252-0027-00: Einführung in die Programmierung

Übungsblatt 8

Abgabe: 11. November 2025, 23:59

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.

Aufgabe 1: Close Neighbors

Schreiben Sie ein Programm, welches für eine sortierte Folge X von `int`-Werten $(x_1, x_2, ..., x_n)$ und einen `int`-Wert `key` die drei unterschiedlichen Elemente x_a, x_b und x_c aus X zurückgibt, die dem Wert `key` am nächsten sind. Für x_a, x_b und x_c muss gelten, dass $|\texttt{key} - x_a| \leq |\texttt{key} - x_b| \leq |\texttt{key} - x_c| \leq |\texttt{key} - x_i|$ für alle $i \neq a, b, c$ und dass $x_a \neq x_b \neq x_c \neq x_a$. Wenn die drei Werte nicht eindeutig bestimmt sind, dann ist jede Lösung zugelassen, die die obige Bedingung erfüllt.

Beispiele:
Die nächsten Nachbarn für `key == 5` in $(1, 4, 5, 7, 9, 10)$ sind $5, 4, 7$.
Die nächsten Nachbarn für `key == 5` in $(1, 4, 5, 6, 9, 10)$ sind $5, 4, 6$ oder $5, 6, 4$.
Die nächsten Nachbarn für `key == 10` in $(1, 4, 5, 6, 9, 10)$ sind $10, 9, 6$.

Implementieren Sie die Berechnung in der Methode `int[] neighbor(int[] sequence, int key)`, welche sich in der Klasse `Neighbor` befindet. Die Deklaration der Methode ist bereits vorgegeben. Sie können davon ausgehen, dass das Argument `sequence` nicht `null` ist, sortiert ist, nur unterschiedliche Elemente enthält, und mindestens drei Elemente enthält. Denken Sie daran, dass der Wert `key` nicht unbedingt in der Folge X auftritt. Sie dürfen das Eingabearray `sequence` nicht ändern.

In der `main` Methode der Klasse `Neighbor` finden Sie die oberen Beispiele als kleine Tests, welche Beispiel-Aufrufe zur `neighbor`-Methode machen und welche Sie als Grundlage für weitere Tests verwenden können. In der Datei `NeighborTest.java` geben wir die gleichen Tests zusätzlich auch als JUnit Test zur Verfügung. Sie können diese ebenfalls nach belieben ändern. Es wird *nicht*

erwartet, dass Sie für diese Aufgabe den JUnit Test verwenden.

Aufgabe 2: Loop-Invariante

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

1.
```
public int compute(int n) {
      // Precondition:  n >= 0
      int x;
      int res;

      x = 0;
      res = x;

      // Loop Invariante:
      while (x <= n) {
          res = res + x;
          x = x + 1;
      }
      // Postcondition:  res == ((n + 1) * n) / 2
      return res;
  }
```

2.
```
public int compute(int a, int b) {
      // Precondition:  a >= 0
      int x;
      int res;

      x = 0;
      res = b;

      // Loop Invariante:
      while (x < a) {
          res = res - 1;
          x = x + 1;
      }
      // Postcondition:  res == b - a
      return res;
  }
```

Aufgabe 3: Rechnungen

In dieser Aufgabe sollen Sie einen Teil des Systems implementieren, das für den lokalen Stromversorger die Rechnungen erstellt.

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 nur korrekt formatierte Eingabedateien unterstützen. Ein Beispiel einer solchen 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 einen Tarif oder 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 diese Zeile einen Tarif beschreibt.
- n is eine positive ganze Zahl, welche die Anzahl der Kilowattstunden-Intervalle angibt, für die jeweils 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 den Verbrauch in diesem Intervall; für ein i mit $1 \leq i \leq n$, ist l_i also der Verbrauch (in Kilowattstunden), bis zu welchem der Strompreis p_i (in Rappen pro Kilowattstunde) in Rechnung gestellt wird ($l_i > 0$ 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:

1. `Tarif 1 100000 30`
   Es gibt ein Intervall. Für jede Kilowattstunde muss 30 Rappen bezahlt werden.
2. `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.
3. `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 im Jahr 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:

$$ID␣v_{q_1}␣v_{q_2}␣v_{q_3}␣v_{q_4}$$

Hierbei gilt für die Parameter:

- ID ist eine positive ganze Zahl.
- v_{q_1} ist eine ganze Zahl, die den Verbrauch im ersten Quartal in Kilowattstunden angibt ($v_{q_1} \geq 0$).
- v_{q_2} ist eine ganze Zahl, die den Verbrauch im zweiten Quartal in Kilowattstunden angibt ($v_{q_2} \geq 0$).
- v_{q_3} ist eine ganze Zahl, die den Verbrauch im dritten Quartal in Kilowattstunden angibt ($v_{q_3} \geq 0$).
- v_{q_4} ist eine ganze Zahl, die den Verbrauch im vierten Quartal in Kilowattstunden angibt ($v_{q_4} \geq 0$).

Hier ist ein Beispiel für eine Verbrauchbeschreibung:

```
115 0 0 0 2000
```

Der Kunde mit ID 115 hat nur im vierten Quartal Strom verbraucht. Da waren es 2000 Kilowattstunden.

Ein einmal gelesener Tarif wird für alle Kunden angewendet, die nach dieser Tarifinformation in der Eingabedatei erscheinen. Wenn ein neuer Tarif erscheint, dann gilt dieser bis auf Weiteres für die nachfolgenden Kunden. 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.

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

$$ID␣b$$

in den `PrintStream` schreiben, der der `process`-Methode in `output` übergeben wird. ID ist die ID des Kunden (`int`) und b ist eine **ganze** Zahl, die die jeweilige Rechnung für den Jahresverbrauch **in Franken** angibt. (Zuerst muss der Jahresverbrauch 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.
In der Datei "BillsTest.java" finden Sie einen einfachen Test, um das Format Ihres Outputs zu testen.

Tipp: Sie können die Aufgabe ohne weitere Vorgaben implementieren. Wir empfehlen, dass Sie sich überlegen, was sinnvolle Klassen sein könnten und was für Teilaufgaben (die dann als Methode implementiert werden können) zweckmässig sind. Sie können, wie bereits gesehen, das `PrintStream`-Objekt genauso wie `System.out` verwenden und mit `println` eine neue Zeile zur Zieldatei hinzufügen.

Aufgabe 4: Minesweeper (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.

Ihre Aufgabe ist es, die Funktionalität einer Runde des Spiels "Minesweeper" zu implementieren.

Spielregeln: In diesem Spiel hat man ein Gitter aus Zellen mit zufällig verteilten Minen, und das Ziel ist es, alle leeren Zellen aufzudecken.
Es gibt drei Verhaltensweisen nach dem Aufdecken einer Zelle (ein Spielzug):

- Beim Aufdecken einer leeren Zelle (linkes Bild) wird eine Zahl angezeigt, die angibt, wie viele Minen um die Zelle herum liegen.
- Das Aufdecken einer leeren Zelle ohne Minen (mittleres Bild) deckt alle benachbarten Zellen rekursiv auf (d.h. "klickt" auf alle acht benachbarten Zellen).
- Das Aufdecken einer Mine (rechtes Bild) deckt alle Minen auf und beendet das Spiel sofort.

Beispiel:

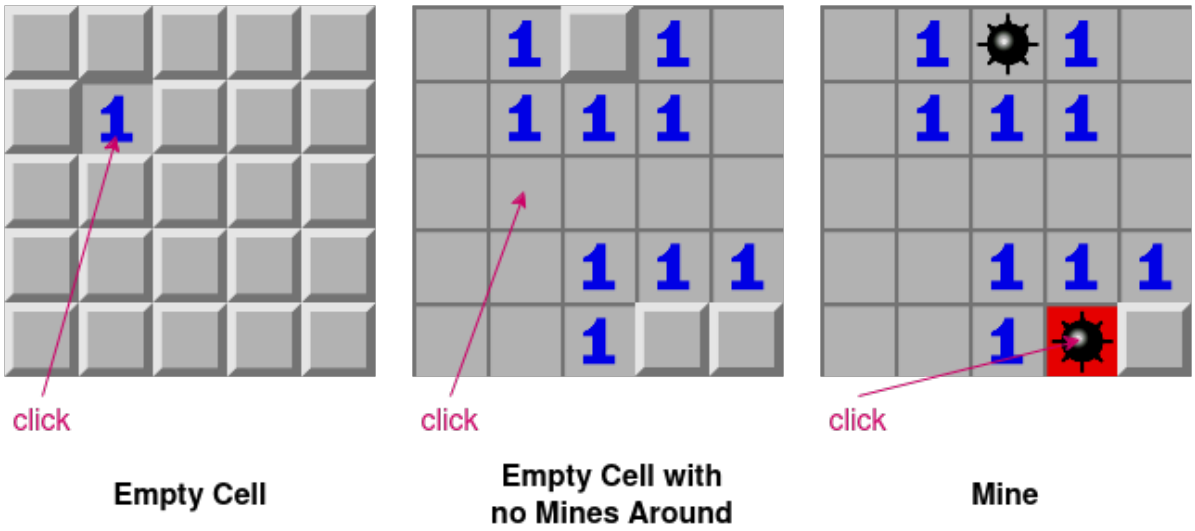

Abbildung 1: Spielbretter nach dem ersten, zweiten und dritten Klick von links nach rechts.

Details zur Implementierung: Das Spielbrett wird durch eine int-Matrix dargestellt. Jede Zelle des Brettes kann einen int-Wert im Bereich [-2, 9] haben:

- -2: die Zelle ist eine nicht aufgedeckte Mine.
- -1: die Zelle ist eine nicht aufgedeckte leere Zelle.
- 0 bis 8: Das Feld ist aufgedeckt und hat die entsprechende Anzahl von Minen um sich herum (d.h. darüber, darunter, links, rechts und alle 4 Diagonalen).
- 9: das Feld ist eine aufgedeckte Mine.

Aufgabenbeschreibung: Ihre Aufgabe ist es, die Funktionalität einer Spielrunde in 'Minesweeper.java' zu implementieren. Sie ist in drei Funktionen unterteilt. Sie können davon ausgehen, dass das Spielbrett `board` rechteckig ist, immer mindestens eine Zeile und eine Spalte hat und Zahlen von -2 bis und mit 9 enthält. Sie dürfen dabei annehmen, dass sich das Spielbrett in einem Zustand befindet, welcher von einem komplett zugedeckten Spielbrett durch legale Spielzüge erreicht werden kann, ohne das Spiel zu beenden. Für die Unteraufgaben 2 und 3 dürfen Sie zusätzlich annehmen, dass die Position (`row`,`col`) jeweils innerhalb des Spielbretts liegt. Für ein Spielbrett mit h Zeilen und b Spalten dürfen Sie also unter anderem folgende Bedingungen annehmen:

- $h \geq 1$ und $b \geq 1$
- $0 \leq row < h$ und $0 \leq col < b$

Aufgabe 1: Implementieren Sie die Methode `void revealAllMines(int[][] board)`, die alle Minen auf dem Spielbrett aufdeckt. Beispiel mit einem 5×5-Spielbrett:

```
board1 = {  {-1, -1, -2, -1, -1}, board1after = {  {-1, -1, 9, -1, -1},
            {-1, -1, -1, -1, -1},                   {-1, -1, -1, -1, -1},
            {-1, -1, -1, -1, -1},                   {-1, -1, -1, -1, -1},
            {-1, -1, -1, -1, -1},                   {-1, -1, -1, -1, -1},
            {-1, -1, -1, -2, -1} };                 {-1, -1, -1, 9, -1} };
```

- `revealAllMines(board1)` verändert `board1` so, dass es äquivalent zu `board1after` ist.

Aufgabe 2: Implementieren Sie die Methode `int computeMineCount(int[][] board, int row, int col)`, das die Anzahl der Minen um eine Zelle an der Position (`row`, `col`) zurückgibt, oder '9' wenn die Zelle eine Mine ist. Beispiel:

```
board2 = {  {-1, -1, -2, -1, -1},
            {-1, -1, -1, -1, -1},
            {-1, -1, -1, -1, -1},
            {-1, -1, -1, -1, -1},
            {-1, -1, -1, -2, -1} };
```

- `computeMineCount(int[][] board2, 1, 1)` gibt 1 zurück.
- `computeMineCount(int[][] board2, 2, 1)` gibt 0 zurück.
- `computeMineCount(int[][] board2, 4, 3)` gibt 9 zurück.

Aufgabe 3: Implementieren Sie die Methode `void gameTurn(int[][] board, int row, int col)`, die einen kompletten Spielzug ausführt und das Spielbrett entsprechend aktualisiert. Gegeben ist ein Klick auf die Zelle an der Position (`row`, `col`). Beispiel:

```
board3 = {  {-1, -1, -2, -1, -1}, board4 = {  {-1, -1, -2, -1, -1},
            {-1, -1, -1, -1, -1},             {-1,  1, -1, -1, -1},
            {-1, -1, -1, -1, -1},             {-1, -1, -1, -1, -1},
            {-1, -1, -1, -1, -1},             {-1, -1, -1, -1, -1},
            {-1, -1, -1, -2, -1} };           {-1, -1, -1, -2, -1} };

board5 = {  {0,  1, -2,  1,  0}, board6 = {  {0,  1,  9,  1,  0},
            {0,  1,  1,  1,  0},             {0,  1,  1,  1,  0},
            {0,  0,  0,  0,  0},             {0,  0,  0,  0,  0},
            {0,  0,  1,  1,  1},             {0,  0,  1,  1,  1},
            {0,  0,  1, -2, -1} };           {0,  0,  1,  9, -1} };
```

(Blau markiert sind jeweils die Änderungen am Spielbrett durch den vorhergehenden Spielzug zur Veranschaulichung)

- `gameTurn(board3, 1, 1)` verändert `board3` so, dass es äquivalent zu `board4` ist.
- `gameTurn(board4, 2, 1)` verändert `board4` so, dass es äquivalent zu `board5` ist.
- `gameTurn(board5, 4, 3)` verändert `board5` so, dass es äquivalent zu `board6` ist.

Tipp: Implementieren Sie `gameTurn` rekursiv und überlegen Sie sich geeignete Abbruchbedingungen.