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());
    }
}



/**
 * 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;
        }
    }

}
```

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 $B \twoheadrightarrow A$, 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.

Lösung:

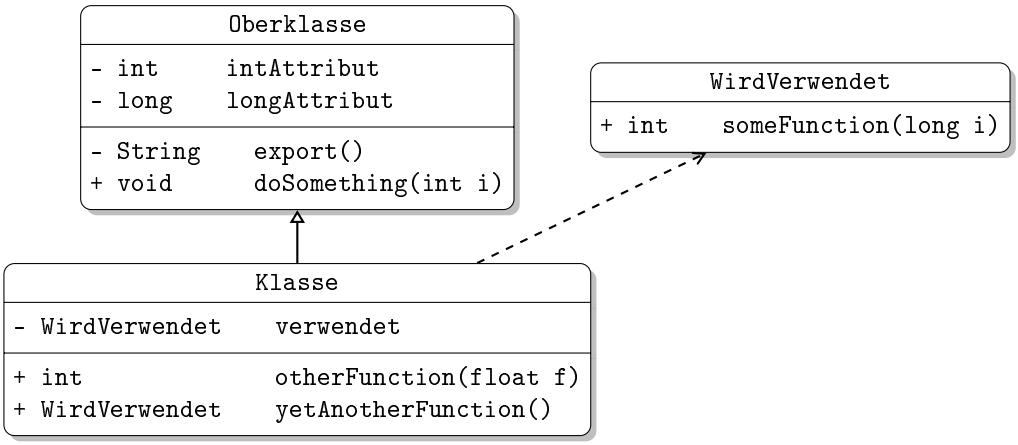

Abbildung 1: Graphische Notation zur Darstellung von Klassen.