P. Thiemann, L. Fennell Sommersemester 2017

Programmieren in Java

`http://proglang.informatik.uni-freiburg.de/teaching/java/2017/`

food

Nahrungsmittel

Woche 06 Aufgabe 1/3

Herausgabe: 2017-05-29
Abgabe: 2017-06-17

Achtung: beachten Sie unbedingt die allgemeinen Hinweise zur Abgabe auf der Homepage.

Project `food`
Package `food`
Klassen

Food

`public Food (String name,`
`  double carbohydrateShare, double fatShare, double proteinShare)`
`public String getName()`
`public double relativeEnergyDensity()`
`@Override public boolean equals(Object r)`
`@Override public int hashCode()`

Meal

`public Meal(String name, Map<Food, double> ingredients)`
`public String getName()`
`public Map<Food, Double> getIngredients();`
`public double getCalorificValue();`

In dieser Aufgabe sollen Klassen für Nahrungsmittel und Mahlzeiten erstellt werden.

Nahrungsmittel haben einen Namen und bestehen anteilig aus den Nährstoffen: Kohlenhydrate, Fett und Proteine (es gibt wohl noch einige mehr, aber der Einfachheit halber beschränken wir uns auf diese drei). Der Konstruktor nimmt den Namen des Nahrungsmittels als String und die Nährstoffanteile als double. Gibt es negative Anteile, oder ergibt die Summe der Anteile ein Wert größer 1, solle eine `IllegalArgumentException` geworfen werden. Für den Namen gibt es eine Getter-Methode. Die Methode `relativeEnergyDensity` soll den relative Energiedichte des Nahrungsmittels in Kilojoule pro Gramm (kJ/g) zurückgeben. Der Berechnung der relativen Energiedichte sollen die Brennwertangaben der EU zugrunde liegen:

`https://de.wikipedia.org/wiki/Physiologischer_Brennwert#Brennwertangaben_in_der_N.C3.A4hrwertkennzeichnung_der_EU`

Nahrungsmittel sollen eindeutig durch ihren Namen bestimmt sein.

Mahlzeiten haben einen Namen und Zutaten. Letztere werden an den Konstruktor als `Map` übergeben, die für jedes enthaltene Nahrungsmittel die Menge in Gramm angibt. Mahlzeiten haben des Weiteren eine Getter-Methode für den Namen und eine Methode `getCalorificValue`, die den Brennwert der Mahlzeit in Kilojoule (kJ) zurückgibt.

Im Skelett zu dieser Aufgabe finden sie die Klasse `FoodTestExamples` mit einigen Beispieltests.

```
package food;

import org.junit.Test;

import java.util.HashMap;
import java.util.Map;

import static org.junit.Assert.*;

public class FoodTestExamples {

    @Test
    public void testMeal() {
        Map<Food, Double> ingredients = new HashMap<>();
        ingredients.put(new Food("Water", 0.0, 0.0, 0.0), 1000.0);
        ingredients.put(new Food("Sugar", 1.0, 0.0, 0.0), 100.0);
        ingredients.put(new Food("Protein", 0.0, 0.0, 1.0), 100.0);

        Meal shake = new Meal("Fitness-Shake", ingredients);
        assertEquals(3400.0, shake.getCalorificValue(), 0.0001);
    }

    @Test(expected = IllegalArgumentException.class)
    public void testIllegalFood() {
        new Food("Burger", 1, 1, 1);
    }
}
```

Das Beispiel demonstriert auch, wie Sie in JUnit auf das Werfen von Exceptions testen können.

Hinweise

- Die Methoden `equals` und `hasCode` werden von den Collectionklaseen (und überall sonst in Java) verwendet um festzustellen, ob zwei Objekte als gleich zu betrachten sind. Der Gleichheitsoperator (`==`) ist *nur bei primitiven Datentypen* geeignet; für Objekte ist er oft unbrauchbar. Unter diesem Link finden Sie Beispiel zum Implementieren von `equals` und `hashCode`.

  `https://www.mkyong.com/java/java-how-to-overrides-equals-and-hashcode/`

In IntelliJ können Sie sich die Methoden auch über `Code -> Generate` generieren lassen.

- Achtung! Die Gleichheit von Nahrungsmitteln ist in dieser Aufgabe spezifiziert; bitte beachten Sie dies.
- Es ist auch dringend zu empfehlen eine `toString` Methode zu implementieren. Diese wird in Java (und vor allem auch von JUnit) dafür verwendet um Informationen über Objekte in Fehlermeldungen anzuzeigen.

P. Thiemann, L. Fennell Sommersemester 2017

Programmieren in Java

`http://proglang.informatik.uni-freiburg.de/teaching/java/2017/`

rationals

Rationale Zahlen

Woche 06 Aufgabe 2/3

Herausgabe: 2017-05-29
Abgabe: 2017-06-17

Achtung: beachten Sie unbedingt die allgemeinen Hinweise zur Abgabe auf der Homepage.

Project `rationals`
Package `rationals`
Klassen

Rational
`public Rational(long nominator, long denominator)`
`public Rational add(Rational r)`
`public Rational multiply(Rational r)`
`public Rational invert(Rational r)`
`public double toDouble()`
`public long getNominator()`
`public long getDenominator()`
`@Override public boolean equals(Object r)`

Rationale Zahlen sind die Teilmenge der Reellen Zahle, die sich durch einen Bruch aus zwei ganzen Zahlen darstellen lassen.

`https://de.wikipedia.org/wiki/Rationale_Zahl`

Implementieren Sie die Klasse `Rational`, deren Objekte rationale Zahlen darstellen. Repräsentiert werden diese durch zwei `long`-Werte, den Zähler (eng. nominator) und den Nenner (eng. denominator) der Zahl. Die `Rational`s haben gegenüber dem Fließkommatyp `double` den Vorteil, dass Berechnungen immer präzise ausgeführt werden können, so lange das Ergebnis nicht den Zahlenbereich von `long` verlässt.

Die Parameter der Konstruktors sind Zähler und Nenner des `Rational`s als `long`-Werte. Wenn der Nenner 0 ist, soll eine `IllegalArgumentException` geworfen werden. Eine rationale Zahl ist per se nicht eindeutig durch zwei `long`-Werte bestimmt (z.B. $\frac{1}{2} = \frac{2}{4} = \ldots$) Sorgen Sie deshalb dafür, dass die interne Darstellung der rationalen Zahl vollständig gekürzt und kanonisch ist.

Die Methoden `add`, `multiply` und `invert` ensprechen den gleichnamigen Rechenoperationen. Ist das Ergebnis der Operation nicht präzise als `Rational` darstellbar, oder ist eine Operation nicht definiert, soll eine `ArithmeticException` geworfen werden.

Die Methode `toDouble` gibt den `double`-Wert der Zahl zurück. (Dieser dieser entspricht natürlich unter Umständen nur ungefähr der rationalen Zahl.)

Die Methoden `getNominator` und `getDenominator` geben jeweils Zähler und Nenner der Zahl zurück. Achten Sie darauf, dass diese Methode für gleiche Zahlen auch das gleiche Ergebnis liefern.

Die Methode `equals` gibt `true` zurück genau dann wenn das Argument `other` ein `Rational` Objekt ist und der gleichen Zahl entspricht.

Im Skelett zu dieser Aufgabe finden sie die Klasse `RationalTestExamples` mit einigen Beispieltests.

```
package rationals;

import org.junit.Test;

import static org.junit.Assert.*;

public class RationalTestExamples {

    @Test
    public void testAddingPrecicely() {
        Rational r1 = new Rational(100000000001, 1);
        Rational r2 = new Rational(1, 2);
        assertEquals(new Rational(200000000011, 2), r1.add(r2));
    }

    @Test
    public void testUniqueNominator() {
        Rational r1 = new Rational(1, 2);
        Rational r2 = new Rational(2, 4);
        assertEquals(r1.getNominator(), r2.getNominator());
        assertEquals(r1.getDenominator(), r2.getDenominator());
    }

    @Test(expected = IllegalArgumentException.class)
    public void testThrowingWithDenominatorZero() {
        new Rational(1, 0);
    }
}
```

Das Beispiel demonstriert auch, wie Sie in JUnit auf das Werfen von Exceptions testen können.

Hinweise

- Zum Kürzen empfehlen wir den Algorithmus von Euklid

  `https://de.wikipedia.org/wiki/Euklidischer_Algorithmus#Beschreibung_durch_Pseudocode_2`

Auch nützlich:

`https://de.wikipedia.org/wiki/Kleinstes_gemeinsames_Vielfaches#Zusammenhang_zwischen_kgV_und_dem_gr.C3.B6.C3.9Bten_gemeinsamen_Teiler`

- Die Klassen `java.lang.Math` enthält Funktionen für Rechenoperationen ohne impliziten Überlauf.
- Um Literale vom Typ `long` zu Schreiben muss ihnen ein „l" hintenangestellt werden (z.B. `9999999999999999l`). Bei zu großen Zahlen, die nicht mehr in `int` passen, gibt es sonst eine Fehlermeldung vom Compiler.

P. Thiemann, L. Fennell Sommersemester 2017

Programmieren in Java

`http://proglang.informatik.uni-freiburg.de/teaching/java/2017/`

block-world

Eine Welt der fallenden Blöcke

Woche 06 Aufgabe 3/3

Herausgabe: 2017-05-29
Abgabe: 2017-06-17

Achtung: beachten Sie unbedingt die allgemeinen Hinweise zur Abgabe auf der Homepage.

Project `block-world`
Package `blockworld`
Klassen

`Block`
`public Block(int x, int y, int velocity, char shape)`

`BlockWorld`
`public BlockWorld(int width, int height, List<Block> blocks, char empty)`
`public int getWidth();`
`public int getHeight();`
`public char[][] observe()`
`public void step()`
`public boolean isDead()`

In dieser Aufgabe werden Welten mit fallenden Blöcken, kurz „Blockwelten", im Computer geschaffen. Blockwelten sind zweidimensional, endlich und diskret.

Eine Blockwelt der Höhe h und Breite w besteht aus einem Gitter von $w \times h$ Positionen. Die obere, linke Ecke des Gitters hat die Koordinaten $(0, 0)$, die untere, rechte Ecke die Koordinaten $(w - 1, h - 1)$. Ferner enthält eine Blockwelt natürlich Blöcke. Jeder Block nimmt eine Position auf dem Gitter ein. Jede Position kann von mehreren Blöcken besetzt werden [1]

Blockwelten ändern sich in diskreten Schritten. Blöcke haben eine Geschwindigkeit, mit der sie in der Welt nach unten fallen. Blockgeschwindigkeiten sind positive (d.h. ≥ 0), ganzahlige Werte und geben die Anzahl der Felder an, die ein Block pro Schritt nach unten fällt. Ist ein Block unten auf dem Boden angelangt, bleibt der dort. Das bedeutet auch, dass Blockwelten nach endlichen vielen Schritten „tot" sind; irgendwann bewegen sich in ihnen keine Blöcke mehr.

Implementieren Sie die Klassen `BlockWorld` und `Block`. Analog zur obigen Beschreibung wird ein `BlockWorld` Objekt durch Angabe von Breite und Höhe und einer Liste von Blöcken

[1] Die führenden Blockwelt-Wissenschaftler sind sich nicht einig, wie das in einer 2D Welt überhaupt möglich sein kann ... aber die experimentellen Beobachtungen lassen keinen anderen Schluss zu.

initialisiert. Zusätzlich hat der Konstruktor noch eine `char`-Argument, das angibt, wie leere Positionen in der Welt beobachtet werden können (s.u.). Blöcke haben (mindestens) eine Position und eine Geschwindigkeit und eine Form als `char`. Positionen und Geschwindigkeiten müssen positiv sein, ansonsten wirft der Konstruktor von `Block` eine `IllegalArgumentException`. Durch Aufrufen der `step` Methode wird ein Schritt in einem `BlockWorld` Objekt ausgeführt. Die Methode `isDead` gibt `true` zurück genau dann wenn die `BlockWorld` kein Block mehr bewegt werden kann. Der Konstruktor von `BlockWorld` wirft eine `IllegalArgumentException`, wenn die ihm übergebenen Blöcke nicht in die Welt passen.

Die Methode `observe` gibt ein `char` Array zurück, der den aktuellen Zustand der Welt darstellt. Die Positionen des Arrays entsprechen dabei den Positionen der Welt. Ist eine Koordinate in der Welt leer, enthält das Array das Zeichen `empty`. Andernfalls enthält es das größte unter den Zeichen der Blöcke, die sich auf der Position befinden. (Das heißt, dass bei einer Welt mit den Blöcken `new Block(0, 0, 1, 'a')` und `new Block(0, 0, 1, 'b')` auf Position $(0, 0)$ das Zeichen `'b'` beobachtet wird)

Im Skelett zu dieser Aufgabe finden Sie die Klasse `BlockWorldExampleTests`, die einige JUnit-Beispieltestfälle enthält.

```
package blockworld;

import org.junit.Test;

import java.util.Arrays;
import java.util.Collections;
import java.util.List;

import static org.junit.Assert.*;

public class BlockWorldExampleTest {

    @Test
    public void testMinimalWorld() {
        List<Block> bs = Collections.singletonList(new Block (0, 0, 1, 'x'));
        BlockWorld w = new BlockWorld(1, 2, bs, '.');
        assertFalse(w.isDead());
        assertArrayEquals(new char[][]{ new char[]{ 'x', '.'}}, w.observe());
        w.step();
        assertTrue(w.isDead());
        assertArrayEquals(new char[][]{ new char[]{ '.', 'x'}}, w.observe());
    }

    @Test(expected = IllegalArgumentException.class)
    public void testFailingBlockConstruction() {
        new Block(-1, 0, 1, 'x');
    }

}
```

Hinweise:

- Die oben beschriebenen Methoden reichen Ihnen zur Lösung der Aufgabe nicht aus. Definieren Sie weitere so wie es Ihnen sinnvoll erscheint. In der Vorlesung werden dann „best practices" zum Klassendesign besprochen.
- Die Funktion `java.util.Arrays.asList` erlaubt das Umwandeln eines Arrays als Liste. Außerdem ermöglicht die Funktion eine kompakte Schreibweise für Listen, z.B. für Testfälle.

```
1  List<Integer> somePrimes = Arrays.asList(2, 3, 5, 7, 11, 17);
```

- Im Skelett des Projekts befindet sich auch eine Klasse `Main`. Diese enthält eine `main`-Methode, die es erlaubt, Blockwelten in einem Fenster grafisch darzustellen und ablaufen zu sehen. Vielleicht inspiriert Sie das ja auch, sich etwas mit GUI-Programmierung in Java zu beschäftigen.

  `https://docs.oracle.com/javase/tutorial/uiswing/TOC.html`

  (In dieser Vorlesung werden wir aber nicht mehr dazu kommen)