1 For-Schleifen

1.1 Wiederholte Ausgabe

1.1.1 Aufgabe

Implementiere eine Funktion `printSquares0_10`. Sie gibt die Quadrate der Zahlen von 0 bis 10 an der Konsole aus.

```
printSquares0_10()
```

```
0
1
4
9
16
25
36
49
64
81
100
```

1.1.2 Aufgabe

Implementiere eine Funktion `printSquares`. Dieser wird eine positive Zahl n übergeben. Sie gibt die ersten n Quadratzahlen an der Konsole aus.

```
printSquares(5)
```

```
0
1
4
9
16
```

1.1.3 Aufgabe

Implementiere eine Funktion `printSquaresNicer`. Dieser wird eine positive Zahl n übergeben. Sie kündigt zuerst an, wie viele Quadratzahlen sie ausgeben wird. Nach der Ausgabe dieser Quadratzahlen verabschiedet sie sich von dem Benutzer.

```
printSquaresNicer(5)
```

```
The first 5 squares are:
0
1
4
9
16
Goodbye
```

1.1.4 Aufgabe

Implementiere eine Funktion `printSquaresStartEnd`. Dieser werden zwei positive Zahlen m und n übergeben. Sie gibt die Quadrate der Zahlen von m bis n aus. Die Funktion gibt zunächst an welche Quadratzahlen sie ausgibt. Nach der Ausgabe dieser Quadratzahlen verabschiedet sie sich von dem Benutzer.

```
1 printSquaresStartEnd(5, 7)
```

```
The square of the numbers from 5 up to 7 are:
25
36
49
Goodbye
```

1.1.5 Aufgabe

Erweitere die Funktion aus der letzten Aufgabe zu einen Funktion `printSquaresStartEndPretty` so, dass in jedem Schritt auch die Rechnung angezeigt wird.

```
1 printSquaresStartEndPretty(5, 7)
```

```
The square of the numbers from 5 up to 7 are:
5 * 5 = 25
6 * 6 = 36
7 * 7 = 49
Goodbye
```

1.2 Akkumulator-Pattern

1.2.1 Aufgabe

Implementiere eine Funktion `summation`. Dieser wird eine positive Zahl n übergeben. Sie berechnet die Summe der Zahlen von 1 bis n. Es gilt z.B. `summation(3)` $= 1+2+3$ und allgemein: `summation(n)` $= 1 + \ldots$ `n`.

```
summation(1)
```

```
1
```

```
summation(3)
```

```
6
```

```
summation(5)
```

```
15
```

`https://www.codewars.com/kata/55d24f55d7dd296eb9000030/train/kotlin`

1.2.2 Aufgabe

Implementiere eine Funktion `numberToPwr`, mit der Potenzen berechnet werden können. Der Funktion wird die Basis und die Hochzahl übergeben. Sie gibt die berechnete Potenz zurück. Beispiel:

$$\texttt{numberToPWR}(2,3) = 2 \cdot 2 \cdot 2$$

```
numberToPwr(3, 1)
```

```
3
```

```
numberToPwr(2, 3)
```

```
8
```

```
numberToPwr(3, 3)
```

```
27
```

1.2.3 Aufgabe

Implementiere eine Funktion `factorial`. Dieser wird eine positive Zahl n übergeben. Sie berechnet das Produkt der Zahlen von 1 bis n. Es gilt z.B. $\texttt{factorial}(3) = 1 * 2 * 3$ und allgemein: $\texttt{factorial}(n) = 1 * \ldots n$. Das leere Produkt (mit null Faktoren) ergibt Eins.

```
factorial(0)
```

```
1
```

```
factorial(1)
```

```
1
```

```
factorial(3)
```

```
6
```

1.2.4 Aufgabe

Implementiere eine Funktion `choose`. Dieser werden zwei ganze Zahlen N und k übergeben. Sie gibt zurück, wie viele Möglichkeiten es gibt k Elemente aus N Elementen auszuwählen. Solange k nicht größer ist als N kann man für die Berechnung Binomialkoeffizienten nutzen.

$$\binom{N}{k} = \frac{n!}{k! \cdot (n-k)!}$$

```
choose(1, 1)
```

```
1
```

```
choose(3, 2)
```

```
3
```

```
choose(5, 10)
```

```
0
```

1.2.5 Aufgabe

Implementiere eine Funktion `countSheep`, die beim Einschlafen hilft. Dieser wird eine natürliche Zahl n übergeben. Sie gibt einen String zurück in dem entsprechend viele Schafe gezählt werden.

```
countSheep(0)
```

```
countSheep(1)
```

```
1 sheep...
```

```
countSheep(3)
```

```
1 sheep...2 sheep...3 sheep...
```

1.2.6 Aufgabe

Implementiere eine Funktion `sumCubes`. Dieser wird eine positive Zahl n übergeben. Sie berechnet Summe der ersten n Kubikzahlen. Es gilt z.B. `sumCubes(3)` $= 1^3 + 2^3 + 3^3$ und allgemein: `sumCubes(n)` $= 1^3 + \ldots n^3$.

```
sumCubes(1)
```

```
1
```

```
sumCubes(2)
```

```
9
```

```
sumCubes(3)
```

```
36
```

1.2.7 Aufgabe

Implementiere eine Funktion `getSum`. Dieser werden zwei ganze Zahlen übergeben. Sie gibt die Summe aller Zahlen dazwischen einschließlich der beiden Zahlen zurück. Es gilt z.B. `getSum(3,1)` $= 1 + 2 + 3$

```
getSum(1, 2)
```

```
3
```

```
getSum(3, 1)
```

```
6
```

1.2.8 Aufgabe

Nutze `for`-Schleifen um eine Funktion `nthFib` zu schreiben, mit der die n-te Fibonacci-Zahl berechnet werden kann. Die ersten beiden Fibonacci-Zahlen sind 0 und 1. Jede weitere Fibonacci-Zahl ist die Summe ihrer beiden Vorgänger.

```
nthFib(1)
```

```
1
```

```
nthFib(2)
```

```
1
```

```
nthFib(3)
```

```
2
```

```
nthFib(4)
```

```
3
```

https://www.codewars.com/kata/522551eee9abb932420004a0/train/kotlin

1.2.9 Aufgabe

Implementiere eine Funktion `solution`. Dieser wird eine ganze Zahl n übergeben. Sie gibt die Summe der Vielfache von 3 oder 5, die kleiner als diese Zahl sind, zurück.

```
solution(6)
```

```
8
```

```
solution(3)
```

```
0
```

https://www.codewars.com/kata/514b92a657cdc65150000006/train/kotlin

1.2.10 Aufgabe

Implementiere eine Funktion `divisors`. Dieser wird eine ganze Zahl n übergeben. Sie gibt die Anzahl der Teiler dieser Zahl zurück. Die Zahl 4 hat die Teiler 1,2 und 4. Also gilt `divisors(4)` = 3.

```
divisors(4)
```

```
3
```

```
divisors(5)
```

```
2
```

1.2.11 Aufgabe

Eine Primzahl ist eine natürliche Zahl größer Eins, die nur durch eins und sich selbst teilbar ist. Implementiere eine Funktion `isPrime`. Dieser wird eine ganze Zahl n übergeben. Sie gibt die zurück, ob die Zahl eine Primzahl ist.

```
isPrime(6)
```

```
false
```

```
isPrime(7)
```

```
true
```

1.2.12 Aufgabe

Eine natürliche Zahl ist perfekt, wenn die Summe ihrer echten Teiler (bis auf sie selbst) genau die Zahl ergibt. Z.B. gilt $1+2+3=6$. Also ist 6 eine perfekte Zahl. Implementiere eine Funktion `isPerfect`. Dieser wird eine ganze Zahl n übergeben. Sie gibt die zurück, ob die Zahl perfekt ist.

```
isPerfect(6)
```

```
true
```

```
isPerfect(7)
```

```
false
```

1.2.13 Aufgabe

Eine natürliche Zahl ist abundant(überladen), wenn die Summe ihrer echten Teiler (bis auf sie selbst) größer als die Zahl selbst ist. Z.B. ist 12 eine abundante Zahl, da die echten Teiler von 12 die Zahlen $1,2,3,4$ und 6 sind. Deren Summe 16 ist größer als 12.

Implementiere eine Funktion `abundantNumber`, die prüft, ob eine natürliche Zahl abundant ist.

```
abundantNumber(12)
```

```
true
```

```
abundantNumber(7)
```

```
false
```

1.2.14 Aufgabe

Zwei natürliche Zahl sind befreundet, wenn sie verschieden sind und die Summe ihrer Teiler jeweils der anderen Zahl entspricht. Z.B. sind 220 und 284 befreundet. Die Teiler von 220 sind $1, 2, 4, 5, 10, 11, 20, 22, 44, 55$ und 110. Die Summe dieser Zahlen ist 284.

Die Teiler von 284 sind $1, 2, 4, 71$ und 142. Deren Summe ist 220. Implementiere eine Funktion `amicableNumbers`, die prüft ob zwei Zahlen befreundet sind.

```
amicableNumbers(220, 284)
```

```
true
```

```
amicableNumbers(10, 11)
```

```
false
```