Universität Bremen

Praktische Informatik 1 Funktionale Verarbeitung von Sammlungen

Thomas Röfer

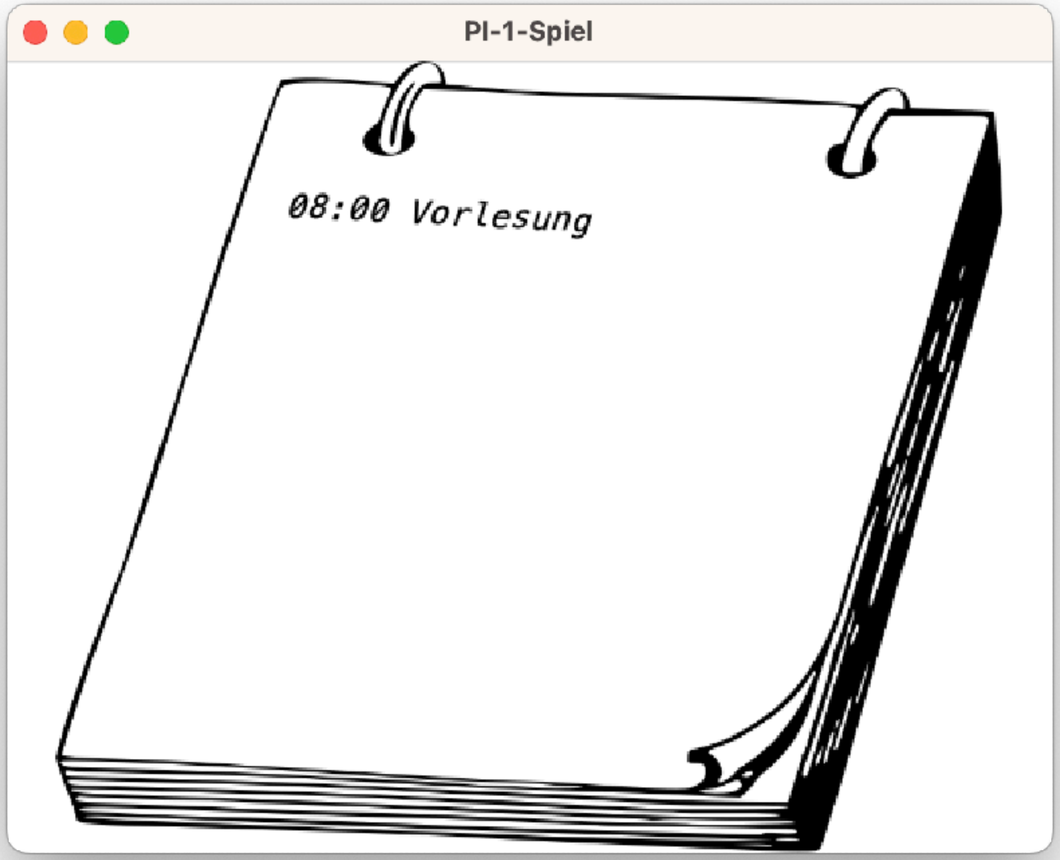
Cyber-Physical Systems
Deutsches Forschungszentrum für Künstliche Intelligenz

Multisensorische Interaktive Systeme
Fachbereich 3, Universität Bremen

Bild: ©Michael Ihle

Universität Bremen

Lambda-Ausdrücke: Demo



Universität Bremen

Lambda-Ausdrücke

- Gehen zurück auf das **λ-Kalkül** nach Alonzo Church (1930er)
- Repräsentieren **namenlose Funktionen**, die wie Werte behandelt, aber auch auf Werte angewendet werden können
- Viele Java-Methoden akzeptieren Lambda-Ausdrücke als Parameter (seit Java 8)

```
for (final String notiz : notizen) {
    darstellung.schreibe(notiz);
}
```

≙

```
notizen.forEach(notiz -> darstellung.schreibe(notiz));
```

Universität Bremen

Lambda-Ausdrücke: Schreibweisen

- Ein Lambda-Ausdruck (z.B. **(Notiz notiz) -> {return notiz.contains(suchtext);}**) besteht aus
  - Liste der formalen Parameter in runden Klammern, jeweils mit Typ und Name
  - Ein Pfeil und dann ein Methodenrumpf in geschweiften Klammern
- Kann der Compiler die Typen der Parameter automatisch ermitteln, können diese weggelassen werden, z.B. **(notiz) -> {return notiz.contains(suchtext);}**
- Gibt es nur einen Parameter ohne Typangabe, kann **( )** weggelassen werden, z.B. **notiz -> {return notiz.contains(suchtext);}**
- Enthält der Rumpf nur eine (**return**-)Anweisung, können **{ }**, **return** und **;** weggelassen werden, z.B. **notiz -> notiz.contains(suchtext)**
- **(**<Parameterliste>**) ->** <Klasse/Objekt>**.**<methode>**(**<selbe Parameterliste>**)** kann durch <Klasse/Objekt>**::**<methode> ersetzt werden (z.B. **darstellung::schreibe** statt **notiz -> darstellung.schreibe(notiz)**)

Universität Bremen

Lambda-Ausdrücke: Sichtbarkeit und Lebensdauer

- Übernehmen den Sichtbarkeitsraum aus der umgebenen Methode in ihren eigenen
- Sie können keine Variablen anlegen, die es in der umgebenen Methode schon gibt
- In ihnen deklarierte Variablen sind aber in der umgebenen Methode nicht sichtbar
- **Closure**: Sie können in der umgebenen Methode sichtbare Variablen verwenden
  - Diese müssen dazu **effektiv final** sein, d.h. sie sind entweder **final** oder das Programm würde auch mit **final** noch übersetzen
  - **this** ist das aus der umgebenen Methode

```
void entferneNotizenMit(final String suchtext)
{
    notizen.removeIf(notiz -> notiz.contains(suchtext));
}
```

Universität Bremen

Unterstützung von Lambda-Ausdrücken in Sammlungen

- **forEach(***Lambda-Ausdruck***)**
  - Führt den übergebenen Lambda-Ausdruck für jedes Element der Sammlung aus

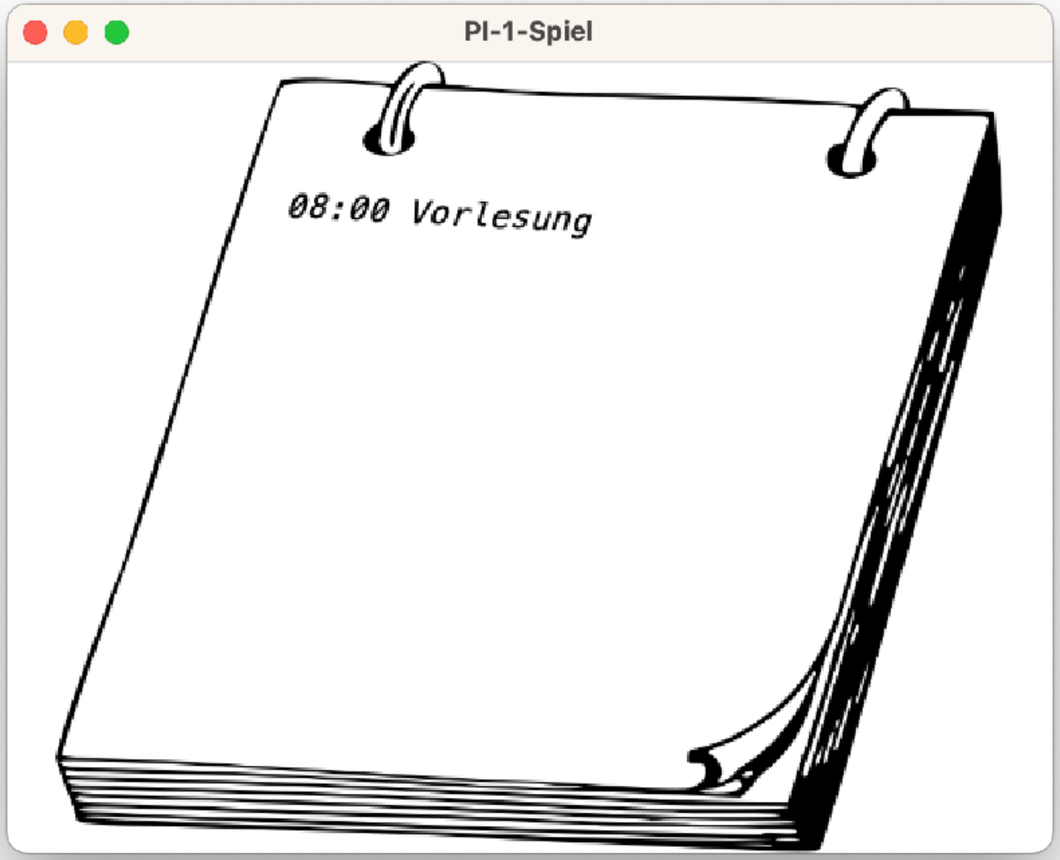
```
notizen.forEach(darstellung::schreibe);
```

- **stream()**
  - Liefert einen Datenstrom, den alle Elemente der Sammlung sequenziell durchlaufen

```
return notizen.stream()
        .filter(notiz -> notiz.contains(suchtext))
        .findFirst()
        .orElse(null);
```

Universität Bremen

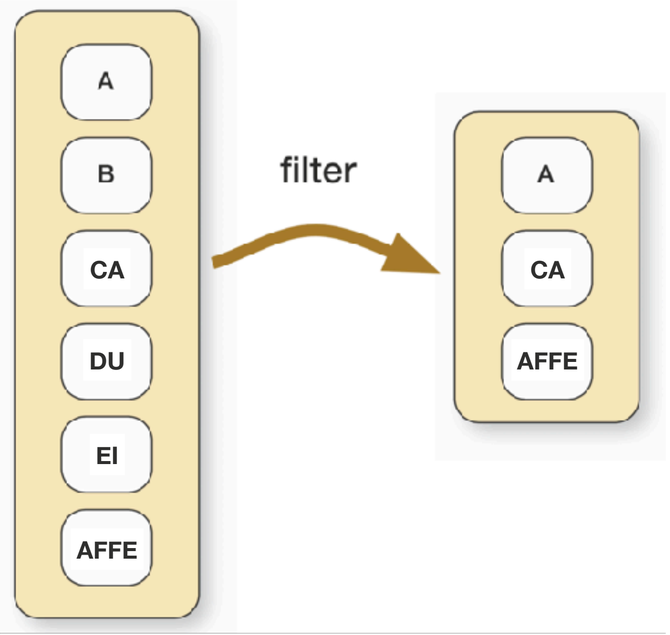
Streams: Demo



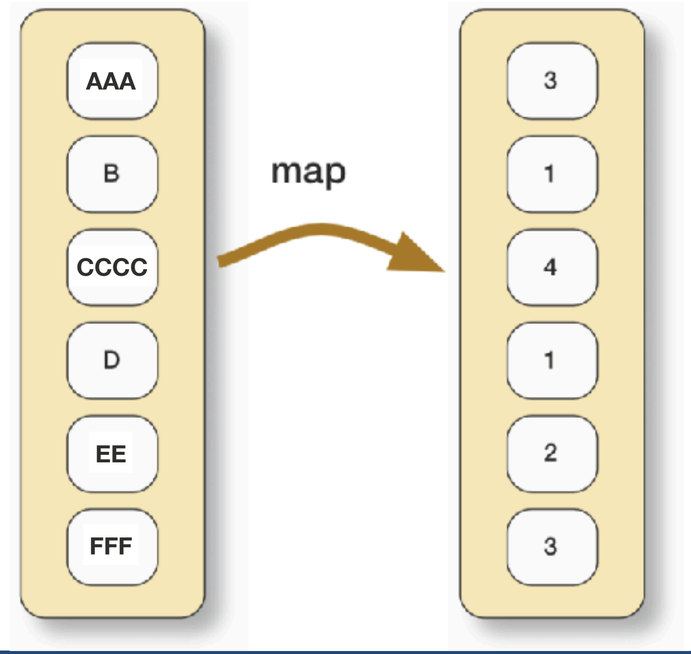
Universität Bremen

Streams: Elemente manipulieren

- **filter**: Filtert alle Elemente, so dass nur die weitergeleitet werden, die ein Kriterium erfüllen, z.B. **filter(notiz -> notiz.contains(suchtext))**

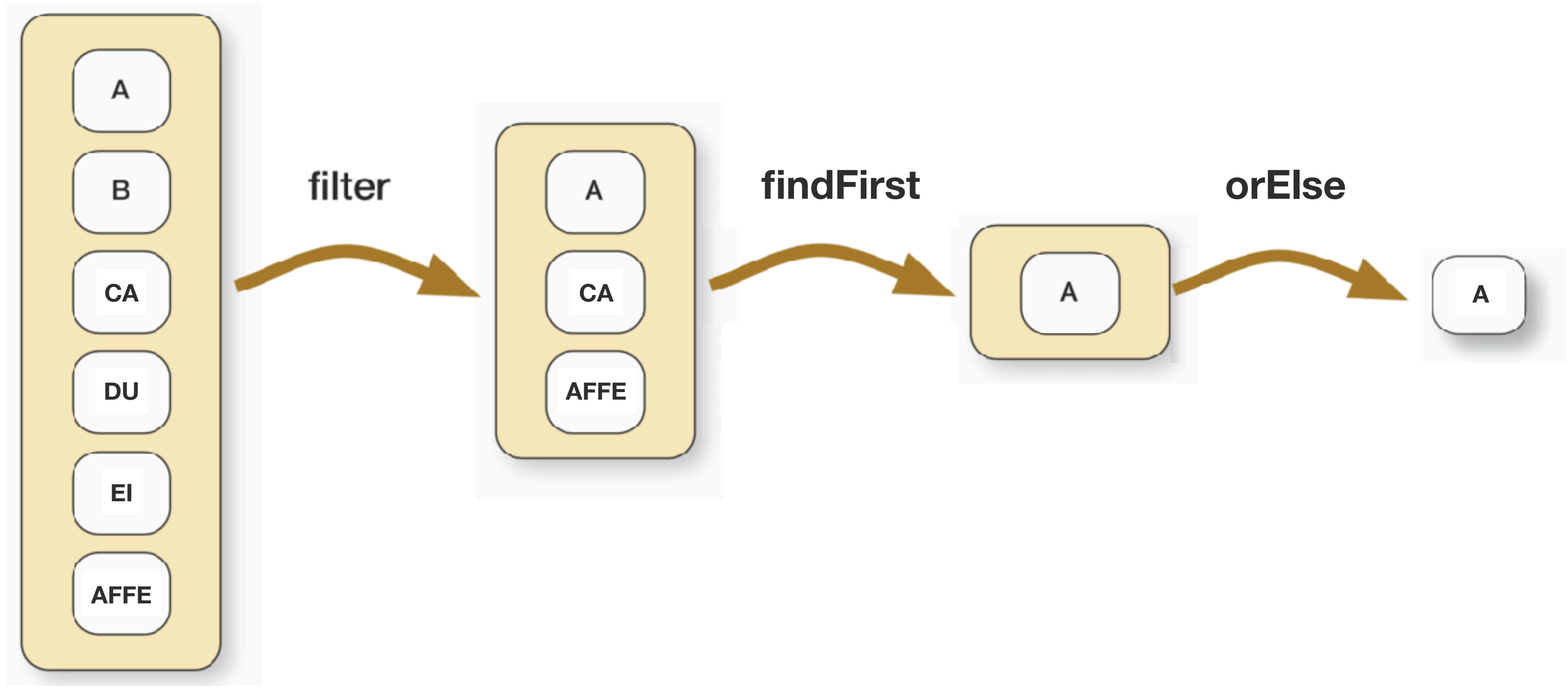


- **map**: Wendet eine Funktion auf alle Elemente des Datenstroms an und erzeugt damit einen neuen Datenstrom, z.B. **map(notiz -> notiz.length())**



Streams: Ergebnisse einsammeln

- Manipulatoren können verkettet werden (z.B. **map**, dann **filter**)
- Die Ergebnisse eines Streams werden am Ende eingesammelt
- **findFirst**: Liefert lediglich das erste Element des Datenstroms als **Optional**, das z.B. mit **get** oder **orElse** gelesen werden kann, z.B. **filter(notiz -> notiz.contains(suchtext)).findFirst().orElse(null)**
- **forEach**: Führt Funktion für alle Elemente aus



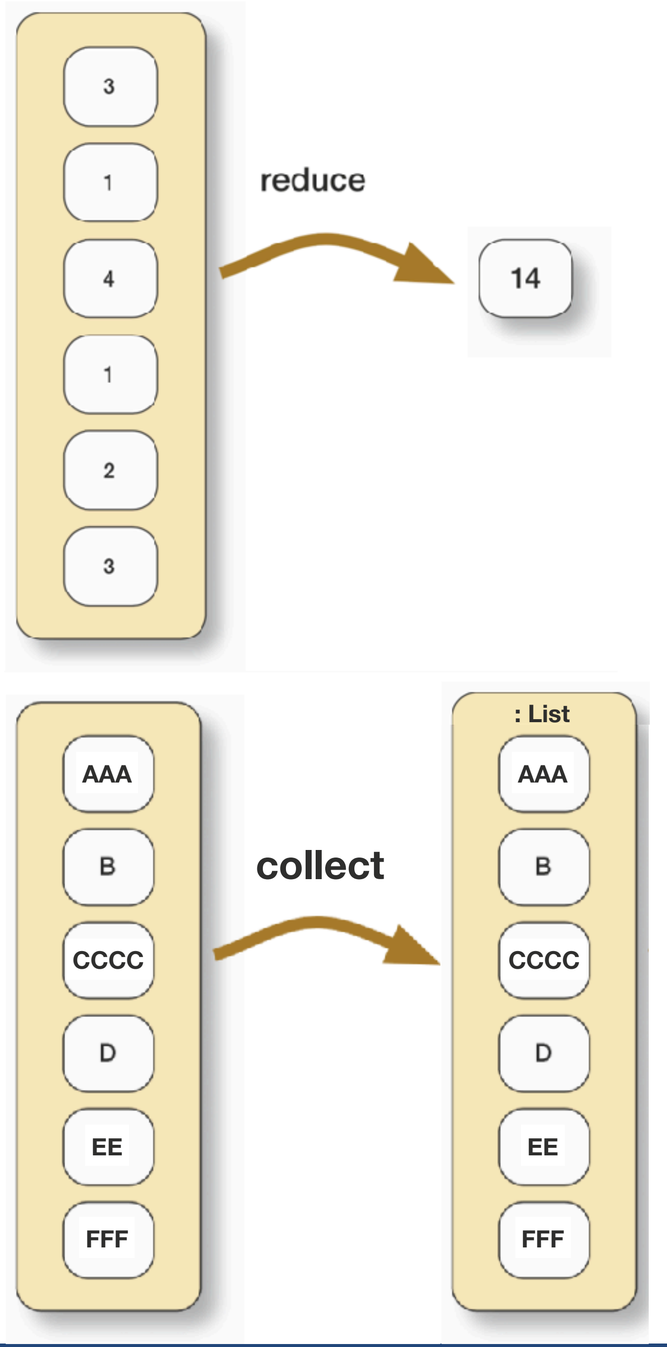

```
notizen.stream().forEach(darstellung::schreibe);
```

Streams: Ergebnisse einsammeln

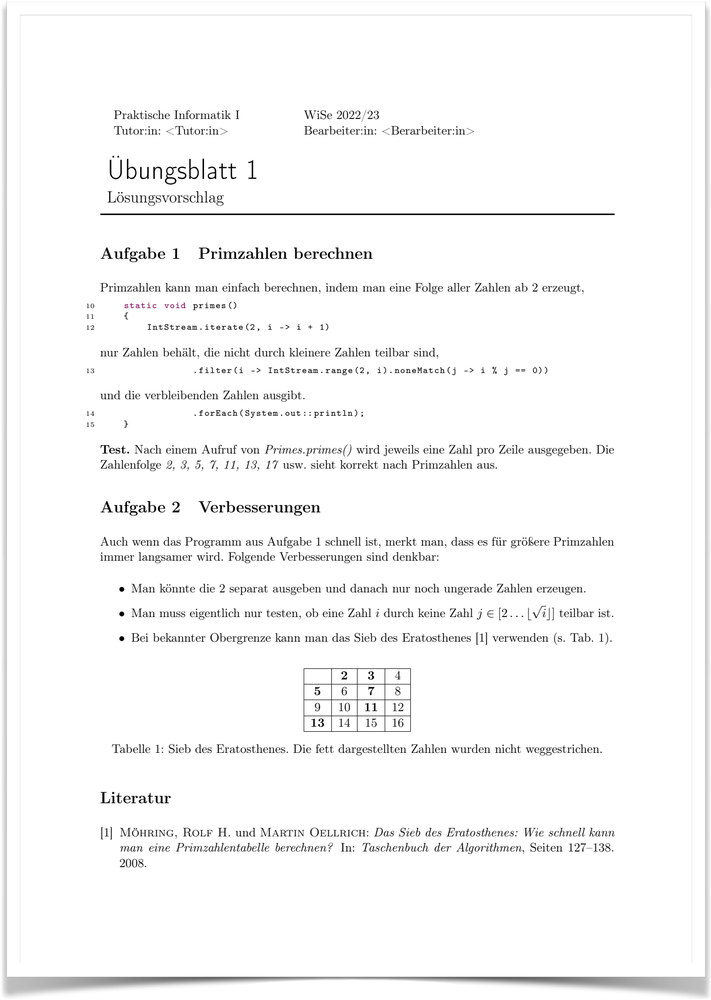
- **reduce**: Wendet für alle Elemente des Stroms, beginnend mit einem Startwert oder dem ersten Element, eine Funktion jeweils auf das bisherige Ergebnis und das nächste Element an, um ein neues Ergebnis zu erhalten (**Faltung**), z.B. **reduce(0, (a, b) -> a + b)**

- **collect**: Sammelt die Elemente mithilfe eines **Collectors** ein, um sie z.B. in einer Sammlung bereitzustellen, z.B. **collect(Collectors.toList())**

- **allMatch**, **noneMatch**, **anyMatch**: Erfüllen alle/kein/mindestens ein Element(e) eine Bedingung?



Andere Generatoren als Sammlungen: Demo



Praktische Informatik I — WiSe 2022/23
Tutor:in: <Tutor:in> — Bearbeiter:in: <Berarbeiter:in>

Übungsblatt 1

Lösungsvorschlag

Aufgabe 1 Primzahlen berechnen

Primzahlen kann man einfach berechnen, indem man eine Folge aller Zahlen ab 2 erzeugt,

```
10      static void primes()
11      {
12          IntStream.iterate(2, i -> i + 1)
```

nur Zahlen behält, die nicht durch kleinere Zahlen teilbar sind,

```
13                  .filter(i -> IntStream.range(2, i).noneMatch(j -> i % j == 0))
```

und die verbleibenden Zahlen ausgibt.

```
14                  .forEach(System.out::println);
15      }
```

Test. Nach einem Aufruf von *Primes.primes()* wird jeweils eine Zahl pro Zeile ausgegeben. Die Zahlenfolge *2, 3, 5, 7, 11, 13, 17* usw. sieht korrekt nach Primzahlen aus.

Aufgabe 2 Verbesserungen

Auch wenn das Programm aus Aufgabe 1 schnell ist, merkt man, dass es für größere Primzahlen immer langsamer wird. Folgende Verbesserungen sind denkbar:

- Man könnte die 2 separat ausgeben und danach nur noch ungerade Zahlen erzeugen.
- Man muss eigentlich nur testen, ob eine Zahl i durch keine Zahl $j \in [2 \dots \lfloor\sqrt{i}\rfloor]$ teilbar ist.
- Bei bekannter Obergrenze kann man das Sieb des Eratosthenes [1] verwenden (s. Tab. 1).

	2	**3**	4
5	6	**7**	8
9	10	**11**	12
13	14	15	16

Tabelle 1: Sieb des Eratosthenes. Die fett dargestellten Zahlen wurden nicht weggestrichen.

Literatur

[1] Möhring, Rolf H. und Martin Oellrich: *Das Sieb des Eratosthenes: Wie schnell kann man eine Primzahlentabelle berechnen?* In: *Taschenbuch der Algorithmen*, Seiten 127–138. 2008.

Universität Bremen

Streams: Andere Generatoren als Sammlungen

- Aus Generatoren von Streams selbst (Paket **java.util.stream**)
  - **IntStream.range(int, int)**: Strom von Zahlen vom ersten Parameter bis zur größten Zahl kleiner als der zweite Parameter, z.B. **IntStream.range(0, 4)**
  - **iterate**: Erzeugt Datenstrom, indem Funktion einmal mit Anfangswert und dann immer wieder mit dem Ergebnis des letzten Aufrufs aufgerufen wird (optional: solange Bedingung wahr), z.B. **IntStream.iterate(0, n -> n + 1)** oder **IntStream.iterate(0, n -> n < 4, n -> n + 1)**
  - **generate**: Erzeugt Datenstrom, indem die übergebene Funktion immer wieder aufgerufen wird, z.B. **DoubleStream.generate(Math::random)**
- Aus Array-Elementen mit **Arrays.stream(...)**, z.B. **Arrays.stream(new int[ ]{1, 2, 3})**
- Aus einem **BufferedReader** mit **lines()**

Universität Bremen

Funktionale Schnittstellen (functional interfaces)

- Sind Java-Schnittstellen mit genau einer zu implementierenden Methode
  - Methoden, die bereits in **Object** vorhanden sind, zählen dabei nicht mit

```
interface Comparator<T>
{
    int compare(T o1, T o2);
    boolean equals(Object obj);
}
```

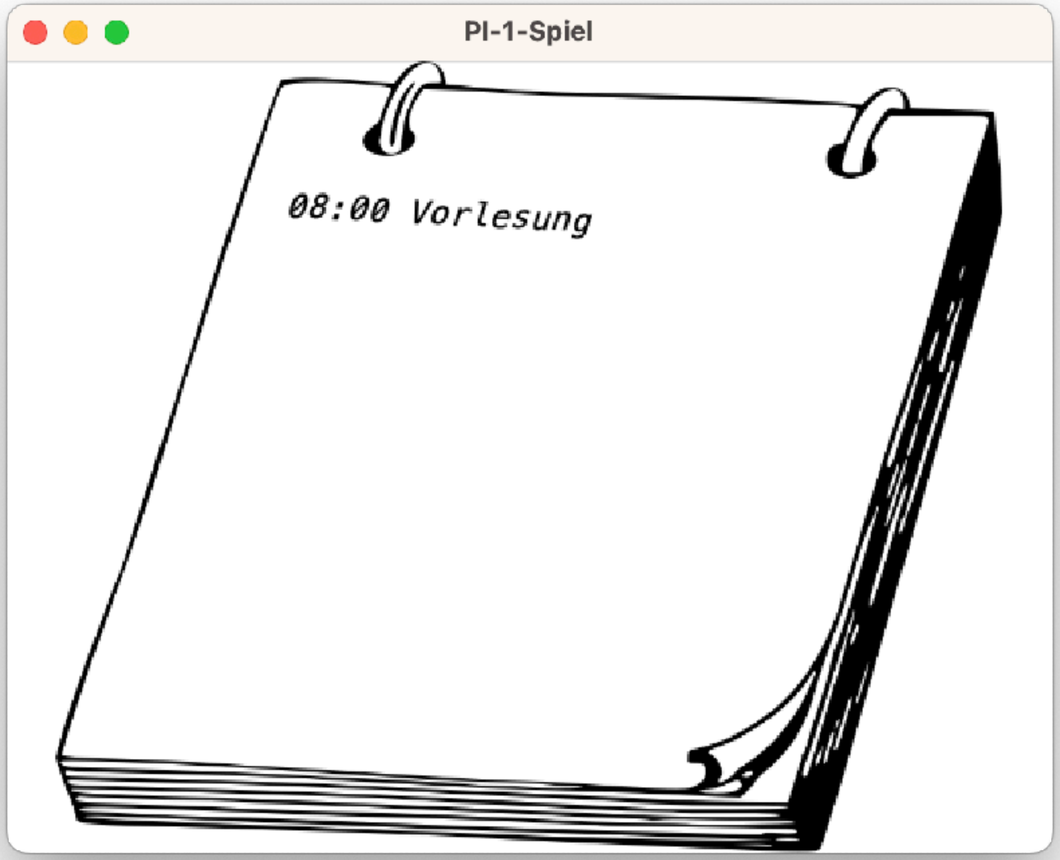
- An Stellen, wo ein Objekt erwartet wird, das eine funktionale Schnittstelle implementiert, kann auch ein Lambda-Ausdruck verwendet werden

```
static <T> void sort(T[ ] a, Comparator<? super T> c)
```

```
final Integer a[ ] = {2, 5, 3, 1, 4};
Arrays.sort(a, (i, j) -> j - i); // Absteigend
```

Universität Bremen

Funktionale Schnittstellen: Demo



Universität Bremen

Funktionale Schnittstellen: Vordefiniert

- Das Paket **java.util.function** enthält bereits viele vordefinierte funktionale Schnittstellen, die für eigene Implementierungen genutzt werden können, z.B.

- **Function<T, R>**, **BiFunction<T, U, R>**: Abbildung von Eingabewert(en) auf einen Ausgabewert (Methoden **R apply(T t)** bzw. **R apply(T t, U u)**), z.B. erwartet **Stream.map** eine **Function**

  - **UnaryOperator<T>**, **BinaryOperator<T>**: Genauso, aber alle Typen gleich

- **Predicate<T>**: Test eines Eingabewerts anhand einer Bedingung (Methode **boolean test(T t)**), z.B. erwarten **Stream.filter** und **Collection.removeIf** ein **Predicate**

  - Beispiel: **n -> n % 2 == 0** passt zu **Predicate<Integer>**

- **Supplier<T>**, **Consumer<T>**: Bereitstellen bzw. „Verbrauchen“ eines Werts (Methoden **T get()** bzw. **void accept(T t)**)

Zusammenfassung der Konzepte

- **Lambda-Ausdruck**
- **Stream**
- **filter**, **map**
- **reduce**, **collect**
- **range, iterate, generate**
- **Funktionale Schnittstelle**