

Übung 1

Aufgabe 1:

Versuchen Sie alle 10 Stufen des Blockly Spiels „Labyrinth“ zu spielen:

<https://blockly.games/maze?lang=de>

Aufgabe 2:

Modifizieren Sie das Programm aus der Vorlesung, so dass

- die Summe von 1 bis 5 berechnet wird. WICHTIG: Sie dürfen nur die Zeile mit dem *while* verändern,
- das Produkt von 1 bis 10 berechnet wird,
- bei der Summen- bzw. Produktbildung die Zwischenergebnisse auch mit auf dem Bildschirm ausgegeben werden.

Aufgabe 3:

Zeichnen Sie die Syntaxdiagramme, die ein Geburtsdatum in folgender Schreibweise erzeugt: tt.mm.jjjj (z.B. 17.09.2001).

Dabei sollen nur Geburtsdaten zwischen dem 01.01.1900 und dem 31.12. 2099 erzeugt werden können (fehlerhafte Daten brauchen nicht abgefangen zu werden, z.B. der 31.02.2011).

Übung 2

Aufgabe 1:

Versuchen Sie möglichst viele Stufen des Blockly Spiels „Vögel“ zu schaffen:

<https://blockly.games/bird?lang=de>

Aufgabe 2:

Schreiben Sie ein Java Programm, das zeigt, dass die booleschen Operatoren in Java die DeMorganschen Gesetze erfüllen. Bedenken Sie, dass Sie alle vier möglichen Wahrheitswertbelegungen für die zwei booleschen Variablen testen.

Aufgabe 3:

Gegeben sind folgende Aussagen:

$(\neg A) \Rightarrow (B \wedge (\neg C)) = (\neg(\neg A)) \vee (B \wedge (\neg C))$.

Erstellen sie ein Java Programm, das beide Aussagen miteinander vergleicht und das Ergebnis ausgibt.

Aufgabe 4:

Welche Werte haben die Variablen am Ende des Programms?

```
boolean a = true, b = false, c = false;
if (a || b)
    b = true;
else
    c = true;
a = b && c;
c = !a;
```

Aufgabe 5:

Was gibt das folgende Programm aus?

```
public class WasGebelchAus{
    public static void main(String[] args) {
        int x = 10;
        boolean a = x > 0 && x <= 10;
        boolean b = x < 5 || x > 9;
        boolean c = !(b || a);
        System.out.println(a);
        System.out.println(b);
        System.out.println(c);
    }
}
```

Übung 3

Aufgabe 1:

Versuchen Sie möglichst viele Stufen des Blockly Spiels „Schildkröte“ zu schaffen:

<https://blockly.games/turtle?lang=de>

Aufgabe 2:

Schreiben Sie ein Programm mit dem Namen *Life*, dass die folgende Ausgabe produziert:

```
X_X_X_X_X_X
X_X_X_X_X
X__X__X__
X___X___X
X____X____X
X_____X_____
```

Dabei soll die Anzahl der Zeilen und Spalten jeweils von der Tastatur eingelesen werden (siehe nächste Aufgabe). Testen Sie Ihr Programm mit unterschiedlichen Spalten und Zeilen.

Aufgabe 3:

Schreiben sie ein Programm, das drei Integer Werte über die Tastatur einliest und den größten dieser Werte ausgibt. Verwenden Sie hierzu die unten genannten Java Befehle, um einen int Wert von der Tastatur einzulesen.

```
import java.util.Scanner; //Importanweisung für den Scanner
```

....

```
Scanner sc = new Scanner(System.in); // neues Scanner-Objekt sc
int i = sc.nextInt(); // lokale Variable speichert die Eingabe
```

.....

Übung 4

Aufgabe 1:

Schreiben Sie ein Java Programm, das den GGT (größten gemeinsamen Teiler) zu zwei int Zahlen a und b berechnet. Lesen Sie die Zahlen wie in der letzten Woche von der Tastatur ein.

Aufgabe 2:

Schreiben Sie ein Java Programm, das das KGV (kleinste gemeinsame Vielfache) zu zwei int Zahlen a und b berechnet. Lesen Sie die Zahlen wie in der letzten Woche von der Tastatur ein.

Aufgabe 3:

Zum Lösen dieser Aufgabe benötigen Sie den sogenannten Modulo Operator %. Der Operator gibt den Rest nach der Division zweier ganzer Zahlen aus.

Beispiel : $10\%3 = 3$ Rest 1. Das Ergebnis ist also 1.

Schreiben Sie ein Java-Programm, das eine vierstellige Ganzzahl von der Tastatur einliest, prüft und ausgibt ob es sich bei der Zahl um ein Schaltjahr handelt oder nicht.

Aufgabe 4:

Berechnen sie mit Hilfe einer while Schleife die Summe der ersten 100 natürlichen Zahlen. Es sollen immer 10 Zahlen in einer Zeile, getrennt durch einen Tabulator ausgegeben werden. Die println() Anweisung darf nicht zum Zeilenumbruch benutzt werden.

Aufgabe 5:

Verändern Sie Ihr *Life* Programm, so dass zunächst ein zweidimensionales boolesches Array angelegt wird, das so groß ist, wie die Ausgabe sein soll.

Verwenden Sie bei der Erzeugung des Arrays für die Größenangaben wieder die int-Variablen.

In einer Initialisierungsphase sollen die Zellen des Arrays wie folgt beschrieben werden (T = true, F = false):

TFTFTFTFTFT

TFFTFFTFFTF

TFFF TFFF TFF

TFFFFTFFFFT

TFFFFFTFFFF

Geben Sie anschließend das Array auf dem Bildschirm aus. Dabei soll die Ausgabe derart aussehen, wie im Programm aus Übung 3.

Übung 5

Aufgabe 1:

Stellen Sie Ihr *Life* Programm auf die for-Schleife um. Verwenden Sie für den Schleifenabbruch die *.length* Information des Arrays.

Nach Ausgabe des Arrays berechnen Sie ein neues boolesches Array der gleichen Größe. Die Regeln, wann eine Zelle auf *true* gesetzt wird, lauten wie folgt:

- ist die Zelle *true* und in der Nachbarschaft gibt es 2 oder 3 Zellen mit *true*, bleibt die Zelle *true*
- ist die Zelle *false* und in der Nachbarschaft gibt es 3 Zellen mit *true*, wird die Zelle *true*
- gibt es mehr als 3 oder weniger als 2 *true* in der Nachbarschaft, wird die Zelle mit *false* besetzt

Dabei bilden die Zellen, die in einem Schritt über, unter, neben und diagonal der untersuchten Zelle liegen, die Nachbarschaft. Somit haben die mittleren Zellen 8 Nachbarn, die Randzellen weniger.

Geben Sie das neue Array aus.

Aufgabe 2:

In den folgenden Übungen sollen Sie ein Java Programm *Rational* entwickeln, das das Rechnen mit rationalen Zahlen implementiert. Eine rationale Zahl soll durch ein int-Array der Länge zwei implementiert werden. An der Position 0 ist der Zähler gespeichert, an der Position 1 der Nenner.

Ihr Programm soll zunächst die zwei rationalen Zahlen $\frac{4}{6}$ und $\frac{15}{24}$ anlegen, diese dann automatisch kürzen und dann ausgeben.

Aufgabe 3:

Schreiben Sie ein Programm, das zwei int Zahlen m und n einliest, m-n berechnet und ausgibt. Dazu dürfen Sie aber nicht die Operatoren - oder * verwenden. Verwenden Sie stattdessen die Binäroperatoren und den + Operator.

Übung 6

Aufgabe 1:

Stellen Sie Ihr *Life* Programm auf Methoden um.

Die Initialisierung, Ausgabe und Neuberechnung soll jeweils durch eine Methode erfolgen.

Verändern Sie die Definition für die Nachbarschaft. Jetzt sollen auch die Zellen am Rand und in den Ecken jeweils 8 Nachbarn haben. Dazu hat der linke Rand als linken Nachbarn den ganz rechten Rand, die oberen Zellen als oberen Nachbarn die ganz unteren Zellen usw. D.h. geht man oben aus dem Feld hinaus, kommt man unten wieder hinein, geht man links heraus, kommt man rechts wieder hinein.

Führen Sie die Neuberechnung und Ausgabe immer wieder durch.

Aufgabe 2:

Erweitern Sie Ihr *Rational* Programm derart, dass die beiden Zahlen einmal addiert und einmal multipliziert werden. Speichern Sie die Werte erneut in int-Arrays der Länge zwei ab. Geben Sie die Ergebnisse einmal vor dem Kürzen und einmal nach dem Kürzen aus.

Aufgabe 3:

Erstellen Sie ein Programm, das je nach Eingabe der Höhe folgende Ausgabe erzeugt:

```
Geben Sie die Höhe der Pyramide ein !8
```

```
  X
  XXX
 XXXXX
XXXXXXX
XXXXXXXXX
XXXXXXXXXX
XXXXXXXXXXX
XXXXXXXXXXXX
XXXXXXXXXXXXX
```

Benutzen Sie dazu die for-Schleife.

Aufgabe 4:

Schreiben Sie ein Programm, das folgende Ausgabe produziert:

```
Geben Sie die Höhe der Raute ein : 5
```

```
  X
  XXX
 XXXXX
  XXX
  X
```

Aufgabe 5:

Schreiben Sie eine Methode, die ein zweidimensionales Array der Größe 6x6 vom Typ int erzeugt und dieses mit zweistelligen Zufallszahlen füllt. Danach sollen die Zeilensumme und die Gesamtsumme aller Felder berechnet und jeweils ausgegeben werden.

Übung 8

Aufgabe 1:

Implementieren Sie die Sortierverfahren aus der Vorlesung für Double Werte. Führen Sie Laufzeitmessungen durch.

Aufgabe 2:

Implementieren Sie das Distribution Counting derart, dass die Methode zunächst selber ermittelt, aus welchem Wertebereich die zu sortierenden Zahlen stammen. Modifizieren Sie die Lösung aus der Vorlesung derart, dass auch negative Zahlen sortiert werden können.

Aufgabe 3:

Erweitern Sie Ihr *Rational* Programm um Methoden, die zwei rationale Zahlen addiert, multipliziert, dividiert bzw. subtrahiert. Die Ergebnisse sollen jeweils gekürzt sein.

Übung 9

Aufgabe 1:

Erweitern Sie Ihr *Rational* Programm, so dass die Berechnung des GGTs durch eine rekursive Methode erfolgt.

Aufgabe 2:

Schreiben Sie ein Java Programm, das das 8 Dame Problem löst. Lösen Sie das Problem rekursiv.

Aufgabe 3:

Schreiben Sie eine rekursive Methode, die die Wurzel zu einer übergebenen int-Zahl mittels der fortgesetzten Intervallhalbierung ermittelt.

Aufgabe 4:

Zwei Zahlen a und b können dadurch multipliziert werden, dass eine Schleife von 0 bis b-1 läuft und immer wieder zum Ergebnis a hinzuaddiert.

Implementieren Sie eine Methode `mult_iter`, die dieses Verfahren iterativ implementiert und eine Methode `mult_rec`, die dieses Verfahren rekursiv implementiert. Beachten Sie, dass a und b auch jeweils negativ sein können.

Übung 10

Aufgabe 1:

Stellen Sie Ihr *Life* Programm auf Klassen Objekte um.

Die Methoden sollen Objektmethoden werden und bekommen keine booleschen Arrays mehr übergeben, sondern greifen auf Objektvariablen zu.

Ersetzen Sie auch das boolesche Array durch ein int-Array.

Mit dem int-Array soll kodiert werden, ob eine Zelle besetzt ist, und wenn Sie besetzt ist, wie lange sie schon besetzt ist.

Verändern Sie die Ausgabe wie folgt:

Wert=0 → Ausgabe: Leerzeichen

Wert=1 → Ausgabe: ‘.’

Wert=2 → Ausgabe: ‘o’

Wert=3 → Ausgabe: ‘O’

Wert>3 → Ausgabe: ‘*’

Aufgabe 2:

Stellen Sie Ihr *Rational* Programm auf Klassen und Objekte um. Die Klasse *Rational* besitzt als Objektvariable zwei int-Variablen *Zähler* und *Nenner*. Der Konstruktor initialisiert diese. Die bisherigen Methoden werden Objektmethoden.

Übung 11

Aufgabe 1:

Machen Sie Ihr *Life* Programm noch „objektorientierter“.

Schreiben Sie dazu eine Klasse *Stone*, die eine Zelle in Ihrem *Life* Programm darstellt.

D.h. das *Life* Programm speichert nicht mehr ein `int`-Array sondern ein *Stone* Array.

Statten Sie Ihre *Stone* Klasse mit notwendigen Methoden aus, wie

- *print*: Ausgabe
- *countNeighbour*: zur Berechnung der Anzahl der Nachbarn
- *isAlive*: ist die Zelle besetzt
- *computeNext*: wird in der nächsten Iteration die Zelle besetzt sein

Aufgabe 2:

Implementieren Sie die drei Sortiermethoden Insertion-, Selection- und Bubblesort derart, dass sie Arrays von *Rational* Objekten sortieren.

Übung 12

Aufgabe 1:

Erweitern Sie Ihr *Stone* Klasse um ein Array, so dass sich jede Zelle Ihre Nachbarn direkt merkt.

Stellen Sie Ihre Methoden auf diese neue Art der Speicherung um.

Leiten Sie von der Klasse *Stone* zwei Klassen *AlwaysStone* und *NeverStone* ab.

Die *AlwaysStone* Klasse repräsentiert eine Zelle, die immer besetzt ist, egal, wie die Nachbarschaft aussieht.

Die *NeverStone* Klasse repräsentiert eine Zelle, die niemals besetzt ist, egal, wie die Nachbarschaft aussieht.

Verwenden Sie diese beiden neuen Klassen zusätzlich zu der *Stone* Klasse bei der Initialisierung der *Life* Klasse.

Aufgabe 2:

Definieren Sie eine abstrakte Klasse *Number* und leiten von dieser Klasse Ihre Klasse *Rational* ab. Stellen Sie Ihre Sortierverfahren um, so dass diese ein Array von *Number* Objekten erwartet. Welche abstrakten Methodendeklarationen braucht die abstrakte Klasse *Number*?

Übung 13

Aufgabe 1:

Definieren Sie ein Interface *Disease* zur Darstellung von Krankheiten mit den Methoden *chanceOfInfection* und *chanceOfDecease*. Beide Methoden liefern ein Double Wert zurück, der zwischen 0.0 und 1.0 liegen soll.

Erweitern Sie die Klasse Stone um ein Array von *Disease* Objekten. Bei der Initialisierung von Stone Objekten sollen ein paar Objekte ein paar unterschiedliche Krankheiten haben.

Erweitern Sie Ihre *computeNext* Methode, so dass Krankheiten berücksichtigt werden. Erzeugen Sie dazu in jedem Iterationsschritt eine Zufallszahl zwischen 0.0 und 1.0 und vergleichen Sie diesen Wert mit den *chanceOfDecease* Werten der Krankheiten des Steins. Ist der Wert höher, stirbt der Stein nicht.

Verfahren Sie entsprechend mit *chanceOfInfection*, um Krankheiten von den Nachbarn zu übernehmen.

Aufgabe 2:

Erweitern Sie Ihre *Rational* Methoden und den Konstruktor derart, dass eine Exception geworfen wird, sollte zu irgendeinem Zeitpunkt der Nenner 0 werden. Von welcher Klasse ist diese selbstgeschriebene Exception sinnvoller Weise abgeleitet?

Übung 14

Aufgabe 1:

Erweitern Sie Ihr Interface *Disease* um die Methoden *chanceOfCure*. Die Methoden liefern ein Double Wert zurück, der zwischen 0.0 und 1.0 liegen soll.

Erweitern Sie Ihre *computeNext* Methode, so dass *chanceOfCure*, also die Chance auf Heilung berücksichtigt wird. Erzeugen Sie dazu wiederum in jedem Iterationsschritt eine Zufallszahl zwischen 0.0 und 1.0 und vergleichen Sie diesen Wert mit den *chanceOfCure* Werten der Krankheiten des Steins. Ist der Wert höher, wird die Krankheit entfernt.

Erweitern Sie Ihre *Life* Klasse um eine weitere Ausgaberoutine, die die Daten in eine Datei speichert, so dass diese Daten wieder eingelesen werden können und die Spielsituation wieder hergestellt werden kann.

Verwenden Sie für die Ausgabe die *DataOutputStream* und für das Einlesen die *DataInputStream* Klasse.