection {
    int    size();
    void   add(Integer i);
    boolean contains(Integer i);
}
```

```
public class Set implements Collection {
    private int counter;

    public Set() { counter=0; }

    @Override
    public int size() { return counter; }

    @Override
    public void add(Integer i) {
        counter++; // INCOMPLETE
    }

    @Override
    public boolean contains(Integer i) {
        return false; // INCOMPLETE
    }
}
```

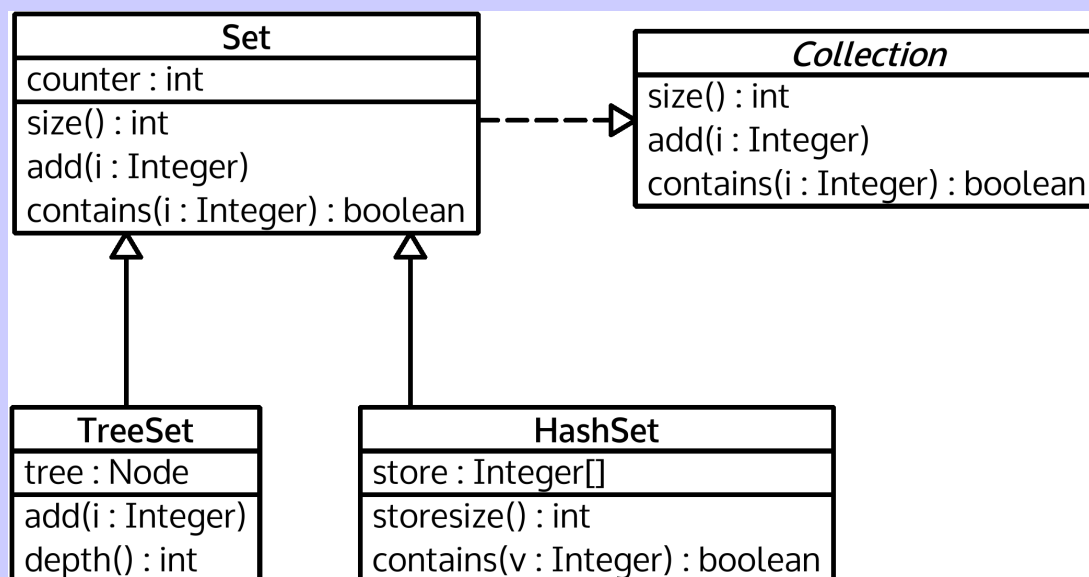
```
public class HashSet extends Set {
    private Integer[] store;

    public HashSet() { store = new Integer[42]; }
    public int storesize() { return 42; }
    @Override
    public boolean contains(Integer v) {
        for (int i=0; i<store.length; i=i+1){
            if (v.equals(store[i])) return true;
        }
        return false;
    }
}
```

```
public class TreeSet extends Set {
    private Node tree;

    public TreeSet() { super(); }
    @Override
    public void add(Integer i) {
        super.add(i);
        tree.insert(i);
    }
    public int depth() {
        return 0; //UNFINISHED
    }
}
```

*Hinweis:* Klasse **Node** aus Teilaufgabe b soll im Klassendiagramm nicht mit abgebildet werden.

**LÖSUNG:**

- b) Gegeben ist die neben angegebene Definition der Klasse **Node**, eine unvollständige **main**-Methode aus einer anderen Klasse, sowie das unten gezeigte Speicherdiagramm.

Fügen Sie Anweisung zur **main**-Methode hinzu, so dass das gegebene Speicherdiagramm zu der markierten Stelle passt!

*Genauer:* Beginnend mit einem leeren Speicher soll nach Ablauf der von ihnen erweiterten **main**-Methode der Speicherinhalt mit dem Speicherdiagramm übereinstimmen. Den gegebenen Code dürfen Sie nicht verändern; Sie dürfen nur Anweisungen in **main** hinter dem Kommentar einfügen. Verwenden Sie möglichst wenige Anweisungen. Achten Sie drauf, dass auch die angegebenen lokalen Variablen exakt<sup>a</sup> übereinstimmen.

```
public class Main {
    public static void
    main(String[] args) {
        int x = 2 * 8;
        Node taa = new Node(x);
        // Anweisungen ab hier anfügen!
```

```
        taa.insert(32);

        taa.insert(64);

        Node tbb =
            new Node(Node.UNDEFINED);

        taa.setLeft(tbb);

        Node tcc = new Node(24);

        tcc.setLeft(tbb);

        taa.getRight().setLeft(tcc);
```

```
        // !!! Speicherbild hier !!!
        System.out.println("So isses!");
    } }
```

<sup>a</sup>Parameter **args** zur Vereinfachung ignoriert.

```
public class Node {
    public final int UNDEFINED = 999;
    private int data;
    private Node left;
    private Node right;

    public Node(int data) {
        this.data = data;
        this.left = null;
        this.right = null;
    }

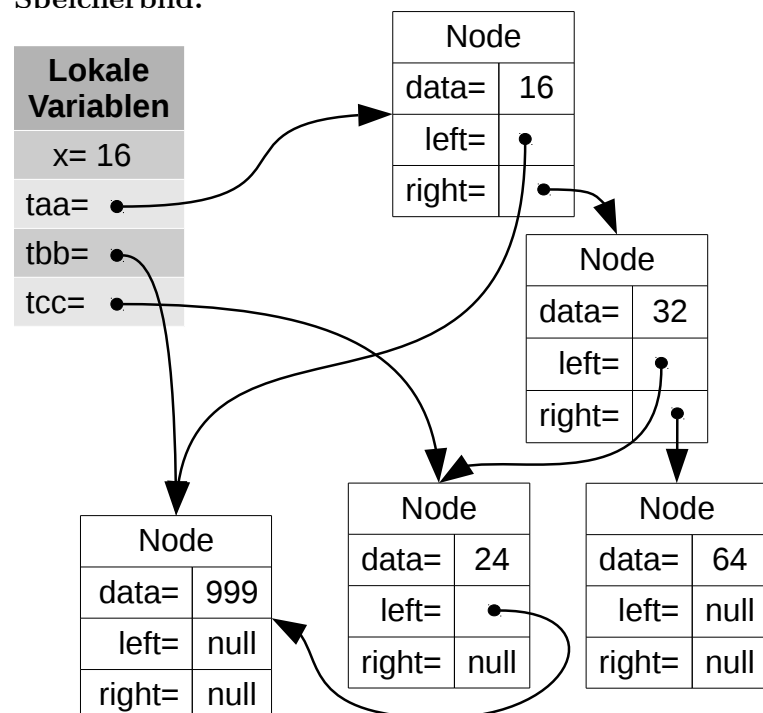
    public Node getRight() { return right; }

    public void setLeft(Node node) { left = node; }

    public void insert(int value) {
        if (value > data) right = place(value, right);
        else left = place(value, left);
    }

    private Node place(int value, Node node) {
        if (node == null) { return new Node(value); }
        else {
            node.insert(value);
            return node;
        }
    }
}
```

**Speicherbild:**



**Lösung Aufgabe 2 (Programmverständnis):****(12 Punkte)**

Welche Ausgabe wird jeweils von dem gegebenen Programm erzeugt? (Jeweils max. 1 von 4 ankreuzen.)

a) public class Main {

```
    public static String c(){
        System.out.print("b");
        return "e";
    }
```

```
    public static void main(String[] args) {
        String c = "c";
        String d = "d";
        System.out.print(("a"+c()+((d)+"c"));
    } }
```

Ausgabe ist:

☐ bacdc☐ aedc☒ baedc☐ acdc

b) public class Main {

```
    public static String foo(Integer i, Double d){return "Hallo";}
    public static String foo(Double d, Object o){return "Hello";}
    public static String foo(Object d){ return " world"; }
    public static String foo(String s){ return "ween"; }
    public static void main(String[] args) {
        String res = foo(6.0,9.0)+foo(3.0+"")+"";
        System.out.println(res);
    } }
```

Ausgabe ist:

☐ Hallo world☐ Halloween☐ Hello world☒ Helloween

c) public class Main {

```
    public static void main(String[] args) {
        int[]idx =
            new int[]{ 6 , 5 , 2 , 7 , 9 ,10 , 1 , 8 , 4 , 3 ,10 };
        char[]val =
            new char[]{'d','a','i','m','e','m','a','d','d','i','n'};
        int i = 1;
        String result = "";
        while (i < idx.length) { result += val[idx[i]];
                                i = idx[i]+1; }
        System.out.println(result);
    } }
```

Ausgabe ist:

☐ mai☐ admin☐ aaimdm☒ maiden

d) public class Fehler extends Exception { }

public class Main {

```
    public static void main(String[] args) {
        try {
            System.out.print("Mot");
            if (0==0.0) throw new Fehler();
            System.out.print("kraft");
        } catch (NumberFormatException n) {
            System.out.print("ten");
        } catch (Fehler f) {
            System.out.print("or");
        } catch (Exception e) {
            System.out.print("örhead");
        } finally {
            System.out.print("rad");
        } } }
```

Ausgabe ist:

☒ Motorrad☐ Motorkraftrad☐ Motten☐ Motörhead

**Lösung Aufgabe 3 (Backus-Naur-Form):****(12 Punkte)**

Gegeben sei folgende BNF-Grammatik mit dem Startsymbol  $\langle \text{Ausdruck} \rangle$ :

$$\begin{aligned}\langle \text{Ausdruck} \rangle &::= \langle \text{BinOp} \rangle \mid [!]\langle \text{Term} \rangle \\ \langle \text{Term} \rangle &::= \langle \text{Atom} \rangle \mid "(" \langle \text{Ausdruck} \rangle ")" \\ \langle \text{BinOp} \rangle &::= \langle \text{Ausdruck} \rangle "\&" \langle \text{Term} \rangle \mid \langle \text{Ausdruck} \rangle "||" \langle \text{Term} \rangle \\ \langle \text{Atom} \rangle &::= (\langle \text{Buchstabe} \rangle)^+ \mid \text{"true"} \mid \text{"false"} \\ \langle \text{Buchstabe} \rangle &::= \text{"x"} \mid \text{"y"} \mid \text{"z"} \\ \langle \text{Funktion} \rangle &::= \text{"f"} \mid \text{"g"}\end{aligned}$$

Entscheiden Sie für die folgenden Wörter jeweils, ob sie in der Sprache dieser Grammatik sind. Geben Sie klar an, ob das Wort in der Sprache ist oder nicht. Begründen Sie Ihre Antwort, falls das Wort nicht in der Sprache ist. Geben Sie für Wörter, welche in der Sprache sind, eine ausführliche Ableitung an! Führen Sie dabei jeden Schritt einzeln aus! Lediglich bei der Ersetzung eines Nichtterminals durch seine Definition dürfen Sie gleich eine Auswahl der Alternative treffen. Die Anführungszeichen um Terminalsymbole dürfen Sie weglassen, wenn Sie möchten.

Die Ableitung des Wortes  $\text{"true"} \& \text{"x"}$  hier als Beispiel:

$$\begin{aligned}\langle \text{Ausdruck} \rangle &\rightarrow \langle \text{BinOp} \rangle \rightarrow \langle \text{Ausdruck} \rangle "\&" \langle \text{Term} \rangle \rightarrow [!]\langle \text{Term} \rangle "\&" \langle \text{Term} \rangle \\ &\rightarrow \langle \text{Term} \rangle "\&" \langle \text{Term} \rangle \rightarrow \langle \text{Term} \rangle "\&" \langle \text{Atom} \rangle \rightarrow \langle \text{Term} \rangle "\&" (\langle \text{Buchstabe} \rangle)^+ \\ &\rightarrow \langle \text{Term} \rangle "\&" \langle \text{Buchstabe} \rangle \rightarrow \langle \text{Term} \rangle "\&" \text{"x"} \rightarrow \langle \text{Atom} \rangle "\&" \text{"x"} \rightarrow \text{"true"} \& \text{"x"}\end{aligned}$$

a)  $(!(\text{"true"}) \& \text{"x"})$

**LÖSUNG:**

$$\begin{aligned}\langle \text{Ausdruck} \rangle &\rightarrow [!]\langle \text{Term} \rangle \rightarrow \langle \text{Term} \rangle \rightarrow "(" \langle \text{Ausdruck} \rangle ")" \rightarrow "(" [!]\langle \text{Term} \rangle ")" \\ &\rightarrow "(" (!\langle \text{Term} \rangle ")" \rightarrow "(" (!\langle \text{Atom} \rangle ")" \rightarrow "(" (!\text{"true"})")\end{aligned}$$

b)  $!(\text{"y"} || \text{"z"})$

**LÖSUNG:**

$$\begin{aligned}\langle \text{Ausdruck} \rangle &\rightarrow \langle \text{BinOp} \rangle \rightarrow \langle \text{Ausdruck} \rangle "||" \langle \text{Term} \rangle \rightarrow [!]\langle \text{Term} \rangle "||" \langle \text{Term} \rangle \\ &\rightarrow (!\langle \text{Term} \rangle "||" \langle \text{Term} \rangle \rightarrow (!\langle \text{Atom} \rangle "||" \langle \text{Term} \rangle \rightarrow (!\langle \text{Atom} \rangle "||" \langle \text{Atom} \rangle \\ &\rightarrow (!(\langle \text{Buchstabe} \rangle)^+ "||" (\langle \text{Buchstabe} \rangle)^+ \rightarrow (!\langle \text{Buchstabe} \rangle "||" \langle \text{Buchstabe} \rangle \\ &\rightarrow (!\text{"y"} || \text{"z"})\end{aligned}$$

c)  $\text{"true"} \& \text{"false"} || \text{"true"}$

**LÖSUNG:**

$$\begin{aligned}\langle \text{Ausdruck} \rangle &\rightarrow \langle \text{BinOp} \rangle \rightarrow \langle \text{Ausdruck} \rangle "||" \langle \text{Term} \rangle \rightarrow \langle \text{BinOp} \rangle "||" \langle \text{Term} \rangle \\ &\rightarrow \langle \text{Ausdruck} \rangle "\&" \langle \text{Term} \rangle "||" \langle \text{Term} \rangle \rightarrow [!]\langle \text{Term} \rangle "\&" \langle \text{Term} \rangle "||" \langle \text{Term} \rangle \\ &\rightarrow \langle \text{Term} \rangle "\&" \langle \text{Term} \rangle "||" \langle \text{Term} \rangle \rightarrow \langle \text{Atom} \rangle "\&" \langle \text{Atom} \rangle "||" \langle \text{Atom} \rangle \\ &\rightarrow \text{"true"} \& \text{"false"} || \text{"true"}\end{aligned}$$

**Lösung Aufgabe 4 (Vererbung):****(12 Punkte)**

Gegeben sind folgende Klassen:

```
public class Bankkonto implements Comparable {
    private int kontonummer;

    public Bankkonto(int knr) {
        this.kontonummer = knr;
    }
    public int getKontonummer() {
        return kontonummer;
    }
    @Override
    public int compareTo(Object o) { // TODO
        return 0;
    }
    @Override
    public String toString() {
        return "Bankkonto(" + kontonummer + ")";
    }
}
```

```
public class Girokonto extends Bankkonto {
    public Girokonto(int kontonummer) {
        super(kontonummer);
    }
}

public class Sparkonto extends Bankkonto {
    private int zinssatz;

    public Sparkonto(int knr, int zinssatz){
        super(knr);
        this.zinssatz = zinssatz;
    }
}
```

a) Welche der folgenden Anweisung erzeugen Typfehler:

- |   |  |   |
|---|--|---|
| (i) <code>Object o = new Bankkonto(4);</code>       | <input checked="" type="checkbox"/> Okay | <input type="checkbox"/> Typfehler            |
| (ii) <code>Bankkonto b = new Girokonto(7);</code>   | <input checked="" type="checkbox"/> Okay | <input type="checkbox"/> Typfehler            |
| (iii) <code>Sparkonto s = new Bankkonto(5);</code>  | <input type="checkbox"/> Okay            | <input checked="" type="checkbox"/> Typfehler |
| (iv) <code>Girokonto g = new Sparkonto(2,7);</code> | <input type="checkbox"/> Okay            | <input checked="" type="checkbox"/> Typfehler |

b) Die Anweisung `System.out.println(new Girokonto(9));` gibt `"Bankkonto(9)"` aus. Ändern Sie die Klasse `Girokonto` so ab, dass `"Girokonto(9)"` ausgegeben wird. Die Klasse `Bankkonto` dürfen Sie nicht verändern!

**LÖSUNG:** Wir fügen zur Klasse `Girokonto` hinzu:

```
@Override
public String toString() {
    return "Girkonto(" + this.getKontonummer() + ")";
}
```

Wichtig: `return "Girkonto(" + this.kontonummer + ")";` geht hier gerade nicht!

Gefährlich umständliche Lösungen, bei denen `Girokonto` eine eigene Instanzvariable mit der Kontonummer spendiert wurde, haben wir auch akzeptiert, sofern dann auch der Konstruktor entsprechend korrekt mit abgeändert wurde.

Fortsetzung von Aufgabe 4:

- c) Vervollständigen Sie die Methode `compareTo` innerhalb der Klasse `Bankkonto`, so dass Bankkonten nach Ihrer Kontonummer verglichen werden.

*Beispiel:* `(new Bankkonto(7)).compareTo(new Bankkonto(3))` evaluiert zu `-1`.

#### LÖSUNG:

```
@Override
public int compareTo(Object o) {
    if (o instanceof Bankkonto){
        Integer otherknr = ((Bankkonto) o).kontonummer;
        return otherknr.compareTo(kontonummer);
    } else {
        throw new IllegalArgumentException();
    } }
}
```

Die Verwendung von `instanceof`, `compareTo` und das Werfen der Ausnahme wurden nicht gefordert. Auch folgende Lösung erzielt die vollen Punktzahl:

```
public int compareTo(Object o) {
    Bankkonto other = (Bankkonto) o;
    if (kontonummer == other.kontonummer) return 0;
    else if (kontonummer > other.kontonummer) return -1;
    else return 1;
}
```

Für die Bewertung spielte es auch keine Rolle, ob die implementierte Ordnung “größer als” oder “kleiner als” implementierte, d.h. `+1` und `-1` durfte man vertauschen. Wichtig war, das überhaupt eine Ordnung implementiert wurde: `b1.compareTo(b2) == -b2.compareTo(b1)`, `b1.compareTo(b1) == 0`, usw.

In Java ergibt `compareTo` auf `Integer`-Objekten mit Werten `7` und `3` auch `+1`, doch eine Ordnung auf Konten können wir natürlich auch umgekehrt definieren — je nachdem, was wir modellieren wollen (z.B. `compareTo` als “älter” — eine kleinere Kontonummer bedeutet meistens auch “älter”).

- d) Gegeben sind zwei korrekte Methoden zum Sortieren von Array-Listen. Die beiden Methoden unterscheiden sich nur durch Ihre Typ-Signaturen:

```
public static void barSort(ArrayList<Bankkonto> bls) { }
public static void fooSort(ArrayList<? extends Bankkonto> bls) { }
```

Erklären Sie den Unterschied zwischen `barSort` und `fooSort`!

**LÖSUNG:** Die Methode `barSort` akzeptiert nur Listen der Klasse `Bankkonto`. Dagegen akzeptiert die Methode `fooSort` auch Listen von Erben von `Bankkonto`, also zum Beispiel eine Liste über `Girokonto`.

Bei dieser Aufgabe über die Vorzüge von MergeSort gegenüber SelectionSort zu schreiben, brachte dagegen gar nichts ein — denn hier ging es ja nur um Generics!



**Lösung Aufgabe 5 (Nebenläufigkeit):****(12 Punkte)**

```

public class Packer implements Runnable {
    String name;
    String ding;
    Koffer koffer;

    public Packer(
        String n, String d, Koffer k) {
        name = n; ding = d; koffer = k;
    }
    @Override
    public synchronized void run() {
        while (true) {
            if (koffer.auspacken().equals(""))
                { System.out.println(name +
                    " packt " + ding + " hinein.");
                  koffer.einpacken(ding);
                }
            else {
                System.out.print(name +
                    " packt " + ding + " ein und ");
                ding = koffer.umpacken(ding);
                System.out.println(ding + " aus.");
            }
        }
    }
}

```

```

public class Koffer {
    private String inhalt;

    public Koffer() { inhalt = ""; }
    public String auspacken() {return inhalt;}
    public void einpacken(String sache) {
        inhalt = sache;
    }
    public synchronized
        String umpacken(String herein) {
        String heraus = inhalt;
        inhalt = herein;
        return heraus;
    }
}

public class Main {
    Koffer k = new Koffer();
    Packer pa = new Packer("A", "hut", k);
    Packer pb = new Packer("B", "ski", k);
    Packer pc = new Packer("C", "axt", k);
    (new Thread(pa)).start();
    (new Thread(pb)).start();
    (new Thread(pc)).start();
    while(true){
        System.out.println("Koffer: " + k.auspacken());
    }
}

```

a) Welche der folgenden Ausgaben sind jeweils nach einem frischem Programmstart möglich:

- (i) C packt axt hinein.  
 B packt ski ein und axt aus.  
 Koffer: ski  
 A packt hut ein und ski aus.  
 Ausgabe ist  
☒ möglich ☐ unmöglich.

- (ii) A packt hut hinein.  
 B packt ski hinein.  
 Koffer: hut  
 C packt axt ein und ski aus.  
 Ausgabe ist  
☒ möglich ☐ unmöglich.

- (iii) A packt hut hinein.  
 A packt hut ein und hut aus.  
 Koffer: hut  
 A packt hut ein und hut aus.  
 Ausgabe ist  
☒ möglich ☐ unmöglich.

- (iv) B packt ski ein und axt aus.  
 C packt axt hinein.  
 A packt hut ein und ski aus.  
 Koffer: hut  
 Ausgabe ist  
☐ möglich ☒ unmöglich.

- (v) A packt hut hinein.  
 B packt ski ein und hut aus.  
 C packt axt ein und hut aus.  
 Koffer: axt  
 Ausgabe ist  
☐ möglich ☒ unmöglich.

- (vi) A packt hut hinein.  
 C packt axt hinein.  
 A packt hut ein und axt aus.  
 Ausgabe ist  
☒ möglich ☐ unmöglich.

Fortsetzung von Aufgabe 5:

- b) Eine seltene Race Condition verursacht bei einem Test folgende hässliche Ausgabe:

```
Koffer: ski
B packt axt ein und A packt hut ein und axt aus.
ski aus.
Koffer: hut
```

Die mittleren beiden Zeilen werden von einer Methode erzeugt, welche bereits **synchronized** ist! Warum ist diese Ausgabe dennoch möglich? *Genauer:* Wie kann man dies mit korrekter Synchronisierung verhindern? (Die **print**-Anweisungen bleiben unverändert an gleicher Stelle.)

**LÖSUNG:** Die Synchronisierung findet auf *verschiedenen* Lock-Objekten statt: Das Schlüsselwort **synchronized** synchronisiert eine Methode auf **this**, also das Objekt, mit dem diese Methode aufgerufen wird. In diesem Fall also jeweils eines der **Packer**-Objekte. Um dies zu verhindern, müssten alle Packer auf ein gemeinsames Objekt synchronisieren.

- c) Kann es in dem gegebenen Programm zu einem Deadlock kommen, so dass keine Ausgabe mehr stattfindet? Begründen Sie anschließend Ihre Antwort mit 2–4 Sätzen!

☐

Ja, Deadlock ist möglich, weil...

☒

Nein, es kann kein Deadlock auftreten, weil...

**LÖSUNG:** Nein, kein Deadlock, es findet eine kontinuierliche Ausgabe statt.

Die einzige Möglichkeit, dass die **run**-Methode der **Packer**-Objekte angehalten wird, ist durch Aufruf der synchronisierten Methode **umpacken**. Diese Methode kann jedoch immer bis zum Ende durchlaufen, so dass das Lock des **Koffer**-Objekts immer wieder frei wird.

- d) Die Methode **wait()** darf nur innerhalb einer synchronisierten Methode (oder innerhalb eines synchronisierten Blocks) aufgerufen werden. Warum? Was ist der Sinn eines Aufrufs von **wait()**?

**LÖSUNG:** Der Sinn von **wait** besteht darin, ein gehaltenes Lock temporär frei zu geben. Außerhalb eines synchronisierten Methode/Programmblocks wird kein Lock gehalten, das frei gegeben werden könnte.

**Lösung Aufgabe 6 (Hoare Logik):****(12 Punkte)**

a) Entscheiden Sie jeweils ohne Angabe eines Beweis, ob es sich um ein Hoare-Tripel handelt<sup>b</sup>, und falls es ein Hoare-Tripel ist, ob dieses gültig ist oder nicht.

(i)  $\{i < j\} \ i = i + 7; \{i + 7 < j\}$

☐

kein Hoare-Tripel

☒

ungültiges Hoare-Tripel

☐

gültiges Hoare-Tripel

(ii)  $\{n = a + b \text{ und } a \leq b\} \ \{b = b + 1; n = n + 1; \} \{a < b\}$

☐

kein Hoare-Tripel

☐

ungültiges Hoare-Tripel

☒

gültiges Hoare-Tripel

(iii)  $\{z = x + y\} \ \text{if } (x > 0) \ \{z = z - x; \} \ \text{else } \{y = y - x; \} \ \{x + y \geq z\}$

☐

kein Hoare-Tripel

☐

ungültiges Hoare-Tripel

☒

gültiges Hoare-Tripel

b) Vervollständigen Sie die Regel des Hoare-Kalküls für **while**-Schleifen:

$$\frac{P \rightarrow I \qquad \{I \wedge b\} \ c \ \{I\} \qquad I \wedge \neg b \rightarrow Q}{\{P\} \ \text{while } (b) \ c \ \{Q\}}$$

c) Beweisen Sie mit Hilfe des Hoare-Kalküls die Gültigkeit des folgenden Hoare-Tripels:

$$\{z = 42 \text{ und } r = (0 \cdot y) + 0\} \ c \ \{z = 42 \text{ und } r = x \cdot y\}$$

wobei all Variablen  $i, k, r, x, y, z$  Typ **int** haben und  $c$  das folgende Programmstück ist:

```
int i = 0;
while (i != x) {
    int k = 0;
    while (k != y) { r++; k++; }
    i++;
}
```

Nachdem Sie Ihren Beweis vollendet haben, schreiben Sie hier die benutzten Invarianten hin:

**Invariante äußere Schleife  $I_1$ :**  $z = 42 \text{ und } r = (i \cdot y) + 0$

**Invariante innere Schleife  $I_2$ :**  $z = 42 \text{ und } r = (i \cdot y) + k$

**LÖSUNG:** Natürlich kann man  $I_1$  auch ganz äquivalent als " $z = 42 \wedge r = y \cdot i$ " schreiben!

*Option zu Vereinfachung:*  $z = 42$  überall weglassen, dafür 1 Punkt Abzug.

<sup>b</sup>Falls es kein Hoare-Tripel ist, dann liegt es *nicht* an einer Spitzfindigkeit wie z.B. ein Semikolon zu viel/zu wenig.

**LÖSUNG:**

Zur Abkürzung schreiben wir  $\wedge$  für “und” und  $\vee$  für “oder” — dies war aber kein Zwang. Die ganzen unsinnig erscheinenden Nullen in den Prämissen waren ein Hinweis auf die gesuchten Invarianten.

Die Gültigkeit des Hoare-Triples

$$\{z = 42 \wedge r = (0 \cdot y) + 0\} \text{ int } i = 0; \{z = 42 \wedge r = (i \cdot y) + 0\}$$

folgt nach der Hoare-Regel für Zuweisungen. Die Nachbedingung dieses Triples ist bereits identisch zu  $I_1$ , somit haben wir bereits die erste Prämisse  $Q \rightarrow I_1$  der While-Regel. Die Negation der Schleifenbedingung  $(i \neq x)$  ergibt  $\neg(i \neq x)$ , also äquivalent  $i = x$ ; es gilt also  $(I_1 \wedge i = x) \rightarrow (z = 42 \wedge r = x \cdot y)$  wie benötigt für die dritte Prämisse der While-Regel.

Wenn wir jetzt noch die zweite Prämisse der While-Regel zeigen, ist der Beweis schon komplett:  $\{I_1 \wedge i \neq x\} \langle \text{Rumpf äußere Schleife} \rangle \{I_2\}$  ist also noch zu zeigen. Mit der Konsequenz-Regel kann man  $i \neq x$  gleich wieder vergessen, wenn man möchte. Es gilt

$$\{z = 42 \wedge r = (i \cdot y) + 0\} \text{ int } k = 0; \{z = 42 \wedge r = (i \cdot y) + k\}$$

womit wir bereits  $I_2$ , die Invariante der inneren Schleife, direkt erhalten. Nach Ablauf der inneren Schleife gilt  $I_2 \wedge k = y$ , was wir zuerst zu  $(z = 42 \wedge r = (i \cdot y) + y)$  umformen und dann weiter zu  $(z = 42 \wedge r = (i + 1) \cdot y)$  (geht natürlich auch in einem Schritt). Die Gültigkeit des Hoare-Triples

$$\{z = 42 \wedge r = (i + 1) \cdot y\} i ++; \{I_1\}$$

beendet den Beweis für die äußere Schleife.

Zuletzt müssen wir noch den Rumpf der inneren Schleife betrachten: wir starten mit  $\{I_2 \wedge k \neq y\}$  und benötigen am Ende  $I_2$ ; wobei wir  $k \neq y$  gar nicht verwenden:

$$\begin{aligned} & \{I_2 \wedge k \neq y\} \\ & \{z = 42 \wedge r = (i \cdot y) + k\} \\ & \{z = 42 \wedge r + 1 = (i \cdot y) + k + 1\} \quad r ++; \quad \{z = 42 \wedge r + 1 = (i \cdot y) + (k + 1)\} \\ & \{z = 42 \wedge r + 1 = (i \cdot y) + (k + 1)\} \quad k ++; \quad \{z = 42 \wedge r + 1 = (i \cdot y) + k\} \end{aligned}$$



