



# 06-Functional-1

Objektorientierte Programmierung | Matthias Tichy



Software Engineering  
Programming Languages



universität  
**uulm**

# Lernziele

- Anwendung von *Lambda Expressions*
- Nutzung des *Stream-Collection-API*
- Verstehen der allgemeinen Funktions-Typen

# Lambda Expressions

„A lambda expression is like a method: it provides a list of formal parameters and a body — an expression or block — expressed in terms of those parameters.“

```
import java.util.function.BinaryOperator;

class LambdaLove {
    public static void main(String[] args) {
        BinaryOperator<Integer> mult =
            (Integer a, Integer b) -> {
                return a * b;
            };
        System.out.println(mult.apply(21, 2));
    }
}
```

Parameter

Lambda Ausdruck

Parameter(s) -> Body

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, Gavin Bierman:  
The Java® Language Specification - Java SE 21 Edition – 2024-05-25 - §15.27 Lambda Expressions  
<https://docs.oracle.com/javase/specs/jls/se21/html/jls-15.html#jls-15.27>

# Lambda Expressions

„Evaluation of a lambda expression produces an **instance** of a functional interface. Lambda expression evaluation does *not* cause the execution of the expression's body; instead, this may occur at a later time **when an appropriate method of the functional interface is invoked.**“

```
import java.util.function.BinaryOperator;
```

```
class LambdaLove {  
    public static void main(String[] args) {  
        BinaryOperator<Integer> mult =  
            (Integer a, Integer b) -> {  
                return a * b;  
            };  
  
        System.out.println(mult.apply(21, 2));  
    }  
}
```

Instanz des  
BinaryOperator  
Interfaces

Aufruf der passenden Methode

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, Gavin Bierman:  
The Java® Language Specification - Java SE 21 Edition – 2024-05-25 - §15.27 Lambda Expressions  
<https://docs.oracle.com/javase/specs/jls/se21/html/jls-15.html#jls-15.27>

# Functional Interface

„A *functional interface* is an interface that [...] has just one abstract method (aside from the methods of Object), and thus represents a single function contract.“

```
@FunctionalInterface
public interface BinaryOperator<T> {
    T apply(T a, T b);
}
```

(Auch wenn man `BinaryOperator` so implementieren könnte, ist dies nicht die tatsächliche Implementation)

Implementation von  
T apply(T a, T b)

```
BinaryOperator<Integer> mult =
    (Integer a, Integer b) -> {
        return a * b;
    };

```

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, Gavin Bierman:  
The Java® Language Specification - Java SE 21 Edition – 2024-05-25 - §9.8 Functional Interfaces  
<https://docs.oracle.com/javase/specs/jls/se21/html/jls-9.html#jls-9.8>

# Nützliche existierende Interfaces

Alle in `java.util.function`

@FunctionalInterface

```
public interface Function<T, R> {  
    R apply(T t);  
}
```

@FunctionalInterface

```
public interface BiFunction<T, U, R> {  
    R apply(T t, U u);  
}
```

@FunctionalInterface

```
public interface Supplier<T> {  
    T get();  
}
```

@FunctionalInterface

```
public interface Consumer<T> {  
    void accept(T t);  
}
```

Kurzschreibweise

```
Function<Integer, String> f =  
    i -> Integer.toString(i);  
System.out.println(f.apply(3));
```

```
BiFunction<Integer, Double, Double> f =  
    (a, b) -> a * b;  
System.out.println(f.apply(3, 2.2));
```

```
Supplier<Integer> f = () -> 42;  
System.out.println(f.get());
```

```
Consumer<String> f =  
    s -> System.out.print(s);  
f.accept("Hey");
```

# Collection Stream-API

# Stream<T>

„A sequence of elements supporting sequential and parallel aggregate operations.“

```
import java.util.List;
```

```
class StreamDream {
```

```
    public static int sumOfSmalls(List<Integer> values) {
```

```
        return values.stream() |  Stream<Integer>
```

```
        .filter(x -> x < 3)
```

```
        .reduce(0, (a, b) -> a + b);
```

 Predicate<Integer>  
boolean test(T t)


```
    }
```

```
    public static void main(String[] args) {
```

```
        System.out.println(sumOfSmalls(List.of(1, 3, 2)));
```

```
    }
```

```
}
```

 BinaryOperator<Integer>

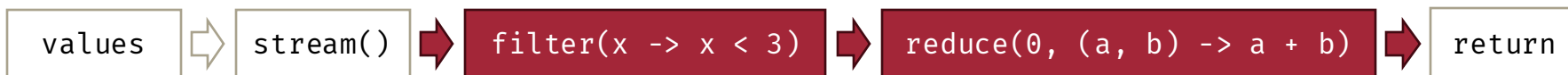
<https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/stream/Stream.html>



# Unterschied Stream vs. Collection

- keine Datenstruktur → Berechnungspipeline für Elemente einer Quelldatenstruktur (datenflussorientierte Verarbeitung)

```
public static int sumOfSmalls(List<Integer> values) {  
    return values.stream()  
        .filter(x -> x < 3)  
        .reduce(0, (a, b) -> a + b);  
}
```



- Builder-Pattern

<https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/stream/package-summary.html>

# Unterschied Stream vs. Collection

- keine Datenstruktur → Berechnungspipeline für Elemente einer Quelldatenstruktur (datenflussorientierte Verarbeitung)
- Funktional / Seiteneffektfrei: z.B. ein Filter erzeugt einen neuen Stream nur mit übrig gebliebenen Elementen
- Typen von Operationen:
  - Stream-produzierende Operationen
  - Werteerzeugende bzw. Seiteneffektproduzierende Operationen
- Potenziell Lazy: stream-prod. Operationen potenziell lazy
- Jedes Element eines Streams wird nur einmal verarbeitet

<https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/stream/package-summary.html>

# Terminale Operationen vs. Intermediate Operationen

- Intermediate Operationen
  - erzeugen neue Streams aus Streams
  - sind lazy
- Terminal Operationen:
  - Produzieren ein Ergebnis direkt oder über Seiteneffekte
  - „Beenden“ den Stream
  - Nicht lazy, sondern eager

```
public static int sumOfSmalls(List<Integer> values) {  
    return values.stream()  
        .filter(x -> x < 3) |  
        .reduce(0, (a, b) -> a + b);  
}
```

intermediate

terminal

## Murmelgruppe 1

Zählen Sie aus einer gegebenen Collection die Elemente, die mit sich selbst multipliziert kleiner 100 sind.

```
import java.util.*;
import java.util.function.*;

class Main {
    public static void main(String[] args) {
        var list = List.of(1, 4, -2, -10, -11, 2, 30, 22, 10, 9);

        var sum = list.stream() /* ... TODO */

        System.out.println("Die Anzahl ist " + sum);
    }
}
```

Verwenden Sie die  
Dokumentation!

# Lazyness

- „Nur das tun, was unbedingt notwendig ist und zum spätestmöglichen Zeitpunkt“:
  - Terminal Operationen starten die komplette Berechnung
  - Insbesondere: statt n-mal die gesamte Liste zu verarbeiten und neue Listen zu generieren
    - jedes Element nur einmal durch die Pipeline verarbeiten lassen

```
public static void streaming(List<String> values) {  
    var stream = values.stream()  
        .map(String::toUpperCase)  
        .filter(x -> x.length() > 3)  
        .skip(3)  
        .toList();  
    System.out.println(stream);  
}
```

Methoden-Referenz als Kurzform

Stream ohne die ersten 3 Elemente

# Stateless vs. Stateful Operationen

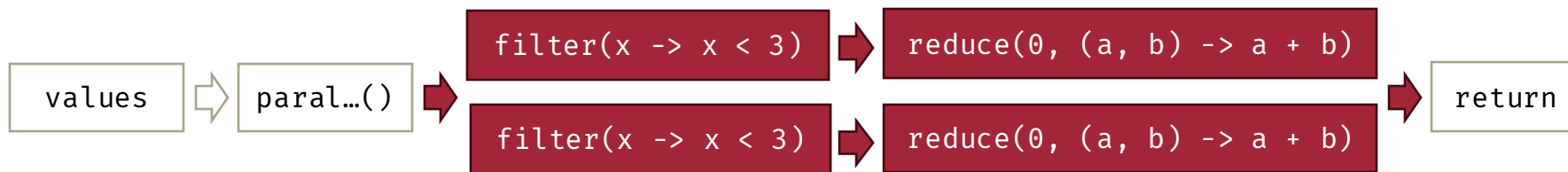
- Stateless:
  - Speichern keinen Zustand, u.a. in Bezug auf vorherige Elemente
  - Jedes Element kann einzeln verarbeitet werden
  - z.B. filter / map
- Stateful:
  - Verarbeitung benötigt Informationen über vorherige Elemente
  - z.B. distinct / sorted
- Pipelines rein mit Stateless Operationen:
  - in einem einzelnen Durchlauf einen Stream verarbeiten
  - Potential für Parallelität

# Parallele Streams

- Parallel Streams verarbeiten die Pipeline nebenläufig  
→ später nochmal Thema im Bereich Threads

```
public static int sumOfSmalls(List<Integer> values) {  
    return values.parallelStream()  
        .filter(x -> x < 3)  
        .reduce(0, (a, b) -> a + b);  
}
```

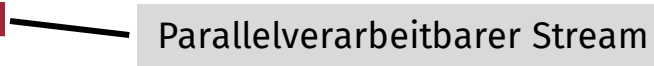
Parallelverarbeitbarer Stream



# Parallele Streams

- Parallel Streams verarbeiten die Pipeline nebenläufig  
→ später nochmal Thema im Bereich Threads

```
public static int sumOfSmalls(List<Integer> values) {  
    return values.parallelStream()  
        .filter(x -> x < 3)  
        .reduce(0, (a, b) -> a + b);  
}
```



Parallelverarbeitbarer Stream

- Nutzerfunktionen (z.B. in Lambda Expressions) müssen *stateless* und *non-interfering* sein, um korrektes Verhalten zu garantieren



# Non-Interfering

- Daten Quellen dürfen (in den meisten Fällen) nicht durch Code in den Pipelines geändert werden.
- **Interfering:**

```
public static void changingASourceIsAskingForTrouble() {  
    List<Integer> values = new CopyOnWriteArrayList(  
        Arrays.asList(new Integer[] {1,2,3,4}));  
    Function<Integer, Integer> crazyMapper = (x) -> {  
        if (x == 3) values.add(4);  
        return x;  
    };  
    values.stream()  
        .map(crazyMapper)  
        .filter(x -> x > 3)  
        .forEach(System.out::println);  
}
```

# Seiteneffekte

- Seiteneffekte in Funktionsargumenten für Streamoperationen:
  - Verletzen Statelessness
  - Gefahren für Thread-Safety
- → Seiteneffekte können zu falschen Ergebnissen führen
- Es gibt keine Garantien, dass:
  - Seiteneffekte Auswirkung auf andere Threads haben
  - Verschiedene Operationen auf dem selben Element im selben Stream im selben Thread ausgeführt werden
  - Funktionsargumente überhaupt ausgeführt werden → wenn das Ergebnis unabhängig (unter Annahme von Statelessness) von dem übergebenen Funktionsargument ist

# Seiteneffekte

- Seiteneffekte können zu falschen Ergebnissen führen

```
public static void sideEffectsIsAskingForTrouble() {  
    final var counter = new IntegerContainer(0);  
    IntUnaryOperator crazyMapper = (x) -> {  
        counter.value+=1;  
        return x;  
    };  
    IntStream.range(1, 100000).parallel()  
        .map(crazyMapper)  
        .filter(x -> x > 3)  
        .forEach(System.out::println);  
    System.out.println("Counter is " + counter.value);  
}
```

```
class IntegerContainer {  
    int value;  
    IntegerContainer(int value)  
    {this.value = value;}  
}
```

# Seiteneffekte

- Seiteneffekte können zu unerwarteten Ergebnissen führen

```
List<String> l = Arrays.asList("A", "B", "C", "D");  
long count = l.stream().peek(System.out::println).count();
```

# Ordnung

- Es gibt Streams ohne ein Ordnung der Elemente oder mit einer Ordnung
- Ungeordnete Streams → keine Garantie über die Reihenfolge der Ausführung / Ergebnisse

`map(x -> x * 2)`

- Für Collections ohne Ordnung mit den Inhalten 1,2,3 sind alle Permutationen als Ergebnis erlaubt.

# Reduktion (aka fold)

- Reduziert einen Stream auf ein einzelnes Ergebnis
- Wiederholte Anwendung eines Kombinerers

```
public static int filteredReduceSum(List<Integer> values) {  
    return values.  
        .stream()  
        .filter(x -> x < 3)  
        .reduce(0, (a, b) -> a + b);  
}
```

```
public static int filteredReduceSum(List<Integer> values) {  
    BinaryOperator<Integer> reducer = (a, b) -> a + b;  
    return values.  
        .stream()  
        .filter(x -> x < 3)  
        .reduce(0, reducer);  
}
```

# Reduktion (aka fold)

- Generelle Form:

```
public static int generalReduceSum(List<Integer> values) {  
    BiFunction<Integer, Integer, Integer> accumulator = (a, b) -> a + b;  
    BinaryOperator<Integer> combiner = (r1, r2) -> r1 + r2;  
  
    return values.stream().reduce(0, accumulator, combiner);  
}
```

```
public static int generalReduceSumMethod(List<Integer> values) {  
    BiFunction<Integer, Integer, Integer> accumulator = (a, b) -> a + b;  
  
    return values.stream().reduce(0, accumulator, Integer::sum);  
}
```

Konkatenieren sie die Strings aus der Liste

```
public class Main
{
    public static void main(String[] args) {
        var strings = Arrays.asList(new String[]{"one", "two", "three"});

        var allStrings = // TODO

        System.out.println("Das Ergebnis ist " + allStrings);
    }
}
```



# Mutierende Reduktion

- Eine mutierende Reduktion erzeugt die Reduktion durch Änderung eines Containers

```
public static String concatStringsViaStringBuffer(List<String> strings) {  
    return strings  
        .stream()  
        .collect(StringBuffer::new,  
                  StringBuffer::append,  
                  StringBuffer::append)  
        .toString();  
}
```

```
public static String concatStringsViaStringCopies(List<String> strings) {  
    return strings  
        .stream()  
        .reduce("", String::concat);  
}
```

# Mutierende Reduktion

## ■ Obacht:

```
public static int collectSum(List<Integer> values) {  
    Supplier<Integer> supplier = () -> 0;  
    BiConsumer<Integer, Integer> accumulator = (r, x) -> r += x;  
    BiConsumer<Integer, Integer> combiner = (r1, r2) -> r1 += r2;  
  
    return values.stream().collect(supplier, accumulator, combiner);  
}
```

```
class IntegerContainer {  
    int value;  
    IntegerContainer(int value)  
    {this.value = value;}  
}  
  
public static int collectSumContainer(List<Integer> values) {  
    Supplier<IntegerContainer> supplier = () -> new IntegerContainer(0);  
    BiConsumer<IntegerContainer, Integer> accumulator = (r, x) -> r.value += x;  
    BiConsumer<IntegerContainer, IntegerContainer> combiner = (r1, r2) -> r1.value += r2.value;  
    return values.stream().collect(supplier, accumulator, combiner).value;  
}
```

# Typen / Interfaces

# Stream<T>

## ■ Auswahl von Methoden:

```
Stream<T> filter(Predicate<? super T> predicate);  
Stream<T> distinct();  
Stream<T> sorted();
```

```
<R> Stream<R> map(Function<? super T, ? extends R> mapper);
```

```
T reduce(T identity, BinaryOperator<T> accumulator);  
Optional<T> min(Comparator<? super T> comparator);
```

```
boolean anyMatch(Predicate<? super T> predicate);  
void forEach(Consumer<? super T> action);
```

<https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/util/stream/Stream.html>

# Predicate<T>

Prüft Element

## ■ Auswahl von Methoden:

@FunctionalInterface

**public interface** Predicate<T> {

**boolean** test(T t);

—— Die Methode wird durch die Lambda Expression implementiert:

**default** Predicate<T> and(Predicate<? **super** T> other) {  
    Objects.requireNonNull(other);  
    **return** (t) -> test(t) && other.test(t);  
}

filter(x -> x < 3)

**default** Predicate<T> negate() {  
    **return** (t) -> !test(t);  
}

<https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/util/function/Predicate.html>

# Stream<T>

„A sequence of elements supporting sequential and parallel aggregate operations.“

```
public static int filteredReduceSum(List<Integer> values) {  
    return values.  
        stream().  
        filter(x -> x < 3).  
        reduce(0, (a, b) -> a + b);  
}
```

The diagram illustrates the functional programming model of Java Streams. It shows three interfaces: `Stream<T>`, `Predicate<T>`, and `BinaryOperator<T>`. `Stream<T>` is represented by a horizontal line with a vertical tick mark. `Predicate<T>` is represented by a horizontal line with a vertical tick mark. `BinaryOperator<T>` is represented by a horizontal line with a vertical tick mark. The `filter` method in the code is annotated with a red box around the lambda expression `x -> x < 3`, indicating it uses a `Predicate<T>`. The `reduce` method is annotated with a red box around the lambda expression `(a, b) -> a + b`, indicating it uses a `BinaryOperator<T>`.

<https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/util/stream/Stream.html>

# BinaryOperator<T>

2 Operanden, 1 Operator, ein Rückgabetyp – alle vom Typ T

## ■ Auswahl von Methoden:

T als Typargument

```
@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T,T,T>

    default ttttstatic <T> BinaryOperator<T> minBy(Comparator<? super T>
comparator)
    {
        Objects.requireNonNull(comparator);
        return (a, b) -> comparator.compare(a, b) <= 0 ? a : b;
    }
}
```

...

Standardimplementierung für kleineren Wert von zwei

<https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/util/function/BinaryOperator.html>

# Stream<T>

„A sequence of elements supporting sequential and parallel aggregate operations.“

```
public static int filteredReduceSum(List<Integer> values) {  
    return values.  
        stream().  
        filter(x -> x < 3)  
        reduce(0, (a, b) -> a + b);  
}
```

The diagram illustrates the functional components of the Java Stream API. It features three gray rectangular boxes: 'Stream<T>' at the top, 'Predicate<T>' on the right, and 'BinaryOperator<T>' at the bottom. A horizontal line connects 'Stream<T>' to 'Predicate<T>', with a vertical red line segment positioned above it. Another horizontal line connects 'Stream<T>' to 'BinaryOperator<T>', with a vertical red line segment positioned above it. A third horizontal line connects 'BinaryOperator<T>' to 'Predicate<T>', with a vertical red line segment positioned to its left. These lines represent the functional interfaces used in the code snippet above.

<https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/util/stream/Stream.html>



# BiFunction<T,U,R>

2 Operanden, 1 Operator, ein Rückgabetyp – unterschiedliche Typen

## ■ Auswahl von Methoden:

```
@FunctionalInterface
```

```
public interface BiFunction<T, U, R> {
```

```
    R apply(T t, U u);
```



Funktionale Method – durch Lambda Expression implementiert

```
    default <V> BiFunction<T, U, V> andThen(Function<? super R, ? extends V> after) {  
        Objects.requireNonNull(after);  
        return (T t, U u) -> after.apply(apply(t, u));  
    }
```

```
}
```

Funktionskomposition

<https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/util/function/BiFunction.html>

# Function<T,R>

1 Operand, 1 Rückgabetyt – unterschiedliche Typen

## ■ Auswahl von Methoden:

@FunctionalInterface

**public interface** Function<T, R> {

R apply(T t);



Funktionale Method – durch Lambda Expression implementiert

**default** <V> Function<T, V> andThen(Function<? **super** R, ? **extends** V> after) {  
Objects.requireNonNull(after);  
**return** (T t) -> after.apply(apply(t));  
}

**static** <T> Function<T, T> identity() {  
**return** t -> t;  
}

Funktionskomposition

<https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/util/function/BiFunction.html>

# Consumer<T>

1 Operand, kein Rückgabetyt – „Consumer is expected to operate via side-effects.“

## ■ Auswahl von Methoden:

```
@FunctionalInterface
```

```
public interface Consumer<T> {
```

```
void accept(T t);
```



Funktionale Method – durch Lambda Expression implementiert

```
default Consumer<T> andThen(Consumer<? super T> after) {
```

```
    Objects.requireNonNull(after);
```

```
    return (T t) -> { accept(t); after.accept(t); };
```

```
}
```

```
...
```

Funktionskomposition

<https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/util/function/Consumer.html>

## Murmelgruppe 3

Java Playground auf  
<https://www.online-java.com/8nYZNomIke>

Erzeugen Sie eine Liste aus der Ausgangsliste, die alle Strings mit mindestens 4 Zeichen einmalig in Kombination mit deren Länge enthält.

```
public class Main
{
    var strings = Arrays.asList(new String[]{"one", "two", "three", "three",
"four"});

    var result = // TODO

    System.out.println("Das Ergebnis ist " + result);
    // Das Ergebnis ist „three“,5 und „four“, 4
}
```

# Lösungsvorschlag Murmelgruppe 3

## Nutzung von Records

```
public static void main(String args[]) {  
    record StringLengthTuple(String s, int length) {  
    }  
  
    var strings = Arrays.asList(  
        new String[] { "one", "two", "three", "three", "four" });  
  
    var result = strings  
        .stream()  
        .distinct()  
        .filter(s -> s.length() >= 4)  
        .map(s -> new StringLengthTuple(s, s.length()))  
        .toList();  
  
    System.out.println("Das Ergebnis ist " + result);  
}
```

record mit 2 Attributen:  
s: String  
length: int

# Lernziele

- Anwendung von Lambda-Expressions
- Nutzung des stream-Collection-API
- Verstehen der allgemeinen Funktions-Typen