Prof. Dr. J. Giesl
F. Frohn, J. Hensel, D. Korzeniewski

Aufgabe 1 (Programmanalyse): (13 + 6 = 19 Punkte)

a) Geben Sie die Ausgabe des Programms für den Aufruf `java M` an. Tragen Sie hierzu jeweils die ausgegebenen Zeichen in die markierten Stellen hinter „`OUT:`“ ein.

```
public class A {

    int x = 3;

    int y = 5;

    public A() {
        setX(x - 3);
    }

    public A(int x) {
        this.x += x;
    }

    public A(float x) {
        this.x *= x;
    }

    void setX(int z) {
        this.x = z;
    }

    public void f(float z) {
        y *= z;
    }
}
```

```
public class B extends A {

    static int x = 7;

    int y = 11;

    void setX(int z) {
        x = z;
    }

    public void f(long z) {
        y += z;
    }

}
```

```
public class M {
    public static void main(String[] args) {
        A a1 = new A();
        System.out.println(a1.x + " " + B.x);       // OUT: [    ] [    ]

        A ab = new B();
        System.out.println(ab.x + " " + B.x);       // OUT: [    ] [    ]

        A a2 = new A(-5L);
        System.out.println(a2.x);                   // OUT: [    ]

        ab.f(-7L);
        System.out.println(ab.y + " " + ((B)ab).y);// OUT: [    ] [    ]

        ((B)ab).f(-11L);
        System.out.println(ab.y + " " + ((B)ab).y);// OUT: [    ] [    ]
    }
}
```

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

```
1  import java.util.*;
2
3  public class C extends A {
4
5      public C(double x) {
6          super(x);
7      }
8
9      public LinkedList<Integer> f(LinkedList<Integer> integers) {
10         List<Integer> res = new LinkedList<>();
11         for (int i: integers) {
12             res.add(i);
13         }
14         return res;
15     }
16
17     public static void main(String[] args) {
18         C c = new C(1);
19         LinkedList<Object> list = new LinkedList<>();
20         c.f(list);
21     }
22 }
```

Hinweise:

- Die Methode `boolean add(E e)` ist im Interface `List<E>` deklariert. Sie dient dazu, ein Element `e` an das Ende einer Liste anzuhängen.

Lösung:

a)

```
public class M {
    public static void main(String[] args) {
        A a1 = new A();
        System.out.println(a1.x + " " + B.x);       // OUT: [  0 ] [  7 ]

        A ab = new B();
        System.out.println(ab.x + " " + ((B)ab).x);// OUT: [  3 ] [  0 ]

        A a2 = new A(-5L);
        System.out.println(a2.x);                   // OUT: [-15 ]

        ab.f(-7L);
        System.out.println(ab.y + " " + ((B)ab).y);// OUT: [-35 ] [ 11 ]

        ((B)ab).f(-11L);
        System.out.println(ab.y + " " + ((B)ab).y);// OUT: [-35 ] [  0 ]
    }
}
```

b)

- `super(x)`: In der Klasse `A` gibt es keinen passenden Konstruktor (da ein `double` implizit weder zu einem `int` noch zu einem `float` konvertiert werden kann.
- `return res`: `List<Integer>` ist kein Subtyp von `LinkedList<Integer>`.

- `c.f(list)`: `LinkedList<Object>` ist kein Subtyp von `LinkedList<Integer>`.

Aufgabe 2 (Hoare-Kalkül): (13 + 4 = 17 Punkte)

Gegeben sei folgendes Java-Programm P, das zu zwei nicht negativen Eingaben $\mathtt{x} = a$ und $\mathtt{y} = b$ den Wert $\max(0, a - b)$ berechnet.

$\langle\ \mathtt{x} \geq 0 \wedge \mathtt{y} \geq 0 \wedge \mathtt{x} = a \wedge \mathtt{y} = b\ \rangle$ (Vorbedingung)

```
while (y > 0) {
    if (x > 0) {
        x = x - 1;
    }
    y = y - 1;
}
```

$\langle\ \mathtt{x} = \max(0, a - b)\ \rangle$ (Nachbedingung)

Hinweise:

- Sie dürfen in beiden Teilaufgaben beliebig viele Zusicherungs-Zeilen ergänzen oder streichen. In der Musterlösung werden allerdings genau die angegebenen Zusicherungen benutzt.
- Bedenken Sie, dass die Regeln des Kalküls syntaktisch sind, weshalb Sie semantische Änderungen (beispielsweise von $x+1 = y+1$ zu $x = y$) nur unter Zuhilfenahme der Konsequenzregeln vornehmen dürfen. Klammern dürfen und müssen Sie jedoch eventuell bei der Anwendung der Zuweisungsregel setzen.

a) Vervollständigen Sie die Verifikation des Algorithmus P auf der folgenden Seite 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.

b) Beweisen Sie die Terminierung des Algorithmus P mit Hilfe des Hoare-Kalküls.

Lösung:

a)

$\langle \mathtt{x} \geq 0 \wedge \mathtt{y} \geq 0 \wedge \mathtt{x} = a \wedge \mathtt{y} = b \rangle$
$\langle \mathtt{x} = \max(0, a - b + \mathtt{y}) \wedge \mathtt{y} \geq 0 \rangle$

```
while (y > 0) {
```

$\langle \mathtt{x} = \max(0, a - b + \mathtt{y}) \wedge \mathtt{y} \geq 0 \wedge \mathtt{y} > 0 \rangle$
$\langle \mathtt{x} = \max(0, a - b + \mathtt{y}) \wedge \mathtt{y} > 0 \rangle$

```
    if (x > 0) {
```

$\langle \mathtt{x} = \max(0, a - b + \mathtt{y}) \wedge \mathtt{y} > 0 \wedge \mathtt{x} > 0 \rangle$
$\langle \mathtt{x} - 1 = \max(0, a - b + \mathtt{y} - 1) \wedge \mathtt{y} - 1 \geq 0 \rangle$

```
        x = x - 1;
```

$\langle \mathtt{x} = \max(0, a - b + \mathtt{y} - 1) \wedge \mathtt{y} - 1 \geq 0 \rangle$

```
    }
```

$\langle \mathtt{x} = \max(0, a - b + \mathtt{y} - 1) \wedge \mathtt{y} - 1 \geq 0 \rangle$

```
    y = y - 1;
```

$\langle \mathtt{x} = \max(0, a - b + \mathtt{y}) \wedge \mathtt{y} \geq 0 \rangle$

```
}
```

$\langle \mathtt{x} = \max(0, a - b + \mathtt{y}) \wedge \mathtt{y} \geq 0 \wedge \neg(\mathtt{y} > 0) \rangle$
$\langle \mathtt{x} = \max(0, a - b) \rangle$

b) Eine gültige Variante für die Terminierung ist $V = \texttt{y}$. Die Schleifenbedingung $B = \texttt{y} > 0$ impliziert $V \geq 0$. Es gilt:

$\langle \texttt{y} = m \wedge \texttt{y} > 0 \rangle$
$\langle \texttt{y} - 1 < m \rangle$
```
if (x > 0) {
```
$\langle \texttt{y} - 1 < m \wedge x > 0 \rangle$
$\langle \texttt{y} - 1 < m \rangle$
```
    x = x - 1;
```
$\langle \texttt{y} - 1 < m \rangle$
```
}
```
$\langle \texttt{y} - 1 < m \rangle$
```
y = y - 1;
```
$\langle \texttt{y} < m \rangle$

Damit ist die Terminierung der einzigen Schleife in P gezeigt.

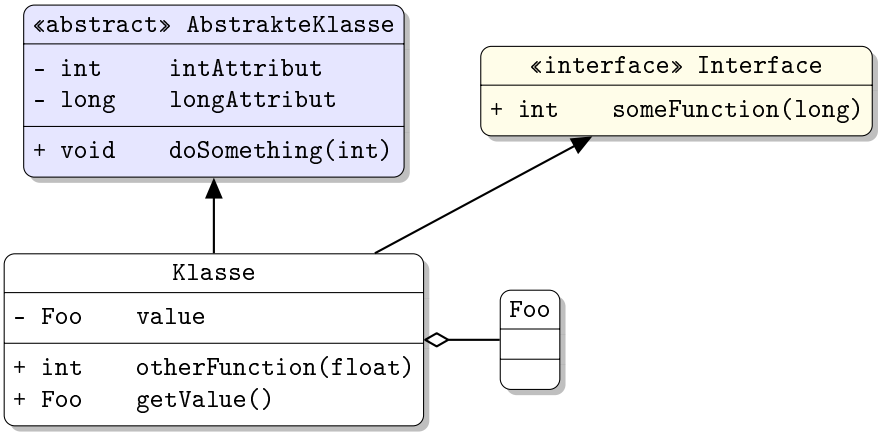
Aufgabe 3 (Klassenhierarchie): (7 + 8 = 15 Punkte)

Ziel dieser Aufgabe ist die Erstellung einer Klassenhierarchie zur Verwaltung von Höhlen, die Abenteurer erforschen und Monster heimsuchen können.

- Eine Höhle wird im Wesentlichen durch ihre Ausdehnung in Metern charakterisiert. Außerdem kann ein Monster in einer Höhle wohnen.
- Ein Höhlensystem besteht aus einer beliebigen Anzahl Höhlen, den Teilen des Höhlensystems.
- Tropfsteinhöhlen sind besondere Höhlen mit beeindruckenden Steinformationen. Bei einer Tropfsteinhöhle ist die Höhe der Stalagmiten von Interesse. Eine Tropfsteinhöhle bietet eine Methode, um sich von einem Abenteurer bewundern zu lassen.
- Es gibt verschiedene Arten von Wesen, die Höhlen durchstreifen können und sie panisch verlassen können. Diese haben nichts gemeinsam, außer der Möglichkeit Höhlen zu durchstreifen oder panisch zu verlassen.
- Abenteurer haben einen Namen und können für die Höhlenforschung gerüstet sein oder auch nicht. Außerdem können sie eine Höhle durchstreifen und panisch wieder verlassen.
- Monster können gesehen werden und geben dabei zurück, ob sie gefährlich aussehen oder nicht. Außerdem können sie Höhlen durchstreifen und panisch verlassen.

a) Entwerfen Sie unter Berücksichtigung der Prinzipien der Datenkapselung eine geeignete Klassenhierarchie für die oben aufgelisteten Sachverhalte. Notieren Sie keine Konstruktoren oder Selektoren. Sie müssen nicht markieren, ob Attribute `final` sein sollen. Achten Sie darauf, dass gemeinsame Merkmale in Oberklassen bzw. Interfaces zusammengefasst werden und markieren Sie alle Klassen als abstrakt, bei denen dies sinnvoll ist.

Verwenden Sie hierbei die folgende Notation:



Eine Klasse wird hier durch einen Kasten dargestellt, in dem der Name der Klasse sowie alle in der Klasse definierten bzw. überschriebenen Attribute und Methoden in einzelnen Abschnitten beschrieben werden. Weiterhin bedeutet der Pfeil $B \blacktriangleright A$, dass A die Oberklasse von B ist (also `class B extends A` bzw. `class B implements A`, falls A ein Interface ist). Der Pfeil $B \multimap A$ bedeutet, dass A ein Objekt vom Typ B benutzt. Benutzen Sie `-`, um `private` abzukürzen, und `+` für alle anderen Sichtbarkeiten (wie z. B. `public`). Fügen Sie Ihrem Diagramm keine Kästen für vordefinierte Klassen wie `String` hinzu.

b) Schreiben Sie eine Java-Methode mit der folgenden Signatur:

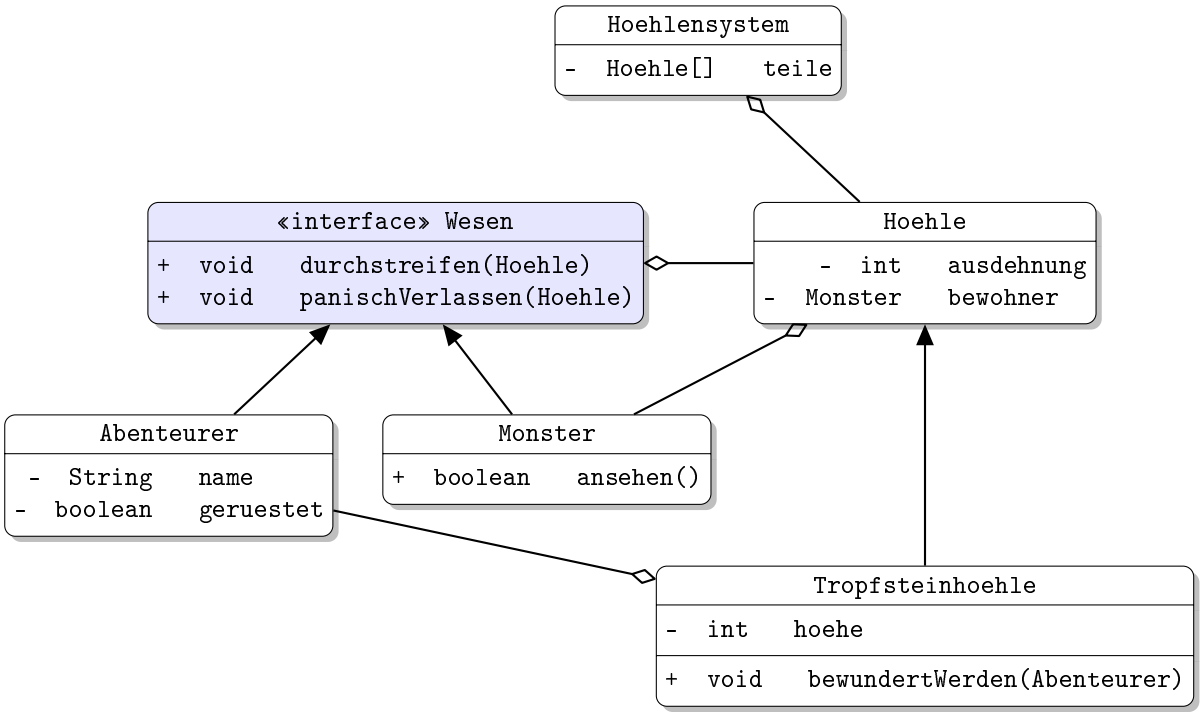
```
public static int hoehlensystemErforschen(
    Hoehlensystem hoehlen,
    Abenteurer abenteurer
)
```

In dieser Methode soll der übergebene Abenteurer das übergebene Höhlensystem auf folgende Weise erforschen: Falls die Höhle von einem Monster bewohnt ist und das Monster gefährlich aussieht, soll die aktuelle Höhle panisch verlassen werden und die weitere Erforschung des gesamten Höhlensystems abgebrochen werden. Andernfalls werden Tropfsteinhöhlen bewundert, alle anderen Höhlen durchstreift der Abenteurer, bevor er zur nächsten Höhle des Systems übergeht. Am Schluss soll die Methode zurückgeben, wieviel Meter des Höhlensystems erforscht wurden. Dazu werden die Ausdehnungen der durchstreiften oder bewunderten Höhlen addiert.

Gehen Sie davon aus, dass es zu jedem Attribut geeignete Selektoren gibt.

Lösung:

a) Die Zusammenhänge können wie folgt modelliert werden:



b)
```
public static int hoehlensystemErforschen(
    Hoehlensystem hoehlen,
    Abenteurer abenteurer
) {
  int erkundet = 0;
  for(Hoehle hoehle : hoehlen.getTeile()) {
    if(hoehle.getBewohner() != null && hoehle.getBewohner().ansehen()) {
      abenteurer.panischVerlassen(hoehle);
      break;
    }

    if (hoehle instanceof Tropfsteinhoehle) {
      ((Tropfsteinhoehle) hoehle).bewundertWerden(abenteurer);
    } else {
      abenteurer.durchstreifen(hoehle);
    }
    erkundet += hoehle.getAusdehnung();
```

```
    }
    return erkundet;
}
```

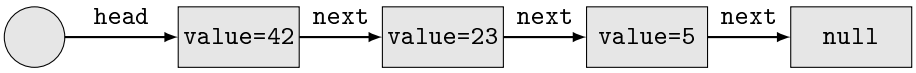
Aufgabe 4 (Listen in Java): (3 + 2 + 5 + 5 + 2 + 3 + 9 = 29 Punkte)

Die Klasse `List` dient zur Repräsentation von Listen. Sie verfügt über ein Attribut `head`, welches auf das erste Listenelement verweist. Jedes Listenelement wird als Objekt der Klasse `Element` dargestellt und enthält einen Wert vom Typ `int` (Attribut `value`) und einen Verweis auf den direkten Nachfolger (Attribut `next`). Die Klassen `List` und `Element` sind im gleichen Paket.
Die leere Liste ist eine Instanz der Klasse `List`, deren Attribut `head` den Wert `null` hat. Der Nachfolger des letzten Elements einer Liste ist immer `null`.

```
public class List {
  Element head = null;
}

public class Element {
  int value;
  Element next;
}
```

In der folgenden Grafik ist dargestellt, wie eine Liste mit den Werten `42, 23, 5` mit der Klasse `List` repräsentiert wird.



Objekte der Klasse `List` sind als Kreise und Objekte der Klasse `Element` sind als Rechtecke dargestellt.
Sie dürfen in allen Teilaufgaben beliebige Hilfsmethoden zu den Klassen `List` und `Element` hinzufügen. Wenn Sie von dieser Möglichkeit Gebrauch machen, müssen Sie eindeutig kennzeichnen, welche Hilfsmethoden zu welchen Klassen gehören.

a) Implementieren Sie die Methode `boolean contains(int n)` in der Klasse `List`. Diese Methode gibt `true` zurück, falls die Liste die Zahl `n` enthält. Sonst gibt die Methode `false` zurück.

Verwenden Sie zur Lösung dieser Teilaufgabe keine Schleifen!

b) Schreiben sie eine eigene Exception-Klasse `DuplicateFoundException` mit einem Attribut `value` vom Typ `int` und einem geeigneten Konstruktor. Diese Klasse soll später dazu genutzt werden, um zu signalisieren, dass ein Element mehrfach in der Liste enthalten ist. Die `toString` Methode der Klasse `DuplicateFoundException` soll `"Doppeltes Element: "` und anschließend das Attribut `value` zurückgeben.

c) Implementieren Sie eine Methode `LinkedList<Integer> copyElements()` in der Klasse `List`. Diese kopiert alle Werte aus der Liste in eine neue `LinkedList` aus dem Java Collections Framework, unter der Annahme, dass jede Zahl in der Liste höchstens einmal vorkommt. Falls eine Zahl doch mehrfach auftritt, soll eine entsprechende `DuplicateFoundException` geworfen werden. Deklarieren Sie die Methode so, dass ersichtlich ist, dass eine `DuplicateFoundException` geworfen werden könnte.

d) Implementieren Sie eine Methode `static LinkedList<Integer> copyAndPrint(List l)`. Diese soll Werte aus `l` in ein neues Objekt der Klasse `LinkedList<Integer>` kopieren. Falls `l` keine Duplikate enthält, sollen Sie anschließend mit einer `for-each` Schleife über die neue `LinkedList` iterieren und dabei jedes Element in einer neuen Zeile ausgeben.

Benutzen Sie hierzu die Methode `copyElements` aus dem vorherigen Aufgabenteil. Fangen Sie die unter Umständen auftretende `DuplicateFoundException` und geben Sie in diesem Fall eine Fehlermeldung aus, in der die mehrfach auftretende Zahl ersichtlich ist. Falls der Fehler auftritt soll eine leere Liste zurückgegeben werden.

e) Passen Sie die Deklarationen der Klassen `List` und `Element` sowie ihrer Attribute so an, dass beliebige Objekte gleichen Typs als Werte gespeichert werden können. Erweitern Sie die Klassen zu diesem Zweck um einen generischen Typparameter `T`, der den Typ der in der Liste gespeicherten Werte angibt.

Es ist nicht notwendig, die Methoden in den Klassen `List` und `Element` anzupassen!

f) Wir geben das folgende Interface vor:

```
public interface Filter<T> {
  boolean check(T value);
}
```

Schreiben Sie eine nicht-abstrakte Klasse `Nat`, welche das Interface `Filter<Integer>` implementiert. Die Methode `check` in der Klasse `Nat` soll für eine Zahl `x` als Eingabe `true` zurück liefern, falls `x` $\in \mathbb{N}$, sonst `false`.

g) Implementieren Sie die Methode `void apply(Filter<T> filter)` in der Klasse `List<T>`. Die Methode entfernt die Werte aus der aktuellen Liste, bei denen die `check` Methode des übergebenen `Filter`-Objekts `true` zurück gibt.

Wenn `a.apply(new Nat())` für die Integer-Liste `a` mit den Werten `42, -23, 5, 0, -1` aufgerufen wird, enthält diese also anschließend die Werte `-23, -1`.

Gehen Sie davon aus, dass das `filter`-Argument nie `null` ist.

Verwenden Sie zur Lösung dieser Teilaufgabe keine Schleifen!

Hinweise:

Es ist hilfreich, in dieser Aufgabe eine Hilfsmehtode `Element apply(Filter<T> filter)` in der Klasse `Element` zu schreiben.

Lösung:

a)

```
//in der Klasse List:
boolean contains(int n) {
    if(this.head != null) {
      return head.contains(n);
    } else {
      return false;
    }
}

//in der Klasse Element:
boolean contains(int n) {
  if(this.value == n) {
    return true;
  } else if (this.next == null) {
    return false;
  } else {
    return this.next.contains(n);
  }
```

b)

```
public class DuplicateFoundException extends Exception {
  private int value;

  public DuplicateFoundException(int value) {
    this.value = value;
  }

  public String toString() {
    return "Doppeltes Element: " + value;
  }
}
```

c)
```
LinkedList<Integer> copyElements() throws DuplicateFoundException {
  LinkedList<Integer> l = new LinkedList<Integer>();
  Element cur = this.head;
  while(cur != null) {
    if(cur.next != null && cur.next.contains(cur.value)) {
      throw new DuplicateFoundException(cur.value);
    }
    l.add(cur.value);
    cur = cur.next;
  }
  return l;
}
```

d)
```
static LinkedList<Integer> copyAndPrint(List l) {
    LinkedList<Integer> myList = new LinkedList<Integer>();
    try {
        myList = l.copyElements();
    } catch (DuplicateFoundException e) {
        System.out.println(e.toString());
    }
    for( Integer i : myList) {
        System.out.println(i.toString());
    }
    return myList;
}
```

e)
```
public class List<T> {
    Element<T> head = null;
}

public class Element<T> {
    T value;
    Element<T> next;
}
```

f)
```
public class Nat implements Filter<Integer> {
    public boolean check(Integer value) {
        return value >= 0;
    }
}
```

g)
```
// in Klasse List
void apply(Filter<T> filter) {
    if (head != null) {
        head = head.apply(filter);
    }
}

// in Klasse Element
Element apply(Filter<T> filter) {
    if(filter.check(value)) {
        if(this.next != null) {
          return this.next.apply(filter);
        } else {
          return null;
```

```
            }
        } else {
            if(this.next != null) {
              this.next = this.next.apply(filter);
            }
            return this;
        }

    }
```