Programmieren II (Java)

1. Praktikum: Grundlagen

Sommersemester 2023

Christopher Auer, Tobias Lehner

Abgabetermine

Lernziele

- Erstes Beschnuppern von Java
- Arbeiten mit Kontrollstrukturen und primitiven Datentypen
- Arithmetik
- Implementieren einer Konsolenanwendung
- Implementierung eines Algorithmus nach einer Spezifikation

Hinweise

- Sie dürfen die Aufgaben *alleine* oder zu *zweit* bearbeiten und abgeben
- Sie müssen *4 der 5* Praktika bestehen
- *Kommentieren* Sie Ihren Code
  - Jede *Methode* (wenn nicht vorgegeben)
  - *Wichtige* Anweisungen/Code-Blöcke
  - *Nicht kommentierter* Code führt zu *Nichtbestehen*
- Bestehen Sie eine Abgabe *nicht* haben Sie einen *zweiten Versuch*, in dem Sie Ihre Abgabe *verbessern müssen*.
- *Wichtig*: Sie sind einer *Praktikumsgruppe* zugewiesen, *nur* in dieser werden Ihre Abgaben *akzeptiert*!

Aufgabe 1: Java zu Fuß

Erstellen Sie eine Java-Datei mit folgendem Inhalt und dem Namen `HelloJava.java`.

```java
public class HelloJava{
  public static void main(String[] args){
    System.out.println("Hello Java!");
  }
}
```

HelloJava.java

- ☑ Installieren Sie das JDK („Java Development Kit“) von Oracle oder OpenJDK
- ☑ Starten Sie eine Kommandozeile und navigieren Sie (mit `cd`) in das Verzeichnis, in dem die Datei `HelloJava.java` liegt.
- ☑ Übersetzen Sie die Datei folgenden Kommando in eine `.class`-Datei:

```
javac HelloJava.java
```

- ☑ Führen Sie das Programm aus mit

```
java HelloJava
```

Das Programm sollte `Hello Java!` ausgeben.

Hinweise

- ▶ Sie benötigen für diese Aufgabe keine Entwicklungsumgebung.
- ▶ Diese Aufgabe müssen Sie *nicht abgeben*

Aufgabe 2: Pseudo-Zufallszahlen

Es ist überraschend schwierig, einem Computer, der nach deterministischen (nicht zufälligen) Regeln Ihre Anweisungen abarbeitet, Zahlen zu entlocken, die *einigermaßen zufällig* sind. In Java gibt es dafür die Methode `Math.random()` und die Klasse `Random`. Diese generieren Pseudo-Zufallswerte, d.h., Werte, die *zufällig „aussehen“*, aber es nicht wirklich sind. In dieser Aufgabe implementieren wir den schon recht alten Lehmer-Pseudo-Zufallszahlengenerator von 1951.

Der Generator startet mit einem *Startwert* $X_0 \in \mathbb{N}$, auch *Seed* genannt, und berechnet daraus *iterativ* eine *Folge* von natürlichen Zahlen $X_1, X_2, \ldots$ mit folgender Rechenvorschrift:

$$X_{i+1} = (aX_i + c) \mod m,$$

wobei

- m ($0 < m$) das *Modul*,
- a ($0 < a < m$) der *Multiplikator*,
- c ($0 \leq c < m$) das *Inkrement* ist.

Die Werte von m, a und c sind je nach Implementierung festgelegt. Wir wählen $m = 2^{31} - 1 = 2147483647$, $a = 48271$ und $c = 0$.

Die Klasse `PseudoRandomNumbers`

Erstellen Sie eine Klasse `PseudoRandomNumbers`:

- Deklarieren Sie in der Klasse *zwei öffentliche konstante* `long`*-Attribute* (Modifier `public final static`) `MODULUS` und `MULTIPLIER` mit den Werten von m und a von oben (c ignorieren wir in der Aufgabe).
- Implementieren Sie eine `main`-Methode, die *zehn Zufallszahlen* $X_1, \ldots, X_{10}$, ausgehend vom Seed $X_0 = 42$, nach obigem Schema berechnet und auf dem Terminal *zeilenweise* ausgibt. Zum Vergleich, hier das zu erwartende Ergebnis (aus *Platzgründen* in zwei Zeilen):

```
2027382 1226992407 551494037 961371815 1404753842
2076553157 1350734175 1538354858 90320905 488601845
```

Einlesen der Anzahl und des Seeds

Modifizieren Sie Ihre `main`-Methode so, dass die Anzahl zu erzeugenden Zufallszahlen und der Seed vom Nutzer auf dem Terminal eingelesen werden:

```
Anzahl Zufallszahlen (>=0): 10
Seed (>0): 42
2027382 1226992407 551494037 961371815 1404753842
2076553157 1350734175 1538354858 90320905 488601845
```

Verwenden Sie zum Einlesen der Zahlen die Klasse `Scanner` und die Methode `nextInt()`.

Zufallszahlen in einem Intervall

Unser Generator liefert Zahlen von 0 bis 2.147.483.647 (ausgeschlossen). Oft benötigt man allerdings Zufallszahlen zwischen zwei *selbstdefinierten* Werten, z.B. 1 bis 6 für einen Würfel. Erweitern Sie Ihr `main`-Programm so, dass vom Nutzer zusätzlich zwei Variablen `min` und `max` eingelesen werden. Per Konvention ist dabei die *Untergrenze* `min` *eingeschlossen* und die *Obergrenze* `max` *ausgeschlossen*. Um aus der Zufallszahl X_i eine Zahl Y_i im Intervall `min` bis `max-1` zu erzeugen, können Sie sich folgender Abbildung bedienen:

$$Y_i = (X_i \mod (\text{max} - \text{min})) + \text{min}\,.$$

Geben Sie nun anstatt X_i die Zahlen Y_i im *vom Nutzer festgelegten Intervall* aus (Ergebnisse aus *Platzgründen* in einer Zeile):

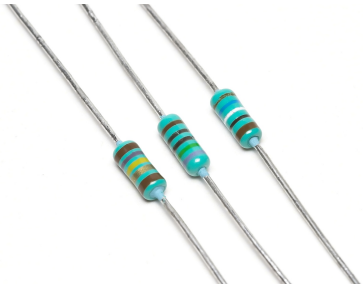
```
Anzahl Zufallszahlen (>=0): 10
Seed (>0): 42
Min: -5
Max (exklusiv, >=Min): 6
0 -2 1 3 3 3 5 2 -1 1
```

Umgang mit fehlerhaften Eingaben

Wenn nicht schon geschehen, testen Sie Ihr Programm mit *fehlerhaften Eingaben* und modifizieren Sie Ihr Programm so, dass es mit solchen Eingaben umgehen kann! Sie können davon ausgehen, dass Ihr Übungsleiter versuchen wird Ihr Programm mit *fehlerhaften Eingaben* zum Absturz zu bringen.

Aufgabe 3: Widerstand

Der Wert eines elektrischen Widerstands sind über eine Widerstandsfarbkodierung auf dem Bauteil dargestellt:



Elektrische Widerstände (Bild: Afrank99 unter CC BY-SA 2.5)

Neben dem Wert des Widerstands (die ersten 3 bis 4 Ringe von links) ist auch noch die *Toleranz* und, seltener, der *Temperaturkoeffizient* angegeben. Im Folgenden betrachten wir uns *nur den Wert des Widerstands* und ignorieren die restlichen Ringe.

Der Wert des Widerstands lässt sich aus der *Wertigkeit* (die ersten 2 bis 3 Ringe) und einem *Multiplikator* (letzter Ring) berechnen. Die Farbcodierung ist wie folgt definiert:

Farbe	Wertigkeit	Multiplikator
Silber	—	$0,01 = 10^{-2}$
Gold	—	$0,1 = 10^{-1}$
Schwarz	0	$1 = 10^{0}$
Braun	1	$10 = 10^{1}$
Rot	2	$100 = 10^{2}$
Orange	3	$1.000 = 10^{3}$
Gelb	4	$10.000 = 10^{4}$
Grün	5	$100.000 = 10^{5}$
Blau	6	$1.000.000 = 10^{6}$
Violett	7	$10.000.000 = 10^{7}$
Grau	8	$100.000.000 = 10^{8}$
Weiß	9	$1.000.000.000 = 10^{9}$

Betrachten wir folgenden Widerstand:

Der letzte Ring (rot) definiert den *Multiplikator* $m = 100$. Die ersten drei Ringe (gelb, grau, grün), die Wertigkeit: $w = 485$. Damit hat der Widerstand einen Wert von $m \cdot w = 485.000\,\Omega$ (Ohm).

Und noch ein Beispiel:

In diesem Fall haben wir nur *drei Ringe*. Der letzte (gelb) definiert wieder den Multiplikator $m = 10.000$, die ersten beiden die Wertigkeit $w = 10$. Damit hat der Widerstand einen Wert von $100.000\,\Omega$.

Implementieren Sie eine Java-Programm `ResistorColorConverter`, das über die *Kommandozeilenargumente drei* oder *vier* Farbwerte übernimmt und den entsprechenden Wert des Widerstands errechnet! Beispiel für einen Aufruf:

```
$ java ResistorColorConverter braun schwarz gelb
Der Widerstand hat den Wert: 100000.000000 Ohm
```

Oder:

```
$ java Gelb GRAU GrÜn ROt
Der Widerstand hat den Wert: 48500.000000 Ohm
```

Die *Groß-/Kleinschreibung* der Farben soll *keine Rolle spielen*.

Hinweise:

- Der Wert des Multiplikator ist gleich 10^w, wobei w der Zahl für die Wertigkeit entspricht (bis auf Silber und Gold).
- Sie können auf die *Kommandozeilenargumente* über den String-Array `args` in `main(String[] args)` zugreifen. Die Länge `args.length` gibt dabei die *Anzahl* der übergebenen Argumente an. Dabei ist `args[0]` das *erste Argument*, `args[1]` das *zweite*, etc. Für *C-Experten*: `args[0]` enthält im Unterschied zu C *nicht den Programmnamen* sondern das erste übergebene Argument.
- Vermeiden Sie *Codeduplikation*: Überlegen Sie sich, welche Funktionen Sie in Methoden (Funktionen) auslagern können. Achten Sie dabei darauf, dass eine Methode **`static`** sein muss (die Gründe dafür werden später klar). Beispielsweise können Sie eine Methode **`private static int`** `colorToValue(String color)` implementieren, die für die übergebene Farbe die *Wertigkeit* zurückgibt. Entsprechendes macht auch für den *Multiplikator* Sinn.
- Achten Sie auf die Verwendung *geeigneter Datentypen*!
- Überlegen Sie sich, wie Sie mit *fehlerhaften* Eingaben umgehen.

Aufgabe 4: Ziegenproblem ★★

Das Ziegenproblem ist ein bekanntes Problem aus der Wahrscheinlichkeitstheorie: In einer Gewinnshow, werden einem Teilnehmer drei verschlossene Tore gezeigt. Hinter einem Tor versteckt sich mit der Wahrscheinlichkeit $\frac{1}{3}$ der *Hauptgewinn*, hinter den beiden anderen jeweils eine *Ziege* (symbolisiert eine *Niete*). Der Gewinnshowteilnehmer entscheidet sich für eine der drei Tore, woraufhin der Showmaster ein Tor, hinter der *garantiert* eine Ziege steht, öffnet. Der Gewinnshowteilnehmer hat nun die Chance bei seiner *ersten Wahl* zu bleiben (Strategie A) oder zu *wechseln* (Strategie B).

Was sagt Ihr *Gefühl*? Macht es *einen Unterschied*, ob man *wechselt* oder *nicht*? Schließlich ändern sich ja nicht *plötzlich die Wahrscheinlichkeiten*, dass sich hinter dem zuerst gewählten Tor eine Ziege befindet. Man kann *Nachrechnen*, ob es einen Unterschied macht, was Sie in der Vorlesung *Statistik* machen werden, oder über eine *Simulation* die beiden Strategien *gegeneinander antreten lassen*.[1]

Als *Grundgerüst* verwenden Sie das mitgelieferte *Gradle-Projekt* und die Java-Klasse in der Datei Ziegenproblem.java . *Importieren* Sie das Gradle-Projekt in Ihre IDE und testen Sie Ihr Programm, indem Sie es mit dem Gradle-Task `run` ausführen.

Die Methode `getRemainingDoor`

In der Klasse `Ziegenproblem` ist eine Methode `int getRemainingDoor(int door1, door2)` deklariert, die wir im Folgenden benötigen. Die drei Tore werden in unserem Programm durch die Ganzzahlen `0`, `1` und `2` (`int`) dargestellt. Die Methode `getRemainingDoor(door1,door2)` liefert das Tor, das von `door1` und `door2` *unterschiedlich* ist, wobei `door1!=door2` Der Aufruf `getRemainingDoor(0,1)` liefert z.B. `2`, der Aufruf `getRemainingDoor(0,2)` liefert `1`, etc. Implementieren Sie `getRemainingDoor` und testen Sie Ihre Methode über die `main`-Methode!

Simulation der Gewinnshow

Implementieren Sie die `main`-Methode wie folgt:

- Deklarieren Sie zwei `int`-Variablen `winsA` und `winsB` (Anzahl Hauptgewinne für die Strategien A und B).
- Simulieren Sie 10.000.000 Gewinnshow-Iterationen:
  - Wir gehen *immer* davon aus, dass der *Hauptgewinn* hinter *Tor 0* ist.[2]
  - *Wahl des Kandidaten*: Deklarieren Sie eine `int`-Variable `candidateDoor` und weisen Sie ihr einen zufälligen Wert von `0` bis `2` zu. Verwenden Sie dazu die Methode `Math.random()`, die eine `double`-Zufallszahl zwischen `0` und `1` liefert, und multiplizieren Sie diese mit 3 um eine Zufallszahl zwischen 0 und 3 zu erhalten. Eine *Typumwandlung* zu `int` erzeugt die gewünschte Zufallszahl.
  - *Showmaster öffnet Ziegen-Tor*: Deklarieren Sie eine Variable `goatDoor` (noch ohne Wert). Die Variable wird die Nummer des Tors erhalten, das der Showmaster als Niete enthüllt.

[1] Man kann auch im Internet recherchieren, aber natürlich „spoilern“ Sie sich dadurch nur selber.
[2] Für Mathe-Experten: Es handelt sich hier um ein „*ohne Beschränkung der Allgemeinheit* “

Sollte der Kandidat Tor `0` gewählt haben, so öffnet der Showmaster mit 50% Wahrscheinlichkeit Tor `1` und mit 50% Wahrscheinlichkeit Tor `2`. Verwenden Sie `Math.random()<0.5` als Bedingung um `goatDoor` auf 1 zu setzen; sollte diese Bedingung nicht erfüllt sein, setzen Sie `goatDoor` auf 2.
Sollte der Kandidat Tor 1 oder 2 gewählt haben, so öffnet der Showmaster das *übriggebliebene Tor*, d.h. Sie setzen `goatDoor` auf den *Rückgabewert* von `getRemainingDoor(0, candidateDoor)`.

- *Strategie A*: Sollte der Kandidat Tor 0 gewählt haben, so erhöhen Sie den Zähler `winsA` um eins.
- *Strategie B*: Ermitteln Sie mit dem Aufruf `getRemainingDoor(candidateDoor, goatDoor)` das Tor, zu dem der Kandidat mit Strategie B wechselt. Sollte das resultierende Tor die Nummer 0 sein, so erhöhen Sie den Zähler `winsB` um eins.

- Geben Sie `winsA` und `winsB` wie folgt aus:

```
Strategie A: "hier steht Wert von winsA" Gewinne (<Prozent winsA> %)
Strategie B: "hier steht Wert von winsB" Gewinne (<Prozent winsB> %)
```

In Klammern steht wieviel *Prozent* aller Iterationen gewonnen wurden (auf *zwei Nachkommastellen* genau). Haben Sie das Ergebnis so *erwartet* oder sind Sie *überrascht*?