

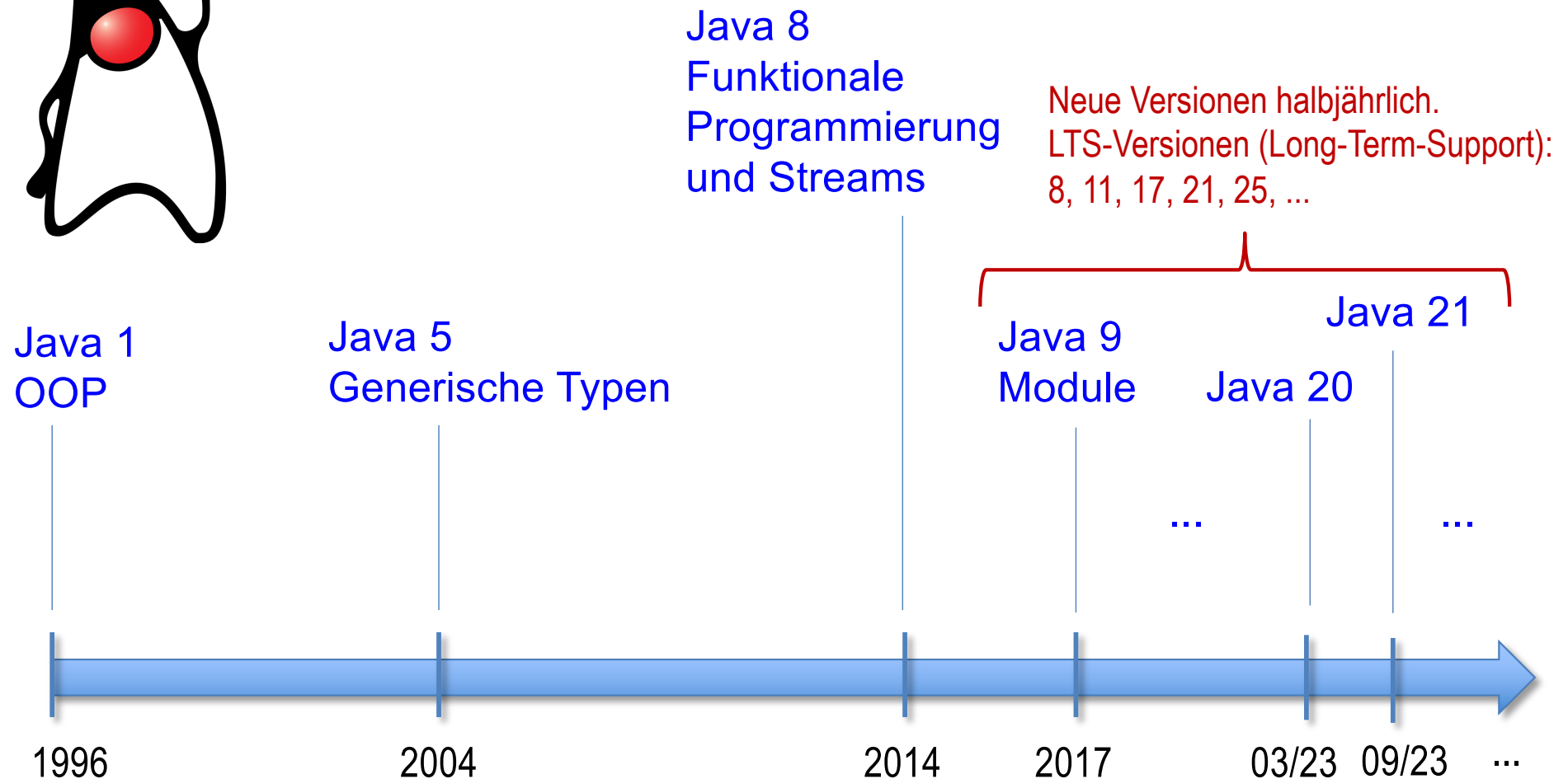
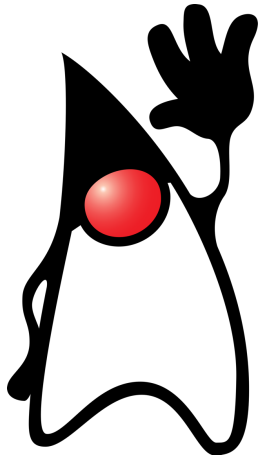
# Kapitel 13: Funktionales Programmieren und Streams

- Java-Historie und Java 8
- Lambda-Ausdrücke
- Funktionale Interfaces
- Ergänzungen zu Lambda-Ausdrücken
  - Currying
  - Methodenreferenz
  - Freie Variablen und Closures
- Erweiterungen der Java-API
  - Collection-Typen
  - Klasse Optional
- Ströme
  - Idee
  - Initiale Stromoperationen
  - Intermediäre Operationen
  - Terminale Operationen
  - Parallele Ströme

# Kapitel 13: Funktionales Programmieren und Streams

- Java-Historie und Java 8
- Lambda-Ausdrücke
- Funktionale Interfaces
- Ergänzungen zu Lambda-Ausdrücken
- Erweiterungen der Java-API
- Ströme

# Wichtige Meilensteine in der Java-Historie



# Kapitel 13: Funktionales Programmieren und Streams

- Java-Historie und Java 8
- Lambda-Ausdrücke
- Funktionale Interfaces
- Ergänzungen zu Lambda-Ausdrücken
- Erweiterungen der Java-API
- Ströme

# Beispiel: ActionListener mit anonymer innerer Klasse

---

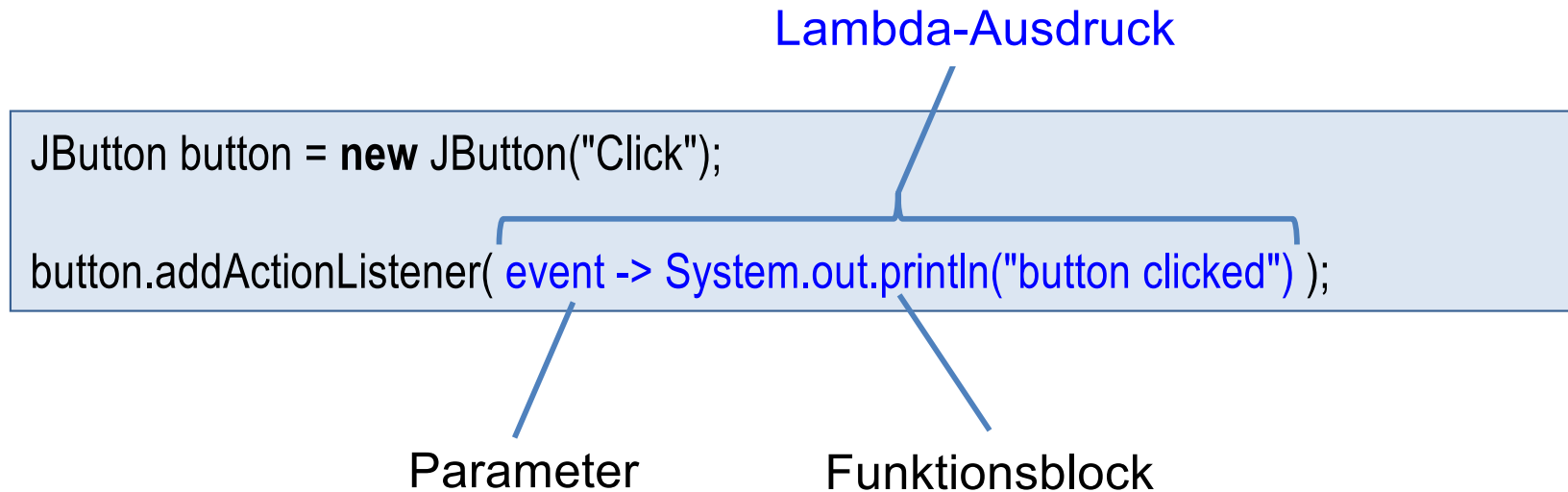
- Das folgende Beispiel zeigt typischen Java-Code, um ein bestimmtes Verhalten (Ausgabe von "button clicked" auf die Console) mit einer Swing-Komponente JButton zu verknüpfen.

```
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. Parameter müssen nicht typisiert werden. 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

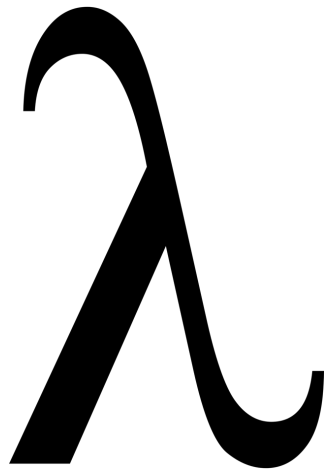
```
(int n) -> {  
    return n + 1;  
}
```

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

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

# Historisches: $\lambda$ -Kalkül

---



- Der Begriff der Lambda-Ausdrücke stammt aus dem  $\lambda$ -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  $\lambda$ -Kalkül ist auch theoretische Grundlage der [funktionalen Programmiersprachen](#) wie z.B. [Lisp](#) (1958) und [Haskell](#) (1990).
- Der  $\lambda$ -Kalkül formalisiert Konzepte wie Funktionsanwendung ([Applikation](#)) und Funktionsbildung ( [\$\lambda\$ -Abstraktion](#)):
  - MN      Applikation: wende  $\lambda$ -Term M auf N an
  - $\lambda x.M$        $\lambda$ -Abstraktion: binde die Variable x im  $\lambda$ -Term M
- Die Auswertung von  $\lambda$ -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

- Java-Historie und Java 8
- Lambda-Ausdrücke
- Funktionale Interfaces
- Ergänzungen zu Lambda-Ausdrücken
- Erweiterungen der Java-API
- 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 vom 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

- Das Paket java.util.function enthält sehr viele funktionale Interfaces.

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

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

# Beispiele

```
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 :

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

other darf auch vom Typ  $T^+ \rightarrow \text{boolean}$  sein, wobei  $T <: T^+$  ist.

- 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
```

} gleichwertig

- Typ der Methode `andThen` mathematisch geschrieben:

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

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

# Funktionales Interface Comparator (1)

- abstrakte Methode: `compare(x,y)`
- zusätzlich: verschiedene statische Methoden `comparing`
- Beispiel: `comparing(keyExtractor)` nimmt eine Funktion `keyExtractor`, die aus einem Objekt einen Comparable-Schlüssel extrahiert, entgegen 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);
    ...
}
```

```
Comparator<Person> cmp1 = (p1, p2) -> p1.getName().compareTo(p2.getName());
```

```
Comparator<Person> cmp2 = Comparator.comparing( p -> p.getName() );
```

```
List<Person> persList = new LinkedList<>();
persList.add(new Person("Maier")); ...
persList.sort(cmp2);
```

cmp1 und cmp2  
sind gleichwertig

# Funktionales Interface Comparator (2)

- zusätzlich: verschiedene Default-Methoden.
- `this.reversed()` liefert die Umkehrrelation zu `this` zurück.
- `this.thenComparing(other)` baut aus `this` und `other` einen zweistufigen lexikographischen Vergleich zusammen

```
@FunctionalInterface
interface Comparator<T> {
    default Comparator<T> reversed();
    default Comparator<T> thenComparing(Comparator<? super T> other)
    ...
}
```

```
Comparator<Person> cmpAge = Comparator.comparing( p -> p.getAge());
Comparator<Person> cmpName = Comparator.comparing( p -> p.getName());
```

```
List<Person> persList = new LinkedList<>(); ...
persList.sort(cmpAge);                // sortiere Personen nach dem Alter aufsteigend
persList.sort(cmpAge.reversed());      // sortiere Personen nach dem Alter absteigend
persList.sort(cmpAge.thenComparing(cmpName)); // sortiere Personen nach dem Alter aufsteigend und innerhalb
                                           // der gleichen Altersstufe alphabetisch nach dem Namen
```



# 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.

# Kapitel 13: Funktionales Programmieren und Streams

- Java-Historie und Java 8
- Lambda-Ausdrücke
- Funktionale Interfaces
- Ergänzungen zu Lambda-Ausdrücken
  - Currying
  - Methodenreferenz
  - Freie Variablen und Closures
- Erweiterungen der Java-API
- Ströme

# Mehrstellige Funktionen (1)

---

- Einstellige Funktionen sind vom Typ **Function**:

```
Function<Double,Double> square = x -> x*x;  
System.out.println(square.apply((3.0)));    // 9.0
```

- Zweistellige Funktionen sind vom Typ **BiFunction**:

```
BiFunction<Double, Double, Double> sum2 = (x, y) -> x+y;  
System.out.println(sum2.apply(1.0, 2.0));    // 3.0
```

- Wie sieht es aber allgemein mit **n-stelligen Funktionen** für  $n \geq 3$  aus?

```
sum3 = (x, y, z) -> x+y+z;
```

# Mehrstellige Funktionen (2)

---

- **Mehrstellige Funktionen** (genauer n-stellige Funktionen für  $n \geq 3$  ) werden von der Java API nicht direkt unterstützt.
- Lassen sich aber mit selbst definierten funktionalen Interfaces realisieren.
- **Beispiel: 3-stellige Funktion sum3**

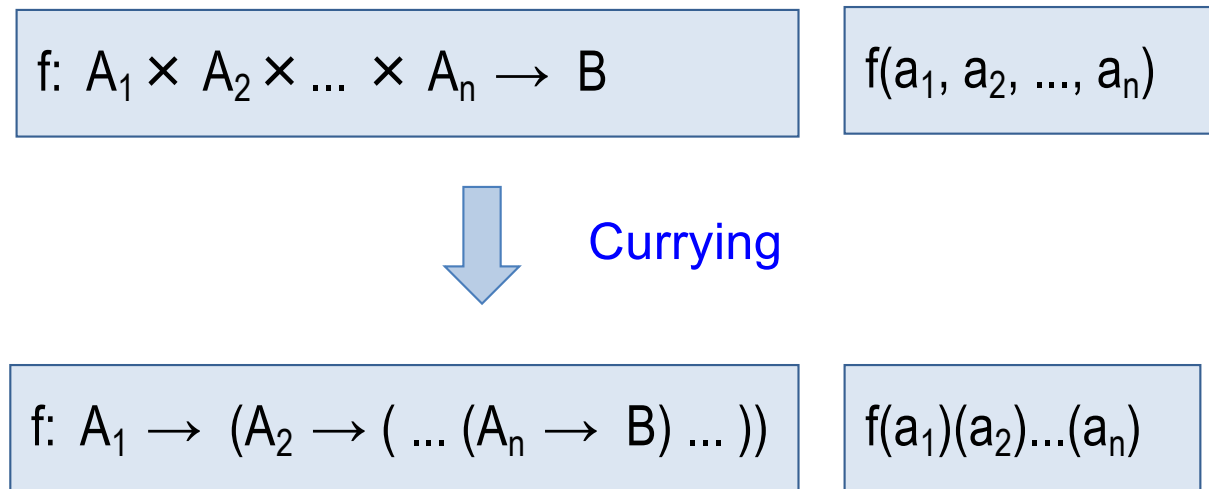
```
@FunctionalInterface  
interface TriFunction<A,B,C,R> {  
    R apply(A a, B b, C c);  
}
```

Mathematisch:  
 $A \times B \times C \rightarrow R$

```
TriFunction<Double, Double, Double, Double> sum3 = (x, y, z) -> x+y+z;  
System.out.println(sum3.apply(1.0, 2.0, 3.0));    // 6.0
```

# Currying

- Eine mehrstellige Funktion lässt sich in eine Sequenz von Funktionen mit jeweils einem Argument umwandeln:



- Verfahren wurde nach [Haskel Brooks Curry](#) benannt.
- Curry war ein US-amerikanischer Logiker und hat beim weltberühmten Mathematiker David Hilbert in Göttingen 1930 promoviert.
- Nach Curry wurde auch die wichtige pure funktionale Programmiersprache [Haskell](#) benannt.

# Currying in Java

- 2-stellige Funktion sum2 und 3-stellige Funktion sum3:

```
BiFunction<Double, Double, Double> sum2 = (x, y) -> x + y;  
TriFunction<Double, Double, Double, Double> sum3 = (x, y, z) -> x+y+z;  
System.out.println(sum2.apply(1.0, 2.0));           // 3.0  
System.out.println(sum3.apply(1.0, 2.0, 3.0));       // 6.0
```

- Currying angewandt auf sum2 und sum3:

```
Function<Double, Function<Double, Double>>  
    sum2_curried = x -> y -> x + y;  
Function<Double, Function<Double, Function<Double, Double>>>  
    sum3_curried = x -> y -> z -> x + y + z;  
System.out.println(sum2_curried.apply(1.0).apply(2.0));           // 3.0  
System.out.println(sum3_curried.apply(1.0).apply(2.0).apply(3.0)); // 6.0
```

-> ist rechtsassoziativ.  
Klammern können daher  
weggelassen werden.

# Kapitel 13: Funktionales Programmieren und Streams

- Java-Historie und Java 8
- Lambda-Ausdrücke
- Funktionale Interfaces
- Ergänzungen zu Lambda-Ausdrücken
  - Currying
  - Methodenreferenz
  - Freie Variablen und Closures
- Erweiterungen der Java-API
- 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 `getName()` aufruft.



`Person::getName`

Referenz auf Methode `getName`



# 4 Arten von Methodenreferenzen

---

|   | Methodenreferenz    | Lambda-Ausruck                                    |
|---|---------------------|---|
| Statische Methode                               | <code>C::m</code>   | <code>(x,y,...) -&gt; C.m(x,y,...)</code>         |
| Instanzmethode                                  | <code>C::m</code>   | <code>(self,x,y,...) -&gt; self.m(x,y,...)</code> |
| Instanzmethode<br>mit Bindung auf<br>Objekt obj | <code>obj::m</code> | <code>(x,y,...) -&gt; obj.m(x,y,...)</code>       |
| Konstruktor                                     | <code>C::new</code> | <code>(x,y,...) -&gt; 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.apply(5.0); // z = 5.0 + 0.0*i
```

```
Function<Double,Complex> f = Complex::new;  
Complex z = f.apply(5.0); // z = 5.0 + 0.0*i
```

Referenz auf einstelligen  
Konstruktor

# Kapitel 13: Funktionales Programmieren und Streams

- Java-Historie und Java 8
- Lambda-Ausdrücke
- Funktionale Interfaces
- Ergänzungen zu Lambda-Ausdrücken
  - Currying
  - Methodenreferenz
  - Freie Variablen und Closures
- Erweiterungen der Java-API
- Ströme

# Zustandslose Funktionen

---

- In den meisten Fällen wird in Lambda-Ausdrücken nicht auf Variablen aus der Umgebung zugegriffen.
- Man erhält dann **zustandslose Funktionen** (**stateless functions**).
- Das bedeutet auch **Funktionen ohne Seiteneffekte**.  
D.h. Variablen aus der Umgebung werden nicht verändert.

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



Jeder Aufruf liefert den  
denselben Wert zurück.

# Lambda-Ausdrücke mit freien Variablen

---

- In einem Lambda-Ausdruck können auch auf Variablen aus der Umgebung zugegriffen werden.  
Die Variable ist dann nicht als Parameter im Lambda-Ausdruck gebunden und wird auch **freie Variable** genannt.

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

freie Variable u

- **Freie Variable ist lokale Variable oder Parameter einer umfassenden Funktion:**  
Dann muss die freie Variable **effektiv final** sein.  
Bei effektiv final darf einer Variablen nur einmal ein Wert zugewiesen werden.  
Beachte: lokale Variablen und Parameter sind im **Stack** gespeichert.
- **Freie Variable ist Instanzvariable:**  
Keine Einschränkung für die freie Variable.  
Beachte: Instanzvariablen sind im **Heap** gespeichert.
- Lambda-Ausdruck ist nicht zustandslos, falls die freie(n) Variable(n) irgendwo verändert wird und der Rückgabewert des Lambda-Ausdruck von der freien Variablen abhängt.
- Die Bindung von freien Variablen an den Erstellungskontext nennt man auch **Closure (Funktionsabschluss)**.



# Beispiel: freie Variable ist lokal und Funktion ist zustandslos

---

```
class Demo {  
    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  
    }  
  
    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.

# Beispiel: freie Variable ist Instanzvariable und Funktion ist zustandsbehaftet

```
class Demo {  
    private int instanceVar = 5;  
  
    public void test() {  
        Function<Integer, Integer> g = (x) -> {  
            instanceVar++;  
            return instanceVar + x*x;  
        };  
  
        System.out.println(g.apply(5)); // 31  
        instanceVar += 2;  
        System.out.println(g.apply(5)); // 34  
    }  
  
    public static void main(String[] args) {  
        new Demo().test();  
    }  
}
```

Zugriff auf Instanzvariable  
instanceVar der Klasse Demo.

instanceVar wird bei jedem Aufruf  
von Funktion g verändert.  
g hat damit einen Seiteneffekt.

Funktion g ist außerdem  
zustandsbehaftet.

# Beispiel: freie Variable ist lokal und Funktion ist zustandsbehaftet

```
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.

# Beispiel: Ergänzung einer Funktion um einen Cache-Mechanismus <sup>1)</sup>

isPrime prüft für jedes n die Primzahleigenschaft

```
static boolean isPrime(long n) {  
    for (int i = 2; i*i <= n; i++)  
        if (n%i == 0)  
            return false;  
    return true;  
}
```

mit Cache-  
Mechanismus  
ausstatten



makeCachable ergänzt Funktion f  
um einen Cache-Mechanismus.

```
static <T, R> Function<T, R> makeCachable(Function<T, R> f) {  
    final Map<T, R> cache = new HashMap<>(); 2)  
    return t -> {  
        R r = cache.get(t);  
        if (r == null) {  
            r = f.apply(t);  
            cache.put(t, r);  
        }  
        return r;  
    };  
}
```

Lokale Variable mit mutablem  
HashMap-Objekt.

Zurückgegebener Lambda-  
Ausdruck greift zwar auf  
mutable Umgebungsvariable  
cache zu, verhält sich aber  
wie ein zustandsloser  
Lambda-Ausdruck.

```
Function<Long, Boolean> isPrimeWithCache = makeCachable(n -> isPrime(n));  
isPrimeWithCache.apply(1013L); // hier wird isPrime(1013L) aufgerufen  
isPrimeWithCache.apply(1013L); // hier wird die in cache abgespeicherte Information verwendet
```

<sup>1)</sup> Beispiel aus [Prähofer]

<sup>2)</sup> HashMap wird in Algorithmen und Datenstrukturen behandelt. Im Vergleich zu TreeMap muss T nicht Comparable sein.

# Kapitel 13: Funktionales Programmieren und Streams

- Java-Historie und Java 8
- Lambda-Ausdrücke
- Funktionale Interfaces
- Ergänzungen zu Lambda-Ausdrücken
- Erweiterungen der Java-API
  - Collection-Typen
  - Klasse Optional
- Ströme

# Interface Iterable und externe Iteration

- Interface Iterable wurde mit Java 8 um die **default-Methode** `forEach` erweitert.

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

Consumer-Funktion ist eine einstellige Funktion ohne Rückgabewert.

- Die Anwendung der `forEach`-Methode wird auch **interne Iteration** genannt.

```
List<String> nameList =  
    Arrays.asList("Maria", "Peter", "Petra", "Robert");
```

```
nameList.forEach(name -> System.out.println(name));
```

`forEach` wendet das Consumer-Argument auf jedes Element des Iterable-Objekts an.

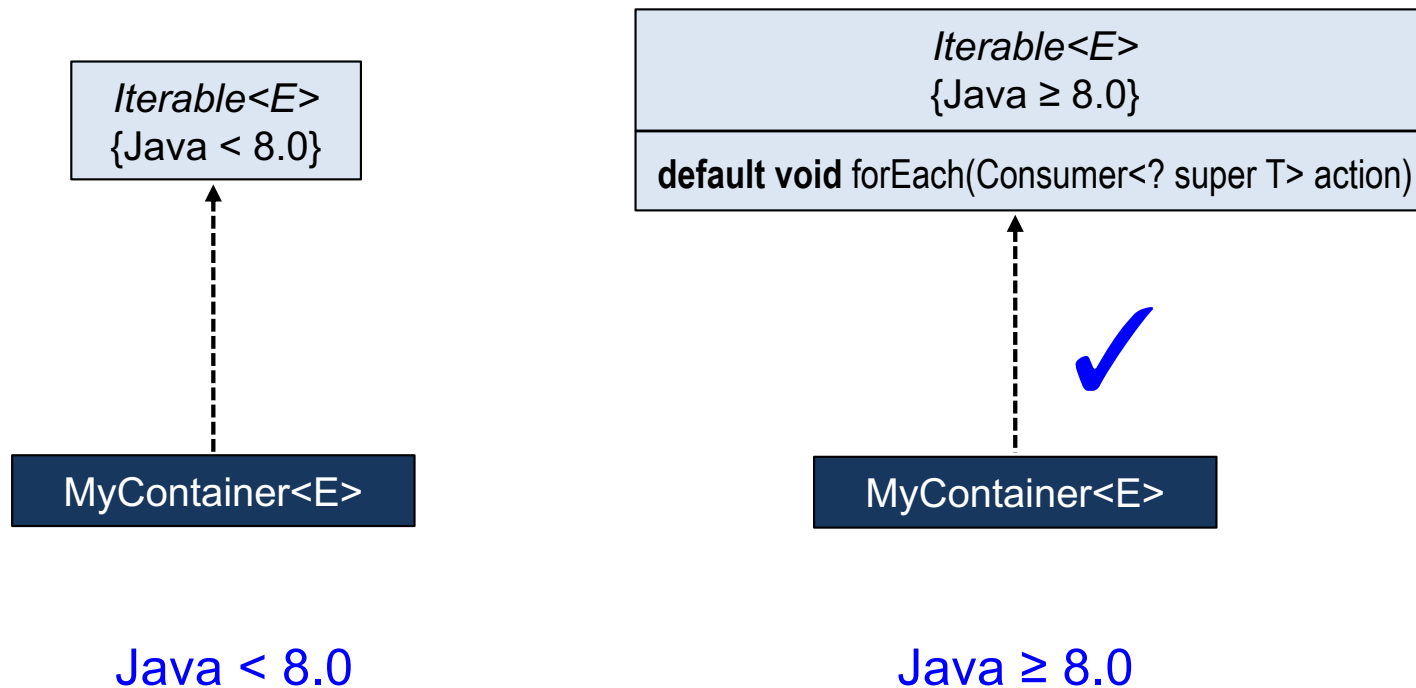
- Zum Vergleich: **externe Iteration** mit einer `for`-each-Schleife:

```
for (String s : nameList)  
    System.out.println(s);
```

- 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).

# Interface Iterable ist abwärtskompatibel

- **default-Technik** bei forEach sorgt für Abwärtskompatibilität.
- Klassen, die Iterable implementieren und vor Java 8 entwickelt wurden, brauchen nicht geändert zu werden!



# Weitere wichtige Default-Methoden

---

| Interface     | Default-Methode   |
|---------------|---|
| Collection<E> | <b>boolean</b> removeIf(Predicate<? <b>super</b> E> filter)   |
| List<E>       | <b>void</b> replaceAll(UnaryOperator<E> operator)<br><b>void</b> sort(Comparator<? <b>super</b> E> c) |

| Interface | Default-Methode  |
|-----------|--|
| Map<K,V>  | compute<br>computeIfAbsent<br>computeIfPresent<br>forEach<br>getOrDefault<br>merge<br>putIfAbsent<br>replace<br>replaceAll |

Details siehe Java-API. **Anschauen!**



# Beispiel zu Default-Methoden aus Map

```
Map<String, Set<String>> besuchteLVen = new TreeMap<>();
```

Für jeden Student (String) wird Menge der besuchten Lehrveranstaltungen (Set<String>) in einer Map gespeichert.

```
if (besuchteLVen.get("Peter") == null)  
    besuchteLVen.put("Peter", new TreeSet<>());  
besuchteLVen.get("Peter").add("Prog2");
```

Peter besucht Prog2.

```
besuchteLVen.computeIfAbsent("Petra", s -> new TreeSet<>() ).add("Symo");
```

**computeIfAbsent** liefert zu key "Petra" aktuellen value zurück.

Lambda-Funktion berechnet value falls key "Petra" nicht vorhanden ist.

Petra besucht Symo.  
Mit **Default-Methode** **computeIfAbsent**.

```
for (var e : besuchteLVen.entrySet())  
    System.out.println(e.getKey() + ": " + e.getValue());
```

besuchteLVen zeilenweise ausgeben.

```
besuchteLVen.forEach((s, lv) -> System.out.println(s + ": " + lv));
```

besuchteLVen zeilenweise ausgeben mit **Default-Methode** **forEach**.

# Klasse Optional<T>

- Optional<T> kapselt einen Wert vom Typ T, der null (leeres Optional) oder nicht-null sein kann.

| Methode  | Beschreibung  |
|--|---|
| <b>static</b> <T> Optional<T> empty()                  | Liefert ein leeres Optional zurück  |
| <b>static</b> <T> Optional<T> of(T value)              | Liefert Instanz mit value zurück. value darf nicht null sein.                           |
| <b>static</b> <T> Optional<T> ofNullable(T value)      | Liefert Instanz mit value zurück, falls value nicht null ist sonst ein leeres Optional. |
| <b>boolean</b> isPresent()<br><b>boolean</b> isEmpty() | Prüft, ob ein Wert bzw. kein Wert vorhanden ist.  |
| V get()  | Liefert Wert, falls vorhanden, sonst NoSuchElementException.                            |
| V orElse(V other)                                      | Liefert Wert, falls vorhanden, sonst other.   |
| ...  |   |

```
Optional<String> s1 = Optional.ofNullable(null);
Optional<String> s2 = Optional.of("abc");
Optional<String> s3 = Optional.empty();

System.out.println(s1.isEmpty());      // true
System.out.println(s2.isEmpty());      // false
System.out.println(s3.isEmpty());      // true
System.out.println(s1.orElse("???"));  // ???
System.out.println(s2.orElse("???"));  // abc
System.out.println(s3.orElse("???"));  // ???
```

# Beispiel: Maximum eines Felds

## Konventionelle Lösung mit Exception:

```
public static int max_V1(int[] a) {  
    if (a == null || a.length == 0)  
        throw new IllegalArgumentException();  
    int max = a[0];  
    for (int e : a)  
        if (e > max)  
            max = e;  
    return max;  
}
```

## Lösung mit OptionalInt:

(OptionalInt ist ein leichtgewichtiger Ersatz für Optional<Integer>)

```
public static OptionalInt max_V2(int[] a) {  
    if (a == null || a.length == 0)  
        return OptionalInt.empty();  
    int max = a[0];  
    for (int e : a)  
        if (e > max)  
            max = e;  
    return OptionalInt.of(max);  
}
```

```
int[] a = ...
```

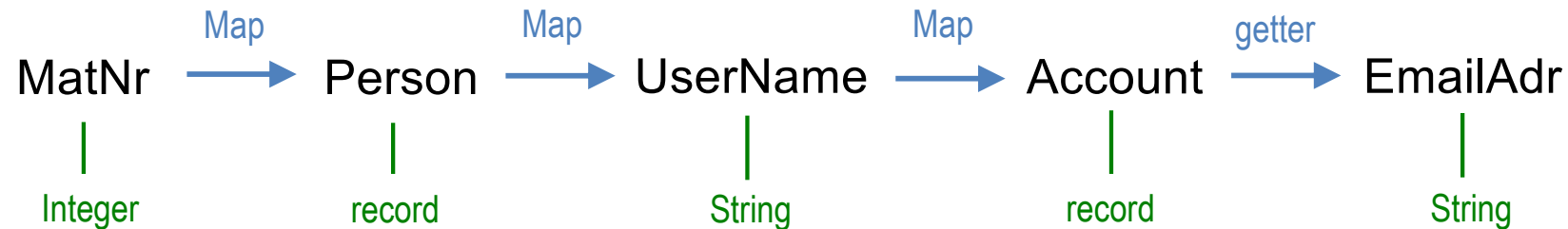
```
System.out.println(max_V1(a));
```

```
System.out.println(max_V2(a));
```

Kann eine Exception auslösen.

Geht immer.

# Beispiel: Verkettung von Maps



```
record Person(String name, LocalDate geb) { }
```

```
record Account(String name, String email) { }
```

```
Map<Integer,Person> matNr2Pers = new HashMap<>();
```

```
Map<Person,String> pers2UserName = new HashMap<>();
```

```
Map<String,Account> userName2Account = new HashMap<>();
```

```
Person p1 = new Person("Peter Maier", LocalDate.of(2000,1,1));
```

```
// ...
```

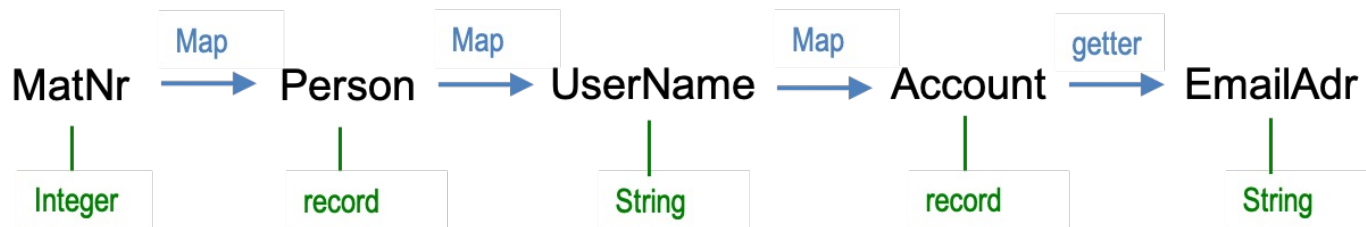
```
matNr2Pers.put(54321, p1);
```

```
pers2UserName.put(p1, "maierpeter");
```

```
userName2Account.put("maierpeter", new Account("maierpeter","peter.maier@htwg-konstanz.de"));
```

```
// ...
```

# Konventionelle Ermittlung einer Email-Adresse mit if-Kaskaden



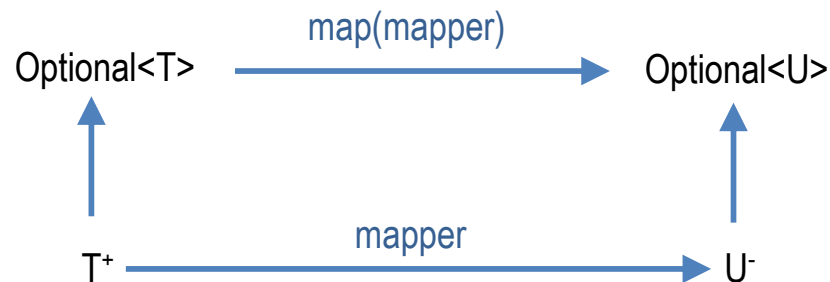
```
Integer matNr = 54321;
String email = "???";

Person pers = matNr2Pers.get(matNr);
if (pers != null) {
    String userName = pers2UserName.get(pers);
    if (userName != null) {
        Account acc = userName2Account.get(userName);
        if (acc != null) {
            email = acc.emailAdr();
        }
    }
}
System.out.println(matNr + ": " + email);
```

# Optional mit map-Methode

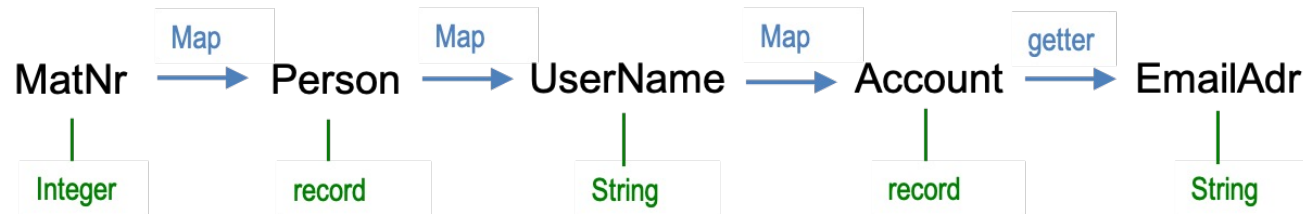
```
class Optional<T> {  
    public <U> Optional<U> map(Function<? super T, ? extends U> mapper) { ... }  
    // ...  
}
```

- Ausgangspunkt ist eine Funktion mapper:  $T^+ \rightarrow U^-$ .
- `map(mapper)` hebt mapper auf  $\text{Optional}<T> \rightarrow \text{Optional}<U>$  hoch.



- `o.map(mapper)` bildet `o` vom Typ `Optional<T>` auf ein Objekt vom Typ `Optional<U>` ab.
- Wenn ein Wert in `o` vorhanden ist, wird ein `Optional` zurückgegeben, das das Ergebnis der Anwendung der Funktion `mapper` auf den Wert beinhaltet, andernfalls wird ein leeres `Optional` zurückgegeben.
- Wenn die Funktion `mapper` null liefert, dann gibt `o.map(mapper)` ein leeres `Optional` zurück.

# Funktionale Ermittlung einer Email-Adresse mit Optional und map



```
Integer matNr = 54321;

String email = Optional.ofNullable(matNr2Pers.get(matNr))
    .map(p -> pers2UserName.get(p))
    .map(u -> userName2Account.get(u))
    .map(a -> a.email())
    .orElse("???");

System.out.println(matNr + ": " + email);
```

- Im Vergleich zu den if-Kaskaden bei der konventionellen Ermittlung der Email-Adresse ist der funktionale Ausdruck abstrakter und prägnanter.
- Die neu eingeführte Klasse Optional spielt im Rahmen der funktionalen Programmierung eine wichtige Rolle.
- Die Klasse Optional wird sehr intensiv bei der Stromverarbeitung (später) verwendet.

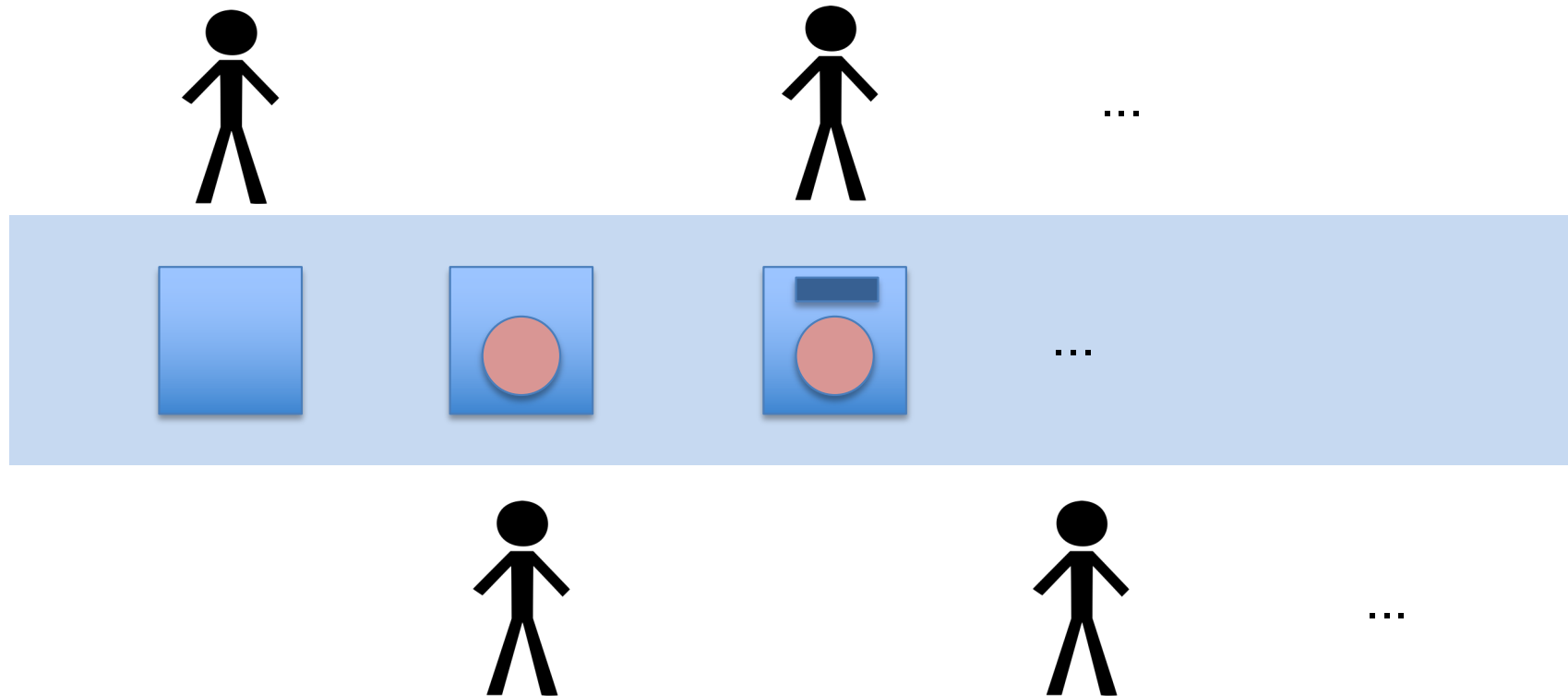
# Kapitel 13: Funktionales Programmieren und Streams

- Java-Historie und Java 8
- Lambda-Ausdrücke
- Funktionale Interfaces
- Ergänzungen zu Lambda-Ausdrücken
- Erweiterungen der Java-API
- **Ströme**
  - Idee
  - Initiale Stromoperationen
  - Intermediäre Operationen
  - Terminale Operationen
  - Parallele Ströme



# Fließband (pipeline)

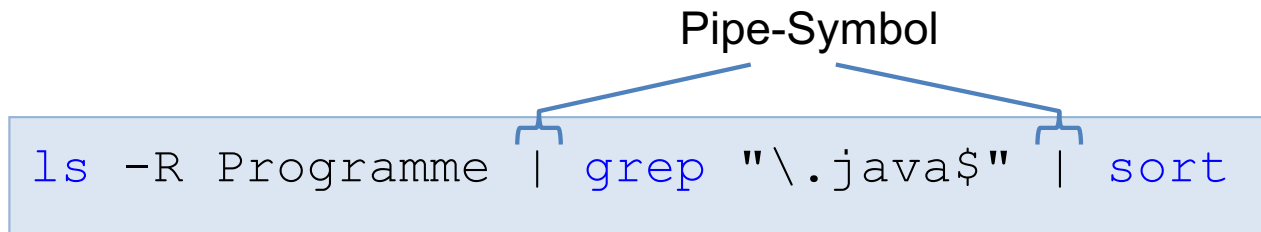
---



- 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.

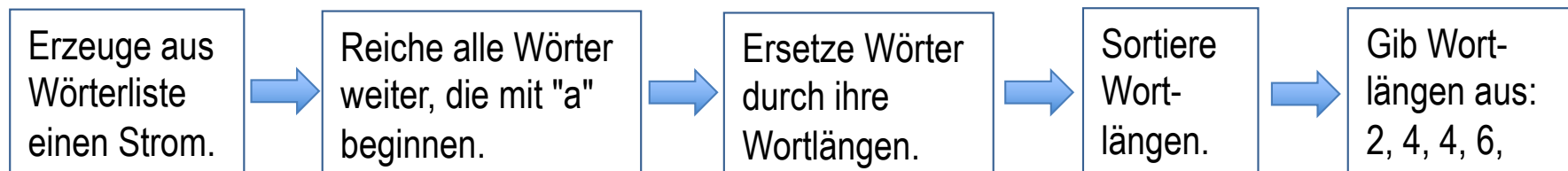


Mit `sort` wird die Ausgabe von `grep` entgegengenommen, sortiert und auf die Konsole ausgegeben.

# Ströme (streams) in Java 8

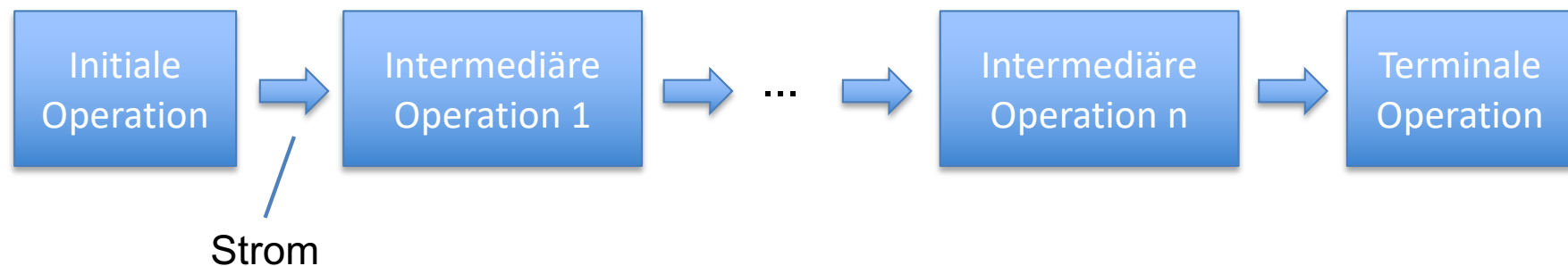
- **Ströme** sind eine (evtl. unendlich lange) Folge von Datenobjekte.
- Die Datenobjekte eines Stroms werden von Methoden verarbeitet und können dann zur nächsten Methode weitergereicht werden (**Pipeline-Verarbeitung**).
- Das Stromkonzept von Java hat damit große Ähnlichkeit zu den Unix-Pipes.

```
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.  
Neben den generischen Strömen **Stream<T>** gibt auch Ströme für Basisdatentypen wie **IntStream**, **LongStream** und **DoubleStream**.



# Beispiel

```
long countPrimes =
```

```
    IntStream.range(2, 1000)
```

```
        .filter(n -> isPrime(n))
```

```
        .peek(n -> System.out.println(n))
```

```
        .count();
```

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

## Initiale Operation:

Erzeuge mit der Fabrikmethode range einen Zahlen-Strom von 2 (einschl.) bis 1000 ausschließl.



## Intermediäre Operation 1:

Lasse nur Primzahlen im Strom.



## Intermediäre Operation 2:

Gib Zahl aus und reiche sie weiter.



## Terminale Operation:

Bestimme die Anzahl der Zahlen im Zahlenstrom und liefere sie als long zurück.

# Verzögerte Generierung der Ströme

---

- Ströme werden nie komplett im voraus generiert.  
Beachte: **Ströme** können **prinzipiell unendlich lang** werden.
- Es werden nur so lange 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

- Java-Historie und Java 8
- Lambda-Ausdrücke
- Funktionale Interfaces
- Ergänzungen zu Lambda-Ausdrücken
- Erweiterungen der Java-API
- **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.

|                |   |  |
|----------------|---|--|
| Collection<T>  | Stream<T> stream()<br>Stream<T> parallelStream()  | sequentiellen bzw. parallelen Strom mit Elementen aus der Collection.    |
| Arrays         | <b>static</b> <T> Stream<T> stream(T[ ] a)<br>... | Strom mit Elementen aus Feld a.<br>Weitere Methoden für Basisdatentypen. |
| String         | IntStream chars()                                 | Strom von Zeichen (als Integer-Strom) aus dem String.                    |
| BufferedReader | lines()   | Strom aus Zeilen des BuffereredReader.                                   |
| ...            |   |  |

- 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`.

|                           |   |
|---------------------------|---|
| <code>empty()</code>      | Leerer Strom.   |
| <code>of(...)</code>      | Strom mit vorgegebenen Elementen  |
| <code>generate(s)</code>  | Generiere Strom durch wiederholtes Aufrufen von <code>s</code> : <code>s()</code> , <code>s()</code> , <code>s()</code> , ... |
| <code>iterate(a,f)</code> | Generiere Strom durch Iteration: <code>a</code> , <code>f(a)</code> , <code>f(f(a))</code> , ...                              |
| <code>range(a,b)</code>   | Generiere Integer-Strom von <code>a</code> einschl. bis <code>b</code> 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)            |
| doubles(streamSize)       | Strom mit streamSize vielen zufälligen double-Zahlen aus [0,1)           |
| doubles(streamSize, a, b) | Strom mit streamSize vielen zufälligen double-Zahlen aus [a,b)           |
| ints()                    | Strom mit unendlich vielen zufälligen int-Zahlen                         |
| ints(streamSize)          | Strom mit streamSize vielen zufälligen int-Zahlen                        |
| ints(streamSize, a, b)    | Strom mit streamSize vielen zufälligen int-Zahlen aus {a, a+1, ..., b-1} |

- Beispiele:

```
IntStream s1 = new Random().ints();           // unendl. Strom mit zufälligen int-Zahlen
IntStream s1 = new Random().ints(10);         // Strom mit 10 zufälligen int-Zahlen
IntStream s1 = new Random().ints(20, 0, 100); // Strom mit 20 Zahlen aus {0, 1, ..., 99}
```

# Kapitel 13: Funktionales Programmieren und Streams

- Java-Historie und Java 8
- Lambda-Ausdrücke
- Funktionale Interfaces
- Ergänzungen zu Lambda-Ausdrücken
- Erweiterungen der Java-API
- **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 Strom-Operationen möglich.

```
strom.op_1(...)  
    .op_2(...)  
  
    ...  
  
    .op_n();
```

# Intermediäre Strom-Operationen

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

|                              |  |
|------------------------------|--|
| <code>filter(pred)</code>    | lasse nur Elemente <code>x</code> im Strom, für die das Prädikat <code>pred(x)</code> zutrifft.  |
| <code>map(f)</code>          | ersetze jedes Element <code>x</code> im Strom durch <code>f(x)</code> .  |
| <code>flatMap(f)</code>      | ersetze jedes Element <code>x</code> im Strom durch einen von <code>f(x)</code> erzeugten Strom.   |
| <code>peek(action)</code>    | führe für jede Methode die rückgabelose Funktion <code>action</code> durch.<br>Wird hauptsächlich zu Debugging-Zwecke eingesetzt.  |
| <code>sorted()</code>        | sortiere die Elemente im Strom (stabiles Sortierverfahren).<br>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 <code>n</code> Elemente aus dem Strom.   |
| <code>limit(n)</code>        | begrenze den Strom auf maximal <code>n</code> Elemente.  |
| <code>takeWhile(pred)</code> | Lasse alle Elemente im Strom, solange das Prädikat <code>pred</code> zutrifft.<br>Sobald Prädikat <code>pred</code> nicht mehr zutrifft, werden alle restlichen Elemente gelöscht. |
| <code>dropWhile(pred)</code> | Entferne alle Elemente von Beginn an, bis das Prädikat nicht mehr zutrifft.  |
| ...                          |  |

# 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

- Java-Historie und Java 8
- Lambda-Ausdrücke
- Funktionale Interfaces
- Ergänzungen zu Lambda-Ausdrücken
- Erweiterungen der Java-API
- **Ströme**
  - Idee
  - Initiale Stromoperationen
  - Intermediäre Operationen
  - Terminale Operationen
  - Parallele Ströme



# Terminale Operationen

---

- Mit einer **terminalen Operation** wird der Strom abgeschlossen.
- **Reduktionsoperationen** liefern ein Resultat zurück (aber keinen Strom) wie z.B. `sum()`:

```
int s = strom.sum();
```

- Operationen ohne Rückgabewert und Seiteneffekt wie z.B. `forEach(action)`.

```
strom.forEach(System.out::println);
```

- Sind im **Paket `java.util.stream`** festgelegt.

# 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.  |

- Operationen werden mittels **short-circuit evaluation** ausgewertet.  
D.h. sobald der Rückgabewert ermittelt werden kann, wird der Strom abgebrochen.
- Beispiele:

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

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

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

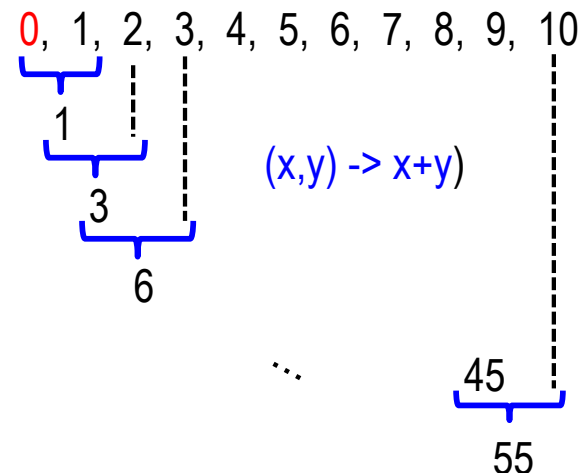
Gibt so lange zufällige Zahlen `x` aus, bis ein `x ∉ [10, 1000)` ist.  
Rückgabewert ist dann false.

# Algebraische Reduktions-Operationen

|                      |  |
|----------------------|--|
| reduce(e, op)        | reduziert einen Strom<br>$x_0, x_1, x_2, \dots$<br>zu dem Wert<br>$(\dots(((e \text{ op } x_0) \text{ op } x_1) \text{ op } x_2) \text{ op } \dots$<br>Dabei ist op ein 2-stelliger assoziativer Operator und e das neutrale Element bzgl. op. |
| count()              | Anzahl der Elemente im Strom.  |
| min(cmp)<br>max(cmp) | Liefert größtes bzw. kleinstes Element des Stroms bzgl. der Comparator-Funktion cmp.<br>Beachte: Rückgabewerttyp ist Optional<T>.  |

- Beispiel: `reduce(0, (x,y) -> x+y)` berechnet die Summe aller Elemente des Stroms.

```
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, (t,v) -> t + 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:

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

# Einschub: Monoid

## ■ Definition

Eine algebraische Struktur  $(M, \oplus, e)$  heisst **Monoid**, falls:

(1)  $\oplus$  ist eine 2-stellige Operation auf M:

$$\oplus : M \times M \rightarrow M$$

(1) e ist neutrales Element:

$$a \oplus e = e \oplus a = a$$

(2)  $\oplus$  ist assoziativ

$$a \oplus (b \oplus c) = (a \oplus b) \oplus c$$

## ■ Beispiele

- $(\mathbb{Z}, +, 0)$
- $(\mathbb{Z}, *, 1)$
- $(\text{String}, +, "")$   
Strings mit Konkationation und leerem String als neutralem Element
- $(\mathbb{R}^{n \times n}, *, I)$  Matrizen mit Multiplikation

- Reihenfolge der Auswertung kann beliebig erfolgen.  
Daher gut für Parallelisierung geeignet.

|   |  |
|---|--|
| $a \oplus (b \oplus (c \oplus (d \oplus (e \oplus f))))$  | (Auswertung von rechts)  |
| $= (((a \oplus b) \oplus c) \oplus d) \oplus e) \oplus f$ | (Auswertung von links)   |
| $= (a \oplus b \oplus c) \oplus (d \oplus e \oplus f)$    | (linker und rechter Teilausdruck lassen sich parallel auswerten) |

- Für die Methode `reduce(op, id)` aus `Stream<T>` muss  $(T, op, id)$  ein Monoid sein.

# Statistik-Operationen für Basisdatentypen

---

|  |   |
|--|---|
| count(), sum()<br>min(), max(),<br>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}
```

# Sammeln der Strom-Elemente in einem Behälter mit collect

|                              |   |   |
|------------------------------|---|---|
| <code>Stream&lt;T&gt;</code> | <pre>&lt;R&gt; R collect(<br/>    Supplier&lt;R&gt; supplier,<br/>    BiConsumer&lt;R, ? <b>super</b> T&gt; accumulator,<br/>    BiConsumer&lt;R,R&gt; combiner<br/>)</pre> | <p>collect sammelt Strom-Elemente vom Typ T in einem Behälter vom Typ R.</p> <p>supplier erzeugt einen initialen Behälter.</p> <p>Mit accumulator werden die Elemente von Typ T in den Behälter abgelegt.</p> <p>combiner beschreibt, wie 2 Container verschmolzen werden (ist für parallele Streams wichtig)</p> |
|------------------------------|---|---|

## ■ Beispiele:

```
Stream<String> wordStream = ...  
List<String> wordList = wordStream.collect(  
    () -> new ArrayList<>(),           // supplier  
    (list, w) -> list.add(w),           // accumulator  
    (list1, list2) -> list1.addAll(list2) // combiner  
);
```

Sammeln von Strings in einer Liste

```
String sentence = wordStream.collect(  
    () -> new StringBuilder(),           // supplier  
    (s, w) -> s.append(w).append(", "), // accumulator  
    (s1, s2) -> s1.append(s2)           // combiner  
)  
.toString();
```

Sammeln von Strings in einem StringBuilder



# Interface Collector und Klasse Collectors

---

- Die für das Sammeln notwendigen Funktionen können auch in einem **Collector-Objekt** zusammengefasst und der collect-Methode übergeben werden.

|                        |  |  |
|------------------------|--|--|
| <b>Stream&lt;T&gt;</b> | <code>&lt;R, A&gt; R collect(<br/>Collector&lt;? <b>super</b> T, A, R&gt; collector<br/>)</code> | collect sammelt Stream-Elemente vom Typ T in einem Behälter vom Typ R mit Hilfe von collector<br><br>A ist ein intermediärer Akkumulationstyp. |
|------------------------|--|--|

- In die Details eines Collector-Objekts soll hier nicht näher eingegangen werden.
- Es gibt in der Klasse **Collectors** bereits zahlreiche **statische Methoden**, um Collector-Objekte zu erzeugen.
- Beispiele auf folgenden beiden Folien (aus [Prähofer]).
- Gute Beispiele außerdem in der Java-API für die Klasse Collectors.

# Collectors-Beispiele: Sammeln in Listen, Sets und Strings

---

```
Stream<String> wordStream = ...  
List<String> wordList = wordStream.collect(Collectors.toList());
```

Sammeln von Strings in einer Liste

```
List<String> wordList = wordStream.toList();
```

Nützliche Kurzform!

```
Set<String> wordSet = wordStream.collect(Collectors.toSet());
```

Sammeln von Strings in einem Set.  
Dabei wird ein HashSet-Objekt eingesetzt.

```
Set<String> wordSet = wordStream.collect(  
    Collectors.toCollection(TreeSet::new));
```

Sammeln von Strings in einem TreeSet.

```
String wordString = wordStream.collect(Collectors.joining(" ", "{", "}" ));
```

Seperator-String      Anfangs- und End-String

Verbinden von Strings zu einem Gesamt-String

# Collectors-Beispiele: Sammeln in Maps

```
Map<String, Integer> wordCount = wordStream.collect(  
    Collectors.toMap(w -> w, w -> 1, (n1,n2) -> n1+n2));
```

Key  
Mapper

Value  
Mapper

Merger: Beschreibt wie 2 Map-Einträge mit  
gleichem Schlüssel zusammengefasst werden

Erstellen einer Häufigkeitstabelle als Map.  
(Kurzform von Aufgabe 1 ☺)

```
Map<Character, List<String> > wordGroups =  
    wordStream.collect(  
        Collectors.groupingBy(w -> w.charAt(0))  
    );
```

Classifier

Gruppierung von Strings nach dem  
Anfangsbuchstaben in einer Map

```
Map<Boolean, List<String> > upperLowerWords =  
    wordStream.collect(  
        Collectors.partitioningBy(w -> Character.isUpperCase(w.charAt(0)))  
    );
```

Prädikat

Partitionierung mit  
einem Prädikat.

# Collectors-Beispiele: Sammeln in Maps mit einem downstream-Collector (1)

---

Erstellen einer Häufigkeitstabelle:

```
List<String> words = List.of("abc", "def", "abc", "def", "abc", "xyz");  
  
Map<String, Integer> wordCount =  
    words.stream.collect(  
        Collectors.groupingBy(w->w, Collectors.counting() )  
    );
```

Key  
Mapper

Down Stream Collector.  
Es wird die Anzahl je Gruppe gebildet.

```
System.out.println(wordCount1);  
// {abc=3, def=2, xyz=1}
```

## Collectors-Beispiele: Sammeln in Maps mit einem downstream-Collector (2)

```
Map<Character, SortedSet<String>> wordGroups =  
    wordStream.collect(  
        Collectors.groupingBy(w -> w.charAt(0), Collectors.toCollection(TreeSet::new))  
    );
```

Classifier

Down Stream Collector.  
Verantwortlich für die Sammlung je Klasse.

Gruppierung von Strings nach dem Anfangsbuchstaben in einer Map.  
Die Gruppen sollen jeweils in einer SortedSet gespeichert werden.

```
Map<City, Set<String>> namesByCity =  
    persList.stream().collect(  
        Collectors.groupingBy(Person::city, Collectors.mapping(Person::lastName, Collectors.toSet()))  
    );
```

Classifier

Down Stream Collector.  
Es werden nur die Nachnamen in einer Liste gesammelt.

Gruppierung von Personen nach der Stadt.  
Bei den Gruppen werden nur die Nachnamen in einer HashSet gespeichert.

# Kapitel 13: Funktionales Programmieren und Streams

- Java-Historie und Java 8
- Lambda-Ausdrücke
- Funktionale Interfaces
- Ergänzungen zu Lambda-Ausdrücken
- Erweiterungen der Java-API
- **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 mutable Umgebungsvariablen zugreifen (zustandsbehaftete Funktionen, stateful functions).
- Das Ergebnis der Stromverarbeitung kann vom zeitlichen Ablauf der zustandsbehafteten Funktionsaufrufe abhängen (race condition)

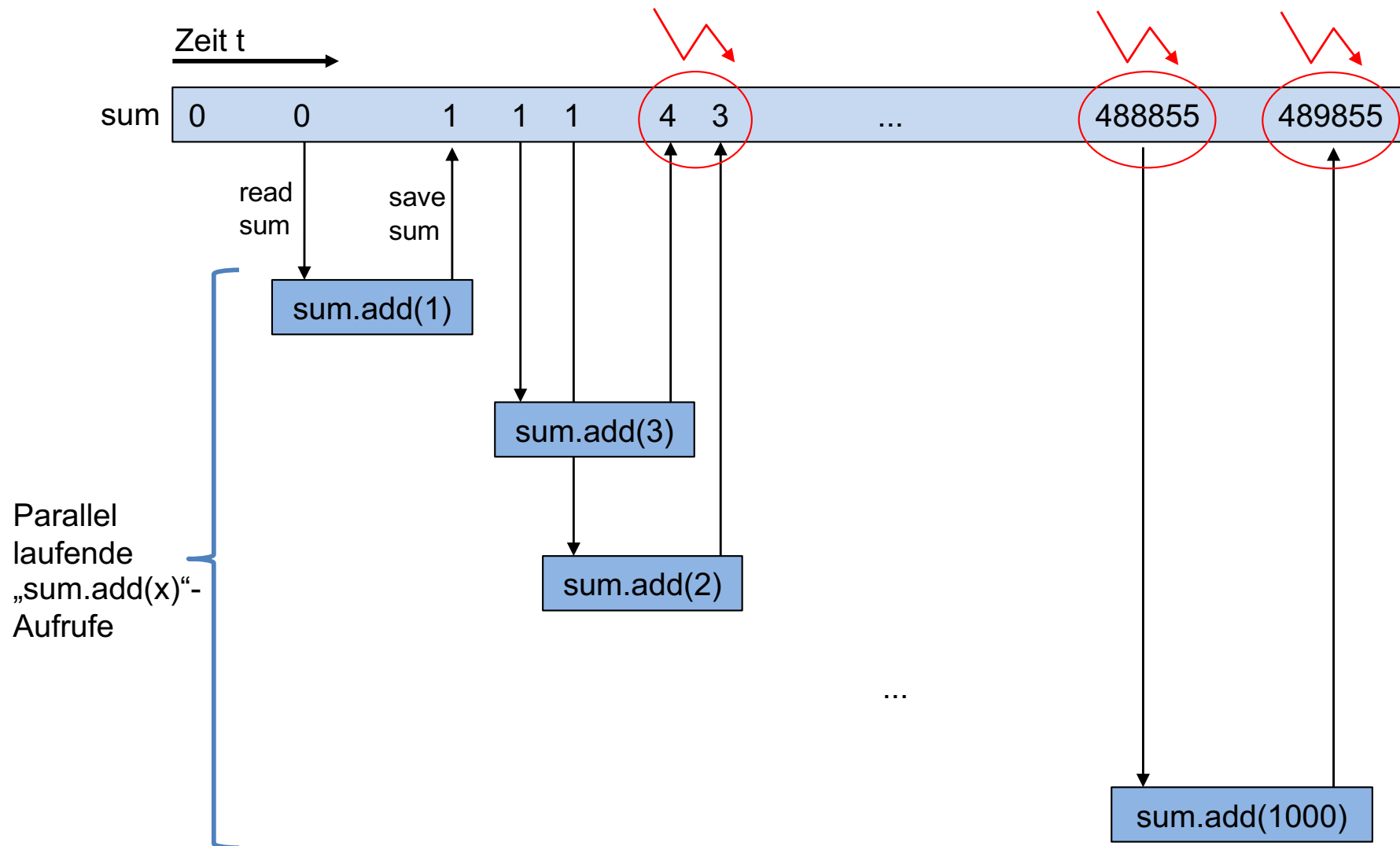
```
MutableInt sum = new MutableInt();
IntStream.range(1,1001)
    .parallel()
    .forEach( x -> sum.add(x) );
System.out.println(sum.get());
```

```
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 Umgebungsvariable 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.