Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Lambdas und Streams

Stephan Fischli

Lambdas

Neues Sprachkonstrukt

- Java ist objektorientiert, also müssen Funktionen als Methoden von Klassen codiert werden:

```
List<String> words = ...;
words.sort(new Comparator<String>() {
    public int compare(String w1, String w2) {
        return w1.length() - w2.length();
    }
});
```

- Mit Lambda-Ausdrücken können Funktionen auch ohne Erzeugung einer Klasse oder eines Objekts übergeben werden:

```
words.sort((String w1, String w2) -> w1.length() - w2.length());
```

Lambda-Syntax

- Lambda-Ausdrücke sind Ausdrücke der Form

  argument-list -> body

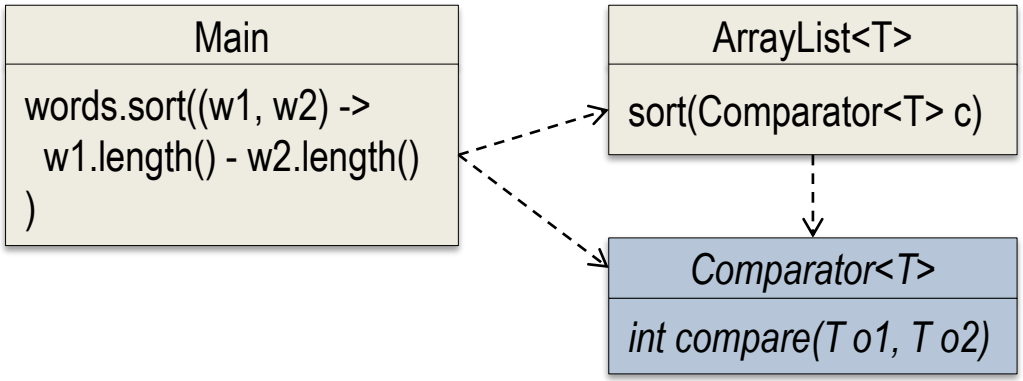
- Der Body kann sein
  - ein einfacher Ausdruck, der ausgewertet wird
  - ein Block von Statements, die ausgeführt werden (kann auch ein Return-Statement enthalten)

- Beispiele

  ```
  (int x, int y, int z) -> x + y + z
  (String s) -> { System.out.println(s); return s.toUpperCase(); }
  () -> 42
  ```

Lambdas und funktionale Interfaces

- Lambda-Ausdrücke sind vom Typ eines funktionalen Interface, also eines Interface mit genau einer Methode

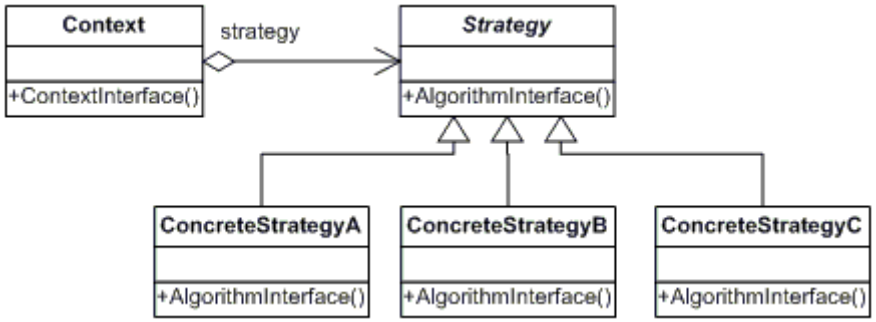


- Beispiele:

```
interface Comparator<T>  { int compare(T o1, T o2); }
interface FileFilter     { boolean accept(File file); }
interface Runnable       { void run(); }
interface ActionListener { void actionPerformed(ActionEvent e); }
```

Verwendung funktionaler Interfaces

- Funktionale Interfaces dienen oft als Parametertypen bei der Implementierung von Verhaltensmustern (Strategy, Visitor, Observer)
- Lambdas können dabei als Argumente verwendet werden



- Beispiel

```
public class ArrayList<T> {
    Object[] data;
    void sort(Comparator<T> c) { ... c.compare(data[i], data[j]) ... }
}
```

Verwendung von Lambdas

- Lambda-Ausdrücke können überall verwendet werden, wo ein Objekt eines funktionalen Interface benötigt wird:
  - Methodenaufrufe
  - Return-Statements
  - Variablendeklarationen
  - Zuweisungen

- Beispiele

```
words.sort((String w1, String w2) -> w1.length() - w2.length());
Comparator<String> getLengthComparator() {
    return (String w1, String w2) -> w1.length() - w2.length();
}
Comparator<String> c = (String w1, String w2) -> w1.length() - w2.length();
```

Zieltyp eines Lambda-Ausdrucks

- Der Compiler leitet den Typ eines Lambda-Ausdrucks aus dem Verwendungskontext ab (Zieltyp)
- Ein Lambda-Ausdruck ist kompatibel zu einem funktionalen Interface, wenn die Parameter, Rückgabewerte und Exceptions zu der Methode des Interface passen
- Da die Parametertypen aus dem Zieltyp bekannt sind, können sie im Lambda-Ausdruck meistens weggelassen werden

```
words.sort((w1, w2) -> w1.length() - w2.length());
Comparator<String> getLengthComparator() {
    return (w1, w2) -> w1.length() - w2.length();
}
Comparator<String> c = (w1, w2) -> w1.length() - w2.length();
```

Methodenreferenzen

- Methodenreferenzen sind wie Lambda-Ausdrücke, referenzieren aber existierende Klassen-, Objektmethoden oder Konstruktoren

- Beispiele:

```
class StringUtil {
    static int compare(String w1, String w2) { return w1.length() - w2.length(); }
}
words.sort(StringUtil::compare);          // Klassenmethode
words.sort(String::compareTo);            // Objektmethode
words.forEach(System.out::println);       // Methode eines bestimmten Objekts
```

Streams

Externe vs interne Iterationen

- Das bisherige Collection-Framework basierte auf externen Iterationen, d.h. Iterationen werden vom Client kontrolliert

```
for (String w: words) { System.out.println(w); }
```

- Bei internen Iterationen delegiert der Client die Iteration an die Bibliothek

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

- Vorteile:
  - Iteration und Logik sind getrennt
  - Optimierte Ausführungen sind möglich

Streams

- Ein Stream
  - repräsentiert eine (evtl. unendliche) Folge von Elementen
  - hat eine Quelle aber speichert selbst keine Elemente
  - hat Operationen zur Manipulation der Elemente (interne Iteration)
  - kann nur einmal verarbeitet werden (konsumierend)

- Ein Stream ist ein Objekt des generischen Typs *Stream<T>* oder der spezialisierten Typen *IntStream*, *LongStream*, *DoubleStream*

Beispiel

- Eine Liste von Wörtern soll gefiltert, der Länge nach sortiert und ausgegeben werden

- Klassische Programmierung

```
List<String> results = new ArrayList<>();
for (String w : words)
    if (w.contains("o")) results.add(w);
results.sort((w1, w2) -> w1.length() - w2.length());
for (String w : results)
    System.out.println(w);
```

- Programmierung mit Streams

```
words.stream().filter(w -> w.contains("o"))
                .sorted((w1, w2) -> w1.length() - w2.length())
                .forEach(System.out::println);
```

Erzeugen von Streams

Streams können aus verschiedenen Quellen erzeugt werden

- Arrays and Collections
  - Stream.of(2,3,5,7,11)
  - Arrays.stream(numbers)
  - words.stream()
  - words.parallelStream()
- Generatoren
  - Stream.empty()
  - Stream.iterate(1, x -> x+1)
  - Stream.generate(Math::random)
  - new Random().doubles()
- Files and Directories
  - Files.lines(path)
  - Files.list(path)
  - Files.walk(path)
- I/O-Streams
  - new BufferedReader(in).lines()
- Strings
  - text.chars()

Stream-Operationen

- Stream-Operationen
  - verwenden funktionale Interfaces als Parameter (Verhalten)
  - verändern die Quelle des Streams nicht (funktional)
  - können zustandslos oder zustandsbehaftet sein
  - werden in Zwischen- und Terminaloperationen unterteilt
  - unterstützen eine fliessende (fluent) Programmierung

- Beispiel

```
int totalLength = words.stream()
    .filter(w -> w.contains("o"))
    .mapToInt(w -> w.length())
    .sum();
```

Zwischenoperationen

- Zwischenoperationen transformieren Streams

```
Stream<T>  distinct()
Stream<T>  limit(long n)
Stream<T>  skip(long n)
Stream<T>  filter(Predicate<T> p)
Stream<R>  map(Function<T,R> f)
Stream<T>  sorted(Comparator<T> c)
Stream<T>  peek(Consumer<T> c)
Stream<T>  parallel()
```

- Zugehörige Interfaces

```
interface Predicate<T>     { boolean test(T o); }
interface Function<T,R>    { R apply(T o); }
interface Comparator<T>  { int compare(T o1, T o2); }
interface Consumer<T>      { void accept(T o); }
```

Terminaloperationen

- Terminaloperationen erzeugen ein Resultat oder einen Nebeneffekt

long	count()
Optional<T>	findAny(), findFirst()
boolean	allMatch(Predicate<T> p), anyMatch(Predicate<T> p), noneMatch(Predicate<T> p)
Optional<T>	min(Comparator<T> c), max(Comparator<T> c)
T	reduce(T identity, BinaryOperator<T> op)
void	forEach(Consumer<T> c)
Object[]	toArray()

- Zugehörige Interfaces

```
interface Predicate<T>        { boolean test(T o); }
interface Comparator<T>       { int compare(T o1, T o2); }
interface BinaryOperator<T>   { T operate(T o1, T o2); }
interface Consumer<T>         { void accept(T o); }
```

Optional

- Ein Optional-Objekt ist ein Container, der einen Wert enthält oder nicht
- Folgende Methoden stehen u.a. zur Verfügung

```
boolean        isPresent()
T              get()              // kann null sein
T              orElse(T o)
T              orElseThrow(Supplier<T> s)
void           ifPresent(Consumer<T> c)
Optional<T>    filter(Predicate<T> p)
Optional<R>    map(Function<T,R> f)
```

- Optionals ermöglichen, optionale Werte ohne Null-Prüfungen weiter zu verarbeiten

Collecting

- Die Terminaloperation collect akkumuliert die Elemente eines Streams mithilfe eines Collector in einem Container

```
R  collect(Collector<T,A,R> c)
```

- Zugehöriges Interface

```
interface Collector<A,T,R> {
    Supplier<A>         supplier();     // erzeugt neue Container
    BiConsumer<A,T>     accumulator();  // fügt Elemente zu Container hinzu
    BinaryOperator<A>   combiner();     // kombiniert zwei Container
    Function<A,R>       finisher();     // transformiert Elemente (optional)
}
```

Collectors

- Die Klasse Collectors stellt viele Kollektoren zur Verfügung

  Collectors.toList()
  Collectors.toSet()
  Collectors.toMap(keyMapper, valueMapper)
  Collectors.joining(delimiter)
  Collectors.groupingBy(classifier)

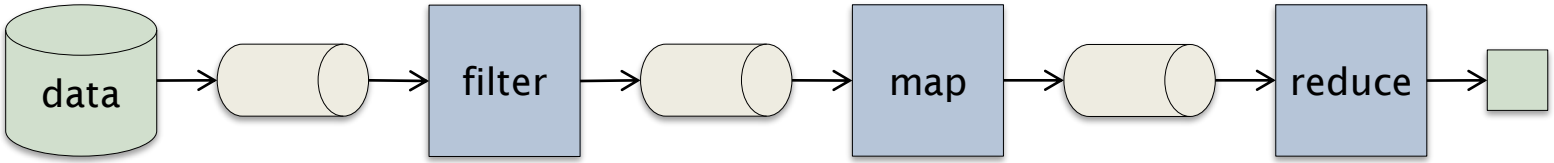
- Beispiele

```
words.stream().filter(w -> w.contains("o")).collect(Collectors.toList())
words.stream().collect(Collectors.toMap(w -> w, String::length)
words.stream().collect(Collectors.joining(", "))
words.stream().collect(Collectors.groupingBy(w -> w.charAt(0)))
```

Pipelines

- Stream-Operationen werden in Pipelines ausgeführt

- Eine Pipeline besteht aus
  - einer Quelle, welche die Elemente liefert
  - Zwischenoperationen, welche den Stream transformieren
  - einer Terminaloperation, die ein Resultat produziert



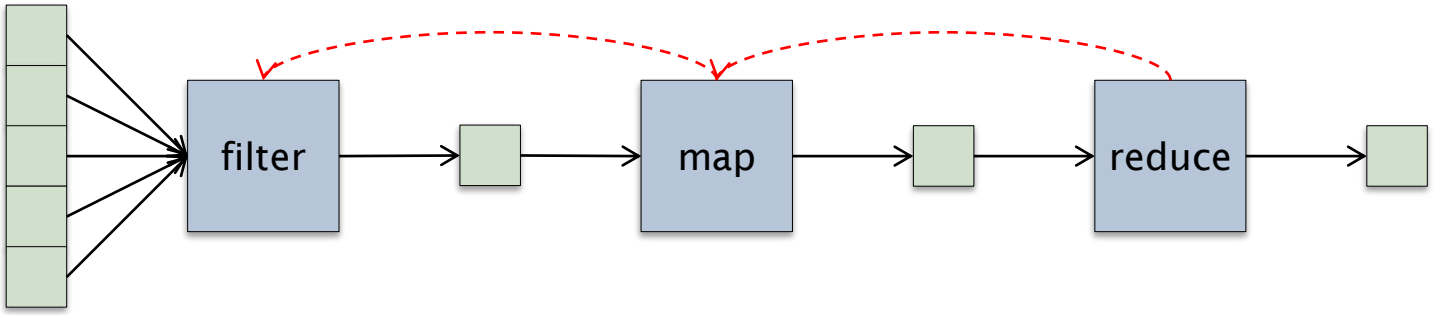
- Beispiel

```
words.stream()
    .filter(w -> w.contains("o")).mapToInt(w -> w.length()).sum();
```

Pipeline-Verarbeitung

- Stream-Operationen werden elementweise und lazy ausgeführt

- Vorteile:
  - Memory-Effizienz (kein Zwischenspeicher nötig)
  - Kurzschlüsse möglich (z.B. vorzeitiger Abbruch beim Suchen)
  - Parallelisierung möglich

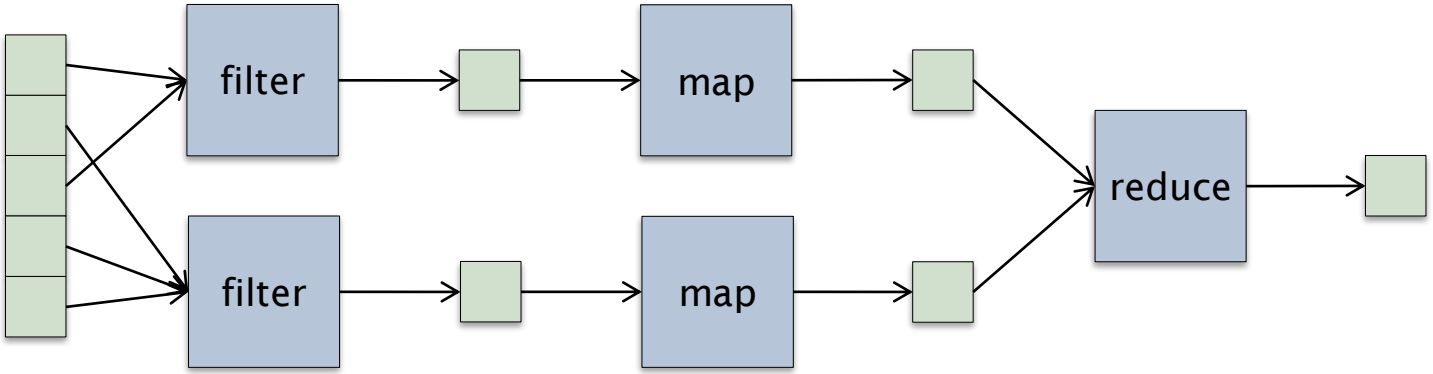


Parallelisierung

- Stream-Pipelines können parallel und damit (auf Multicore-Systemen) effizienter ausgeführt werden

- Beispiel

```
words.parallelStream().filter(w -> w.contains("o"))
    .mapToInt(w -> w.length()).sum();
```



Bemerkungen

- Die Reihenfolge der Elemente bleibt bei parallelen Streams eventuell nicht erhalten
- Zustandsbehaftete Operationen (z.B. distinct, sorted) führen zu Barrieren in der Pipeline-Verarbeitung
- Die verwendeten Lambda-Ausdrücke sollten
  - die Stream-Quelle nicht verändern (non-interference)
  - zustandslos sein und keine Seiteneffekte haben
- Die Effizienzsteigerung ist nur bei grossen Datenmengen und rechenintensiven Operationen wesentlich

Anhang

Referenzen

- Brian Goetz, State of the Lambda
  http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-final.html
  http://cr.openjdk.java.net/~briangoetz/lambda/lambda-libraries-final.html
  http://cr.openjdk.java.net/~briangoetz/lambda/lambda-translation.html

- Angelika Langer, Lambda Expressions and Streams in Java
  http://www.angelikalanger.com/Lambdas/Lambdas.html

- The Java Tutorial, Lambda Expressions
  http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

- Mark Reinholds, Closures for Java
  https://blogs.oracle.com/mr/entry/closures