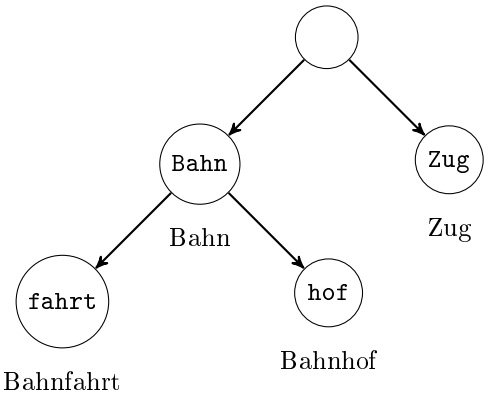
Prof. Dr. M. Müller C. Aschermann, J. Hensel, J. Protze, P. Reble

Allgemeine Hinweise:

- Die **Hausaufgaben** sollen in Gruppen von je **3 Studierenden** aus der **gleichen Kleingruppenübung (Tutorium)** bearbeitet werden. **Namen und Matrikelnummern** der Studierenden sind auf jedes Blatt der Abgabe zu schreiben. **Heften bzw. tackern Sie die Blätter!**
- Die **Nummer der Übungsgruppe** muss **links oben** auf das **erste Blatt** der Abgabe geschrieben werden. Notieren Sie die Gruppennummer gut sichtbar, damit wir besser sortieren können.
- Die Lösungen müssen **bis Montag, den 18.01.2016 um 15:00 Uhr** in den entsprechenden Übungskasten eingeworfen werden. Sie finden die Kästen am Eingang Halifaxstr. des Informatikzentrums (Ahornstr. 55). Alternativ können Sie die Lösungen auch vor der Abgabefrist direkt bei Ihrer Tutorin/Ihrem Tutor abgeben.
- In einigen Aufgaben müssen Sie in `Java` programmieren und `.java`-Dateien anlegen. **Drucken** Sie diese aus **und** schicken Sie sie per **E-Mail** vor Montag, dem 18.01.2016 um 15:00 Uhr an Ihre Tutorin/Ihren Tutor.
  Stellen Sie sicher, dass Ihr Programm von `javac` **akzeptiert** wird, ansonsten werden keine Punkte vergeben.
- Aufgaben, die mit einem * markiert sind, sind Bonusaufgaben und tragen nicht zur Summe der erreichbaren Punkte bei, die für die Klausurzulassung relevant ist, jedoch werden Ihnen die in solchen Aufgaben erreichten Punkte ganz normal gutgeschrieben. **Das bedeutet allerdings nicht, dass diese Aufgaben nicht klausurrelevant sind.**

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-`String`s, anhand derer die Suche durch den Baum nach einem Element gesteuert wird. Die Konkatenation der Teil-`String`s 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 Iterators 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-`String`s aber ein gemeinsames nicht-leeres Präfix haben. Sie finden nützliche Methoden zur Verarbeitung von `String`s in der zugehörigen `Java` API.

Aufgabe 2 (Klassenhierarchien): (9* Punkte)

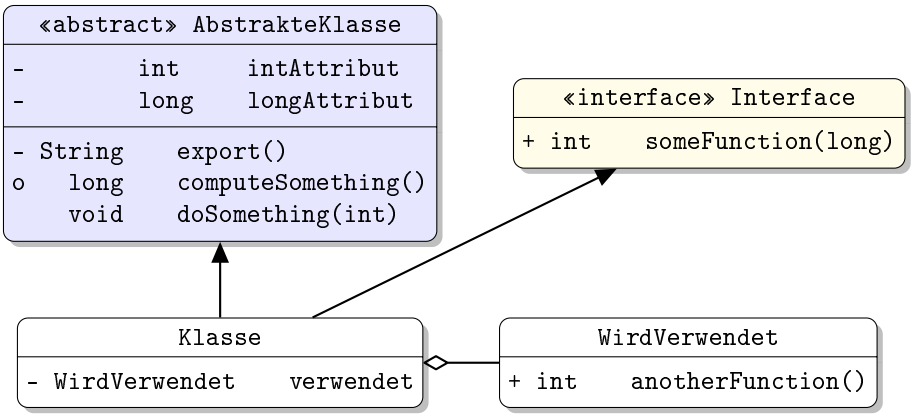
In dieser Aufgabe betrachten wir verschiedene Baumaschinen und deren Zusammenhänge und erstellen eine entsprechende Klassenhierarchie. Dabei sollen folgende Fakten beachtet werden:

- 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 $B \blacktriangleright A$, 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.

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 $\texttt{min} \leq x \leq \texttt{max}$ 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` zurückgegeben, 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 Rechentrainer, 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 Rechentrainer.

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.

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`) 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 Parallele Version ihrer Lösung zur effektiven Nutzung eines Multi-core Prozessors für die Berechnung der Simulation. Nutzen Sie dazu Java Threads und implementieren Sie eine neue Klasse `GameOfLifeCPUThreaded` die von `GameOfLifeCPU` ableitet werden kann. Zusätzliche Informationen 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 Parallele Version ihrer Lösung zur effektiven Nutzung einer Grafikkarte für die 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 Preisverleihung in 2016 des Lehrpreises zur Parallelen Programmierung berücksichtigt.