Programmierkurs
Winter 2025/26

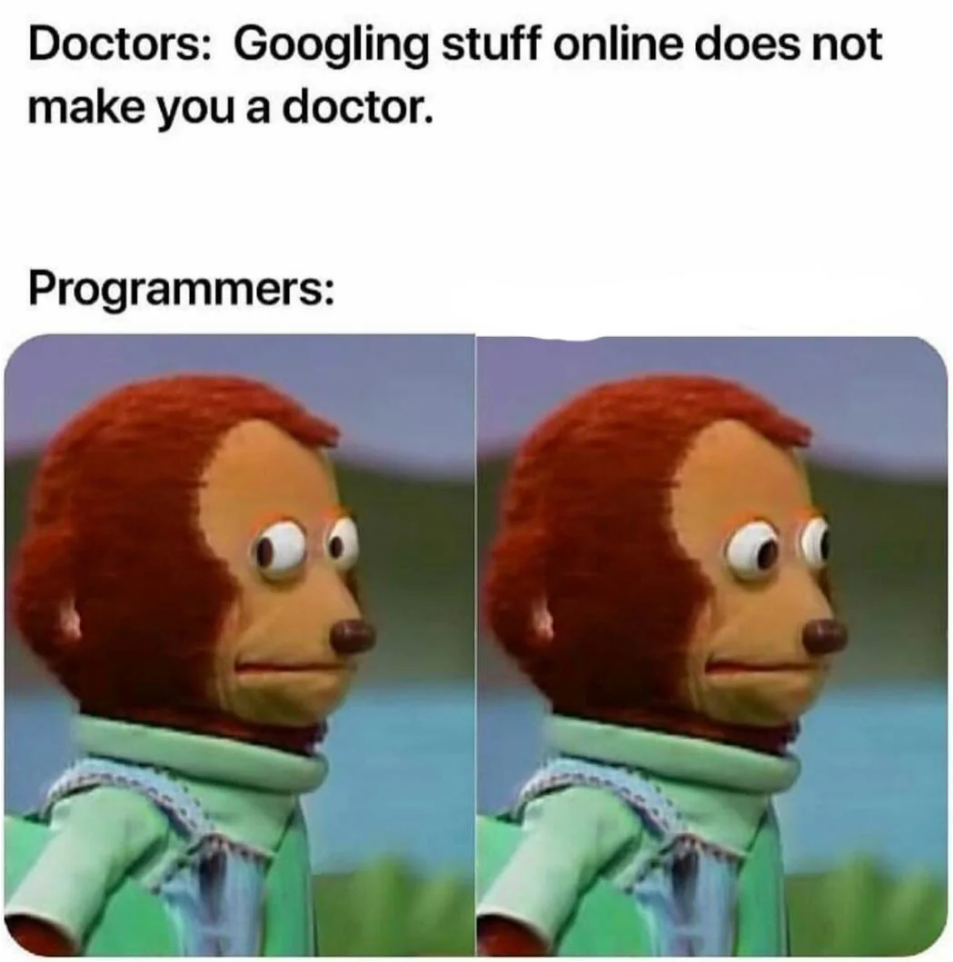
Department Mathematik / Informatik
Dr. Stefan Zellmann
Simon Wolf

Übungen 12.12-19.12.2025
Blatt 9

Abgabe der Aufgaben bis spätestens **Freitag, 19.12.2025 um 23:59 Uhr** via git.
Besprechung der Aufgaben in der Woche vom 05.01.2025.

Hinweise:

- Es gelten diese Modifikatoren-Regeln (es sei denn, es steht in der Aufgabe explizit anders):
  - Klassen: `public`
  - Objektattribute: `private`
  - Objektmethoden: `public` (insbesondere Konstruktoren)
- Konstruktoren: Implementieren Sie ausschließlich die von uns explizit verlangten Konstruktoren, und keine anderen.



Präsenzaufgabe 1 [API anwenden am Beispiel `Random`]

Gehen Sie auf folgende Internetseite:
`https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Random.html`
Dort ist die Klasse **`Random`** beschrieben, und auch die beiden Methoden **`nextInt`** (mit unterschiedlichen Signaturen). Verwenden Sie diese Methoden für das Lösen der folgenden Aufgaben.
Hinweis: Mit zufällig meinen wir tatsächlich, dass das Ergebnis nicht vorhersehbar ist. Das bedeutet, dass, wenn wir eine Methode mehrfach aufrufen, das Resultat nicht immer gleich sein darf (es sei denn es handelt sich um einen Randfall, in dem es nicht anders geht).

Die Methode `randomArray`

Schreiben Sie die Methode **`randomArray`**.

- Modifikatoren: **`public static`**
- Rückgabetyp: **`int`**-Array
- Name: **`randomArray`**
- Übergabeparameter: **`int x`**

Die Methode **`randomArray`** erzeugt ein **`int`**-Array der Länge **`x`**. Darin werden nacheinander zufällige Zahlen, die aus dem gesamten Wertebereich der Klasse Integer stammen können, in jede der **`x`** Zellen eingefügt. Dieses Array wird zurückgegeben. Sie dürfen davon ausgehen, dass **`x`** mindestens den Wert 1 hat.

- Beispiel: **`randomArray(2)`** gibt ein Array mit 2 Zellen zurück, in dem die Zahl 6 und die Zahl -1122 gespeichert ist. Beim nächsten Mal könnte es mit -6354 und 33345 gefüllt sein.

Die Methode `randomArray2`

Schreiben Sie die Methode **`randomArray2`**.

- Modifikatoren: **`public static`**
- Rückgabetyp: **`int`**-Array
- Name: **`randomArray2`**
- Übergabeparameter: **`int x`**

Die Methode **`randomArray2`** erzeugt ein **`int`**-Array der Länge **`x`**. Darin werden nacheinander zufällige Zahlen in jede der **`x`** Zellen eingefügt. Dabei müssen die zufälligen Zahlen zwischen 0 (inklusive) und **`x`** (exklusive) sein. Sie dürfen davon ausgehen, dass **`x`** mindestens den Wert 1 hat. Dieses Array wird zurückgegeben.

- Beispiel: Wird **`randomArray2(2)`** aufgerufen, dann wird ein 2-Zellen Array zurückgegeben, das zufällig mit den Zahlen 0 und 1 gefüllt ist. Das kann z.B. auch $\{0, 0\}$ sein.

Die Methode `randomArray3`

Schreiben Sie die Methode **`randomArray3`**.

- Modifikatoren: **`public static`**
- Rückgabetyp: **`int`**-Array
- Name: **`randomArray3`**
- Übergabeparameter: keine

Die Methode **`randomArray3`** erzeugt ein neues **`int`**-Array mit 6 Zellen. Diese Zellen werden mit zufälligen Zahlen zwischen 55 (inklusive) und 75 (inklusive) belegt.

- Beispiel: Eine mögliche Belegung ist **`{75,70,60,60,64,63}`**.

Die Methode `randomArray4`

Schreiben Sie die Methode **`randomArray4`**.

- Modifikatoren: **`public static`**
- Rückgabetyp: **`int`**-Array
- Name: **`randomArray4`**
- Übergabeparameter: **`int n`**

Diese Methode erzeugt ein **`int`**-Array mit **`n+1`** Zellen. Weiterhin werden **`n`** zufällige **`int`**-Zahlen zwischen 1 (inklusive) und 50 (inklusive) erzeugt und in die ersten **`n`** Zellen eingespeichert. Anschließend werden alle gespeicherten Zahlen nacheinander addiert, mittels Division der Mittelwert gebildet und in der letzten Zelle dieser Mittelwert, auf die nächste, ganze Zahl abgerundet, gespeichert. Zum Beispiel ist bei **`n`**-Wert 3 und den zufälligen Zahlen 2,1,2 der in der letzten Zelle gespeicherte Wert eine 1. Das entsprechend gefüllte Array wird von der Methode zurückgegeben. Sie dürfen davon ausgehen, dass **`n`** mindetens den Wert 1 hat.

Präsenzaufgabe 2 [Minesweeper]

Betrachten Sie folgendes Gitter:

	0	1	2
0			
1	b		a
2			

Mit **`[i][j]`** bezeichnen wir die Zelle in Zeile i und Spalte j für $i, j \in \{0, 1, 2\}$.

Beispiel: a steht in **`[1][2]`**, b steht in **`[1][0]`**.

Wir nennen `[i][j]` *benachbart* zu `[s][t]`, falls die Zelle `[s][t]` die Zelle `[i][j]` in mindestens einem Punkt berührt und die Zelle `[s][t]` und die Zelle `[i][j]` nicht identisch sind.

Beispiel: Zu `[1][2]` sind `[0][2]`, `[0][1]`, `[1][1]`, `[2][1]`, `[2][2]` benachbart.

Schreiben Sie ein Programm, welches folgende Anforderungen erfüllt:

- Erzeugen Sie in der **`main`**-Klasse ein 3 x 3 - `int`-Array **`grid`** (**`static`**, **`public`**). Belegen Sie alle Zellen mit dem Wert 0, bis auf drei Zellen: Suchen Sie sich drei Zellen aus, denen Sie den Wert `-1` zuweisen.
- Für die Zellen, welche den Wert 0 speichern: Speichern Sie in jeder solchen Zelle, nennen wir sie mal z, die Anzahl der Zellen, die zu z benachbart sind und gleichzeitig eine `-1` enthalten. Schreiben Sie dafür eine Methode in der **`main`**-Klasse. Überlegen Sie sich, was ein sinnvoller Rückgabetyp sein könnte und welche Übergabeparameter hier sinnvoll wären.
- Die Ausgabe auf die Konsole soll so gestaltet sein, dass der Inhalt aller 9 Zellen auf der Konsole zu sehen ist und die Zellen des 3 x 3 - Gitters (zumindest ungefähr) nachbildet. Schreiben Sie eine entsprechende Methode.
- Implementieren Sie das Programm so, dass NUR durch die Änderung der Zellen, in denen Sie eine `-1` rein setzen, sich alle anderen Werte automatisch (und natürlich richtig) anpassen. Das bedeutet: Die Ausgabe soll sich anpassen, egal, wo die `-1` gesetzt werden. Wenn Sie jetzt das Array (bzw. die Zellen mit `-1`) ändern, und das Programm erneut starten, so erscheint die korrekte Ausgabe auf der Konsole.

Hier ein Beispiel für eine gültige Belegung. Ungefähr so sollte auch die Ausgabe Ihres Programms strukturiert sein. Die mit `[0][1]`, `[1][2]` und `[2][2]` bezeichneten Zellen wurden zu Anfang mit `-1` belegt:

```
 1  -1   2
 1   3  -1
 0   2  -1
```

Hausaufgabe 1 [ArrayList - Patient] **30 Punkte**

Erstellen Sie im Repository ein Package `h1` und fügen Sie diesem Package die Klasse `H1_main` hinzu, welche die `main`-Methode enthält. Stellen Sie sich vor, Sie müssen in einer Notfallpraxis die Patienten, die bereits priorisiert sind, der Reihe nach aufrufen und auf die diensthabenden Ärzt:innen verteilen. Dabei gibt es die Klasse `Patient`, die einer Person entspricht, welche die Notaufnahme aufsucht, und `PrioListe`, welche die Patienten verwaltet, welche im Wartezimmer darauf warten, aufgerufen zu werden.

- Schreiben Sie die Klasse `Patient`. `Patient` hat als Objekt-Attribute das `String`-Attribut `name` und das `int`-Attribut `prio`. Der Konstruktor `Patient(String name, int prio)`, setzt die beiden Attribute entsprechend.
- Schreiben Sie die Klasse `PrioListe`.
  - `PrioListe` hat das `ArrayList`-Attribut `myList`, welches `Patient`-Objekte aufnimmt. Initialisieren Sie `myList` als leere `ArrayList` für `Patient`-Objekte.
  - `PrioListe` besitzt die Methoden `addPatient`, `getNextPatient` und `getPosition`.
- Die Methode `addPatient`:
  - Modifikatoren: `public`
  - Rückgabetyp: `void`
  - Name: `addPatient`
  - Übergabeparameter: `Patient p`

  Diese Methode fügt die übergebene Person `p` in `myList` ein - abhängig vom Wert von `prio`. Dabei ist zu beachten, dass je kleiner `prio` ist, desto weiter vorne ist `p` in `myList` einzufügen. Beim Einsortieren von `p` in `myList` dürfen Sie also davon ausgehen, dass `myList` bereits sortiert ist (der:die Patient:in mit der kleinsten Priorität ist in der ersten Zelle zu finden, z.B.), und Sie nun `p` lediglich an die richtige Stelle einsortieren müssen. Sie dürfen davon ausgehen, dass die Prioritäten stets paarweise verschieden sind (es gibt nicht zwei Personen mit der gleichen Priorität). Ist `myList` leer, wird `p` in die erste Zelle von `myList` eingefügt.
  - Beispiel: Die `Patient`-Objekte A, B und C haben jeweils die Priorität 1, 10 und 5. Alle drei Personen sind schon in `myList` eingefügt, also in der Reihenfolge A, C und B. Nun gibt es eine weitere Person D mit Priorität 7, die mit `addPatient` hinzugefügt wird. Dann ist die Reihenfolge der `Patient`-Objekte in `myList` anschließend A, C, D und B.
- Die Methode `getNextPatient`.
  - Modifikatoren: `public`
  - Rückgabetyp: `Patient`
  - Name: `getNextPatient`

  - Übergabeparameter: keine

  Die Methode gibt das **`Patient`**-Objekt aus **`myList`** zurück, das ganz vorne in der Liste steht, und löscht dieses dann aus **`myList`**.

  - In unserem Beispiel würde A zurückgegeben werden; **`myList`** wäre nach Abschluss der Methode dann mit C, D und B gefüllt.

- Die Methode **`getPosition`**.

  - Modifikatoren: **`public`**
  - Rückgabetyp: **`int`**
  - Name: **`getPosition`**
  - Übergabeparameter: **`Patient p`**

  Diese Methode gibt die Position des übergebenen Objekts (als Zellenindex) in **`myList`** zurück. Ist das übergebene **`Patient`**-Objekt nicht in der Liste enthalten, wird -1 zurück gegeben.

  - Am obigen Beispiel: Ist **`myList`** mit C, D und B gefüllt, gibt **`getPosition(D)`** den Wert 1 zurück.

Hausaufgabe 2 [ArrayList - Bus] **70 Punkte**

Erstellen Sie im Repository ein Package **`h2`** und fügen Sie diesem Package die Klasse **`H2_main`** hinzu, welche die **`main`**-Methode enthält.

Schreiben Sie die Klasse **`Passenger`**.

- **`Passenger`** besitzt das **`String`**-Attribut **`name`** (speichert Namen des Passagiers),
- Weiterhin hat **`Passenger`** das **`int`**-Attribut **`planned`** (für die Anzahl der Stationen, die gefahren werden sollen, siehe Beispiel). Sie können davon ausgehen, dass jeder Passagier mindestens eine Station fahren will.
- Außerdem besitzt **`Passenger`** das **`int`**-Attribut **`visited`** (für die bereits angefahrenen Stationen, siehe Beispiel).
- Zudem hat **`Passenger`** das **`boolean`**-Attribut **`ticket`** (ist genau dann **`true`**, wenn der Passagier einen Fahrschein besitzt).
- Der Konstruktor **`Passenger(String name, int planned, boolean ticket)`**, der die entsprechenden Attribute setzt. Außerdem wird im Konstruktor der Wert von **`visited`** auf 0 gesetzt.

Beispiel:
Ein Passagier steigt bei A ein und möchte über B nach C fahren. Bei Station A hat **`planned`** den Wert 2 (Stationen B und C) und **`visited`** den Wert 0. Hält der Bus bei B, so ist **`planned`**

noch immer 2, **visited** bei 1 (da B bereits angefahren wurde). Ein Passagier steigt dann aus, wenn **planned** und **visited** übereinstimmen.

Schreiben Sie die Klasse **Bus**. Diese steht für einen Bus, der Passagiere mitnehmen kann und Haltestellen anfährt.

- **Bus** besitzt eine **Passenger**-ArrayList **passengers**, diese steht für die Passagierliste, also die **Passenger**-Objekte, die sich gerade im Bus befinden.
- **Bus** besitzt einen öffentlichen Konstruktor ohne Parameter, der **passengers** mit einer leeren ArrayList initialisiert.
- **Bus** besitzt die öffentliche **void**-Methode **enterBus(Passenger p)**, welche **p** an das Ende von **passengers** hinzufügt.
- **Bus** besitzt die private **void**-Methode **exitBus()**, welche für alle Passagiere prüft, ob diese jetzt aussteigen (also ob die Anzahl der angefahrenen Haltestellen mit der der geplanten Haltestellen übereinstimmt). Muss ein Passagier jetzt aussteigen, wird er aus **passengers** gelöscht.
- **Bus** besitzt die öffentliche **void**-Methode **nextStop(Passenger[] boarding)**. Diese simuliert, dass der Bus die nächste Haltestelle angefahren hat: bei allen im Bus befindlichen Passagieren werden die Werte von **visited** angepasst. Anschließend wird **exitBus()** aufgerufen. Danach sollen alle Passagiere, die in **boarding** enthalten sind, an das Ende von **passengers** in der in **boarding** vorgegebenen Reihenfolge hinzugefügt werden.
- Überladen Sie die **nextStop** Methode für den Fall, dass keine neuen Passagiere zusteigen. Diese **nextStop** Methode hat keine Parameter. Ansonsten verhält sich diese Methode analog zu der obigen Variante. Auch Zugriffsmodifikator und Rückgabetyp sind identisch.
- **Bus** besitzt die öffentliche Methode **findPassengersWithoutTickets()**, welche alle Passagiere ohne Ticket im Bus ausfindig macht und diese aus **passengers** entfernt. Die Methode gibt im Anschluss eine ArrayList mit allen Passagieren zurück, die aus dem Bus entfernt wurden in der Reihenfolge, in der sie auch in **passengers** enthalten waren.
- Zusätzlich besitzt **Bus** die öffentliche **void**-Methode **transferPassengers(Bus otherBus, String[] passengerNames)**. Diese Methode simuliert das Umsteigen von Passagieren in einen anderen Bus. Dabei werden die Passagiere, deren Namen in **passengerNames** enthalten sind, aus dem aktuellen Bus entfernt und in den Bus **otherBus** hinzugefügt in der Reihenfolge, in der sie auch ursprünglich im ersten Bus enthalten waren. Sie können davon ausgehen, dass kein Name doppelt vorkommt.

  Beispiel:

  - Bus A enthält die Passagiere: Anna, Ben, Clara.
  - Bus B ist leer.
  - Anna und Clara steigen aus Bus A aus und in Bus B ein.
  - Nach der Ausführung enthält Bus A nur Ben, und Bus B enthält Anna und Clara.