252-0027-00: Einführung in die Programmierung

Übungsblatt 11 (ohne Bonus)

Abgabe: 2. Dezember 2025, 23:59

Die Bonusaufgabe für diese Übung wird erst am Dienstag Abend der Folgewoche (also am 02.12) um 17:00 Uhr publiziert und Sie haben dann 2 Stunden Zeit, diese Aufgabe zu lösen. Die Projekte erhalten Sie etwas früher, damit Sie genügend Zeit haben, diese zu importieren. Der Abgabetermin für die anderen Aufgaben ist wie gewohnt am Dienstag Abend um 23:59. Bitte planen Sie Ihre Zeit entsprechend.

Ab Woche 5, werden die TAs Ihnen nur dann Feedback geben, wenn Sie eine entsprechende Datei namens "requestfeedback.txt" in dem Projekt-Ordner hochladen. In der Text-Datei geben Sie an, für welche Aufgaben Sie Feedback erhalten möchten. Um eine neue Text-Datei in IntelliJ zu erstellen, rechts-klicken Sie den uXX Projekt-Ordner und wählen *New* → *File*. Im erscheinenden Fenster, benennen Sie die Datei mit "requestfeedback.txt" und drücken die Enter-Taste. Danach wählen Sie *Add*, falls IntelliJ Sie fragt, ob Sie die Datei zu git hinzufügen möchten.

Checken Sie mit IntelliJ, wie bisher die neue Übungs-Vorlage aus. Importieren Sie das IntelliJ-Projekt für die Aufgaben ohne Bonus. Vergessen Sie nicht, Tests zu schreiben! Auch Ausnahmefälle (Exceptions) können mit JUnit getestet werden, z.B. so:

```
boolean thrown = false;
try {
    // code
} catch (SomeException e) {
    thrown = true;
}
if (!thrown) {
    fail("expected some exception");
}
```

Aufgabe 1: Loop-Invariante

Gegeben den Pre- und Postcondition formulieren Sie *nützliche Schleifeninvarianten* für folgende Programme in der Datei "LoopInvariante.txt".

1. Um die Loop-Invariante einfacher schreiben zu können, dürfen Sie `min(arr, i)` benutzen. Hier steht `min(arr, i)` für das minimale Element im Array `arr` von Index `0` bis Index `i` (exklusiv). Alternativ könnte man auch formale Notation benutzen, in dem man mit Quantoren arbeitet. Zum Beispiel, falls $\texttt{m} == \texttt{min(arr, i)}$, dann könnten Sie äquivalent Folgendes schreiben

$$\forall\, 0 \leq \texttt{j} < \texttt{i}\,(\texttt{arr[j]} \leq \texttt{m})$$

```
int min(int[] arr) {
  // Precondition: arr != null && 0 < arr.length
  int m = arr[0];
  int i = 1;

  // Loop-Invariante:
  while (i < arr.length) {
    if (arr[i] < m) {
      m = arr[i];
    }

    i++;
  }

  // Postcondition: m = min(arr, arr.length)
  return m;
}
```

2.
```
String append(String str1, String str2) {
  // Precondition: str1 != null && str2 != null
  String s1 = str1;
  String s2 = str2;

  // Loop-Invariante:
  while (!s2.equals("")) {
    s1 = s1 + s2.charAt(0);
    s2 = s2.substring(1);
  }

  // Postcondition: s1.equals(str1 + str2)
  return s1;
}
```

Achtung: Die Bedingung `str1 != null && str2 != null` ist wichtig, damit Aufrufe wie `s2.equals()`, `s2.charAt(0)` und `s2.substring(1)` überhaupt möglich sind. Der Aufruf `s2.substring(1)` produziert das gleiche Resultat wie `s2.substring(1, s2.length())`.

Aufgabe 2: ItemFactory

In dieser Aufgabe implementieren Sie für eine Datenbank von Personengesundheitsdaten das Deklassifizieren von Einträgen (Task a) und das Verlinken von Einträgen (Task b). Alle Unteraufgaben können separat gelöst werden.

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 eine Liste von Integern definiert, welche in einem Attribut der Klasse `Level` gespeichert wird und von der Methode `Level.getPoints()` zurückgegeben wird. Ein Level `a` ist *verwandt* mit einem Level `b`, falls die Summe der Werte in `a.getPoints()` gleich der Summe der Werte in `b.getPoints()` ist. Zum Beispiel ist das Level $[1, 2, 3, 4]$ verwandt mit den Levels $[10]$ und $[4, 6]$ (die Summe ist überall 10), aber nicht mit dem Level $[4, 5]$.

`ItemFactory` Die Klasse `ItemFactory` wird verwendet, um Datenbankeinträge zu erstellen. Die Methode `ItemFactory.createItem(Level level, int id, int age, int health)` gibt eine Instanz 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 Instanz 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.

1. Implementieren Sie die Methode `ItemFactory.createDeclass(Level level, int id, int targetId)`, die einen *Deklassifikationseintrag* zurückgibt. Ein Deklassifikationseintrag ist selber ein Eintrag, also ein Instanz der Klasse `Item`. Ein Deklassifikationseintrag hat damit

auch eine ID, ein Sicherheitslevel, ein Alter, und einen Gesundheitswert, welche von den entsprechenden getter-Methoden zurückgegeben werden. ID und Sicherheitslevel eines Deklassifikationseintrags sind jeweils das `id` und `level` Argument des `createDeclass` Aufrufs, mit welchem der Eintrag erstellt wurde. Das Alter und der Gesundheitswert eines Deklassifikationseintrags sind jeweils das Alter und der Gesundheitswert des *Zieleintrags* vom Deklassifikationseintrag. Der Zieleintrag von einem Deklassifikationseintrag `d` ist der Eintrag `e`, so dass

- `e.getID()` gleich dem Parameter `targetId` ist, mit welchem `d` erstellt wurde; *und*
- `e` aus der Datenbank ist, mit welcher die `ItemFactory` assoziiert ist, mit welcher `d` erstellt wurde.

Falls es keinen Zieleintrag gibt, wird eine `IllegalArgumentException` von der Methode `createDeclass` geworfen. Beachten Sie, dass Zieleinträge selber Deklassifikationseinträge sein können. Ein Aufruf der Methode `Item.setHealth(h)` auf einem Deklassifikationseintrag hat keinen Effekt; dies wird nicht in den Tests überprüft.

Ein Deklassifikationseintrag `r` *erreicht* einen Eintrag `a`, falls entweder `a` der Zieleintrag von `r` ist oder falls der Zieleintrag von `r` ein Deklassifikationseintrag ist, welcher `a` erreicht. Die Methode `createDeclass` wirft eine `IllegalArgumentException`, falls der zurückzugebene Deklassifikationseintrag `r` einen Eintrag erreicht, dessen Level verwandt ist mit dem Level von `R`. Zur Erinnerung: Der Paragraph über die Klasse `Level` erklärt, wann zwei Level verwandt sind.

2. Implementieren Sie die Methode `Database.createLink(List<Integer> ids)`. Der Methodenaufruf `d.createLink(ids)` *verlinkt* alle Einträge der Datenbank `d` miteinander, welche eine ID haben, die im Argument `ids` enthalten ist. Wenn `e.setHealth(h)` auf einem Eintrag `e` aufgerufen wird, dann wird der Gesundheitswert aller Einträge, welche mit `e` verlinkt sind, auf das Argument `h` gesetzt. Einträge können beliebig oft verlinkt werden und verlinken ist transitiv, das heisst, wenn ein Eintrag `a` mit einem Eintrag `b` verlinkt ist und `b` mit einem Eintrag `c` verlinkt ist, dann ist `a` auch mit `c` verlinkt. Verlinken ist auch immer symmetrisch, das heisst, wenn `a` mit `b` verlinkt ist, dann ist auch `b` mit `a` verlinkt. Zusätzlich ist verlinken reflexiv, das heisst, ein Eintrag ist immer mit sich selber verlinkt.

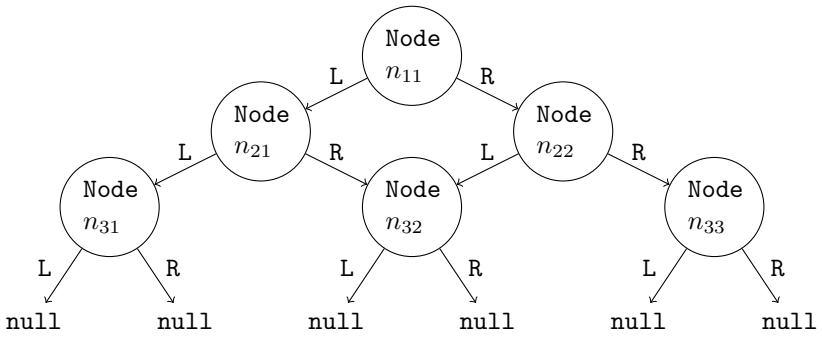
   Der Aufruf `d.createLink(ids)` soll eine `IllegalArgumentException` werfen, falls es eine ID im Argument `ids` gibt, für welche es keinen Eintrag mit der gleichen ID in der Datenbank `d` gibt.

Wir stellen 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.

Aufgabe 3: Pyramide

Die Klasse `Node` repräsentiert einen Knoten in einem gerichteten Graphen, wobei es für jeden Knoten n_1 höchstens zwei gerichtete Kanten von n_1 zu anderen Knoten n_2, n_3 geben kann (n_2 und n_3 können gleich sein). Wir unterscheiden dabei zwischen dem linken und dem rechten Knoten. Die Methode `Node.getLeft()` gibt den linken Knoten und `Node.getRight()` den rechten Knoten zurück (als `Node`-Objekt). Wenn der linke Knoten von n_1 nicht existiert, dann gibt `Node.getLeft()` `null` zurück (analog für den rechten Knoten).

Das Ziel dieser Aufgabe ist, für ein `Node`-Objekt zu entscheiden, ob der durch das `Node`-Objekt definierte Graph einer Pyramide entspricht. Zum Beispiel entspricht der folgende Graph einer Pyramide.

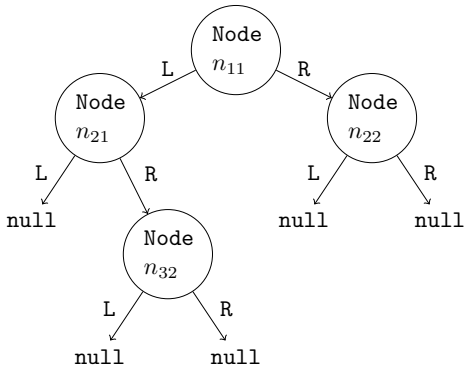


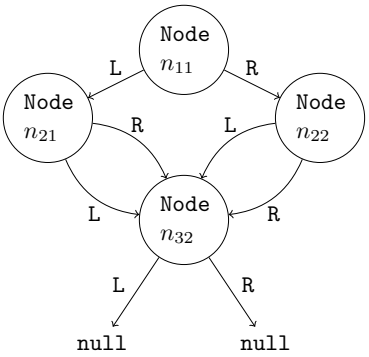
Beachten Sie, dass der rechte Knoten von n_{21} gleich ist wie der linke Knoten von n_{22} (das heisst die Node-Objekte sind gleich!). Ein Graph (wie oben repräsentiert) definiert eine Pyramide genau dann, wenn folgende Bedingungen gelten:

- Der Graph kann in $k \geq 1$ Stufen (Stufe 1, Stufe 2,..., Stufe k) aufgeteilt werden, wobei Stufe i aus i unterschiedlichen Knoten n_{i1}, n_{i2}, ..., n_{ii} besteht. Falls der Graph k Stufen hat, dann hat dieser genau $\frac{k(k+1)}{2}$ unterschiedliche Knoten (Knoten aus verschiedenen Stufen sind unterschiedlich).
- Für Stufe i ($1 \leq i < k$) gilt: der linke Knoten von n_{ij} ($1 \leq j \leq i$) ist durch $n_{(i+1)j}$ gegeben und der rechte Knoten von n_{ij} ist durch $n_{(i+1)(j+1)}$ gegeben.
- Für Stufe k gilt: es gibt keinen linken und keinen rechten Knoten für n_{kj} $(1 \leq j \leq k)$.

Die folgenden Graphen entsprechen zum Beispiel keinen Pyramiden:
Implementieren Sie die `boolean isPyramid(Node node)`-Methode, welche, für den Graph G durch `node` definiert, entscheidet, ob G eine Pyramide definiert. Sie dürfen annehmen, dass G keine Zyklen hat. Die Methode soll eine `IllegalArgumentException` werfen, wenn das Argument `null` ist.

Tipp: Prüfen Sie die Bedingungen Stufe für Stufe, beginnend bei Stufe 1.





Aufgabe 4: Rechnungen (Erweitert aus u08)

In dieser Aufgabe erweitern Sie eine vorherige Aufgabe, in welcher ein System für Stromverbräuche Rechnungen erstellt. Konkret gibt es drei Erweiterungen: (1) Es sollen auch nicht korrekt formatierte Eingabedateien gehandhabt werden. (2) Ein Kunde kann eine beliebige Anzahl von Verbrauchswerten haben. (3) Es gibt eine neue Unteraufgabe b. In der folgenden Aufgabenbeschreibung für Unteraufgabe a sind die Änderungen in **bold** markiert.

a) Vervollständigen Sie die `process`-Methode in der Klasse `Bills`. Die Methode hat zwei Argumente: einen `Scanner`, von dem Sie den Inhalt der Eingabedatei lesen sollen, und einen `PrintStream`, in welchen Sie die unten beschriebenen Informationen schreiben.

   Ihr Programm muss **auch mit manchen nicht korrekt formatierten Eingabedatein umgehen**. **Die Aufgabestellung gibt an, wie mit nicht korrekt formatierten Eingaben umzugehen ist**. Ein Beispiel einer korrekt formatierten Datei finden Sie im Projekt unter dem Namen "Data.txt". Exceptions im Zusammenhang mit Ein- und Ausgabe können Sie ignorieren.

   Eine valide Eingabedatei enthält Zeilen, die entweder den Tarif, der angewendet werden soll, oder die Daten für den Stromverbrauch eines Kunden beschreiben. Der Verbrauch eines Kunden ist niemals grösser als 100000 Kilowattstunden.

   Eine Tarifbeschreibung hat folgendes Format:

   $$\texttt{Tarif}\,␣\,n\,␣\,l_1\,␣\,p_1\,\ldots\,l_n\,␣\,p_n$$

   Folgendes gilt für die Parameter:

   - `Tarif` (so geschrieben) ist ein Keyword, das angibt, dass die Zeile einen Tarif beschreibt.
   - n is eine positive ganze Zahl, welche die Anzahl der Intervalle angibt, für welche ein Strompreis festgelegt ist.
   - Auf n folgt eine Folge von n Paaren von ganzen Zahlen ($l_1␣p_1\,\ldots\,l_n␣p_n$). Die erste Zahl eines Paares gibt die Obergrenze des Intervalls an und die zweite den Preis für diesen Verbrauch; für ein i, so dass $1 \leq i \leq n$, ist l_i also der Verbrauch (in Kilowattstunden), bis zu welchem der Strompreis p_i (in Rappen pro Kilowattstunde) zur Anwendung kommt ($l_i > 0$ und $p_i \geq 0$). Die Paare sind jeweils mit einem Whitespace voneinander getrennt (und l_i und p_i jeweils voneinander auch).

Hier sind einige Beispiele für Tarifbeschreibungen:

- `Tarif 1 100000 30`
  Es gibt ein Intervall und für jede Kilowattstunde müssen 30 Rappen bezahlt werden.
- `Tarif 2 1000 10 100000 30`
  Es gibt zwei Intervalle. Die ersten 1000 Kilowattstunden kosten 10 Rappen pro Kilowattstunde. Der Rest kostet 30 Rappen pro Kilowattstunde.
- `Tarif 3 100 40 1000 10 100000 30`
  Es gibt drei Intervalle. Die ersten 100 Kilowattstunden kosten 40 Rappen pro Kilowattstunde. Die nächsten 1000 Kilowattstunden kosten 10 Rappen pro Kilowattstunde. Der Rest kostet 30 Rappen pro Kilowattstunde.

Wenn ein Kunde insgesamt 2000 Kilowattstunden verbraucht, so beträgt die Rechung für das erste Beispiel 600 Franken, im zweiten Beispiel 400 Franken und 410 Franken im dritten.

Die Beschreibung des Stromverbrauchs eines Kunden hat folgendes Format:

$$ID \textvisiblespace v_{q_1} \textvisiblespace v_{q_2} \ldots v_{q_m}$$

Hierbei gilt für die Parameter:

- ID ist eine positive ganze Zahl.
- v_{q_i} **ist eine ganze Zahl, die den Verbrauch im i-ten Quartal in Kilowattstunden angibt ($v_{q_i} \geq 0$). Es kann eine beliebige Anzahl von Verbrauchwerten geben (auch keine).**

Hier ist ein Beispiel für eine Verbrauchbeschreibung:

```
115 0 0 0 10 2000 12
```

Der Kunde mit ID 115 hat im vierten Quartal 10 Kilowattstunden, im füften Quartal 2000 Kilowattstunden, und im sechsten Kilowattstunden 12 Kilowattstunden Strom verbraucht.

Ein einmal gelesener Tarif wird für alle Kunden angewendet, die nach dieser Tarifinformation in der Eingabedatei erscheinen. Wenn ein neuer Tarif erscheint, dann gilt der danach für die weiteren Kunden bis auf Weiteres. Sie können davon ausgehen, dass eine Kunden-ID nur einmal in der Eingabedatei vorkommen kann und dass die erste Zeile der Eingabedatei eine Tarifbeschreibung ist. Implementieren Sie die **Ihr Program soll Zeilen im Input ignorieren, welche weder mit "Tarif" noch einem positiven Integer beginnen. Falls ein Verbrauchswert im Input kein positiver Integer ist, dann soll eine `IllegalFileFormatException` geworfen werden. Die `IllegalFileFormatException` wird zur Verfügung gestellt.**

Die Methode `process` soll die Eingabedatei verarbeiten und für jeden Kunden eine Zeile

$$ID \textvisiblespace b$$

in den der Methode in `output` übergebenen `PrintStream` schreiben. *ID* ist die ID des Kunden (`int`) und *b* ist eine ganze Zahl, die die jeweilige Rechnung für den Gesamtverbrauch **in Franken** angibt. (Zuerst muss der Gesamtverbrauch berechnet werden, dann kann der entsprechende Tarif angewendet werden.) Berechnen Sie den Rechnungsbetrag und runden Sie das Resultat anschliessend (vor der Ausgabe, aber nach den Berechnungen) auf die nächste ganze Zahl. Sie können hierfür die Methode `Math.round(double a)` verwenden. Die Ausgabe darf keine weiteren Zeichen enthalten. Sie können den Betrag so ausgeben, wie er von der `println`-Anweisung herausgegeben wird, d.h. Sie brauchen das Ergebnis nicht zu formatieren.

b) Implementieren Sie zusätzlich die Methode `query(int lowB, int upB)`. Diese Methode wird nur nach einem Aufruf von `process` aufgerufen. Die Methode `query(int lowB, int upB)` gibt die Id des Kunden zurück, der den grössten Rechnungsbetrag im Interval $[lowB, upB]$ hatte. Der Rechnungsbetrag R' dieses Kunden K' erfüllt die Bedingungen

   - `lowB` $\leq R' \leq$ `upB` *und*
   - Es gibt keinen Kunden K'' mit Rechnungsbetrag R''. so dass $R' < R''$ und $R'' \leq$ `upB`

   Die Methode gibt -1 zurück, falls kein Kunde die Kriterien erfüllt. Falls mehrere Kunden die Kriterien erfüllen, dann kann eine beliebige ID dieser Kunden zurückgegeben werden.

   Ändern Sie die Methode `process` so, dass für jeden Kunden der Rechnungsbetrag in einer geeigneten Datenstruktur gespeichert wird. Die Methode `query` soll dann diese Datenstruktur verwenden um die Rückgabewerte zu berechnen. Es ist Ihnen überlassen, welche Datenstruktur Sie zur Speicherung der Informationen über die Kunden benutzen. Eine (lineare) Liste ist akzeptabel. Es ist aber wichtig, dass Ihr Programm die Inputdatei nur einmal liest und dass Ihr Programm eine beliebige Anzahl von Kundeneinträgen verarbeiten kann.

In der Datei "BillsTest.java" finden Sie einen einfachen Test, um das Format Ihres Outputs zu testen.