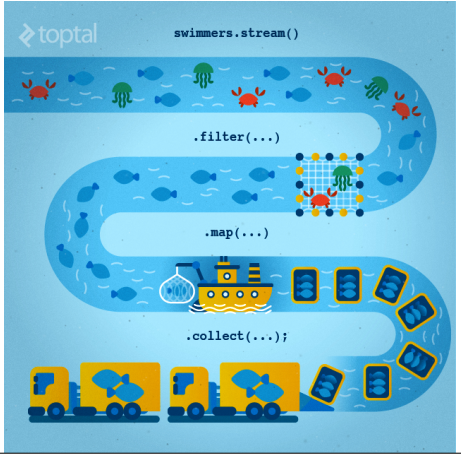
FOP Recap #12

Streams



Gude!

Das steht heute auf dem Plan

TECHNISCHE UNIVERSITÄT DARMSTADT

Wiederholung

Was sind Streams?

Warum Streams?

Umgang mit Streams

Code-Style

Das steht heute auf dem Plan

TECHNISCHE UNIVERSITÄT DARMSTADT

Wiederholung
- Lambda-Funktionen
- „Double Colon“ Operator (`::`)

Was sind Streams?

Warum Streams?

Umgang mit Streams

Code-Style

Wiederholung
Lambda-Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Sind „Funktionen ohne Namen“
- Haben auch Parameter und Methodenrumpf

Syntax

```
1  // Keine Parameter
2  () -> rumpf
3  // Ein Parameter
4  param -> rumpf
5  // Mehrere Parameter
6  (param_1, ..., param_n) -> rumpf
7  // Mehrere Anweisungen im Rumpf
8  (param_1, ..., param_n) -> {
9      anweisung_1
10     // ...
11     anweisung_n // z.B. return
12 }
```

Wiederholung
Lambda-Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel: „Liste Sortieren“

```java
List<Integer> numbers = Arrays.asList(3, 2, 1, 4, 5);
```

Sortieren ohne Lambda

```java
numbers.sort(new Comparator<Integer>() {
    @Override
    public int compare(Integer a, Integer b) {
        return a - b;
    }
});
```

Sortieren mit Lambda

```java
numbers.sort((a, b) -> a - b);
```

```
$ [1,2,3,4,5]
```

Wiederholung

„Double Colon“ Operator (`::`)

TECHNISCHE UNIVERSITÄT DARMSTADT

Der „Double Colon“ Operator (`::`) ist eine Kurzschreibweise für Methoden- und Konstruktorenreferenzen. Beispiel:

Sortieren mit Methodenreferenz

```
1  // Langform (mit Lambda)
2  numbers.sort((a, b) -> a.compareTo(b));
3  // Kurzform (mit Methodenreferenz)
4  numbers.sort(Integer::compareTo);
```

Das steht heute auf dem Plan

TECHNISCHE UNIVERSITÄT DARMSTADT

Wiederholung

Was sind Streams?
- Definition
- Merkmale
- Beispiel

Warum Streams?

Umgang mit Streams

Code-Style

Was sind Streams?

Definition

TECHNISCHE UNIVERSITÄT DARMSTADT

Stream — Ein Stream ist eine Folge von Objekten, die verschiedene Methoden unterstützt, die in einer Kette (Pipeline) verarbeitet werden können, um das gewünschte Ergebnis zu erzielen.[1]

- Einführung der Stream-API mit Java 8
- Ein Stream ist **keine** Datenstruktur, sondern eine Hilfsklasse zur Verarbeitung von Daten
- Streams verändern die ursprüngliche Datenstruktur nicht

[1] `https://www.geeksforgeeks.org/stream-in-java/`

Was sind Streams?
Merkmale

TECHNISCHE UNIVERSITÄT DARMSTADT

- Streams können **unendlich** groß sein
- Streams sind „immutable“ (nach der Erstellung nicht mehr veränderbar)
- Streams sind **parallelisierbar**
- Die Elemente werden „on demand“ berechnet
- Sind „lazy“ (wird erst berechnet, wenn es gebraucht wird)
- Jeder Zwischenschritt liefert wieder einen Stream als Ergebnis
  - Verkettung „chaining“ von Operationen möglich
  - Endgültige Operationen markieren das Ende des Streams und liefern das Ergebnis.
- Ähneln den `Sequences` aus Hausübung 09 (die diese von Kotlin geklaut hat)

Was sind Streams?

Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

„Jedes n-te Wort“

Ansatz ohne Streams

```java
public static String[] everyNthWordIterative(int n, String[] words) {
    String[] result = new String[(int) Math.ceil((double) words.length / n)];
    for (int i = 0, s = 0; i < result.length; i++, s += n) {
        result[i] = words[s];
    }
    return result;
}
```

Ansatz mit Streams

```java
public static String[] everyNthWordStream(int n, String[] words) {
    return IntStream
        .range(0, words.length)
        .filter(x -> x % n == 0)
        .mapToObj(x -> words[x])
        .toArray(String[]::new);
}
```

Das steht heute auf dem Plan

TECHNISCHE UNIVERSITÄT DARMSTADT

Wiederholung

Was sind Streams?

Warum Streams?
- Vorteile von Streams
- Arrays vs Listen vs Streams

Umgang mit Streams

Code-Style

Warum Streams?

Vorteile von Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

- Deklarativer Style
- Funktionaler Ansatz
- Einfacher zu lesen (meiner Meinung Nach)
- Können unendlich groß sein
- Einfache (automatische) Parallelisierung
- Einfache Fehlerbehandlung durch `Optionals`

Warum Streams?

Vorteile von Streams — Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

„Nur Gerade Zahlen“

```java
int[] numbers = new int[]{-1, 4, -25, 42, -69, 1337};
```

Ansatz mit for-Schleifen

```java
int evenNumberCount = 0;
for(int n : numbers){
    if(n % 2 == 0){
        evenNumberCount++;
    }
}
evenNumbers = new int[evenNumberCount];
for(int i=0,j=0;i<numbers.length;i++) {
    if(numbers[i] % 2 == 0) {
        evenNumbers[j++] = numbers[i];
    }
}
```

Ansatz mit Streams

```java
int[] evenNumbers = Arrays
    .stream(numbers)
    .filter(x -> x % 2 == 0)
    .toArray();
```

Warum Streams?

Arrays vs Listen vs Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

Eigenschaft	Arrays	Listen	Streams
Gesamtgröße	fest	dynamisch	dynamisch/unendlich
Wiederverwendbarkeit	beliebig	beliebig	nur einmal
Hinzufügen v. Elementen	nein	ja	nein

Tabelle: Arrays vs Listen vs Streams

Das steht heute auf dem Plan

TECHNISCHE UNIVERSITÄT DARMSTADT

Wiederholung

Was sind Streams?

Warum Streams?

Umgang mit Streams
- Arten von Streams
- Wie erstelle ich einen Stream?
- Methoden von Streams
- Streams und Optionals
- Weitere Tricks

Code Style

„Klingt gut, aber wie erstelle ich jetzt einen Stream?“

Umgang mit Streams

Arten von Streams

TECHNISCHE UNIVERSITÄT DARMSTADT

- Stream<T>
- IntStream
- LongStream
- DoubleStream

$\Rightarrow$ also **kein** Stream für char, da dieser als int repräsentiert wird

Umgang mit Streams

Wie erstelle ich einen Stream?

TECHNISCHE UNIVERSITÄT DARMSTADT

Streams erstellen

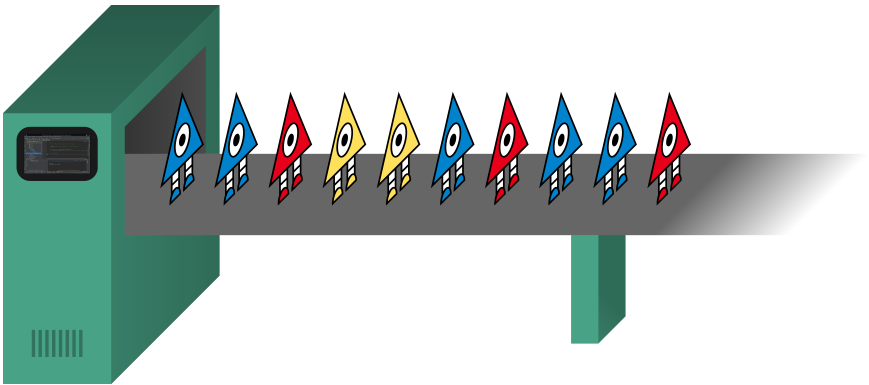
```java
// From Array
int[] array = new int[] {3, 4, 5};
IntStream stream = Arrays.stream(array);
int[] array2 = new String[];
Stream<String> stream2 = Arrays.stream(array2);
// From List
Stream<String> stream3 = new ArrayList<String>().stream();
// From varargs
Stream<Integer> stream4 = Stream.of(7, 4, 3); // Since java 9
// Range (intStream)
IntStream stream5 = IntStream.range(1, 11);
```

„Und was kann ich damit alles machen?“

Umgang mit Streams

Methoden von Streams — filter

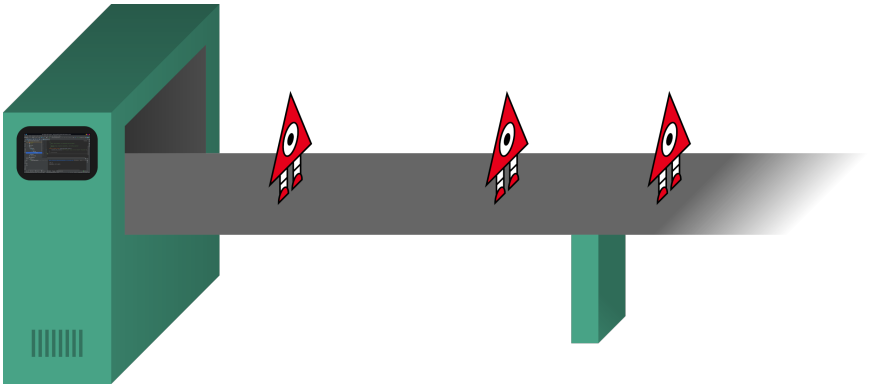
TECHNISCHE UNIVERSITÄT DARMSTADT



Umgang mit Streams

Methoden von Streams — filter

TECHNISCHE UNIVERSITÄT DARMSTADT



Umgang mit Streams

Methoden von Streams — filter

TECHNISCHE UNIVERSITÄT DARMSTADT

„Nur Gerade Zahlen“

```java
int[] numbers = new int[]{-1, 4, -25, 42, -69, 1337};
```

Ansatz mit for-Schleifen

```java
int evenNumberCount = 0;
for(int n : numbers){
    if(n % 2 == 0){
        evenNumberCount++;
    }
}
evenNumbers = new int[evenNumberCount];
for(int i=0,j=0;i<numbers.length;i++) {
    if(numbers[i] % 2 == 0) {
        evenNumbers[j++] = numbers[i];
    }
}
```

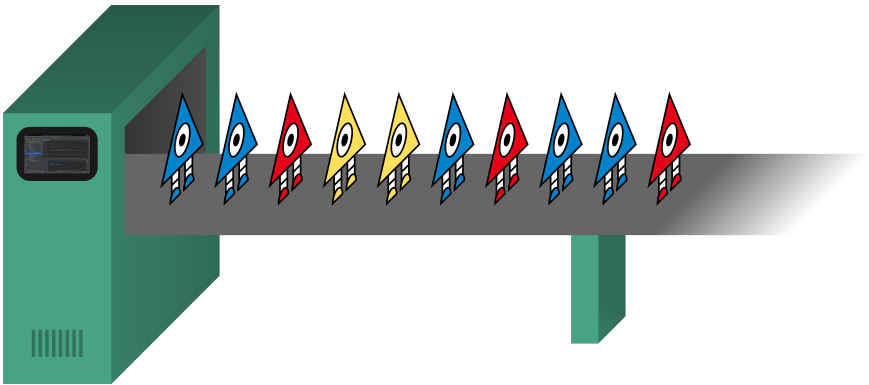
Ansatz mit Streams

```java
int[] evenNumbers = Arrays
    .stream(numbers)
    .filter(x -> x % 2 == 0)
    .toArray();
```

Umgang mit Streams

Methoden von Streams — reduce

TECHNISCHE UNIVERSITÄT DARMSTADT



Umgang mit Streams

Methoden von Streams — reduce

TECHNISCHE UNIVERSITÄT DARMSTADT

Umgang mit Streams

Methoden von Streams — reduce

TECHNISCHE UNIVERSITÄT DARMSTADT

„Die Summe aller Elemente eines Arrays“

```java
int[] numbers = new int[]{-1, 4, -25, 42, -69, 1337};
```

Ansatz mit for-Schleifen

```java
int sum = 0;
for(int n : numbers) {
    sum += n;
}
```

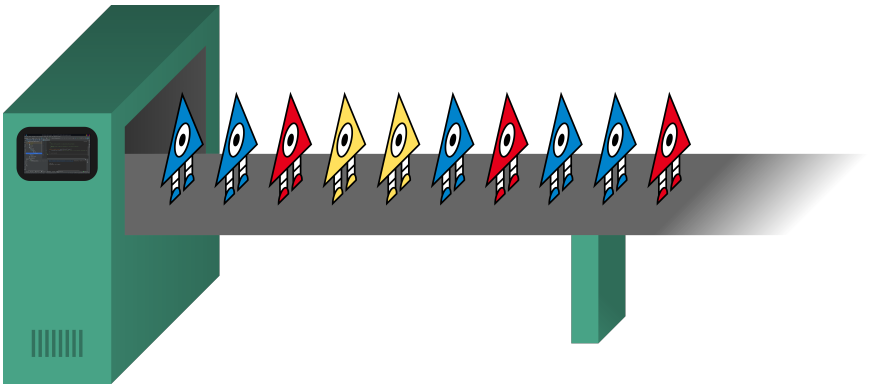
Ansatz mit Streams

```java
int sum = Arrays
    .stream(numbers)
    .reduce(0, (x,y) -> x + y);
```

Umgang mit Streams

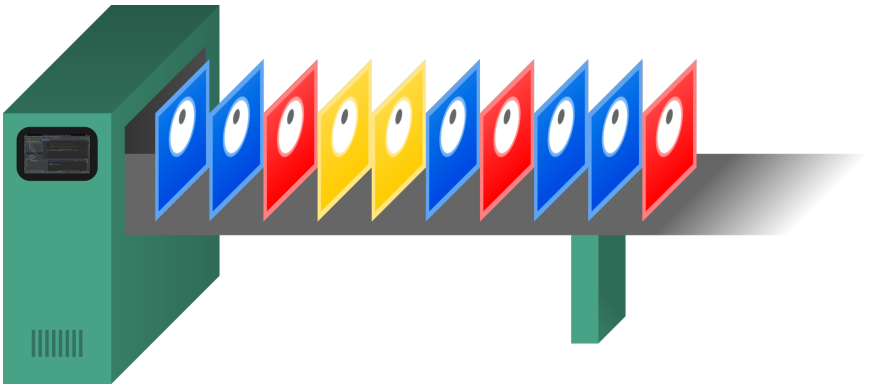
Methoden von Streams — map

TECHNISCHE UNIVERSITÄT DARMSTADT



Umgang mit Streams

Methoden von Streams — map



Umgang mit Streams

Methoden von Streams — map

TECHNISCHE UNIVERSITÄT DARMSTADT

„Alle Elemente um 20 erhöhen“

```
1  int[] numbers = new int[]{-1, 4, -25, 42, -69, 1337};
```

Ansatz mit for-Schleifen

```
1  int[] result = new int[numbers.length];
2  for (int i = 0; i < numbers.length; i++) {
3      result[i] = numbers[i] + 20;
4  }
```

Ansatz mit Streams

```
1  int[] result = Arrays
2      .stream(numbers)
3      .map(x -> x + 20)
4      .toArray();
```

```
$  result ==> int[6] { 19,24,-5,62,-49,1357 }
```

Umgang mit Streams

Methoden von Streams — flatMap

TECHNISCHE UNIVERSITÄT DARMSTADT

„Alle Einzelwörter aus mehrzeiligem String extrahieren“

```
1  String sentences = "Hello world\n" +
2  "Lorem ipsum dolor sit amet\n" +
3  "Streams sind toll";
```

Ansatz mit for-Schleifen

```
1  List<String> words = new
   ↪  ArrayList<>();
2  for (String s :
   ↪  sentences.split("\n")) {
3      for (String w : s.split(" ")) {
4          words.add(w);
5      }
6  }
```

Ansatz mit Streams

```
1  List<String> words = Arrays
2      .stream(sentences.split("\n"))
3      .flatMap(sentence ->
       ↪  Stream.of(sentence.split("
       ↪  ")))
4      .collect(Collectors.toList());
```

Umgang mit Streams

Methoden von Streams — distinct

TECHNISCHE UNIVERSITÄT DARMSTADT

„Alle doppelten Wörter entfernen“

```
1  String text = "Hello hello world world world this is a a test";
```

Ansatz mit for-Schleifen

```
1  StringBuilder resultBuilder = new
   ↪  StringBuilder();
2  Set<String> words = new HashSet<>();
3  for (String w : text.split(" ")) {
4      if (!words.contains(w)) {
5          words.add(w);
6          result.append(w).append(" ");
7      }
8  }
9  String result = resultBuilder.toString();
```

Ansatz mit Streams

```
1  String result = Arrays
2      .stream(text.split(" "))
3      .distinct()
4      .collect(
5          Collectors.joining(" ")
6      );
```

```
$  result ==> "Hello hello
↪  world this is a test"
```

TECHNISCHE UNIVERSITÄT DARMSTADT

Kleinste Zahl finden

```java
int[] numbers = {5, 3, 8, 1, 2, 9, 4, 7, 6};
```

Ansatz mit for-Schleife

```java
int min = numbers[0];
for (int i = 1; i < numbers.length;
↪   i++) {
    if (numbers[i] < min) {
        min = numbers[i];
    }
}
```

Ansatz mit Streams

```java
int min = Arrays
    .stream(numbers)
    .min()
    .getAsInt();
```

```
$ min ==> 1
```

Umgang mit Streams

Methoden von Streams — min

TECHNISCHE UNIVERSITÄT DARMSTADT

Kleinste Zahl finden

```
1  int[] numbers = {5, 3, 8, 1, 2, 9, 4, 7, 6};
```

Ansatz mit for-Schleife

```
1  int min = numbers[0];
2  for (int i = 1; i < numbers.length;
   ↪  i++) {
3      if (numbers[i] < min) {
4          min = numbers[i];
5      }
6  }
```

Ansatz mit Streams

```
1  int min = Arrays
2      .stream(numbers)
3      .min((a, b) -> a - b)
4      .getAsInt();
```

```
$  min ==> 1
```

TECHNISCHE UNIVERSITÄT DARMSTADT

Zahl mit kleinster Quersumme finden

```java
int[] numbers = {13, 538, 81, 142, 1337, 69, 42};
```

Ansatz mit for-Schleife

```java
int min = 0;
int quersummeMin = Integer.MAX_VALUE;
for (int n : numbers) {
    int sum = 0, temp = n;
    while (temp > 0) {
        sum += temp % 10;
        temp /= 10;
    }
    if (sum < quersummeMin) {
        min = n;
        quersummeMin = sum;
    }
}
```

Ansatz mit Streams

```java
int min = Arrays.stream(numbers)
.mapToObj(Integer::valueOf).min(
    Comparator.comparingInt(n ->
    Integer.toString(n)
        .chars()
        .map(Character::getNumericValue)
        .sum()
    )
).get()/*.intValue()*/;
```

```
$  min ==> 13
```

TECHNISCHE UNIVERSITÄT DARMSTADT

Größte Zahl finden

```java
int[] numbers = {5, 3, 8, 1, 2, 9, 4, 7, 6};
```

Ansatz mit for-Schleife

```java
int max = numbers[0];
for (int i = 1; i < numbers.length;
↪   i++) {
    if (numbers[i] > max) {
        max = numbers[i];
    }
}
```

Ansatz mit Streams

```java
int max = Arrays
    .stream(numbers)
    .max()
    .getAsInt();
```

```
$  max ==> 9
```

Umgang mit Streams

Methoden von Streams — sorted

TECHNISCHE UNIVERSITÄT DARMSTADT

Elemente sortieren

```java
int[] numbers = {5, 3, 8, 1, 2, 9, 4, 7, 6};
```

Ansatz mit for-Schleife

```java
for (int i = 0; i < numbers.length; i++) {
    for (int j = i + 1; j <
    ↪  numbers.length; j++) {
        if (numbers[i] > numbers[j]) {
            int temp = numbers[i];
            numbers[i] = numbers[j];
            numbers[j] = temp;
        }
    }
}
```

Ansatz mit Streams

```java
int[] sorted = Arrays
    .stream(numbers)
    .sorted()
    .toArray();
```

```
$  sorted ==>
↪  [1,2,3,4,5,6,7,8,9]
```

Umgang mit Streams

Methoden von Streams — foreach

TECHNISCHE UNIVERSITÄT DARMSTADT

„Zeile für Zeile ausgeben“

```
1  String text = "Hello\nWorld\n!";
```

Ansatz mit for-Schleife

```
1  for (String line : text.split("\n")) {
2      System.out.println(line);
3  }
```

Ansatz mit Streams

```
1  Arrays
2      .stream(text.split("\n"))
3      .forEach(System.out::println);
```

Achtung: Die Reihenfolge der Ausführung ist **nicht** garantiert! Wenn die Reihenfolge wichtig ist, kann stattdessen `forEachOrdered` verwendet werden.

Umgang mit Streams

Methoden von Streams — peek

TECHNISCHE UNIVERSITÄT DARMSTADT

Hilfreich für Debugging:

peek()

```
1  Stream.of("one", "two", "three", "four")
2  .filter(e -> e.length() > 3)
3  .peek(e -> System.out.println("Filtered value: " + e))
4  .map(String::toUpperCase)
5  .peek(e -> System.out.println("Mapped value: " + e))
6  .toList();
```

```
$  Filtered value: three
            $  Mapped value: THREE
            $  Filtered value: four
            $  Mapped value: FOUR
```

TECHNISCHE UNIVERSITÄT DARMSTADT

Erzeugen von unendlichen Streams:

generate()

```
1  Stream.generate(Math::random) // Math::random == () ->Math.random()
2  .forEach(System.out::println);
```

```
$  0.123456789
           $  0.987654321
           $  0.123456789
           $  0.987654321
           $  0.123456789
           # ... (würde unendlich weitergehen)
```

TECHNISCHE UNIVERSITÄT DARMSTADT

Erzeugen von exakt 5 Zufallszahlen:

generate()

```
1  Stream.generate(Math::random)
2  .limit(5)
3  .forEach(System.out::println);
```

```
$  0.123456789
$  0.987654321
$  0.123456789
$  0.987654321
$  0.123456789
```

TECHNISCHE UNIVERSITÄT DARMSTADT

Erzeugen von unendlichen Streams der Serlo-Reihe:

iterate()

```
1  Stream.iterate(new double[]{1, 1}, t -> new double[]{t[0] + 1d/(t[1] * 2),
   ↪  t[1]*2})
2      .limit(10)
3      .map(t -> t[0])
4      .forEach(n -> System.out.print(n + " "));
```

```
$  1.0 1.5 1.75 1.875 1.9375 1.96875 1.984375 1.9921875 1.99609375
 ↪  1.998046875
```

TECHNISCHE UNIVERSITÄT DARMSTADT

Solange mit zwei multiplizieren, wie die Zahl kleiner als 1000 ist:

takeWhile()

```
1  Stream.iterate(1, n -> n * 2)
2      .takeWhile(n -> n < 1000)
3      .forEach(n -> System.out.print(n + " "));
```

```
$  1 2 4 8 16 32 64 128 256 512
```

⇒ Ähnlich zu `limit()`, aber durch Lambda noch flexibler.

TECHNISCHE UNIVERSITÄT DARMSTADT

Erzeugen von Serlo-Reihe, aber die ersten 50 überspringen:

skip()

```
1  Stream.iterate(new double[]{1, 1}, t -> new double[]{t[0] + 1d/(t[1] * 2),
   ↪  t[1]*2})
2      .skip(50)
3      .limit(5)
4      .map(t -> t[0])
5      .forEach(n -> System.out.print(n + " "));
```

```
$  1.9999999999999991 1.9999999999999996 1.9999999999999998 2.0
↪   2.0
```

Wie oft muss man mit zwei multiplizieren, bis die Zahl größer als 1000 ist?

count()

```
1  long count = Stream.iterate(1, n -> n * 2)
2      .takeWhile(n -> n < 1000)
3      .count();
```

```
$  count ==> 9
```

Umgang mit Streams

Streams und Optionals — Was waren nochmal Optionals?

TECHNISCHE UNIVERSITÄT DARMSTADT

> **Optionals** sind ein Wrapper für Werte, die entweder vorhanden oder nicht vorhanden sind. – Github Copilot

- Hilfsklasse, um **null**-Werte zu vermeiden
- Bietet Methoden für einfaches **null**-Handling
- Wichtige Methoden: `isPresent()`, `get()`, `orElse()`, `orElseGet()`, `orElseThrow()`

Umgang mit Streams

Streams und Optionals — findFirst und findAny

TECHNISCHE UNIVERSITÄT DARMSTADT

Finde die erste Zweierpotenz, die größer als 1000 ist:

findFirst()

```
1  Optional<Integer> first = Stream.iterate(1, n -> n * 2)
2      .filter(n -> n > 1000)
3      .findFirst();
4  first.ifPresent(System.out::println);
```

```
$ 1024
```

- `findFirst()` liefert das erste Element des Streams
- `findAny()` liefert ein beliebiges Element des Streams zurück, welches nicht unbedingt das erste ist (z.B. bei parallelen Streams)

TECHNISCHE UNIVERSITÄT DARMSTADT

Genieriere 10 Zufallszahlen zwischen 0 und 1000 und gib die erste aus, die größer als 500 ist. Wenn keine Zahl größer als 500 ist, gib -1 aus:

orElse()

```
1  int first = new Random().ints(10, 0, 1000) // liefert IntStream mit 10
   ↪  Zufallszahlen
2      .filter(n -> n > 500)
3      .findFirst()
4      .orElse(-1);
5  System.out.println(first);
```

run 1:

```
$ 523
```

run 2:

```
$ -1
```

Genieriere 10 Zufallszahlen zwischen 0 und 1000 und gib die erste aus, die größer als 500 ist. Wenn keine Zahl größer als 500 ist, wirf eine Exception:

orElse()

```
1  int first = new Random().ints(10, 0, 1000) // liefert IntStream mit 10
   ↪  Zufallszahlen
2      .filter(n -> n > 500)
3      .findFirst()
4      .orElseThrow(() -> new RuntimeException("Keine Zahl war größer als 500"));
5  System.out.println(first);
```

run 1:

```
$ 523
```

run 2 (passt nicht ganz auf die Folie):

```
$ Exception in thread "main"
  ↪  java.lang.RuntimeException:
  ↪  Keine Zahl war größer als
```

Umgang mit Streams

Streams und Optionals — allMatch, anyMatch, noneMatch

TECHNISCHE UNIVERSITÄT DARMSTADT

Prüfe, ob alle Zahlen im Stream größer als 500 sind:

allMatch()

```
1  boolean all = new Random().ints(10, 0, 1000) // liefert IntStream mit 10
   ↪  Zufallszahlen
2      .allMatch(n -> n > 500);
3  System.out.println(all);
```

```
$ false
```

Umgang mit Streams

Streams und Optionals — allMatch, anyMatch, noneMatch

TECHNISCHE UNIVERSITÄT DARMSTADT

Prüfe, ob mindestens eine Zahl im Stream größer als 500 ist:

anyMatch()

```
1  boolean any = new Random().ints(10, 0, 1000) // liefert IntStream mit 10
   ↪  Zufallszahlen
2      .anyMatch(n -> n > 500);
3  System.out.println(any);
```

```
$  true
```

Umgang mit Streams

Streams und Optionals — allMatch, anyMatch, noneMatch

TECHNISCHE UNIVERSITÄT DARMSTADT

Prüfe, ob keine Zahl im Stream größer als 500 ist:

noneMatch()

```
1  boolean none = new Random().ints(10, 0, 1000) // liefert IntStream mit 10
   ↪  Zufallszahlen
2      .noneMatch(n -> n > 500);
3  System.out.println(none);
```

```
$  false
```

Umgang mit Streams

Weitere Tricks — Klasse Predicate

TECHNISCHE UNIVERSITÄT DARMSTADT

Predicate<T> ist eine Funktion, die ein Objekt vom Typ T entgegennimmt und einen „boolean“ zurückgibt.
Für Streams:

- `filter()` nimmt z.B. ein Predicate entgegen
- Praktische Methoden: `and()`, `or()`,`not()`, `isEqual()`

Alle nicht-leeren Strings ausgeben:

Predicate

```
1  List<String> strings = Arrays.asList("a", "", "b", "", "c", "d", "e");
2  strings.stream()
3      // .filter(s ->!s.isEmpty()) // unschön
4      .filter(Predicate.not(String::isEmpty)) // schöner
5      .forEach(s -> System.out.println(s + " "));
```

```
$ a b c d e
```

Umgang mit Streams

Weitere Tricks — Klasse Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

Comparator<T> ist eine Funktion, die zwei Objekte vom Typ T entgegennimmt und einen „int“ zurückgibt.
Für Streams:

- `sorted()` nimmt z.B. ein Comparator entgegen
- Wichtig: `Comparator.comparing()` und `Comparator.comparingInt()`

Umgang mit Streams

Weitere Tricks — Klasse Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

Zahl mit kleinster Quersumme ausgeben:

```
1  int[] numbers = { 123, 456, 789, 12, 34, 56, 78, 90, 1234, 5678, 9012 };
```

Unschöner Ansatz

```
1  int min = Arrays.stream(numbers)
2      .mapToObj(Integer::valueOf)
3      .min((n1, n2) -> {
4          int sum1 = Integer.toString(n1)
5              .chars()
6            .map(Character::getNumericValue)
7              .sum();
8          int sum2 = Integer.toString(n2)
9              .chars()
10           .map(Character::getNumericValue)
11             .sum();
12         return sum1 - sum2;
13     }).get()/*.intValue()*/;
```

Schöner Ansatz

```
1  int min = Arrays.stream(numbers)
2  .mapToObj(Integer::valueOf).min(
3      Comparator.comparingInt(n ->
4      Integer.toString(n)
5          .chars()
6          .map(Character::getNumericValue)
7          .sum()
8      )
9  ).get()/*.intValue()*/;
```

`Collectors` ist eine Klasse, die viele nützliche Methoden zum Sammeln von Daten in Streams bereitstellt. Die wichtigsten:

- `.toList()`, `.toSet()`, `.toMap()`
- `.joining()`

Liste in Format [a, b, c, ...] darstellen:

Collectors.joining()

```
1  List<String> strings = Arrays.asList("a", "b", "c", "d", "e");
2  String joined = strings.stream()
3      .collect(Collectors.joining(",", "[", "]"));
4  System.out.println(joined);
```

```
$ [a,b,c,d,e]
```

Das steht heute auf dem Plan

TECHNISCHE UNIVERSITÄT DARMSTADT

Wiederholung

Was sind Streams?

Warum Streams?

Umgang mit Streams

Code-Style
- Do’s
- Don’ts
- Beispiel: Übungsblatt 00 (2022/2023) in einer Zeile

Code-Style
Do's

TECHNISCHE UNIVERSITÄT DARMSTADT

- Mehrfaches nutzen von Methoden wie `map()`, `filter()`, `reduce()` für bessere Lesbarkeit.
  - Es gibt keine/kaum Performance-Verluste durch dieses „chaining“
- Verwenden von `Optional` um NullpointerExceptions zu vermeiden
- Verwenden von `peek()` zum Debugging von Streams
- Verwenden des `::` Operators wenn möglich (z.B. `map(String::toUpperCase)`)

Code-Style
Don'ts

TECHNISCHE UNIVERSITÄT DARMSTADT

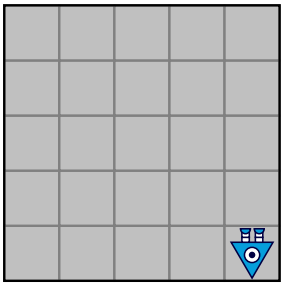
- Unnötige Operationen/Zwischenschritte (jede Operation kostet Zeit)
- Verwenden von `forEach()` in Streams zur Änderung von Daten.
- Verwenden von `count()` in Streams, wenn nur die Anzahl der Elemente benötigt werden
  - `list.stream().count()` muss einmal durch den **kompletten** Stream iterieren
  - `list.size()` ist hingegen **instantan**.
- Streams dazu missbrauchen, um Java als funktionales Programmiersprache zu verkaufen oder „Semikola zu sparen“ (siehe Beispiel auf den nächsten Folien)

Code-Style

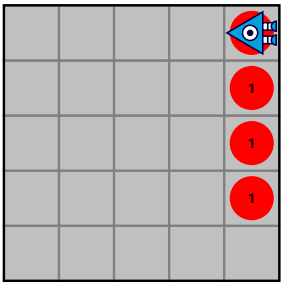
Beispiel: Übungsblatt 00 (2022/2023) in einer Zeile

TECHNISCHE UNIVERSITÄT DARMSTADT

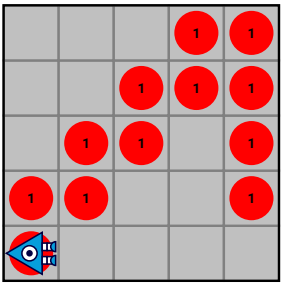
Aufgabenstellung: Roboter soll folgendes Bewegungsmuster ausführen:



(a) Startzustand: Der Roboter startet an Position (4,0) mit Blick nach unten.



(b) Zwischenzustand nach der ersten `for`-Schleife und nachfolgender Linksdrehung.



(c) Endzustand nach der zweiten `for`-Schleife.

Normaler Ansatz

```java
public static void doExercise() {
    Robot robby = new Robot(4, 0, DOWN,
    ↪  12);

    // <solution H4>
    // Zunächst drehen wir uns nach oben
    while (robby.getDirection() != UP) {
        robby.turnLeft();
    }
    // Dann laufen wir nach oben und legen
    ↪  nach jedem schritt eine Münze ab
    for (int i = 0; i < World.getHeight()
    ↪  - 1; i++) {
        robby.move();
        robby.putCoin();
    }
    // nach links schauen
    robby.turnLeft();

```

Normaler Ansatz

```java

    // jetzt gehen wir stufenweise nach
    ↪  unten links
    for (int i = 0; i < 4; i++) {
        // Schritt nach Links + Münze legen
        robby.move();
        robby.putCoin();

        // Nach unten schauen
        robby.turnLeft();

        // Schritt nach Unten
        robby.move();
        robby.putCoin();

        // Rechtsdrehung
        robby.turnLeft();
        robby.turnLeft();
        robby.turnLeft();
    }
    // </solution>
}
```

Ansatz in „einer Zeile“ (bitte **nicht** nachmachen)

```
1  public static void doExercise() {
2      final Robot robby = new Robot(4, 0, DOWN, 12);
3
4      // <solution H4>
5      Stream.<Runnable>of(() -> IntStream.range(1, World.getHeight()).forEach(i ->
       ↪  Stream.<Runnable>of(() -> Stream.generate(robby::getDirection).takeWhile(d ->
       ↪  d.ordinal() != 0).forEach(d -> robby.turnLeft()), robby::move,
       ↪  robby::putCoin).forEach(Runnable::run)), robby::turnLeft, () ->
       ↪  Stream.generate(robby::isFrontClear).takeWhile(x -> x).forEach((i) ->
       ↪  Stream.<Runnable>of(() -> Stream.generate(robby::getDirection).takeWhile(d ->
       ↪  d.ordinal() != 3).forEach(d -> robby.turnLeft()),robby::move,robby::putCoin,() ->
       ↪  Stream.generate(robby::getDirection).takeWhile(d -> d.ordinal() != 2).forEach(d ->
       ↪  robby.turnLeft()),robby::move,robby::putCoin).forEach(Runnable::run)),() ->
       ↪  Stream.generate(robby::getDirection).takeWhile(d -> d.ordinal() != 3).forEach(d ->
       ↪  robby.turnLeft())).forEach(Runnable::run); // Zeile ist trivial
6      // </solution>
7  }
```

Live-Coding!