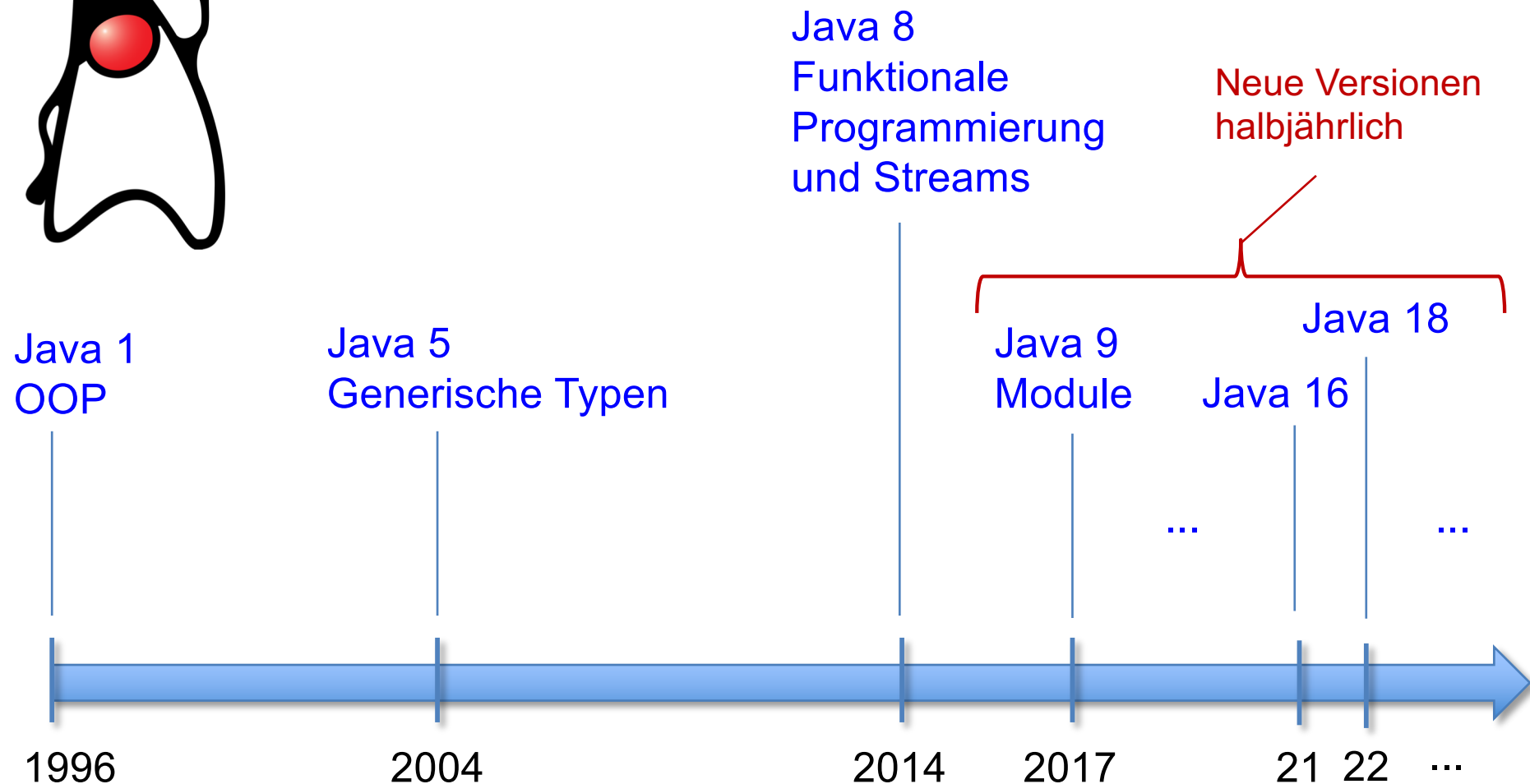
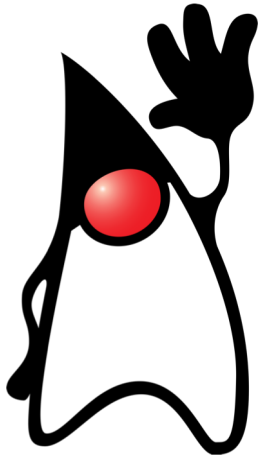


Kapitel 13: Funktionales Programmieren und Streams mit Java 8

- Java-Historie und Java 8
- Lambda-Ausdrücke
- Funktionale Interfaces
- Ergänzungen zu Lambda-Ausdrücken
 - Methodenreferenz
 - Gültigkeitsbereich von Variablen
- Ströme
 - Idee
 - Initiale Stromoperationen
 - Intermediäre Operationen
 - Terminale Operationen
 - Parallele Ströme

Wichtige Meilensteine in der Java-Historie



Kapitel 13: Funktionales Programmieren und Streams mit Java 8

- Java-Historie und Java 8
- Lambda-Ausdrücke
- Funktionale Interfaces
- Ergänzungen zu Lambda-Ausdrücken
 - Methodenreferenz
 - Gültigkeitsbereich von Variablen
- Ströme
 - Idee
 - Initiale Stromoperationen
 - Intermediäre Operationen
 - Terminale Operationen
 - Parallele Ströme

Beispiel: ActionListener mit anonymer innerer Klasse

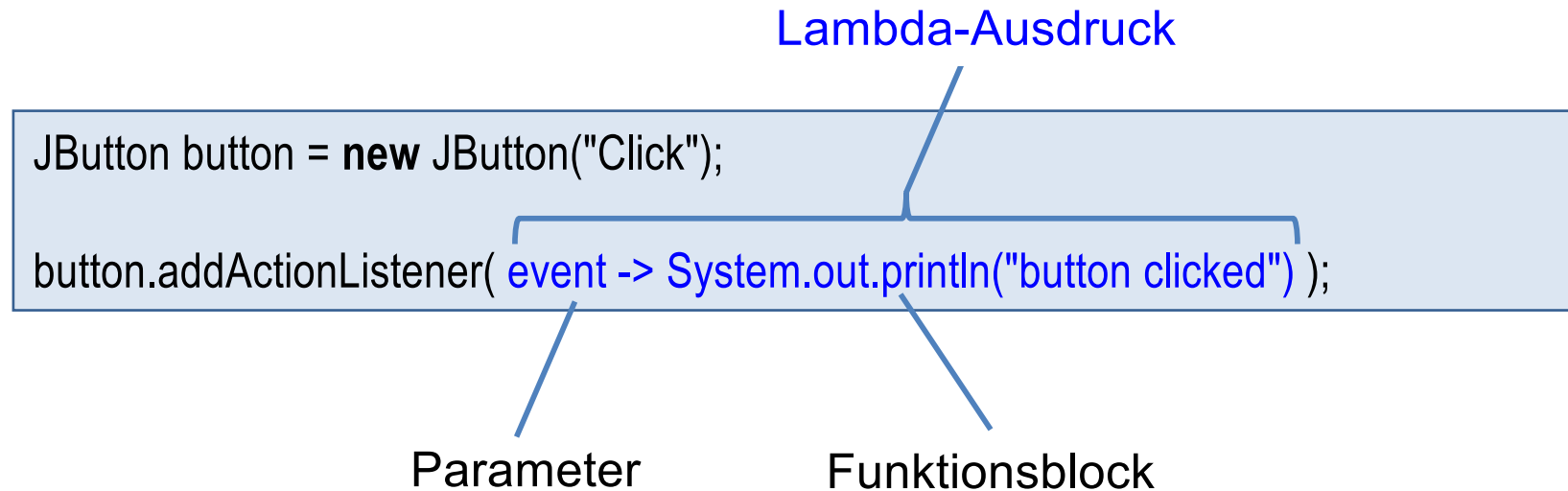
- Das folgende Beispiel zeigt typischen Java-Code, um ein bestimmtes Verhalten (Ausgabe von "button clicked" auf die Console) mit einer Swing-Komponente JButton zu verknüpfen.

```
JButton button = new JButton("Click");
button.addActionListener( new ActionListener() {
    public void actionPerformed(ActionEvent event)
        System.out.println("button clicked");
    }
});
```

- Es wird zuerst ein **neues Objekt** erzeugt, das das **Interface ActionListener** implementiert. Dazu muss die Methode actionPerformed() implementiert werden.
- Das Objekt ist damit **Instanz einer anonymen, inneren Klasse**.
- Das Objekt wird mit der Methode addActionListener() bei der Swing-Komponente button registriert.
- Es muss ein **großer syntaktischer Aufwand** betrieben werden, um ein gewünschtes Verhalten mit einer Swingkomponente zu verknüpfen.

Beispiel: ActionListener mit Lambda-Ausdruck

- Mit einem **Lambda-Ausdruck** geht es prägnanter:



- Lambda-Ausdrücke sind anonyme (d.h. namenlose) Funktionen.
- Beachte: der Parameter event muss nicht typisiert werden. Der Parametertyp wird vom Java-Compiler hergeleitet.

Beispiel: Comparator mit anonymer innerer Klasse

- Im folgenden Beispiel wird eine **Integer-Liste absteigend sortiert** mit Hilfe eines **Comparator-Objekts**.
- Das Comparator-Objekt wird neu erzeugt und implementiert das Interface Comparator und ist damit Instanz einer anonymen, inneren Klasse.

```
List<Integer> intList = Arrays.asList(5, 2, 7, 8, 9, 1, 4, 3, 6, 10);  
intList.sort( new Comparator<Integer>(){  
    public int compare(Integer x, Integer y) {  
        return y.compareTo(x);  
    }  
});
```

- Es muss ein **großer syntaktischer Aufwand** betrieben werden, um das Sortierverfahren mit der gewünschten Vergleichsmethode zu parameterisieren.
- Beachte: seit Java 8 bietet das Interface List<E> auch eine Sortiermethode (stabiles Sortierverfahren) an:

void sort(Comparator<? **super** E> c)

Beispiel: Comparator mit Lambda-Ausdruck

- Mit einem **Lambda-Ausdruck** geht es prägnanter:

```
List<Integer> intList = Arrays.asList(5, 2, 7, 8, 9, 1, 4, 3, 6, 10);  
intList.sort( (x,y) -> y.compareTo(x) );
```



Lambda-Ausdruck

- Beachte: hier hat der Lambda-Ausdruck zwei Parameter x, y. Beide Parameter müssen nicht typisiert werden. Der Parametertyp wird vom Java-Compiler hergeleitet.

Lambda-Ausdrücke (1)

- Lambda-Ausdrücke haben die allgemeine Bauart:

```
(Parameterliste) -> Funktionsblock
```

- Beispiel:

```
(x, y, z) -> x + y + z
```

- Die Parameterliste kann leer sein:

```
( ) -> System.out.println("Hallo");
```

- Hat die Parameterliste genau einen nicht typisierten Parameter, dann kann die Klammer entfallen.

```
(x) -> x+1  
x -> x+1
```

- Die Parameter können typisiert werden (in manchen Situationen ist das auch erforderlich). Die Klammer muss dann geschrieben werden.

```
(String s) -> s + "!"  
(int x, int y) -> x + y
```


Lambda-Ausdrücke (2)

- Der Funktionsblock bei Lambda-Termen folgt den gleichen Regeln wie bei Methoden.
- Wird ein Rückgabewert erwartet, dann muss ein return erfolgen (Kurzschreibweise möglich: siehe unten). Erfolgt kein Rückgabewert, dann kann return entfallen.

```
(int n) -> {  
    int p = 1;  
    for (int i = 1; i <= n; i++)  
        p *= i;  
    return p;  
}
```

```
(int n) -> {  
    for (int i = 1; i <= n; i++)  
        System.out.println();  
}
```

- Besteht der Funktionsblock nur aus einer return-Anweisung oder einem Funktionsaufruf, dann gibt es folgende Kurzschreibweisen:

```
(int n) -> n + 1
```

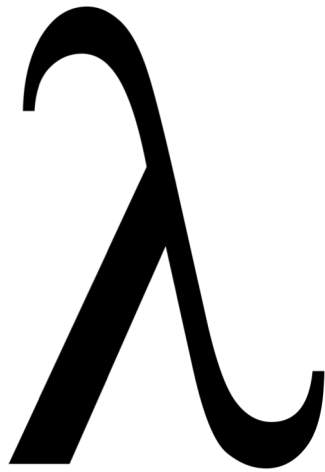
statt

```
(int n) -> {  
    return n + 1;  
}
```

```
() -> System.out.println("Hallo")
```

```
() -> {  
    System.out.println("Hallo");  
}
```

Historisches: λ -Kalkül



- Der Begriff der Lambda-Ausdrücke stammt aus dem λ -Kalkül, der von den beiden US-amerikanischen Mathematikern und Logikern [Alonzo Church](#) und [Stephen Cole Kleene](#) in den 30er-Jahren entwickelt wurde. Church und Kleene gehören zu den Begründern der theoretischen Informatik.
- Der λ -Kalkül ist auch theoretische Grundlage der [funktionalen Programmiersprachen](#) wie z.B. [Lisp](#) (1958) und [Haskell](#) (1990).
- Der λ -Kalkül formalisiert Konzepte wie Funktionsanwendung ([Applikation](#)) und Funktionsbildung ([\$\lambda\$ -Abstraktion](#)):
 - MN Applikation: wende λ -Term M auf N an
 - $\lambda x.M$ λ -Abstraktion: binde die Variable x im λ -Term M
- Die Auswertung von λ -Termen wird mit Reduktionsregeln festgelegt:
 - $\lambda x.x+1 \rightarrow \lambda y.y+1$ [\$\alpha\$ -Konversion](#) (gebundene Umbenennung)
 - $(\lambda x.x+1) 2 \rightarrow 2+1$ [\$\beta\$ -Konversion](#) (Funktionsanwendung; ersetze x durch 2 in x+1)

Kapitel 13: Funktionales Programmieren und Streams mit Java 8

- Java-Historie und Java 8
- Lambda-Ausdrücke
- Funktionale Interfaces
- Ergänzungen zu Lambda-Ausdrücken
 - Methodenreferenz
 - Gültigkeitsbereich von Variablen
- Ströme
 - Idee
 - Initiale Stromoperationen
 - Intermediäre Operationen
 - Terminale Operationen
 - Parallele Ströme

Interface: default-Methoden und abstrakte Methoden

- In einem Interface dürfen Methoden vordefiniert werden: **default-Methoden**. Sie werden dazu mit dem Schlüsselwort **default** gekennzeichnet.
- Default-Methoden dürfen in implementierenden Klassen oder abgeleiteten Interfaces überschrieben werden.
- Methoden, die dagegen nur deklariert werden, werden auch **abstrakt** genannt.
- **Wichtiger Unterschied zu abstrakten Klassen:**
in einem Interface sind keine Instanzvariablen möglich.

```
interface Set<E> extends Iterable<E> {  
    boolean contains(E e);  
    default boolean containsAll(Set<E> s) {  
        for (E e : s)  
            if ( ! contains(e) )  
                return false;  
        return true;  
    }  
}
```

abstrakte Methode

default-Methode

Funktionales Interface

- Ein **funktionales Interface** ist ein Interface mit genau einer abstrakten Methode.
- Default- und statische Methoden dürfen dagegen in beliebiger Anzahl vorkommen.
- Ein funktionales Interface deklariert mit seiner abstrakten Methode den Typ einer Funktion.
- Annotation verwenden: **@FunctionalInterface**

```
@FunctionalInterface
interface BiFunction {
    double apply(double x, double y);
}
```

BiFunction beschreibt den Typ von Funktionen, die zwei double-Werte auf einen double-Wert abbilden:

Mathematisch:

$\text{double} \times \text{double} \rightarrow \text{double}$

```
@FunctionalInterface
interface Predicate {
    boolean test(int x);
}
```

Predicate beschreibt den Typ von Funktionen, die einen int-Wert auf einen Booleschen Wert abbilden.

Mathematisch:

$\text{int} \rightarrow \text{boolean}$

Solche Funktionen werden auch Prädikate genannt (siehe Prädikatenlogik).

Lambda-Ausdrücke und funktionale Interfaces (1)

- Ein Lambda-Ausdruck wird immer im Kontext eines funktionalen Interfaces definiert.
- Dabei legt das funktionale Interface den Typ des Lambda-Ausdrucks fest.
- Durch die abstrakte Methode des funktionalen Interface wird festgelegt, wie der Lambda-Ausdruck benutzt (aufgerufen) werden kann.

```
interface BiFunction {  
    double apply(double x, double y);  
}
```

```
BiFunction add = (x,y) -> x+y;  
BiFunction mult = (x,y) -> x*y;  
BiFunction max = (x,y) -> {  
    if (x >= y)  
        return x;  
    else  
        return y;  
};
```

```
System.out.println(add.apply(4, 5));  
System.out.println(mult.apply(4, 5));  
System.out.println(max.apply(4, 5));
```

Definition von Lambda-Ausdrücken

Benutzung (Aufruf) der Lambda-Ausdrücke

Lambda-Ausdrücke und funktionale Interfaces (2)

- Lambda-Ausdrücke können auch als Parameter übergeben werden.

```
interface Predicate {  
    boolean test(int x);  
}
```

```
boolean forAll(int[ ] a, Predicate p) {  
    for (int x : a)  
        if (! p.test(x))  
            return false;  
    return true;  
}
```

forAll(a, p) prüft, ob
alle Elemente aus dem Feld a
das Prädikat p erfüllen.

```
Predicate isPositive = x -> x >= 0;  
  
int [ ] a = {3, 5, -6, 5};  
System.out.println(forAll(a, isPositive));
```

isPositive prüft, ob ein
Element x positiv ist.

prüfe, ob alle Elemente
aus Feld a positiv sind.

Typinferenz

- Die Parameter der Lambda-Ausdrücke müssen in der Regel nicht typisiert werden.
- Der Parametertyp wird vom Java-Compiler aus dem funktionalen Interface hergeleitet (Typinferenz)

```
interface BiFunction {  
    double apply(double x, double y);  
}
```

```
BiFunction add_V1 = (double x, double y) -> x+y;  
BiFunction add_V2 = (x, y) -> x+y;
```

Lambda-Ausdrücke sind
gleichwertig

Funktionale Interfaces in java.util.function (1)

- Das in Java 8 eingeführte Paket `java.util.function` enthält sehr viele funktionale Interfaces. Z.B.:

Funktionales Interface	Abstrakte Methode	Beschreibung
<code>Predicate<T></code>	<code>boolean test(T t)</code>	1-stelliges Prädikat vom Typ $T \rightarrow \text{boolean}$
<code>BiPredicate<T, U></code>	<code>boolean test(T t, U u)</code>	2-stelliges Prädikat vom Typ $T \times U \rightarrow \text{boolean}$
<code>Function<T, R></code>	<code>R apply(T t)</code>	1-stellige Funktion vom Typ $T \rightarrow R$
<code>BiFunction<T, U, R></code>	<code>R apply(T t, U u)</code>	2-stellige Funktion vom Typ $T \times U \rightarrow R$
<code>UnaryOperator<T></code>	<code>T apply(T t)</code>	1-stelliger Operator vom Typ $T \rightarrow T$ (ist abgeleitet von <code>Function<T, T></code>)
<code>BinaryOperator<T></code>	<code>T apply(T t, T u)</code>	2-stelliger Operator vom Typ $T \times T \rightarrow T$ (ist abgeleitet von <code>BiFunction<T, T, T></code>)
<code>Consumer<T></code>	<code>void accept(T t)</code>	Funktion, die ein T-Parameter entgegen nimmt.
<code>Supplier<T></code>	<code>T get()</code>	Funktion, die ein T-Element zurückliefert.

- Für Basisdatentypen `int`, `long` und `double` gibt es außerdem noch spezielle Interfaces, die analog aufgebaut sind.

Funktionale Interfaces in java.util.function (2)

- Beispiele für Lambda-Ausdrücke:

```
Predicate<Integer> isEven = (x) -> x%2 == 0;
```

```
IntPredicate isEven = (x) -> x%2 == 0;
```

isEven mit
int-spezifischem Interface

```
Predicate<String> endsWithDollar = s -> s.endsWith("$");
```

```
BiPredicate<String,Integer> endsWithInt = (s, x) -> s.endsWith(String.valueOf(x));  
System.out.println(endsWithInt.test("Hallo123",123));
```

```
BinaryOperator<Double> sumSquares = (x, y) -> x*x + y*y;
```

```
DoubleBinaryOperator sumSquares = (x, y) -> x*x + y*y;
```

sumSquares mit
double-spezifischem Interface

```
BinaryOperator<String> pair = (s1, s2) -> "(" + s1 + ", " + s2 + ")";
```

```
Consumer<Person> persPrinter = p -> System.out.println(p.getName());
```

```
Supplier<Point> newZeroPoint = () -> { return new Point(0,0); };
```

Funktionales Interface Predicate und default-Methoden

- Viele funktionale Interfaces in der Java API enthalten nicht nur die für ein funktionales Interface notwendige abstrakte Methode sondern auch noch verschiedene default-Methoden.
- Das Interface **Predicate** enthält beispielsweise die default-Methoden **and**, **or** und **negate**, mit denen Prädikate aussagenlogisch verknüpft werden können.

```
@FunctionalInterface
interface Predicate<T> {
    boolean test(T x);
    default Predicate<T> and(Predicate<? super T> other);
    default Predicate<T> or(Predicate<? super T> other);
    default Predicate<T> negate( );
}
```

```
Predicate<Integer> isEven = x -> x%2==0;
Predicate<Integer> isPositive = x -> x > 0;
Predicate<Integer> isEvenAndPositive = isEven.and(isPositive);
```

Funktionen höherer Ordnung

- and, or und negate werden auch Funktionen höherer Ordnung genannt: Parameter und/oder return-Werte sind Funktionen.

```
@FunctionalInterface
interface Predicate<T> {
    boolean test(T x);
    default Predicate<T> and(Predicate<? super T> other);
    ...
}
```

```
Predicate<Integer> isEvenAndPositive = isEven.and(isPositive);
```

- and nimmt 2 Prädikate entgegen (this und other) und liefert ein Prädikat als return-Wert zurück.
- mathematisch geschrieben:

$$\text{and: } \underbrace{(T \rightarrow \text{boolean})}_{\text{this}} \times \underbrace{(T \rightarrow \text{boolean})}_{\text{other}} \rightarrow \underbrace{(T \rightarrow \text{boolean})}_{\text{return}}$$

- Funktionen höherer Ordnung sind typisch für funktionale Programmiersprachen.

Funktionales Interface Function

- Das Interface `Function` enthält die default-Methoden `andThen` und `compose` zur Komposition von Funktionen:

```
interface Function<T, R> {  
    R apply(T x);  
    default <V> Function<T,V> andThen(Function<? super R, ? extends V> after)  
    default <V> Function<V,R> compose(Function<? super V, ? extends T> before)  
}
```

```
Function<Double, Double> square = x -> x*x;  
Function<Double, Double> incr3 = x -> 3 + x;  
Function<Double, Double> f = square.andThen(incr3);  
Function<Double, Double> g = incr3.compose(square);  
Function<Double, Double> h = square.compose(incr3);  
System.out.println(f.apply(2.0));    // 7.0  
System.out.println(g.apply(2.0));    // 7.0  
System.out.println(h.apply(2.0));    // 25.0
```

- Typ der Methode `andThen` mathematisch geschrieben:

$$\text{andThen: } \underbrace{(T \rightarrow R)}_{\text{this}} \times \underbrace{(R \rightarrow V)}_{\text{other}} \rightarrow \underbrace{(T \rightarrow V)}_{\text{return}}$$

other darf auch vom Typ $R^+ \rightarrow V^-$ sein, wobei $R <: R^+$ und $V^- <: V$ ist.

Funktionales Interface Comparator

- Das funktionale Interface `Comparator` definiert nicht nur die abstrakte Methode `compare(x,y)`, sondern bietet auch zahlreiche default-Methoden und statische Methoden an.
- Häufig ist es notwendig Instanzen von selbst gebauten Klassen (mit `compare`) vergleichen zu können. Beispielsweise besteht ein Personen-Objekt aus einem Familien-Namen. Der Vergleich zweier Personen soll auf den lexikographischen Vergleich der Familiennamen zurückgeführt werden.
- Die statische Methode `comparing` nimmt eine Funktion `keyExtractor` entgegen, die aus einem Objekt einen Comparable-Schlüssel extrahiert, und liefert einen Comparator zurück.

```
@FunctionalInterface
interface Comparator<T> {
    int compare(T x, T y);
    static <T, U extends Comparable<? super U>>
        Comparator<T> comparing(Function<? super T, ? extends U> keyExtractor);
    ...
}
```

```
List<Person> persList = new LinkedList<>();
persList.add(new Person("Maier")); ...
Comparator<Person> comp = comparing(p->p.getName());
persList.sort(comp);
```

Nicht jedes Interface ist funktional!

- Die Java-API enthält Interfaces, die genau eine abstrakte Methode enthalten, aber nicht als funktionale Interfaces intendiert sind.
- Es **fehlt** dann die Annotation `@FunctionalInterface`.
- Beispiele: `Iterable`, `Comparable`
- Lambda-Ausdrücke haben im Gegensatz zu herkömmlichen Objekten keine Instanzvariablen. Daher wäre ein Lambda-Ausdruck, der `Comparable` oder `Iterable` wäre, sinnlos.

Interface Iterable

- Das Interface Iterable wurde mit Java 8 um die **default-Methode** `forEach` erweitert.
- `forEach` nimmt eine Consumer-Funktion (einstellige Funktion ohne Rückgabewert) entgegen und wendet diese auf jedes Element des Iterable-Objekts an. Damit bietet `forEach` eine interessante Alternative zu einer `foreach`-Schleife an.

```
default void forEach(Consumer<? super T> action)
```

```
List<String> nameList = Arrays.asList("Maria", "Peter", "Petra", "Robert");  
nameList.forEach(name -> System.out.println(name));
```

- Die `forEach`-Methode wird auch **interne Iteration** genannt im Gegensatz zur **externen Iteration** mit einer `foreach`-Schleife.
- Java-Entwickler haben die Möglichkeit, die `forEach`-Methode in einem Iterable-Container (z.B. `ArrayList`) geeignet zu überschreiben, um Effizienzgewinne zu erzielen (z.B. durch Parallelisierung).
- Man beachte auch die Eleganz der default-Technik bei `Iterable.forEach`. Klassen, die `Iterable` implementieren und vor Java 8 entwickelt wurden, brauchen nicht geändert zu werden!

Kapitel 13: Funktionales Programmieren und Streams mit Java 8

- Java-Historie und Java 8
- Lambda-Ausdrücke
- Funktionale Interfaces
- Ergänzungen zu Lambda-Ausdrücken
 - Methodenreferenz
 - Gültigkeitsbereich von Variablen
- Ströme
 - Idee
 - Initiale Stromoperationen
 - Intermediäre Operationen
 - Terminale Operationen
 - Parallele Ströme

Methoden-Referenz für Lambda-Ausdruck

- Lambda-Ausdrücke, die genau ein Methodenaufruf darstellen, lassen sich durch den Namen der Methode ersetzen (Methodenreferenz):

`(Person p) -> p.getName()`

Lambda-Ausdruck, der nur die Methode `p.getName()` aufruft.



`Person::getName`

Referenz auf Methode `getName`

4 Arten von Methodenreferenzen

	Methodenreferenz	Lambda-Ausruck
Statische Methode	<code>C::m</code>	<code>(x,y,...) -> C.m(x,y,...)</code>
Instanzmethode	<code>C::m</code>	<code>(self,x,y,...) -> self.m(x,y,...)</code>
Instanzmethode mit Bindung auf Objekt obj	<code>obj::m</code>	<code>(x,y,...) -> obj.m(x,y,...)</code>
Konstruktor	<code>C::new</code>	<code>(x,y,...) -> new C(x,y,...)</code>

Beispiel für Referenz auf statische Methode

```
class Utilities {  
    public static boolean isNotPrime(int n) { ... }  
}
```

```
List<Integer> intList = new LinkedList<>();  
for (int i = 1; i < 50; i++)  
    intList.add(i);  
  
intList.removeIf(Utilities::isNotPrime);
```

`removeIf` ist im `Interface Collection` als Default-Methode definiert:

```
default boolean removeIf(Predicate<? super E> filter)
```

`col.removeIf(p)` entfernt alle Elemente aus dem Container `col`, auf die das Prädikat `p` zutrifft.

Beispiele für Referenz auf Instanz-Methoden

```
List<String> nameList  
    = Arrays.asList("Maria", "Peter", "Petra", "Robert");  
nameList.replaceAll(String::toUpperCase);  
System.out.println(nameList);
```

Die Methode [toUpperCase](#) aus der Klasse [String](#)
`String toUpperCase()`
wandelt Klein- in Großbuchstaben und liefert den
gewandelten String zurück.

[MARIA, PETER, PETRA, ROBERT]

```
List<Object> objList  
    = Arrays.asList(new Complex(2,1), 12.3, "Robert");  
objList.replaceAll(Object::toString);  
System.out.println(objList);
```

[2.0 + 1.0*i, 12.3, Robert]

[replaceAll](#) ist im [Interface List](#) als Default-Methode definiert:

```
default void replaceAll(UnaryOperator<E> operator)  
list.replaceAll(f) ersetzt jedes Element x in list durch f(x).
```

Beispiel für Referenz auf Instanzmethode mit Objekt-Bindung

```
List<String> nameList  
    = Arrays.asList("Maria", "Peter", "Petra", "Robert");  
nameList.forEach(System.out::println);
```

Referenz auf Instanz-Methode mit Bindung an Objekt System.out.
forEach erwartet als Parameter eine Consumer-Funktion
(einstellige Funktion ohne Rückgabewert)

Beispiele für Referenz auf Konstruktor

```
Supplier<Complex> s = () -> new Complex();  
Complex z = s.get();
```

```
Supplier<Complex> s = Complex::new ;  
Complex z = s.get();
```

Referenz auf parameterlosen
Konstruktor

```
Function<Double,Complex> f = x -> new Complex(x);  
Complex z = f.get();      // z = 0 + 0*i
```

```
Function<Double,Complex> f = x -> Complex::new;  
Complex z = f.apply(5);   // z = 5 + 0*i
```

Referenz auf einstelligen
Konstruktor

Kapitel 13: Funktionales Programmieren und Streams mit Java 8

- Java-Historie und Java 8
- Lambda-Ausdrücke
- Funktionale Interfaces
- Ergänzungen zu Lambda-Ausdrücken
 - Methodenreferenz
 - Gültigkeitsbereich von Variablen
- Ströme
 - Idee
 - Initiale Stromoperationen
 - Intermediäre Operationen
 - Terminale Operationen
 - Parallele Ströme

Zugriff auf Umgebungsvariablen in Lambda-Ausdrücken (1)

- In den meisten Fällen wird in Lambda-Ausdrücken nicht auf Variablen aus der Umgebung zugegriffen. Man erhält dann **zustandslose Funktionen** (**stateless function**) **ohne Seiteneffekte**. Jeder Aufruf liefert den denselben Wert zurück. Variablen aus der Umgebung werden nicht verändert.

```
Function<Integer, Integer> f = x -> x*x;  
System.out.println(f.apply(5)); // 25
```

- Jedoch kann in einem Lambda-Ausdruck auch auf Variablen aus der Umgebung zugegriffen werden.

```
Function<Integer, Integer> f = x -> u + x*x;
```

u ist eine Variable aus der Umgebung

- **Zugriff auf eine lokale Variable (oder auch Parameter) in einem umfassenden Block.** Diese Variable muss **effektiv final** sein (Der Variablen darf nur einmal ein Wert zugewiesen werden). Der Lambda-Ausdruck sollte damit (muss aber nicht) zustandslos und seiteneffektfrei sein.
- **Zugriff auf eine Instanzvariablen der umfassenden Klasse.** Hier gibt es keine Einschränkungen an die Variable. Der Lambda-Ausdruck kann damit **zustandsbehaftet** sein (**stateful function**) und Seiteneffekte haben.

Zugriff auf Umgebungsvariablen in Lambda-Ausdrücken (2)

```
class Demo {  
    private int instanceVar = 5;  
  
    public void test() {  
        int local = 5;  
        // local++; nicht erlaubt!  
        Function<Integer, Integer> f = x -> local + x*x;  
        System.out.println(f.apply(5)); // 30  
        System.out.println(f.apply(5)); // 30  
  
        Function<Integer, Integer> g = (x) -> {  
            instanceVar++;  
            return instanceVar + x*x;  
        };  
        System.out.println(g.apply(5)); // 31  
        System.out.println(g.apply(5)); // 32  
    }  
  
    public static void main(String[ ] args) {  
        new Demo().test();  
    }  
}
```

Zugriff auf lokale Variable local
aus der Umgebung.

local muss effektiv final sein und darf
nach der Initialisierung nicht mehr
verändert werden.

Funktion f ist damit zustandslos.

Zugriff auf Instanzvariable
instanceVar der Klasse Demo.

instanceVar wird verändert.

Funktion g ist damit zustandsbehaftet
und hat einen Seiteneffekt.

Zugriff auf Umgebungsvariablen in Lambda-Ausdrücken (3)

```
class Demo {  
    public static void main(String[] args) {  
        MutableInt mutableLocal = new MutableInt();  
        Function<Integer, Integer> f = x -> {  
            mutableLocal.incr();  
            return mutableLocal.get() + x*x;  
        };  
        System.out.println(f.apply(5)); // 26  
        System.out.println(f.apply(5)); // 27  
    }  
}
```

```
class MutableInt {  
    private int i = 0;  
    public int get() { return i; }  
    public void incr() { i++; }  
}
```

Zugriff auf lokale Variable mutableLocal.

mutableLocal ist zwar effektiv final, ist aber eine Referenz auf ein mutables Objekt.

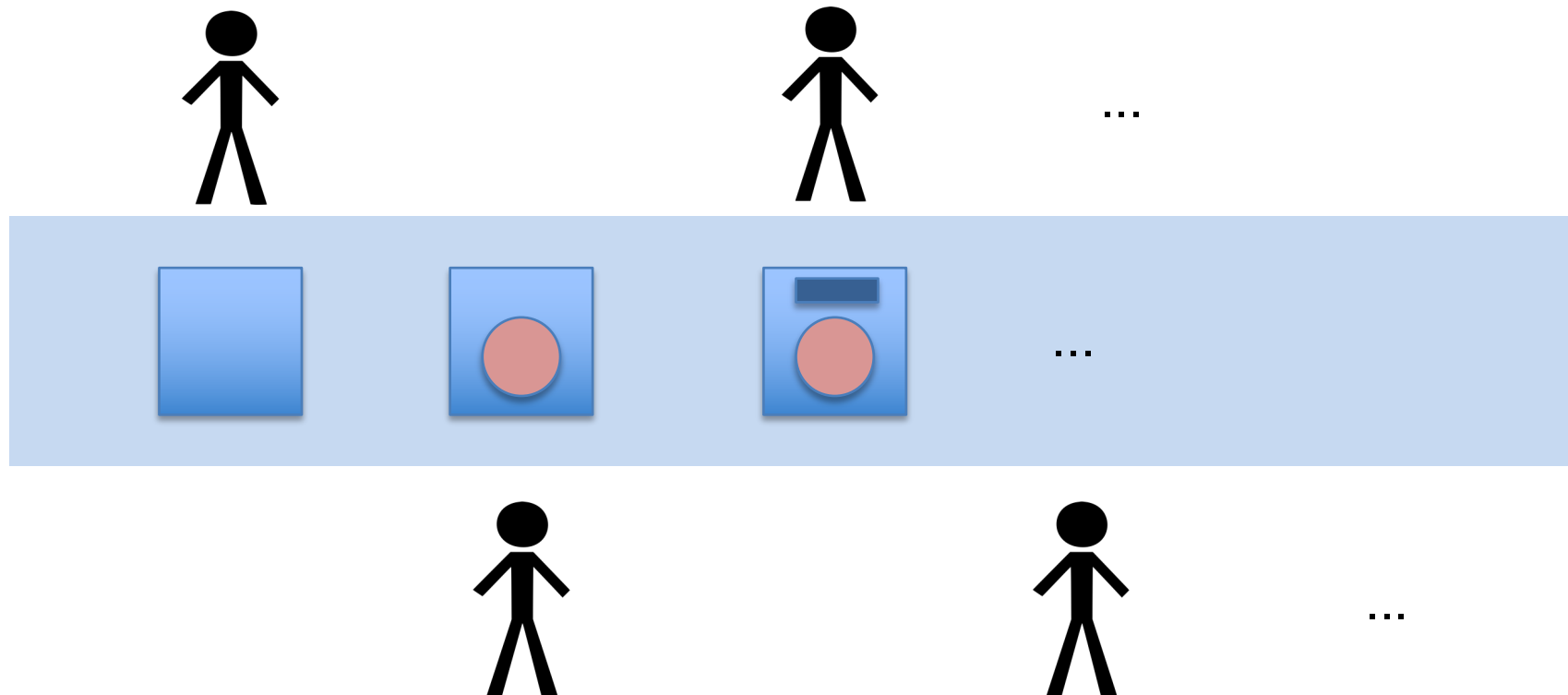
Die Funktion f ist zustandsbehaftet und hat einen Seiteneffekt.

MutableInt ist eine mutable Klasse, die ein int-Wert kapselt.

Kapitel 13: Funktionales Programmieren und Streams mit Java 8

- Java-Historie und Java 8
- Lambda-Ausdrücke
- Funktionale Interfaces
- Ergänzungen zu Lambda-Ausdrücken
 - Methodenreferenz
 - Gültigkeitsbereich von Variablen
- **Ströme**
 - Idee
 - Initiale Stromoperationen
 - Intermediäre Operationen
 - Terminale Operationen
 - Parallele Ströme

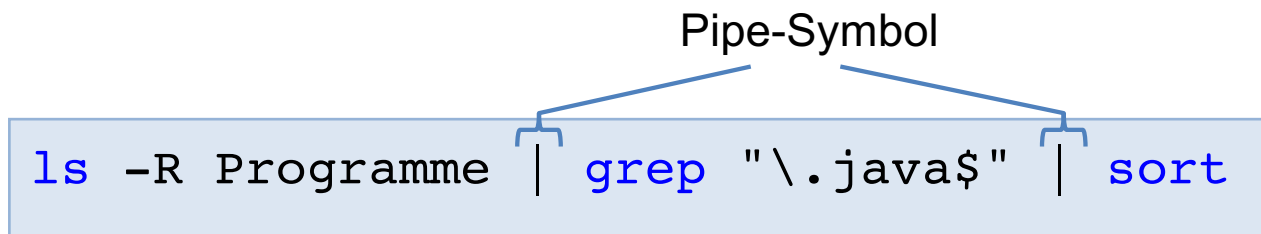
Fließband (pipeline)



- Produktion einer Waschmaschine an einem Fließband
- Personen arbeiten parallel mit jeweils anderer Tätigkeit

Unix Pipes

- Der Pipe-Mechanismus wurde Anfang der 70er-Jahre in Unix eingeführt.
- Er gestattet den Austausch von Daten zwischen zwei Programmen.
- Damit läßt sich eine **Kette von Programmen** zusammenbauen: jedes Programm nimmt Daten entgegen, verarbeitet sie und reicht seine Ausgaben an das nächste Programm weiter (**Pipeline-Verarbeitung**).
- Die Programme laufen dabei (soweit möglich) **parallel!**



Mit **ls** wird eine Liste aller Dateinamen im Verzeichnis **Programme** und dessen Unterverzeichnisse erzeugt.



Mit **grep** werden die Dateinamen, die mit **".java"** enden, herausgesucht.

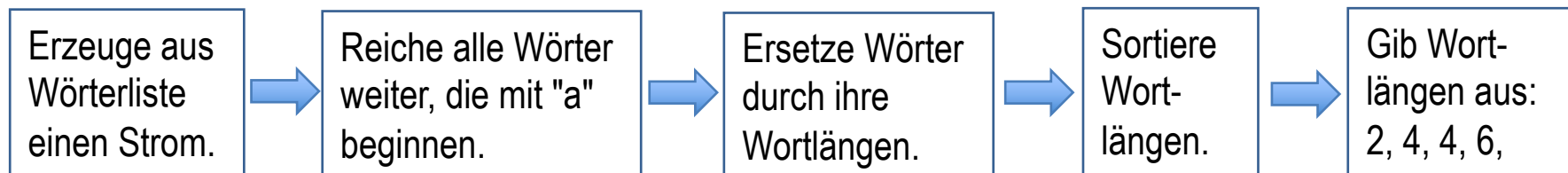


Mit **sort** wird die Ausgabe von **grep** entgegengenommen, sortiert und auf die Konsole ausgegeben.

Ströme (streams) in Java 8

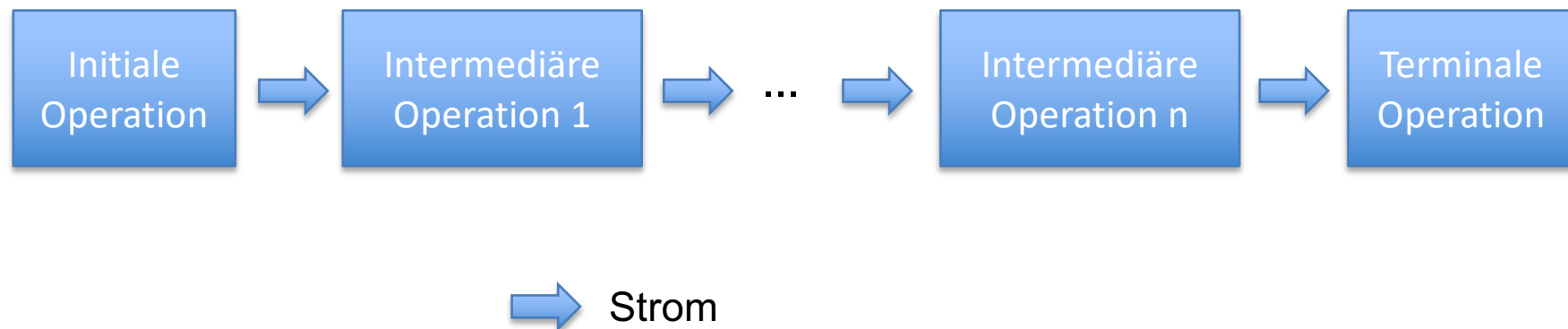
- **Ströme** sind eine (evtl. unendlich lange) Folge von Datenobjekte.
- Die Datenobjekte eines Stroms werden von Methoden verarbeitet und können dann zur nächsten Methode weitergereicht werden (**Pipeline-Verarbeitung**).
- Das Stromkonzept von Java hat damit große Ähnlichkeit zu den Unix-Pipes.

```
List<String> wordList = Arrays.asList("achten", "auch", "zum", "an", "bei", "aber", "vor");  
  
wordList.stream()  
    .filter( s -> s.startsWith("a") )  
    .mapToInt( s -> s.length() )  
    .sorted()  
    .forEach( n -> System.out.print(n + ", ") );  
  
System.out.println("");
```

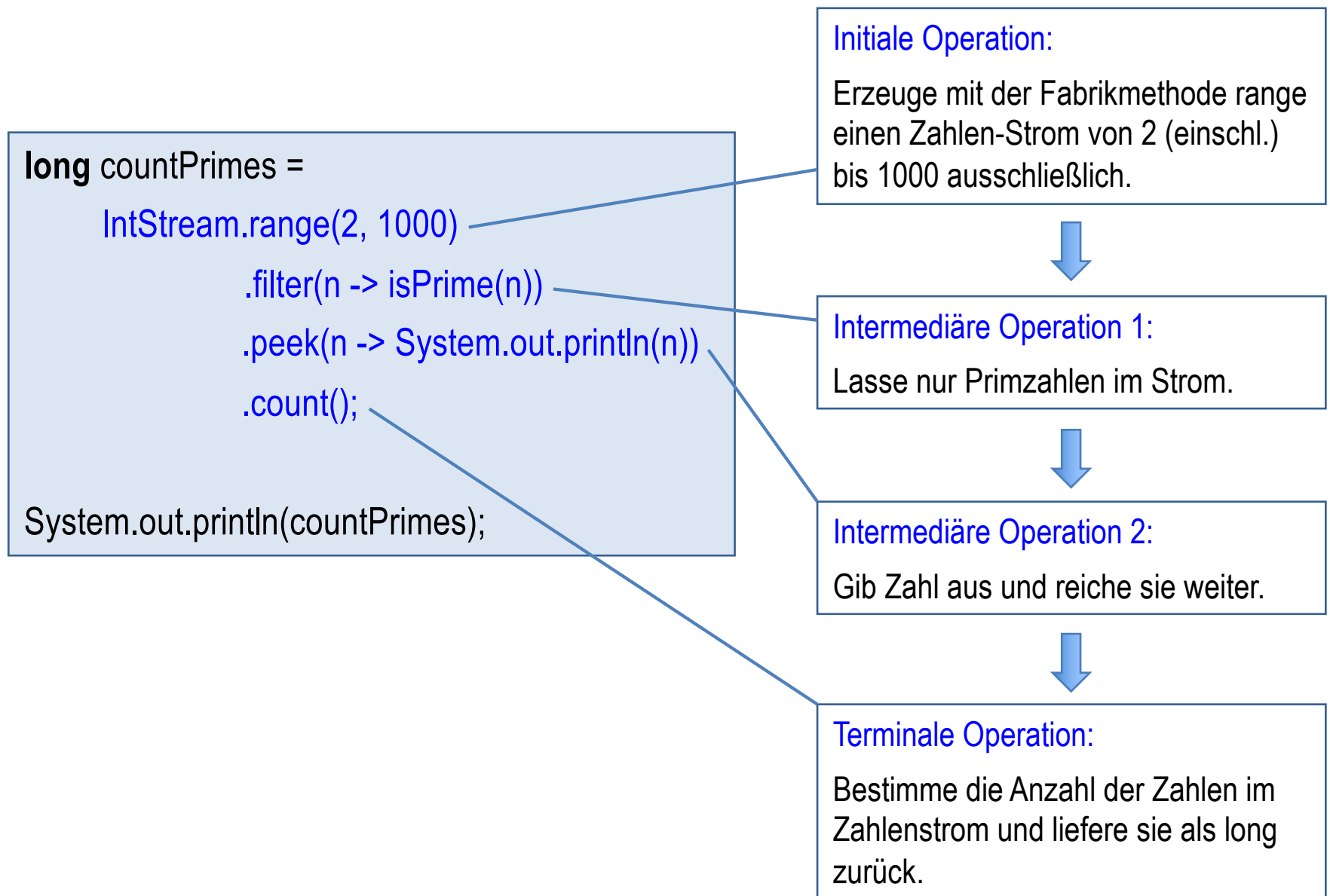


Aufbau eines Stroms

- Mit einer **initialen Operation** wird ein Strom erzeugt.
Initiale Strom-Operationen werden von verschiedenen Klassen der Java-API angeboten (wie z.B. Collection-Klassen, Arrays, diverse Fabrikmethoden aus stream-Klassen, ...)
- Mit (einer oder mehreren) **intermediären Operationen** (**intermediate operations**) werden Ströme transformiert.
Rückgabewert einer intermediären Operation ist wieder ein Strom.
- Mit einer **terminalen Operation** (**terminal operation**) wird der Strom abgeschlossen.
Terminale Operationen liefern ein Resultat (aber keinen Strom) zurück oder haben keinen Rückgabewert und evtl. einen Seiteneffekt.
- Intermediäre und terminale Operationen sind im Paket `java.util.stream` festgelegt.



Beispiel



Verzögerte Generierung der Ströme

- Ströme werden nie komplett im voraus generiert.
Beachte: **Ströme** können **prinzipiell unendlich lang** werden.
- Es werden nur solange Daten für den Strom generiert, wie die terminale Operation noch Daten benötigt. Der **Strom** wird **verzögert generiert** (**lazy evaluation**).
- Beispiel:

```
new Random().ints()  
    .map( n -> Math.abs(n)%1000 )  
    .peek( System.out::println )  
    .anyMatch(n -> 10 <= n && n < 20);
```

- Die **initiale Operation** **ints()** der Klasse **Random** erzeugt einen **prinzipiell unendlichen Strom** von Zufallszahlen.
- Die **intermediäre map-Operation** transformiert die Zufallszahlen in das Intervall [0,1000).
- Die **intermediäre peek-Operation** gibt jede Zahl aus und reicht sie weiter.
- Die **terminale Operation** **anyMatch** bricht mit Rückgabe von true ab, sobald eine Zahl im Intervall [10,20) liegt.

Kapitel 13: Funktionales Programmieren und Streams mit Java 8

- Java-Historie und Java 8
- Lambda-Ausdrücke
- Funktionale Interfaces
- Ergänzungen zu Lambda-Ausdrücken
 - Methodenreferenz
 - Gültigkeitsbereich von Variablen
- **Ströme**
 - Idee
 - Initiale Stromoperationen
 - Intermediäre Operationen
 - Terminale Operationen
 - Parallele Ströme

Initiale Strom-Operationen für Datenbehälter

- Ströme können aus zahlreichen **Datenbehälter der Java API** erzeugt werden.
Beispiele:

Collection.stream() Collection.parallelStream()	Instanz-Methode zum Erzeugen eines sequentiellen bzw. parallelen Stroms.
Arrays.stream(T[] a)	statische Methode zum Erzeugen eines Strom aus dem Feld a. Weitere Methoden für Basisdatentypen.
BufferedReader.lines()	Instanz-Methode liefert einen Strom bestehend aus Zeilen.

- Beispiele:

```
List<String> wordList = Arrays.asList("achten", "auch", "zum", "an", "bei", "aber", "vor");  
Stream<String> s1 = wordList.stream();    // Strom mit den Strings aus wordList
```

```
int[] a = new int[] {1,2,3,4,5};  
IntStream s0 = Arrays.stream(a);    // Strom mit den int-Zahlen aus a
```

```
BufferedReader in = new BufferedReader(new FileReader("test.txt"));  
Stream<String> s2 = in.lines();    // Strom mit Zeilen der Datei test.txt
```

Initiale Strom-Operationen aus Paket Stream

- Es gibt zahlreiche **statische Fabrik-Methoden** aus den Stream-Klassen (Stream<T>, IntStream, DoubleStream, LongStream) im **Paket java.util.stream**.
Beispiele.:

empty()	Leerer Strom.
of(...)	Strom mit vorgebenen Elementen
generate(s)	Generiere Strom durch wiederholtes Aufrufen von s: s(), s(), s(), ...
iterate(a,f)	Generiere Strom durch Iteration: a, f(a), f(f(a)), ...
range(a,b)	Generiere Integer-Strom von a einschl. bis b ausschl.

- Beispiele:

```
IntStream s3 = IntStream.of(1, 2, 3, 4);    // Strom mit den Zahlen 1, 2, 3, 4
IntStream s4 = IntStream.iterate(1, x -> 2*x);    // Unendlicher Strom mit allen 2-er Potenzen
// Unendlicher Strom mit sin(x), wobei x eine Zufallszahl aus [0,1) ist:
DoubleStream s5 = DoubleStream.generate( () -> Math.sin( Math.random()) );
IntStream s6 = IntStream.range(1,10);    // Strom mit int-Zahlen von 1 bis 9 (einschl.).
```

Initiale Strom-Operationen aus Klasse Random

doubles()	Strom mit unendlich vielen zufälligen double-Zahlen aus [0,1)
ints()	Strom mit unendlich vielen zufälligen int-Zahlen

- Beispiele:

```
IntStream s1 = new Random().ints();  
DoubleStream s2 = new Random().doubles();
```

Kapitel 13: Funktionales Programmieren und Streams mit Java 8

- Java-Historie und Java 8
- Lambda-Ausdrücke
- Funktionale Interfaces
- Ergänzungen zu Lambda-Ausdrücken
 - Methodenreferenz
 - Gültigkeitsbereich von Variablen
- **Ströme**
 - Idee
 - Initiale Stromoperationen
 - Intermediäre Operationen
 - Terminale Operationen
 - Parallele Ströme

Intermediäre Strom-Operationen

- Intermediäre Operationen transformieren Ströme.
- Rückgabewert ist wieder ein Strom.
- Damit ist die typische Verkettung von mehreren Operationen möglich.

```
strom.op1(...)  
    .op2(...)  
  
    ...  
  
    .opN();
```


Intermediäre Strom-Operationen

- Beispiele aus Paket `java.util.stream`:

<code>filter(pred)</code>	lasse nur Elemente x im Strom, für die das Prädikat <code>pred(x)</code> zutrifft.
<code>map(f)</code>	ersetze jedes Element x im Strom durch <code>f(x)</code> .
<code>flatMap(f)</code>	ersetze jedes Element x im Strom durch einen von <code>f(x)</code> erzeugten Strom.
<code>peek(action)</code>	führe für jede Methode die rückgabefunktion <code>action</code> durch. Wird hauptsächlich zu Debugging-Zwecke eingesetzt.
<code>sorted()</code>	sortiere die Elemente im Strom (stabiles Sortierverfahren). Es gibt auch eine überladene Methode mit einem Comparator-Parameter.
<code>distinct()</code>	entferne Duplikate aus dem Strom.
<code>skip(n)</code>	entferne die ersten n Elemente aus dem Strom.
<code>limit(n)</code>	begrenze den Strom auf maximal n Elemente.

Beispiel mit map und flatMap

Datei test.txt:

Dies ist eine
kleine
Test Datei

```
BufferedReader in = new BufferedReader(  
    new FileReader("test.txt"));  
  
in.lines()  
    .peek(System.out::println)  
    .flatMap(line -> Arrays.stream(line.split(" +")))  
    .map(s -> s.toUpperCase())  
    .forEach(System.out::println);
```

Ausgabe
(ohne peek-Aufruf):

DIES
IST
EINE
KLEINE
TEST
DATEI

Initiale Operation in.lines:

Erzeuge einen Strom mit den Zeilen der Datei test.txt. Jede Zeile ist vom Typ String.



Intermediäre Operation peek: Gib Zeile aus.



Intermediäre Operation flatMap:

Arrays.stream zerlegt jede Zeile in einen Strom von Strings. Die Ströme werden mit flatMap aneinandergehängt (flach gemacht).



Intermediäre Operation map:

Ersetze jeden String durch einen String mit Großbuchstaben.



Terminale Operation forEach:

Gib jeden String aus.

Beispiel: Stabiles Sortieren nach zwei Schlüsseln

```
List<Person> persList = new LinkedList<>();  
persList.add(new Person("Klaus", 1961));  
persList.add(new Person("Peter", 1959));  
persList.add(new Person("Maria", 1959));  
persList.add(new Person("Petra", 1961));  
persList.add(new Person("Albert", 1959));  
persList.add(new Person("Anton", 1961));  
persList.add(new Person("Iris", 1959));  
  
persList.stream()  
    .sorted(Comparator.comparing(Person::getName))  
    .sorted(Comparator.comparing(Person::getGeb))  
    .forEach(System.out::println);
```

Baue eine Liste von Personen auf, bestehend aus Name und Geburtsjahr.

Bilde aus der Personen-Liste einen Strom.

Sortierte zuerst Personen nach dem Namen.

Sortiere dann nach dem Geburtsjahr.

Sortiervorgang ist stabil:
Personen sind nach dem Geburtsjahr sortiert und innerhalb einer Jahrgangsstufe alphabetisch sortiert.

Kapitel 13: Funktionales Programmieren und Streams mit Java 8

- Java-Historie und Java 8
- Lambda-Ausdrücke
- Funktionale Interfaces
- Ergänzungen zu Lambda-Ausdrücken
 - Methodenreferenz
 - Gültigkeitsbereich von Variablen
- **Ströme**
 - Idee
 - Initiale Stromoperationen
 - Intermediäre Operationen
 - Terminale Operationen
 - Parallele Ströme

Terminale Operationen

- Mit einer **terminalen Operation** wird der Strom abgeschlossen.
- Terminale Operationen liefern ein Resultat zurück (aber keinen Strom) oder haben keinen Rückgabewert und evtl. einen Seiteneffekt.
- Sind im **Paket `java.util.stream`** festgelegt.

```
value = strom.terminalOperation(...);
```

```
strom.terminalOperation(...);
```

Logische Operationen

<code>anyMatch(pred)</code>	liefert true, falls <code>pred(x)</code> für ein Element <code>x</code> des Stroms zutrifft.
<code>allMatch(pred)</code>	liefert true, falls <code>pred(x)</code> für alle Elemente <code>x</code> des Stroms zutrifft.
<code>noneMatch(pred)</code>	liefert true, falls <code>pred(x)</code> für kein Element <code>x</code> des Stroms zutrifft.

■ Beispiele

```
new Random().ints()  
    .map( n -> Math.abs(n) % 1000 )  
    .peek(System.out::println)  
    .anyMatch(n -> 10 <= n && n < 20);
```

Gibt solange zufällige Zahlen `x` aus, bis ein `x ∈ [10, 20)`.
Rückgabewert ist dann true.

```
new Random().ints()  
    .map( n -> Math.abs(n)%1000 )  
    .peek(System.out::println)  
    .allMatch(n -> 10 <= n && n < 1000);
```

Gibt solange zufällige Zahlen `x` aus, bis ein `x ∉ [10, 1000)`.
Rückgabewert ist dann false.

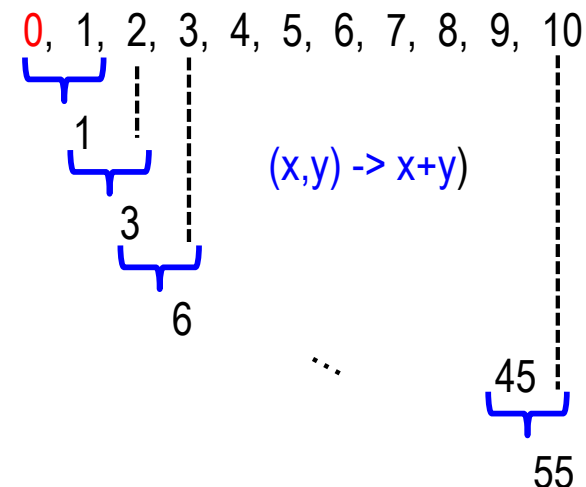
Reduktions-Operationen (1)

- Reduktions-Operationen:

<code>reduce(e, op)</code>	reduziert einen Strom x_0, x_1, x_2, \dots zu dem Wert $(\dots(((e \text{ op } x_0) \text{ op } x_1) \text{ op } x_2) \text{ op } \dots$. Dabei ist <code>op</code> ein 2-stelliger assoziativer Operator und <code>e</code> das neutrale Element bzgl. <code>op</code> . Beispiel: <code>reduce(0, (x,y) -> x+y)</code> berechnet die Summe aller Elemente des Stroms.
<code>count()</code>	Anzahl der Elemente im Strom.
<code>min(cmp)</code> <code>max(cmp)</code>	Liefert größtes bzw. kleinstes Element des Stroms bzgl. der Comparator-Funktion <code>cmp</code> . Beachte: Rückgabewerttyp ist <code>Optional<T></code> . Rückgabewert ist bei leerem Strom undefiniert und sonst ein Wert vom Typ <code>T</code> . Mit <code>isPresent()</code> kann geprüft werden, ob der Rückgabewert definiert ist, und mit <code>get()</code> wird der Wert geholt.

- Beispiel zu `reduce`

```
int sum = IntStream.range(1,11)
                    .reduce(0, (x,y) -> x+y);
System.out.println(sum);    // 55
```



Beispiel: harmonisches Mittel mit reduce-Operation

- Harmonisches Mittel von x_0, x_1, \dots, x_{n-1} :

$$\bar{x}_{\text{harm}} = \frac{n}{\frac{1}{x_0} + \frac{1}{x_1} + \dots + \frac{1}{x_{n-1}}}$$

- Anwendung: auf einer Teilstrecke von jeweils 1 km werden folgende Geschwindigkeiten gefahren: 50, 100, 80, 120, 90 km/h.

Dann ist die Durchschnittsgeschwindigkeit der Gesamtstrecke gerade das harmonische Mittel der Einzelgeschwindigkeiten: $v_{\text{harm}} = 80.71$ km/h

```
double[] v_array = {50, 100, 80, 120, 90}; // Geschw. fuer jeweils 1 km
```

```
double v_harm =  
    Arrays.stream(v_array)  
           .reduce(0, (s,v) -> s + 1/v );  
v_harm = v_array.length/v_harm;  
System.out.println(v_harm); // 80.71
```

Erzeuge aus double-Feld einen Strom von double-Werten.

Terminale reduce-Operation:

Berechne für den Strom von double-Werten v_0, v_1, v_2, \dots den Wert:

$$(((0 + 1/v_0) + 1/v_1) + 1/v_2) \dots$$

Statistik-Operationen für Basisdatentypen

count(), sum() min(), max(), average()	Liefert Anzahl, Summe, Minimum, Maximum bzw. Durchschnittswert der Elemente eines Stroms zurück.
summaryStatistics()	Liefert einen Wert vom Typ IntSummaryStatistics (bzw. DoubleIntSummaryStatistics, ...) zurück, der Anzahl, Summe, Minimum, Maximum und Durchschnittswert umfasst.

Beispiel: Zeilenstatistik für eine Datei

Datei test.txt:

```
a  
bc  
def  
gehi  
jklmn  
opq  
rs
```

```
BufferedReader in = new BufferedReader(  
    new FileReader("test.txt"));  
  
DoubleSummaryStatistics stat =  
    in.lines()  
        .peek(System.out::println)  
        .mapToDouble(s -> s.length())  
        .summaryStatistics();  
System.out.println(stat);
```

Erzeuge einen Strom mit den Zeilen
der Datei test.txt

Ersetze jede Zeile durch ihre Länge
(als double).

Terminale Operation:
Bilde aus den double-Werten eine
Statistik.

Ausgabe (ohne peek):

```
DoubleSummaryStatistics{count=7, sum=20,000000, min=1,000000, average=2,857143, max=5,000000}
```

Terminale Operationen collect und forEach

- Collect-Operationen:

collect(collector)	<p>Kann benutzt werden, um Elemente des Stroms in einem Container aufzusammeln.</p> <p>Beispielsweise werden mit folgender Anweisung alle Elemente eines String-Strom in einer Liste abgespeichert:</p> <pre>List<String> asList = stringStream.collect(Collectors.toList());</pre>
--------------------	---

- forEach-Operationen:

forEach(action)	<p>führe für jedes Element des Stroms die Consumer-Funktion action (einstellige Funktion ohne Rückgabewert) durch.</p>
-----------------	--

Beispiel zu collect-Operation

```
List<Person> persList = new LinkedList<>();  
persList.add(new Person("Klaus", 1961));  
persList.add(new Person("Anton", 1959));  
persList.add(new Person("Maria", 1959));  
persList.add(new Person("Petra", 1961));  
persList.add(new Person("Anton", 1973));  
persList.add(new Person("Peter", 1970));  
persList.add(new Person("Anton", 1961));  
persList.add(new Person("Klaus", 1959));
```

```
List<String> nameList =  
    persList.stream()  
        .map(Person::getName)  
        .sorted(Comparator.naturalOrder())  
        .distinct()  
        .collect(Collectors.toList());
```

```
System.out.println(nameList);
```

[Anton, Klaus, Maria, Peter, Petra]

Ersetze in dem Strom persList.stream() mittels der Operation map jede Person durch ihren Namen.

Sortiere die Namen alphabetisch und entferne Duplikate.

Terminale Operation:

Samme (collect) die Namen in einer Liste.

Kapitel 13: Funktionales Programmieren und Streams mit Java 8

- Java-Historie und Java 8
- Lambda-Ausdrücke
- Funktionale Interfaces
- Ergänzungen zu Lambda-Ausdrücken
 - Methodenreferenz
 - Gültigkeitsbereich von Variablen
- **Ströme**
 - Idee
 - Initiale Stromoperationen
 - Intermediäre Operationen
 - Terminale Operationen
 - **Parallele Ströme**

Parallele Ströme

- Ströme können parallelisiert werden.
- Mit einem Mehrkernprozessor kann damit die Performance verbessert werden.

```
int max = 10_000_000;  
long numberOfPrimes  
    = IntStream.range(1, max)  
                .filter(isPrime)  
                .count();
```

Sequentielle Berechnung der Anzahl
der Primzahlen zwischen 1 und max.

```
int max = 10_000_000;  
long numberOfPrimes  
    = IntStream.range(1, max)  
                .parallel()  
                .filter(isPrime)  
                .count();
```

Parallele Berechnung der Anzahl
der Primzahlen zwischen 1 und max.

Indeterministische Reihenfolge bei parallelen Strömen

- Bei der parallelen Bearbeitung eines Stroms ist die Reihenfolge, in der auf die Elemente des Stroms zugegriffen wird, nicht vorhersehbar.

```
IntStream.range(1, 21)  
    .parallel()  
    .forEach(System.out::println);
```

Elemente von 1 bis 20 werden in einer nicht vorhersehbaren Reihenfolge ausgegeben. Z.B.:

11, 6, 7, 8, 9, 10, 12, 1, 2, 3, 4, 5, 13, 16, 17, 18, 19, 20, 14, 15,

Vorsicht bei zustandsbehafteten Funktionen

- Es ist Vorsicht geboten bei Funktionen, die auf nicht-lokale Datenstrukturen zugreifen (zustandsbehaftete Funktionen, stateful functions).
- Das Ergebnis der Stromverarbeitung kann vom zeitlichen Ablauf der zustandsbehafteten Funktionsaufrufe abhängen (race condition)

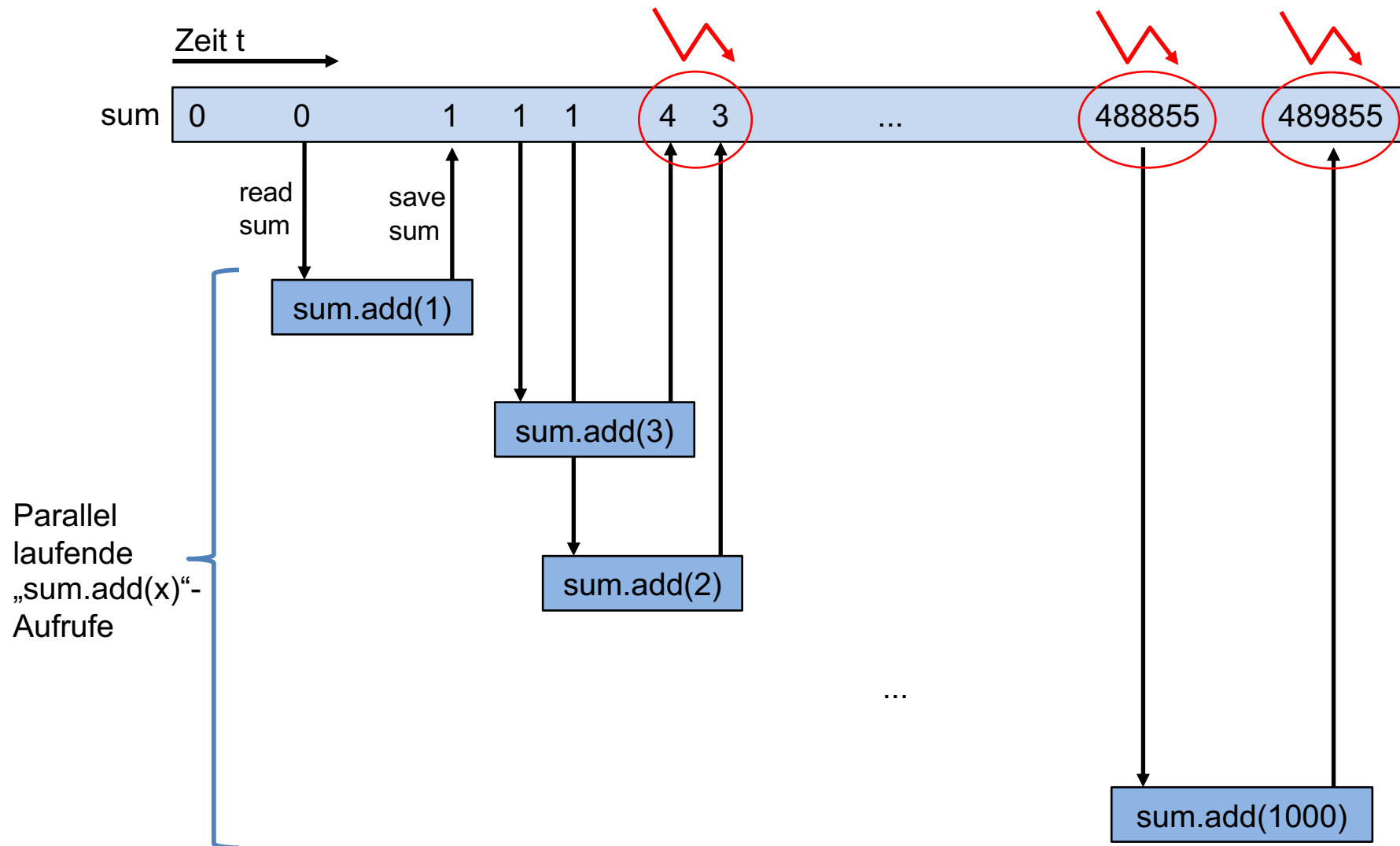
```
MutableInt sum = new MutableInt();
IntStream.range(1,1001)
    .parallel()
    .forEach( x -> sum.add(x) );
System.out.println(sum.get());
```

```
class MutableInt {
    int n = 0;
    int get() { return n; }
    void add(int x) { n += x; }
}
```

Es wird eine **zustandsbehaftete Funktion** aufgerufen, die auf die **mutable nicht-lokale Variable sum** zugreift. (Variable sum ist außerhalb des Lambda-Ausdrucks definiert.)

Ergebnis sollte eigentlich $1 + 2 + 3 + \dots + 1000 = 500500$ sein, liegt aber meistens unter 500000.

Race Condition



- Race Condition: die Berechnung von $\text{sum} = 1 + 2 + 3 \dots + 1000 = 500500$ hängt vom zeitlichen Ablauf der „`sum.add(x)`“-Aufrufe ab.

Vermeidung von Race Conditions mit synchronisierten Datentypen

- Das Paket `java.util.concurrent` enthält verschiedene Datentypen, die eine nebenläufige Benutzung unterstützen, indem der Zugriff auf die Daten geeignet koordiniert (synchronisiert) wird. Beispielsweise darf höchstens ein Thread schreibend zugreifen.
- `AtomicInteger` kapselt einen Integer-Wert und führt Änderungen des Integer-Werts atomar („in einem Schritt“) durch.
- Daher keine Race Conditions.
- Beachte: Programm wird durch Synchronisierung langsamer.

```
AtomicInteger sumAtomic = new AtomicInteger(0);  
  
IntStream.range(1,1001)  
    .parallel()  
    .forEach( x -> sumAtomic.addAndGet(x) );  
  
System.out.println(sumAtomic.get());
```

Ergebnis ist immer 500500.

Bessere Lösung: auf zustandsbehaftete Funktionen verzichten!

```
int sum = IntStream.range(1,1001)
                    .parallel()
                    .sum();
System.out.println(sum);
```

Es wird die **zustandslose Funktion** `sum()` aufgerufen.
Ergebnis ist immer wie erwartet 500500.