

252-0027-00: Einführung in die Programmierung

Übungsblatt 5

Abgabe: 21. Oktober 2025, 23:59

Ab dieser Übung gibt es ein IntelliJ-Projekt für die Bonusaufgabe (Ordner *uXX-bonus* in Ihrem Repository, z.B. *u05-bonus* diese Woche) und ein IntelliJ-Projekt für die restlichen Aufgaben (Ordner *u05*). Checken Sie mit IntelliJ wie bisher die neue Übungs-Vorlage aus. Beachten Sie, dass Sie mehrere unabhängige Programme im bonusunabhängigen IntelliJ-Projekt haben werden. Bevor Sie ein Programm starten, achten Sie deshalb darauf, dass Sie die richtige Datei ausgewählt oder im Editor geöffnet haben. *Vergessen Sie nicht, Ihren Programmcode zu kommentieren!*

Ab dieser Woche, 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 u05 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: Sieb des Eratosthenes

Schreiben Sie ein Programm "Sieb.java", das eine Zahl *limit* einliest und die Anzahl der Primzahlen, die grösser als 1 und kleiner oder gleich dem *limit* sind, ausgibt. Dazu ermitteln Sie in einem ersten Schritt alle Primzahlen, die kleiner oder gleich *limit* sind. Dieses Teilproblem können Sie mit dem [Sieb des Eratosthenes](#) lösen. Das Sieb des Eratosthenes findet Primzahlen bis n . Man betrachtet alle Zahlen von 2 bis n und streicht zuerst alle Vielfachen der ersten Zahl (2). Dann geht man zur nächsten ungestrichenen Zahl (3) und wiederholt das Streichen ihrer Vielfachen. Das macht man, bis man dies für alle Zahlen gemacht hat. Sie können ein Boolean-Array verwenden, um zu speichern, welche Zahlen Primzahlen sind und welche nicht. Übrig bleiben die Primzahlen. Danach können Sie die Anzahl der gefundenen Primzahlen anhand dieses Arrays bestimmen.

Beispiel: Für $limit = 13$ sollte Ihr Programm 6 ausgeben (Primzahlen: 2, 3, 5, 7, 11, 13).

Hinweis: Es ist nicht zwingend nötig von 2 bis n zu gehen. Von 2 bis $\sqrt{n}$ zu gehen reicht bereits aus, da eine Zahl $\leq n$ nicht einen Teiler grösser als $\sqrt{n}$ ausser sich selbst haben kann.

Aufgabe 2: Arrays

In dieser Aufgabe implementieren Sie Methoden, welche Arrays verwenden.

1. Implementieren Sie die Methode `ArrayUtil.zeroInsert(int[] x)` in der Datei `"ArrayUtil.java"`. Die Methode nimmt ein Array `x` als Argument und gibt ein Array zurück. Das zurückgegebene Array soll die gleichen Werte wie `x` haben, ausser: Wenn eine positive Zahl direkt auf eine negative Zahl folgt oder wenn eine negative Zahl direkt auf eine positive Zahl folgt, dann wird dazwischen eine 0 eingefügt.

Beispiele:

- Wenn `x` gleich `[3, 4, 5]` ist, dann wird `[3, 4, 5]` zurückgegeben.
- Wenn `x` gleich `[3, 0, -5]` ist, dann wird `[3, 0, -5]` zurückgegeben.
- Wenn `x` gleich `[-3, 4, 6, 9, -8]` ist, dann wird `[-3, 0, 4, 6, 9, 0, -8]` zurückgegeben.

Versuchen Sie, die Methode rekursiv zu implementieren.

2. Implementieren Sie die Methode `ArrayUtil.tenFollows(int[] x, int index)`. Die Methode gibt einen Boolean zurück. Die Methode soll `true` zurückgeben, wenn im Array `x` ab Index `index` der zehnfache Wert einer Zahl `n` direkt der Zahl `n` folgt. Dies muss nur für das erste Auftreten der Zahl `n` ab Index `index` im Array `x` geprüft werden. Ansonsten soll die Methode `false` zurückgeben.

Beispiele:

- `tenFollows([1, 2, 20], 0)` gibt `true` zurück.
- `tenFollows([1, 2, 7, 20], 0)` gibt `false` zurück.
- `tenFollows([3, 30], 0)` gibt `true` zurück.
- `tenFollows([3], 0)` gibt `false` zurück.
- `tenFollows([1, 2, 20, 5], 1)` gibt `true` zurück.
- `tenFollows([1, 2, 20, 5], 2)` gibt `false` zurück.

Die `main` Methode in `ArrayUtil` gibt die oben genannten Beispielaufrufe sowie das entsprechende Ergebnis der jeweiligen Methode aus. Hiermit können Sie überprüfen, ob Ihre Implementierungen die richtigen Ergebnisse zurückliefern. In `"ArrayUtilTest.java"` im Ordner `"test"` in der Übungsvorlage finden Sie zusätzlich einige Unit-Tests für beide Methoden (für eine detaillierte Beschreibung zu automatisiertem Testen und der Ausführung solcher Tests siehe Aufgabe 4). Sie können die `main` Methode und die Tests beliebig abändern und/oder mit Ihren eigenen Inputs erweitern.

Aufgabe 3: 2D Arrays

Gegeben einer Matrix M , prüfen Sie zuerst, ob diese eine $n \times n$ Matrix ist, deren Elemente positive ganze Zahlen sind. Danach prüfen Sie, ob zusätzlich alle Zahlen kleiner gleich n^2 sind. Somit gilt nun $0 < m_{i,j} \leq n^2$. Prüfen Sie ebenfalls, ob die Elemente der Matrix jeweils genau einmal vorkommen, sprich ob $m_{x,y} = m_{p,q} \Rightarrow (x = p) \wedge (y = q)$ gilt. Wir sagen, dass die Matrix M *perfekt* ist, wenn zusätzlich alle Zeilensummen und Spaltensummen gleich sind (also $\sum_{k=0}^{n-1} m_{i,k} = \sum_{k=0}^{n-1} m_{j,k}$ für alle i, j und $\sum_{k=0}^{n-1} m_{k,i} = \sum_{k=0}^{n-1} m_{k,j}$ für alle i, j mit $0 \leq i, j < n$).

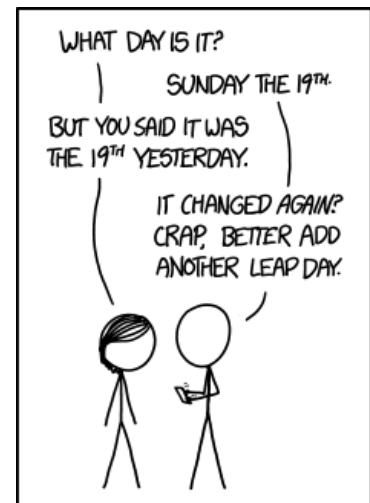
Vervollständigen Sie die Methode `boolean checkMatrix(int[] [] m)` von der Klasse `Matrix`, so dass diese Methode `true` zurückgibt, wenn die Input Matrix *perfekt* ist, und `false` sonst. Sie können davon ausgehen, dass der Parameter `m` nicht `null` ist. Alle anderen Eigenschaften müssen Sie selber testen. Eine Matrix ist nur perfekt, wenn alle genannten Eigenschaften gelten.

Testen Sie Ihr Programm ausgiebig - am besten mit JUnit - und pushen Sie die Lösung vor dem Abgabetermin. Wir haben Ihnen einen JUnit Test in der Klasse `MatrixTest` bereits erstellt.

Aufgabe 4: Testen mit JUnit

Bisher haben Sie Ihre Programme "von Hand" getestet, das heisst, Sie haben das Programm mehrmals ausgeführt und verschiedene Eingaben ausprobiert. In dieser Aufgabe lernen Sie das Konzept der *Testautomatisierung* kennen. Das Testen eines Programms mit verschiedenen Eingaben wird dabei wiederum von einem Programm übernommen. Dabei kann ein Programm als Ganzes oder es können einzelne Teile davon separat getestet werden. Automatische Tests haben den Vorteil, dass sie nur einmal geschrieben werden müssen und danach bei jeder Änderung des Programms ohne Aufwand ausgeführt werden können.

In der Übungsvorlage finden Sie das Programm "PerpetualCalendar.java", welches für jedes (gültige) Datum den Wochentag berechnet. Leider enthält das Programm noch Fehler, die aber vom Compiler nicht erkannt werden. Das Programm ist also ein gültiges Java-Programm, aber es verhält sich nicht so, wie der Autor es beabsichtigt hat. Glücklicherweise hat der Autor Tests geschrieben, welche die Korrektheit der Teile des Programms überprüfen. Sie finden diese in der Datei "PerpetualCalendarTest.java", welche sich im Ordner "test" befindet. Ihre Aufgabe ist nun, mithilfe dieser Tests die Fehler im Programm zu finden und zu beheben.



MY NEW SIMPLIFIED CALENDAR SYSTEM ASSUMES THE DATE NEVER CHANGES, THEN CORRECTS ANY DRIFT VIA LEAP DAYS.

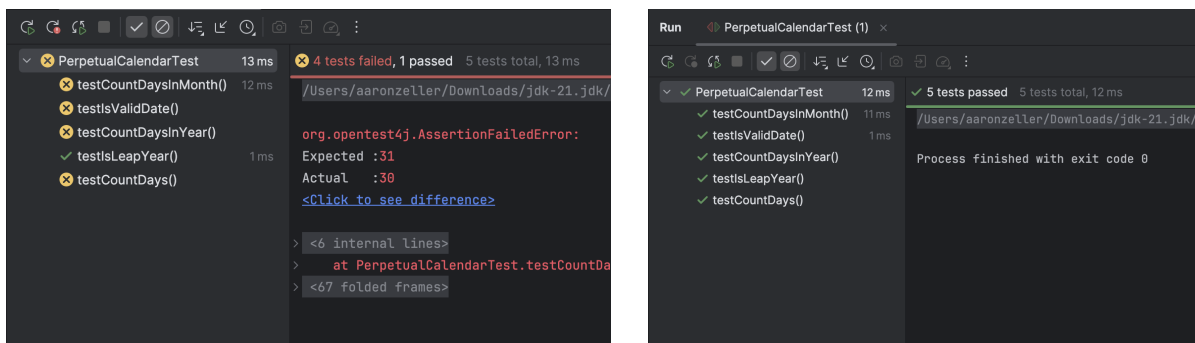
xkcd: PermaCal by Randall Munroe

(CC BY-NC 2.5)

- Öffnen Sie zuerst das Programm "PerpetualCalendar.java" und führen Sie es aus. Geben Sie ein paar Daten ein und überprüfen Sie die Ausgabe. Sie werden feststellen, dass das Programm noch nicht korrekt funktioniert.

- b) Öffnen Sie nun die Datei “PerpetualCalendarTest.java” und drücken Sie , gefolgt von `Run 'PerpetualCalendarTest'`, um alle Tests in dieser Datei auszuführen. Es öffnet sich die *JUnit*-Ansicht und die Tests werden ausgeführt. Im oberen Bereich der Ansicht sehen Sie “4 tests failed, 1 passed 5 Tests total”. Es sind also 5 Tests ausgeführt worden, von denen 4 fehlgeschlagen sind.
- c) Im mittleren Bereich der Ansicht sehen Sie eine Auflistung der Tests. Für jede Hilfsmethode im Programm gibt es einen Test. Im Moment scheint nur die `isLeapYear()`-Methode korrekt zu sein. Klicken Sie auf einen fehlgeschlagenen Test, um im unteren Bereich der Ansicht den Fehler anzuzeigen. Beim Test `testCountDaysInYear` steht “expected:<365> but was:<366>”.
- d) Doppelklicken Sie auf `testCountDaysInYear`, um zum Code für diesen Test zu springen. Sie sehen, dass die Methode `countDaysInYear()` der Klasse `PerpetualCalendar` mit dem Argument 1900 aufgerufen wird. Ausserdem wird die spezielle Methode `assertEquals()` aufgerufen, welche das Resultat von `countDaysInYear()` mit dem erwarteten Wert 365 vergleicht. Dass der Test auf dieser Zeile fehlschlägt, bedeutet also, dass `countDaysInYear(1900)` nicht den erwarteten Wert 365 zurückgibt (sondern 366).
- In anderen Tests werden statt `assertEquals()` die Methoden `assertTrue()` und `assertFalse()` verwendet. Diese Methoden überprüfen, ob der boolesche Ausdruck, der als zweites Argument übergeben wird, `true` bzw. `false` ist.
- e) Öffnen Sie wieder das Programm “PerpetualCalendar.java” und gehen Sie zur Methode `countDaysInYear()`. Finden und beheben Sie den Fehler in dieser Methode und führen Sie die Tests erneut aus. Wenn Sie den Fehler korrekt behoben haben, sollte der Test `testCountDaysInYear` ohne Fehler durchlaufen und oben sollte “3 tests failed, 2 passed 5 Tests total” stehen.
- f) Finden Sie nun die restlichen Fehler, indem Sie den fehlschlagenden Tests nachgehen. Wenn Sie alle Fehler behoben haben, zeigt die *JUnit*-Ansicht einen grünen Balken an.

Keep the bar green to keep the code clean!



Aufgabe 5: Matching Numbers

Implementieren Sie die Methode `Match.matchNumber(long A, int M)`. Die Methode soll für eine Zahl A und eine nicht-negative drei-stellige Zahl M die Position von M in A zurückgeben. Sei M eine Zahl mit den Ziffern $M_2M_1M_0$ (das heisst, es gilt $M = M_0 + 10 \cdot M_1 + 100 \cdot M_2$), wobei jede Ziffer 0 sein kann. Zusätzlich sei A eine Zahl, sodass A_i die i -te Ziffer von A ist (das heisst, es gilt für (den Betrag von A) $|A| = \sum_i 10^i \cdot A_i$), wobei A beliebig viele führende Nullen hat. Die Position von M in A ist die kleinste Zahl j , sodass $A_j = M_0$ und $A_{j+1} = M_1$ und $A_{j+2} = M_2$ gilt. Die Methode soll -1 zurückgeben, falls es kein solches j gibt.

Beispiele:

`matchNumber(32857890, 789)` soll 1 zurückgeben.

`matchNumber(37897890, 789)` soll 1 zurückgeben.

`matchNumber(1800765, 7)` soll 2 zurückgeben.

`matchNumber(1800765, 8)` soll -1 zurückgeben (die drei Ziffern von 8 sind 008).

`matchNumber(75, 7)` soll 1 zurückgeben (da 007 an Position 1 von 0075 ist).

Implementieren Sie die Berechnung in der Methode `int matchNumber(long A, int M)`, welche sich in der Klasse `Match` befindet. Die Deklaration der Methode ist bereits vorgegeben. Sie können davon ausgehen, dass $0 \leq M < 1000$ gilt.

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

Tipp: Die Methode `Integer.toString(int i)` wandelt einen Integer in einen String um. Alternativ können Sie auch `i + ""` benutzen.

Aufgabe 6: Substring-Counter (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](#).

Gegeben sind ein String `str` und ein Substring `sub`. Implementieren Sie die folgenden Methoden in der Klasse `Counting`. Die Deklarationen der Methoden sind bereits vorhanden.

1. `boolean containsSubstringAt(String str, int position, String sub)`: Diese Methode soll überprüfen, ob `sub` im String `str` an der Position `position` vorkommt. Zum Beispiel:

- `containsSubstringAt("abcd", 0, "ab")`: `true`
- `containsSubstringAt("abcd", 1, "ab")`: `false`
- `containsSubstringAt("abcd", 4, "ab")`: `false`
- `containsSubstringAt("abcd", -1, "ab")`: `false`

Hinweis: Achten Sie darauf, dass die Methode auch `false` zurückgeben soll, wenn `position` kein valider Index des Strings `str` ist (also wenn `position` negativ oder zu gross ist).

2. `int countSubstrings(String str, String sub)`: Diese Methode soll zählen, wie oft der String `sub` als Substring in `str` vorkommt - die Substrings können sich hier überlappen. Zum Beispiel:

- `countSubstrings("aaa", "aa")`: `2`
- `countSubstrings("aaaa", "aa")`: `3`
- `countSubstrings("12341234123", "123412")`: `2`

3. `int countDisjointSubstrings(String str, String sub)`: Diese Methode soll zählen, wie oft `sub` *ohne Überlappungen* in `str` vorkommt. Das erste Vorkommen (von links) eines Substrings `sub` trägt also zum Rückgabewert der Methode bei. Jedes weitere Vorkommen trägt nur zum Rückgabewert bei, wenn sich dieses nicht mit einem beitragenden Vorkommen weiter links in `str` überschneidet. Zum Beispiel:

- `countDisjointSubstrings("abdcdba", "abc")`: `1`
- `countDisjointSubstrings("abdcdba", "c")`: `2`
- `countDisjointSubstrings("0101010", "010")`: `2`

Die `main`-Methode der Klasse `Counting` verwendet die obigen Beispiele als kleine Tests. Sie führt Beispielaufrufe dieser drei Methoden durch, die Sie als Grundlage für weitere Tests verwenden können. In der Datei `CountingTest.java` werden die gleichen Tests zusätzlich als JUnit-Tests bereitgestellt. Diese können Sie nach Belieben anpassen. Es wird nicht erwartet, dass Sie für diese Aufgabe die JUnit-Tests verwenden oder neue hinzufügen, aber wir empfehlen es, da es Ihnen hilft, Ihre Lösung selbst auf Richtigkeit zu überprüfen.