

# Objektorientierte Modellierung und Programmierung

Dr. Christian Schönberg

# Streams

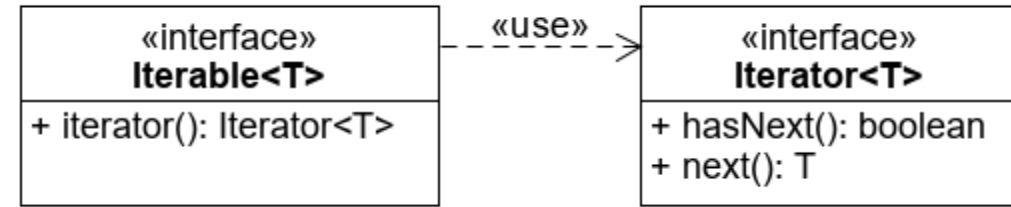
- Lokale Klassen
- Anonyme Klassen
- Lambda-Ausdrücke
- Streams

# Lokale Klassen

- Verschachtelte Klassen (Inner Classes, Static Classes)
  - enger technischer oder semantischer Zusammenhang mit der Outer Class
  - wird nur in lokalem Kontext verwendet
- Was tun wir, wenn die Klasse nur in einem sehr engen Kontext (z.B. innerhalb einer Methode) benötigt wird?  
Ist es möglich, aus dieser Klasse auf Variablen und Parameter der Methode zuzugreifen?
  - Lokale Klasse
- Was tun wir, wenn die Klasse nur ein einziges Mal benötigt wird?
  - Anonyme Klasse

- Verschachtelte Klassen (insb. Member Classes) können überall dort definiert werden, wo andere Elemente einer Klasse auch definiert werden können (Attribute, Methoden, ...)
- **Lokale Klassen** liegen noch näher am Programmcode: Sie werden innerhalb von Blöcken `{ ... }` definiert
- Insbesondere können Lokale Klassen innerhalb von Methoden definiert werden, aber auch innerhalb von Schleifen oder Bedingungs-Blöcken

# Beispiel: Lokale Klasse



```
public class ReverseArrayCollection<T> implements Iterable<T> {

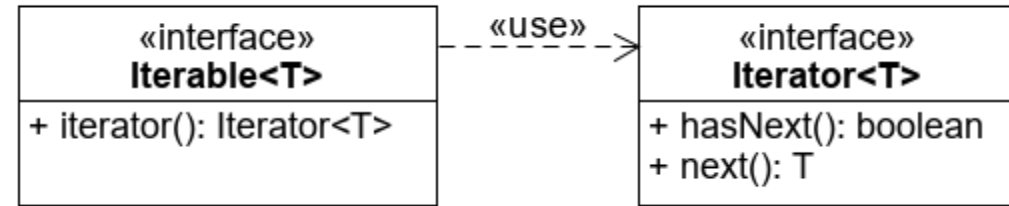
    protected T[] data;

    public ReverseArrayCollection(T[] data) {
        this.data = data;
    }

    public T[] getData() {
        return data;
    }

    public void setData(T[] data) {
        this.data = data;
    }
}
```

# Beispiel: Lokale Klasse



```
@Override
public Iterator<T> iterator() {
    class ReverseArrayIterator implements Iterator<T> {

        private int index = data.length - 1;

        @Override
        public boolean hasNext() {
            return data != null && index >= 0;
        }

        @Override
        public T next() {
            T result = data[index];
            index--;
            return result;
        }
    }
    return new ReverseArrayIterator();
}
```




# Eigenschaften von Lokalen Klassen

- Lokale Klassen haben Zugriff auf
  - alle Elemente (auch `private`) der Outer Class
    - Lokale Klassen in statischen Methoden können nur auf die statischen Elemente der Outer Class zugreifen
  - alle lokalen Variablen und Parameter der umgebenden Methode, die **final** oder *effectively final* (Java >= 8) sind
    - alle Werte, die nach der Zuweisung nicht mehr verändert werden
- Lokale Klassen sind nur innerhalb des Blocks sichtbar, in dem sie definiert werden
  - für sie kann keine Sichtbarkeit (**public**, ...) definiert werden
  - sie können nicht statisch sein
  - sie können keine statischen Elemente außer Konstanten enthalten
- Interfaces können **nicht** lokal definiert werden

## Beispiel: Lokale Klasse (2)

```
public Iterator<T> randomIterator(final long seed) {  
    class RandomArrayIterator implements Iterator<T> {  
  
        private Random random = new Random(seed);  
  
        @Override  
        public boolean hasNext() {  
            return data != null;  
        }  
  
        @Override  
        public T next() {  
            return data[random.nextInt(data.length)];  
        }  
    }  
    return new RandomArrayIterator();  
}
```



# Anonyme Klassen

- **Anonyme Klassen** sind Lokale Klassen ohne Namen
- Sie können deshalb nur einmal instanziiert werden, da es ja keinen Namen gibt, auf den sich die Instanziierung beziehen könnte
- Sie können deshalb auch keinen (expliziten) Konstruktor haben
- Sie werden immer als Unterklasse einer gegebenen Klasse oder als implementierende Klasse eines gegebenen Interfaces definiert

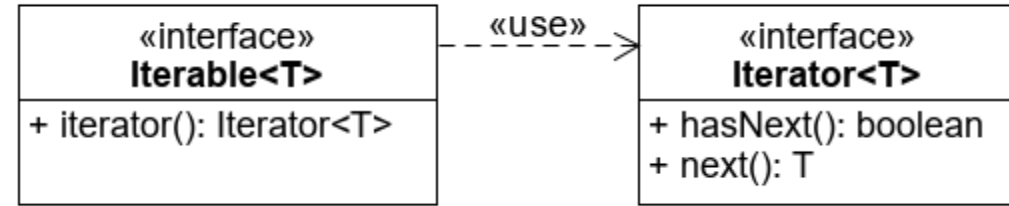
```
Supertype instance = new Supertype() {  
    // class definition  
};
```

# Beispiel: Anonyme Klasse

```
// create a (pseudo-)random number generator
// for only positive integers
Random random = new Random() {
    @Override
    public int nextInt() {
        return Math.abs(super.nextInt());
    }
};
for (int i = 0; i < 100; i++) {
    System.out.println(random.nextInt());
}
```

**random** ist eine Instanz einer anonymen Klasse, die eine Unterklasse von **Random** ist.

# Beispiel: Anonyme Klasse (2)



```
@Override
public Iterator<T> iterator() {
    return new Iterator<T>() {

        private int index = data.length - 1;

        @Override
        public boolean hasNext() {
            return data != null && index >= 0;
        }

        @Override
        public T next() {
            T result = data[index];
            index--;
            return result;
        }
    };
}
```

## Beispiel: Anonyme Klasse (3)

```
public Iterator<T> randomIterator(final long seed) {  
    return new Iterator<T>() {  
  
        private Random random = new Random(seed);  
  
        @Override  
        public boolean hasNext() {  
            return data != null;  
        }  
  
        @Override  
        public T next() {  
            return data[random.nextInt(data.length)];  
        }  
    };  
}
```

# Initialisierungs-Anweisungen

```
public class InitExample {  
  
    private List<String> list = new ArrayList<>();  
  
    private static Set<Integer> set = new HashSet<>();  
  
}
```

## Initialisierungs-Anweisungen

- können aus einer einzigen Anweisung oder aus einem Block bestehen
- werden **vor** dem Konstruktor ausgeführt
- werden **nach** den Konstruktoren der Oberklassen ausgeführt
- können auf alle Elemente der Klasse zugreifen

## Initialisierungs-Blöcke

- können nicht-statisch sein (Instanz-Initialisierer)
- können statisch sein (statische Initialisierer)

```
public class InitExample {  
  
    private List<String> list;  
    {  
        list = new ArrayList<>();  
        list.add("a");  
    }  
  
    private static Set<Integer> set;  
    static {  
        set = new HashSet<>();  
        set.add(1);  
    }  
  
}
```

## Beispiel: Anonyme Klasse (4)

```
public Iterator<T> orderedIterator(Comparator<? super T> comparator) {  
    return new Iterator<T>() {  
  
        private Iterator<T> iterator;  
        { // initializer block  
            List<T> orderedData = Arrays.asList(data);  
            orderedData.sort(comparator);  
            iterator = orderedData.iterator();  
        }  
  
        @Override  
        public boolean hasNext() {  
            return iterator.hasNext();  
        }  
  
        @Override  
        public T next() {  
            return iterator.next();  
        }  
    };  
}
```





- **Lambda-Ausdrücke** sind keine anonymen Klassen, sondern **anonyme Methoden**
- Sie werden aber eingesetzt, um die Implementierung von funktionalen Interfaces durch einen einfachen, kompakten Ausdruck zu ersetzen
  - funktionale Interfaces sind Interfaces, die nur eine einzige nicht-statische und nicht-default Methode definieren
  - Instanzen von funktionalen Interfaces können direkt durch einen Lambda-Ausdruck ersetzt werden
  - dabei wird die Lambda-Funktion auf die eine Methode des Interfaces abgebildet
- Aufbau: `FunctionalInterface instance = (parameters) -> expression/block;`



- **Lambda-Ausdrücke** sind keine anonymen Klassen, sondern **anonyme Methoden**
- Sie werden aber eingesetzt, um die Implementierung von funktionalen Interfaces durch einen einfachen, kompakten Ausdruck zu ersetzen
  - funktionale Interfaces sind Interfaces, die nur eine einzige nicht-statische und nicht-default Methode definieren
  - Instanzen von funktionalen Interfaces können direkt durch einen Lambda-Ausdruck ersetzt werden
  - dabei wird die Lambda-Funktion auf die eine Methode des Interfaces abgebildet
- Aufbau: `FunctionalInterface instance = (parameters) -> expression/block;`  
eine beliebige Anzahl von Parametern



- **Lambda-Ausdrücke** sind keine anonymen Klassen, sondern **anonyme Methoden**
- Sie werden aber eingesetzt, um die Implementierung von funktionalen Interfaces durch einen einfachen, kompakten Ausdruck zu ersetzen
  - funktionale Interfaces sind Interfaces, die nur eine einzige nicht-statische und nicht-default Methode definieren
  - Instanzen von funktionalen Interfaces können direkt durch einen Lambda-Ausdruck ersetzt werden
  - dabei wird die Lambda-Funktion auf die eine Methode des Interfaces abgebildet
- **Aufbau:** `FunctionalInterface instance = (parameters) -> expression/block;`  
wird abgebildet auf



- **Lambda-Ausdrücke** sind keine anonymen Klassen, sondern **anonyme Methoden**
- Sie werden aber eingesetzt, um die Implementierung von funktionalen Interfaces durch einen einfachen, kompakten Ausdruck zu ersetzen
  - funktionale Interfaces sind Interfaces, die nur eine einzige nicht-statische und nicht-default Methode definieren
  - Instanzen von funktionalen Interfaces können direkt durch einen Lambda-Ausdruck ersetzt werden
  - dabei wird die Lambda-Funktion auf die eine Methode des Interfaces abgebildet
- **Aufbau:** `FunctionalInterface instance = (parameters) -> expression/block;`  
eine beliebige Anzahl von Ausdrücken

# Syntax-Beispiele: Lambda-Ausdrücke

```
None noneInst = () -> System.out.println("Lambda");  
noneInst.method(); // Lambda
```

```
One oneInst = (int x) -> x * x;  
int a = oneInst.method(5); // 25
```

```
Two twoInst = (x, y) -> {  
    Random random = new Random();  
    int result = 0;  
    for (int i = 0; i < x; i++) {  
        result += random.nextInt(y);  
    }  
    return result;  
};  
int a = twoInst.method(3, 8); // e.g., 10
```

```
interface None {  
    void method();  
}
```

```
interface One {  
    int method(int a);  
}
```

```
interface Two {  
    int method(int a, int b);  
}
```

Parametertypen müssen nicht angegeben werden.

Rückgabewerte werden (falls erforderlich) bei einem einzelnen Ausdruck automatisch zurückgegeben, bei Blöcken muss eine **return**-Anweisung verwendet werden.

# Beispiel: Lambda-Ausdruck

Collections
<u>+ sort(list: List&lt;T&gt;, comp: Comparator&lt;T&gt;)</u>

«interface» Comparator<T>
+ compare(o1: T, o2: T): int

```
List<Integer> list = new ArrayList<>();  
list.add(3);  
list.add(1);  
list.add(2);  
list.add(4);  
  
Collections.sort(list, (i1, i2) -> Integer.compare(i1, i2));  
System.out.println(list); // [1, 2, 3, 4]  
  
Collections.sort(list, (i1, i2) -> Integer.compare(i1, i2) * -1);  
System.out.println(list); // [4, 3, 2, 1]
```

## Beispiel: Lambda-Ausdruck (2)

```
class RegularComparator implements Comparator<Integer> {  
    @Override  
    public int compare(Integer i1, Integer i2) {  
        return Integer.compare(i1, i2);  
    }  
}
```

```
class InverseComparator implements Comparator<Integer> {  
    @Override  
    public int compare(Integer i1, Integer i2) {  
        return Integer.compare(i1, i2) * -1;  
    }  
}
```

```
Collections.sort(list, new RegularComparator());  
Collections.sort(list, new InverseComparator());
```

Reguläre Klassen

## Beispiel: Lambda-Ausdruck (2)

```
Collections.sort(list, new Comparator<Integer>() {  
    @Override  
    public int compare(Integer i1, Integer i2) {  
        return Integer.compare(i1, i2);  
    }  
});
```

```
Collections.sort(list, new Comparator<Integer>() {  
    @Override  
    public int compare(Integer i1, Integer i2) {  
        return Integer.compare(i1, i2) * -1;  
    }  
});
```

Anonyme Klassen



## Beispiel: Lambda-Ausdruck (2)

```
Collections.sort(list, (i1, i2) -> Integer.compare(i1, i2));  
Collections.sort(list, (i1, i2) -> Integer.compare(i1, i2) * -1);
```

Anonyme Methoden ( $\lambda$ )

# Lambdas in Java: Beispiel (3)

```
interface Calculable {  
    int calculate(int a);  
}
```

```
Calculable inverse = (x) -> x * -1;  
Calculable square = (x) -> x * x;  
System.out.println(inverse.calculate(3)); // -3  
System.out.println(square.calculate(3)); // 9
```

## Lambdas in Java: Beispiel (3)

```
private List<Integer> apply(Calculable c, List<Integer> list) {  
    List<Integer> result = new ArrayList<>();  
    for (int i : list) {  
        result.add(c.calculate(i));  
    }  
    return result;  
}
```

```
// [1, 2, 3, 4]  
list = apply(square, list);  
System.out.println(list); // [1, 4, 9, 16]  
  
list = apply((x) -> x + 3, list);  
System.out.println(list); // [4, 7, 12, 19]
```

# Iterieren mit Lambdas

- Das Interface **Iterable** wurde um die Methode **default void forEach(Consumer<T> c)** erweitert
- **Consumer** ist ein funktionales Interface, das nur die Methode **void accept(T t)** enthält
- Der Aufruf von **forEach** wendet die **accept**-Methode auf jedes Element an, über das iteriert wird

# Iterieren mit Lambdas: Beispiel

```
for (int i : list) {  
    System.out.println(i);  
}
```

```
list.forEach((i) -> System.out.println(i));
```

# Methoden-Referenzen

- In manchen Fällen macht ein Lambda-Ausdruck nichts anderes, als eine bestehende Methode aufzurufen (z.B. **Integer.compare()**)
- In diesen Fällen ist es möglich, direkt eine Referenz auf die Methode anzugeben statt eines Lambda-Ausdrucks

- auf eine statische Methode

```
list = apply(Math::abs, list);
```

- auf eine Methode eines Objekts

```
Random random = new Random();  
list = apply(random::nextInt, list);
```

- auf eine Methode eines beliebigen Objekts eines bestimmten Typs

```
Collections.sort(list, Integer::compareTo);  
Collections.sort(list, (a,b) -> a.compareTo(b));
```

- auf einen Konstruktor

```
ClassName::new
```

# Beispiel: Methoden-Referenzen

```
Collections.sort(list, (i1, i2) -> Integer.compare(i1, i2));  
Collections.sort(list, (i1, i2) -> Integer.compare(i1, i2) * -1);
```

```
Collections.sort(list, Integer::compare);  
Collections.sort(list, Integer::compare * -1);
```

The target type of this expression must be a functional interface

# Wann verwende ich was?

- **Verschachtelte Klasse**
  - enger semantischer oder technischer Zusammenhang
  - wird nur lokal verwendet
- **Statische Klasse**
  - keine Instanz der äußeren Klasse nötig
- **Member Klasse**
  - Instanz der äußeren Klasse nötig
- **Lokale Klasse**
  - wird nur in sehr engem Kontext verwendet
  - hat meist mehr als eine Methode
- **Anonyme Klasse**
  - wird nur einmal benötigt
  - hat meist mehr als eine Methode
- **Lambda Ausdruck**
  - wird nur einmal benötigt
  - funktionales Interface
- **Methoden Referenzen**
  - wie Lambda Ausdruck
  - ruft ausschließlich eine existierende Methode auf



# Streams

- **Streams** sind Folgen von Elementen, die sowohl sequentielle als auch parallele Verarbeitungsoperationen unterstützen
- Sie haben **nichts** mit **InputStream** bzw. **OutputStream** zu tun  
→ später
- Sie bieten Methoden zum Filtern, Verändern und Aggregieren der Elemente an
- Diese Methoden liefern wiederum jeweils einen Stream zurück, so dass auf dem Ergebnis direkt weitergearbeitet werden kann  
→ verkettete Methodenaufrufe

# Java Streams: Beispiel

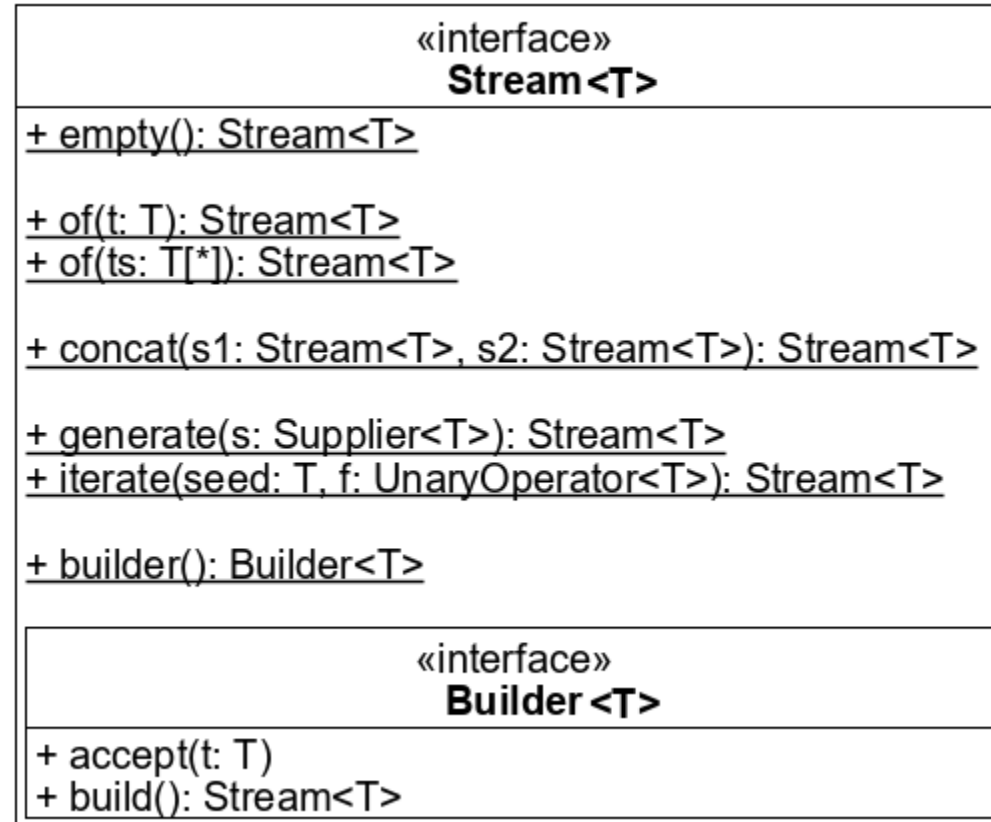
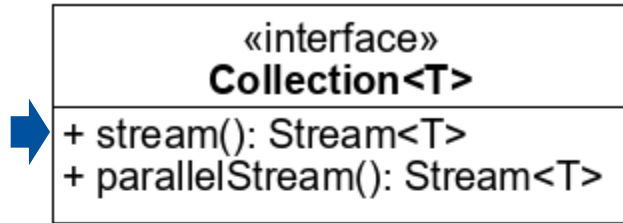
```
List<Integer> list = new ArrayList<>();  
list.add(4); list.add(7); list.add(12); list.add(19);  
// [4, 7, 12, 19]  
  
String text = list.stream()  
    .filter((i) -> i % 2 == 0)           // keep only even values [ 4, 12 ]  
    .map((i) -> i + 1)                  // increase each value by one [ 5, 13 ]  
    .map((i) -> Integer.toString(i))   // convert int values to String [ "5", "13" ]  
    .map((s) -> s += "#")               // append "#" to each value [ "5#", "13#" ]  
    .reduce((s, t) -> s + t)           // concatenate values to a single value "5#13#"  
    .get();                             // return this value  
  
System.out.println(text);               // 5#13#
```

# Erzeugen von Streams

«interface» <b>Collection&lt;T&gt;</b>
+ stream(): Stream<T> + parallelStream(): Stream<T>

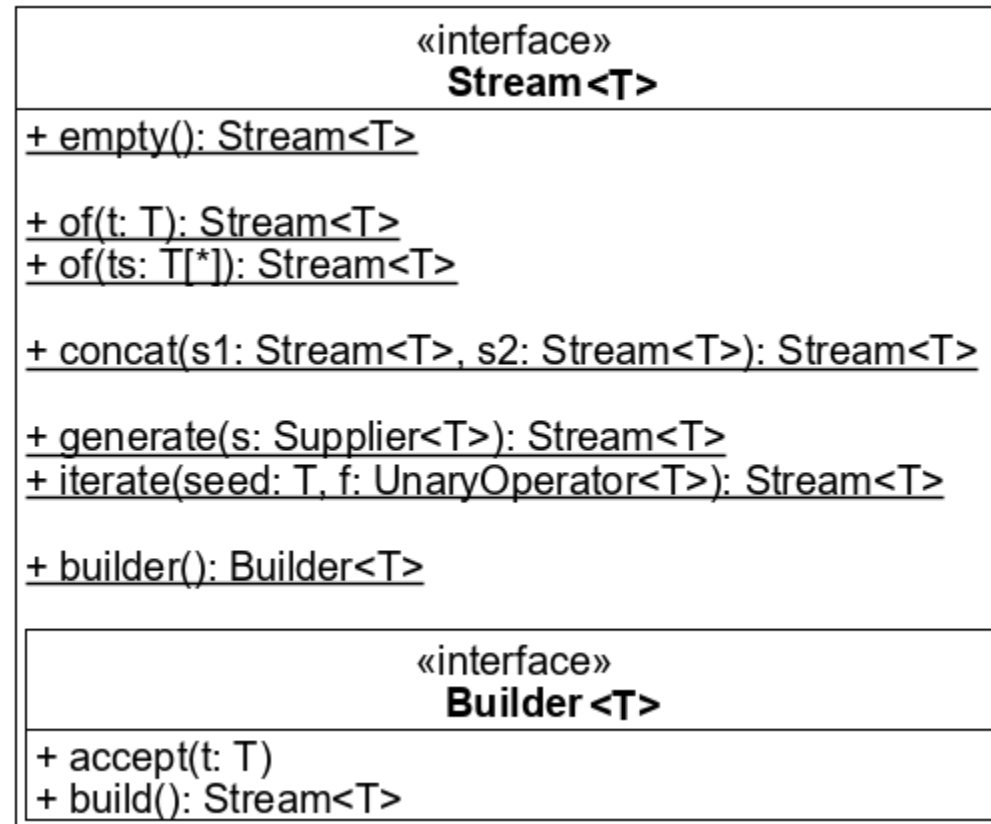
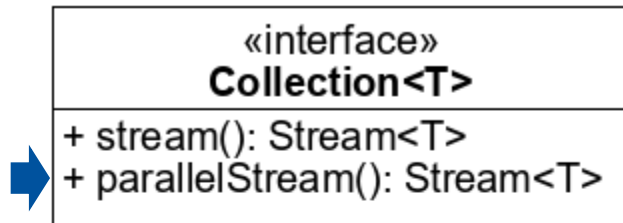
«interface» <b>Stream&lt;T&gt;</b>
+ <u>empty(): Stream&lt;T&gt;</u>  + <u>of(t: T): Stream&lt;T&gt;</u> + <u>of(ts: T[*]): Stream&lt;T&gt;</u>  + <u>concat(s1: Stream&lt;T&gt;, s2: Stream&lt;T&gt;): Stream&lt;T&gt;</u>  + <u>generate(s: Supplier&lt;T&gt;): Stream&lt;T&gt;</u> + <u>iterate(seed: T, f: UnaryOperator&lt;T&gt;): Stream&lt;T&gt;</u>  + <u>builder(): Builder&lt;T&gt;</u>
«interface» <b>Builder&lt;T&gt;</b>
+ accept(t: T) + build(): Stream<T>

# Erzeugen von Streams



erzeugt einen zu der Sammlung (Liste, Menge, ...) gehörigen Stream

# Erzeugen von Streams



erzeugt einen zu der Sammlung gehörigen Stream,  
dessen Methoden u.U. parallel ausgeführt werden

# Erzeugen von Streams

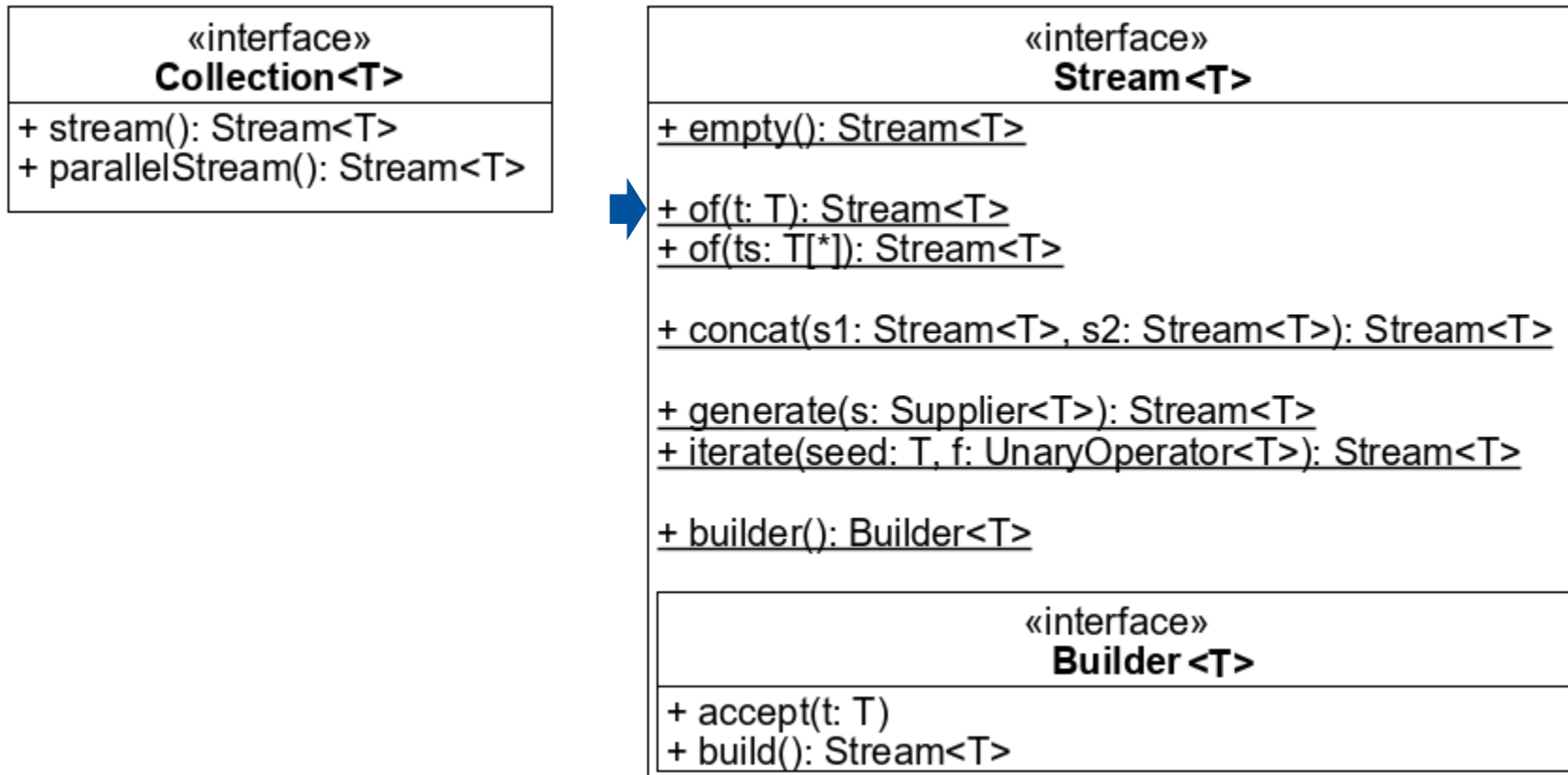
«interface» <b>Collection&lt;T&gt;</b>
+ stream(): Stream<T> + parallelStream(): Stream<T>



«interface» <b>Stream&lt;T&gt;</b>
+ <u>empty(): Stream&lt;T&gt;</u>  + <u>of(t: T): Stream&lt;T&gt;</u> + <u>of(ts: T[*]): Stream&lt;T&gt;</u>  + <u>concat(s1: Stream&lt;T&gt;, s2: Stream&lt;T&gt;): Stream&lt;T&gt;</u>  + <u>generate(s: Supplier&lt;T&gt;): Stream&lt;T&gt;</u> + <u>iterate(seed: T, f: UnaryOperator&lt;T&gt;): Stream&lt;T&gt;</u>  + <u>builder(): Builder&lt;T&gt;</u>
«interface» <b>Builder&lt;T&gt;</b>
+ accept(t: T) + build(): Stream<T>

erzeugt einen leeren Stream

# Erzeugen von Streams



erzeugt einen Stream mit einem einzigen Element



# Erzeugen von Streams

«interface» <b>Collection&lt;T&gt;</b>
+ stream(): Stream<T> + parallelStream(): Stream<T>



«interface» <b>Stream&lt;T&gt;</b>
+ <u>empty(): Stream&lt;T&gt;</u> + <u>of(t: T): Stream&lt;T&gt;</u> + <u>of(ts: T[*]): Stream&lt;T&gt;</u> + <u>concat(s1: Stream&lt;T&gt;, s2: Stream&lt;T&gt;): Stream&lt;T&gt;</u> + <u>generate(s: Supplier&lt;T&gt;): Stream&lt;T&gt;</u> + <u>iterate(seed: T, f: UnaryOperator&lt;T&gt;): Stream&lt;T&gt;</u> + <u>builder(): Builder&lt;T&gt;</u>
«interface» <b>Builder&lt;T&gt;</b>
+ accept(t: T) + build(): Stream<T>

erzeugt einen Stream mit allen Elementen des Arrays in der gegebenen Reihenfolge

# Erzeugen von Streams

«interface» <b>Collection&lt;T&gt;</b>
+ stream(): Stream<T> + parallelStream(): Stream<T>



«interface» <b>Stream&lt;T&gt;</b>
+ <u>empty(): Stream&lt;T&gt;</u>  + <u>of(t: T): Stream&lt;T&gt;</u> + <u>of(ts: T[*]): Stream&lt;T&gt;</u>  + <u>concat(s1: Stream&lt;T&gt;, s2: Stream&lt;T&gt;): Stream&lt;T&gt;</u>  + <u>generate(s: Supplier&lt;T&gt;): Stream&lt;T&gt;</u> + <u>iterate(seed: T, f: UnaryOperator&lt;T&gt;): Stream&lt;T&gt;</u>  + <u>builder(): Builder&lt;T&gt;</u>
«interface» <b>Builder&lt;T&gt;</b>
+ accept(t: T) + build(): Stream<T>

erzeugt einen Stream mit allen Elementen des ersten Streams gefolgt von allen Elementen des zweiten Streams

# Erzeugen von Streams

«interface» <b>Collection&lt;T&gt;</b>
+ stream(): Stream<T> + parallelStream(): Stream<T>



«interface» <b>Stream&lt;T&gt;</b>
+ <u>empty(): Stream&lt;T&gt;</u>  + <u>of(t: T): Stream&lt;T&gt;</u> + <u>of(ts: T[*]): Stream&lt;T&gt;</u>  + <u>concat(s1: Stream&lt;T&gt;, s2: Stream&lt;T&gt;): Stream&lt;T&gt;</u>  + <u>generate(s: Supplier&lt;T&gt;): Stream&lt;T&gt;</u> + <u>iterate(seed: T, f: UnaryOperator&lt;T&gt;): Stream&lt;T&gt;</u>  + <u>builder(): Builder&lt;T&gt;</u>
«interface» <b>Builder&lt;T&gt;</b>
+ accept(t: T) + build(): Stream<T>

erzeugt einen unendlichen, nicht-geordneten Stream mit Elementen, die der Supplier erzeugt

**Supplier<T>** enthält nur eine Methode **get(): T**, die Elemente in beliebiger Reihenfolge und ggf. auch mehrfach zurückgibt

# Erzeugen von Streams

«interface» <b>Collection&lt;T&gt;</b>
+ stream(): Stream<T> + parallelStream(): Stream<T>



«interface» <b>Stream&lt;T&gt;</b>
+ <u>empty(): Stream&lt;T&gt;</u>  + <u>of(t: T): Stream&lt;T&gt;</u> + <u>of(ts: T[*]): Stream&lt;T&gt;</u>  + <u>concat(s1: Stream&lt;T&gt;, s2: Stream&lt;T&gt;): Stream&lt;T&gt;</u>  + <u>generate(s: Supplier&lt;T&gt;): Stream&lt;T&gt;</u> + <u>iterate(seed: T, f: UnaryOperator&lt;T&gt;): Stream&lt;T&gt;</u>  + <u>builder(): Builder&lt;T&gt;</u>
«interface» <b>Builder&lt;T&gt;</b>
+ accept(t: T) + build(): Stream<T>

erzeugt einen unendlichen, geordneten Stream mit Elementen, die der UnaryOperator f basierend auf dem Initialwert seed erzeugt: seed, f(seed), f(f(seed)), f(f(f(seed))), ...  
**UnaryOperator<T>** enthält eine Methode **apply(t: T): T**, die eine Funktion auf ein gegebenes Element anwendet und das Ergebnis zurückgibt

# Erzeugen von Streams

«interface» <b>Collection&lt;T&gt;</b>
+ <u>stream(): Stream&lt;T&gt;</u> + <u>parallelStream(): Stream&lt;T&gt;</u>



«interface» <b>Stream&lt;T&gt;</b>
+ <u>empty(): Stream&lt;T&gt;</u>  + <u>of(t: T): Stream&lt;T&gt;</u> + <u>of(ts: T[*]): Stream&lt;T&gt;</u>  + <u>concat(s1: Stream&lt;T&gt;, s2: Stream&lt;T&gt;): Stream&lt;T&gt;</u>  + <u>generate(s: Supplier&lt;T&gt;): Stream&lt;T&gt;</u> + <u>iterate(seed: T, f: UnaryOperator&lt;T&gt;): Stream&lt;T&gt;</u>  + <u>builder(): Builder&lt;T&gt;</u>
«interface» <b>Builder&lt;T&gt;</b>
+ <u>accept(t: T)</u> + <u>build(): Stream&lt;T&gt;</u>

erzeugt einen **Stream.Builder**, der mit **accept(t: T)** eine Reihe von Elementen annimmt und anschließend mit **build()** den entsprechenden Stream dazu erzeugt

«interface»  
**Stream<T>**

```
+ distinct(): Stream<T>
+ filter(p: Predicate<T>): Stream<T>
+ limit(maxSize: long): Stream<T>
+ map(f: Function<T, R>): Stream<R>
+ sorted(): Stream<T>
+ sorted(c: Comparator<T>): Stream<T>
+ parallel(): Stream<T>
+ sequential(): Stream<T>

+ forEach(c: Consumer<T>)
+ reduce(f: BinaryOperator<T>): Optional<T>
+ toArray(): Object[*]
+ iterator(): Iterator<T>

+ findFirst(): Optional<T>
+ findAny(): Optional<T>
+ max(c: Comparator<T>): Optional<T>
+ min(c: Comparator<T>): Optional<T>
+ count(): long

+ allMatch(p: Predicate<T>): boolean
+ anyMatch(p: Predicate<T>): boolean
+ noneMatch(p: Predicate<T>): boolean
```


«interface»  
**Stream<T>**

➔

- + distinct(): Stream<T>
- + filter(p: Predicate<T>): Stream<T>
- + limit(maxSize: long): Stream<T>
- + map(f: Function<T, R>): Stream<R>
- + sorted(): Stream<T>
- + sorted(c: Comparator<T>): Stream<T>
- + parallel(): Stream<T>
- + sequential(): Stream<T>
- 
- + forEach(c: Consumer<T>)
- + reduce(f: BinaryOperator<T>): Optional<T>
- + toArray(): Object[\*]
- + iterator(): Iterator<T>
- 
- + findFirst(): Optional<T>
- + findAny(): Optional<T>
- + max(c: Comparator<T>): Optional<T>
- + min(c: Comparator<T>): Optional<T>
- + count(): long
- 
- + allMatch(p: Predicate<T>): boolean
- + anyMatch(p: Predicate<T>): boolean
- + noneMatch(p: Predicate<T>): boolean

erzeugt einen Stream, dessen Elemente eindeutig sind  
(bezüglich **equals()**)

«interface»  
**Stream<T>**




- + distinct(): Stream<T>
- + filter(p: Predicate<T>): Stream<T>
- + limit(maxSize: long): Stream<T>
- + map(f: Function<T, R>): Stream<R>
- + sorted(): Stream<T>
- + sorted(c: Comparator<T>): Stream<T>
- + parallel(): Stream<T>
- + sequential(): Stream<T>
  
- + forEach(c: Consumer<T>)
- + reduce(f: BinaryOperator<T>): Optional<T>
- + toArray(): Object[\*]
- + iterator(): Iterator<T>
  
- + findFirst(): Optional<T>
- + findAny(): Optional<T>
- + max(c: Comparator<T>): Optional<T>
- + min(c: Comparator<T>): Optional<T>
- + count(): long
  
- + allMatch(p: Predicate<T>): boolean
- + anyMatch(p: Predicate<T>): boolean
- + noneMatch(p: Predicate<T>): boolean

erzeugt einen Stream, der **nur** die Elemente enthält, die das Prädikat erfüllen

**Predicate<T>** definiert eine Methode **test(t: T): boolean**



«interface»  
**Stream<T>**



- + distinct(): Stream<T>
- + filter(p: Predicate<T>): Stream<T>
- + limit(maxSize: long): Stream<T>
- + map(f: Function<T, R>): Stream<R>
- + sorted(): Stream<T>
- + sorted(c: Comparator<T>): Stream<T>
- + parallel(): Stream<T>
- + sequential(): Stream<T>
- 
- + forEach(c: Consumer<T>)
- + reduce(f: BinaryOperator<T>): Optional<T>
- + toArray(): Object[\*]
- + iterator(): Iterator<T>
- 
- + findFirst(): Optional<T>
- + findAny(): Optional<T>
- + max(c: Comparator<T>): Optional<T>
- + min(c: Comparator<T>): Optional<T>
- + count(): long
- 
- + allMatch(p: Predicate<T>): boolean
- + anyMatch(p: Predicate<T>): boolean
- + noneMatch(p: Predicate<T>): boolean

erzeugt einen Stream, der höchstens **maxSize** Elemente enthält

# Stream-Methoden

«interface»  
**Stream<T>**

+ distinct(): Stream<T>  
+ filter(p: Predicate<T>): Stream<T>  
+ limit(maxSize: long): Stream<T>  
+ map(f: Function<T, R>): Stream<R>  
+ sorted(): Stream<T>  
+ sorted(c: Comparator<T>): Stream<T>  
+ parallel(): Stream<T>  
+ sequential(): Stream<T>  
  
+ forEach(c: Consumer<T>)  
+ reduce(f: BinaryOperator<T>): Optional<T>  
+ toArray(): Object[\*]  
+ iterator(): Iterator<T>  
  
+ findFirst(): Optional<T>  
+ findAny(): Optional<T>  
+ max(c: Comparator<T>): Optional<T>  
+ min(c: Comparator<T>): Optional<T>  
+ count(): long  
  
+ allMatch(p: Predicate<T>): boolean  
+ anyMatch(p: Predicate<T>): boolean  
+ noneMatch(p: Predicate<T>): boolean


erzeugt einen Stream, und wendet die Funktion **f** auf jedes Element an

**Function<T, R>** definiert eine Methode **apply(t: T): R**

Analog gibt es **mapToDouble**, **mapToInt** und **mapToLong**.

Diese Methoden geben jeweils einen **DoubleStream**, **IntStream** bzw. **LongStream** mit speziellen Aggregations-Methoden wie **average()** oder **sum()** zurück.


«interface»  
**Stream<T>**



- + distinct(): Stream<T>
- + filter(p: Predicate<T>): Stream<T>
- + limit(maxSize: long): Stream<T>
- + map(f: Function<T, R>): Stream<R>
- + sorted(): Stream<T>
- + sorted(c: Comparator<T>): Stream<T>
- + parallel(): Stream<T>
- + sequential(): Stream<T>
- 
- + forEach(c: Consumer<T>)
- + reduce(f: BinaryOperator<T>): Optional<T>
- + toArray(): Object[\*]
- + iterator(): Iterator<T>
- 
- + findFirst(): Optional<T>
- + findAny(): Optional<T>
- + max(c: Comparator<T>): Optional<T>
- + min(c: Comparator<T>): Optional<T>
- + count(): long
- 
- + allMatch(p: Predicate<T>): boolean
- + anyMatch(p: Predicate<T>): boolean
- + noneMatch(p: Predicate<T>): boolean

erzeugt einen Stream, der basierend auf der natürlichen Ordnung der Elemente sortiert ist


«interface»  
**Stream<T>**



- + distinct(): Stream<T>
- + filter(p: Predicate<T>): Stream<T>
- + limit(maxSize: long): Stream<T>
- + map(f: Function<T, R>): Stream<R>
- + sorted(): Stream<T>
- + sorted(c: Comparator<T>): Stream<T>
- + parallel(): Stream<T>
- + sequential(): Stream<T>
- 
- + forEach(c: Consumer<T>)
- + reduce(f: BinaryOperator<T>): Optional<T>
- + toArray(): Object[\*]
- + iterator(): Iterator<T>
- 
- + findFirst(): Optional<T>
- + findAny(): Optional<T>
- + max(c: Comparator<T>): Optional<T>
- + min(c: Comparator<T>): Optional<T>
- + count(): long
- 
- + allMatch(p: Predicate<T>): boolean
- + anyMatch(p: Predicate<T>): boolean
- + noneMatch(p: Predicate<T>): boolean

erzeugt einen Stream, der basierend auf dem gegebenen Comparator sortiert ist

«interface»  
**Stream<T>**




- + distinct(): Stream<T>
- + filter(p: Predicate<T>): Stream<T>
- + limit(maxSize: long): Stream<T>
- + map(f: Function<T, R>): Stream<R>
- + sorted(): Stream<T>
- + sorted(c: Comparator<T>): Stream<T>
- + parallel(): Stream<T>
- + sequential(): Stream<T>
- 
- + forEach(c: Consumer<T>)
- + reduce(f: BinaryOperator<T>): Optional<T>
- + toArray(): Object[\*]
- + iterator(): Iterator<T>
- 
- + findFirst(): Optional<T>
- + findAny(): Optional<T>
- + max(c: Comparator<T>): Optional<T>
- + min(c: Comparator<T>): Optional<T>
- + count(): long
- 
- + allMatch(p: Predicate<T>): boolean
- + anyMatch(p: Predicate<T>): boolean
- + noneMatch(p: Predicate<T>): boolean

erzeugt einen Stream, dessen Methoden u.U. parallel ausgeführt werden

# Stream-Methoden


«interface»  
**Stream<T>**



- + distinct(): Stream<T>
- + filter(p: Predicate<T>): Stream<T>
- + limit(maxSize: long): Stream<T>
- + map(f: Function<T, R>): Stream<R>
- + sorted(): Stream<T>
- + sorted(c: Comparator<T>): Stream<T>
- + parallel(): Stream<T>
- + sequential(): Stream<T>
- 
- + forEach(c: Consumer<T>)
- + reduce(f: BinaryOperator<T>): Optional<T>
- + toArray(): Object[\*]
- + iterator(): Iterator<T>
- 
- + findFirst(): Optional<T>
- + findAny(): Optional<T>
- + max(c: Comparator<T>): Optional<T>
- + min(c: Comparator<T>): Optional<T>
- + count(): long
- 
- + allMatch(p: Predicate<T>): boolean
- + anyMatch(p: Predicate<T>): boolean
- + noneMatch(p: Predicate<T>): boolean

erzeugt einen Stream, dessen Methoden immer sequentiell ausgeführt werden

«interface»  
**Stream<T>**

+ distinct(): Stream<T>  
+ filter(p: Predicate<T>): Stream<T>  
+ limit(maxSize: long): Stream<T>  
+ map(f: Function<T, R>): Stream<R>  
+ sorted(): Stream<T>  
+ sorted(c: Comparator<T>): Stream<T>  
+ parallel(): Stream<T>  
+ sequential(): Stream<T>  
 + forEach(c: Consumer<T>)  
+ reduce(f: BinaryOperator<T>): Optional<T>  
+ toArray(): Object[\*]  
+ iterator(): Iterator<T>  
  
+ findFirst(): Optional<T>  
+ findAny(): Optional<T>  
+ max(c: Comparator<T>): Optional<T>  
+ min(c: Comparator<T>): Optional<T>  
+ count(): long  
  
+ allMatch(p: Predicate<T>): boolean  
+ anyMatch(p: Predicate<T>): boolean  
+ noneMatch(p: Predicate<T>): boolean

wendet den gegebenen Consumer auf jedes Element des Streams an

**Consumer<T>** definiert eine Methode **accept(t: T)**

# Stream-Methoden

«interface»  
**Stream<T>**

+ distinct(): Stream<T>  
+ filter(p: Predicate<T>): Stream<T>  
+ limit(maxSize: long): Stream<T>  
+ map(f: Function<T, R>): Stream<R>  
+ sorted(): Stream<T>  
+ sorted(c: Comparator<T>): Stream<T>  
+ parallel(): Stream<T>  
+ sequential(): Stream<T>  
  
+ forEach(c: Consumer<T>)  
+ reduce(f: BinaryOperator<T>): Optional<T>  
+ toArray(): Object[\*]  
+ iterator(): Iterator<T>  
  
+ findFirst(): Optional<T>  
+ findAny(): Optional<T>  
+ max(c: Comparator<T>): Optional<T>  
+ min(c: Comparator<T>): Optional<T>  
+ count(): long  
  
+ allMatch(p: Predicate<T>): boolean  
+ anyMatch(p: Predicate<T>): boolean  
+ noneMatch(p: Predicate<T>): boolean

reduziert den Stream mithilfe des gegebenen BinaryOperator auf ein einziges Element

**BinaryOperator<T>** definiert eine Methode **apply(t1: T, t2: T): T**

**Optional<T>** ist ein Container für Werte, die möglicherweise **null** sind.

**Optional<T>** definiert eine Methode **isPresent(): boolean**, eine Methode **get(): T** und eine Methode **orElse(t: T): T**



«interface»  
**Stream<T>**

+ distinct(): Stream<T>  
+ filter(p: Predicate<T>): Stream<T>  
+ limit(maxSize: long): Stream<T>  
+ map(f: Function<T, R>): Stream<R>  
+ sorted(): Stream<T>  
+ sorted(c: Comparator<T>): Stream<T>  
+ parallel(): Stream<T>  
+ sequential(): Stream<T>  
  
+ forEach(c: Consumer<T>)  
+ reduce(f: BinaryOperator<T>): Optional<T>  
+ toArray(): Object[\*]  
+ iterator(): Iterator<T>  
  
+ findFirst(): Optional<T>  
+ findAny(): Optional<T>  
+ max(c: Comparator<T>): Optional<T>  
+ min(c: Comparator<T>): Optional<T>  
+ count(): long  
  
+ allMatch(p: Predicate<T>): boolean  
+ anyMatch(p: Predicate<T>): boolean  
+ noneMatch(p: Predicate<T>): boolean

konvertiert den Stream in ein Array

# Stream-Methoden

«interface»  
**Stream<T>**

+ distinct(): Stream<T>  
+ filter(p: Predicate<T>): Stream<T>  
+ limit(maxSize: long): Stream<T>  
+ map(f: Function<T, R>): Stream<R>  
+ sorted(): Stream<T>  
+ sorted(c: Comparator<T>): Stream<T>  
+ parallel(): Stream<T>  
+ sequential(): Stream<T>  
  
+ forEach(c: Consumer<T>)  
+ reduce(f: BinaryOperator<T>): Optional<T>  
+ toArray(): Object[\*]  
+ iterator(): Iterator<T>  
  
+ findFirst(): Optional<T>  
+ findAny(): Optional<T>  
+ max(c: Comparator<T>): Optional<T>  
+ min(c: Comparator<T>): Optional<T>  
+ count(): long  
  
+ allMatch(p: Predicate<T>): boolean  
+ anyMatch(p: Predicate<T>): boolean  
+ noneMatch(p: Predicate<T>): boolean

gibt einen Iterator für den Stream zurück

«interface»  
**Stream<T>**

+ distinct(): Stream<T>  
+ filter(p: Predicate<T>): Stream<T>  
+ limit(maxSize: long): Stream<T>  
+ map(f: Function<T, R>): Stream<R>  
+ sorted(): Stream<T>  
+ sorted(c: Comparator<T>): Stream<T>  
+ parallel(): Stream<T>  
+ sequential(): Stream<T>

+ forEach(c: Consumer<T>)  
+ reduce(f: BinaryOperator<T>): Optional<T>  
+ toArray(): Object[\*]  
+ iterator(): Iterator<T>

➔ + findFirst(): Optional<T>  
+ findAny(): Optional<T>  
+ max(c: Comparator<T>): Optional<T>  
+ min(c: Comparator<T>): Optional<T>  
+ count(): long

+ allMatch(p: Predicate<T>): boolean  
+ anyMatch(p: Predicate<T>): boolean  
+ noneMatch(p: Predicate<T>): boolean

gibt das erste Element des Streams zurück

# Stream-Methoden

«interface»  
**Stream<T>**

+ distinct(): Stream<T>  
+ filter(p: Predicate<T>): Stream<T>  
+ limit(maxSize: long): Stream<T>  
+ map(f: Function<T, R>): Stream<R>  
+ sorted(): Stream<T>  
+ sorted(c: Comparator<T>): Stream<T>  
+ parallel(): Stream<T>  
+ sequential(): Stream<T>  
  
+ forEach(c: Consumer<T>)  
+ reduce(f: BinaryOperator<T>): Optional<T>  
+ toArray(): Object[\*]  
+ iterator(): Iterator<T>  
  
+ findFirst(): Optional<T>  
+ findAny(): Optional<T>  
+ max(c: Comparator<T>): Optional<T>  
+ min(c: Comparator<T>): Optional<T>  
+ count(): long  
  
+ allMatch(p: Predicate<T>): boolean  
+ anyMatch(p: Predicate<T>): boolean  
+ noneMatch(p: Predicate<T>): boolean

gibt ein beliebiges Element des Streams zurück

«interface»  
**Stream<T>**

+ distinct(): Stream<T>  
+ filter(p: Predicate<T>): Stream<T>  
+ limit(maxSize: long): Stream<T>  
+ map(f: Function<T, R>): Stream<R>  
+ sorted(): Stream<T>  
+ sorted(c: Comparator<T>): Stream<T>  
+ parallel(): Stream<T>  
+ sequential(): Stream<T>  
  
+ forEach(c: Consumer<T>)  
+ reduce(f: BinaryOperator<T>): Optional<T>  
+ toArray(): Object[\*]  
+ iterator(): Iterator<T>  
  
+ findFirst(): Optional<T>  
+ findAny(): Optional<T>  
➔ + max(c: Comparator<T>): Optional<T>  
+ min(c: Comparator<T>): Optional<T>  
+ count(): long  
  
+ allMatch(p: Predicate<T>): boolean  
+ anyMatch(p: Predicate<T>): boolean  
+ noneMatch(p: Predicate<T>): boolean

gibt das maximale Element bezüglich des gegebenen Comparators zurück

«interface»  
**Stream<T>**

+ distinct(): Stream<T>  
+ filter(p: Predicate<T>): Stream<T>  
+ limit(maxSize: long): Stream<T>  
+ map(f: Function<T, R>): Stream<R>  
+ sorted(): Stream<T>  
+ sorted(c: Comparator<T>): Stream<T>  
+ parallel(): Stream<T>  
+ sequential(): Stream<T>  
  
+ forEach(c: Consumer<T>)  
+ reduce(f: BinaryOperator<T>): Optional<T>  
+ toArray(): Object[\*]  
+ iterator(): Iterator<T>  
  
+ findFirst(): Optional<T>  
+ findAny(): Optional<T>  
+ max(c: Comparator<T>): Optional<T>  
+ min(c: Comparator<T>): Optional<T>  
+ count(): long  
  
+ allMatch(p: Predicate<T>): boolean  
+ anyMatch(p: Predicate<T>): boolean  
+ noneMatch(p: Predicate<T>): boolean

gibt das minimale Element bezüglich des gegebenen Comparators zurück

# Stream-Methoden

«interface»  
**Stream<T>**

+ distinct(): Stream<T>  
+ filter(p: Predicate<T>): Stream<T>  
+ limit(maxSize: long): Stream<T>  
+ map(f: Function<T, R>): Stream<R>  
+ sorted(): Stream<T>  
+ sorted(c: Comparator<T>): Stream<T>  
+ parallel(): Stream<T>  
+ sequential(): Stream<T>  
  
+ forEach(c: Consumer<T>)  
+ reduce(f: BinaryOperator<T>): Optional<T>  
+ toArray(): Object[\*]  
+ iterator(): Iterator<T>  
  
+ findFirst(): Optional<T>  
+ findAny(): Optional<T>  
+ max(c: Comparator<T>): Optional<T>  
+ min(c: Comparator<T>): Optional<T>  
+ count(): long  
  
+ allMatch(p: Predicate<T>): boolean  
+ anyMatch(p: Predicate<T>): boolean  
+ noneMatch(p: Predicate<T>): boolean

gibt die Anzahl der Elemente des Streams zurück

«interface»  
**Stream<T>**

- + distinct(): Stream<T>
- + filter(p: Predicate<T>): Stream<T>
- + limit(maxSize: long): Stream<T>
- + map(f: Function<T, R>): Stream<R>
- + sorted(): Stream<T>
- + sorted(c: Comparator<T>): Stream<T>
- + parallel(): Stream<T>
- + sequential(): Stream<T>
- + forEach(c: Consumer<T>)
- + reduce(f: BinaryOperator<T>): Optional<T>
- + toArray(): Object[\*]
- + iterator(): Iterator<T>
- + findFirst(): Optional<T>
- + findAny(): Optional<T>
- + max(c: Comparator<T>): Optional<T>
- + min(c: Comparator<T>): Optional<T>
- + count(): long
- + allMatch(p: Predicate<T>): boolean
- + anyMatch(p: Predicate<T>): boolean
- + noneMatch(p: Predicate<T>): boolean

prüft, ob das gegebene Prädikat für alle Elemente des Streams gilt



«interface»  
**Stream<T>**

- + distinct(): Stream<T>
- + filter(p: Predicate<T>): Stream<T>
- + limit(maxSize: long): Stream<T>
- + map(f: Function<T, R>): Stream<R>
- + sorted(): Stream<T>
- + sorted(c: Comparator<T>): Stream<T>
- + parallel(): Stream<T>
- + sequential(): Stream<T>
- + forEach(c: Consumer<T>)
- + reduce(f: BinaryOperator<T>): Optional<T>
- + toArray(): Object[\*]
- + iterator(): Iterator<T>
- + findFirst(): Optional<T>
- + findAny(): Optional<T>
- + max(c: Comparator<T>): Optional<T>
- + min(c: Comparator<T>): Optional<T>
- + count(): long
- + allMatch(p: Predicate<T>): boolean
- + anyMatch(p: Predicate<T>): boolean
- + noneMatch(p: Predicate<T>): boolean

prüft, ob das gegebene Prädikat für mindestens ein Element des Streams gilt

«interface»  
**Stream<T>**

+ distinct(): Stream<T>  
+ filter(p: Predicate<T>): Stream<T>  
+ limit(maxSize: long): Stream<T>  
+ map(f: Function<T, R>): Stream<R>  
+ sorted(): Stream<T>  
+ sorted(c: Comparator<T>): Stream<T>  
+ parallel(): Stream<T>  
+ sequential(): Stream<T>  
  
+ forEach(c: Consumer<T>)  
+ reduce(f: BinaryOperator<T>): Optional<T>  
+ toArray(): Object[\*]  
+ iterator(): Iterator<T>  
  
+ findFirst(): Optional<T>  
+ findAny(): Optional<T>  
+ max(c: Comparator<T>): Optional<T>  
+ min(c: Comparator<T>): Optional<T>  
+ count(): long  
  
+ allMatch(p: Predicate<T>): boolean  
+ anyMatch(p: Predicate<T>): boolean  
+ noneMatch(p: Predicate<T>): boolean

prüft, ob das gegebene Prädikat für kein Element des Streams gilt

# Java Streams: Beispiel (Wiederholung)

```
List<Integer> list = new ArrayList<>();  
list.add(4); list.add(7); list.add(12); list.add(19);  
// [4, 7, 12, 19]  
  
String text = list.stream()  
    .filter((i) -> i % 2 == 0)           // keep only even values [ 4, 12 ]  
    .map((i) -> i + 1)                  // increase each value by one [ 5, 13 ]  
    .map((i) -> Integer.toString(i))    // convert int values to String [ "5", "13" ]  
    .map((s) -> s += "#")               // append "#" to each value [ "5#", "13#" ]  
    .reduce((s, t) -> s + t)            // concatenate values to a single value "5#13#"  
    .get();                             // return this value  
  
System.out.println(text);               // 5#13#
```

# Java Streams: Beispiel (Wiederholung)

```
List<Integer> list = new ArrayList<>();  
list.add(4); list.add(7); list.add(12); list.add(19);  
// [4, 7, 12, 19]  
  
String text = list.stream()  
    .filter((i) -> i % 2 == 0)           // Stream<Integer> -> Stream<Integer>  
    .map((i) -> i + 1)                   // Stream<Integer> -> Stream<Integer>  
    .map((i) -> Integer.toString(i))     // Stream<Integer> -> Stream<String>  
    .map((s) -> s += "#")                 // Stream<String> -> Stream<String>  
    .reduce((s, t) -> s + t)             // Stream<String> -> Optional<String>  
    .get();                              // Optional<String> -> String  
  
System.out.println(text);
```

# Java Streams: Beispiel (Wiederholung)

```
List<Integer> list = new ArrayList<>();  
list.add(4); list.add(7); list.add(12); list.add(19);  
// [4, 7, 12, 19]  
  
String text = list.stream()  
    .filter((i) -> i % 2 == 0)           // p: T -> boolean  
    .map((i) -> i + 1)                  // f: T -> R  
    .map((i) -> Integer.toString(i))    // f: T -> R  
    .map((s) -> s += "#")               // f: T -> R  
    .reduce((s, t) -> s + t)           // f: (T, T) -> T  
    .get();  
  
System.out.println(text);
```

## Java Streams: Beispiel (2)

```
class Rectangle {  
    int x, y, width, height;  
}
```

```
List<Rectangle> list = new ArrayList<>();  
Random rnd = new Random(0L);  
for (int i = 0; i < 1000000; i++) {  
    list.add(new Rectangle(rnd.nextInt(), rnd.nextInt(),  
                           rnd.nextInt(10) + 1, rnd.nextInt(10) + 1));  
}  
  
int area = list.stream()  
    .filter((r) -> r.getX() >= 0 && r.getY() >= 0)  
    .distinct()  
    .mapToInt((r) -> r.getWidth() * r.getHeight())  
    .sum();  
System.out.println(area); // 7562792  
  
System.out.println("Contains squares: " +  
    list.stream().anyMatch((r) -> r.getWidth() == r.getHeight())); // true
```

# Java Streams: Beispiel (3)

```
private static Stream<Rectangle> createStream(long size) {  
    return Stream.iterate(new Rectangle(0, 0, 1, 1), (r) -> {  
        Random rnd = new Random((long) (r.getX() + r.getY() + r.getWidth() + r.getHeight()));  
        return new Rectangle(rnd.nextInt(), rnd.nextInt(),  
                               rnd.nextInt(10) + 1, rnd.nextInt(10) + 1);  
    }).limit(size);  
}
```

```
int area = createStream(STREAM_SIZE)  
    .map((r) -> r.getWidth() * r.getHeight())  
    .reduce((i1, i2) -> i1 + i2)  
    .get();  
System.out.println(area); // 3020190
```

```
createStream(STREAM_SIZE)  
    .filter((r) -> r.getWidth() * r.getHeight() < 5)  
    .filter((r) -> r.getX() % 42 == 0 && r.getY() % 42 == 0)  
    .forEach((r) -> System.out.print("."));  
// ....
```

- Lokale Klassen
- Anonyme Klassen
- Lambda-Ausdrücke
- Streams