Closures in Java 8

Lambda-Ausdrücke

Solymosi

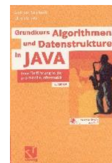
- s = sch
- ly = j
- scheumoschi
- www.solymosi.com
- public.beuth-hochschule.de/~solymosi/lehre/Closures.pdf

Lebenslauf

- 1947: Budapest
- 1966: Leningrad, UdSSR
- 1972: München
- 1976: Erlangen
- 1978: Standford, CA
- APSIS GmbH
- 1989: Berlin
- 1992: Schwante

Angebot

Preisbindung: € 32,99
Angebot: € 33,00



Closures in Java 8

1. Anonyme Sprachelemente
2. Callback
3. Lambda-Ausdrücke
4. Funktionale Schnittstellen
5. Beispiele
6. Funktionale Programmierung

Anonyme Sprachelemente

- anonym = namenlos
- anonymes Objekt

```
Klasse ref = new Klasse(); methode(ref);
methode(new Klasse());
```

- nur wenn einmal

Anonyme Klassen

- Klasse ohne Namen – nur wenn einmal

```
class Klasse implements Typ { … }
    …
new Klasse();
```

- einfacher:

```
new Typ() { … };
```

- auch mit `extends`

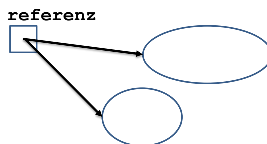
Beispiel

```
JButton knopf = new JButton();
knopf.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    System.out.println("Knopf gedrückt");
  }
});
```

- anonyme Klasse + anonymes Objekt

Anonyme Objekte?

- Alle Objekte sind anonym



- Aber:

```
final Klasse referenz = new Klasse();
```


Beispiel

```
final JButton knopf = new JButton();
this.add(knopf);
```

- anonym:

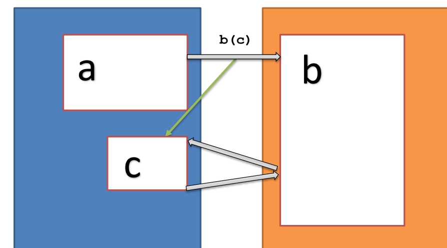
```
this.add(new JButton());
```

- nur einmal

Anonyme Blocks

- nicht in Java <8
- Java 8: anonyme Methoden

Rückruf (Callback)



„Methode übergeben“

- Parameter in Java: Variable
  - primitiv (z.B. `int`)
  - Referenz (mit Objekt)
- Parameter in C/C++: auch Methode/Funktion
- C#: `delegate`

Workaround in Java <8

1. Interface mit einer Methode definieren
2. Klasse implementieren
3. Objekt erzeugen
4. Objekt übergeben

- Beispiel: **`JButton`** + **`ActionListener`**
- Java 8: Lambda-Ausdruck
- Russell & Whitehead: *Principia Mathematica* ˆ
- Alonzo Church, 1936: λ

Lambda-Ausdruck

- alt:

```
knopf.addActionListener(
 new ActionListener() {
  public void actionPerformed(ActionEvent e) {
   System.out.println("Knopf gedrückt"); }});
```

- neu:

```
knopf.addActionListener(
  e -> System.out.println("Knopf gedrückt");
);
```


Beispiel

- Studentenliste

```
static void studenten(
      List<Student> kurs) {
  for (Student student : kurs)
    if (student.getNote() == 1.0)
      System.out.println(student);
}
```

- Problem: Wiederverwendbarkeit

Verallgemeinerung

```
static void studenten(
      List<Student> kurs, int min, int max) {
  for (Student student : kurs)
    if (student.getNote() < min &&
      student.getNote() < max)
      System.out.println(student);
}
```

- Problem: Wiederverwendbarkeit

Verallgemeinerung

```
static void studenten(
      List<Student> kurs, Kriterium kriterium) {
  for (Student student : kurs)
    if (kriterium.test(student))
      System.out.println(student);
}
interface Kriterium {
    boolean test(Student s);
}
```

Aufruf

```
class EinsNull implements Kriterium {
  public boolean test(Student s) {
    return s.getNote() == 1.0;
  }
}
boolean einsNull = new EinsNull();
studenten(kurs, einsNull);
```

Anonyme Implementierung

```
studenten(erstesSemester, new Kriterium() {
  public boolean test(Student s) {
    return s.getNote() == 1.0;
  }
});
```

- `Kriterium` = funktionale Schnittstelle

Funktionale Schnittstelle

- SAM: nur eine abstrakte Methode (ggf. `void`)
- (evtl. weitere default oder static Methoden)
- nicht generisch
- `@FunctionalInterface`
- nur eine → anonym → Lambda-Ausdruck

Lambda-Ausdruck

```
studenten(kurs,
    (Student s) -> s.getNote() == 1.0);
```

- Parameter des Lambda-Ausdrucks: `s`
- Wert des Lambda-Ausdrucks
  `s.getNote() == 1.0`
- Typ des Lambda-Ausdrucks: `boolean`

Runnable

```
Runnable alt = new Runnable() {
  public void run() {
    System.out.println("Alt");
  }
};
new Thread(alt).start();

new Thread(() ->
   System.out.println("Neu")).start();
```

Funktionale Schnittstellen in Java

- `Runnable`
- `Callable<T>`
- `Comparator<T>`
- `java.util.function`
- …

Syntax

1. Liste der formalen Parameter in Klammern, getrennt durch Kommata
   - Typ kann man weglassen (Inferenz)
   - nur ein Parameter → Klammern weglassen
   - kein Parameter → leere Klammern `()`
2. Pfeil `->`
3. Rumpf
   - Ausdruck
   - oder Block in `{` und `}` mit `return`

Formale Syntax

```
LambdaExpression:
  LambdaParameters '->' LambdaBody
LambdaParameters:
  Identifier
LambdaParameters:
  Identifier
  '(' ParameterList ')'
LambdaBody:
  Expression
  Block
```

Methodenreferenz

```
Comparator<String> c =
  (String links, String rechts) ->
    links.compareTo(rechts);
int vgl = c.compare("Hallo", "Welt");
```

- oder

```
Comparator<String> c =
  (String links, String rechts) -> {
    System.out.println("Ich vergleiche" +
      links + " mit " + rechts);
    return links.compareTo(rechts);
  };
int vgl = c.compare("Hallo", "Welt");
```

Typinferenz

```
Comparator<String> c =
  (links, rechts) -> {
    System.out.println("Ich vergleiche" +
      links + " mit " + rechts);
    return links.compareTo(rechts);
  };
int vgl = c.compare("Hallo", "Welt");
```

Vergleich

Lambda-Ausdruck	Äquivalente Methode
`() -> { System.out.println(); }`	`void ♠ () { System.out.println(); }`
`(int x) -> { return x+1; }` `x -> x+1`	`int ♣ (int x) {` `return x+1; }`
`(int x, int y) -> {return x+y;}` `(x, y) -> x+y`	`int ♥ (int x, int y) {` `return x+y; }`

`(String... args) ->`
`{ return args.length; }`
`args -> args.length`
	`int ♦ (String... args) {`
	`return args.length;`
	`}`

`args ->`
`(args != null) ?`
`args.length`
`: 0;`
	`int ☺ (String[] args) {`
	`if (args != null)`
	`return args.length;`
	`else`
	`return 0; }`

Beispiel: `Array.sort()`

```
public static <T> void sort(T[] a,
  Comparator<? super T> c)
```

- Aufruf:

```
Arrays.sort(wörter, (links, rechts) ->
  Integer.compare(
    links.length(), rechts.length()));
```

„Fertige“ Methodenreferenz

```
knopf.addActionListener(e ->
     System.out.println(e));
```

- oder:

```
knopf.addActionListener(System.out::println);
```

Syntax:

- Referenz::instanzmethode
- Klasse::methode (Instanz oder `static`)

Parameterübergabe

```
x -> System.out.println(x)
```

- `System.out::println`

```
(x, y) -> Math.pow(x, y)
```

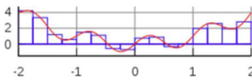
- `Math::pow`

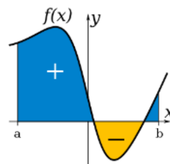
```
Arrays.sort(wörter,
     (x, y) -> x.compare(y))
```

- `Arrays.sort(wörter, String::compare)`
- 1. Parameter = Instanz

Integral

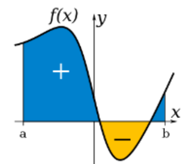
$$\int_a^b f(x)dx = F(b) - F(a)$$





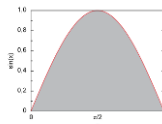
Integral in Java <8

```
interface Funktion {
     double f(double x);
}
static double integral(
     Funktion f,
     double a, double b)
```



Aufruf in Java <8

```
double erg = integral(
  new Funktion() {
    public double f(double x) {
      return Math.sin(x); }},
 0, Math.PI);
```



Integral in Java 8

```
erg = integral(
   (double x) -> Math.sin(x),
   0, Math.PI);
```

- oder

```
erg = integral(
   Math::sin, 0, Math.PI);
```

Kriterium

`Funktion` (Parameter von **`integral`**)
ist eine funktionale Schnittstelle
(= hat nur eine abstrakte nicht-**`void`**-Methode)
→ Lambda-Ausdruck verwendbar

Weitere Features

- Konstruktorreferenz
- Gültigkeit von Variablen
- Default Methods
- Methoden in Schnittstellen

`java.util.function`

- **`Predicate<T>`** Eigenschaft eines Objekts
  - **`test`**, **`and`**, **`or`**, **`xor`**, **`negate`**
- **`Consumer<T>`** Aktion an einem Objekt
- **`Function<T,R>`**: Transformation eines Werts
- **`Supplier<T>`**: Fabrikmethode **`() -> T`**
- **`UnaryOperator<T>`**: Unärer Operator **`T -> T`**
- **`BinaryOperator<T>`**: Binärer Operator
  `T, T -> T`

Streams

- Kommandopipeline →
  - Lesbarkeit
  - Parallelisierbarkeit

Idee: verkettbare Funktionen

```
interface Typ {
 Typ funktion1();
 Typ funktion2(); … }
Typ referenz = …;
referenz.funktion1().funktion2(). …
```

Streams in `Collection`

```
interface Collection { // Java 8
 default Stream<E> stream() { … }
 default Stream<E> parallelStream() { … }

Collection<String> liste = … ;
long zählen = liste.stream().filter(
  new Predicate<String>() {
    public boolean test(String e) {
      return e.length() > 4;}}).count();
```

Stream mit Lambda

```
long zählen = liste.stream().filter(
     e -> e.length() > 4).count();

double durchschnitt = studenten.
stream().filter(p ->
     p.getGeschlecht() == Student.Sex.MALE).
 mapToInt(Student::getAlter).
 average().
 getAsDouble();
```

Verkettbare Funktionen

- intermediate operations:

```
java.util.stream
interface Stream<T> {
  Stream<T> filter(
    Predicate<? super T> predicate)
  Stream<T> limit(long maxSize)
  Stream<T> sorted(Comparator)
  Stream<T> distinct()
  Stream<T> peek(Consumer)
  <R> Stream<R> map(…)
```

Abschließende Operationen

- terminal operations:

```
int count()
Optional<T> min()
Optional<T> max()
T reduce()
Optional<T> findFirst()
```

Enttäuschung 1

- Inkompabilität zu `Object`

```
Object o = () -> System.out.println("");
```

- Syntaxfehler: `Object` implementiert keine funktionale Schnittstelle
- Workaround:

```
Object o = (Runnable) () ->
    System.out.println("");
```

Enttäuschung 2

- Rekursion

```
UnaryOperator<Integer> fak =
  n -> { return n == 0 ? 1 :
    n * fak.apply(n-1); };
```

// the local variable `fak` may not have been initialized

- Workaround: anonyme innere Klassen

Enttäuschung 3

- Generizität

```
interface Fabrik {
  <T> Generisch<T> neu(); }
class Generisch<X> { … }
Fabrik f = () -> new Generisch<>();
f = () -> new Generisch<?>();
f = () -> new Generisch<Long>();
    // lambda is not generic
```

`Quicksort` traditionell

```
public class QuickSort<E extends Comparable<E>> {
 private void sort(E[] sammlung, int links, int rechts) {
  int auf = links, ab = rechts; // linke und rechte Grenze
  final E anker = sammlung[(links + rechts) / 2];
  do {
   while (sammlung[auf].compareTo(anker) < 0) auf ++;
     // suchen größeres Element von links an
   while (anker.compareTo(sammlung[ab]) < 0) ab --;
    // suchen kleineres Element von rechts an
   if (auf <= ab) { // austauschen auf und ab:
    final E temp = sammlung[auf];
    sammlung[auf] = sammlung[ab]; sammlung[ab] = temp;
    auf ++; ab --; }; // linke und rechte Grenze verschieben
  } while (auf <= ab); // Überschneidung
  if (links < ab) sort(sammlung, links, ab); // linke Hälfte sortieren
  if (auf < rechts) sort(sammlung, auf, rechts); } } // rechte Hälfte
```

Quicksort mit Lambdas

```
import com.google.common.base.Predicate;
import com.google.common.collect.ImmutableList;

import static com.google.common.collect.Iterables.filter;
import static com.google.common.collect.Iterables.get;
import static com.google.common.collect.Iterables.skip;
import static com.google.common.collect.Iterables.size;
import static com.google.common.base.Predicates.not;

public class Quicksort<E extends Comparable<E>> { …
```

Quicksort Rumpf

```
public static Iterable<E> quicksort(Iterable<E> s) {
  if (size(s) <= 1) { return s; }
  final E anker = get(eingabe, 0);
  final Iterable<E> rest = skip(s, 1);
  final Predicate<E> istKleiner =
    element -> element.compareTo(anker) < 0;
  return ImmutableList.<E>builder()
  .addAll(quicksort(filter(rest, istKleiner), c))
  .add(anker)
  .addAll(quicksort(filter(rest, not(istKleiner)), c))
  .build(); } …
```

Quicksort mit Comparator

```
public static <E> Iterable<E> quicksort(
  Iterable<E> eingabe, final Comparator<E> c) {
  if (size(eingabe) <= 1) { return eingabe; }
  final E anker = get(eingabe, 0);
  final Iterable<E> rest = skip(eingabe, 1);
  final Predicate<E> istKleiner =
    element -> c.compare(element, anker) < 0;
  return ImmutableList.<E>builder()
  .addAll(quicksort(filter(rest, istKleiner), c))
  .add(anker)
  .addAll(quicksort(filter(rest, not(istKleiner)), c))
  .build(); } …
```

Funktionale Programmierung

- Nur Funktionen, keine Nebenwirkungen
  - keine globale Variablen/Daten
  - keine Zuweisungen
- 1930 ← Lambda-Kalkül
- Turing-Vollständigkeit
- Funktionen höherer Ordnung
- LISP (1958), Scheme (1980), Haskell (1990), XSLT (1999), Scala (2004), Java 8 (2013)

Rekursion

- anstelle von Wiederholung
- uneingeschränkt → Turing-vollständig
- eingeschränkt →
  - totale funktionale Programmierung
  - Halteproblem lösbar (Turner 2004)
  - Korrektheit verifizierbar

Links

- Java SE 8: Lambda Quick Start

www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart
www.oracle.com/technetwork/articles/java/architect-lambdas-part1-2080972.html
www.geekyarticles.com/2014/02/functional-programming-with-java-8.html

Angelika Langer & Klaus Kreft: Lambda Expressions in Java
http://www.angelikalanger.com/Lambdas/Lambdas.pdf (unfertig)

- Quicksort and Java 8

drew.thecsillags.com/quicksort-and-java-8

- Vorlesungsfolien

public.beuth-hochschule.de/~solymosi/lehre/Closures.pdf