Aufgabe 1 (Programmanalyse): (18 Punkte)

Gegeben sei das folgende Java-Programm. Dieses Programm gibt sechs Zeilen Text in der Form
`v[0]: Unterklasse(Oberklasse(name=Attribut(name=1)), name=2)`
auf der Konsole aus. Tragen Sie die Ausgabe an den markierten Stellen in den Kommentaren ein.

```
class Verwendet {
    int a;
    Verwendet(int i) { a = i; }
    public String toString() { return "Verwendet(a=" + a + ")"; }
}

class Ober {
    Verwendet verw;
    Ober(Ober o) { verw = o.verw; }
    Ober(int i) { verw = new Verwendet(i); }
    public String toString() {
        return "Ober(verw=" + verw + ")";
    }
    void f(int x) { verw.a += x; }
    Ober kopie() { return new Ober(this); }
}

class Unter extends Ober {
    int b = 4;
    Unter() { super(100); }
    Unter(Unter o) { super(o); b = o.b; }
    Unter(int i) { this(); b = i + 5; }
    public String toString() {
        return "Unter(" + super.toString()
            + ", b=" + b + ")";
    }
    void f(int x) { super.f(x); b -= x; }
    void f(short x) { b += x; }
    Unter kopie() { return new Unter(this); }
}

class Programm {
    static public void main(String[] p) {
        Ober[] v = new Ober[3];
        v[0] = new Unter(15);
        v[1] = v[0].kopie();
        System.out.println("v[0]: " // Unter(Ober(verw=Verwendet(a=___)), b=___)
                           + v[0]);
        System.out.println("v[1]: " // Unter(Ober(verw=Verwendet(a=___)), b=___)
                           + v[1]);
        v[0].f(1);
        System.out.println("v[0]: " // Unter(Ober(verw=Verwendet(a=___)), b=___)
                           + v[0]);
        System.out.println("v[1]: " // Unter(Ober(verw=Verwendet(a=___)), b=___)
                           + v[1]);
        v[1].f((short)2);
        System.out.println("v[0]: " // Unter(Ober(verw=Verwendet(a=___)), b=___)
                           + v[0]);
        System.out.println("v[1]: " // Unter(Ober(verw=Verwendet(a=___)), b=___)
                           + v[1]);
    }
}
```

Lösung:

Das Programm gibt folgendes aus:

```
v[0]: Unter(Ober(verw=Verwendet(a=100)), b=20)
v[1]: Unter(Ober(verw=Verwendet(a=100)), b=20)
v[0]: Unter(Ober(verw=Verwendet(a=101)), b=19)
v[1]: Unter(Ober(verw=Verwendet(a=101)), b=20)
v[0]: Unter(Ober(verw=Verwendet(a=103)), b=19)
v[1]: Unter(Ober(verw=Verwendet(a=103)), b=18)
```

Aufgabe 2 (Hoare-Kalkül): (4 + 6 + 2 = 12 Punkte)

Gegeben sei folgender *Java*-Algorithmus P zur Berechnung der Multiplikation zweier Zahlen k und n.

$\langle\varphi\rangle$ (Vorbedingung)

```
res = k;
i = 0;
while (i + 1 < n) {
    res = res + k;
    i = i + 1;
}
```

$\langle\psi\rangle$ (Nachbedingung)

a) Berechnen Sie zunächst anhand einiger einfacher Beispiele, in welcher Beziehung `res`, `k`, `i` und `n` bei Überprüfung der Schleifenbedingung zueinander stehen, und geben Sie anschließend eine geeignete Schleifeninvariante für die gegebene `while`-Schleife an. Verwenden Sie dafür die vorgefertigten Tabellen. Sie benötigen hierbei nicht immer alle Zeilen.

res	k	i	n
	2		4

res	k	i	n
	3		2

Invariante:

b) Als Vorbedingung für den oben aufgeführten Algorithmus P gelte $n > 0$ und als Nachbedingung

$$res = n \cdot k.$$

Vervollständigen Sie auf der nächsten Seite die 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.

c) Beweisen Sie die Terminierung des Algorithmus P. Geben Sie hierzu eine Variante für die `while`-Schleife an. Zeigen Sie, dass es sich tatsächlich um eine Variante handelt, und beweisen Sie damit unter Verwendung des Hoare-Kalküls mit der Voraussetzung $n > 0$ die Terminierung.

Lösung:

a)

res	k	i	n
2	2	0	4
4	2	1	4
6	2	2	4
8	2	3	4

res	k	i	n
3	3	0	2
6	3	1	2

Invariante: $res = (i+1) \cdot k$

b)

$\langle n > 0 \rangle$

$\langle n > 0 \wedge k = k \rangle$

```
res = k;
```

$\langle n > 0 \wedge res = k \rangle$

$\langle n > 0 \wedge res = k \wedge 0 = 0 \rangle$

```
i = 0;
```

$\langle n > 0 \wedge res = k \wedge i = 0 \rangle$

$\langle i + 1 \leq n \wedge res = (i+1) \cdot k \rangle$

```
while (i + 1 < n) {
```

$\langle i + 1 < n \wedge i + 1 \leq n \wedge res = (i+1) \cdot k \rangle$

$\langle i + 1 < n \wedge res + k = (i+2) \cdot k \rangle$

```
    res = res + k;
```

$\langle i + 1 < n \wedge res = (i+2) \cdot k \rangle$

$\langle (i+1) + 1 \leq n \wedge res = ((i+1)+1) \cdot k \rangle$

```
    i = i + 1;
```

$\langle i + 1 \leq n \wedge res = (i+1) \cdot k \rangle$

```
}
```

$\langle i + 1 \not< n \wedge i + 1 \leq n \wedge res = (i+1) \cdot k \rangle$

$\langle res = n \cdot k \rangle$

c) Eine Variante ist $V = n - i$, denn aus der Schleifenbedingung $i + 1 < n$ folgt $V = n - i \geq 0$. Somit:

$\langle n - i = m \wedge i < n \rangle$

$\langle n - i = m \rangle$

```
res = res + k;
```

$\langle n - i = m \rangle$

$\langle n - (i + 1) < m \rangle$

```
i = i + 1;
```

$\langle n - i < m \rangle$

Damit ist die Terminierung der einzigen Schleife in P gezeigt.

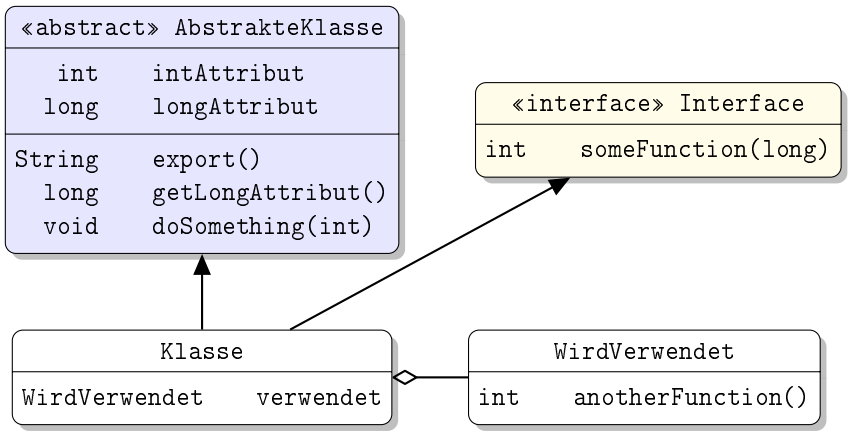
Aufgabe 3 (Sitzmöglichkeiten): (10 + 6 = 16 Punkte)

In dieser Aufgabe betrachten wir Sitzmöbel, die in einem üblichen Haushalt vorkommen können.

a) In dieser Teilaufgabe geht es um den Entwurf einer entsprechenden Klassenhierarchie, mit der die verschiedenen Eigenschaften von Sitzmöbeln sinnvoll in einem Programm gehandhabt werden können.

- Ein Kissen kann kuschelig sein. Außerdem soll zum Aufschütteln eine Methode `void schuetteln()` zur Verfügung gestellt werden.
- Für Sitzmöbel ist bekannt, wie viele Personen gleichzeitig darauf sitzen können. Es können außerdem eine Reihe von Kissen darauf liegen.
- Manche Sitzmöbel sind auch zum Liegen geeignet. Für diese ist dann auch bekannt, wie viele Leute gleichzeitig darauf liegen können.
- Ein Sofa ist ein Sitzmöbel, auf dem man auch liegen kann. Ein Sofa ist mit einem Stoff bezogen, dessen Namen gespeichert werden soll.
- Eine Holzbank ist ebenfalls ein zum Liegen geeignetes Sitzmöbel, das aus einer bestimmten Holzart hergestellt ist. Der Name dieses Holzes soll gespeichert werden.
- Ein Stuhl ist ein Sitzmöbel, das bequem sein kann – oder auch nicht.
- Ein Ledersessel ist ebenfalls ein Sitzmöbel, für das auch die Farbe des Leders bekannt ist. Der Name der Farbe soll gespeichert werden.
- Holzbänke, Stühle und Ledersessel sind abwischbar. Sie sollen eine Funktion `wischen()` zur Verfügung stellen, deren Rückgabewert angibt, wie viel Flüssigkeit (in Millilitern) beim Wischen verbraucht wurde. Beachten Sie, dass nicht alle abwischbaren Dinge Sitzmöbel sind!

Entwerfen Sie eine geeignete Klassenhierarchie für die Sitzmöbel. Notieren Sie **keine Konstruktoren, Getter und Setter**. Sie müssen **nicht markieren**, ob Attribute `final` sein sollen oder welche Zugriffsrechte (also z.B. `public` oder `private`) für sie gelten sollen. Achten Sie darauf, dass gemeinsame Merkmale in Oberklassen zusammengefasst werden. Notieren Sie Ihren Entwurf graphisch und verwenden Sie dazu die folgende Notation:



Eine Klasse wird hier durch einen Kasten beschrieben, in dem der Name der Klasse sowie Felder und Methoden in einzelnen Abschnitten beschrieben werden. Weiterhin bedeutet der Pfeil $B \rightarrow A$, dass A die Oberklasse von B ist (also **class** B **extends** A bzw. **class** B **implements** A, falls A ein Interface ist) und $A \diamond\!\!- B$, dass A den Typ B verwendet (z.B. als Typ eines Feldes oder in einem Array). Benutzen Sie + und - um `public` und `private` abzukürzen.

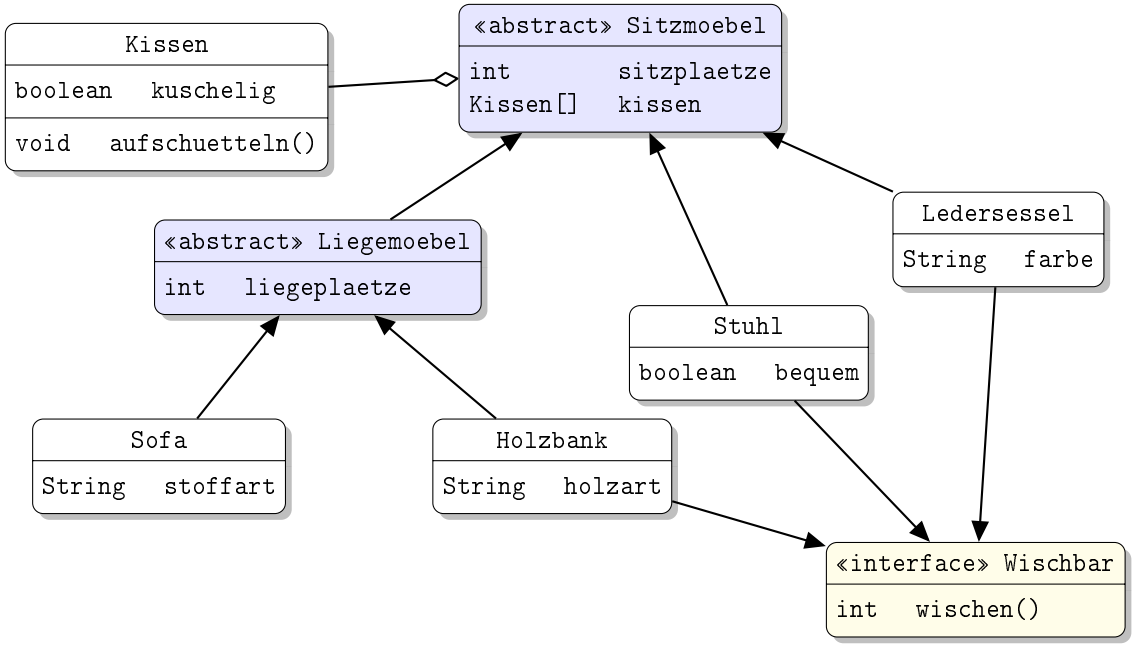
Tragen Sie keine vordefinierten Klassen (String, etc.) in Ihr Diagramm ein. Geben Sie für jede Klasse ausschließlich den jeweiligen Namen und die Namen und Datentypen ihrer Attribute an. Methoden müssen nur in dem allgemeinsten Typen (d.h. Klasse oder Interface) angegeben werden, in dem sie deklariert werden. Implementierungen dieser Methoden müssen dann nicht mehr explizit angegeben werden.

b) Eine Reihe von Sitzmöbeln soll nun geputzt werden. Wir wollen wissen, wie viel Wischwasser dabei verbraucht wird. Implementieren Sie in Java eine Funktion `alleWischen`, die ein Array vom Typ `Sitzmoebel[]` übergeben bekommt und dann alle wischbaren übergebenen Sitzmöbel wischt. Der Rückgabewert soll die Menge des verbrauchten Wischwassers sein.

Gehen Sie davon aus, dass die Methode `wischen()` an den wischbaren Sitzmöbeln bereits implementiert wurde; Sie können sie einfach verwenden. Sie können voraussetzen, dass das übergebene Array immer existiert und nie die `null`-Referenz ist. Kennzeichnen Sie die Methode mit dem Schlüsselwort `static`, falls dies angebracht ist.

Lösung:

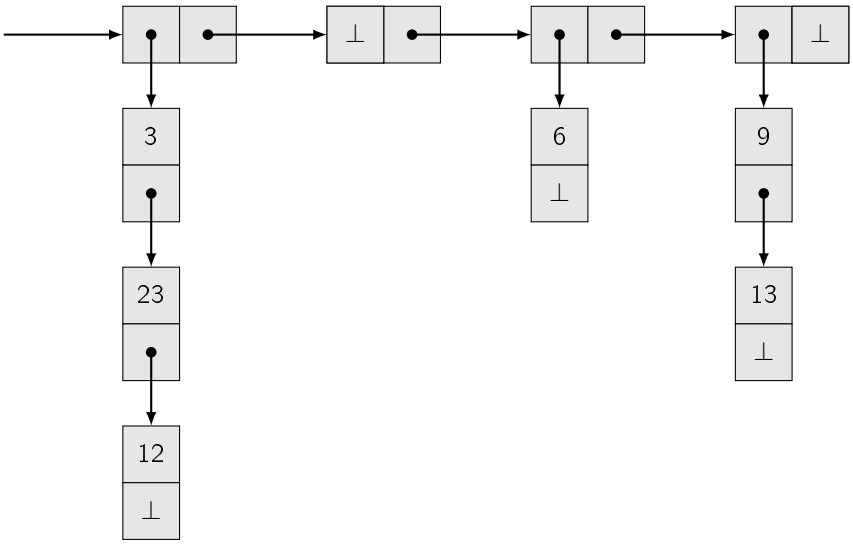
a) Eine mögliche Modellierung der geschilderten Zusammenhänge ist diese:



b)
```
public static int alleWischen(Sitzmoebel[] moebel) {
  int res = 0;
  for (int i = 0; i < moebel.length; i++) {
    if (moebel[i] instanceof Wischbar) {
      res += ((Wischbar) moebel[i]).wischen();
    }
  }
  return res;
}
```

Aufgabe 4 (Listen von Listen von Zahlen): (5 + 8 + 7 = 20 Punkte)

Wir betrachten eine Liste von Listen, die jeweils Zahlen vom Typen `int` enthält. Betrachten Sie dazu das folgende Bild:



Dargestellt ist eine Liste mit vier Elementen, welche jeweils eine Liste von Integern enthalten. Hierbei steht ⊥ für die `null`-Referenz, die verwendet wird, um das Ende einer Liste zu markieren. Die erste Teilliste enthält die drei Zahlen 3, 23 und 12. Die zweite Teilliste ist leer, die dritte Liste enthält eine 6 und die vierte Liste eine 9 und dann eine 13.
Zur Implementierung werden die beiden Klassen `ListOfList` und `ListOfInt` verwendet, die jeweils eine verkettete Liste darstellen. In jeder Instanz wird jeweils ein Wert und eine Referenz zum nächsten Listenelement gespeichert. Die Klasse `ListOfList` stellt dafür zwei Selektoren zur Verfügung:

- `public ListOfInt getItem()`
  Gibt den Wert (Liste von Integern) des aktuellen Elements zurück.
- `public ListOfList getNext()`
  Gibt das nachfolgende Element zurück.

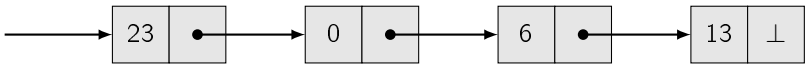
Die Klasse `ListOfInt` ist wie `ListOfList` definiert, verwaltet aber Zahlen vom Typ `int`. Sie stellt neben den Selektoren auch einen Konstruktor zur Verfügung:

- `public int getItem()`
  Gibt den Wert (als `int`) des aktuellen Elements zurück.
- `public ListOfInt getNext()`
  Gibt das nachfolgende Element zurück.
- `public ListOfInt(int item, ListOfInt next)`
  Erstellt eine `ListOfInt`, die den Wert `item` enthält und den Nachfolger `next` hat.

a) Implementieren Sie eine Methode `int getMax()` für die Klasse `ListOfInt`, die aus der Liste das Maximum bestimmt. Verwenden Sie für die Implementierung **ausschließlich Rekursion**. Sie dürfen aber Hilfsfunktionen schreiben.

b) Implementieren Sie eine Methode `ListOfInt getMaxList()` für die Klasse `ListOfList`, die eine Liste der Maxima der Unterlisten zurückgibt. Das Maximum einer leeren Liste ist 0. Verwenden Sie für die Implementierung **ausschließlich Rekursion**. Sie dürfen aber Hilfsfunktionen schreiben.

Im oberen Beispiel würde die folgende Liste zurückgegeben werden:



Hinweise:

- Sie dürfen die Funktion aus Aufgabe a) verwenden, auch wenn Sie sie nicht implementiert haben.

c) Implementieren Sie die Funktion `int sumOfMax()` für die Klasse `ListOfList`, die die Summe der Maxima der Unterlisten zurückgibt. Verwenden Sie **ausschließlich Schleifen**, also keine Rekursion um diese Funktion zu implementieren.

Im Beispiel würde 42 zurückgegeben werden.

Hinweise:

- Sie dürfen natürlich auch rekursive Funktionen aufrufen. Schreiben Sie aber keine zusätzlichen rekursiven Hilfsfunktionen.
- Sie dürfen die Funktionen aus Aufgabenteilen a) und b) verwenden, auch wenn Sie sie nicht implementiert haben.

Lösung:

a)
```
public int getMax(){
  return getMax(this.item);
}

private int getMax(int oldMax){
  int max = (this.getItem() > oldMax) ? this.getItem() : oldMax;

  if (this.getNext() == null) {
    return max;
  } else {
    return this.getNext().getMax(max);
  }
}
```

b)
```
public ListOfInt getMaxList() {
  int max = 0;
  if (this.getItem() != null) {
    max = this.getItem().getMax();
  }

  if (this.getNext() != null) {
    return new ListOfInt(max, this.getNext().getMaxList());
  } else{
    return new ListOfInt(max, null);
  }
}
```

c) Zwei Versionen, die (b) verwenden:

```
public int sumOfMax() {
  ListOfInt maxList = this.getMaxList();
  int sum = maxList.getItem();
  while (maxList.getNext() != null) {
    maxList = maxList.getNext();
```

```
    sum += maxList.getItem();
  }
  return sum;
}

public int sumOfMax2() {
  ListOfInt maxList = this.getMaxList();
  int sum = 0;
  while (maxList != null) {
    sum += maxList.getItem();
    maxList = maxList.getNext();
  }
  return sum;
}
```

Eine Version, die direkt (a) verwendet:

```
public int sumOfMax3() {
  ListOfList cur = this;
  int sum = 0;
  while (cur != null) {
    int max = 0;
    if (cur.getItem() != null) {
      max = cur.getItem().getMax();
    }
    sum += max;
    cur = cur.getNext();
  }
  return sum;
}
```