Programmierung 2

Lambda-Ausdrücke und funktionale Programmierung

Überblick

- Programmierparadigmen
- Lambda-Ausdrücke statt anonymen inneren Klassen
- funktionaler Ersatz einiger objektorientierter Muster
- Stream-Processing

Programmier-paradigmen

Imperative Programmierung

- Programm: Liste von Instruktionen, die nacheinander ausgeführt werden
- Zugriff auf Variablen im Hauptspeicher, die von den Instruktionen geändert werden können
- so arbeiten die herkömmlichen Rechnerarchitekturen (und die JVM)
- spezielle Paradigmen der imperativen Programmierung:
  - strukturiert
  - prozedural
  - objekt-orientiert
- Zustand: Belegung aller globalen Variablen (bei OO: zusätzlich Instanzvariablen aller Objekte)

Deklarative Programmierung

Spezifikation eines Sachverhalts mit den Mitteln einer deklarativen Programmiersprache

- funktionale Sprachen
- logische Sprachen (z.B. Prolog)
- (funktional-logische Sprachen)
- Mengen-orientierte Abfragesprachen (z.B. SQL)
- Constraint-orientierte Sprachen

Lambda-Ausdrücke

Das Problem

- Innere Klassen erlauben die Übergabe von Methoden an andere Methoden
- Übergabe erfolgt über den Umweg einer Klasse und eines Objektes
- Syntax ist umständlich und verwirrend
- Wieso kann man nicht einfach direkt eine Methode übergeben?

Higher Order Functions

Folie mit Anmerkungen

Funktionen höherer Ordnung haben Funktionen als Eingangsparameter oder geben Funktionen als Ergebnis zurück.

```
public interface Funktion {
    public int anwenden(int a, int b);
}
public class Berechnung {
    public static int berechne(int input1, int input2, Funktion funktion) {
        return funktion.anwenden(input1, input2);
    }
}
var ergebnis = Berechnung.berechne(5, 7, new Funktion() {
    @Override
    public int anwenden(int a, int b) {
        return a + b;
    }
}
);
```

anonyme innere Klasse

Higher Order Functions

Funktionen höherer Ordnung haben Funktionen als Eingangsparameter oder geben Funktionen als Ergebnis zurück.

```
public interface Funktion {
    public int anwenden(int a, int b);
}
public class Berechnung {
    public static int berechne(int input1, int input2, Funktion funktion) {
        return funktion.anwenden(input1, input2);
    }
}
var ergebnis = Berechnung.berechne(5, 7, (x, y) -> x + y);

Funktion add = (x, y) -> x + y;
Funktion sub = (x, y) -> x - y;
var ergebnis2 = add.anwenden(5, 7);
var ergebnis3 = sub.anwenden(10, 8);
```

Folie mit Anmerkungen

Functional Interface

- Grundlage ist Interface mit einer einzigen abstrakten Methode
  - **SAM-Interface (single abstract method interface)**
  - **funktionales Interface (“functional interface”)**
- Annotation `@FunctionalInterface` verhindert Hinzufügen weiterer Methoden
- Compiler setzt Lambda in eine Implementierung des Interfaces um

```
@FunctionalInterface
public interface Aktion {
    public void run(String param);
}
```

Folie mit Anmerkungen

Syntax von Lambda-Ausdrücken

- Lambda implementiert die (einzige) Methode des funktionalen Interfaces
- Welches Interface implementiert wird, ergibt sich aus dem Kontext
  - Typ der Variable
  - Typ des Parameters
- Aufbau des Lambdas
  1. Parameter der Methode in Klammern (mit oder ohne Typ), durch Komma getrennt. Bei nur einem Parameter kann die Klammer entfallen
  2. Ein Pfeil `->`
  3. Rumpf der Methode. Bei nur einem Statement, kein Block { } nötig
- `return` ist nicht erforderlich, wenn nur ein Ausdruck im Lambda vorkommt

Folie mit Anmerkungen

Typ-Inferenz und Lambda-Ausdrücke

Der Java-Compiler muss wissen, zu welchem Interface ein Lambda gehört. Diese Information leitet er aus dem Kontext ab, in dem das Lambda vorkommt.

- Wird es einer Variable zugewiesen, ergibt sich der Typ des Interfaces aus dem Typ der Variable.

  `Aktion a = s -> System.out.println(s);` (`var` funktioniert hier nicht)

- Wird das Lambda an eine Methode übergeben, ergibt sich das funktionale Interface aus dem Typ des Parameters der Methode.

```
public static void executor(Aktion a) {
    a.run("Hallo");
}
public static void main(String[] args) {
    executor(s -> System.out.println(s));
}
```

Syntax von Lambda-Ausdrücken möglichen Varianten

Folie mit Anmerkungen

- **ohne Typen:**
  ```
  (a, b) -> a + b
  ```
- **mit Typen:**
  ```
  (int a, int b) -> a + b
  ```
- **mit Block:**
  ```
  (a, b) -> { int s = a + b; return s; }
  ```
- **ohne Klammer:**
  ```
  a -> a * a
  ```
- **mit Klammer:**
  ```
  (a) -> a * a
  ```
- **ohne Argumente:**
  ```
  () -> { System.out.println("Hallo"); }
  ```

Folie mit Anmerkungen

Vorgefertigte Interfaces 1/2

Im Paket `java.util.function` gibt es bereits eine Reihe von vorgefertigten funktionalen Interfaces

- **`BiConsumer<T,U>`**
  Nimmt zwei Parameter vom Typ T und U und gibt kein Ergebnis zurück
- **`BiFunction<T,U,R>`**
  Nimmt zwei Parameter vom Typ T und U und gibt ein Ergebnis vom Typ R zurück
- **`BinaryOperator<T>`**
  Nimmt zwei Parameter vom Typ T und gibt ein Ergebnis vom Typ T zurück
- **`Consumer<T>`**
  Nimmt einen Parameter vom Typ T und gibt kein Ergebnis zurück

Folie mit Anmerkungen

Vorgefertigte Interfaces 1/2

Im Paket `java.util.function` gibt es bereits eine Reihe von vorgefertigten funktionalen Interfaces

- **`Function<T,R>`**
  Nimmt einen Parameter vom Typ T und gibt ein Ergebnis vom Typ R zurück
- **`Predicate<T>`**
  Nimmt einen Parameter vom Typ T und gibt einen Wahrheitswert zurück
- **`Supplier<T>`**
  Nimmt keinen Parameter und gibt ein Ergebnis vom Typ T zurück
- ...

Beispiel Autoboxing-Interfaces

```
import java.util.function.IntBinaryOperator;
public class Rechner {
    public static int berechne(int input1, int input2,
                               IntBinaryOperator fun) {
        return fun.applyAsInt(input1, input2);
    }

    public static void main(String... args) {
        IntBinaryOperator sub = (a, b) -> a - b;
        IntBinaryOperator add = (a, b) -> a + b;

        System.out.println(berechne(5, 3, sub));
        System.out.println(berechne(1, 7, add));
    }
}
```

Beispiel mit generischen Interfaces

```
import java.util.function.BiFunction;
public class Rechner {
    public static int berechne(int input1, int input2,
                               BiFunction<Integer, Integer, Integer> fun) {
        return fun.apply(input1, input2);
    }

    public static void main(String... args) {
        BiFunction<Integer, Integer, Integer> sub = (a, b) -> a - b;
        BiFunction<Integer, Integer, Integer> add = (a, b) -> a + b;

        System.out.println(berechne(5, 3, sub));
        System.out.println(berechne(1, 7, add));
    }
}
```

Folie mit Anmerkungen

Closure

- Lambda hat bei der Deklaration Zugriff auf die umgebenden Variablen
- Werte sind später beim Aufruf des Lambdas vorhanden
- Variablen dürfen nach Erzeugung des Lambdas nicht mehr verändert werden (am besten **`final`** machen) (muss *“effectively final”* sein)
- Lambdas mit Zugriff auf externe Variablen nennt man *capturing lambdas*

```
final var minusEins = -1;
Comparator<String> cmp = (a, b) -> a.compareTo(b) * minusEins;
```

Folie mit Anmerkungen

Lambda-Ausdrücke als Rückgabewert

Methoden können Lambda-Ausdrücke zurückgeben:

```
public class Factory {
    public Comparator<String> createLexiStringComparator() {
        return (a, b) -> a.compareTo(b);
    }
}
```

Folie mit Anmerkungen

Scoping bei Lambda-Ausdrücken

- Lambda-Ausdrücke sind im Wesentlichen eine verkürzte Schreibweise für anonyme innere Klassen
- Wichtiger Unterschied: **`this`** bezieht sich nicht auf das Objekt des Lambda-Ausdrucks, sondern auf das **umgebende Objekt**

```
public class Scoping {
    public String toString() {
        return "Scoping";
    }
    public Runnable createRunner() {
        return () -> System.out.println(this.toString());
    }
    public static void main(String... args) {
        new Scoping().createRunner().run();
    }
}
```

Folie mit Anmerkungen

Scoping bei Lambda-Ausdrücken

- Lambda-Ausdrücke sind im Wesentlichen eine verkürzte Schreibweise für anonyme innere Klassen
- Wichtiger Unterschied: **`this`** bezieht sich nicht auf das Objekt des Lambda-Ausdrucks, sondern auf das **umgebende Objekt**

```
public class Scoping {
    public String toString() {
        return "Scoping";
    }
    public Runnable createRunner() {
        return new Runnable() {
            public void run() {System.out.println(this.toString());}
        };
    }
    public static void main(String... args) {
        new Scoping().createRunner().run();
    }
}
```

Entwurfsmuster aus der Funktionalen Programmierung

Folie mit Anmerkungen

Funktionale Programmierung

- funktionale Sprache
  - fortwährende Anwendung von Funktionen
  - kein globaler Zustand (Variablen/Objekte im Heap)
- z.B. “provides the tools to **avoid mutable state**, provides **functions as first-class** objects, and **emphasizes recursive iteration** instead of side-effect based looping”
  (http://clojure.org/about/functional_programming)
- Funktionen sind “First Class Citizens”, ermöglichen Funktionen höherer Ordnung:
  - Funktionen können als Argument übergeben werden
  - Funktionen können als Resultat zurückgegeben werden
- aus Praktikabilitätsgründen:
  - anonyme Funktionen,
  - Variablen Funktionen als Wert zuweisen können

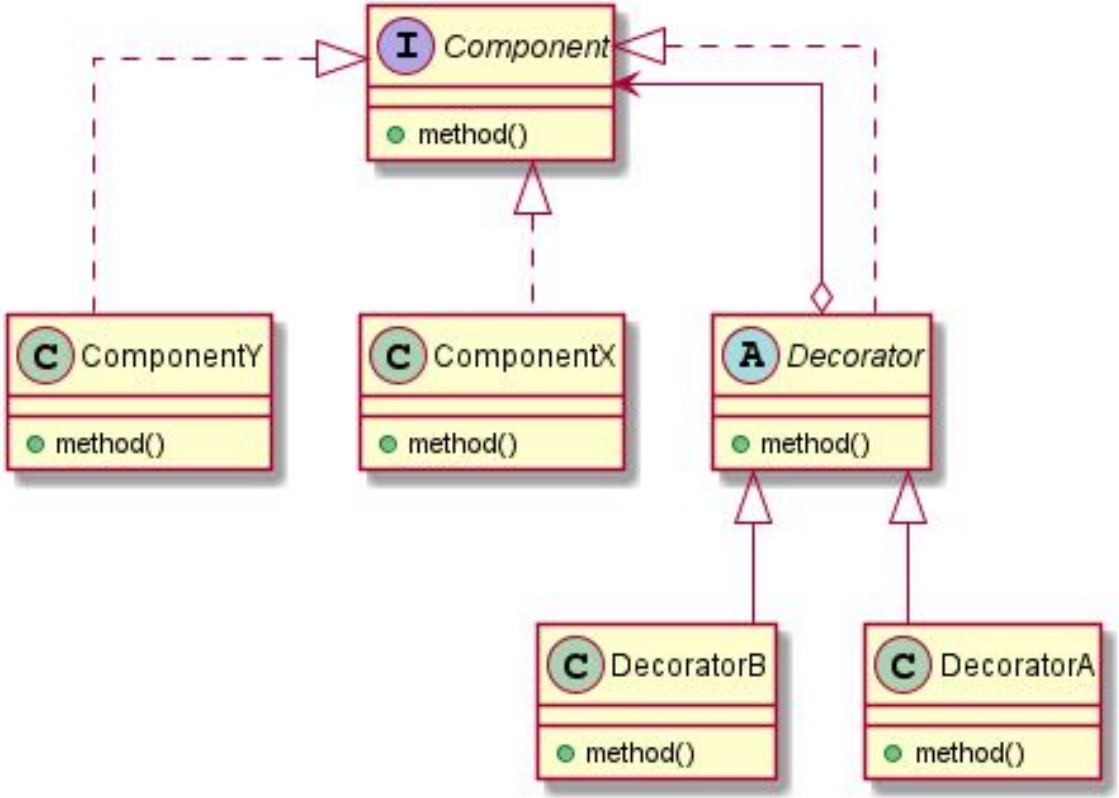
Folie mit Anmerkungen

Decorator Pattern

Das Decorator Pattern ist ein gutes Beispiel dafür, wie Prinzipien aus der funktionalen Programmierung das Software Design ändern können.

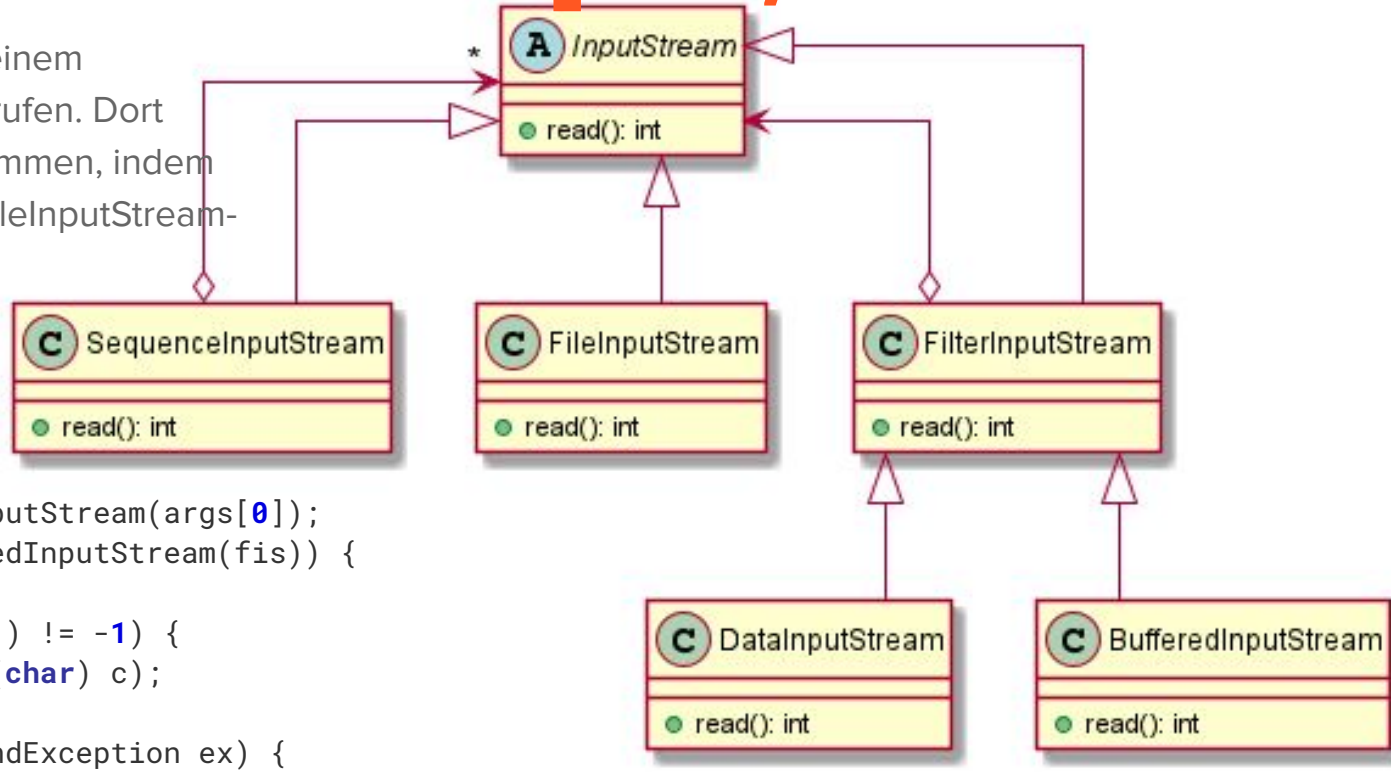
Ziel der Anwendung des Decorator Patterns ist es, das “**Single-Responsibility-Prinzip**” umzusetzen. Funktionalitäten und ihre Implementierung sollten isoliert werden, um die Weiterentwicklung zu erleichtern und in einer konkreten Umsetzung keine unnötigen Features zu haben, die nicht verwendet werden.

Decorator Pattern: generell



Decorator Pattern: Beispiel I/O-Streams

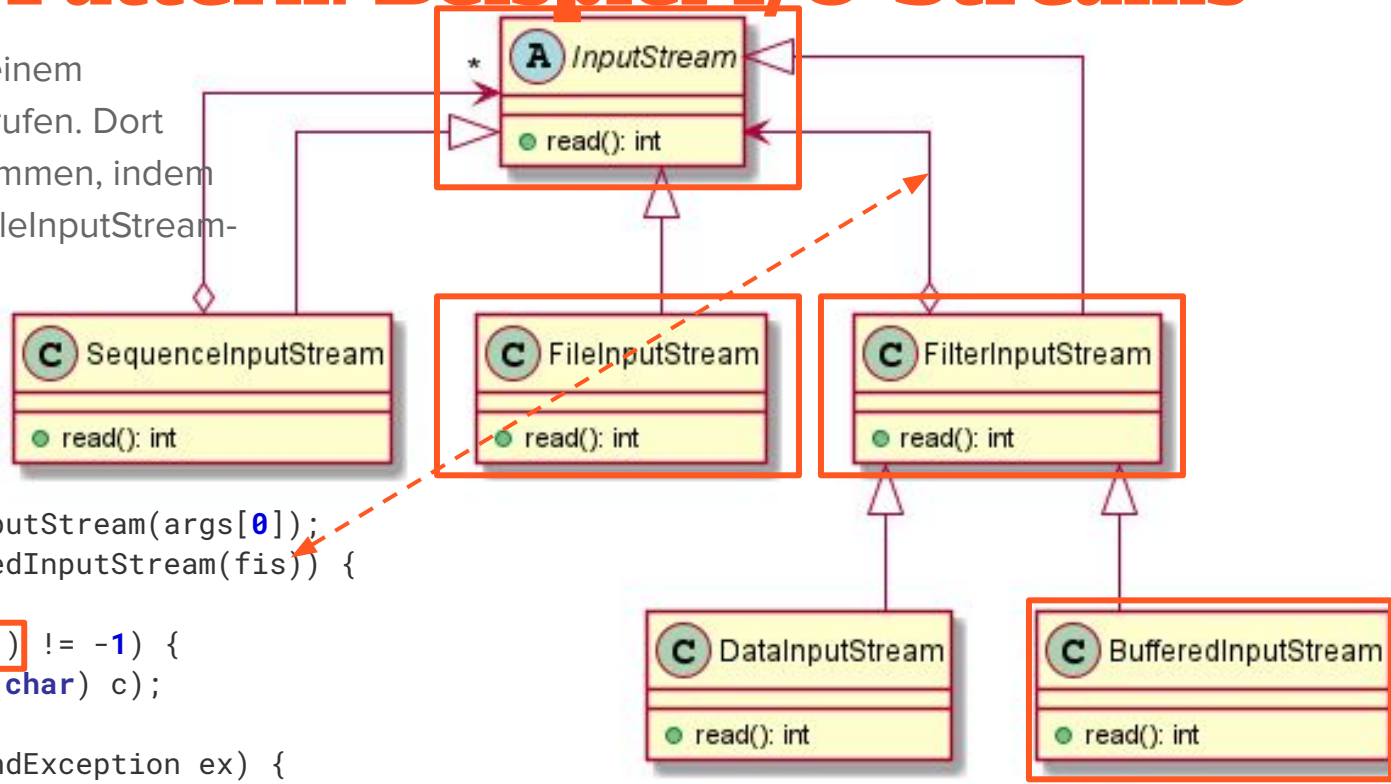
Die Methode read wird an einem BufferedInputStream aufgerufen. Dort wird die Pufferung vorgenommen, indem read an dem verknüpften FileInputStream-Objekt aufgerufen wird.




```
try (var fis = new FileInputStream(args[0]);
     var bis = new BufferedInputStream(fis)) {
    int c;
    while ((c = bis.read()) != -1) {
        System.out.print((char) c);
    }
} catch (final FileNotFoundException ex) {
    System.err.println("Datei nicht gefunden: " + ex.getMessage());
} catch (final IOException ex) {
    System.err.println("I/O-Fehler: " + ex.getMessage());
}
```

Decorator Pattern: Beispiel I/O-Streams

Die Methode read wird an einem BufferedInputStream aufgerufen. Dort wird die Pufferung vorgenommen, indem read an dem verknüpften FileInputStream-Objekt aufgerufen wird.




```
try (var fis = new FileInputStream(args[0]);
     var bis = new BufferedInputStream(fis)) {
    int c;
    while ((c = bis.read()) != -1) {
        System.out.print((char) c);
    }
} catch (final FileNotFoundException ex) {
    System.err.println("Datei nicht gefunden: " + ex.getMessage());
} catch (final IOException ex) {
    System.err.println("I/O-Fehler: " + ex.getMessage());
}
```

Function Builder

Allgemein: Eine neue Funktion erzeugen und zurückgeben.

bspw.: eine Funktion, die zwei gegebene Funktionen nacheinander ausführt:

```
public interface Funktion {
    void apply();
}
public class FunctionBuilder {
    public static Funktion mkFun(Funktion f1, Funktion f2) {
        return () -> {
            f1.apply();
            f2.apply();
        };
    }
}
```

Function-Builder als Ersatz für Decorator-Pattern

Es geht wie beim Decorator-Pattern darum, Funktionalität zu kapseln. Hier aber in Form von Funktionen:

Angenommen eine Logging-Funktionalität soll bei jedem Methoden-Aufruf hinzugefügt werden, kann die ursprüngliche Funktion durch eine erweiterte Funktion ersetzt werden. Dafür wird eine neue Funktion erzeugt, die das tut, was ursprünglich vorgesehen war, aber zusätzlich loggt.

Diese Funktion kann dann durch weitere Funktionen ergänzt werden, wie z.B. Zugriffkontrolle.

Function-Builder als Ersatz für Decorator-Pattern

Folie mit Anmerkungen

```
public class LoggingAspect {
    public static Funktion addLog(Funktion f) {
        return () -> {
            System.out.println("Funktionsaufruf: " + f);
            f.apply();
        };
    }
}

public class AccessAspect {
    public static Funktion addAccess(Predicate<Object> p, Funktion f) {
        return () -> {
            if (p.test(null)) {
                f.apply();
            }
        };
    }
}
```

Filter, Map, ForEach, Reduce auf Streams

Wesentliches Muster in der funktionalen Programmierung

- Transformation eines Stroms von Daten durch Verkettung von
  - Filter: nur passende Daten weiterleiten
  - Map: Transformation eines jeden Elements des Stroms nach einer festen Regel
  - Reduce: Aggregation von Elementen des Stroms
- Filter, Map und Reduce können als Funktionen höherer Ordnung umgesetzt werden
  - Parametrierung mit Filter-, Transformations- und Aggregierungsfunktionen
- Funktionsanwendungen unabhängig voneinander => Parallelisierungspotenzial
- Komplexe Verarbeitung bei “Big Data”
  - Anwendung von Filter-Map-Reduce bei Batch-Processing: “Lambda-Architektur”
  - z.B. Hadoop

Interface Predicate<T>

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
```

Da `Predicate` ein *functional Interface* ist, kann es in Lambda-Ausdrücken verwendet werden:

```
String[] str = //...
int anzahl = count(str, (s -> (s.length() == 3)));
```

```
public static <T> int count(T[] array, Predicate<T> pred) {
    var count = 0;
    for(var t : array) {
        if(pred.test(t))
            count++;
    }
    return count;
}
```

Interface Function<T,R>

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

Da `Function` ein *functional Interface* ist, kann es in Lambda-Ausdrücken verwendet werden:

```
String[] str = //...
Object[] res = transfer(str, s -> s.length());
```

```
public static <T,R> R[] transfer(T[] array, Function<T,R> func) {
    var res = (R[]) new Object[array.length];
    for(var i=0; i < array.length; i++ ) {
        res[i] = func.apply(array[i]);
    }
    return res;
}
```

Interface Consumer<T>

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

Da `Function` ein *functional Interface* ist, kann es in Lambda-Ausdrücken verwendet werden:

```
String[] str = //...
transferAndConsume(str, s -> s.length(), i -> System.out.println(i));
```

```
public static <T,R> void transferAndConsume(T[] array,Function<T,R> func, Consumer<R> consumer) {
    for(var i=0; i < array.length; i++ ) {
        var r = func.apply(array[i]);
        consumer.accept(r);
    }
}
```

Predicate, Function, Consumer

- auf einen `Stream<T>` kann man wiederholt transformative Funktionen anwenden
  - `Predicate<T>`
  - `Function<T, X>`
  - `Consumer<T>`
- Pseudocode-Notation: `foreach e in` Einagbe-`Stream<T>`:
  - ***filter*** mit Predicate<T> p: Wenn `p.test(e)`, dann `e` in Ausgabe-`Stream<T>`
  - ***map*** mit `Function<T, X>` `f`: Ausgabe-`Stream<X>` ist `f.apply(e)`
  - ***forEach*** mit `Consumer<T>` `c`: es gibt keinen Ausgabe-Stream, sondern `c.accept(e)` wird ausgeführt
  - ***reduce*** mit Bi`Function<T, T, T>` `f` und `Akkumulator<T>` a: a := f.apply(e,a)

Predicate, Function, Consumer

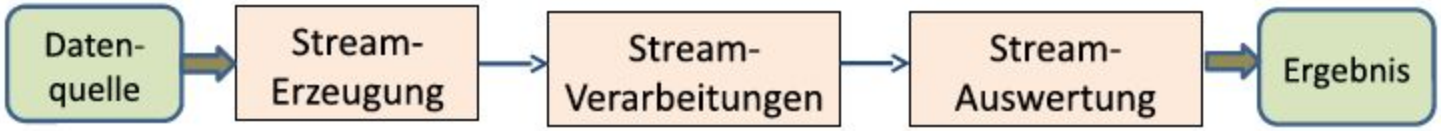
- Dabei sind beliebige Kombinationen möglich:

```
Stream<String> st = //...
int result = st.map(s -> s.toUpperCase())
    .map(s -> s.indexOf("ABC"))
    .filter(n -> n>3)
    .map(n -> n-3)
    .reduce(0, (a, b) -> a+b);

st.map(s -> s.toUpperCase())
    .map(s -> s.indexOf("ABC"))
    .filter(n -> n>3)
    .map(n -> n-3)
    .forEach(n -> System.out.println(n));
```

Stream-Verarbeitung

1. **Erzeugungsoperation:** Erzeugung eines Streams
2. **intermediäre Operationen:** Verarbeitung/Transformation der Elemente
3. **terminale Operation:** Auswertung der Zwischenergebnisse



Bildquelle: Hettel & Tran, 2016

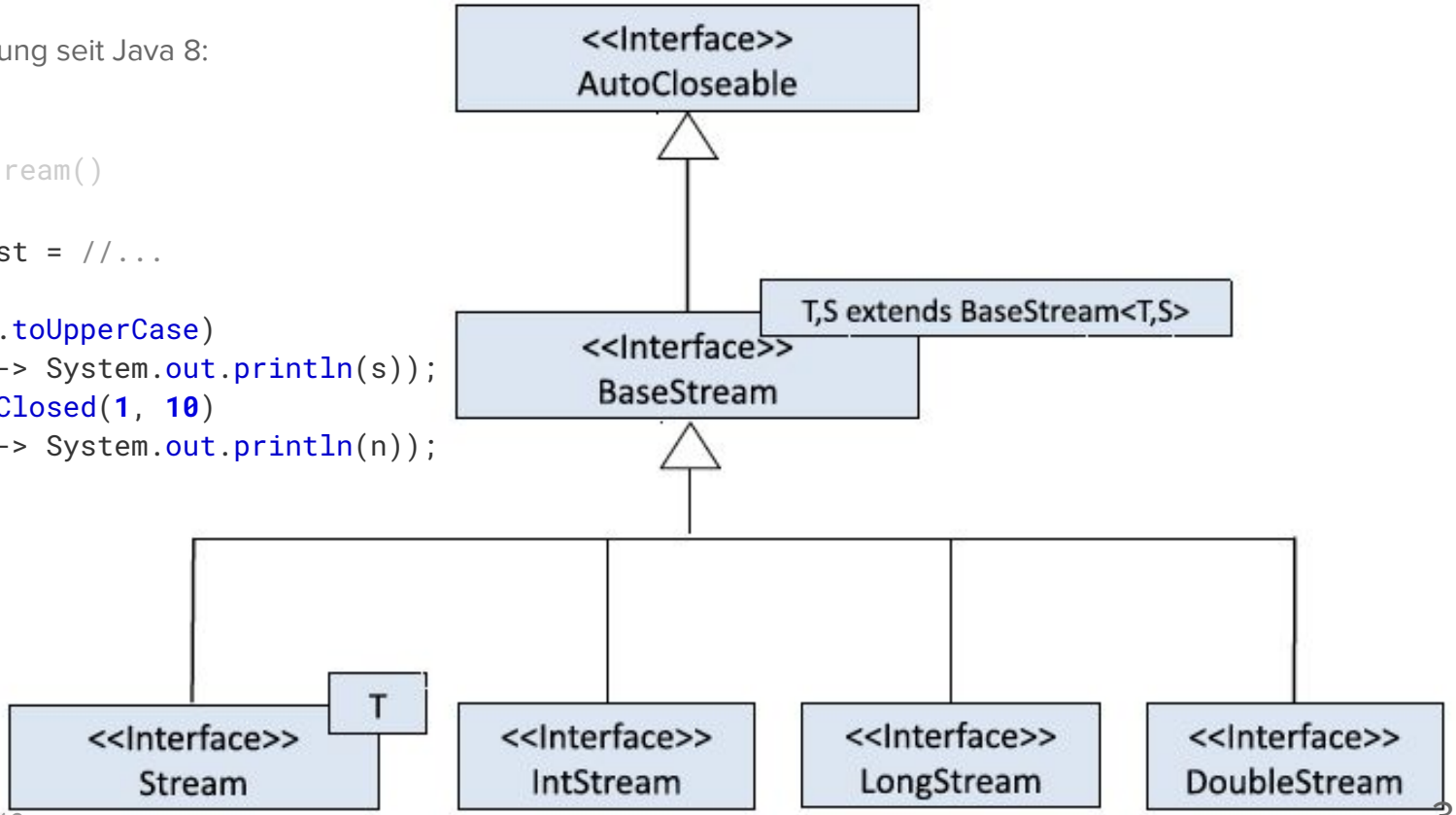
Folie mit Anmerkungen

Arten von Streams

Collection-Erweiterung seit Java 8:

- `stream()`
- `parallelStream()`

```
List<String> list = //...
list.stream()
    .map(s -> s.toUpperCase)
    .forEach(s -> System.out.println(s));
IntStream.rangeClosed(1, 10)
    .forEach(n -> System.out.println(n));
```



Bildquelle: Hettel & Tran, 2016

Sortierung/Begrenzung eines Streams

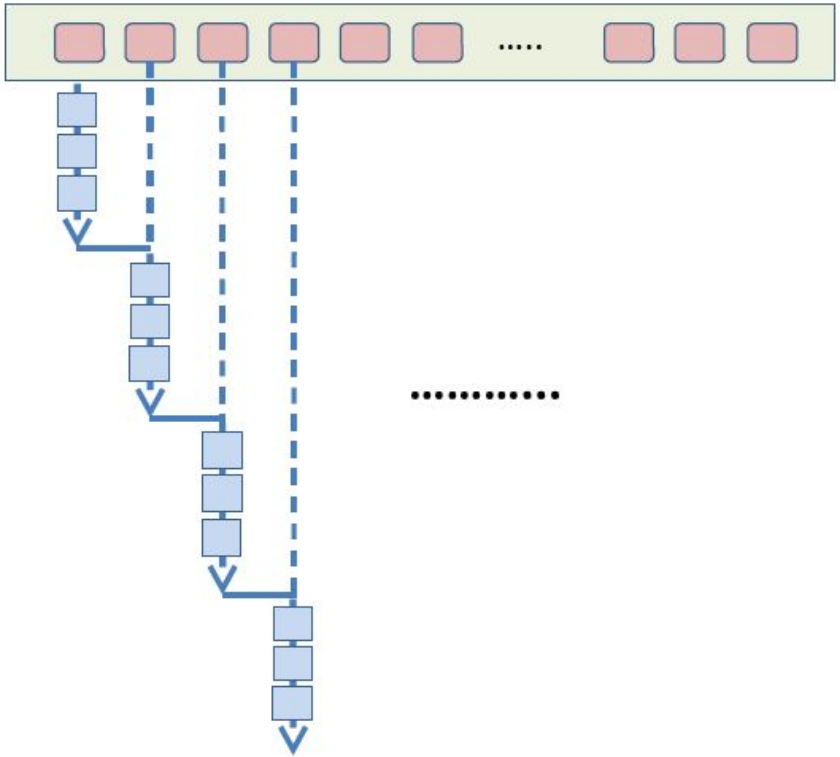
Folie mit Anmerkungen

Methode	Parameter	Rückgabe
sorted	void	Stream<T>
sorted	Comparator: (T,T) -> int	Stream<T>
skip	long	Stream<T>
limit	long	Stream<T>
distinct	void	Stream<T>

Bildquelle: Hettel & Tran, 2016

sequentielle Stream-Verarbeitung

jedes Element des Streams wird nacheinander durch die Verarbeitungspipeline geschickt.

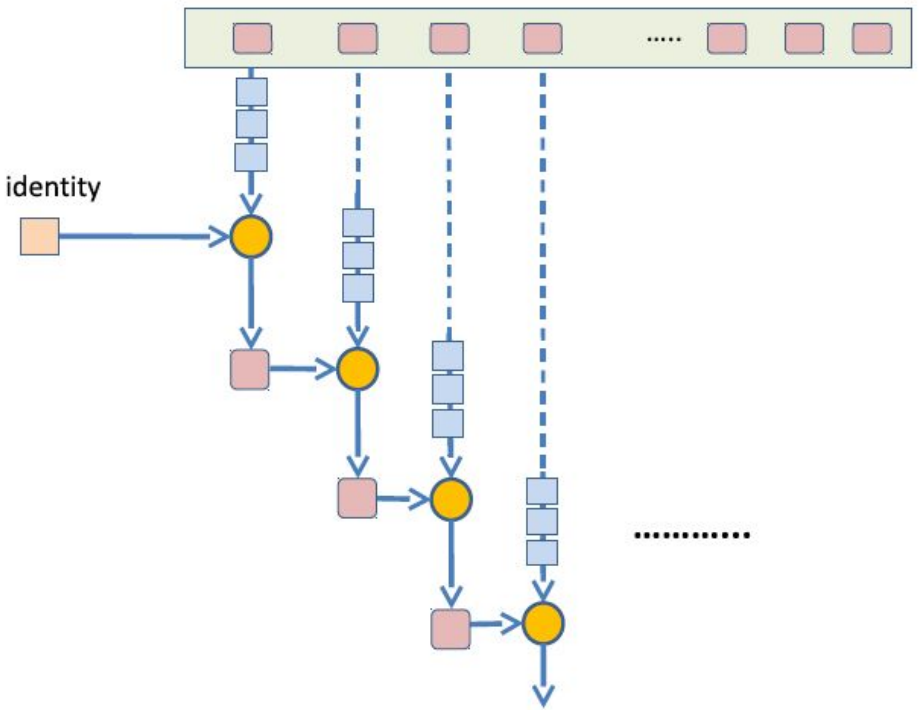


Bildquelle: Hettel & Tran, 2016

sequentielle Stream-Verarbeitung

Folie mit Anmerkungen

Das Ergebnis vom Typ T der Verarbeitungskette jedes Elements wird zusammen mit dem vorigen Teilergebnis vom Typ T durch die reduce-Funktion zum nächsten Teilergebnis berechnet. Zum Start wird das neutrale Element benutzt (z.B. bei Addition 0, bei Multiplikation 1).



Bildquelle: Hettel & Tran, 2016