Prof. Dr. J. Giesl
M. Brockschmidt, F. Emmes, C. Otto, T. Ströder

Aufgabe 1 (Programmanalyse): (9 + 1 = 10 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 {
  public int x;
  public static int y = 0;

  public A() {
    this(8);
    y++;
  }

  public A(int i) {
    x = i;
    y++;
  }

  public int f(double d) {
    return 11;
  }

  public int f(int i) {
    return 2;
  }
}
```

```
public class B extends A {
  public int y = 5;

  public B() {
    super(9);
  }

  public B(int x) {
    x = 13;
  }

  public int f(double d) {
    return 7;
  }

  public int f(float f) {
    return 10;
  }
}
```

```
public class M {
  public static void main(String[] args) {
    A a = new A(6);
    System.out.println(a.x);              // OUT: [     ]

    System.out.println(a.y);              // OUT: [     ]

    B b = new B(12);
    System.out.println(b.x);              // OUT: [     ]

    System.out.println(b.y);              // OUT: [     ]

    System.out.println(A.y);              // OUT: [     ]

    System.out.println(b.f(14));          // OUT: [     ]

    A ab = b;
    System.out.println(ab.f(1.5f));       // OUT: [     ]

    System.out.println(new B().x);        // OUT: [     ]

    System.out.println(ab.y);             // OUT: [     ]
  }
}
```

b) Geben Sie an, welche Ausgabe das folgende Programm für den Aufruf `java N` erzeugt.

```
import java.util.*;
public class N {
  public static void main(String[] args) {
    LinkedList<Integer> list = new LinkedList<>();
    list.addFirst(8);
    list.addFirst(7);
    list.addFirst(4);
    list.addFirst(3);
    Iterator<Integer> it = list.iterator();
    int modsum = 0;
    while (it.hasNext()) {
      modsum += it.next();
      if (modsum % 7 == 0) {
        it.remove();
      }
    }
    for (int i : list) {
      System.out.println(i);
    }
  }
}
```

Lösung:

a)
```
public class M {
  public static void main(String[] args) {
    A a = new A(6);
    System.out.println(a.x);              // OUT: [  6  ]

    System.out.println(a.y);              // OUT: [  1  ]

    B b = new B(12);
    System.out.println(b.x);              // OUT: [  8  ]

    System.out.println(b.y);              // OUT: [  5  ]

    System.out.println(A.y);              // OUT: [  3  ]

    System.out.println(b.f(14));          // OUT: [  2  ]

    A ab = b;
    System.out.println(ab.f(1.5f));       // OUT: [  7  ]

    System.out.println(new B().x);        // OUT: [  9  ]

    System.out.println(ab.y);             // OUT: [  4  ]
  }
}
```

b)
```
3
8
```

Aufgabe 2 (Hoare-Kalkül): (7 + 2 = 9 Punkte)

Gegeben sei folgendes *Java*-Programm P, das zu einer Eingabe $n \geq 0$ den Wert n^2 berechnet.

$\langle\, \texttt{n} \geq 0 \,\rangle$ (Vorbedingung)

```
i = 0;
res = 0;
while (i < n) {
    res = res + 2 * i + 1;
    i = i + 1;
}
```

$\langle\, \texttt{res} = \texttt{n}^2 \,\rangle$ (Nachbedingung)

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.

Hinweise:

- Sie dürfen 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.

b) Untersuchen Sie den Algorithmus P auf seine Terminierung. Für einen Beweis der Terminierung muss eine Variante angegeben werden und unter Verwendung des Hoare-Kalküls die Terminierung unter der Voraussetzung $\texttt{n} \geq 0$ bewiesen werden. Begründen Sie, warum es sich bei der von Ihnen angegebenen Variante tatsächlich um eine gültige Variante handelt.

Lösung:

a)

$\langle \texttt{n} \geq 0 \rangle$
$\langle \texttt{n} \geq 0 \wedge 0 = 0 \wedge 0 = 0 \rangle$

```
i = 0;
```

$\langle \texttt{n} \geq 0 \wedge \texttt{i} = 0 \wedge 0 = 0 \rangle$

```
res = 0;
```

$\langle \texttt{n} \geq 0 \wedge \texttt{i} = 0 \wedge \texttt{res} = 0 \rangle$
$\langle \texttt{res} = \texttt{i}^2 \wedge \texttt{i} \leq \texttt{n} \rangle$

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

$\langle \texttt{res} = \texttt{i}^2 \wedge \texttt{i} \leq \texttt{n} \wedge \texttt{i} < \texttt{n} \rangle$
$\langle \texttt{res} + 2 \cdot \texttt{i} + 1 = (\texttt{i}+1)^2 \wedge \texttt{i} + 1 \leq \texttt{n} \rangle$

```
    res = res + 2 * i + 1;
```

$\langle \texttt{res} = (\texttt{i}+1)^2 \wedge \texttt{i} + 1 \leq \texttt{n} \rangle$

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

$\langle \texttt{res} = \texttt{i}^2 \wedge \texttt{i} \leq \texttt{n} \rangle$

```
}
```

$\langle \texttt{res} = \texttt{i}^2 \wedge \texttt{i} \leq \texttt{n} \wedge \texttt{i} \not< \texttt{n} \rangle$
$\langle \texttt{res} = \texttt{n}^2 \rangle$

b) Eine gültige Variante für die Terminierung ist $V = \texttt{n} - \texttt{i}$, denn die Schleifenbedingung $B = \texttt{i} < \texttt{n}$ impliziert $\texttt{n} - \texttt{i} \geq 0$ und es gilt:

$\langle \texttt{n} - \texttt{i} = m \wedge \texttt{i} < \texttt{n} \rangle$
$\langle \texttt{n} - (\texttt{i} + 1) < m \rangle$
`res = res + 2 * i + 1;`
$\langle \texttt{n} - (\texttt{i} + 1) < m \rangle$
`i = i + 1;`
$\langle \texttt{n} - \texttt{i} < m \rangle$

Damit ist die Terminierung der einzigen Schleife in P gezeigt.

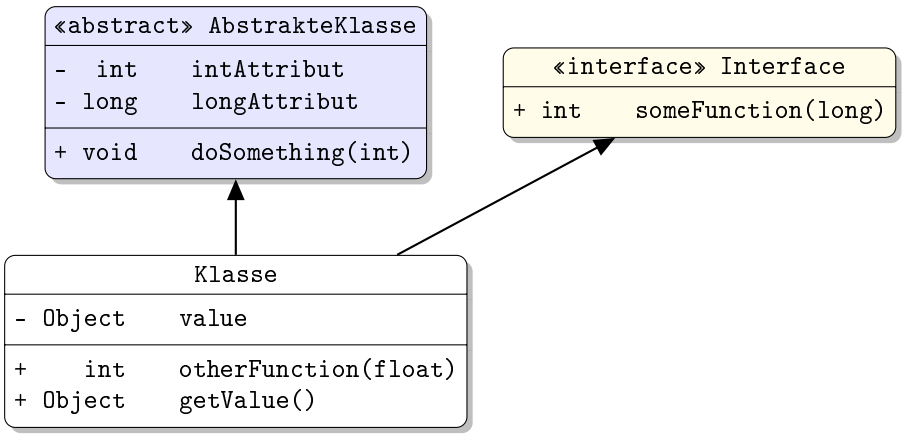
Aufgabe 3 (Klassen-Hierarchie): (5 + 3 = 8 Punkte)

Ziel dieser Aufgabe ist die Erstellung einer Hierarchie zur Verwaltung von Kunstwerken.

- Ein Kunstobjekt wurde von einen Künstler erschaffen, der durch einen `String` repräsentiert wird.
- Eine Malerei ist ein Kunstobjekt und zeichnet sich durch ihren Untergrund aus, welcher als `String` gespeichert wird.
- Ein Portrait ist eine Malerei, deren Alter (in Jahren) von Interesse ist.
- Eine Zeichnung ist ein Kunstobjekt, bei dem die Breite ihrer Linien wichtig ist, welche in Millimetern angegegeben wird.
- Eine Bildhauerei ist ein Kunstobjekt, das sich durch das verwendete Material auszeichnet, welches als `String` dargestellt wird.
- Eine Skulptur ist eine Bildhauerei, die mit einem Werkzeug herausgeschlagen wird. Hier ist der Name des Werkzeugs von Interesse.
- Eine Plastik ist eine Bildhauerei, die gegossen wird. Hier ist das Volumen (in Litern) der Plastik von Bedeutung.
- Portraits und Zeichnungen lassen sich an der Wand befestigen. Dazu stellen sie die Methode `void aufhaengen()` zur Verfügung.
- Bildhauereien lassen sich an ihren Platz rücken und bieten dazu die Methode `void verruecken()` an.
- Jedes Kunstobjekt ist entweder eine Malerei, eine Zeichnung oder eine Bildhauerei.
- Eine Bildhauerei muss nicht zwangsläufig eine Skulptur oder Plastik sein.

a) Entwerfen Sie unter Berücksichtigung der Prinzipien der Datenkapselung eine geeignete Klassenhierarchie für die oben aufgelisteten Arten von Kunstwerken. 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:

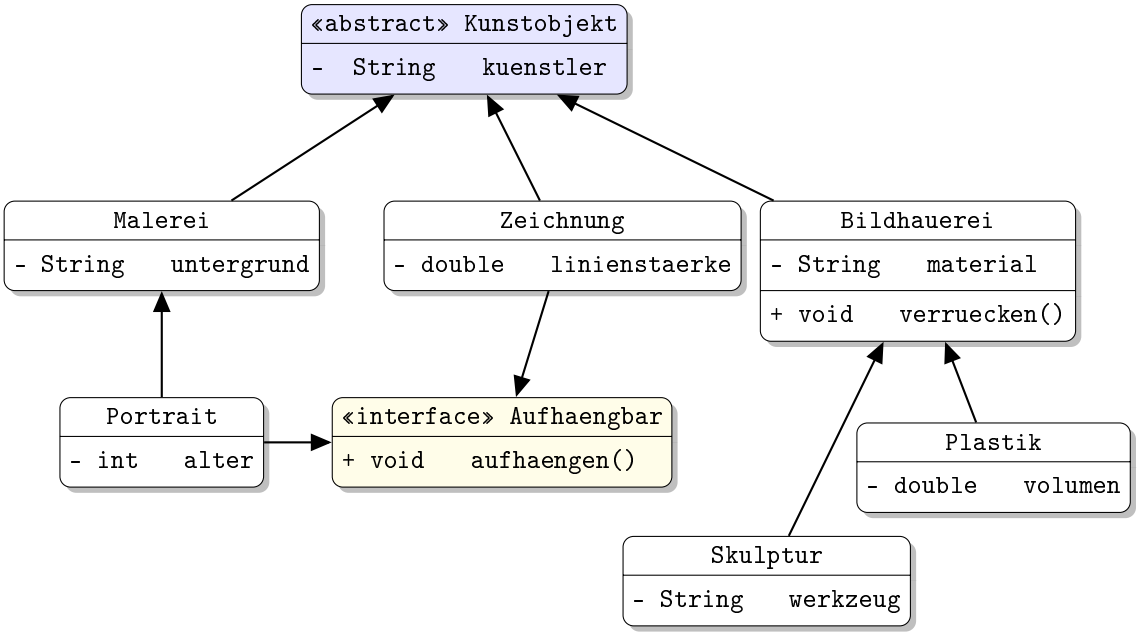


Eine Klasse wird hier durch einen Kasten beschrieben, in dem der Name der Klasse sowie Attribute und Methoden in einzelnen Abschnitten beschrieben werden. Weiterhin bedeutet der Pfeil $B \blacktriangleright A$, dass A die Oberklasse von B ist (also `class B extends A` bzw. `class B implements A`, falls A ein Interface ist). Benutzen Sie `-`, um `private` abzukürzen, und `+` für alle anderen Sichtbarkeiten (wie z. B. `public`).

b) Schreiben Sie eine Java-Methode mit der folgenden Signatur:
`public static void galerieEinrichten(Kunstobjekt[] kunstobjekte)`
Diese Methode soll alle Kunstobjekte, die sich an einer Wand befestigen lassen, aufhängen und alle Bildhauereien an ihren Platz rücken. Nehmen Sie dazu an, dass das übergebene Array `kunstobjekte` nicht `null` ist.

Lösung:

a) Die Zusammenhänge können wie folgt modelliert werden:



b)
```
public static void galerieEinrichten(Kunstobjekt[] kunstobjekte) {
    for (Kunstobjekt k : kunstobjekte) {
        if (k instanceof Aufhaengbar) {
            ((Aufhaengbar)k).aufhaengen();
        } else if (k instanceof Bildhauerei) {
            ((Bildhauerei)k).verruecken();
        }
    }
}
```

Aufgabe 4 (Listen in Java): (3 + 3 + 1 + 4 + 1 + 3 = 15 Punkte)

Die Klasse `List` dient zur Repräsentation von Listen. Jedes Listenelement wird als Objekt der Klasse `List` dargestellt. Es enthält einen Wert vom Typ `int` und einen Verweis auf den direkten Nachfolger. Der Wert wird in dem Attribut `value` gespeichert und das Attribut `next` zeigt auf das nächste Element der Liste.
Das letzte Element der Liste hat keinen Nachfolger, so dass dessen Attribut `next` auf `null` zeigt. Die leere Liste stellen wir als `null` dar.

```
public class List {
  int value;
  List next;

  public List(int v, List n) {
    this.value = v;
    this.next = n;
  }
}
```

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 Rechtecke dargestellt, die den im `value`-Attribut gespeicherten Wert enthalten. Die Elemente dieser Liste sind jeweils durch das Attibut `next` mit dem direkten Nachfolger verbunden. Attribute, deren Wert `null` ist, werden in der Grafik als Punkt ohne ausgehenden Pfeil angezeigt.

a) Implementieren Sie die statische Methode `createList(int len)` in der Klasse `List`. Die Methode gibt eine Liste der Länge `len` zurück, wobei die Elemente dieser Liste `len, len-1, ..., 1` sind. Sie dürfen davon ausgehen, dass `len` nicht negativ ist.

Verwenden Sie dazu **ausschließlich Schleifen** und keine Rekursion.

Für den Aufruf `createList(3)` soll also eine Liste mit den Elementen `3, 2, 1` zurückgegeben werden.

b) Implementieren Sie die nicht-statische Methode `append(List other)` in der Klasse `List`. Die Methode gibt eine Liste zurück, die daraus entsteht, dass man die Liste `other` an das Ende der aktuellen Liste anhängt. Hierbei dürfen die ursprünglichen Listen nicht verändert werden.

Wenn `a` die Liste aus der obigen Grafik ist und `b` eine Liste mit den Werten `1, 2, 3` ist, soll für den Aufruf `a.append(b)` also eine Liste mit den Werten `42, 23, 5, 1, 2, 3` zurückgegeben werden. Die Variable `a` zeigt danach immer noch auf die unveränderte Liste mit den Werten `42, 23, 5`.

Verwenden Sie dazu **ausschließlich Rekursion** und keine Schleifen.

c) Wir wollen die Deklaration der Klasse `List` jetzt so anpassen, dass statt nur `int`-Werten beliebige Typen als Werte benutzt werden können. Hierfür erweitern wir die Klasse um einen generischen Typparameter `T`, der den Typ der in der Liste gespeicherten Werte angibt.

Passen Sie die in der Aufgabenstellung gegebene Klassendeklaration inklusive des Konstruktors entsprechend an und geben Sie die neuen Deklarationen an.

Sie dürfen für die weiteren Teilaufgaben davon ausgehen, dass die Methoden `createList` und `append` für die generische Klasse implementiert sind! Die Methode `createList` hat also ab sofort eine Rückgabe vom Typ `List<Integer>` und bei der Methode `append` hat sowohl der formale Parameter als auch das Ergebnis den Typ `List<T>`.

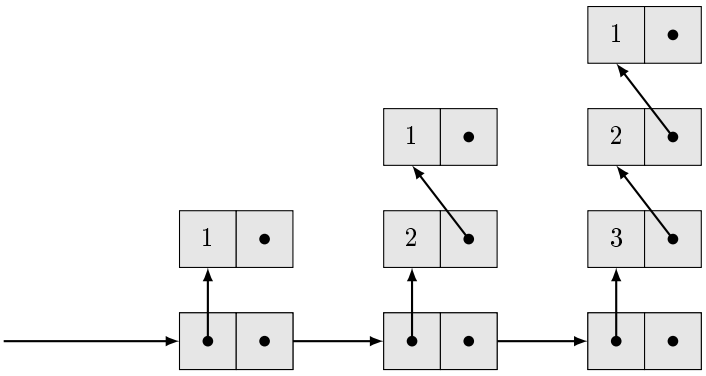
d) Implementieren Sie die statische Methode `stairs(int len)` in der Klasse `List<T>`. Die Methode gibt eine Liste der Länge `len` zurück, wobei die Elemente dieser Liste wiederum Listen sind. Hierbei sind die Werte dieser inneren Listen Zahlen (d. h. Objekte vom Typ `Integer`). Die Liste im ersten Element soll dabei die Länge 1 haben und den Wert `1` enthalten, die Liste im zweiten Element soll die Länge 2 haben

und die Werte `2, 1` enthalten, ... Das letzte Element der Liste enthält schließlich eine Liste der Länge `len` mit den Werten `len, len-1, ..., 1`.

Sie dürfen davon ausgehen, dass `len` nicht negativ ist.

Verwenden Sie dazu **ausschließlich Rekursion** und keine Schleifen.

Für den Aufruf `stairs(3)` soll also eine Liste der folgenden Form zurückgegeben werden:



Hinweise:

- Verwenden Sie die Methode `createList`.
- Sie dürfen hierbei Hilfsmethoden deklarieren und/oder die Methode `append` benutzen.

e) Wir geben das folgende Interface vor:

```
public interface Updater<T> {
  T update(T value);
}
```

Schreiben Sie eine nicht-abstrakte Klasse `Inc`, welche das Interface `Updater<Integer>` implementiert. Die Methode `update` in der Klasse `Inc` soll für eine Zahl x als Eingabe die Zahl $x + 1$ zurückgeben. Gehen Sie davon aus, dass die Eingabe nie `null` ist.

f) Implementieren Sie die nicht-statische Methode `apply(Updater<T> updater)` in der Klasse `List<T>`. Die Methode hat keine Rückgabe und ersetzt die Werte der aktuellen Liste mit Hilfe des übergebenen `Updater`-Objekts. Hierbei wird jedes `value`-Attribut mit dem Wert überschrieben, den die `update`-Methode des Arguments zurückgibt, wenn man den ursprünglichen Wert des Attributs als Eingabe übergibt.

Wenn `x` eine Instanz der Klasse `Inc` ist und `a.apply(x)` für die Integer-Liste `a` mit den Werten `42, 23, 5` aufgerufen wird, enthält diese also anschließend die Werte `43, 24, 6`.

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

Verwenden Sie dazu **ausschließlich Rekursion** und keine Schleifen.

Lösung:

a)

```
public static List createList(int len) {
  List res = null;
  for (int i = 0; i < len; i++) {
    res = new List(i+1, res);
  }
  return res;
}
```

```
  public static List createList(int len) {
    if (len == 0) {
      return null;
    }
    List start = new List(len, null);
    List cur = start;
    for (int i = len - 1; i > 0; i--) {
      cur.next = new List(i, null);
      cur = cur.next;
    }
    return start;
  }
```

b)

```
  public List append(List other) {
    if (this.next == null) {
      return new List(this.value, other);
    } else {
      return new List(this.value, this.next.append(other));
    }
  }
```

c)

```
public class List<T> {
  T value;
  List<T> next;

  public List(T v, List<T> n) {
    this.value = v;
    this.next = n;
  }
}
```

d)

```
  public static List<List<Integer>> stairs(int len) {
    return help(1, len);
  }

  private static List<List<Integer>> stairsHelper(int current, int len) {
    if (current > len) {
      return null;
    }
    return new List<>(createList(current), stairsHelper(current + 1, len));
  }
```

Alternative Lösung:

```
  public static List<List<Integer>> stairs(int len) {
    if (len == 0) {
      return null;
    } else if (len == 1) {
      return new List<>(createList(1), null);
    } else {
      return stairs(len - 1).append(new List<>(createList(len), null);
    }
  }
```

e)

```
class Inc implements Updater<Integer> {
  public Integer update(Integer value) {
```

```
      return value + 1;
    }
  }
```

f)
```
  public void apply(Updater<T> updater) {
    this.value = updater.update(this.value);
    if (this.next != null) {
      this.next.apply(updater);
    }
  }
```