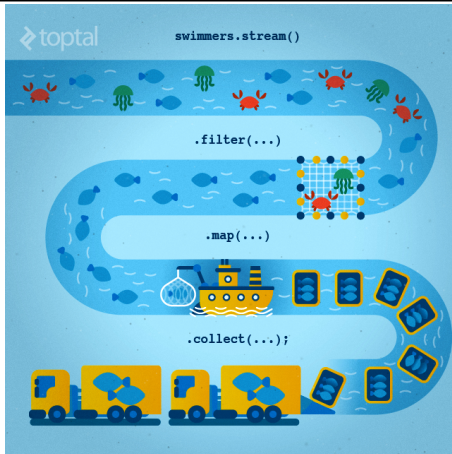


# FOP Recap #12



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

## 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



- 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 (param1, ..., paramn) -> rumpf
7 // Mehrere Anweisungen im Rumpf
8 (param1, ..., paramn) -> {
9     anweisung1
10    // ...
11    anweisungn // z.B. return
12 }
```



### Beispiel: „Liste Sortieren“

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



#### Sortieren ohne Lambda



```
1 numbers.sort(new  
2     ↳ Comparator<Integer>() {  
3         @Override  
4         public int compare(Integer a,  
5             ↳ Integer b) {  
6             return a - b;  
7         }  
8     });
```



#### Sortieren mit Lambda



```
1 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.<sup>1</sup>

- 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

---

<sup>1</sup><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



```
1 public static String[] everyNthWordIterative(int n, String[] words) {
2     String[] result = new String[(int) Math.ceil((double) words.length / n)];
3     for (int i = 0, s = 0; i < result.length; i++, s += n) {
4         result[i] = words[s];
5     }
6     return result;
7 }
```



### Ansatz mit Streams



```
1 public static String[] everyNthWordStream(int n, String[] words) {
2     return IntStream
3         .range(0, words.length)
4         .filter(x -> x % n == 0)
5         .mapToObj(x -> words[x])
6         .toArray(String[]::new);
7 }
```

# 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“

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



#### Ansatz mit for-Schleifen



```
1  int evenNumberCount = 0;
2  for(int n : numbers){
3      if(n % 2 == 0){
4          evenNumberCount++;
5      }
6  }
7  evenNumbers = new int[evenNumberCount];
8  for(int i=0,j=0;i<numbers.length;i++) {
9      if(numbers[i] % 2 == 0) {
10         evenNumbers[j++] = numbers[i];
11     }
12 }
```



#### Ansatz mit Streams



```
1  int[] evenNumbers = Arrays
2      .stream(numbers)
3      .filter(x -> x % 2 == 0)
4      .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





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

# Umgang mit Streams

## Arten von Streams



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- Stream<T>
- IntStream
- LongStream
- DoubleStream

⇒ also **kein** Stream für char, da dieser als int repräsentiert wird

# Umgang mit Streams

## Wie erstelle ich einen Stream?



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT



### Streams erstellen



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



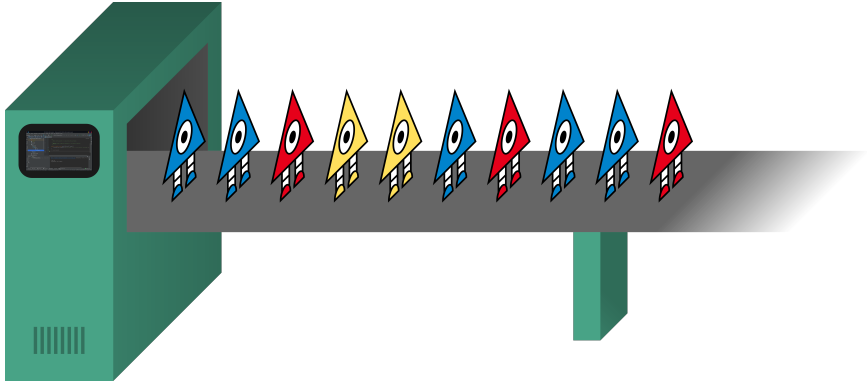
# „Und was kann ich damit alles machen?“

# 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“

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



#### Ansatz mit for-Schleifen



```
1  int evenNumberCount = 0;
2  for(int n : numbers){
3      if(n % 2 == 0){
4          evenNumberCount++;
5      }
6  }
7  evenNumbers = new int[evenNumberCount];
8  for(int i=0,j=0;i<numbers.length;i++) {
9      if(numbers[i] % 2 == 0) {
10         evenNumbers[j++] = numbers[i];
11     }
12 }
```



#### Ansatz mit Streams



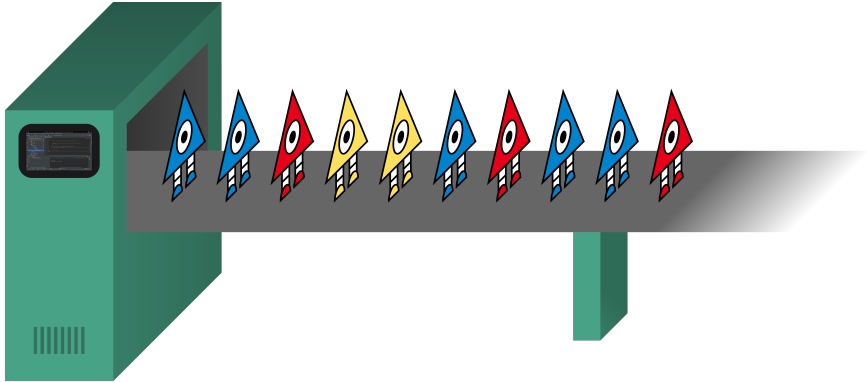
```
1  int[] evenNumbers = Arrays
2      .stream(numbers)
3      .filter(x -> x % 2 == 0)
4      .toArray();
```

# 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“

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



#### Ansatz mit for-Schleifen



```
1 int sum = 0;
2 for(int n : numbers) {
3     sum += n;
4 }
```



#### Ansatz mit Streams



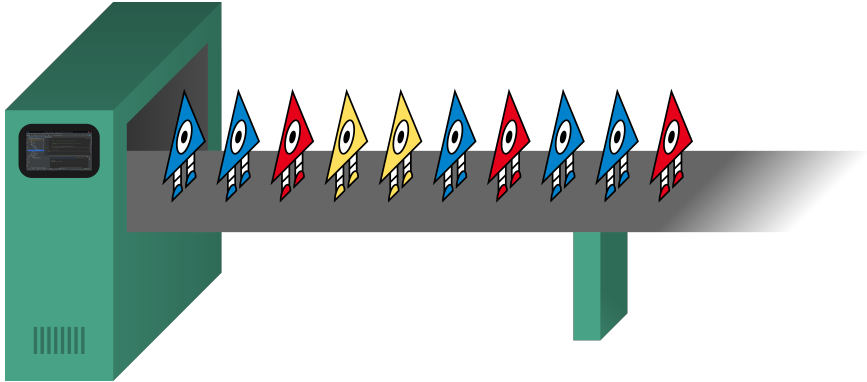
```
1 int sum = Arrays
2     .stream(numbers)
3     .reduce(0, (x,y) -> x + y);
```

# Umgang mit Streams

## Methoden von Streams – map



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

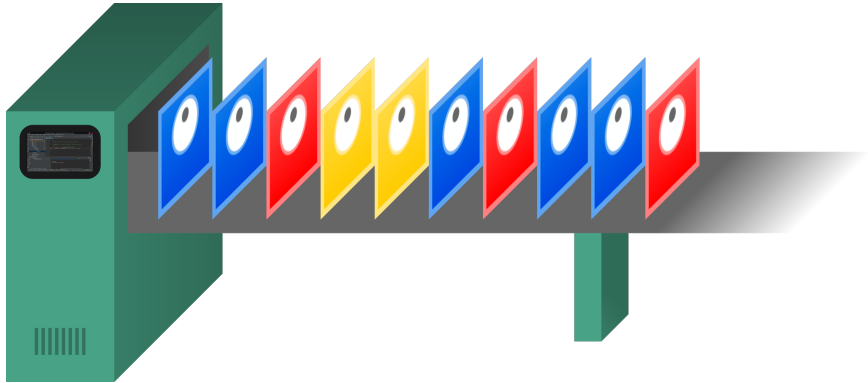


# Umgang mit Streams

## Methoden von Streams – map



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT



# 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());
```



### „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"
```



### 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()  
4     .getAsInt();
```

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





### 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
```



### Zahl mit kleinster Quersumme finden

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



#### Ansatz mit for-Schleife



```
1 int min = 0;
2 int quersummeMin = Integer.MAX_VALUE;
3 for (int n : numbers) {
4     int sum = 0, temp = n;
5     while (temp > 0) {
6         sum += temp % 10;
7         temp /= 10;
8     }
9     if (sum < quersummeMin) {
10         min = n;
11         quersummeMin = sum;
12     }
13 }
```



#### Ansatz mit Streams



```
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();
```

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

# Umgang mit Streams

## Methoden von Streams – max



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

### Größte Zahl finden

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



#### Ansatz mit for-Schleife



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



#### Ansatz mit Streams



```
1 int max = Arrays
2     .stream(numbers)
3     .max()
4     .getAsInt();
```

\$ max ==> 9



### Elemente sortieren

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



#### Ansatz mit for-Schleife



```
1 for (int i = 0; i < numbers.length; i++) {  
2     for (int j = i + 1; j <  
    ↪ numbers.length; j++) {  
3         if (numbers[i] > numbers[j]) {  
4             int temp = numbers[i];  
5             numbers[i] = numbers[j];  
6             numbers[j] = temp;  
7         }  
8     }  
9 }
```



#### Ansatz mit Streams



```
1 int[] sorted = Arrays  
2     .stream(numbers)  
3     .sorted()  
4     .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.



### 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
```



### 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)
```



### 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





### 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
```

# Umgang mit Streams

## Methoden von Streams – takeWhile



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.

# Umgang mit Streams

## Methoden von Streams – skip



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
```



**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)

# Umgang mit Streams

## Streams und Optionals — Error Handling



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Geniere 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

# Umgang mit Streams

## Streams und Optionals — Error Handling



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Geniere 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



`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
```



`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()`



### 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 ? n1 : n2;
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

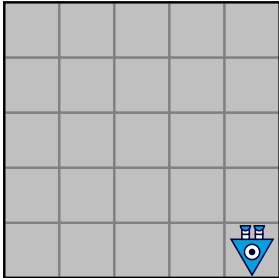


- Mehrfaches nutzen von Methoden wie `map()`, `filter()`, `reduce()` für bessere Lesbarkeit.
  - ▣ Es gibt keine/kaum Performance-Verluste durch dieses „chaining“
- Verwenden von `Optional` um `NullPointerExceptions` zu vermeiden
- Verwenden von `peek()` zum Debugging von Streams
- Verwenden des `::`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

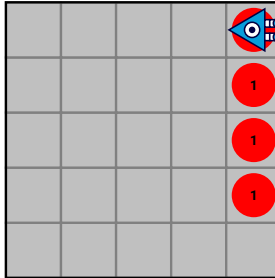


- Unnötige Operationen/Zwischenschritte (jede Operation kostet Zeit)
- Verwenden von `forEach()` in Streams zur Änderung von Daten.
- Verwenden von `count()` in Streams, wenn nur die Anzahl der Elemente benötigt werden
  - ▣ `list.stream().count()` muss einmal durch den **kompletten** Stream iterieren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

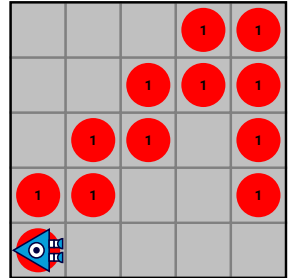
**Aufgabenstellung:** Roboter soll folgendes Bewegungsmuster ausführen:



(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



```
1 public static void doExercise() {
2     Robot robby = new Robot(4, 0, DOWN,
3         ↪ 12);
4
5     // <solution H4>
6     // Zunächst drehen wir uns nach oben
7     while (robby.getDirection() != UP) {
8         robby.turnLeft();
9     }
10    // Dann laufen wir nach oben und legen
11    ↪ nach jedem schritt eine Münze ab
12    for (int i = 0; i < World.getHeight()
13        ↪ - 1; i++) {
14        robby.move();
15        robby.putCoin();
16    }
17    // nach links schauen
18    robby.turnLeft();
19 }
```



## Normaler Ansatz



```
16
17     // jetzt gehen wir stufenweise nach
18     ↪ unten links
19     for (int i = 0; i < 4; i++) {
20         // Schritt nach Links + Münze legen
21         robby.move();
22         robby.putCoin();
23
24         // Nach unten schauen
25         robby.turnLeft();
26
27         // Schritt nach Unten
28         robby.move();
29         robby.putCoin();
30
31         // Rechtsdrehung
32         robby.turnLeft();
33         robby.turnLeft();
34         robby.turnLeft();
35     }
36     // </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!