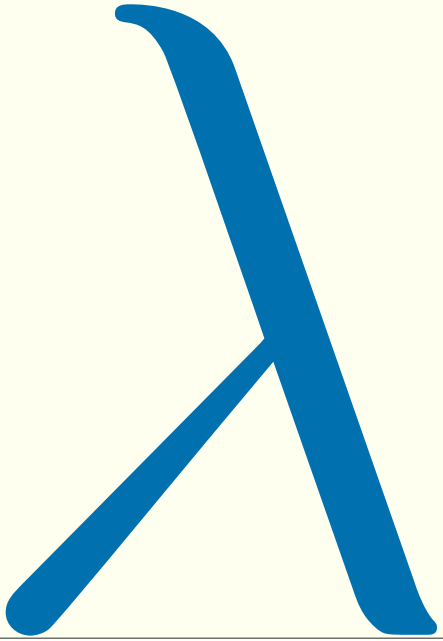
Lambdas
Java-8-Streams
Applikative Programmierung
Funktionale Formen als Kontrollstrukturen
Funktionale Elemente in Java



Charakter der funktionalen Programmierung (Wiederholung)

- → ohne Seiteneffekte keine Kommunikation über gemeinsame Variablen
- → Aliase harmlos, Original und Kopie nicht unterscheidbar (*referenzielle Transparenz*)
- → „sauber“: aufgesammeltes Wissen geht nie verloren
- → Funktion höherer Ordnung = funktionale Form, kann jede Kontrollstruktur ersetzen
- → bei hohen Abstraktionsgraden eher λ-Abstraktion oder strukturelle Abstraktion
- → ausschließlich Rekursion statt Schleifen
- → heute meist vollständig statisch typisiert (Typinferenz), ältere Sprachen dynamisch
- → Lazy-Evaluation einfach

ganz anders als prozedurale oder objektorientierte Programmierung,
radikales Umdenken (Einlassen auf neue Denkweise) nötig

- → vermeiden, Kontrollfluss (oder Typinferenz) nachvollziehen zu wollen
- → vermeiden, alles über nominale Abstraktionen beschreiben zu wollen
- → man darf sich auf den Compiler verlassen (oft fehlerfrei wenn Typen konsistent)
- → es kommt nicht darauf an, wie mächtig Sprachkonzepte sind, sondern wie man sie einsetzt

Lambda als anonyme innere Klasse

```
public class List<A> implements Collection<A> {
    private Node<A> head = ...;
    public Iterator<A> iterator() {
        return new Iterator<A>() {
            private Node<A> p = head;
            public boolean hasNext() { return p != null; }
            public boolean next() { ... }
        }
    }            // Compiler erzeugt Klasse, z.B. List$1.class
}
```

Lambda stärker eingeschränkt als anonyme innere Klasse, aber sehr ähnliches Konzept

- → funktionales Interface muss implizit gegeben sein (vergleichbar mit `Iterator<A>`)
- → eine Methode pro Lambda (Signatur aus funktionalem Interface)
- → keine veränderbaren Variablen aus Umgebung zugreifbar (wohl aber unveränderbare)

Funktionale Interfaces

aus Sicht von Java:

jedes Interface mit nur einer abstrakten Methode ist funktionales Interface

aus Sicht von Programmierer_innen:

funktionales Interface ist mit `@FunctionalInterface` annotiert,
beruht auf **struktureller Abstraktion** (Namen und Kommentare bedeutungslos)

→ fundamental verschieden von typischen Interfaces in objektorientierter Programmierung

wegen struktureller Abstraktion reichen wenige generische Interfaces für alle Zwecke

→ vordefinierte funktionale Interfaces in `java.util.function`, z. B.:

```
Function<T,R>          R apply(T t)
BiFunction<T,U,R>      R apply(T t, U u)
Consumer<T>            void accept(T t)
Supplier<T>            T get()
Predicate<T>           boolean test(T t)
```

Beispiele zu Lambdas

```
Consumer<String> p = s -> System.out.println(s);
p.accept("Hello world.");
Function<Integer,String> value = i -> "value = " + i;
p.accept(value.apply(8));
BiFunction<String,Boolean,String> opt = (s,b) -> b ? s : "";
p.accept(opt.apply("maybe", true));

value = i -> { String r = "value = "; // value änderbar
               r += i;                // lokale Variable r änderbar
               p.accept(r);           // p aus Umgebung, nicht änderbar
               return r;
             };
value.apply(6);
```

Weitere Beispiele zu Lambdas

```
BiFunction<String,String,Integer> cmp = String::compareTo;
                 // entspricht    cmp = (s,t) -> s.compareTo(t);

BiFunction<Object,Object,Boolean> eq = Objects::equals;
                 // entspricht    eq = (a,b) -> Objects.equals(a,b);

Function<StringBuilder,String> mk = String::new;
                 // entspricht mk = sb -> new String(sb);
```

Mächtigkeit von Lambdas

untypisierter λ-Kalkül hat die Mächtigkeit der Turing-Maschine

aber Lambdas in Java sind (auf einfache Weise) statisch typisiert

- → Typen (aufgebaut aus funktionalen Interfaces) können nicht unendlich groß sein
- → Lambdas alleine können daher keine Rekursion darstellen
- → Lambdas alleine haben **nicht** die Mächtigkeit der Turing-Maschine
- → Lambdas werden nicht alleine verwendet, sondern mit anderen Sprachkonstrukten
- → gesamte Sprache (Java) hat die Mächtigkeit der Turing-Maschine

Vergleich zu statisch typisierten rein funktionalen Sprachen:

- → Einschränkung der Mächtigkeit wegen endlicher statischer Typen inakzeptabel
- → zusätzliche Regel (Y-Kombinator), die statische Typisierung bei Rekursion ermöglicht
- → in Java nicht nötig, weil Rekursion und Schleifen ohnehin vorhanden

Basis von Java-8-Streams: Spliterator

Java-8-Streams sind aneinandergereihte Iteratoren mit Zusatzfunktionalität (vereinfacht):

```
interface Spliterator<T> {
    // execute action on next element; returns false if no more element
    boolean tryAdvance(Consumer<? super T> action);

    // execute action on all remaining elements
    default void forEachRemaining(Consumer<? super T> action)
        { do { } while (tryAdvance(action)); }

    // divide elements in this in two parts of about equal size;
    // return one part as new Spliterator, continue with other part
    Spliterator<T> trySplit();
    ...
}
```

Unterschiede zwischen Iterator und Spliterator

Über `Iterator<T>` von außen in Schleife zugegriffen (`next` und `hasNext`),
über `Spliterator<T>` kann Schleife auch im Iterator liegen (`forEachRemaining`)

`next` in `Iterator<T>` gibt Elemente zurück, die danach verarbeitet werden,
`tryAdvance` verarbeitet Element direkt in übergebenem Lambda

Über `Iterator<T>` erfolgen Zugriffe immer nacheinander in gegebener Reihenfolge,
`Spliterator<T>` kann über `trySplit` Aufspaltung (Änderung der Reihenfolge) erlauben

`java.lang.Iterable<T>` erzeugt über `iterator` ein `Iterator`-Objekt, über
`spliterator` ein `Spliterator`-Objekt und über `forEach` einen internen Iterator

Operationen auf Datenströmen

Datenströme bestehen aus `Spliterator`-Objekten mit Info, wie diese verknüpft sind

Datenstrom-erzeugende Operationen:

Methoden, die einen Datenstrom (z. B. `Stream<T>`) erzeugen

→ `stream()`, `parallelStream()` in Standard-Klassen, die `Iterable<T>` implementieren

→ Methoden in `StreamSupport`, um aus `Spliterator`-Objekt `Stream`-Objekt zu erzeugen

Datenstrom-modifizierende Operationen:

Methoden in z. B. `Stream<T>`, die `Stream`-Objekte erzeugen

→ nehmen Lambdas als Parameter und geben sie an `tryAdvance` weiter

Datenstrom-abschließende Operationen:

Methoden in z. B. `Stream<T>`, die keine `Stream`-Objekte erzeugen

→ stoßen Berechnungen durch `tryAdvance` an und sammeln Ergebnisse auf

Datenströme als zusammengefügte Spliteratoren

abschließende Operation ruft wiederholt `tryAdvance` in Datenstrom davor auf, solange Daten verfügbar

modifizierende Operation ruft bei Aufruf von `tryAdvance` ebenso `tryAdvance` in Datenstrom davor auf und gibt modifizierte Daten an Aufrufer weiter

...

erzeugende Operation erzeugt bei Aufruf von `tryAdvance` nächstes Element

Lazy-Evaluation ergibt sich automatisch aus der Zusammensetzung von Spliteratoren

Aufgabe auf bestehende Funktionalität zurückführen

Ziel: Aufgabe auf bestehende Lösungsansätze zurückführen

```
public static long fact(int n) {
    return LongStream.rangeClosed(2, n).reduce(1, (i,j) -> i*j);
}
```

erfordert Kreativität

Beispiel für applikative Programmierung

```
public static Map<String, Map<String, Long>>
              toMap(Collection<Set<String>> sales) {
    return sales.stream()
        .flatMap(set -> set.stream()
            .flatMap(p -> set.stream()
                .filter(q -> !p.equals(q))
                .map(q -> new AbstractMap.SimpleEntry<>(p, q))
            )
        )
        .collect(Collectors.groupingBy(e -> e.getKey(),
             Collectors.groupingBy(e -> e.getValue(),
                  Collectors.counting())));
}
```

Beispiel für imperative Herangehensweise mit Lambdas

```
public static Map<String, Map<String, Long>>
              toMap2(Collection<Set<String>> sales) {
    Map<String, Map<String, Long>> res = new HashMap<>();
    sales.forEach(set ->
        set.forEach(p -> {
            Map<String, Long> map = res.computeIfAbsent(p,
                                        k -> new HashMap<>());
            set.forEach(q -> {
                if (!p.equals(q))
                    map.compute(q, (k,v) -> v==null ? 1 : v+1);
            });
        })
    );
    return res;
}
```

Applikative Programmierung in der Praxis

Abarbeitung erfolgt nach restriktivem, fixem Schema, etwa **Map-Reduce**
→ Einschränkung als Vorteil sehen

viele Aufgaben gut nach diesem Schema lösbar (aber nicht alle)

überschaubare Menge vorgefertigter Funktionen reicht für Schema

manchmal andere Methoden nötig als üblich (z. B. `compute` statt `get` und `put`)

Generizität wichtig, Typen meist durch Typinferenz ermittelt
→ Typen erst in fertigem Programm konsistent, Empfehlungen der IDE nicht hilfreich

wenn Typen konsistent, dann Programm oft fehlerfrei (strukturelle Typen)

„Wer (nur) einen Hammer hat, sieht in jedem Problem einen Nagel."
→ nicht das Schema selbst ist großartig, sondern der gewohnte Umgang damit

Empfehlungen zur applikative Programmierung

Vorgehensweisen durch Kommentare skizzieren,
Zusicherungen auf Hilfsmethoden (Lambdas) dagegen sinnlos

im Umfeld Variablen so verwenden, als ob sie `final` wären

ganz in funktionaler oder ganz in prozedural-objektorientierter Denkweise bleiben

Funktionen (höherer Ordnung) so allgemein wie möglich halten,
Zustandsänderungen (wenn nicht vermeidbar) lokal halten

Nachbildung von Fallunterscheidungen über dynamisches Binden

```
interface Bool {
    <A> A ifThenElse(A t, A f);
    default Bool negate() { return ifThenElse(False.VALUE, True.VALUE); }
    default Bool and(Bool b) { return ifThenElse(b, False.VALUE); }
    default Bool or(Bool b) { return ifThenElse(True.VALUE, b); }
}
final class True implements Bool {
    private True() {}
    public static final True VALUE = new True();
    public <A> A ifThenElse(A t, A f) { return t; }
}
final class False implements Bool {
    private False() {}
    public static final False VALUE = new False();
    public <A> A ifThenElse(A t, A f) { return f; }
}
```

Kurzschlussoperatoren durch verzögerte Auswertung

Problem: Argumente von `ifThenElse` sofort ausgewertet, auch wenn unnötig („&“, „|“)

Lösung: verzögerte Auswertung durch Verwendung von Funktionen statt Werten:

funktionales Interface `java.util.function.Supplier<T>` mit Methode `T get()`

```
default <T> T getIfThenElse(Supplier<T> t, Supplier<T> f) {
    return ifThenElse(t, f).get();
}
default Bool andThen(Supplier<Bool> b) {        // entspricht &&
    return getIfThenElse(b, () -> False.VALUE);
}
default Bool orElse(Supplier<Bool> b) {         // entspricht ||
    return getIfThenElse(() -> True.VALUE, b);
}
```

Beliebig verzögerte Auswertung

```
import java.util.function.*;
@FunctionalInterface
interface LazyBool extends Supplier<Bool> {
    static final LazyBool TRUE = () -> True.VALUE;
    static final LazyBool FALSE = () -> False.VALUE;
    default <T> Supplier<T> ifThenElse(Supplier<T> t, Supplier<T> f)
        { return () -> get().ifThenElse(t, f).get(); }
    default LazyBool negate()
        { return () -> get().ifThenElse(False.VALUE, True.VALUE); }
    default LazyBool and(LazyBool b)
        { return () -> get().ifThenElse(b, FALSE).get(); }
    default LazyBool or(LazyBool b)
        { return () -> get().ifThenElse(TRUE, b).get(); }
}
```

Faustregel zum Ausführungszeitpunkt

Es gibt nur zwei sinnvolle Ausführungszeitpunkte:

- so früh wie möglich (eager evaluation)
- so spät wie möglich (lazy evaluation)

„so früh wie möglich“ erlaubt trotzdem verzögerte Ausführungen, die semantisch nötig sind (z. B. für Kurzschlussoperatoren)

Simulation von Hintereinanderausführung und Wiederholung

```
public static <T,V,R> Function<T,R>
              compose(Function<V,R> f, Function<T,V> g) {
    return t -> f.apply(g.apply(t));
}

public static <T> Function<T,T>
              loopWhile(Function<T,Bool> cond, Function<T,T> iter) {
    Function<T,T> doIt = i -> loopWhile(cond, iter).apply(iter.apply(i));
    Function<T,T> done = i -> i;
    return init -> cond.apply(init).ifThenElse(doIt, done).apply(init);
}
```

Ähnlichkeiten erkennen und nutzen

```
public static <T> void arrayMap(T[] xs, Function<T,T> f) {
    for (int i = 0; i < xs.length; i++)
        xs[i] = f.apply(xs[i]);
}

public static <T> void arrayMap2(T[] xs, Function<T,T> f) {
    Arrays.setAll(xs, i -> f.apply(xs[i]));
}
```

Vor dem Implementieren einer Funktion höherer Ordnung vergewissern, ob es nicht schon eine ähnliche Funktion gibt. Vordefinierte Funktion bevorzugen.

Optional

```
public static Optional<FileReader> openFile(String... path) {
    return Stream.of(path)
        .map(String::trim)
        .reduce((s, t) -> s + "/" + t)
        .map(s -> {try{return new FileReader(s);}
                  catch(java.io.IOException e){return null;}});
}
```

Methoden von `Optional<T>` (unvollständig):

```
boolean isPresent()
T orElse(T other)
T orElseGet(Supplier<? extends T> other)
<U> Optional<U> map(Function<? super T, ? extends U> mapper)
```

Currying

```
BiFunction<String,String,String> f = (s, t) -> s + t;
Function<String,Function<String,String>> g = s -> t -> s + t;

String s = f.apply("a", "b");
String t = g.apply("a").apply("b");
```

→ keine funktionalen Interfaces für Methoden mit mehr als einem Parameter nötig
→ `BiFunction` existiert nur aus Bequemlichkeit, nicht kompatibel zu `Function`
→ Syntax von Lambdas verlangt keinen Zusatzaufwand für Currying
→ Aufwand für Auswertung mit Currying etwas größer
→ Currying erhöht Flexibilität bei Auswertung, diese Flexibilität ist aber selten nötig
→ Typen können mit Currying sehr komplex sein, `var`-Deklarationen dafür nicht anwendbar
→ komplexe Typen verringern Fehlerwahrscheinlichkeit