Prof. Dr. J. Giesl
D. Cloerkes, S. Dollase, N. Lommen, D. Meier, F. Meyer

Tutoraufgabe 1 (Überblickswissen):

a) Was versteht man unter *Casting* und wie ist die Syntax dafür in Java?

b) Welche großen Vorteile bietet das Konzept der Vererbung in Java?

c) Viele Vorteile der Vererbung zwischen zwei Klassen `A` und `B1` lassen sich auch erzielen, ohne dass diese Klassen tatsächlich voneinander erben. So ist es beispielsweise möglich, die Klasse `B1` in eine Klasse `B2` zu überführen, welche sich genau wie `B1` verhält, ohne jedoch von `A` zu erben.

Wir passen die Klasse `B1` an, indem wir die `extends`-Klausel weglassen und stattdessen ein neues Attribut `a` vom Typ `A` in die Klasse `B2` einfügen und diesem Attribut ein neues Objekt der Klasse `A` zuweisen. Wenn in `B1` Attribute oder Methoden der Oberklasse `A` verwendet werden, müssen diese sich in `B2` nun auf das Attribut `a` beziehen.

Wir haben also die *Vererbungsbeziehung* zwischen `A` und `B1` durch eine *Nutzungsbeziehung* zwischen `A` und `B2` ersetzt. Was sind die Vor- und Nachteile dieser beiden Varianten?

```
class A {
    int i;
    void m() {}
}
```

```
class B1 extends A {
    void foo() {
        m();
        i += 1;
    }
}
```

```
class B2 {
    A a = new A();
    void foo() {
        a.m();
        a.i += 1;
    }
}
```

Lösung:

a) Die eine Art von *Casting* ist die explizite Typumwandlung zwischen primitiven Typen. Bei solch einer Typumwandlung können Informationen verlorengehen, z.B. wenn ein `double`-Wert in einen `int`-Wert umgewandelt wird. So gilt nach der Zuweisung `int i = (int)2.7;` die Gleichheit `i == 2`, da in einer `int`-Variable nur ganze Zahlen gespeichert werden können.

Solche expliziten Typumwandlungen sollte man nur mit Bedacht durchführen, wenn man sich sicher ist, dass man die verlorene Information nicht benötigt. Nicht umsonst wird der Compiliervorgang mit dem Fehler `error: incompatible types: possible lossy conversion from double to int` abgebrochen, wenn man es statt expliziter mit impliziter Typumwandlung versucht (`int i = 2.7;`).

Die zweite Art von *Casting* ist die Zuweisung eines Objekts `superobj` einer Oberklasse `Superclass` zu einem Objekt `subobj` einer Unterklasse `Subclass`. Da nicht sicher ist, ob `superobj` tatsächlich vom Typ `Subclass` ist oder nur vom Typ `Superclass`, darf eine solche Zuweisung nicht ohne Weiteres durchgeführt werden. Es gibt aber Situationen, in denen wir sicher wissen, dass `superobj` vom (spezielleren) Typ `Subclass` ist. Nach einer erfolgreichen Abfrage `superobj instanceof Subclass` wäre das bspw. der Fall. Wenn wir nun auf `superobj` eine Methode der Klasse `Subclass` aufrufen wollen, die es in `Superclass` nicht gibt, ist dies zunächst verboten. Um dem Compiler mitzuteilen, dass wir `superobj` als Objekt der Klasse `Subclass` betrachten wollen, benutzen wir *Casting*. In Java weisen wir mit der Anweisung `Subclass subobj = (Subclass)superobj;` der Variablen `subobj` nun das Objekt `superobj` zu, ändern dabei aber dessen Typ von `Superclass` zu `Subclass`. Auf `subobj` können wir nun alle Methoden der Klasse `Subclass` aufrufen.

In neueren Java-Versionen lässt sich der `instanceof`-Check mit der Casting-Anweisung zu einer Anweisung zusammenfassen, indem `instanceof` mit Pattern Matching verwendet wird:

```
if (superobj instanceof Subclass) {
    Subclass subobj = (Subclass)superobj;
    // use subobj
}
```

```
if (superobj instanceof Subclass subobj) {
    // use subobj
}
```

Allgemein sollte mit Casting vorsichtig und sparsam umgegangen werden. Wenn ein Cast der zweiten Art fehlschlägt, führt dies zu Laufzeitfehlern, die es zu vermeiden gilt. Häufig lassen sich auch Wege finden, die entsprechende Stelle des Codes anders zu implementieren. Wenn man sich zum Casting gezwungen sieht, sollte man sich immer die folgenden Fragen stellen: Warum weiß der Compiler an dieser Stelle nicht, dass das Objekt eigentlich von einem anderen, spezielleren Typ ist? Sollte diese Information eigentlich bekannt sein, und wenn ja, wie kann ich das Programm umstrukturieren, um das Casting zu vermeiden?

b) Vererbung ist ein mächtiges und charakteristisches Werkzeug der objektorientierten Programmierung. Durch die modulare Erweiterbarkeit von Klassen wird eine hohe Wiederverwendbarkeit derselben gewährleistet. Man kann grundlegende Funktionen von spezielleren Funktionen strukturell trennen und so später einfach auf erstere zurückgreifen, auch wenn letztere im neuen Kontext keine Anwendung mehr finden (sollen). Außerdem kann man in Klassenhierarchien gemeinsame Eigenschaften verschiedener Klassen an einem Ort, nämlich einer Oberklasse, zusammenführen. Will man später die Implementierung ändern, braucht man das nun nicht mehr in allen Klassen zu tun, sondern nur in der gemeinsamen Oberklasse. So vermeidet man sog. *Code-Duplications*, die häufige Fehlerquellen sind. Ferner kann Vererbung auf einer konzeptionellen Ebene dafür sorgen, dass man die Zusammenhänge der Klassen im Programm schnell versteht. Wären alle Funktionalitäten innerhalb einer monolithischen Klasse gekapselt, gäbe es keinen solchen (syntaktischen) Überblick über die (eigentlich semantischen) Beziehungen der einzelnen Funktionalitäten.

c) Zunächst einmal haben beide Varianten viele vergleichbare Eigenschaften. Beide verhalten sich gleich. Beide bieten die Möglichkeit, den Code zu modularisieren. Beide bieten die Möglichkeit, die semantische Struktur des Codes auch syntaktisch auszudrücken. Beide bieten die Möglichkeit, Code-Duplizierung zu vermeiden.

Es gibt jedoch auch einige Unterschiede. Beispielsweise wäre die Nutzungsbeziehung nicht möglich, falls Attribute oder Methoden in `A` als `protected` gekennzeichnet wären, welche `B2` nutzen müsste. Die Kopplung ist also bei der Vererbungsbeziehung größer als bei der Nutzungbeziehung. Da man meist ohnehin eine enge Kopplung vermeiden will, ist die Nutzungbeziehung hier im Vorteil, da sie eine zu enge Kopplung verhindert.

Ein weiterer Unterschied ist, dass es immer nur eine Oberklasse geben kann, jedoch beliebig viele Attribute. Es ist also nicht möglich, dass eine Klasse Vererbungsbeziehungen zu mehreren anderen Klassen hat. Mehrere Nutzungbeziehungen sind jedoch ohne Weiteres möglich. Auch dies ist also ein Vorteil der Nutzungbeziehung.

Bezüglich der Übersichtlichkeit hat die Nutzungbeziehung gegenüber der Vererbungsbeziehung auch oft einen Vorteil, denn leider werden komplexe Vererbungshierarchien schnell schwer nachvollziehbar. Das gilt insbesondere dann, wenn in der Vererbungshierarchie viel mit dem Verdecken von Attributen oder dem Überschreiben von Methode gearbeitet wird (beide Konzepte werden bald in der Vorlesung vorgestellt).

Es gibt jedoch auch Fälle, in denen die Nutzungsbeziehung nicht ausreichend ist, sondern nur die Vererbungsbeziehung weiterhilft. Dies ist immer dann der Fall, wenn wir *Subtyping* benötigen, also wenn wir ein Objekt der Unterklasse in einer Variablen vom Typ der Oberklasse speichern wollen. In unserem Beispiel wäre etwa die Zuweisung `A a = new B1();` gültig, wohingegen die Zuweisung `A a = new B2();` einen Compilerfehler generieren würde.

Aufgrund der obigen Abwägung wird heutzutage meistens empfohlen, unnötige Vererbungsbeziehungen zu vermeiden und durch Nutzungsbeziehungen zu ersetzen (*composition over inheritance*). Vererbungsbeziehungen sollten nur dann genutzt werden, wenn die Unterklasse auch wirklich alle Methoden anbieten soll, welche die Oberklasse anbietet (*Liskov substitution principle*).

Tutoraufgabe 2 (Rekursive Datenstrukturen):

In dieser Aufgabe geht es um einfach verkettete Listen als Beispiel für eine dynamische Datenstruktur. Wir legen hier besonderen Wert darauf, dass eine einmal erzeugte Liste nicht mehr verändert werden kann. Achten Sie also in der Implementierung darauf, dass die Attribute der einzelnen Listen-Elemente `nur` im Konstruktor geschrieben werden.

Für diese Aufgabe benötigen Sie die Klasse `ListExercise.java`, welche Sie aus dem Moodle-Lernraum herunterladen können.
In der gesamten Aufgabe dürfen Sie **keine Schleifen** verwenden (die Verwendung von Rekursion ist hingegen erlaubt). Ergänzen Sie in Ihrer Lösung für alle öffentlichen Methoden außer Konstruktoren und Selektoren geeignete javadoc-Kommentare.

a) Erstellen Sie eine Klasse `List`, die eine einfach verkettete unveränderliche Liste als rekursive Datenstruktur realisiert. Die Klasse `List` muss dabei mindestens die folgenden öffentlichen Methoden und Attribute enthalten:

- `static final List EMPTY` ist die einzige `List`-Instanz, die die *leere* Liste repräsentiert
- `List(List n, int v)` erzeugt eine neue Liste, die mit dem Wert `v` beginnt, gefolgt von allen Elementen der Liste `n`
- `List getNext()` liefert die von `this` referenzierte Liste ohne ihr erstes Element zurück
- `int getValue()` liefert das erste Element der Liste zurück

b) Implementieren Sie in der Klasse `List` die öffentlichen Methoden `int length()` und `String toString()`. Die Methode `length` soll die Länge der Liste zurück liefern. Die Methode `toString` soll eine textuelle Repräsentation der Liste zurück liefern, wobei die Elemente der Liste durch Kommata separiert hintereinander stehen. Beispielsweise ist die textuelle Repräsentation der Liste mit den Elementen 2, 3 und 1 der `String` `"2, 3, 1"`.

c) Implementieren Sie in der Klasse `ListExercise` die öffentliche Methode `int skipSum`, welche eine Liste als Argument erhält. Die Methode `skipSum` soll dabei, beginnend mit dem ersten Element als Startelement, jedes zweite Element der Liste aufsummieren. Beispielsweise sollte für die Liste mit den Elementen 2, 3, 1 und 5 die Summe $2 + 1 = 3$ berechnet werden.

d) Ergänzen Sie die Klasse `List` darüber hinaus noch um eine öffentliche Methode `getSublist`, welche ein Argument i vom Typ `int` erhält und eine unveränderliche Liste zurückliefert, welche die ersten i Elemente der aktuellen Liste enthält. Sollte die aktuelle Liste nicht genügend Elemente besitzen, wird einfach eine Liste mit allen Elementen der aktuellen Liste zurückgegeben.

e) **Video**
Vervollständigen Sie die Methode `merge` in der Klasse `ListExercise.java`. Diese Methode erhält zwei Listen als Eingabe, von denen wir annehmen, dass diese bereits aufsteigend sortiert sind. Sie soll eine Liste zurückliefern, die alle Elemente der beiden übergebenen Listen in aufsteigender Reihenfolge enthält.

Hinweise:

- Verwenden Sie zwei Zeiger, die jeweils auf das kleinste noch nicht in die Ergebnisliste eingefügte Element in den Argumentlisten zeigen. Vergleichen Sie die beiden Elemente und fügen Sie das kleinere ein, wobei Sie den entsprechenden Zeiger ein Element weiter rücken. Sobald eine der Argumentlisten vollständig eingefügt ist, können die Elemente der anderen Liste ohne weitere Vergleiche hintereinander eingefügt werden.

f) **Video**
Vervollständigen Sie die Methode `mergesort` in der Klasse `ListExercise.java`. Diese Methode erhält eine unveränderliche Liste als Eingabe und soll eine Liste mit den gleichen Elementen in aufsteigender Reihenfolge zurückliefern. Falls die übergebene Liste weniger als zwei Elemente enthält, soll sie unverändert zurück geliefert werden. Ansonsten soll die übergebene Liste mit der vorgegebenen Methode `divide` in zwei kleinere Listen aufgespalten werden, welche dann mit `mergesort` sortiert und mit `merge` danach wieder zusammengefügt werden.

Hinweise:

- Sie können die ausführbare `main`-Methode verwenden, um das Verhalten Ihrer Implementierung zu überprüfen. Um beispielsweise die unveränderliche Liste mit den Elementen 2, 4 und 3 sortieren zu lassen, rufen Sie die `main`-Methode durch `java ListExercise 2 4 3` auf.

Lösung:

Listing 1: List.java

```
public class List {

  public static final List EMPTY = new List(null, 0);

  private final List next;
  private final int value;

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

  public List getNext() {
    return this.next;
  }

  public int getValue() {
    return this.value;
  }

  /**
   * @return true iff this list is empty
   */
  public boolean isEmpty() {
      return this == EMPTY;
  }

  /**
   * @return a String representation of this list
   */
  public String toString() {
    if (this.isEmpty()) {
      return "";
    } else if (this.next.isEmpty()) {
      return String.valueOf(this.value);
    } else {
      return this.value + ", " + this.next.toString();
    }
  }

  /**
   * @return the length of the list
  */
  public int length() {
    if (this.isEmpty()) {
      return 0;
    } else {
      return 1 + this.next.length();
    }
  }

  /**
   * Computes a list containing the first <code>length</code> elements
   * of the current list. If this list does not contain enough
   * elements, the whole list is returned instead.
   * @param length the length of the sublist to compute
   * @return the computed sublist
   */
  public List getSublist(int length) {
    if (length <= 0 || this.isEmpty()) {
      return EMPTY;
    } else {
      List newNext = this.getNext().getSublist(length - 1);
      return new List(newNext, this.value);
    }
  }

}
```

Listing 2: ListExercise.java

```
public class ListExercise {


     /**
     * Computes the sum of every second element, starting with the first
     * @param list the list of elements to be added
     * @return the sum
     */
    public static int skipSum(List list) {
        if (list.isEmpty()) {
            return 0;
```

```
    } else {
        return list.getValue() + skipSumHelper(list.getNext());
    }
}

/**
 * Helper function for skipSum, does nothing for current element and calls skipSum again
 * @param list the list of elements to be added
 * @return the sum
 */
private static int skipSumHelper(List list) {
    if (list.isEmpty() || list.getNext().isEmpty()) {
        return 0;
    } else {
        return skipSum(list.getNext());
    }
}
```

bis auf die Elemente teilen
=> danach sortiert wieder zusammensetzen

```
/**
 * Sorts the given list.
 * @param list the list that will be sorted
 * @return the sorted list
 */
public static List mergesort(List list) {
    if (list.isEmpty() || list.getNext().isEmpty()) {
        return list;
    } else {
        List[] twoLists = divide(list);
        List newListA = mergesort(twoLists[0]);
        List newListB = mergesort(twoLists[1]);
        return merge(newListA, newListB);
    }
}

/**
 * Merges two sorted lists to one sorted list.
 */
private static List merge(List first, List second) {
    if (first.isEmpty()) {
        return second;
    }
    if (second.isEmpty()) {
        return first;
    }
    if (first.getValue() > second.getValue()) {
        return new List(merge(first, second.getNext()), second.getValue());
    } else {
        return new List(merge(first.getNext(), second), first.getValue());
    }
}

/**
 * Divides a list of at least two elements into two lists of the same
 * length (up to rounding).
 */
private static List[] divide(List list) {
    List[] res = new List[2];
    int length = list.length() / 2;
    res[0] = list.getSublist(length);
    for (int i = 0; i < length; i++) {
        list = list.getNext();
    }
    res[1] = list;
    return res;
}

/**
 * Creates a list from the given inputs and outputs the sorted list and
 * the original list afterwards.
 * @param args array of all list elements
 */
public static void main(String[] args) {
    if (args != null && args.length > 0) {
        List list = buildList(0,args);
        System.out.println(mergesort(list));
        System.out.println(list);
    }
}

/**
 * Builds a list from the given input array.
 */
```

```
    private static List buildList(int i, String[] args) {
        if (i < args.length) {
            return new List(buildList(i + 1,args), Integer.parseInt(args[i]));
        } else {
            return List.EMPTY;
        }
    }

}
```

Aufgabe 3 (Rekursive Datenstrukturen): (8 + 4 + 3 + 14 + 7 = 36 Punkte)

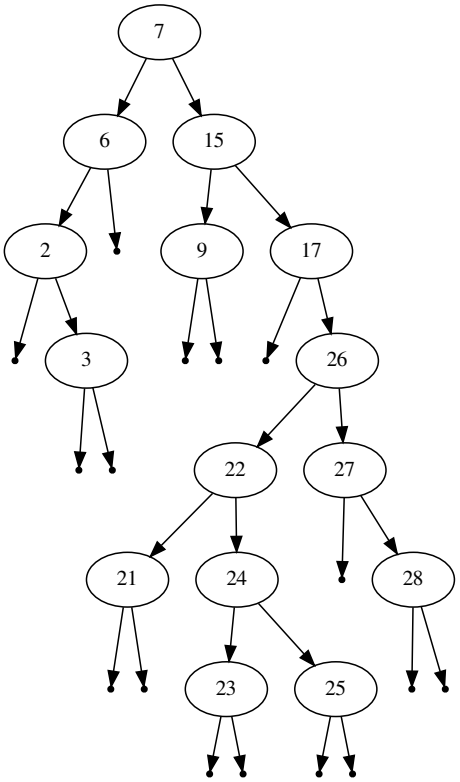
In dieser Aufgabe sollen einige rekursive Algorithmen auf sortierten Binärbäumen implementiert werden.
Aus dem Moodle-Lernraum können Sie die Klassen `BinTree` und `BinTreeNode` herunterladen. Die Klasse `BinTree` repräsentiert einen Binärbaum, entsprechend der Klasse Baum aus den Vorlesungsfolien. Einzelne Knoten des Baums werden mit der Klasse `BinTreeNode` dargestellt. Alle Methoden, die Sie implementieren, sorgen dafür, dass in dem Teilbaum `left` nur Knoten mit kleineren Werten als in der Wurzel liegen und in dem Teilbaum `right` nur Knoten mit größeren Werten.
Um den Baum zu visualisieren, ist eine Ausgabe als dot Datei bereits implementiert. In dieser einfachen Beschreibungssprache für Graphen steht eine Zeile `x -> y;` dafür, dass der Knoten `y` ein Nachfolger des Knotens `x` ist. In Dateien, die von dem vorgegebenen Code generiert wurden, steht der linke Nachfolger eines Knotens immer vor dem rechten Nachfolger in der Datei. Optional können Sie mit Hilfe der Software Graphviz, wie unten beschrieben, automatisch Bilder aus dot Dateien generieren.
Die Klasse `BinTree` enthält außerdem eine `main` Methode, die einige Teile der Implementierung testet.

Am Schluss dieser Aufgabe sollte der Aufruf `java BinTree t1.dot t2.dot` eine Ausgabe der folgenden Form erzeugen. Die Zahlen sind teilweise Zufallszahlen.

```
Aufgabe b): Zufaelliges Einfuegen
Baum als DOT File ausgegeben in Datei t1.dot
Aufgabe a): Suchen nach zufaelligen Elementen
17 ist enthalten
19 ist nicht enthalten
12 ist nicht enthalten
15 ist enthalten
12 ist nicht enthalten
13 ist nicht enthalten
3 ist enthalten
17 ist enthalten
2 ist enthalten
15 ist enthalten
26 ist enthalten
9 ist enthalten
18 ist nicht enthalten
29 ist nicht enthalten
Aufgabe c): geordnete String-Ausgabe
tree(2, 3, 6, 7, 9, 15, 17, 21, 22, 23, 24, 25, 26, 27, 28)
Aufgabe d): Suchen nach vorhandenen Elementen mit Rotation.
Baum nach Suchen von 15, 3 und 23 als DOT File ausgegeben in Datei t2.dot
Aufgabe e): merge
tree(2, 3, 6, 7, 9, 15, 17, 21, 22, 23, 24, 25, 26, 27, 28)
tree(2, 4, 5, 7, 9)
tree(2, 3, 4, 5, 6, 7, 9, 15, 17, 21, 22, 23, 24, 25, 26, 27, 28)
```

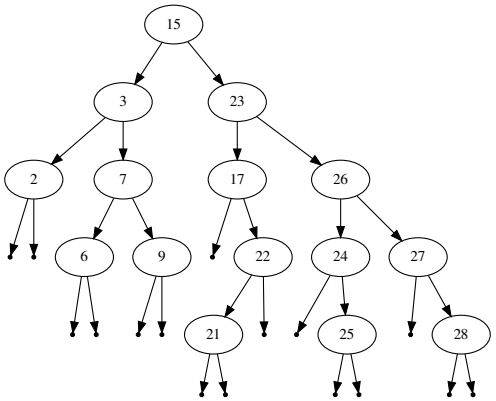
Falls Sie anschließend mit `dot -Tpdf t1.dot > t1.pdf` und `dot -Tpdf t2.dot > t2.pdf` die dot Dateien in PDF umwandeln[1], sollten Sie Bilder ähnlich zu den Folgenden erhalten.

[1] Sie benötigen hierfür das Programm Graphviz.



Listing 3: t1.dot

```
digraph {
graph [ordering="out"];
7 -> 6;
6 -> 2;
null0[shape=point]
2 -> null0;
2 -> 3;
null1[shape=point]
3 -> null1;
null2[shape=point]
3 -> null2;
null3[shape=point]
6 -> null3;
7 -> 15;
15 -> 9;
null4[shape=point]
9 -> null4;
null5[shape=point]
9 -> null5;
15 -> 17;
null6[shape=point]
17 -> null6;
17 -> 26;
26 -> 22;
22 -> 21;
null7[shape=point]
21 -> null7;
null8[shape=point]
21 -> null8;
22 -> 24;
24 -> 23;
null9[shape=point]
23 -> null9;
null10[shape=point]
23 -> null10;
24 -> 25;
null11[shape=point]
25 -> null11;
null12[shape=point]
25 -> null12;
26 -> 27;
null13[shape=point]
27 -> null13;
27 -> 28;
null14[shape=point]
28 -> null14;
null15[shape=point]
28 -> null15;
}
```



Listing 4: t2.dot

```
digraph {
graph [ordering="out"];
15 -> 3;
3 -> 2;
null0[shape=point]
2 -> null0;
null1[shape=point]
2 -> null1;
3 -> 7;
7 -> 6;
null2[shape=point]
6 -> null2;
null3[shape=point]
6 -> null3;
7 -> 9;
null4[shape=point]
9 -> null4;
null5[shape=point]
9 -> null5;
15 -> 23;
23 -> 17;
null6[shape=point]
17 -> null6;
17 -> 22;
22 -> 21;
null7[shape=point]
21 -> null7;
null8[shape=point]
21 -> null8;
null9[shape=point]
22 -> null9;
23 -> 26;
26 -> 24;
null10[shape=point]
24 -> null10;
24 -> 25;
null11[shape=point]
25 -> null11;
null12[shape=point]
25 -> null12;
26 -> 27;
null13[shape=point]
27 -> null13;
27 -> 28;
null14[shape=point]
28 -> null14;
null15[shape=point]
28 -> null15;
}
```

Wie oben erwähnt, sind die meisten Zahlen zufällig bei jedem Aufruf neu gewählt. In jedem Fall aber sollten die obersten Knoten in der zweiten Grafik die Zahlen 3, 15 und 23 sein.

In dieser Aufgabe dürfen Sie *keine* Schleifen verwenden. Die Verwendung von Rekursion ist hingegen erlaubt.

a) Implementieren Sie Methoden zum Suchen nach einer Zahl im Baum. Vervollständigen sie hierzu die Methoden `simpleSearch` in den Klassen `BinTree` und `BinTreeNode`.

Die Methode `simpleSearch` in der Klasse `BinTree` prüft, ob eine Wurzel existiert (d.h., ob der Baum nicht leer ist). Falls er leer ist, wird sofort `false` zurückgegeben. Existiert hingegen die Wurzel, wird die Methode `simpleSearch` auf der Wurzel aufgerufen.

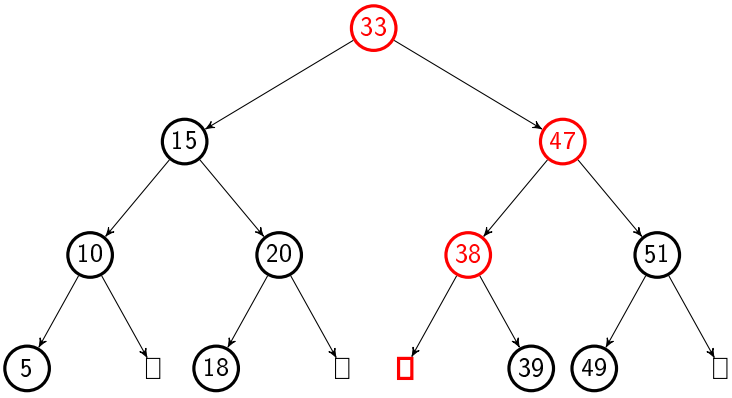
Die Methode `simpleSearch` in der Klasse `BinTreeNode` durchsucht nun den Baum nach der übergebenen Zahl. Hat der aktuelle Knoten den gesuchten Wert gespeichert, soll `true` zurückgegeben werden. Andernfalls wird eine Fallunterscheidung durchgeführt. Da der Baum sortiert ist, wird nach Zahlen, die

kleiner sind als der im aktuellen Knoten gespeicherte Wert, nur im linken Teilbaum weiter gesucht. Für Zahlen, die größer sind, muss nur im rechten Teilbaum gesucht werden. Trifft diese Suche irgendwann auf `null`, kann die Suche abgebrochen werden und es wird `false` zurückgegeben.

b) Implementieren Sie Methoden zum Einfügen einer Zahl in den Baum. Vervollständigen Sie dazu die Methoden `insert` in den Klassen `BinTreeNode` und `BinTree`.

In der Klasse `BinTree` muss zunächst überprüft werden, ob eine Wurzel existiert. Falls nein, so sollte das neue Element als Wurzel eingefügt werden. Existiert eine Wurzel, dann wird `insert` auf der Wurzel aufgerufen. In der Klasse `BinTreeNode` wird zunächst nach der einzufügenden Zahl gesucht. Wird sie gefunden, braucht nichts weiter getan zu werden (die Zahl wird also kein zweites Mal eingefügt). Existiert die Zahl noch nicht im Baum, muss ein neuer Knoten an der Stelle eingefügt werden, wo die Suche abgebrochen wurde.

Wird zum Beispiel im folgenden Baum die Zahl 36 eingefügt, beginnt die Suche beim Knoten 33, läuft dann über den Knoten 47 und wird nach Knoten 38 abgebrochen, weil der linke Nachfolger fehlt. An dieser Stelle, als linker Nachfolger von 38, wird nun die 36 eingefügt.



Hinweise:

Obwohl dem eigentlichen Einfügen eine Suche vorausgeht, ist es nicht sinnvoll, die Methode `simpleSearch` in dieser Teilaufgabe zu verwenden.

c) Schreiben Sie `toString` Methoden für die Klassen `BinTree` und `BinTreeNode`.

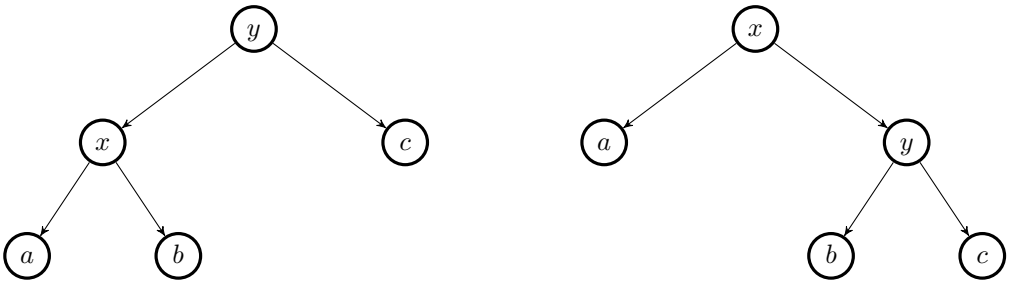
Die `toString` Methode der Klasse `BinTreeNode` soll alle Zahlen, die im aktuellen Knoten und seinen Nachfolgern gespeichert sind, aufsteigend sortiert und mit Kommas getrennt ausgeben. Ruft man beispielsweise `toString` auf dem Knoten aus dem Baum oben auf, der die Zahl 15 gespeichert hat, wäre die Ausgabe `5, 10, 15, 18, 20`.

Die `toString` Methode der Klasse `BinTree` soll die Ausgabe `tree(5, 10, 15, 18, 20, 33, 38, 39, 47, 49, 51)` für das obige Beispiel erzeugen.

d) Implementieren Sie in dieser Teilaufgabe die Methoden `search` und `rotationSearch` in der Klasse `BinTree` beziehungsweise `BinTreeNode`. Diese sollen einen alternativen Algorithmus zur Suche nach einem Wert im Baum implementieren.

Es ist sinnvoll, Elemente, nach denen häufig gesucht wird, möglichst weit oben im Baum zu speichern. Das kann realisiert werden, indem der Baum beim Aufruf der Suche so umstrukturiert wird, dass das gesuchte Element, falls es existiert, in der Wurzel steht und die übrige Struktur weitgehend erhalten wird. Da außerdem unbedingt die Sortierung erhalten bleiben muss, sollte ein spezieller Algorithmus verwendet werden.

Um einen Knoten eine Ebene im Baum nach oben zu befördern, kann die sogenannte *Rotation* verwendet werden. Soll im folgenden Beispiel x nach oben rotiert werden, wird die `left` Referenz des Vorgängerknotens y auf die `right` Referenz von x gesetzt. Anschließend wird die `right` Referenz von x auf y gesetzt. Das Ergebnis ist der rechts daneben gezeichnete Baum. Um im rechten Baum y nach oben zu rotieren, wird die Operation spiegelbildlich ausgeführt.



Diese Rotation kann nun so lange wiederholt werden, bis der Knoten mit der gesuchten Zahl in der Wurzel ist. Ist die gesuchte Zahl nicht enthalten, wird der Knoten, bei dem die Suche erfolglos abgebrochen wird, in die Wurzel rotiert.

Hinweise:

Die Signatur und Dokumentation der vorgegebenen Methoden geben Ihnen weitere Hinweise, wie die Rotation eines Knotens in die Wurzel rekursiv implementiert werden kann.

e) Implementieren Sie in der Klasse `BinTree` die statische Methode `merge`, welche als Parameter eine beliebige Anzahl von `BinTree`-Objekten erhält und ein neues `BinTree`-Objekt erstellt, welches genau die Zahlen enthält, die auch in mindestens einem der übergebenen `BinTree`-Objekte enthalten waren. Das neu erstellte `BinTree`-Objekt darf keine Zahl mehrfach enthalten.

Lösung:

Listing 5: BinTree.java

```
/**
* Import von Paketen, in denen benoetigte Library-Hilfsfunktionen zu finden sind
*/
import java.util.Random ;
import java.nio.file.*;
import java.io.*;
/**
 * Implementiert einen sortierten Binaerbaum mit Rotation-zur-Wurzel Optimierung.
 */
public class BinTree {
  /**
   * Wurzel des Baums
   */
  private BinTreeNode root;
  /**
   * Erstellt einen leeren Baum
   */
  public BinTree() {
    this.root = null ;
  }
  /**
   * Erstellt einen Baum mit den vorgegebenen Zahlen
   * @param xs die einzupflegenden Zahlen
   */
  public BinTree(int... xs) {
    for ( int x : xs ) {
      this.insert(x);
    }
  }
  /**
   * Test ob der Baum leer ist
   * @return true, falls der Baum leer ist, sonst false
   */
  public boolean isEmpty() {
    return this.root == null ;
  }

  /**
   * Fuegt alle Zahlen aus den Baeumen in einen neuen Baum ein und gibt diesen zurueck.
   * @param trees Die Baeume mit den einzufuegenden Zahlen.
   * @return Der neue Baum mit allen Zahlen.
   */
  public static BinTree merge(BinTree... trees) {
    BinTree result = new BinTree();
    doMerge(result, trees, 0);
```

```
  return result;
}

/**
 * Fuegt alle Zahlen aus den Baeumen mit einem gegebenen Mindestindex in den Ergebnisbaum ein.
 * @param result Der Ergebnisbaum.
 * @param trees Die Baeume mit den einzufuegenden Zahlen.
 * @param index Der Mindestindex.
 */
private static void doMerge(BinTree result, BinTree[] trees, int index) {
  if (index < trees.length) {
    insertAll(result, trees[index].root);
    doMerge(result, trees, index + 1);
  }
}

/**
 * Fuegt alle Zahlen in den Ergebnisbaum ein.
 * @param result Der Ergebnisbaum.
 * @param toInsert Der Baumknoten mit den einzufuegenden Zahlen.
 */
private static void insertAll(BinTree result, BinTreeNode toInsert) {
  if (toInsert != null) {
    insertAll(result, toInsert.getLeft());
    result.insert(toInsert.getValue());
    insertAll(result, toInsert.getRight());
  }
}

/**
 * Fuegt eine Zahl ein. Keine Aenderung, wenn das Element
 * schon enthalten ist.
 * @param x einzufuegende Zahl
 */
public void insert(int x) {
  if (this.isEmpty()) {
    this.root = new BinTreeNode(x);
  } else {
    this.root.insert(x);
  }
}
/**
 * Sucht x, ohne den Baum zu veraendern.
 * @return true, falls x im Baum enthalten ist, sonst false
 * @param x der gesuchte Wert
 */
public boolean simpleSearch(int x) {
  if (this.isEmpty ()) {
    return false;
  } else {
    return this.root.simpleSearch(x);
  }
}
/**
 * Sucht x und rotiert den Knoten, bei dem die Suche nach x endet, in die Wurzel.
 * @param x der gesuchte Wert
 * @return true, falls x im Baum enthalten ist, sonst false
 */
  public boolean search(int x) {
    if (this.isEmpty()) {
      return false;
    } else {
      this.root = this.root.rotationSearch(x);
      return this.root.getValue () == x ;
    }
  }
/**
 * @return Sortierte Ausgabe aller Elemente.
 */
public String toString() {
  if ( this.isEmpty ()) {
    return "tree()";
  }
  return "tree(" + this.root.toString () + ")";
}
/**
 * Wandelt den Baum in einen Graphen im dot Format um.
 * @return der umgewandelte Baum
 */
public String toDot() {
  if ( this.isEmpty ()) {
    return "digraph { null[shape=point]; }";
  }
  StringBuilder str = new StringBuilder ();
```

```
    this.root.toDot (str, 0);
    return "digraph { " + System.lineSeparator ()
      + "graph[ordering=\"out\"]; " + System.lineSeparator ()
      + str.toString ()
      + "}" + System.lineSeparator ();
  }
  /**
   * Speichert die dot Repraesentation in einer Datei.
   *
   * @param path Pfad unter dem gespeichert werden soll (Dateiname)
   * @return true, falls erfolgreich gespeichert wurde, sonst false
   * @see toDot
   */
  public boolean writeToFile(String path) {
    boolean retval = true;
    try {
      Files.write(FileSystems.getDefault().getPath(path), this.toDot().getBytes());
    } catch (IOException x) {
      System.err.println("Es ist ein Fehler aufgetreten.");
      System.err.format("IOException: %s%n" , x);
      retval = false;
    }
    return retval;
  }
  /**
   * Main-Methode, die einige Teile der Aufgabe testet.
   *
   * @param args Liste von Dateinamen, unter denen Baeume als dot
   * gespeichert werden sollen. Es werden nur die ersten beiden verwendet.
   */
  public static void main(String[] args) {
    Random prng = new Random();
    int nodeCount = prng.nextInt(10) + 5;
    BinTree myTree = new BinTree();
    System.out.println("Aufgabe b): Zufaelliges Einfuegen");
    for(int i = 0; i < nodeCount; ++i) {
      myTree.insert(prng.nextInt(30));
    }
    myTree.insert(15);
    myTree.insert(3);
    myTree.insert(23);
    if (args.length > 0) {
      if (myTree.writeToFile(args[0])) {
        System.out.println("Baum als DOT File ausgegeben in Datei " + args [0]);
      }
    } else {
      System.out.println("Keine Ausgabe des Baums in Datei, zu wenige Aufrufparameter.");
    }
    System.out.println("Aufgabe a): Suchen nach zufaelligen Elementen");
    for(int i = 0; i < nodeCount; ++i) {
      int x = prng.nextInt (30);
      if(myTree.simpleSearch(x)) {
        System.out.println(x + " ist enthalten");
      } else {
        System.out.println(x + " ist nicht enthalten");
      }
    }
    System.out.println("Aufgabe c): geordnete String-Ausgabe");
    System.out.println(myTree.toString());
    System.out.println("Aufgabe d): Suchen nach vorhandenen Elementen mit Rotation.");
    myTree.search(3);
    myTree.search(23);
    myTree.search(15);
    if (args.length > 1) {
      if (myTree.writeToFile(args[1])) {
        System.out.println("Baum nach Suchen von 15, 3 und 23 als DOT File ausgegeben in Datei "
            + args [1]);
      }
    } else {
      System.out.println("Keine Ausgabe des Baums in Datei, zu wenige Aufrufparameter.");
    }

    System.out.println("Aufgabe e): merge");
    BinTree tree2 = new BinTree(4, 7, 2,9 ,5);
    System.out.println(myTree.toString());
    System.out.println(tree2.toString());
    System.out.println(BinTree.merge(myTree, tree2).toString());
  }
}
```

Listing 6: BinTreeNode.java

```
/**
```

```
 * Ein Knoten in einem binaeren Baum.
 *
 * Der gespeicherte Wert ist unveraenderlich,
 * die Referenzen auf die Nachfolger koennen aber
 * geaendert werden.
 *
 * Die Klasse bietet Methoden, um Werte aus einem Baum
 * zu suchen und einzufuegen. Die Methode zur Suche gibt
 * es noch in einer optimierten Variante, um
 * rotate-to-root Baeume zu verwalten.
 */
public class BinTreeNode {
  /**
   * Linker Nachfolger
   */
  private BinTreeNode left;
  /**
   * Rechter Nachfolger
   */
  private BinTreeNode right;
  /**
   * Wert, der in diesem Knoten gespeichert ist
   */
  private final int value;

  /**
   * Erzeugt einen neuen Knoten ohne Nachfolger
   * @param val Wert des neuen Knotens
   */
  public BinTreeNode(int val) {
    this.value = val;
    this.left = null;
    this.right = null;
  }

  /**
   * Erzeugt einen neuen Knoten mit den gegebenen Nachfolgern
   * @param val Wert des neuen Knotens
   * @param left linker Nachfolger des Knotens
   * @param right rechter Nachfolger des Knotens
   */
  public BinTreeNode(int val, BinTreeNode left, BinTreeNode right) {
    this.value = val;
    this.left = left;
    this.right = right;
  }
  /**
  * @return Wert des aktuellen Knotens
  *
  */
  public int getValue() {
    return this.value;
  }

  /**
   * @return Der gespeicherte Wert, umgewandelt in einen String
   */
  public String getValueString() {
    return Integer.toString(this.value);
  }

  /**
   * @return true, falls der Knoten einen linken Nachfolger hat, sonst false
   */
  public boolean hasLeft() {
    return this.left != null;
  }

  /**
   * @return true, falls der Knoten einen rechten Nachfolger hat, sonst false
   */
  public boolean hasRight() {
    return this.right != null;
  }
  /**
   * @return linker Nachfolger des aktuellen Knotens
   */
  public BinTreeNode getLeft() {
    return this.left;
  }
  /**
   * @return rechter Nachfolger des aktuellen Knotens
   */
  public BinTreeNode getRight() {
```

```
  return this.right;
}

/**
 * Sucht in diesem Teilbaum nach x, ohne den Baum zu veraendern.
 * @param x der gesuchte Wert
 * @return true, falls x enthalten ist, sonst false
 */
public boolean simpleSearch(int x) {
  if(this.value == x) {
    return true;
  } else if(this.value > x && this.hasLeft()) {
    return this.left.simpleSearch(x);
  } else if(this.value < x && this.hasRight()) {
    return this.right.simpleSearch(x);
  } else {
    return false;
  }
}

/**
 * Fuegt x in diesen Teilbaum ein.
 * @param x der einzufuegende Wert
 */
public void insert(int x) {
  if(this.value == x) {
    return;
  }else if(this.value > x) {
    if(this.hasLeft()) {
      this.left.insert(x);
    } else {
      this.left = new BinTreeNode(x);
    }
  } else {
    if(this.hasRight()) {
      this.right.insert(x);
    } else {
      this.right = new BinTreeNode(x);
    }
  }
}

/**
 * Sucht in diesem Teilbaum nach x und rotiert den Endpunkt der Suche in die
 * Wurzel.
 * @param x der gesuchte Wert
 * @return die neue Wurzel des Teilbaums
 */
public BinTreeNode rotationSearch(int x) {
  if(this.value > x && this.hasLeft()) {
    BinTreeNode root = this.left.rotationSearch(x);
    this.left = root.right;
    root.right = this;
    return root;
  } else if(this.value < x && this.hasRight()) {
    BinTreeNode root = this.right.rotationSearch(x);
    this.right = root.left;
    root.left = this;
    return root;
  } else {
    return this;
  }
}

/**
 * @return Geordnete Liste aller Zahlen, die in diesem Teilbaum gespeichert sind.
 */
public String toString() {
  String str = this.getValueString();
  if(this.hasLeft()) {
    str = this.left.toString() + ", " + str;
  }
  if(this.hasRight()) {
    str = str + ", " + this.right.toString();
  }
  return str;
}

/**
 * Erzeugt eine dot Repraesentation in str
 * @param str Stringbuilder Objekt zur Konstruktion der Ausgabe
 * @param nullNodes Hilfsvariable, um Nullknoten zu indizieren. Anfangswert sollte 0 sein.
 * @return Den nullNodes Wert fuer den behandelten Baum
 */
```

```
  public int toDot(StringBuilder str, int nullNodes) {
    if(this.hasLeft()) {
      str.append(this.getValueString() + " -> " + this.left.getValueString() + ";"
        + System.lineSeparator());
      nullNodes = this.left.toDot(str, nullNodes);
    } else {
      str.append("null" + nullNodes + "[shape=point]" + System.lineSeparator()
        + this.getValueString() + " -> null" + nullNodes + ";" + System.lineSeparator());
      nullNodes += 1;
    }
    if(this.hasRight()) {
      str.append(this.getValueString() + " -> " + this.right.getValueString() + ";"
        + System.lineSeparator());
      nullNodes = this.right.toDot(str, nullNodes);
    } else {
      str.append("null" + nullNodes + "[shape=point]" + System.lineSeparator()
        + this.getValueString() + " -> null" + nullNodes + ";" + System.lineSeparator());
      nullNodes += 1;
    }
    return nullNodes;
  }

}
```

Tutoraufgabe 4 (Entwurf einer Klassenhierarchie):

In dieser Aufgabe sollen Sie einen Teil der Tierwelt modellieren.

- Ein Tier kann ein Säugetier, ein Wurm oder ein Insekt sein. Jedes Tier hat ein Alter.
- Ein spezielles Merkmal der Säugetiere ist ihr Fell. Für jedes Säugetier ist somit die Anzahl der Haare pro Quadratzentimeter Haut bekannt.
- Verschiedene Wurmarten haben im Allgemeinen wenig gemeinsam. Jeder Wurm hat jedoch eine bekannte Länge in Zentimetern.
- Alle Insekten haben einen Chitinpanzer. Bekannt ist, wie viel Druck in Pascal der Chitinpanzer eines Insektes aushalten kann.
- Menschen sind Säugetiere. Sie sind der Meinung, dass Intelligenz eines ihrer besonderen Merkmale sei. Deswegen ist der IQ jedes Menschen bekannt.
- Bandwürmer sind Würmer. Sie haben die Angewohnheit, Menschen zu befallen. Ihr vielleicht wichtigstes Merkmal ist ihr Wirt, ein Mensch, ohne den sie nicht lange überleben können.
- Bienen und Ohrwürmer sind Insekten. Das heißt insbesondere, dass Ohrwürmer keine Würmer sind.
- Bienen stechen Säugetiere, wenn sie sich bedroht fühlen.
- Ohrwürmer verfügen über Zangen, deren Größe in Millimeter in der Welt der Ohrwürmer von großer Bedeutung ist. Folglich ist die Zangengröße jedes Ohrwurms bekannt. Außerdem verwend(et)en Menschen Ohrwürmer als Medizin zur Behandlung von Erkrankungen der Ohren eines Menschen.

Entwerfen Sie unter Berücksichtigung der Prinzipien der Datenkapselung eine geeignete Klassenhierarchie für die oben aufgelisteten Arten von Tieren. Notieren Sie keine Konstruktoren. Um Schreibarbeit zu sparen, brauchen Sie keine Selektoren anzugeben. Achten Sie darauf, dass gemeinsame Merkmale in Oberklassen zusammengefasst werden. Ergänzen Sie außerdem geeignete Methoden, um die Behandlung von Ohrenerkrankungen, den Bandwurm-Befall und den Bienenstich abzubilden.
Verwenden Sie hierbei die Notation aus Abb. 1. 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 $B \longrightarrow\!\!\!\!\triangleright\, A$, dass A die Oberklasse von B ist (also `class B extends A`) und A `->` B, dass A den Typ B in den Typen seiner Attribute oder in den Ein- oder Ausgabeparametern seiner Methoden verwendet. Benutzen Sie ein `-` um `private` und ein `+` um `public` abzukürzen.
Tragen Sie keine vordefinierten Klassen (`String`, etc.) oder Pfeile dorthin in Ihr Diagramm ein.

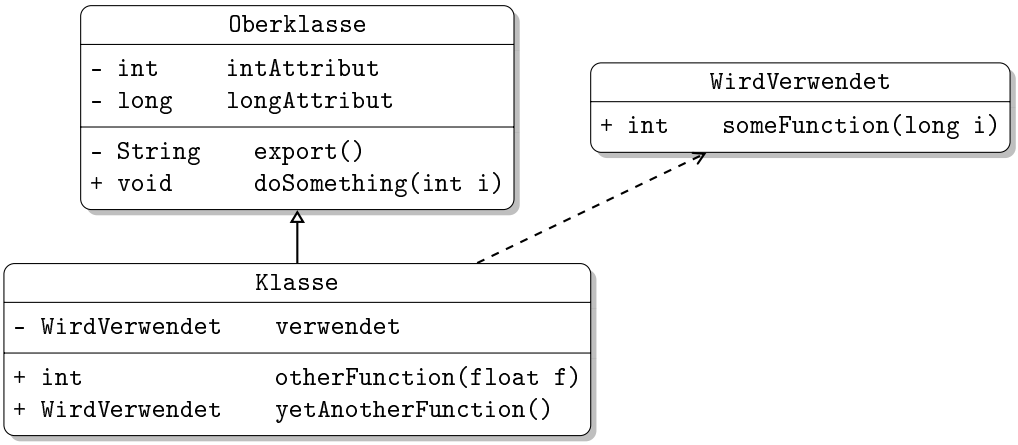
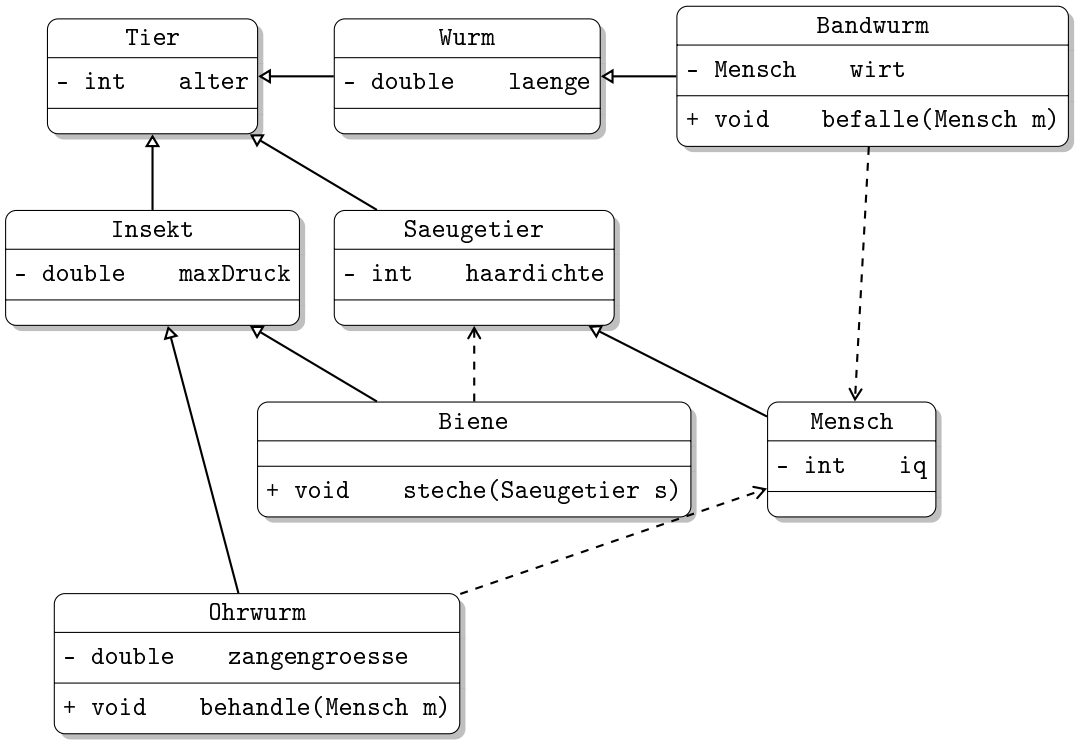


Abbildung 1: Graphische Notation zur Darstellung von Klassen.

Lösung:



Aufgabe 5 (Entwurf einer Klassenhierarchie): (14 Punkte)

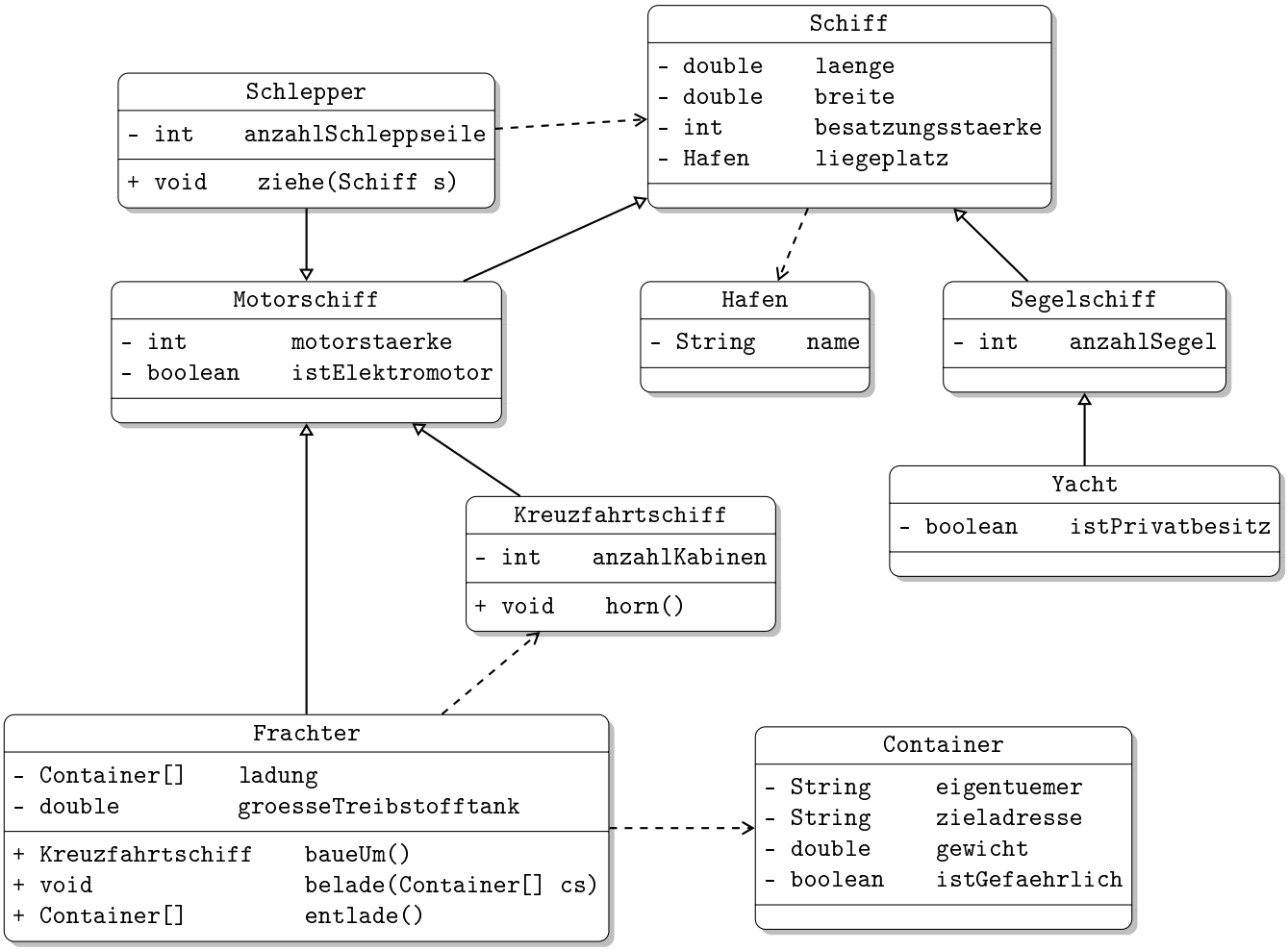
In dieser Aufgabe sollen Sie einen Teil der Schifffahrt modellieren.

- Die wichtigste Eigenschaft eines Hafens ist sein Name.
- Ein Schiff kann ein Segelschiff oder ein Motorschiff sein. Jedes Schiff hat eine Länge und eine Breite in Metern und eine Anzahl an Besatzungsmitgliedern. Ein Schiff hat seinen Liegeplatz in einem Hafen.
- Ein Segelschiff zeichnet sich durch die Anzahl seiner Segel aus.

- Die wichtigste Kennzahl eines Motorschiffs ist die PS-Stärke des Motors. Der Motor kann außerdem ein Dieselmotor oder ein Elektromotor sein.
- Ein Kreuzfahrtschiff ist ein Motorschiff. Es hat eine Anzahl an Kabinen und kann das Schiffshorn ertönen lassen.
- Eine Yacht ist ein Segelschiff. Yachten können in Privatbesitz sein oder nicht.
- Schlepper sind Motorschiffe mit einer Anzahl von Schleppseilen. Ein Schlepper kann ein beliebiges Schiff ziehen.
- Frachter sind Motorschiffe. Sie enthalten eine Ladung, die aus einer Sammlung von Containern besteht. Außerdem sind sie durch die Größe ihres Treibstofftanks in Litern gekennzeichnet. Ein Frachter kann mit Containern be- und entladen werden, wobei beim Entladen alle Container vom Frachter entfernt werden.
- Ein Container zeichnet sich durch seinen Eigentümer, seine Zieladresse und das Gewicht seines Inhalts aus. Zudem kann der Inhalt gefährlich sein oder nicht.
- In einem aufwendigen Verfahren kann ein Frachter zu einem Kreuzfahrtschiff umgebaut werden.

Entwerfen Sie unter Berücksichtigung der Prinzipien der Datenkapselung eine geeignete Klassenhierarchie für die oben aufgelisteten Arten von Schiffen. Notieren Sie keine Konstruktoren. Um Schreibarbeit zu sparen, brauchen Sie keine Selektoren anzugeben. Achten Sie darauf, dass gemeinsame Merkmale in Oberklassen zusammengefasst werden, falls dies sinnvoll ist. Ergänzen Sie außerdem geeignete Methoden, um das Beladen, Entladen, das Umbauen, das Ziehen und das Erklingen lassen des Horns abzubilden.
Tragen Sie keine vordefinierten Klassen (`String`, etc.) oder Pfeile dorthin in Ihr Diagramm ein.
Verwenden Sie hierbei die Notation aus der entsprechenden Tutoriumsaufgabe.

Lösung:



Aufgabe 6 (Deck 6): (Codescape)

Lösen Sie die Missionen von Deck 6 des Codescape Spiels. Ihre Lösung für die Codescape Missionen wird nur dann für die Zulassung gezählt, wenn Sie Ihre Lösung vor der einheitlichen Codescape Deadline am Samstag, den 22.01.2022, um 23:59 Uhr abschicken.

Lösung: