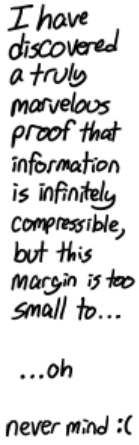
252-0027-00: Einführung in die Programmierung

Übungsblatt 6

Abgabe: 5. November 2019, 10:00

Checken Sie mit Eclipse wie bisher die neue Übungs-Vorlage aus. Beachten Sie, dass Sie mehrere unabhängige Programme im selben Eclipse-Projekt haben werden. Bevor Sie ein Programm starten, achten Sie deshalb darauf, dass Sie die richtige Datei im Package Explorer ausgewählt oder im Editor geöffnet haben. *Vergessen Sie nicht, Ihren Programmcode zu kommentieren!*



xkcd: Margin
Randall Munroe
(CC BY-NC 2.5)

Aufgabe 1: Schweizer Uhrzeit (Bonus!)

Laut Donald Knuth «*hat eine Person etwas erst richtig verstanden, nachdem sie es einem Computer beigebracht hat, d.h. es als Algorithmus ausgedrückt hat.*» In dieser Bonus-Aufgabe sollen Sie zeigen, dass Sie verstanden haben, wie in der Deutschschweiz die Uhrzeit ausgedrückt wird.

Vervollständigen Sie die Methode `toSwissGerman` in der Klasse `SwissTime`. Diese Methode nimmt als Parameter einen String mit dem Format `hh:mm`, wobei `hh` die Stunden und `mm` die Minuten sind, und soll einen String zurückgeben, der diese Uhrzeit auf Schweizerdeutsch enthält.

Beispiele:

```
00:00 -> 12i znacht
01:45 -> viertel vor 2 znacht
09:25 -> 5 vor halbi 10i am morge
12:01 -> 1 ab 12i am mittag
16:46 -> 14 vor 5i am namittag
21:51 -> 9 vor 10i am abig
22:37 -> 7 ab halbi 11i znacht
```

Wie Sie sehen, geht die Schweizer Zeit nur von 1 bis 12, dafür gibt es verschiedene Tageszeiten-Suffixe (“znacht”, “am morge”, usw.). Gegenüber dem `hh:mm`-Format wird die Stunde zudem um 1 erhöht, falls die Anzahl Minuten grösser oder gleich 25 ist (“5 vor halbi 10i”). Das Audrücken der Minuten selbst ist noch komplizierter: wenn es weniger als 25 sind, sagt man “ab”, sonst grundsätzlich “vor”; allerdings zwischen Minute 25 und 39 sind sie “vor halbi” oder “ab halbi”, und wenn es genau 15, 30 oder 45 sind, sagt man “viertel ab”, “halbi” oder “viertel vor”.

Im “test”-Ordner finden Sie eine grössere Menge von JUnit-Tests, welche das Format noch genauer spezifizieren. Diese (und weitere) werden für die Bewertung verwendet. Versuchen Sie also, diese Tests zu lesen und zu verstehen, und passen Sie Ihre Lösung so lange an, bis sie alle davon besteht. Ihre Lösung muss nur korrekt formatierte Eingaben unterstützen.

PS: Falls die Spezifikation nicht Ihrem eigenen Dialekt entspricht, dürfen Sie gerne eine Kopie von `SwissTime` anfertigen und nach Ihrem Gusto gestalten. Für die Abgabe, d.h. in der Klasse `SwissTime`, müssen sich aber an die vorgegebene Spezifikation halten. `:P`

Aufgabe 2: Datenanalyse mit Personen

In der letzten Übung haben Sie Körpergrössen analysiert. In dieser Aufgabe werten Sie einen Personen-Datensatz mit mehreren "Spalten" aus. Die Spalten enthalten Werte für Gewicht, Alter, Geschlecht, usw. Um einfach mit diesen Daten zu arbeiten, entwerfen Sie eine Klasse `Person`, welche alle Eigenschaften einer Person als Felder enthält. Weiter schreiben Sie ein Programm "PersonenAnalyse.java", welche die Personendaten aus einer Datei einliest und diese auswertet. Sie finden wie gewohnt Tests für Teile des Programms im "test"-Ordner.

Die Daten befinden sich in der Datei "body.dat.txt", welche wie folgt aufgebaut ist: Auf der ersten Zeile steht die Anzahl Datensätze in der Datei. Der Rest der Datei ist tabellarisch aufgebaut, wobei jede Zeile die Daten einer Person enthält. Die Beschreibung der Spalten und die Typen, den Sie für die Felder der Klasse `Person` verwenden sollen, sehen Sie in Tabelle 1.

Spalte	Beschreibung	Java-Typ
0	Schulterbreite in cm	`double`
1	Brusttiefe in cm	`double`
2	Brustbreite in cm	`double`
3	Alter in Jahren	`int`
4	Gewicht in kg	`double`
5	Grösse in cm	`double`
6	Geschlecht (1: männlich, 0: weiblich)	`boolean`

Tabelle 1: Datenbeschreibung von "body.dat.txt"

a) Vervollständigen Sie die Klasse `Person` (in der Datei "Person.java"), indem Sie sie mit Feldern und Methoden ergänzen. Fügen Sie für jede Spalte im Datensatz ein Feld mit dem entsprechenden Java-Typ hinzu und implementieren Sie den (schon vorhandenen) Konstruktor so, dass er diese Felder initialisiert.

   Füllen Sie dann die Methode `beschreibung()` so aus, dass sie einen `String` zurück gibt, den die Person beschreibt. Beispielsweise könnte die Methode folgendes zurückgeben:

   ```
   Person (m, 43 Jahre, 179.7 cm, 86.4 kg)
   ```

b) Als ersten Schritt der Datenanalyse sollten Sie die Daten einlesen und ein Array von `Person`-Objekten daraus erstellen. Implementieren Sie dazu `PersonenAnalyse.liesPersonen()` und rufen Sie die Methode mit dem richtigen Argument aus Ihrer `main()`-Methode auf.

c) Ihre erste Analyse soll "ungesunde" Personen anhand des Body-Mass-Index (BMI) finden. Schreiben Sie eine Methode `bodyMassIndex()` in der Klasse `Person`, welche den BMI dieser Person nach folgender Formel berechnet:

$$\text{BMI} = \frac{\text{Gewicht}_{kg}}{\text{Grösse}^2_m}$$

Die Weltgesundheitsorganisation (WHO) definiert folgende Gewichtsklassifikation:

$$\begin{array}{rl} \text{BMI} < 18.5 & \text{untergewichtig} \\ 18.5 \leq \text{BMI} < 25.0 & \text{normalgewichtig} \\ 25.0 \leq \text{BMI} < 30.0 & \text{übergewichtig} \\ 30.0 \leq \text{BMI} & \text{fettleibig} \end{array}$$

In der Method `druckeUngesunde()` sollen Sie nun alle Personen, welche nicht normalgewichtig sind, mithilfe des gegebenen `PrintStreams` ausgeben. Geben Sie für jede solche Person zuerst ihre `beschreibung()` und dann die entsprechende Gewichtsklasse aus. Eine Zeile in der Ausgabe könnte etwa so aussehen:

```
Person (m, 34 Jahre, 167.6 cm, 75.5 kg) ist übergewichtig
```

Um die Ausgabe anzusehen, rufen Sie die Methode `druckeUngesunde()` in der `main()`-Methode mit `System.out` als Argument auf. Ausserdem können Sie die Methode mit den Tests in "PersonenAnalyseTest.java" testen. Nachdem Sie überprüft haben, dass die Ausgabe korrekt ist, ändern Sie Ihr Programm schliesslich so ab, dass die Ausgabe in eine Datei "ungesund.txt" geleitet wird.

d) Glücklicherweise sind die Daten ideal um ein weiteres wichtiges Problem zu lösen: Sie ermöglichen uns, zu jeder Person einen geeigneten Trainingspartner fürs Fitnessstudio zu ermitteln. Da die Geräte im Studio immer auf die Person eingestellt sein müssen, sollten Trainingspartner ähnliche "Dimensionen" aufweisen. Wir beschreiben die Ähnlichkeit von zwei Personen p_1 und p_2 betreffend dieses Kriteriums als *Partner-Qualität* $Q(p_1, p_2)$, die wir wie folgt definieren:

$$\begin{aligned} \mathit{grössenDiff}(p_1, p_2) &= \mathit{grösse}(p_1) - \mathit{grösse}(p_2) \\ \mathit{brustDiff}(p_1, p_2) &= \mathit{brustTiefe}(p_1) \cdot \mathit{brustBreite}(p_1) - \mathit{brustTiefe}(p_2) \cdot \mathit{brustBreite}(p_2) \\ \mathit{schulterDiff}(p_1, p_2) &= \mathit{schulterBreite}(p_1) - \mathit{schulterBreite}(p_2) \\ Q(p_1, p_2) &= \frac{1}{1 + \mathit{grössenDiff}(p_1, p_2)^2 + \frac{\mathit{abs}(\mathit{brustDiff}(p_1, p_2))}{5} + \frac{\mathit{schulterDiff}(p_1, p_2)^2}{2}} \end{aligned}$$

i) Implementieren Sie diese Formel in der Methode `PersonenAnalyse.partnerQualitaet()` und überprüfen Sie die Korrektheit mithilfe der Tests in "PersonenAnalyseTest.java". Folgende Methoden könnten dabei nützlich sein: `Math.pow()` und `Math.abs()`.

ii) Schreiben Sie nun eine Methode `druckeGuteTrainingsPartner()`, welche die Qualität aller möglichen Paare berechnet. Sofern die Qualität eines Paares 0.8 übersteigt, sollen die Partner-Qualität, sowie die Beschreibungen und Gewichtsklassen beider Personen ausgegeben werden. Achten Sie darauf, dass kein Paar doppelt ausgegeben wird (eine bestimmte Person darf jedoch in mehreren Paaren auftreten). Ein Beispiel eines solchen Paares ist:

```
Person (m, 21 Jahre, 177.8 cm, 79.5 kg), übergewichtig
Person (m, 19 Jahre, 177.8 cm, 76.6 kg), normalgewichtig
Qualität: 1.0
```

Kennen Sie eine bessere Formel für Partner-Qualität?

Aufgabe 3: Black-Box Testing

Im letzten Übungsblatt haben Sie Testautomatisierung mit JUnit kennengelernt. In dieser Aufgabe sollen Sie nun Tests für eine Methode schreiben, deren Implementierung Sie nicht kennen. Dadurch werden Sie weniger durch möglicherweise falsche Annahmen beeinflusst, die bei einer Implementierung getroffen wurden. Sie müssen sich also überlegen, wie sich *jede* fehlerfreie Implementierung verhalten muss. Diesen Ansatz nennt man Black-Box Testing.

In Ihrem "U06"-Projekt befindet sich eine "blackbox.jar"-Datei, welche eine kompilierte Klasse `BlackBox` enthält. Den Code dieser Klasse können Sie nicht sehen, aber sie enthält eine Methode `void rotateArray(int[] values, int steps)`, welche Sie aus einer eigenen Klasse oder einem Unit-Test aufrufen können. Diese Methode "rotiert" ein `int`-Array um eine gegebene Anzahl Schritte.

Vereinfacht macht die Methode `rotateArray()` Folgendes: Eine Rotation mit `steps=1` bedeutet, dass alle Elemente des Arrays um eine Position nach rechts verschoben werden. Das letzte Element wird dabei zum ersten. Mit `steps=2` wird alles um zwei Positionen nach rechts rotiert, usw. Eine Rotation nach links kann mit einer negativen Zahl für `steps` erreicht werden. Das folgende Beispiel ist der erste, einfache Test, den Sie in der Datei "BlackBoxTest.java" finden:

```
int[] values = new int[] { 1, 2 };
int[] expected = new int[] { 2, 1 };
BlackBox.rotateArray(values, 1);
assertArrayEquals(expected, values);
```

Dieser Test prüft, dass das Array `{ 1, 2 }` nach einer Rotation mit `steps=1` aussieht wie `{ 2, 1 }`. Die Methode `assertArrayEquals(expected, values)` prüft, dass die beiden Arrays `expected` und `values` die selben Elemente enthalten. Wenn nicht, schlägt der Test fehl.

Die genaue Definition von `rotateArray` lautet wie folgt: Nach einem Aufruf ist das Element am Index i gleich dem Element, das zuvor am Index $(i - \texttt{steps}) \mod \texttt{values.length}$ war, für alle i zwischen 0 und $\texttt{values.length} - 1$, inklusive. "mod" steht für *modulo* und bezeichnet den Rest einer Ganzzahl-Division (siehe Wikipedia). Mit diesem Wissen sollen Sie nun weitere Tests schreiben, die möglichst gut prüfen, ob sich eine Implementierung wunschgemäss verhält. Überlegen Sie sich genau, was für die Parameter `values` und `steps` angegeben werden kann, und wie `values` nach dem Aufruf von `rotateArray()` aussieht. Der gegebene Test prüft beispielsweise nur, dass bei einem Array mit zwei Elementen nach einer Rotation um `1` die Elemente vertauscht sind. Eine Implementierung, die ein Array einfach umkehrt, würde den Test auch bestehen.

Um die Stärke Ihrer Tests zu beurteilen, werden wir verschiedene, teilweise fehlerhafte Implementierungen mithilfe Ihrer Tests prüfen. Je mehr Fehler Ihre Tests aufdecken, desto besser. Tests sollten fehlschlagen, falls die Implementierung fehlerhaft ist, und erfolgreich durchlaufen, falls keine Fehler vorhanden sind.

Aufgabe 4: Linked List

Bisher haben Sie Arrays verwendet, wenn Sie mit einer grösseren Anzahl von Werten gearbeitet haben. Ein Nachteil von Arrays ist, dass die Grösse beim Erstellen des Arrays festgelegt werden muss und danach nicht mehr verändert werden kann. In dieser Aufgabe implementieren Sie selbst

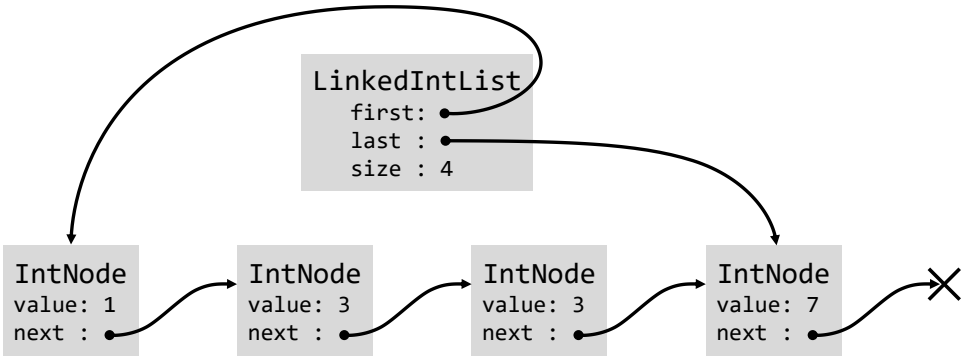


Abbildung 1: Verkettete Liste mit Werten 1, 3, 3, 7.

eine Datenstruktur, bei welcher die Grösse im Vornherein nicht bestimmt ist und welche jederzeit wachsen und schrumpfen kann: Eine *linked list* oder *verkettete Liste*.

Eine verkettete Liste besteht aus mehreren Objekten, welche Referenzen zueinander haben. Für diese Aufgabe besteht jede Liste aus einem "Listen-Objekt" der Klasse `LinkedIntList`, welches die gesamte Liste repräsentiert, und aus mehreren "Knoten-Objekten" der Klasse `IntNode`, eines für jeden Wert in der Liste. Die Liste heisst "verkettet", weil jedes Knoten-Objekt ein Feld mit einer Referenz zum nächsten Knoten in der Liste enthält. Das `LinkedIntList`-Objekt schliesslich enthält eine Referenz zum ersten und zum letzten Knoten und hat ausserdem ein Feld für die Länge der Liste.

Abbildung 1 zeigt eine Liste, welche die Werte 1, 3, 3, 7 enthält. Beachten Sie, dass das `next`-Feld des letzten Knotens in der Liste auf kein Objekt zeigt, d.h. den Wert `null` enthält. Ausserdem wird eine leere Liste so repräsentiert, dass beide Felder `first` und `last` den Wert `null` enthalten (und `size` gleich `0` ist).

a) Schreiben Sie die Klassen `LinkedIntList` und `IntNode`, welche zusammen eine verkettete Liste von `int`s ergeben. Erstellen Sie die Klassen und fügen Sie die benötigten Felder hinzu. Zuerst sollen Sie der Klasse `LinkedIntList` eine `addLast()`-Methode hinzufügen, welche einen `int`-Wert entgegen nimmt und diesen am Ende der Liste anhängt.

   Erweitern Sie danach die `LinkedIntList`-Klasse mit folgenden Methoden, um die Klasse benutzerfreundlicher zu machen:

Name	**Parameter**	**Rückg.-Typ**	**Beschreibung**
`addFirst`	`int value`	`void`	Fügt einen Wert am Anfang der Liste ein
`removeFirst`		`int`	Entfernt den ersten Wert und gibt ihn zurück
`removeLast`		`int`	Entfernt den letzten Wert und gibt ihn zurück
`clear`		`void`	Entfernt alle Wert in der Liste
`isEmpty`		`boolean`	Gibt zurück, ob die Liste leer ist
`get`	`int index`	`int`	Gibt den Wert an der Stelle `index` zurück
`set`	`int index, int value`	`void`	Ersetzt den Wert an der Stelle `index` mit `value`

   Einige dieser Methoden dürfen unter gewissen Bedingungen nicht aufgerufen werden. Zum Beispiel darf `removeFirst()` nicht aufgerufen werden, wenn die Liste leer ist, oder `get()` darf

nicht aufgerufen werden, wenn der gegebene Index grösser oder gleich der aktuellen Länge der Liste ist. In solchen Situationen soll sich Ihr Programm mit einer Fehlermeldung beenden. Verwenden Sie folgendes Code-Stück dafür:

```
if(condition) {
    Errors.error(message);
}
```

Ersetzen Sie *condition* mit der Bedingung, unter welcher das Programm beendet werden soll, und *message* mit einer hilfreichen Fehlermeldung. Die `Errors`-Klasse befindet sich bereits in Ihrem Projekt, aber Sie brauchen sie im Moment nicht zu verstehen.

b) Erstellen Sie ein Programm `Echo.java`, welches vom Benutzer `int`-Werte entgegen nimmt, diese in einer `LinkedIntList` speichert und zum Schluss alle Werte in der Liste wieder ausgibt. Das Programm soll solange Werte einlesen, bis der Benutzer eine ungültige Eingabe macht. Verwenden Sie dazu `Scanner.hasNextInt()`.

Um alle Werte auszugeben, soll Ihr Programm von Knoten zu Knoten "springen", angefangen beim ersten Knoten und solange, bis das Ende der Liste erreicht d.h. der nächste Knoten gleich `null` ist. Sie können folgendes Code-Stück dafür verwenden:

```
for(IntNode n = list.first; n != null; n = n.next) { ... }
```

Aufgabe 5: Analoge Uhr

In dieser Übung geht es darum, eine "Analoge Uhr" zu programmieren, die die aktuelle Zeit inklusive Sekunden anzeigt. Dafür kommt die bereits bekannte Klasse "Window" zum Zuge. Sie befindet sich bereits in Ihrem Projektordner. Im Programm "Uhr.java" finden Sie bereits ein Template für die Aufgabe vor.

1. Zuerst sollen Sie die Zeiger Ihrer Uhr für eine gegebene Uhrzeit richtig zeichnen (z.B. für 13:37). Überlegen Sie sich, wie Sie die Start- und Endpunkte der Zeiger berechnen können. Dazu benötigen Sie für jeden Zeiger den Winkel zwischen 12Uhr und der gegebenen Uhrzeit.

   Hinweis: Benutzen Sie trigonometrische Funktionen (`Math.cos(x)` und `Math.sin(x)`). Die Zeiger können Sie als Linien per `drawLine(x1,y1,x2,y2)` zeichnen und mit der Methode `setStrokeWidth(width)` die Breite der Linien verändern. Die `Window`-Klasse bietet auch noch andere Gestaltungsmöglichkeiten. Suchen Sie nach weiteren `draw...()`- und `fill...()`-Methoden mithilfe der Autovervollständigung von Eclipse.

2. Nun soll die Uhr natürlich die aktuelle Zeit anzeigen (Stunden, Minuten und Sekunden seit Mitternacht). Diese können Sie mithilfe der Methode `System.currentTimeMillis()` ermitteln. Diese Methode gibt zurück, wie viele Millisekunden seit Mitternacht, 01.01.1970 vergangen sind. Beachten Sie auch, dass die Zeit in UTC berechnet wird, weshalb Sie momentan noch eine Stunde dazu addieren müssen.