

252-0027-00: Einführung in die Programmierung

Übungsblatt 12

Abgabe: 19. Dezember 2023, 23:59

Checken Sie die neue Übungs-Vorlage aus. Importieren Sie beide Eclipse-Projekte (das Projekt für den Bonus und das Projekt für die restlichen Aufgaben). Vergessen Sie nicht, Tests zu schreiben!

Aufgabe 1: Hoare Triple

Welche dieser Hoare Tripel sind (un)gültig? Bitte geben Sie für ungültige Tripel ein Gegenbeispiel an. Die Anweisungen sind Teil einer Java Methode. Alle Variablen sind vom Typ `int` und es gibt keinen Overflow.

1. `{ x >= 0 || y >= 0 } z = x * y; { z > 0 }`
2. `{ x <= 0 && y >= 0 && x = y } z = x * y; { z <= 0 }`
3. `{ x > 10 } z = x % 10; { z > 0 }`
4. `{ x > 0 } y = x * x; z = y / 2; { z > 0 }`
5. `{ true }`

```
if (x > y) {  
    y = x;  
} else {  
    y = - x;  
}
```

```
{ y >= x }
```

6. `{ b > c }`

```
if (x > b) {  
    a = x;  
} else {  
    a = b;  
}
```

```
{ a > c }
```

Aufgabe 2: Generische Listen

In dieser Aufgabe implementieren Sie eine generische verkettete Liste. Anhang B zeigt eine generische Version eines Interfaces für Listen. Vervollständigen Sie die Klasse `MyListImpl`, sodass die Klasse das `MyList` Interface implementiert. Dem Interface wurden zwei neue Methoden hinzugefügt. Die Methode `MyListNode getNode(int index)` gibt den Knoten zurück, welcher den Wert der Liste an Position `index` speichert. `MyListNode` ist selber ein Interface (siehe Anhang C) mit Methoden, welche jeweils den gespeicherten Wert des Knoten, den nächsten Knoten, und ob es einen nächsten Knoten gibt zurückgeben. Damit überprüfen wir, dass `MyListImpl` eine verkettete Liste implementiert. Die Methode `Iterator<T> iterator()` gibt einen Iterator für die Datenstruktur zurück. Implementieren Sie einen neuen Iterator, das heisst eine Klasse, welche das `Iterator` Interface implementiert, und geben Sie nicht den Iterator einer anderen Datenstruktur zurück (zum Beispiel den Iterator einer `ArrayList`). Dies können wir in den Tests der Korrektur testen. Die `void remove()` Methode vom Iterator wird nicht benötigt. Die Methode `void addAll(MyList<T> other)` sollte eine konstante Laufzeit haben. Ein paar Tests finden Sie in `MyListTest`.

Aufgabe 3: Notenauswertung

Die Klasse `Service` stellt verschiedene Analysen für Prüfungsergebnisse von S Studierenden zur Verfügung. Die Liste von Ergebnissen besteht aus S Einträgen, also jeweils ein Eintrag pro Student/in. Jeder Eintrag besteht aus einer Zeile und enthält (in dieser Reihenfolge):

1. die Immatrikulationsnummer des Studierenden (ein identifizierender positiver `int`-Wert)
2. drei Noten (drei reelle Zahlen im Bereich von 1.0 bis 6.0, getrennt durch Leerzeichen)

Die drei Noten gehören zu den Fächern *Fach 1*, *Fach 2* und *Fach 3*. Zusätzliche Leerzeilen und -zeichen sollen ignoriert werden. Eine Beispiel für eine Liste für 3 Studierende ist:

```
111111004  5.0  5.0  6.0
111111005  3.75 3.0  4.0
111111006  4.5  2.25 4.0
```

Ihre Aufgabe ist es nun, die `Service`-Klasse und ihre Analysen zu implementieren. Die `Service`-Klasse hat einen Konstruktor, welcher alle Prüfungsergebnisse aus einem Scanner auslesen und damit das `Service`-Objekt initialisieren soll. Das Objekt soll so initialisiert werden, dass die vorgegebenen Methoden ihre Analysen durchführen können. Sie dürfen dabei Attribute und zusätzliche Methoden frei bestimmen.

- a) Implementieren Sie nun die Methode `critical()`, welche die zwei Argumente `bound1` und `bound2` erwartet. Die Methode sucht alle "kritischen" Fälle und gibt eine Liste dieser Studierenden zurück. Ein Student darf maximal einmal in der Liste vorkommen. Die zurückgegebene Liste besteht aus den Immatrikulationsnummern dieser Studierenden (in beliebiger Reihenfolge).

Ein/e Student/in gilt als kritisch, wenn die Note in *Fach 1* \leq `bound1` und die Summe der Noten für *Fach 2* und *Fach 3* kleiner als `bound2` ist.

Für das obige Beispiel gäbe `critical(4, 8)` eine Liste mit dem Element `111111005` zurück.

- b) Implementieren Sie nun die Methode `top()`, welche die Studierenden mit den besten Ergebnissen zurückgeben soll. Der Parameter `limit` bestimmt die maximale Anzahl der zurückzugebenden Studierenden. Falls weniger Ergebnisse als `limit` existieren, sollen einfach alle gefundenen zurückgegeben werden.

Der Rückgabewert der Methode ist wieder eine Liste der Immatrikulationsnummern. Ein Student darf maximal einmal in der Liste vorkommen. Diese Liste soll absteigend nach der Leistung sortiert sein (der/die Student/in mit dem besten Ergebnis zuerst). Dabei gilt, dass ein Ergebnis *A* besser ist als ein Ergebnis *B*, wenn die Summe aller Noten von *A* grösser ist als die Summe der Noten von *B*. Sind die Summen gleich, sind die Ergebnisse gleich gut (und die Reihenfolge in der Liste somit egal).

Für das obige Beispiel gäbe `top(2)` entweder die Liste `[111111004, 111111006]` oder die Liste `[111111004, 111111005]` zurück (beide wären richtig).

In der Klasse `ServiceTest` finden Sie einen ersten kleinen JUnit-Test als Starthilfe. Ausserdem dürfen Sie folgende Annahmen machen: Der Parameter `limit` ist immer grösser als 0 und die beiden Parameter für `critical()` sind immer im Bereich von 0.0 bis 100.0.

Tipp: Verwenden Sie die `Collections.sort(...)` Funktion einer Kollektion, welche mit `import java.util.Collections;` importiert werden kann. Beachten Sie, dass dafür die Klasse, welche Sie für die Elemente der Kollektion verwenden, das Interface `Comparable<T>` (T sollte die Klasse selber sein), und damit auch eine Funktion `compareTo` implementieren muss. Diese Funktion nimmt eine Instanz der selben Klasse und gibt 0 zurück, wenn `this` und das Argument gleich sind, gibt 1 zurück, wenn `this` grösser als das Argument ist, und gibt -1 zurück, wenn `this` kleiner als das Argument ist.

Aufgabe 4: Interpreter

In der letzten Übung implementierten Sie einen Evaluator für mathematische Ausdrücke. In dieser Aufgabe erweitern Sie ihn so, dass er statt einzelnen Ausdrücken einfache Programme mit mehreren Anweisungen auswertet. Das nennt man auch *interpretieren* und ein solches Programm entsprechend *Interpreter*.

Abbildung 1 zeigt die EBNF-Beschreibung für Programme (*prog*). Es gibt nur eine Art von Anweisung (*stmt*), nämlich eine Zuweisung eines Ausdrucks zu einer Variable. Beachten Sie, dass die Beschreibung zugunsten der Lesbarkeit nicht alle Tokenizer-Regeln explizit enthält: die RHS der Parser-Regeln enthalten teilweise auch direkt Buchstaben (blau). Der aktualisierte Tokenizer in der Übungsvorlage stellt aber auch für die Buchstaben `() = ;` die benötigten Methoden zur Verfügung.

In der Vorlage befindet sich die `Interpreter`-Klasse, welche (abgesehen von Klassen- und Methodennamen) dem fertigen `ExprEvaluator` der letzten Übung entspricht. Die `Interpreter`-Klasse befindet sich in einem *Paket* (engl. package) namens `language`. Platzieren Sie alle Klassen, welche Sie für diese Aufgabe erstellen, ebenfalls in diesem Paket.

- a) Ändern Sie den Interpreter so, dass er Programme, die der Beschreibung in Abbildung 1 entsprechen, interpretiert. Die Semantik von Zuweisungen soll dieselbe sein wie in Java.

```

digit  ⇐ 0 | 1 | ... | 9
char   ⇐ A | B | ... | Z | a | b | ... | z
num    ⇐ digit { digit } [ . digit { digit } ]
var     ⇐ char { char }
func    ⇐ char { char } (
op      ⇐ + | - | * | / | ^
atom    ⇐ num | var
term    ⇐ ( expr ) | func expr ) | atom
expr    ⇐ term [ op term ]
stmt    ⇐ var = expr ;
prog    ⇐ { stmt }

```

Abbildung 1: EBNF-Beschreibung von *prog*

```

alpha = i * ((2*PI) * (1 / 6.05));
size = (0.25 * cos(t/2)) + 0.75;

x = cos(alpha + (0.3 * t)) * size;
y = sin((1.5 * alpha) + t) * size;

r = (cos(alpha + (2 * t)) + 1) / 2;
g = (sin(alpha + (2 * t)) + 1) / 2;
b = (cos(alpha + (PI/2)) + 1) / 2;

```

Abbildung 2: Beispiel-Programm

Aufgrund der Möglichkeit, Variablen zu definieren, reicht die einfache rekursive Struktur des ExprEvaluator nicht aus, um Programme zu interpretieren. Der Interpreter muss sich zusätzlich den Zustand des Programms, d.h. die definierten Variablen und deren Werte, merken. Dafür eignet sich eine Map, oder genauer, da wir eine Abbildung von Variablen-Namen auf Werte brauchen, eine `Map<String,Double>`. Erstellen Sie eine solche im Interpreter-Konstruktor und verwenden Sie sie wo nötig. Neu könnte der Konstruktor auch eine Map von Variablennamen und -werten entgegennehmen anstatt dem einzelnen Wert für die *x*-Variable.

Schreiben Sie die fehlenden `interpret*(...)`-Methoden und ändern Sie `interpret(String)` entsprechend. Beachten Sie, dass Anweisungen (*stmts*) im Gegensatz zu Ausdrücken (*exprs*) selbst keinen Wert haben, sondern nur einen Effekt auf den Programmzustand (z.B. die Änderung eines Wertes einer Variable). Ein Programm als Ganzes hat ebenfalls keinen Wert, also können Sie den Rückgabetypp von `interpret(String)` auf `void` ändern.

- b) Schreiben Sie ein Java-Programm `Repl` (im Paket `language`), welches eine *REPL* implementiert. "REPL" steht für *read-eval-print loop* und bezeichnet ein (Java-)Programm, welches wiederholt Anweisungen von der Konsole liest (*read*), diese auswertet (*eval*) und das Resultat ausgibt (*print*). Der Programmzustand (d.h. die Werte von Variablen) wird über mehrere Anweisungen hinweg mitgeführt und durch das Interpretieren von Anweisungen verändert.

Sie können sich an der `EvaluatorApp` der letzten Übung orientieren. Da Programme selber keinen Wert haben, gibt es kein explizites Resultat zum Ausgeben. Geben Sie stattdessen nach jeder Ausführung alle definierten Variablen und deren Werte aus. Dafür brauchen Sie Zugriff auf die Variablen-Map des Interpreters. Achten Sie darauf, dass die `ProgramException` richtig behandelt wird.

Aufgabe 5: Programmatisches Zeichnen

Mit Ihrem Interpreter können Sie noch interessantere Dinge zeichnen als die einfachen $f(x)$ -Funktionen der letzten Übung. In der Vorlage finden Sie eine `DrawingApp`, welche ein GUI mit einem Eingabefeld für ein Programm und einem für eine Anzahl Repetitionen erstellt. Die Idee ist, dass das

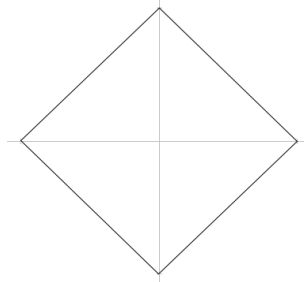
Programm wiederholt ausgeführt wird und dass dieses in jeder Iteration ein (x, y) -Koordinaten-Paar und zusätzlich ein (r, g, b) -Farbtripel liefert (r, g und b sind dabei jeweils zwischen 0.0 und 1.0). Nach jeder Repetition¹ wird eine Linie zwischen dem alten und dem neuen Koordinaten-Paar mit der neuen Farbe gezeichnet. Als Input, d.h. als Start-Programmzustand, bekommt das Programm Werte für folgende Variablen:

n die Anzahl Repetitionen
 i die aktuelle Iteration ($0 \leq i \leq n$)
 x, y, r, g, b die Resultate der letzten Ausführung (in der ersten Iteration: 0.0)
 PI, E, \dots mathematische Konstanten

Mit diesem Mechanismus lassen sich verschiedenste Formen zeichnen. Ein Programm, das ein gleichmässiges n -Eck zeichnet, sieht z.B. so aus:

```
alpha = i * ((2*PI) / n);
x = cos(alpha);
y = sin(alpha);
```

Für $n = 4$ liefert dieses Programm die Koordinaten-Paare $(1, 0)$, $(0, 1)$, $(-1, 0)$, $(0, -1)$, welche zu folgendem Quadrat verbunden werden:



Im Anhang finden Sie weitere Programme und die daraus resultierenden Zeichnungen.

- Vervollständigen Sie die `run()`-Methode in `DrawingApp`. Es ist bereits Code vorhanden, der Arrays für die x - und y -Koordinaten und deren Farben erstellt. Die `drawLines()`-Methode nimmt diese Arrays entgegen und zeichnet sie. Was noch fehlt, ist der Code, der diese Werte berechnet. Erstellen Sie einen Interpreter, initialisieren Sie die oben aufgeführten Variablen, führen Sie das im Feld `program` gespeicherte Programm wiederholt aus und lesen Sie nach jeder Iteration die Variablen-Werte aus. Jede Ausführung übernimmt dabei den Programmzustand der vorhergehenden Ausführung. Einzige Ausnahme: Die Variable i müssen Sie vor jeder Ausführung aktualisieren!
- Die Schleife der Methode `run()` zeichnet das Bild immer wieder neu. Das erlaubt uns, animierte Zeichnungen zu machen! Dazu ist nur eine zusätzliche Variable t nötig, welche vor jeder Ausführung von `program` mit der aktuellen Zeit initialisiert wird².

Wir definieren die "aktuelle Zeit" als die Anzahl Sekunden, die seit der letzten Änderung des Programms verstrichen sind. Diese erhalten Sie, indem Sie in `setProgram()` mit `System.currentTimeMillis()` den Zeitpunkt "0" speichern und in `run()` die Differenz zur aktuellen Zeit berechnen. Beachten Sie, dass t `double`-Genauigkeit haben soll.

¹D.h. nach jeder Iteration ausser der ersten – die Anzahl Iterationen ist also um 1 grösser als die Anzahl Repetitionen!

²Sie können den selben Wert für t für alle Iterationen verwenden.

Um eine Zeichnung zu animieren, verwenden Sie t an einem geeigneten Ort, z.B. so:

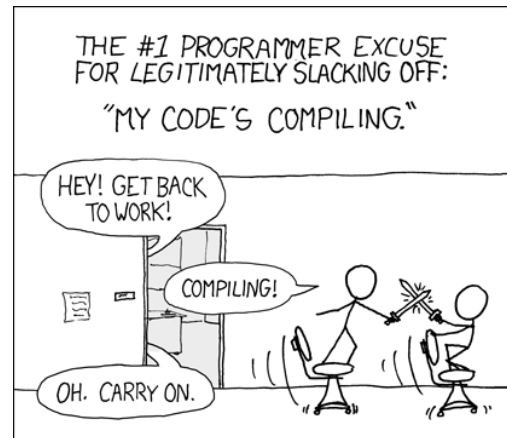
```
alpha = i * ((2*PI) / n);  
x = cos(alpha + t);  
y = sin(alpha + t);
```

Geben Sie auch mal das Beispiel-Programm in Abbildung 2 ein und setzen Sie die Anzahl der Repetitionen auf 121, um zu sehen, welche interessanten Animationen möglich sind.

Aufgabe 6: Compiler

Wie Sie beim Herumspielen mit der DrawingApp vielleicht festgestellt haben, kommt das Programm bei vielen Repetitionen an seine Grenzen. Im Panel unten rechts werden die *FPS* (frames per second) angezeigt. Dies sind die Anzahl Bilder, die das Programm pro Sekunde zeichnen kann. Wenn dieser Wert zu klein wird, sieht die Animation nicht mehr flüssig aus.

Das Problem ist, dass das Interpretieren von Quellcode ineffizient und langsam ist. Deshalb werden Java-Programme auch zuerst *kompiliert*, bevor sie ausgeführt werden. Kompilieren heisst, den Quellcode in eine Form zu übersetzen, die vom Computer direkt(er) ausgeführt werden kann. In dieser Übung schreiben Sie einen einfachen Compiler, der den Quellcode von Programmen von Aufgabe 4 in eine Serie von Instruktionen übersetzt, die effizient ausgeführt werden können.



xkcd: Compiling by Randall Munroe (CC BY-NC 2.5)

Die Programmiersprache in Aufgabe 4 hat eine rekursive Struktur: Ausdrücke können mehrere Ausdrücke enthalten, welche wiederum mehrere Ausdrücke enthalten können, usw. Um eine solche Struktur in eine lineare Folge von Instruktionen umzuwandeln, verwenden wir einen *Operanden-Stack*. Dies ist ein Stack (wie Sie ihn in der Vorlesung gesehen haben), der Zwischenresultate von Berechnungen speichert. Instruktionen können Werte auf den Stack “pushen” oder Werte ab dem Stack “poppen” und verwenden. Es gibt folgende Arten von Instruktionen:

- CONST** c Pusht den konstanten Wert c auf den Stack
- LOAD** v Lädt den Wert der Variable v und pusht ihn auf den Stack
- STORE** v Poppt einen Wert vom Stack und speichert ihn in der Variable v
- OP** \oplus Poppt zwei Werte r und l vom Stack (zuerst r , dann l) berechnet $l \oplus r$ und pusht das Resultat zurück auf den Stack
- FUNC** f Poppt einen Wert x vom Stack, berechnet $f(x)$ und pusht das Resultat zurück auf den Stack

Unten sehen Sie ein kleines Programm, das aus solchen Instruktionen besteht. Es lädt zuerst den Wert der Variable x und dann einen konstanten Wert 2 auf den Stack. Die nächste Instruktion holt sich die beiden Werte vom Stack, multipliziert sie und pusht das Resultat zurück. Die letzte Instruktion schliesslich holt diesen Wert vom Stack und speichert ihn zurück in die Variable x :

```
LOAD x
CONST 2
OP *
STORE x
```

Sie sollen nun einen Compiler schreiben, welcher ein Programm in eine Liste solcher Instruktionen kompiliert. Der Compiler geht grundsätzlich gleich vor wie der Interpreter: er parst das Programm rekursiv und berechnet gleichzeitig ein Resultat. Im Gegensatz zum Interpreter berechnet er aber keine Werte, sondern generiert Listen von Instruktionen.

Um zu verstehen, wie diese Instruktionen genau generiert werden, betrachten Sie Tabelle 1,

Programmteil	Instruktionen
b	LOAD b
1	CONST 1
b + 1	LOAD b CONST 1 OP +
(b + 1)	LOAD b CONST 1 OP +
2	CONST 2
c	LOAD c
2 * c	CONST 2 LOAD c OP *
sin(2 * c)	CONST 2 LOAD c OP * FUNC sin
(b + 1) / sin(2 * c)	LOAD b CONST 1 OP + CONST 2 LOAD c OP * FUNC sin OP /
a = (b + 1) / sin(2 * c);	LOAD b CONST 1 OP + CONST 2 LOAD c OP * FUNC sin OP / STORE a

Tabelle 1: Kompilieren der Anweisung $a = (b + 1) / \sin(2 * c);$

welche zeigt, wie der Ausdrucks $a = (b + 1) / \sin(2 * c);$ kompiliert wird. In der linken Spalte stehen die Programmteile (in der Reihenfolge, in der sie auch schon vom Interpreter fertig geparkt werden) und in der rechten Spalte die entsprechenden Instruktionen. Zum Beispiel sehen Sie, dass der Compiler für die Zahl 1 im Quellcode die Instruktion **CONST 1** generiert. Oder, dass er für einen binären Ausdruck zuerst die Instruktionen des linken Teils, dann die Instruktionen des rechten Teils und schliesslich eine **OP**-Instruktion generiert.

- a) Erstellen Sie zuerst einige Klassen `ConstInstr`, `LoadInstr`, usw. für die verschiedenen Arten von Instruktionen. Erstellen Sie dazu am besten ein Unterpaket `language.instructions` und deklarieren Sie die `*Instr`-Klassen als `public`. Für die **OP**- und **FUNC**-Instruktionen können Sie entscheiden, ob Sie jeweils eine einzige Klasse verwenden möchten oder mehrere Subklassen, eine für jeden unterstützten Operator, bzw. für jede Funktion.

Alle Instruktionen sollen ein Interface `Instr` (ebenfalls im `language.instructions`-Paket zu erstellen) mit einer `execute()`-Methode implementieren. Diese Methode nimmt als Argumente den Operanden-Stack und die Variablen-Map und führt die Instruktion aus. Falls ein Fehler auftritt (z.B., wenn eine Variable nicht definiert ist oder wenn der Stack nicht die erwartete Grösse hat), soll die Methode eine Ausnahme der Klasse `ExecutionException` (die Sie erstellen) werfen. Diese Art von Exception soll *checked* sein.

- b) Erstellen Sie eine `Program`-Klasse, welche eine Liste von Instruktionen enthält und eine `execute()`-Methode hat, welche diese Instruktionen ausführt. Diese Methode nimmt eine Variablen-Map entgegen, die vom Programm verwendet und aktualisiert wird.

Das Ausführen der Instruktionen ist denkbar einfach: Erst wird ein Operanden-Stack erstellt, und dann wird eine Instruktion nach der anderen über ihre `execute()`-Methode ausgeführt.

- c) Schreiben Sie jetzt die `Compiler`-Klasse. Sie können sie analog zur `Interpreter`-Klasse entwerfen, mit `compile*()`- statt `interpret*()`-Methoden. Jede dieser Methoden soll eine Liste zurückgeben, welche die Instruktionen enthält, die dem geparkten Programm-Teil entsprechen. Die Haupt-Methode `compile(String)`, schliesslich, soll alle Instruktionen in eine `Program`-Instanz packen und diese zurückgeben.
- d) Ändern Sie zum Schluss die `DrawingApp` so, dass sie statt dem Interpreter den Compiler verwendet, um die eingegebenen Programme auszuführen. Sinnvollerweise sollten Sie das Programm in `setProgram()` kompilieren, und zwar nur wenn es sich verändert hat. In `run()`

wird das Programm dann “nur” noch ausgeführt. Sie sollten aber weiterhin Fehler beim Kompilieren und beim Ausführen abfangen und die Fehlermeldung auf dem Panel ausgeben. Wenn Sie alles richtig gemacht haben, sollten Sie jetzt Zeichnungen mit einer deutlich höheren Anzahl an Repetitionen flüssig animieren können!

Aufgabe 7: Datenbanken (Bonus!)

Achtung: Diese Aufgabe gibt Bonuspunkte (siehe “Leistungskontrolle” im www.vvz.ethz.ch). Die Aufgabe muss eigenhändig und alleine gelöst werden. Die Abgabe erfolgt wie gewohnt per Push in Ihr Git-Repository auf dem ETH-Server. Verbindlich ist der letzte Push vor dem Abgabetermin. Auch wenn Sie vor der Deadline committen, aber nach der Deadline pushen, gilt dies als eine zu späte Abgabe. Bitte lesen Sie zusätzlich [die allgemeinen Regeln](#).

Diese Aufgabe basiert auf dem Bonus von Aufgabenblatt 10 mit einer anderen Unteraufgabe. Änderungen sind in bold markiert. In dieser Aufgabe implementieren Sie für eine Datenbank von Personengesundheitsdaten **das Erheben von Statistiken (Task a)**.

Die Datenbank selber ist bereits mit der Klasse `Database` implementiert. Die Datenbank hält eine Liste von Einträgen, welche durch die Klasse `Item` repräsentiert werden. Die folgenden 4 Paragraphen erklären alle in der Vorlage gegebenen Klassen im Detail.

Item Die Klasse `Item` repräsentiert einen Datenbankeintrag mit 4 Attributen: eine ID (int), ein Alter (int), einen Gesundheitswert (int), und ein Sicherheitslevel, welches durch die Klasse `Level` repräsentiert wird. Alter und Gesundheitswert sind immer ≥ 0 . Die Methoden `Item.getID()`, `Item.getAge()`, `Item.getHealth()`, `Item.getLevel()` geben jeweils die ID, das Alter, den Gesundheitswert, und das Sicherheitslevel eines Eintrags zurück. Die Methode `Item.setHealth(int newHealth)` setzt den Gesundheitswert auf `newHealth`. Die anderen Attribute können nicht geändert werden.

Level Die Klasse `Level` repräsentiert ein Sicherheitslevel. Ein Sicherheitslevel wird über ein **Set** von Integern definiert, welches in einem Attribut der Klasse `Level` gespeichert wird und von der Methode `Level.getPoints()` zurückgegeben wird. **Ein Level A ist schwächer als ein Level B, falls es für jeden Wert in `A.getPoints()` einen grösseren Wert in `B.getPoints()` gibt. Zum Beispiel sind die Level `{2,3}` und `{3,4}` schwächer als das Level `{3,5}`, da 5 grösser als die anderen Werte ist. Die Level `{5}` und `{4,6}` hingegen sind nicht schwächer als `{3,5}`, da es keinen Wert grösser als 5 und 6 in `{3,5}` gibt.**

ItemFactory Die Klasse `ItemFactory` wird verwendet, um Datenbankeinträge zu erstellen. Die Methode `ItemFactory.createItem(Level level, int id, int age, int health)` gibt ein Exemplar der Klasse `Item` zurück, deren Attribute mit den Argumenten initialisiert wurden.

Database Die Klasse `Database` repräsentiert eine Datenbank und hat folgende vorgegebene Methoden:

- `Database.getItemFactory()` gibt ein Exemplar von `ItemFactory` zurück. Die `ItemFactory` `I` ist assoziiert mit der Datenbank `D`, falls `I` von `D.getItemFactory()` zurückgegeben wird.

- `Database.add(Item item)` fügt der Datenbank den Eintrag `item` hinzu.
 - `Database.getItems()` gibt die Liste aller Einträge zurück, welcher der Datenbank hinzugefügt wurden. Sie dürfen annehmen, dass für eine Datenbank `D` alle Einträge in `D.getItems()` eine einzigartige ID haben, über `D.add` hinzugefügt wurden, über `D.getItemFactory()` erstellt wurden, und keiner anderen Datenbank hinzugefügt werden. Ein hinzugefügter Eintrag wird nie wieder entfernt.
- a) Implementieren Sie die Methode `Database.summary(Level groupLevel)`, welche die durchschnittlichen Alter für Gruppen von Einträgen berechnet. Die Gruppen sind wie folgt definiert: Für jede ganze Zahl k , die ein Vielfaches von 10 ist, besteht die Gruppe für k aus der Menge aller Einträge E , so dass
- (a) das Level von E schwächer ist als das Argument `groupLevel`; *und*
 - (b) der abgerundete Gesundheitswert von E gleich k ist (in Java gilt `E.getHealth() / 10 * 10 == k`).

Die Methode gibt eine Map `<Integer, Integer>` zurück, die für jede wie oben definierte Gruppe das durchschnittliche Alter speichert. Für einen Schlüssel k gibt die Map den Wert `null` zurück, falls k kein Vielfaches von 10 ist oder die Gruppe für k leer ist. Die Map gibt für einen Schlüssel k einen Wert v ungleich `null` zurück, genau dann wenn **alle** folgenden Bedingungen erfüllt sind:

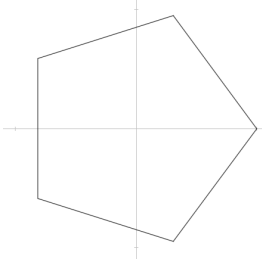
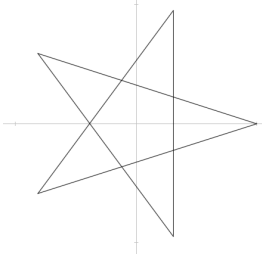
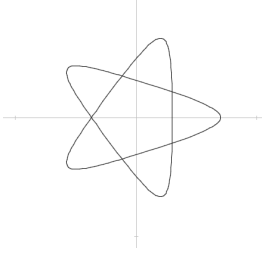
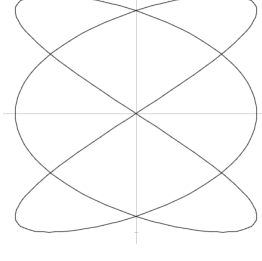
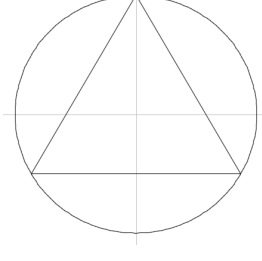
- k ist ein Vielfaches von 10;
- Die Gruppe der gefundenen Einträge für k ist nicht leer;
- v ist das runter-gerundete durchschnittliche Alter aller Einträge in der Gruppe.

ID	Level	Alter	Health
1	{1,2}	20	100
2	{2,3}	31	109
3	{5,6}	50	100
15	{4}	20	133

Beispiel: Für die obige Datenbank mit 4 Einträgen soll `summary` für das Level `{3,5}` die Map `{100 \mapsto 25, 130 \mapsto 20}` zurückgeben. Das Level der Einträge mit ID 1, 2 und 15 ist schwächer als `{3,5}`. Die beiden Einträge 1 und 2 sind in einer Gruppe für $k = 100$. Das gerundete durchschnittliche Alter ist $51/2 = 25$. Der Eintrag mit ID 15 ist allein in der Gruppe für $k = 130$. Die Map hat keine weiteren Einträge. Der Eintrag mit ID 3 hat keinen Einfluss auf das Ergebnis, weil sein Level nicht schwächer als `{3,5}` ist.

Wir geben zwei Testdateien zur Verfügung. “DatabaseTest.java” enthält Tests, welche wir an einer Prüfung geben würden. “GradingDatabaseTest.java” enthält Tests, welche wir zum Korrigieren einer Prüfung verwenden würden. Testen Sie Ihre Lösung zuerst ausgiebig mit “DatabaseTest.java” (am besten fügen Sie selber neue Tests hinzu) und dann können Sie “GradingDatabaseTest.java” verwenden, um zu sehen wie Ihre Lösung an einer Prüfung abgeschnitten hätte.

Anhang: Programme und Zeichnungen

Bezeichnung	Programm	n	Zeichnung
n -Eck	<pre>alpha = i * ((2*PI) / n); x = cos(alpha); y = sin(alpha);</pre>	$n \geq 3$	
n -Stern	<pre>m = 2; alpha = i * ((2*PI) * (m/n)); x = cos(alpha); y = sin(alpha);</pre>	$n \geq 5$, n ungerade	
Hypotrochoid (Wikipedia)	<pre>rA = 0.5; rB = 0.3; d = 0.5; theta = i * ((2*PI) / (n/3)); diff = rA - rB; x = (diff * cos(theta)) + (d * cos((diff / rB) * theta)); y = (diff * sin(theta)) - (d * sin((diff / rB) * theta));</pre>	grosse n	
Lissajous-Figur (Wikipedia)	<pre>pA = 3; pB = 2; alpha = i * ((2*PI) / n); x = cos(pA * alpha); y = sin(pB * alpha);</pre>	grosse n	
Kombination (mit signum -Trick)	<pre>alphaA = (i * ((2*PI) / 200)) + (PI/2); xA = cos(alphaA); yA = sin(alphaA); alphaB = (i * ((2*PI) / 3)) + ((7/6)*PI); xB = cos(alphaB); yB = sin(alphaB); caseB = ((signum(i-200.5) + 1) / 2); caseA = 1 - caseB; x = (caseA * xA) + (caseB * xB); y = (caseA * yA) + (caseB * yB);</pre>	$n = 203$	

Anhang B: MyList Interface

```
public interface MyList<T> {

    /**
     * Return the value at position 'index'.
     * Throws a NoSuchElementException if the argument exceeds the list size.
     */
    public T get(int index);

    /**
     * Return the list node at position 'index'.
     * Throws a NoSuchElementException if the argument exceeds the list size.
     */
    public MyListNode<T> getNode(int index);

    /** Set the value at position 'index' to 'value'. */
    public void set(int index, T value);

    /** Returns whether the list is empty (has no values). */
    public boolean isEmpty();

    /** Returns the size of the list. */
    public int getSize();

    /** Inserts 'value' at position 0 in the list. */
    public void addFirst(T value);

    /** Appends 'value' at the end of the list. */
    public void addLast(T value);

    /** Appends the 'other' list to the end of the list. */
    public void addAll(MyList<T> other);

    /**
     * Removes and returns the first value of the list.
     * Throws a NoSuchElementException if the List is empty.
     */
    public T removeFirst();

    /**
     * Removes and returns the last value of the list.
     * Throws a NoSuchElementException if the List is empty.
     */
    public T removeLast();

    /** Removes all values from the list, making the list empty. */
    public void clear();

    /** Returns an iterator to the data structure. */
    public Iterator<T> iterator();
}
```

Anhang C: MyListNode Interface

```
public interface MyListNode<T> {  
  
    /** Returns the value stored in the node. */  
    public T value();  
  
    /** Sets the value stored in the node. */  
    public void setValue();  
  
    /** Returns false iff this is the last node of the list. */  
    public boolean hasNext();  
  
    /** Returns next node. */  
    public MyListNode<T> next();  
  
    /** Sets the next node. */  
    public void setNext();  
}
```