Universität Stuttgart

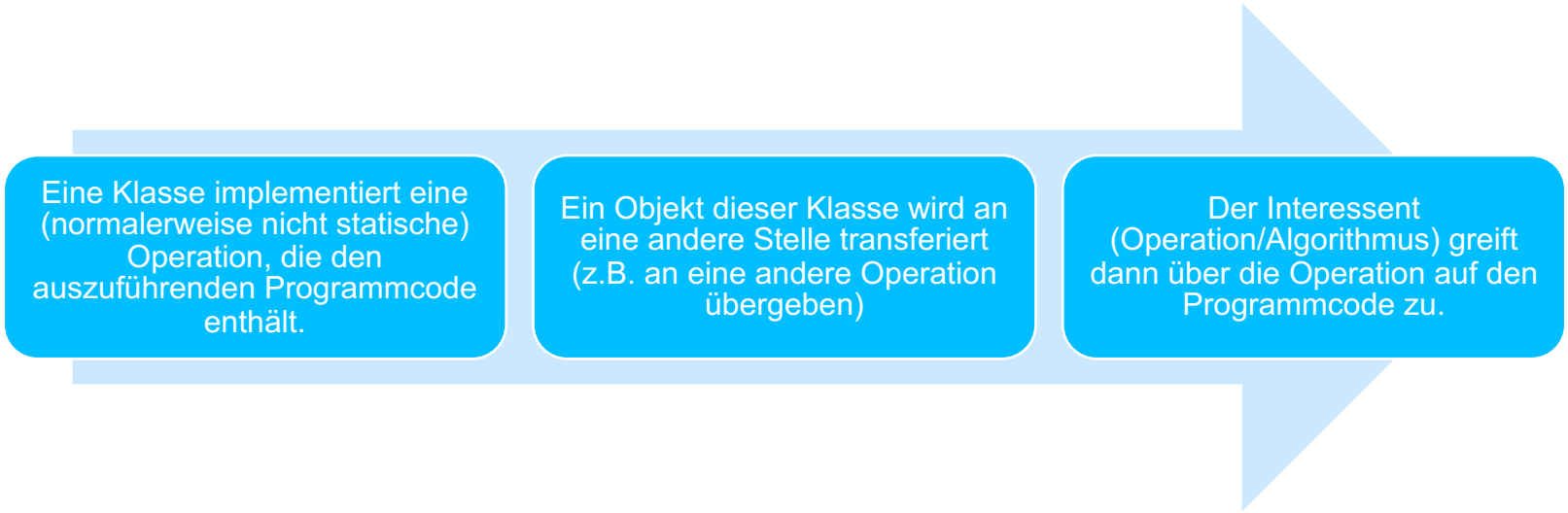
PSE:
Java Stream API

Prof. Dr.-Ing. Steffen Becker

Vorlesung 23


Universität Stuttgart
Recap
PSE:
Lambdas
Prof. Dr.-Ing. Steffen Becker
Vorlesung 22

Muster des Transports von Code in Java mit Objekten



Diesen Mechanismus werden wir uns nun in verschiedenen Varianten näher ansehen.

Beispiel: Sortierung abgeschnittener Strings – Verwendung von Lambdas (seit Java 1.8)

```java
import java.util.*;

public class CompareTrimmedStrings {

    public static void main(String[] args) {

        String[] words = { "M", "\nSkyfall", " Q", "\t\tAdele\t" };

        Arrays.sort(words, (String s1, String s2) -> {
            return s1.trim().compareTo(s2.trim());
        });

        System.out.println(Arrays.toString(words));

    }
}
```

Code übergeben als ein **Lambda**-Ausdruck (in fetter Schrift)

Der Java-Begriff "Lambda-Ausdruck" geht auf den Lambda-Kalkül (auch als λ Kalkül geschrieben) aus den 1930er Jahren zurück und ist eine formale Sprache für die Untersuchung von Funktionen.

Inferierter Typ des Lambda Ausdrucks

```java
import java.util.*;

public class CompareTrimmedStrings {

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

        final String[] words = { "M", "\nSkyfall", " Q", "\t\tAdele\t" };

        final var stringComparator =
            (final String s1, final String s2) -> {
                return s1.trim().compareTo(s2.trim());
            };
        final var stringComparator2 = (s1, s2) -> {
            return s1.trim().compareTo(s2.trim());
        };
        final Comparator<String> typedComparator = stringComparator;
        typedComparator.compare("Test1", "Test2");

    }
}
```

Beispiel: Sortierung abgeschnittener Strings – Verwendung von inneren Klassen

```java
import java.util.*;

public class CompareTrimmedStrings {

    public static void main(String[] args) {

        class TrimmingComparator implements Comparator<String> {

            @Override
            public int compare(final String s1, final String s2) {
                return s1.trim().compareTo(s2.trim());
            }
        }

        String[] words = { "M", "\nSkyfall", " Q", "\t\tAdele\t" };

        Arrays.sort(words, new TrimmingComparator() );

        System.out.println(Arrays.toString(words));
    }
}
```

Zu übergebender Code (hier: innere Klasse; könnte auch eine öffentliche Klasse sein)

Code wird "transportiert".

Beispiel: Sortierung abgeschnittener Strings – Verwendung von inneren Klassen

```java
import java.util.*;

public class CompareTrimmedStrings {

    public static void main(String[] args) {

        class TrimmingComparator implements Comparator<String> {

            @Override
            public int compare(final String s1, final String s2) {
                return s1.trim().compareTo(s2.trim());
            }
        }

        String[] words = { "M", "\nSkyfall", " Q", "\t\tAdele\t" };

        Arrays.sort(words, new TrimmingComparator());

        System.out.println(Arrays.toString(words));
    }
}
```

Code, der als Objekt einer **anonymen inneren** Klasse übergeben wird

Funktionale Schnittstellen

Nicht jede Schnittstelle kann in einem Lambda-Ausdruck verwendet werden, und es gibt eine Schlüsselbedingung, wann ein Lambda-Ausdruck verwendet werden kann.

Interfaces, die nur eine Operation (abstrakte Methode) haben, werden als funktionale Interfaces bezeichnet. Eine abstrakte Klasse mit genau einer abstrakten Methode zählt nicht als funktionales Interface.

Java (>8) kommt mit vielen Schnittstellen, die als funktionale Schnittstellen gekennzeichnet sind. Darüber hinaus führt Java 8 mit dem Paket java.util.function mehr als 40 neue funktionale Schnittstellen ein. Beispiele:

- `interface Runnable { void run(); }`
- `interface Supplier<T> { T get(); } (Java 8)`
- `interface Consumer<T> { void accept(T t); } (Java 8)`
- `interface Comparator<T> { int compare(T o1, T o2); }`
- `interface ActionListener { void actionPerformed(ActionEvent e);`

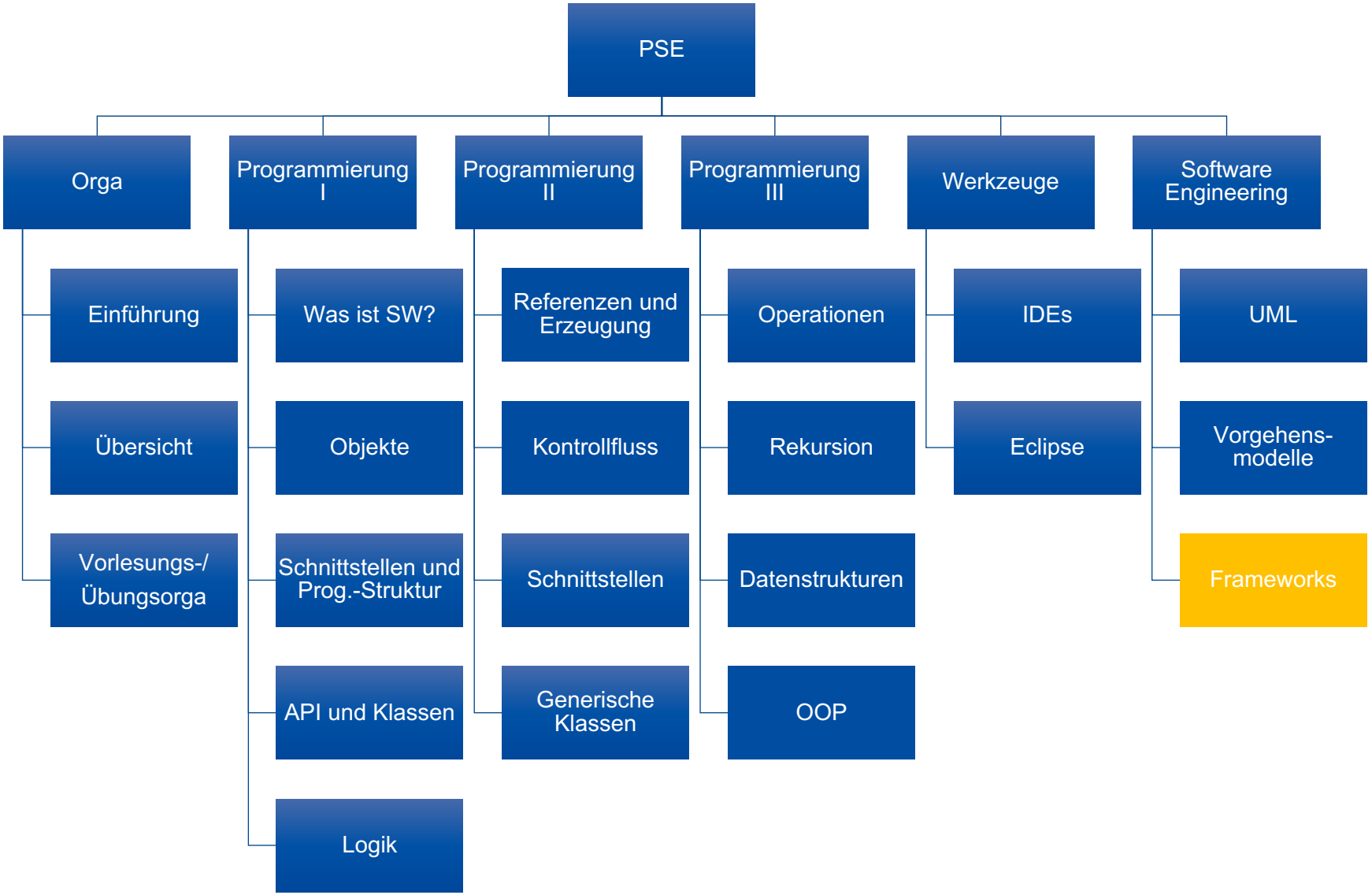
Universität Stuttgart

PSE:
Java Stream API

Prof. Dr.-Ing. Steffen Becker

Vorlesung 23

Vorlesungsübersicht



Lernziele

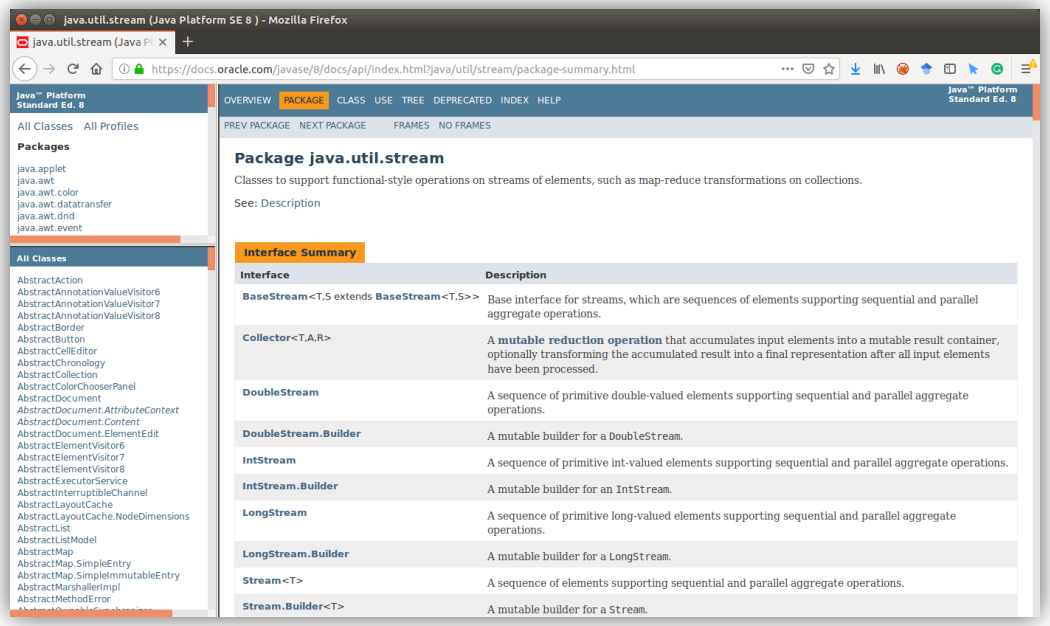
- Eine funktional orientierte Bibliothek kennen lernen
- Die Verwendung von Lambdas vertiefen
- Datenströme beherrschen
- Transformationen auf Datenströmen durchführen

Didaktik der Vorlesung

- Vortragen von Informationen
- Vorher gelerntes aufgreifen und wiederholen sowie vertiefen
- Aktivieren der Studierenden durch kleine interaktive Aufgaben

Literatur für diese Vorlesung

Dieser Foliensatz basiert weitgehenst auf der JavaDoc der Streaming API (java.util.stream):



https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html

Javas Stream-API - worum geht es da?

Die Stream-API befindet sich im Paket `java.util.stream`

"Klassen, um **Operationen nach dem funktionalen Stil** auf Streams von Elementen zu unterstützen, wie bspw. Map-reduce **Transformationen auf Collections**.

Beispiel:

```
Collection<Widget> widgets = …;

int sum = widgets.stream()
                 .filter(b -> b.getColor() == RED)
                 .mapToInt(b -> b.getWeight())
                 .sum();
```

Beispielanwendung der Java Stream API

- Was macht dieser Teil des Codes?
- Können Sie den Unterausdrücken "filtern", "abbilden" und "reduzieren" zuweisen?

```
Collection<Widget> widgets = …;

int sum = widgets.stream()
      .filter(b -> b.getColor() == RED)
      .mapToInt(b -> b.getWeight())
      .sum();
```

Stream

Filter

Map

Reduce

Die Schnittstelle Stream<T> am Beispiel

```
public interface Stream<T>
extends BaseStream<T,Stream<T>>

A sequence of elements supporting sequential and parallel aggregate operations.
```

"Eine Folge von Elementen, die sequentielle und parallele Aggregatoperationen unterstützen."

In unserem Beispiel:

1. Wir erzeugen einen Stream an `Widget`-Objekten über `Collection.stream()`,
2. filtern, um einen Strom zu erzeugen, der nur die roten Widgets enthält, und dann
3. in einen Strom von `int` Werten transformieren, die das Gewicht jedes roten Widgets repräsentieren.
4. Dann wird dieser Stream summiert, um ein Gesamtgewicht zu erhalten.

```
Collection<Widget> widgets = …;

int sum = widgets.stream()
              .filter(b -> b.getColor() == RED)
              .mapToInt(b -> b.getWeight())
              .sum();
```

Stream

Filter → Stream

Map → Stream

Reduce

Stream Operationen und Pipelines

- Streamoperationen warden in **Intermediär-** und **Terminaloperationen** unterteilt und zu **Stream-Pipelines** zusammengefasst.
- Eine Strompipeline besteht aus
  - einer **Quelle** (bspw. Collection, einer Generatorfunktion, oder einem I/O Channel)
  - gefolgt von null oder mehr **Intermediäroperationen** (z.B. `Stream.filter`, `Stream.map`)
  - und eine **Terminaloperation** (z.B. `Stream.forEach`, `Stream.reduce`)

Einen neuen Stream zurückgeben

Kann den Stream traversieren, um ein Ergebnis oder einen Nebeneffekt zu erzielen

Funktionale Schnittstellen

- Java (>8) wird mit vielen Schnittstellen ausgeliefert, die als funktionale Schnittstellen gekennzeichnet sind. Darüber hinaus führt Java 8 mit dem Paket java.util.function mehr als 40 neue funktionale Schnittstellen ein. Beispiele:
  - `interface Runnable { void run(); }`
  - `interface Supplier<T> { T get(); } (Java 8)`
  - `interface Consumer<T> { void accept(T t); } (Java 8)`
  - `interface Comparator<T> { int compare(T o1, T o2); }`
  - `interface ActionListener { void actionPerformed(ActionEvent e); }`

Besonders relevant für Streams

- `interface Predicate<T> { boolean test( T t ); } (Java 8)`
- `Interface Function<T,R> { R apply( T t ) }`

plus Standardoperationen
`and, negate, or, ..`

Beispiel-Operationen

- Zwischenoperationen (zustandslos vs. zustandsbehaftet)

Stream<T>	distinct()
Stream<T>	filter(Predicate<? super T> predicate)
<R> Stream<R>	map(Function<? super T,? extends R> mapper)
Stream<T>	peek(Consumer<? super T> action)
Stream<T>	sorted()

- Terminal- (oder Reduzier-/Falt-) Operationen

boolean	allMatch(Predicate<? super T> predicate)
<R,A> R	collect(Collector<? super T,A,R> collector)
long	count()
void	forEach(Consumer<? super T> action)
int	sum()

Beispieloperation in einer bestimmten Klasse von Streams (hier: IntStream)

Streams vs. Collections

Arten von Streams

- `Stream<T>` ist ein allgemeiner Stream über Objekte
- Die API bietet spezialisierte Streams der primitiven Typen int, long und double: `IntStream`, `LongStream`, and `DoubleStream`
- `Stream<T>` bietet map-Funktionen, um spezialisierte Streams zu erhalten:
  - `DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper)`
  - `IntStream mapToInt(ToIntFunction<? super T> mapper)`
  - `LongStream mapToLong(ToLongFunction<? super T> mapper)`

Parallelität

Die Verarbeitung von Elementen mit einer expliziten for-Schleife ist von Natur aus seriell

Alle Stream-Operationen können entweder seriell oder parallel ausgeführt werden

Die Stream-Implementierungen im JDK erzeugen serielle Streams, es sei denn, die Parallelität wird ausdrücklich gewünscht:

```
Collection<Widget> widgets = …;

int sum = widgets.parallelStream()
                 .filter(b -> b.getColor() == RED)
                 .mapToInt(b -> b.getWeight())
                 .sum();
```

Paralleler Stream

Abgesehen von der Nichtdeterminiertheit sollte die Tatsache, ob ein Datenstrom sequentiell oder parallel ausgeführt wird, das Ergebnis der Berechnung nicht verändern.

Reduktions-Operationen

- Eine Reduktionsoperation (auch Faltung genannt) nimmt eine Folge von Eingabeelementen und kombiniert sie zu einem einzigen zusammenfassenden Ergebnis
- Beispiele:
  - die Summe oder das Maximum einer Menge von Zahlen zu finden
  - Elemente zu einer Liste zu akkumulieren
  - Allgemeine Reduktionsmaßnahmen von Stream
    - `reduce()`
    - `collect()`
  - Spezialisierte Reduktionsoperationen, die von IntStream usw. bereitgestellt werden
    - `sum()`
    - `max()`
    - `count()`

Reduktionsoperationen - nicht-strombasiert vs. strombasiert

Natürlich können solche Operationen leicht als einfache sequentielle Schleifen implementiert werden, wie in

```
int sum = 0;
for (final int x : numbers) {
   sum += x;
}
```

Stream-basierte Implementierungen

```
int sum = numbers.stream().reduce(0, Integer::sum);
```

```
int sum = numbers.parallelStream().reduce(0, Integer::sum);
```

Zusammenfassung

Gewonnene Erkenntnisse

- Einführung in die Stream-API von Java
- Stream-Pipelines und Operationen.
- Verwendung von Lambda-Ausdrücken
- Verhältnis von Streams und Collections (Sammlungen)
- Spezialisierte Streams für primitive Typen

Universität Stuttgart

PSE:
Zusammenfassung

Prof. Dr.-Ing. Steffen Becker

Vorlesung 26