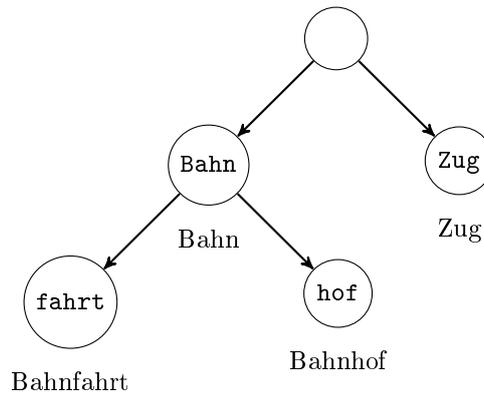


**Aufgabe 1 (Collections):**

**(9 + 3\* + 2\* + 3\* = 9 + 8\* Punkte)**

In dieser Aufgabe geht es ebenfalls um die Implementierung einer Datenstruktur für Mengen, welche in das bestehende Collections Framework eingebettet werden soll. Sie benötigen dafür die Klassen `TrieSet` und `TrieNode`, welche Sie als `.java` Dateien von unserer Webseite herunterladen können.

Die in dieser Aufgabe zu betrachtende Mengenstruktur basiert auf einer Suchbaumstruktur zur Indizierung der Elemente. Die hier verwendeten Suchbäume werden *Tries* genannt (von retrieval). Jedem Element der Menge wird ein Schlüssel als `String` zugeordnet. Die Knoten des Tries enthalten nun Teil-Strings, anhand derer die Suche durch den Baum nach einem Element gesteuert wird. Die Konkatenation der Teil-Strings entlang der besuchten Knoten im Baum ergibt den Schlüssel des gesuchten Elements. Die Wurzel enthält dabei immer den leeren `String`, da dieser ein Präfix für jeden `String` ist. Folgender Trie speichert beispielsweise vier Elemente mit den Schlüsseln `Bahn`, `Bahnfahrt`, `Bahnhof` und `Zug`:



In diesem Beispiel ist jedes Element identisch zu seinem Schlüssel. Dies muss im Allgemeinen nicht so sein. Insbesondere kann ein Knoten in einem Trie auch mehrere verschiedene Elemente speichern, wenn deren Schlüssel identisch sind. Allerdings werden gleiche Elemente nicht mehrfach gespeichert, da es sich ja um eine Mengenstruktur handelt. Jeder Knoten enthält daher ein Attribut `elements` vom Typ `Set`.

Die vorgegebene Klasse `TrieSet` implementiert bereits das `Set` Interface bis auf die Methoden `iterator`, `retainAll` und `add` (letztere ist eigentlich in `TrieSet` vollständig implementiert, aber die von ihr aufgerufene `add` Methode in der Klasse `TrieNode` ist es nicht).

Sie können die `main` Methode der Klasse `TrieSet` nutzen, um Ihre Implementierungen der folgenden Aufgabenteile zu testen. Dabei können Sie auch Teile des Tests auskommentieren, falls Sie nur Teile der Aufgaben bearbeitet haben.

- a) Implementieren Sie die Methode `iterator` in der Klasse `TrieSet`. Implementieren Sie dazu eine generische Klasse `TrieIterator<E>`, welche das Interface `Iterator<E>` aus dem Package `java.util` implementiert. Schlagen Sie für die zu implementierenden Methoden `hasNext`, `next` und `remove` die Funktionalitäten in der Java API für das Interface `Iterator` nach (die `remove` Operation soll durch Ihren `Iterator` unterstützt werden). Dies betrifft insbesondere auch die durch diese Methoden zu werfenden Exceptions. Die Klasse `TrieIterator` muss im gleichen Package wie `TrieNode` implementiert werden, damit Sie auf die Getter-Methoden der Klasse `TrieNode` zugreifen können.

**Hinweise:**

- Denken Sie daran, dass die `Set` Attribute an den jeweiligen Knoten bereits implementierte Iteratoren anbieten.
  - Sie finden Dokumentationen zu weiteren Klassen aus dem Collections Framework (wie z. B. `TreeMap`) ebenfalls in der Java API.
- b) Implementieren Sie die Methode `retainAll` in der Klasse `TrieSet`. Schlagen Sie deren Funktionalität ebenfalls in der Java API nach. Benutzen Sie zur Implementierung dieser Methode nicht die `add` oder `remove` Methoden der Klassen `TrieSet` und `TrieNode`. Sie dürfen jedoch die `remove` Methode des `Iterator` aus der vorigen Teilaufgabe nutzen.

- c) Überschreiben Sie die Methode `public String toString()` in der Klasse `TrieSet`, sodass diese einen `String` zurückliefert, der mit einer öffnenden geschweiften Klammer beginnt, dann die Elemente der Menge durch Kommata getrennt aufzählt und schließlich mit einer schließenden geschweiften Klammer endet. Wenn ein `TrieSet` also beispielsweise die Elemente 1, 2 und 3 enthält, soll der von `toString` zurückgelieferte `String` "{1, 2, 3}" sein.
- d)\* Implementieren Sie die Methode `add` in der Klasse `TrieNode`, sodass die vorgegebene Implementierung der `add` Methode in der Klasse `TrieSet` korrekt bzgl. der Spezifikation im `Set` Interface ist. Achten Sie darauf, dass es zu keiner Zeit einen Knoten im Trie geben darf, der verschiedene Kindknoten hat, deren Teil-Strings aber ein gemeinsames nicht-leeres Präfix haben. Sie finden nützliche Methoden zur Verarbeitung von `Strings` in der zugehörigen Java API.

Lösung: \_\_\_\_\_

Listing 1: TrieSet.java

```
import java.lang.reflect.*;
import java.util.*;

/**
 * A TrieSet stores prefix Strings in a search tree to find its elements.
 * @author Thomas Stroeder
 * @param <E> The type of this TrieSet's elements.
 */
public class TrieSet<E> implements Set<E> {

    /**
     * Test program.
     * @param args Ignored.
     */
    public static void main(String[] args) {
        TrieSet<Integer> intSet = TrieNode.createIntegerTestCase();
        /*
         * iterator test should yield:
         * null
         * 10
         * 12
         * 20
         * 200
         * 205
         */
        Iterator<Integer> it = intSet.iterator();
        while (it.hasNext()) {
            Integer element = it.next();
            System.out.println(element);
            if (element == null || element % 2 != 0) {
                it.remove();
            }
        }
        // toString() test should yield: {10, 12, 20, 200}
        System.out.println(intSet);
        // retainAll() test should yield: {10, 200}
        Set<Integer> otherSet = new LinkedHashSet<Integer>();
        otherSet.add(3);
        otherSet.add(10);
        otherSet.add(200);
        intSet.retainAll(otherSet);
        System.out.println(intSet);
        // add() test should not throw an exception
        intSet.clear();
        otherSet.clear();
        Random gen = new Random();
        for (int i = 0; i < 100000; i++) {
            Integer v = gen.nextInt(300) + 1;
            if (gen.nextInt(3) > 0) {
                intSet.add(v);
                otherSet.add(v);
            } else {
                final int size = otherSet.size();
                if (size > 0 && gen.nextInt(3) > 0) {
                    v = gen.nextInt(size);
                    for (Integer e : otherSet) {
                        v--;
                        if (v <= 0) {
                            v = e;
                            break;
                        }
                    }
                }
            }
        }
    }
}
```

```

        }
    }
    intSet.remove(v);
    otherSet.remove(v);
}
if (!intSet.equals(otherSet)) {
    throw new IllegalStateException("The two sets are not equal!");
}
}
}

/**
 * The root node of the Trie storing the elements of this set.
 */
private final TrieNode<E> root;

/**
 * Creates an empty TrieSet.
 */
public TrieSet() {
    this.root = new TrieNode<E>();
}

/**
 * Creates a TrieSet containing the elements in the specified collection.
 * @param c The collection.
 */
public TrieSet(Collection<? extends E> c) {
    this();
    this.addAll(c);
}

/**
 * Creates a TrieSet with the specified node as root.
 * @param node The root of the Trie.
 */
protected TrieSet(TrieNode<E> node) {
    this.root = node;
}

@Override
public boolean add(E e) {
    return this.root.add(e == null ? "" : e.toString(), e);
}

@Override
public boolean addAll(Collection<? extends E> c) {
    boolean res = false;
    for (E element : c) {
        res |= this.add(element);
    }
    return res;
}

@Override
public void clear() {
    this.root.clear();
}

@Override
public boolean contains(Object o) {
    return this.root.contains(o == null ? "" : o.toString(), o);
}

@Override
public boolean containsAll(Collection<?> c) {
    for (Object o : c) {
        if (!this.contains(o)) {
            return false;
        }
    }
    return true;
}

@Override
public boolean equals(Object o) {
    if (o instanceof Set) {
        Set<?> s = (Set<?>)o;
        return this.containsAll(s) && s.containsAll(this);
    } else if (o instanceof Collection) {
        Collection<?> c = (Collection<?>)o;
        if (c.containsAll(this)) {
            Set<Object> check = new LinkedHashSet<Object>();

```

```

        for (Object element : c) {
            if (check.contains(element)) {
                return false;
            }
            check.add(element);
            if (!this.contains(element)) {
                return false;
            }
        }
        return true;
    }
}
return false;
}

@Override
public int hashCode() {
    int res = 101;
    for (E element : this) {
        res += element.hashCode() * 17;
    }
    return res;
}

@Override
public boolean isEmpty() {
    return this.iterator().hasNext();
}

@Override
public Iterator<E> iterator() {
    return new TrieIterator<E>(this.root);
}

@Override
public boolean remove(Object o) {
    return this.root.remove(o == null ? "" : o.toString(), o);
}

@Override
public boolean removeAll(Collection<?> c) {
    boolean res = false;
    for (Object o : c) {
        res |= this.remove(o);
    }
    return res;
}

@Override
public boolean retainAll(Collection<?> c) {
    Iterator<E> iterator = this.iterator();
    boolean res = false;
    while (iterator.hasNext()) {
        E element = iterator.next();
        if (!c.contains(element)) {
            iterator.remove();
            res = true;
        }
    }
    return res;
}

@Override
public int size() {
    int res = 0;
    for (Iterator<E> it = this.iterator(); it.hasNext(); it.next()) {
        res++;
    }
    return res;
}

@Override
public Object[] toArray() {
    Object[] res = new Object[this.size()];
    int index = 0;
    for (E element : this) {
        res[index++] = element;
    }
    return res;
}

@Override
public <T>T[] toArray(T[] a) {

```

```

    int size = this.size();
    T[] res = a;
    if (a.length < size) {
        res = (T[])Array.newInstance(a.getClass(), this.size());
    }
    int index = 0;
    for (E element : this) {
        try {
            res[index++] = (T)element;
        } catch (ClassCastException e) {
            throw new ArrayStoreException(
                "Element " + element + " cannot be cast to type of specified array!"
            );
        }
    }
    if (res.length > index) {
        res[index] = null;
    }
    return res;
}

@Override
public String toString() {
    StringBuilder res = new StringBuilder();
    res.append("{");
    boolean notFirst = false;
    for (E element : this) {
        if (notFirst) {
            res.append(", ");
        } else {
            notFirst = true;
        }
        res.append(element);
    }
    res.append("}");
    return res.toString();
}
}
}

```

Listing 2: TrieNode.java

```

import java.util.*;

/**
 * A node in a TrieSet stores elements whose String representation has been consumed completely and further
 * prefixes to search objects by their remaining prefixes.
 * @author Thomas Stroeder
 * @param <E> The type of this TrieSet's elements.
 */
public class TrieNode<E> {

    /**
     * @return A TrieSet containing Integers for testing purposes.
     */
    public static TrieSet<Integer> createIntegerTestCase() {
        TrieNode<Integer> res = new TrieNode<Integer>();
        res.elements.add(null);
        TrieNode<Integer> oneNode = new TrieNode<Integer>();
        TrieNode<Integer> tenNode = new TrieNode<Integer>();
        tenNode.elements.add(10);
        TrieNode<Integer> twelveNode = new TrieNode<Integer>();
        twelveNode.elements.add(12);
        oneNode.prefixes.put("0", tenNode);
        oneNode.prefixes.put("2", twelveNode);
        TrieNode<Integer> twentyNode = new TrieNode<Integer>();
        twentyNode.elements.add(20);
        TrieNode<Integer> twoHundredNode = new TrieNode<Integer>();
        twoHundredNode.elements.add(200);
        TrieNode<Integer> twoHundredAndFiveNode = new TrieNode<Integer>();
        twoHundredAndFiveNode.elements.add(205);
        twentyNode.prefixes.put("0", twoHundredNode);
        twentyNode.prefixes.put("5", twoHundredAndFiveNode);
        res.prefixes.put("1", oneNode);
        res.prefixes.put("20", twentyNode);
        return new TrieSet<Integer>(res);
    }

    /**
     * The elements stored at this node (i.e., their String representation is completely consumed at this
     * node).
     */
    private final Set<E> elements;
}

```

```

/**
 * The remaining prefixes of elements stored in the TrieSet represented by this node and its
 * successors.
 */
private final TreeMap<String, TrieNode<E>> prefixes;

/**
 * Creates a new TrieNode without any elements.
 */
public TrieNode() {
    this.elements = new LinkedHashSet<E>();
    this.prefixes = new TreeMap<String, TrieNode<E>>();
}

/**
 * Adds the specified element to this Trie.
 * @param key The remaining String representation of the element.
 * @param element The element.
 * @return True if this Trie has been modified by adding the element (if the element was already
 * present, it is not added again). False otherwise.
 */
public boolean add(String key, E element) {
    if ("".equals(key)) {
        return this.elements.add(element);
    }
    String firstChar = key.substring(0, 1);
    for (Map.Entry<String, TrieNode<E>> entry : this.prefixes.entrySet()) {
        String existingKey = entry.getKey();
        if (key.startsWith(existingKey)) {
            return entry.getValue().add(key.substring(existingKey.length()), element);
        } else if (existingKey.startsWith(firstChar)) {
            char[] existingChars = existingKey.toCharArray();
            char[] newChars = key.toCharArray();
            int index = 1;
            final int limit = Math.min(existingChars.length, newChars.length);
            while (index < limit && newChars[index] == existingChars[index]) {
                index++;
            }
            TrieNode<E> newNode = new TrieNode<E>();
            TrieNode<E> existingNode = entry.getValue();
            this.prefixes.remove(existingKey);
            this.prefixes.put(key.substring(0, index), newNode);
            if (index == newChars.length) {
                newNode.elements.add(element);
            } else {
                TrieNode<E> furtherNode = new TrieNode<E>();
                newNode.prefixes.put(key.substring(index), furtherNode);
                furtherNode.elements.add(element);
            }
            newNode.prefixes.put(existingKey.substring(index), existingNode);
            return true;
        }
    }
    TrieNode<E> node = new TrieNode<E>();
    node.elements.add(element);
    this.prefixes.put(key, node);
    return true;
}

/**
 * Removes all elements from this Trie.
 */
public void clear() {
    this.prefixes.clear();
    this.elements.clear();
}

/**
 * @param key The remaining String representation of the object.
 * @param o The object.
 * @return True if the remaining TrieSet represented by this node and its successors contains the
 * specified object. False otherwise.
 */
public boolean contains(String key, Object o) {
    if ("".equals(key)) {
        return this.elements.contains(o);
    }
    for (Map.Entry<String, TrieNode<E>> entry : this.prefixes.entrySet()) {
        if (key.startsWith(entry.getKey())) {
            return entry.getValue().contains(key.substring(entry.getKey().length()), o);
        }
    }
    return false;
}

```

```

}

/**
 * Removes the specified object from this Trie.
 * @param key The remaining String representation of the object.
 * @param o The object.
 * @return True if this Trie has been modified by the removal. False otherwise.
 */
public boolean remove(String key, Object o) {
    if ("".equals(key)) {
        return this.elements.remove(o);
    }
    for (Map.Entry<String, TrieNode<E>> entry : this.prefixes.entrySet()) {
        if (key.startsWith(entry.getKey())) {
            return entry.getValue().remove(key.substring(entry.getKey().length()), o);
        }
    }
    return false;
}

/**
 * @return The elements stored at this node.
 */
Set<E> getElements() {
    return this.elements;
}

/**
 * @return The remaining prefixes of elements stored in the TrieSet represented by this node and its
 * successors.
 */
Map<String, TrieNode<E>> getPrefixes() {
    return this.prefixes;
}
}
}

```

Listing 3: TrieIterator.java

```

import java.util.*;

/**
 * Iterator for Tries.
 * @author Thomas Stroeder
 * @param <E> The type of the elements in the iterated Trie.
 */
public class TrieIterator<E> implements Iterator<E> {

    /**
     * Iterators for the sets contained in the Trie in the order induced by the Trie.
     */
    private final Deque<Iterator<E>> iterators;

    /**
     * The iterator to fetch the next elements from.
     */
    private Iterator<E> nextIterator;

    /**
     * The iterator to remove an element from in case the remove method is called.
     */
    private Iterator<E> previousIterator;

    /**
     * Creates an iterator over the elements in the Trie by gathering iterators for all sets
     * contained in the Trie in the corresponding order.
     * @param root The root node of the Trie.
     */
    public TrieIterator(TrieNode<E> root) {
        this.iterators = new ArrayDeque<Iterator<E>>();
        this.collectIterators(root);
        this.nextIterator = this.iterators.pollFirst();
        this.previousIterator = null;
    }

    @Override
    public boolean hasNext() {
        if (this.nextIterator == null) {
            return false;
        }
        if (this.nextIterator.hasNext()) {
            return true;
        }
    }
}

```

```

    }
    this.nextIterator = this.iterators.pollFirst();
    return this.hasNext();
}

@Override
public E next() {
    if (this.nextIterator == null) {
        throw new NoSuchElementException();
    }
    if (this.nextIterator.hasNext()) {
        E res = this.nextIterator.next();
        this.previousIterator = this.nextIterator;
        return res;
    }
    this.nextIterator = this.iterators.pollFirst();
    return this.next();
}

@Override
public void remove() {
    if (this.previousIterator == null) {
        throw new IllegalStateException(
            "The next method has not been called since the last remove operation!"
        );
    }
    this.previousIterator.remove();
    this.previousIterator = null;
}

/**
 * Collects all iterators for the sets in the specified Trie in the corresponding order. Note that
 * the map is ordered.
 * @param node The root node of the Trie to collect the iterators from.
 */
private void collectIterators(TrieNode<E> node) {
    this.iterators.add(node.getElements().iterator());
    for (Map.Entry<String, TrieNode<E>> entry : node.getPrefixes().entrySet()) {
        this.collectIterators(entry.getValue());
    }
}
}
}

```

## Aufgabe 2 (Klassenhierarchien):

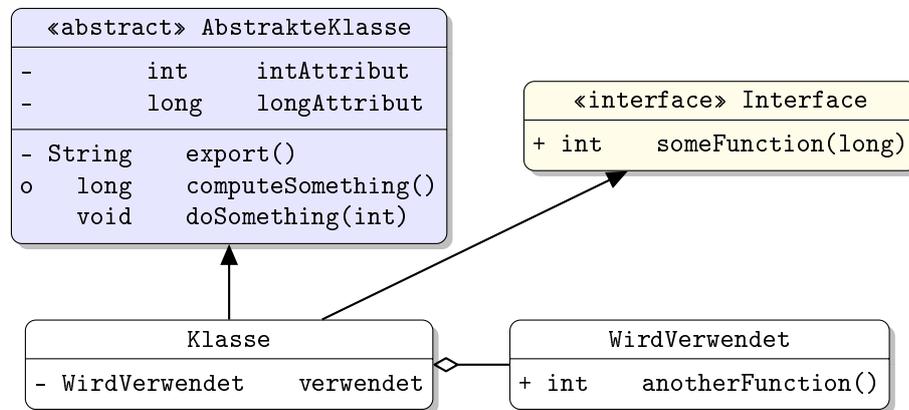
(9\* Punkte)

In dieser Aufgabe betrachten wir verschiedene Baumaschinen und deren Zusammenhänge und erstellen eine entsprechende Klassenhierarchie. Dabei sollen folgende Fakten beachtet werden:

- Die Motorleistung einer Baumaschine wird in PS gemessen.
- Bagger sind bestimmte Baumaschinen. Jeder Bagger hat eine Anzahl an Grabwerkzeugen (meist eins).
- Ein Schaufelradbagger ist ein Bagger, bei dem vor allem der Durchmesser des Schaufelrades (in Metern gemessen) von Bedeutung ist.
- Ein Seilbagger ist ein Bagger, dessen Grabwerkzeug an einer Seilwinde montiert ist. Hier ist die Seillänge von Interesse.
- Transportgeräte sind Baumaschinen, die dem Transport von Schüttgütern dienen.
- Förderbänder sind Transportgeräte, die sich durch die Breite des Förderbandes auszeichnen.
- Kipplaster sind Transportgeräte, bei denen der maximale Kippwinkel gespeichert werden soll.
- Bagger und Kipplaster sind Fahrzeuge.
- Zu jedem Fahrzeug gibt es eine Methode, die seine Höchstgeschwindigkeit in km/h berechnet.
- In unserer Modellierung gibt es keine weiteren Transportgeräte oder Baumaschinen, allerdings kann es weitere Bagger geben.

- a) Entwerfen Sie unter Berücksichtigung der Prinzipien der Datenkapselung eine geeignete Klassenhierarchie für Baumaschinen. Notieren Sie keine Konstruktoren, Getter und Setter. Sie müssen nicht markieren, ob Attribute `final` sein sollen. Achten Sie darauf, dass gemeinsame Merkmale in Oberklassen bzw. Interfaces zusammengefasst werden.

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  $B \rightarrow A$ , 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 Attributs oder in der Signatur einer Methode). Benutzen sie +, - und o um public, private und protected abzukürzen.

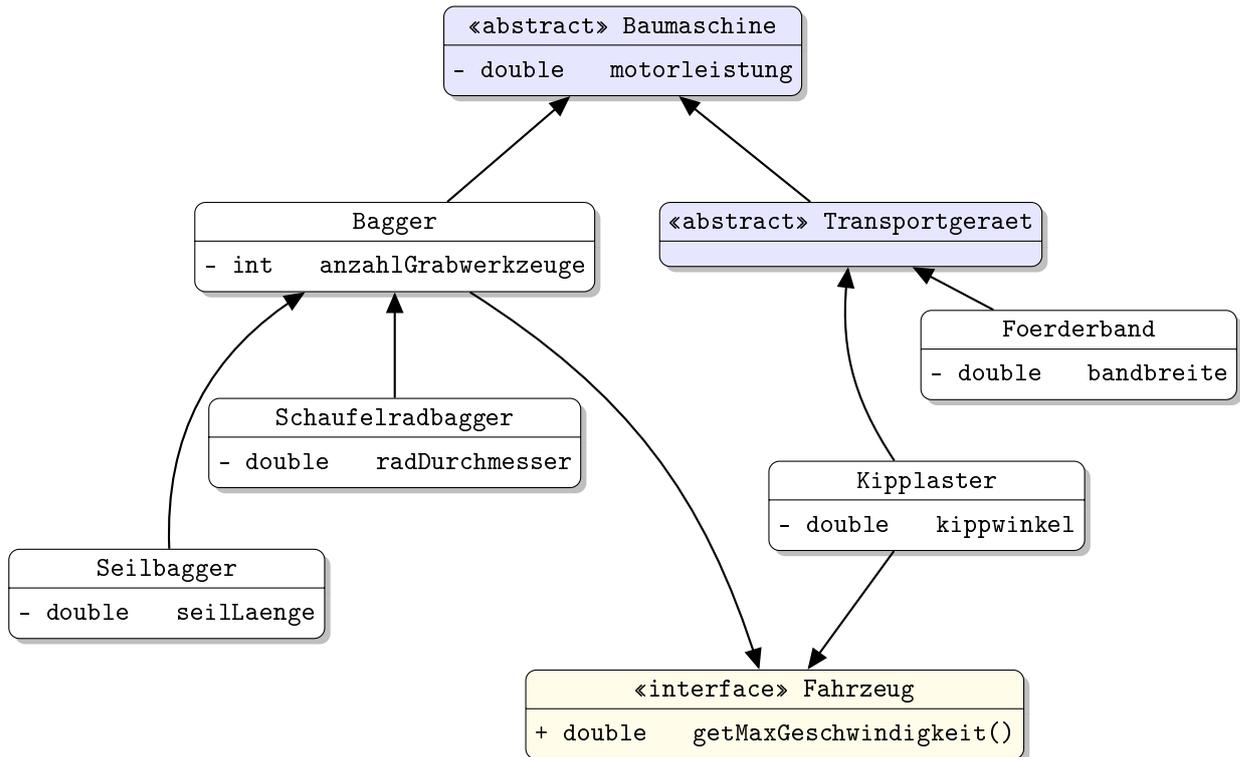
Tragen Sie keine vordefinierten Klassen (String, etc.) oder Pfeile dorthin in ihr Diagramm ein.

- b) Ein Bauherr möchte nun wissen, wie lange es dauert, alle Baumaschinen zu einer  $n$  Kilometer weit entfernten Baustelle zu fahren. Implementieren sie dazu die Methode `transportDauer`, die ein Array von Baumaschinen sowie eine Strecke in Kilometern übergeben bekommt. Falls eine dieser Baumaschinen kein Fahrzeug ist, soll die Methode `null` zurück geben. Anderenfalls gibt sie ein Objekt vom Typ `Integer` zurück, das die Anzahl an Minuten kapselt, die das langsamste Fahrzeug für die Strecke benötigt.

**Sie müssen die Funktion nicht kompilieren**, da es nicht nötig ist, die gesamte Klassenhierarchie zu implementieren. Schreiben Sie nur diese eine Funktion und kennzeichnen Sie die Funktion mit dem Schlüsselwort `static`, falls dies angebracht ist.

Lösung: \_\_\_\_\_

- a) Die Zusammenhänge können wie folgt modelliert werden:



b) Eine mögliche Implementierung:

```

public static Integer transportDauer(Baumaschine[] bms, int laenge) {
    double minGeschwindigkeit = Double.MAX_VALUE;
    for (Baumaschine b : bms) {
        if (b instanceof Fahrzeug) {
            Fahrzeug f = (Fahrzeug) b;
            minGeschwindigkeit = Math.min(minGeschwindigkeit,
                f.getMaxGeschwindigkeit());
        } else {
            return null;
        }
    }
    return laenge / minGeschwindigkeit;
}

```

### Aufgabe 3 (Ausnahmebehandlung):

(5\* Punkte)

a) Es soll eine Helferklasse `Eingabe` implementiert werden, mit der Eingaben vom Nutzer angefordert, überprüft und zurückgegeben werden können. Dazu soll exemplarisch die Methode `leseInt(String meldung, int min, int max)` implementiert werden. In dieser Methode soll zuerst die Eingabeaufforderung `meldung` ausgegeben werden. Dann soll ein `int`-Wert `x` vom Nutzer eingelesen werden, der dann darauf überprüft wird, ob er in den von `min` und `max` gegebenen Grenzen liegt, d.h., es soll  $\min \leq x \leq \max$  gelten. Ist dies nicht der Fall, soll eine passende Exception geworfen werden sollen.

Sie sollen dazu zwei Exception-Klassen `EingabeZuKleinException` und `EingabeZuGrossException` implementieren, die eine Methode `getFehlermeldung` zur Verfügung stellen. Diese soll einen `String` zurückgeben, der beschreibt, warum die Eingabe illegal ist. Verwenden Sie zur Implementierung eine gemeinsame Oberklasse `EingabeIllegalException`, die aber nie instanziiert werden soll. Achten Sie bei der Implementierung darauf, gemeinsame Merkmale in dieser Oberklasse zu deklarieren. Beachten Sie hierbei auch das Prinzip der Datenkapselung.

b) Schreiben Sie nun einen kleinen Rechentruiner, der

- 1) die Anzahl  $n$  von zu stellenden Aufgaben vom Nutzer einliest,
- 2)  $n$  zufällige Aufgaben zur Multiplikation von zwei natürlichen Zahlen unter 10 erzeugt, diese dem Nutzer ausgibt und eine Lösung abfragt,
- 3) die eingegebene Antwort mit der Lösung vergleicht und überprüft, ob die Antwort zu klein, zu groß oder richtig ist und dies in jeweils einem Zähler vermerkt,
- 4) zum Schluss ausgibt, wie häufig der Nutzer richtig lag und wie oft die Antwort zu hoch bzw. zu niedrig war.

Benutzen Sie für das Abfragen der Lösung **nur** die Klasse `Eingabe` aus Teil a) und benutzen Sie für den Punkt 3) **keine Vergleichsoperatoren** in Ihrem Rechentruiner.

Beachten Sie, dass einige Eingaben zu Ausnahmen führen können. Fangen Sie nur die ab, die Sie für die Lösung der beschriebenen Aufgaben benötigen.

Hinweise:

- `Math.random()` gibt einen zufälligen `double`-Wert zwischen 0 und 1 zurück.

Lösung: \_\_\_\_\_

```
a) public class Eingabe {
    public static int leseInt(String meldung, int min, int max)
        throws EingabeIllegalException {
        System.out.print(meldung);
        int e = Integer.parseInt(System.console().readLine());
        if (e < min) {
            throw new EingabeZuKleinException(min, e);
        } else if (e > max) {
            throw new EingabeZuGrossException(max, e);
        }
        return e;
    }
}

public abstract class EingabeIllegalException extends Exception {
    protected final int grenze;
    protected final int wert;

    public EingabeIllegalException(int g, int w) {
        this.grenze = g;
        this.wert = w;
    }

    public abstract String getFehlermeldung();
}

public class EingabeZuGrossException extends EingabeIllegalException {
    public EingabeZuGrossException(int g, int w) {
        super(g, w);
    }

    public String getFehlermeldung() {
        return wert + " ist groesser als die obere Grenze " + grenze;
    }
}
```

```
public class EingabeZuKleinException extends EingabeIllegalException {
    public EingabeZuKleinException(int g, int w) {
        super(g, w);
    }

    public String getFehlermeldung() {
        return wert + " ist kleiner als die untere Grenze " + grenze;
    }
}
```

```
b) public class MultTest {
    public static void main(String[] a) throws EingabeIllegalException {
        System.out.print("Wieviele Aufgaben wollen Sie rechnen? ");
        int aufgabenZahl = Integer.parseInt(System.console().readLine());

        int zuNiedrig = 0;
        int zuHoch = 0;
        for (int i = 0; i < aufgabenZahl; i++) {
            int x = (int) (Math.random() * 10);
            int y = (int) (Math.random() * 10);
            int lsg = x * y;

            try {
                int nutzerLsg =
                    Eingabe.leseInt("Was ist " + x + "*" + y + "? ", lsg, lsg);
            } catch (EingabeZuKleinException e) {
                zuNiedrig++;
            } catch (EingabeZuGrossException e) {
                zuHoch++;
            }
        }

        System.out.println("Sie haben " + zuNiedrig
            + " mal ein zu kleines Ergebnis angegeben.");
        System.out.println("Sie haben " + zuHoch
            + " mal ein zu hohes Ergebnis angegeben.");
        System.out.println("Sie lagen "
            + (aufgabenZahl - zuNiedrig - zuHoch) + " mal richtig.");
    }
}
```

#### Aufgabe 4 (Game of Life):

(5\* + 0 + 0 Punkte)

In dieser Aufgabe sollen Sie ein Programm erweitern und parallelisieren, das Conways "Game Of Life" ([http://de.wikipedia.org/wiki/Conways\\_Spiel\\_des\\_Lebens](http://de.wikipedia.org/wiki/Conways_Spiel_des_Lebens)) simuliert und als Pixelgrafik ausgibt. Das "Game Of Life" ist eine Simulation, die auf einem 2-dimensionalem Spielfeld in diskreten Zeitschritten durchgeführt wird. Jede Zelle kann zu jedem Zeitpunkt  $t$  entweder lebendig oder tot sein und der Zustand im nächsten Zeitschritt  $t + 1$  bestimmt sich durch die Anzahl lebender Nachbarn zum Zeitpunkt  $t$ . Lebende Zellen überleben, falls sie entweder 2 oder 3 lebende Nachbarn haben. Tote Zellen werden "belebt", wenn sie genau 3 lebende Nachbarn haben. Unter allen anderen Bedingungen ist die Zelle im Zeitpunkt  $t + 1$  tot. Wir werden im Gegensatz zu der Originalversion nur ein beschränktes Spielfeld betrachten und durch ein 2-dimensionales Array von `byte` Werten repräsentieren (wobei 1 eine lebende und 0 eine tote Zelle darstellt).

a) Implementieren Sie die Methode `iterate` der Klasse `GameOfLifeCPU`. Es soll in einem Zeitschritt über

das Spielfeld iteriert und für jede Zelle ein neuer Zustand berechnet werden. Verwenden Sie zwei verschachtelte Schleifen, um alle Felder in der Nachbarschaft zu besuchen.

Zur Lösung dieser Aufgabe sind die Methoden: `int getNumberOfActiveNeighbors(int row, int column)`, `boolean isActive(row, column)`, `void setActive(int row, int column, boolean active)` und `void swapCurrentAndTemp()` hilfreich.

- b) Falls Sie am Programmierwettbewerb teilnehmen möchten, erstellen Sie eine Variante ihrer Lösung zur effektiven Nutzung eines Multi-core Prozessors für die parallele Berechnung der Simulation. Ziel ist eine Verringerung der Laufzeit. Nutzen Sie dazu Java Threads und implementieren Sie eine neue Klasse `GameOfLifeCPUThreaded` die von `GameOfLifeCPU` ableitet werden kann. Zusätzliche Informationen zur Nutzung von Java Threads finden Sie zum Beispiel unter: <https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html>.
- c) Falls Sie am Programmierwettbewerb teilnehmen möchten, erstellen Sie eine Variante ihrer Lösung zur effektiven Nutzung einer Grafikkarte für die parallele Berechnung der Simulation. Nutzen Sie dazu JCuda und implementieren Sie eine neue Klasse `GameOfLifeJCuda` die von `GameOfLife` abgeleitet werden kann. Zusätzliche Informationen zur Nutzung von JCuda finden Sie unter: <http://www.jcuda.org/documentation/Documentation.html>.

**Hinweise:**

- Wenn Sie Unterpunkt b) oder c) bearbeitet haben und an der Umfrage unter <https://app.lamapoll.de/ProductivityHPCStudents/> teilgenommen haben, werden Sie bei der Vergabe unseres Lehrpreises zur Parallelen Programmierung berücksichtigt.

Lösung: \_\_\_\_\_

```
public class GameOfLifeCPU extends GameOfLife {

    protected byte[] temp;

    public GameOfLifeCPU() {
        temp = createArray();
    }

    private void setActive(int row, int column, boolean active) {
        temp[getIndex(row, column)] = (byte) (active ? 1 : 0);
    }

    protected int getNumberOfActiveNeighbors(int row, int column) {
        int[] rows = new int[] { (row == 0 ? HEIGHT - 1 : row - 1), row,
            (row == HEIGHT - 1 ? 0 : row + 1) };
        int[] columns = new int[] { (column == 0 ? WIDTH - 1 : column - 1),
            column, (column == WIDTH - 1 ? 0 : column + 1) };

        int alive = 0;
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                if (i == 1 && j == 1) {
                    continue;
                }

                if (isActive(rows[i], columns[j])) {
                    alive++;
                }
            }
        }
    }
}
```

```
    }

    return alive;
}

protected void swapCurrentAndTemp() {
    byte[] newCurrent = temp;

    // re-use for next iteration
    temp = current;
    current = newCurrent;
}

@Override
public void iterate() {
    for (int row = 0; row < HEIGHT; row++) {
        for (int column = 0; column < WIDTH; column++) {
            boolean cellActive = isActive(row, column);
            int activeNeighbors = getNumberOfActiveNeighbors(row, column);

            boolean newActive = (activeNeighbors == 3
                || (activeNeighbors == 2 && cellActive));
            setActive(row, column, newActive);
        }
    }

    swapCurrentAndTemp();
}
}
```