Lucerne University of
Applied Sciences and Arts

HOCHSCHULE
LUZERN

Technik & Architektur

"Functional Thinking" -

Funktionale Programmierung mit Java 8

```
                    \
                  int
                 _,l;\
                char*I,
               *O[]={"",
              "gjstu","t"
             "fdpoe","uij"
            "se","gpvsui",\
             "gjgui","t"
             "jyui","tfwf"
            "oui","fjhiui",
           "ojoui","ufoui",\
          "fmfwfoui","uxfmgu"
         "i","b!qbsusjehf!jo!"
        "b!qfbs!usff/\xb\xb",""
           "uxp!uvsumf!epwf"
          "t-\xb","uisff!gsf"
         "odi!ifot-!","gpvs!d"
        "bmmjoh!cjset-!","gjwf"
       "!hpme!sjoht<\xb","tjy!h"
      "fftf!b.mbzjoh-!","tfwfo!t"
     "xbot!b.txjnnjoh-\xb","fjhiu"
    "!nbjet!b.njmljoh-!","ojof!mbe"
        "jft!ebodjoh-!","ufo!m"
      "pset!b.mfbqjoh-\xb","fm"
     "fwfo!qjqfst!qjqjoh-!","ux"
    "fmwf!esvnnfst!esvnnjoh-!",""
   "Po!uif!","!ebz!pg!Disjtunbt!n"
  "z!usvf!mpwf!hbwf!up!nf\xb","boe"
 "!"};int putchar(int);int main(void
 ){while(l<(sizeof O/sizeof*O-2)/2-1){
      I=O[_=!_?sizeof O/sizeof*O-
     3:_<(sizeof(O)/sizeof*O-2)/2?
    sizeof O/sizeof*O-2:_==(sizeof(
   O)/sizeof*O-2)/2?++l,0:_<(sizeof(
   O)/sizeof(*O))-3?(_-1)==(sizeof(O)/
  sizeof*O-2)/2?sizeof O/sizeof*O-1:_-1
 :_<sizeof(O)/sizeof*O-2?l+1:_<sizeof(O)
/sizeof*O-1?l+(sizeof O/sizeof(*O)-2)/2:(
sizeof(O)/sizeof*O-2)/2];while(*I){putchar(
                *I++-1);}}
                return 0;}
```

http://digitrode.ru/images/2013/09/18/P2.png

Mindblast Informatik
9. Dezember 2015

Prof. Dr. Ruedi Arnold
ruedi.arnold@hslu.ch

Zusammenfassung *[ausgeblendete Folie!]*

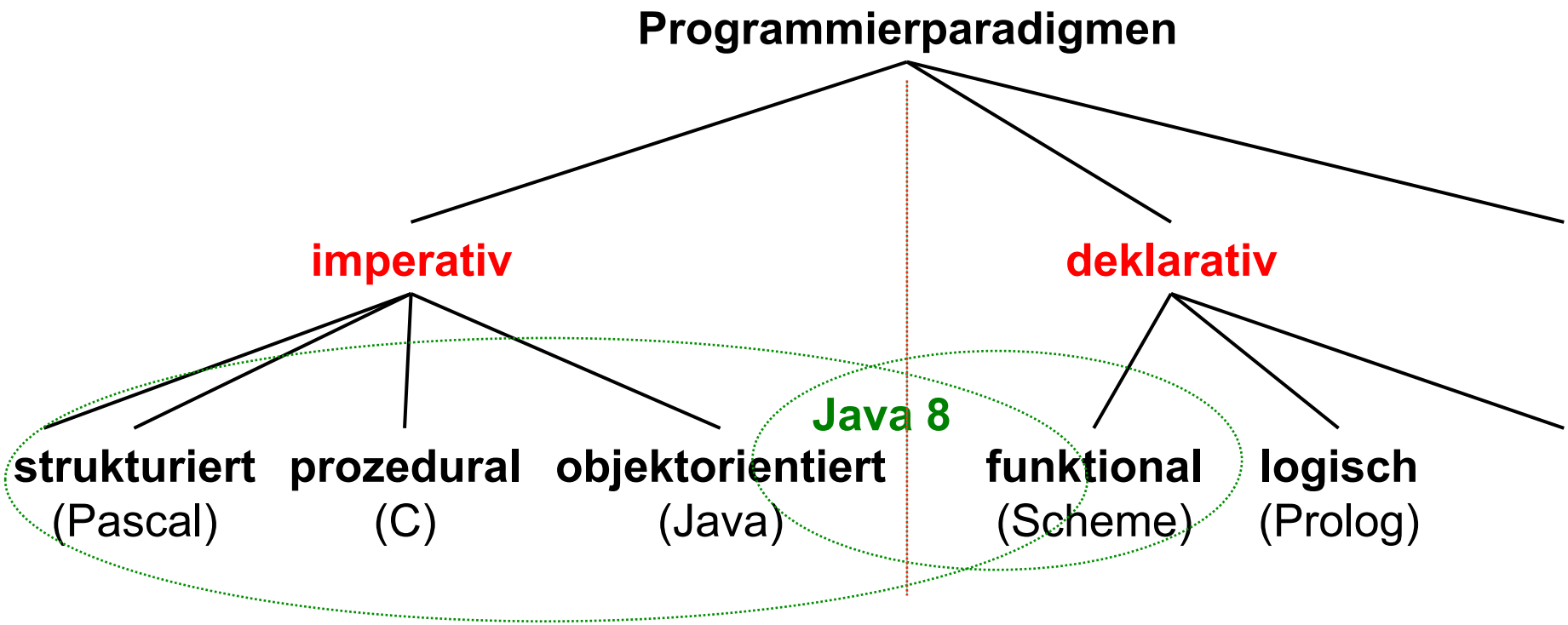
Funktionale Programmierung bietet eine höhere Stufe der Abstraktion, das setzt eine andere Denkweise als beim imperativen oder objekt-orientierten Paradigma voraus. Anhand von typischen Bausteinen der funktionalen Programmierung und mit Code-Beispielen aus dem Buch "Functional Thinking" (2014) von Neal Ford erläutere ich in diesem Vortrag, was das konkret bedeutet. Das ganze wird abgerundet durch praktische Anwendungsbeispiele der mit Java 8 eingeführten funktionalen Sprachmittel Lambda-Ausdrücke und dem Stream-API sowie einer Nebenläufigkeits-Demo mit CompletableFutures.

Einleitende Bemerkungen

- Ausgangslage: Funktionale Programmierung?!.. hier hoffentlich (Teil-)Antworten auf Fragen wie:
  - Worum geht's dabei, was ist die Essenz?
  - Vergleich zu imperativ oder OOP?
  - Warum aktuelles „Revival"?

- Folien aus dem Modul: „Programming Concepts and Paradigms" (PCP)
  - Untermenge davon gezeigt im Mindblast „Java 8: Lambdas, Streams & Co" vom 10. April 2014

PCP-Modul: Paradigmen & typische Sprache



➔ Java ist ein typisches Beispiel für viele „moderne“ Programmiersprachen, welche mehrere Paradigmen unterstützen (= Multi-Paradigmen-Sprache), insbesondere das objektorientierte und das funktionale

imperativ vs. deklarativ

- Imperative Programmierung: **WIE**
  - Problemlösung (Berechnungslogik) wird schrittweise in Befehlen beschrieben: Wie ist das Problem zu lösen?
  - Imperativ = unbedingte Anweisung
- Deklarative Programmierung: **WAS**
  - Berechnungslogik wird beschrieben ohne den Kontrollfluss zu definieren: Was sind die Fakten zum Problem?
  - Deklaration = Erklärung

http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg

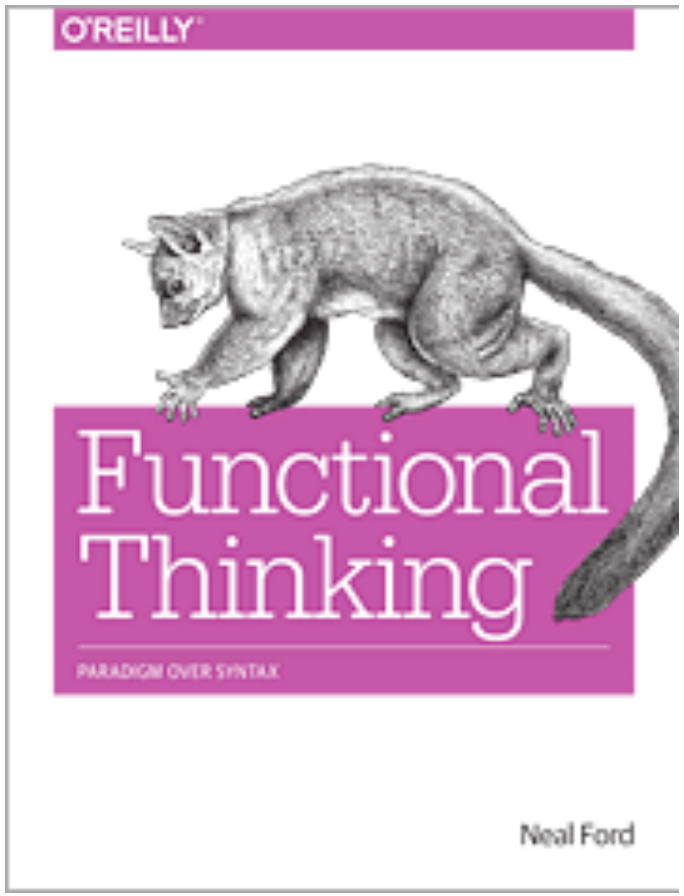
Functional Thinking

„Functional Thinking“

- Was macht funktionale Programmierung aus?
  - Unterschied zu imperativer Programmierung?

- Gutes Buch zum Thema:
  Functional Thinking -
  Paradigm Over Syntax,
  Neal Ford, O'Reilly Media, 2014
  - Viele gute Beispiele in Java (vor-8 und 8), Groovy, Clojure und Scala

  - Wir schauen ein paar Code-Beispiele & Ideen aus diesem Buch an...



Analogie „Garbage Collectors“ ☺

„Once garbage collection became mainstream, it simultaneously eliminated entire categories of hard to debug problems and allowed the runtime to manage a process that is complex and error prone for developers. Functional programming aims to do the same thing for the algorithms you write, allowing you to work at a higher level of abstraction while freeing the runtime to perform sophisticated optimizations.“

(Auszug aus dem Buch „Functional Thinking“, auch zu finden im Blog vom Autor Neal Ford unter http://thoughtworks.github.io/p2/issue08/lambas/)

http://devmix.org/site/wp-content/themes/striking/cache/images/garbage-collector-596x250.jpg

Imperative Verarbeitung

- Imperative Programmierung beschreibt einen Programmierungsstil modelliert aus einer Sequenz von Befehlen, welche einen Zustand modifizieren.

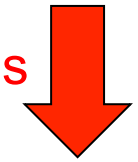
- Typisches Beispiel: eine traditionelle for-Schleife
  - Zuerst wird ein initialer Zustand eingerichtet, dann werden für jede Iteration dieser Schleife eine Reihe von Befehlen ausgeführt (welche i.A. den initialen Zustand verändern)

Beispiel-Problem: „The Company Process“

- Typisches Beispiel-Problem „Company Process“:
  - Gegeben: Liste X mit Namen (Strings)
  - Gesucht: String S, basierend auf Namen aus der Liste X, wobei:
    - Namen der Länge 1 werden entfernt
    - Namen werden umgewandelt, so dass sie mit Grossbuchstaben beginnen
    - Namen werden Komma-getrennt in einen String zusammengefasst

Eingabe: `{"sue", "x", "joe", "Daniela", "J"}`

Company Process



...wie würden Sie das programmieren?

Ausgabe: `"Sue,Joe,Daniela"`

Company Process – Imperative Lösung

Java-Implementierung, Paradigma imperativ (vor Java 8)

```
import java.util.List;

public class TheCompanyProcess {

    public String cleanNames(List<String> listOfNames) {
        StringBuilder result = new StringBuilder();
        for (int i = 0; i < listOfNames.size(); i++) {
            if (listOfNames.get(i).length() > 1) {
                result.append(capitalize(
                        listOfNames.get(i))).append(",");
            }
        }
        return result.substring(0, result.length() - 1).toString();
    }

    private String capitalize(String s) {
        return s.substring(0, 1).toUpperCase() +
                s.substring(1, s.length());
    }
}
```

filter (1)

transform (2)

convert (3)

Bemerkungen zur Imperativen Lösung

- Die imperative Programmierung fördert die Ausführung von Operationen in Schleifen
- Im Beispiel sehen wir konkret drei Operationskategorien:
  1. „filter“: eliminiert Namen der Länge 1
  2. „transform“: wandelt Listenelemente um (Grossschreibweise)
  3. „convert“: erstellt aus der Liste einen einzigen String
- In imperativen Sprachen muss der gleiche „low-level“ Mechanismus (nämlich Iteration über die Liste) für all diese drei Operationen angewandt werden

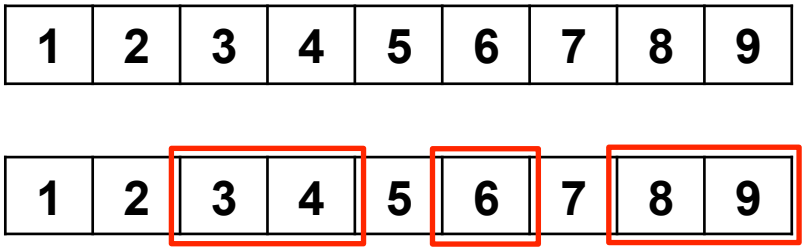
  ➔Funktionale Sprachen bieten spezifische Unterstützung für derartige Operationen ☺

Typische Bausteine funktionaler Programmierung

- Die erwähnten Operationskategorien („filter“, „transform“ & „convert“) sind typisch für funktionale Sprachen und Frameworks, und somit eine Art „Standard-Bausteine“ praktischer aller funktionalen Programmiersprachen
- Wir betrachten folgende drei Operationskategorien:
  - filter
  - map („transform“)
  - fold/reduce („convert“)

Funktionale Bausteine: Filter-Operationen

- Kreieren eine (potentiell) kleinere Liste basierend auf Benutzer-definierten Kriterien

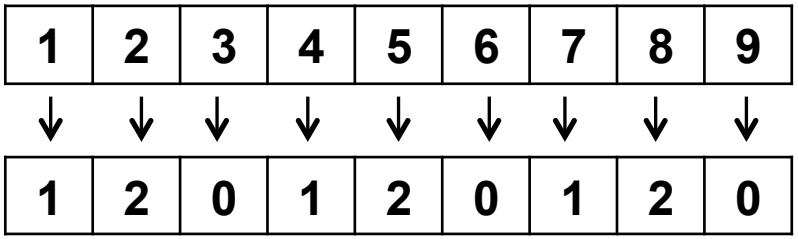


Quizfrage: Wie scheint hier die Filter-Operation definiert zu sein?

- `java.util.stream.Stream` unterstützt u.a. folgende Filter-Operationen: `distinct`, `filter`, `limit`, `peek`, `skip`
  - Andere Sprachen kennen weitere Filter-Operationen, wie z.B. findAll, partition, oder split

Funktionale Bausteine: Map-Operationen

- Transformieren eine Liste in eine neue Liste, indem sie eine Funktion auf jedes Element anwenden

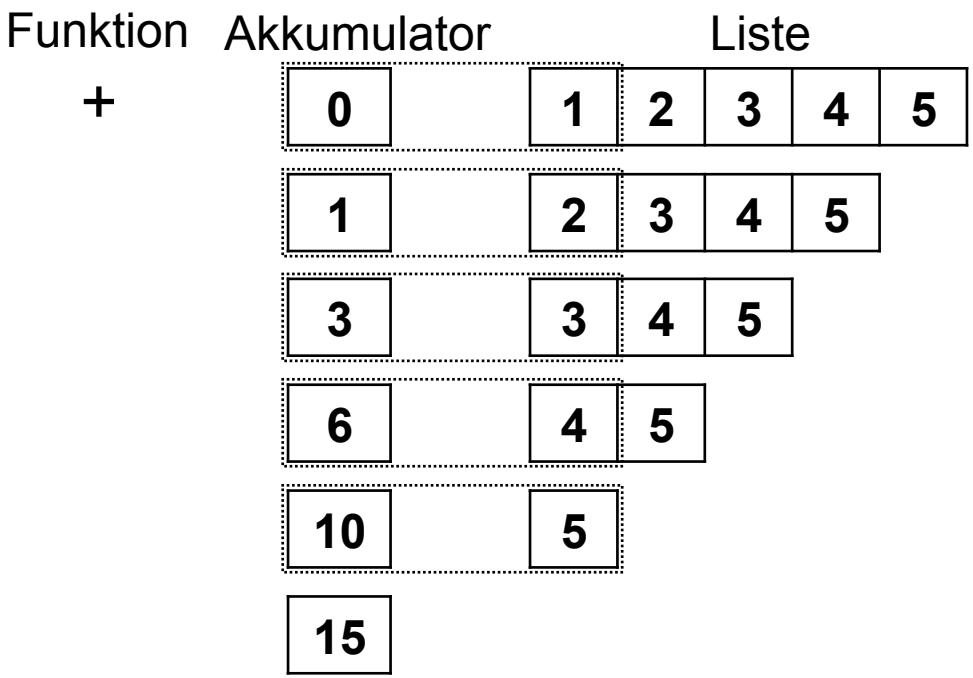


Quizfrage: Wie scheint hier die Map-Operation definiert zu sein?

- `java.util.stream.Stream` unterstützt u.a. folgende Map-Operationen: `flatmap`, `map`, `mapTo<Primitive>`
  - flatmap: Bildet Elemente von Stream auf neuen Stream ab. Erlaubt z.B. ein „Zusammenfassen“ von Stream of Collection-Objekten zu einem Stream mit allen Collection-Elementen

Funktionale Bausteine: Fold/Reduce-Operation

- Reduzieren eine Liste auf einen Wert, indem sie unter Verwendung von einem Akkumulator eine Funktion nach und nach auf alle Elemente anwenden



Fold/Reduce-Operationen

- `java.util.stream.Stream` unterstützt u.a folgende Fold/Reduce-Operationen: `reduce`, `collect`
  - `collect` kann in Java sehr vielseitig und mit verschiedensten Kollektoren (= Akkumulatoren) eingesetzt werden

- Andere Sprachen kennen weitere Fold/Reduce-Operationen wie: reduceLeft, reduceRight, foldLeft, foldRigth, inject
  - ...Right und ...Left gibt die Reihenfolge an, in welcher die Funktion auf die Elemente der Liste angewendet wird

Funktionale Verarbeitung

- Funktionale Programmierung beschreibt Programme durch Ausdrücke und Transformationen, modelliert mathematische Funktionen und versucht, veränderbaren Zustand zu vermeiden
- Funktionale Programmiersprachen kategorisieren Probleme anders als imperative Sprachen: Die drei vorhin beschriebenen Operationskategorien (filter, transform & convert) werden als Funktionen repräsentiert, welche die „low-level“-Transformation (konkret: Iteration) implementieren und darauf angewiesen sind, dass die Entwicklerin diese „low-level-Maschinerie“ individuell anpasst durch eine Funktion höherer Ordnung (higher order function), welche als Argumente Funktionen erhält.

Funktionaler Pseudo-Code

- Unser Beispiel vom „Company Process“ sieht also konzeptionell in funktionalem Pseudo-Code so aus:

```
ListOfNames
   -> filter(x.length > 1)
   -> transform(x.capitalize)        // aka “map”
   -> convert(x + “,” + y)           // aka “reduce”
```

➔ Funktionale Programmiersprachen erlauben eben genau das Modellieren dieser konzeptionellen Lösung ohne sich um gewisse Details kümmern zu müssen

Company Process – Funktionale Lösung

Java-Implementierung, Paradigma funktional (Java 8)

```
import java.util.List;

public class TheJava8CompanyProcess {

    public String cleanNames(List<String> names) {
        return names
                .stream()
                .filter(name -> name.length() > 1)
                .map(name -> capitalize(name))
                .collect(Collectors.joining(","));
    }

    private String capitalize(String e) {
        return e.substring(0, 1).toUpperCase() +
                e.substring(1, e.length());
    }
}
```

Company Process – Bsp. Null-Behandlung

- Falls in der Liste auch Null-Elemente sein können, kann zur Implementierung einfach ein weiteres Filter-Kriterium hinzugefügt werden:

```
return names
   .stream()
   .filter(name -> name != null)
   .filter(name -> name.length() > 1)
   .map(name -> capitalize(name))
   .collect(Collectors.joining(","));
}
```

Company Process – Bsp. Multi-Thread

- Falls die Ausführung von unserem „Company Process“ auf mehrere Threads verteilt werden soll, geht das in Java einfach mittels `parallelStream()`:

```
return names
   .parallelStream()
   .filter(name -> name.length() > 1)
   .map(name -> capitalize(name))
   .collect(Collectors.joining(","));
}
```

  – Bemerkung: Die Parallelisierung einer imperativen Lösung mit Hilfe von eigenen Threads benötigt i.A. viel mehr Code (und „Gehirnschmalz“) als diese recht elegante funktionale Lösung ☺

http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg

Java 8:
Lambdas & Streams

Lambda-Ausdrücke (Kurz: „Lambdas“)

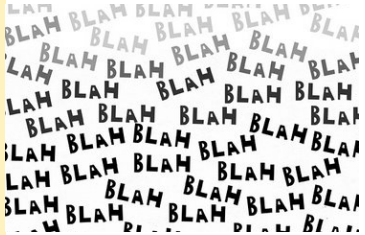
- Lambdas stammen von funktionalen Sprachen ab
  - „Anonyme Methode“ (Scheme: λ-Ausdruck)
    - „ad-hoc“ Implementierung von Funktionalität: „Code as Data“
  - **Datentyp für „Funktion“:** „first class citizen“
    - Funktionalität als Parameter/Rückgabewert herumreichen
  - Überlegen gegenüber (anonymen) inneren Klassen
    - Prägnante Syntax, weniger Code, lesbarer...

      ...“funktionaler“! ☺

(M)eine Motivation für Lambdas in Java...

- Bsp.: Event-Listener registrieren...
  - **Java < 8** („was wir bisher sagen mussten...“)

```
btn.setOnAction(new EventHandler<ActionEvent>() {
        @Override
        public void handle(ActionEvent event) {
                System.out.println("Hello World!");
        }
});
```



http://fatmohnscoop.files.wordpress.com/2010/06/blahblah.jpg

  - **ab Java 8** („...was wir eigentlich meinten“ ;-)

```
btn.setOnAction(event -> System.out.println("Hello World!“));
```

http://1.bp.blogspot.com/-bLPVKKfLakE/T-0JWedUwXI/AAAAAAAAADI/Clp7TnY048o/s1600/Short-and-to-the-point.jpg

Hinweis: Beide Beispiele benutzen für den Button die Klasse javafx.scene.control.ButtonBase

Code-Bsp. von http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#lambda-expressions-in-gui-applications

Typ von Lambdas = **functional interface (SAM)**

```
@FunctionalInterface                    // annotation is optional
interface Consumer<T> {
        public void accept(T a);
}

Consumer<Account> myLambda =
            (Account a) -> {if (a.balance() < limit) a.alert(); };
```

➔ Lambdas: Umwandlung in Instanzen von funktionalen Interfaces

- Funktionales Interface ≈ Interface mit 1 abstrakten Methode (SAM)
- Parametertyp(en), Return-Typ, Checked Exceptions müssen passen
- Irrelevant: Name vom Interface und Name der abstrakten Methode
- Umwandlung benötigt Type-Ableitung (type inference)!
  - Lambdas nur möglich, wenn Ziel-Typ vom Kontext ableitbar
    - Ableitung durch Zuweisung, Methodenaufruf, Return-Anweisung, Casts, ...

Seit Java 8: Package java.util.function

- Vorgegebene Sammlung von funktionalen Interfaces
- Verschieden Typen (Namenskonvention):
  - **Consumer**: Produziert kein Resultat (keine Rückgabe)
  - **Function**: Produziert Resultat von beliebigem Typ
  - **Operator**: Produziert Resultat vom Argument-Typ
  - **Supplier**: Produziert Resultat ohne Argument
  - **Predicate**: Produziert boolean-Resultat

➔ Wir halten uns an diese Namenskonvention, wenn wir eigene funktionale Interfaces deklarieren!

Was können/sollen wir mit Lambdas tun?

- Code Vereinfachung: direkt „Funktion(alität)“ mitgeben
  - Siehe einleitendes Beispiel „Listener-Registrierung“
    - Analog: Runnable-Lambda
  - Neue Möglichkeiten zur Bearbeitung von Collections
    - Iterable-Interface hat neue forEach-Methode
- Neue, auf Lambdas ausgelegte Datenstruktur **Streams**, ermöglichen verschiedenste „Aggregate Operations“ (= higher order functions) mit Hilfe von Lambdas

Lambda-Beispiel: Runnable

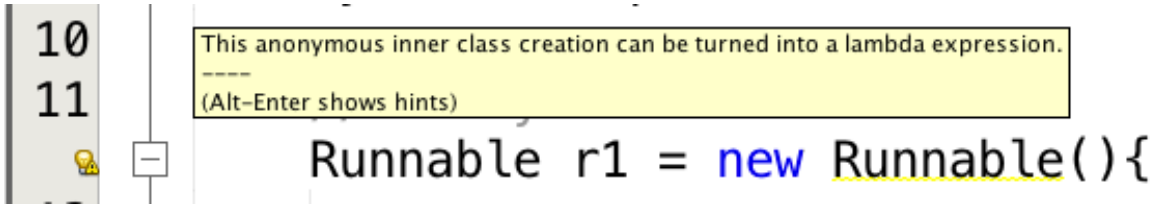
```
@FunctionalInterface
public interface Runnable
```

```
// Anonymous Runnable (pre Java 8 style)
Runnable r1 = new Runnable(){
        @Override
        public void run(){
                System.out.println("Hello world one!");
        }
};

// Lambda Runnable (Java 8 style)
Runnable r2 = () -> System.out.println("Hello world two!");

// Run em!
r1.run();
r2.run();
```

- Netter Hinweis von NetBeans:



Lambda-Beispiel: Iteration über Collections

- Das Interface Iterable<T> bietet neu folgende Methode:

```
default void forEach(Consumer<? super T> action)
```

  - Idee: Anwenden eines Consumers auf jedes Element
  - Collection<E>-Interface erbt von Iterable<T>
    - d.h. wir können neu mittels Consumer-Lambda über Collections iterieren
  - Schlüsselwort default garantiert Implementierung bei **allen** (d.h. auch „alten“) Iterable-Implementierungen
    - Rückwärtskompatibilität sichergestellt durch „defender method“ ☺

Lambda-Beispiel: Iterable.forEach(Consumer c)

```
String[] namesArray = {"Joe", "Sue", "Tim" };
List<String> names = Arrays.asList(namesArray);

// Iteration using a method reference
names.forEach(System.out::println);

// Same iteration using a lambda expression
names.forEach(s -> System.out.println(s));
```

- So können wir neu relativ einfach und elegant über Collections iterieren mit Hilfe von Iterable.forEach(...) + Lambda-Ausdrücken! ☺
  - Und brauchen keine Schleife mehr dazu, wie vor Java 8

Aggregate Operations (aka. Bulk Operations)

- Idee: Operationen (Funktionen) elegant auf Datenstrukturen anwenden können
  - Funktionale Sicht: Sequenzen + Operationen
    - Wie in Scheme: Funktion auf Liste anwenden
    - Aggregate Operations = **Funktionen höherer Ordnung**
  - OO-Sicht: Collections + interne Iteration
    - Wir müssen nicht selber iterieren, dies ist neu „intern“ möglich und wir können der Operation „Funktions-Argument“ mitgeben
- Kurz: for-each/filter/map/reduce/usw... für Java ☺
  - Und diese Operationen (= Funktionen höherer Ordnung) nehmen als Argument einen Lambda-Ausdruck entgegen

Beispiel-Operationen.: for-each, filter, map, reduce

- **for-each**: Funktionalität auf jedes Element anwenden

```
accounts.forEach(a -> a.addInterest());
```

- **filter**: Neue Sequenz erstellen aus den Resultaten, wenn auf jedes originale Element ein Filter angewandt wurde

```
accounts.filter(a -> a.balance() > 1_000_000 );
```

- **map**: Neue Sequenz erstellen aus den Resultaten, wenn auf jedes originale Element eine Abbildung angewandt wurde.

```
accounts.map(a -> a.balance());
```

- **reduce**: Produziert einzelnes Resultat aus allen Elementen

```
accounts.map(a -> a.balance())
        .reduce(0, (b1, b2) -> b1+b2);
```

intermediate / terminal Operations

Es gibt Stream-Operationen, welche...

- wieder einen Stream produzieren: filter(), map(), ...
  - **intermediate** (lazy)

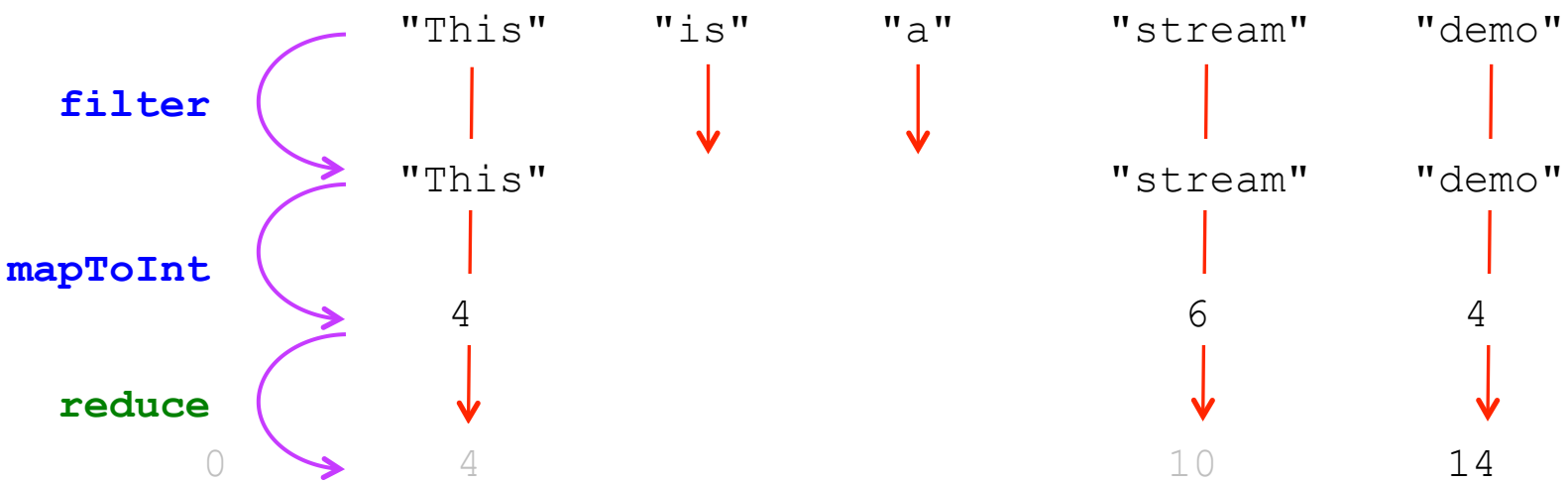
- etwas anderes tun: forEach(), reduce(),...
  - **terminal** (eager)

```
double sum = accountCol.stream()
                        .mapToDouble(a -> a.balance())
                        .reduce(0, (b1, b2) -> b1+b2);
```

Stream-Auswertung bei terminal-Operation

- intermediate-Streams werden nicht direkt ausgewertet
  - sondern erst wenn eine terminal-Operation aufgerufen wird

```
String[] txt = { "This", "is", "a", "stream", "demo"};
int i = Arrays.stream(txt).filter(s -> s.length() > 3)
                          .mapToInt(s -> s.length())
                          .reduce(0, (l1, l2) -> l1 + l2);
```



Code sieht so aus (eine Operation nach der andern)

wirklich ausgeführt (in einem Durchgang!)

http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg

Nebenläufigkeit mit Completable Futures

Interface java.util.concurrent.Future<V> (Seit Java 5)

- JavaDoc: „A Future represents the result of an asynchronous computation. Methods are provided to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation. The result can only be retrieved using method get when the computation has completed, blocking if necessary until it is ready. ...“
- Instanzen von Future werden oft / typischerweise im Zusammenspiel mit einer Instanz vom Interface Executor verwendet

Quellen: - http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html
- http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executor.html

Neu in Java 8: Klasse CompletableFuture<T>

- JavaDoc: „A Future that may be explicitly completed (setting its value and status), and may be used as a CompletionStage, supporting dependent functions and actions that trigger upon its completion. “
- Bietet interessante Möglichkeiten für asynchrone Ausführung von Code...
  - Methoden-Suffix: „-Async“, z.B.: `supplyAsync(...)`
  - **Erlaubt kompakte, ansprechende Syntax für asynchrone „funktionale“ Programmierung ☺**

Quelle: http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html

Demo-Code: CompletableFuture „in Action“ (1/2)

- Code-Beispiel, zeigt wie mit `ComputableFuture<T>` (und Lambda-Ausdrücken) asynchrone Aufgaben relativ kompakt implementiert werden kann
  - Relevant: Asynchroner Code blockiert NICHT Java-main-Thread! (Problem: Main-Thread = GUI-Thread, usw.)
- Blockierende Hilfs-Methode (ganz naiv implementiert):

```
private static final int THREE_SECONDS = 3000;
private static final int HALF_SECOND = 500;

public void doBlockingWait(long waitingTimeMillis) {
    try {
        Thread.sleep(waitingTimeMillis);
    } catch (InterruptedException ie) {
        // nop.
    }
}
```

Demo-Code: CompletableFuture „in Action“ (2/2)

```
private void doCompletableFutureDemo() {
    final CompletableFuture<String> longLastingTaskFuture
        = CompletableFuture.supplyAsync(() -> {
            doBlockingWait(THREE_SECONDS);
            System.out.print("1");
            return "42";
        });
    final CompletableFuture<String> evenLongerLastingTaskFuture
        = longLastingTaskFuture.thenApplyAsync((String s) -> {
            doBlockingWait(THREE_SECONDS);
            System.out.print("2");
            return "The answer is " + s;
        });
    evenLongerLastingTaskFuture.thenApplyAsync((String s) -> {
         doBlockingWait(THREE_SECONDS);
         System.out.print(s);
    });
    System.out.println("-> Now waiting for things to happen!");
        for (int i = 0; i < 20; i++) {
           System.out.print(".");
            doBlockingWait(HALF_SECOND);
        }
    System.out.println();
    System.out.println("-> Done.");
}
```

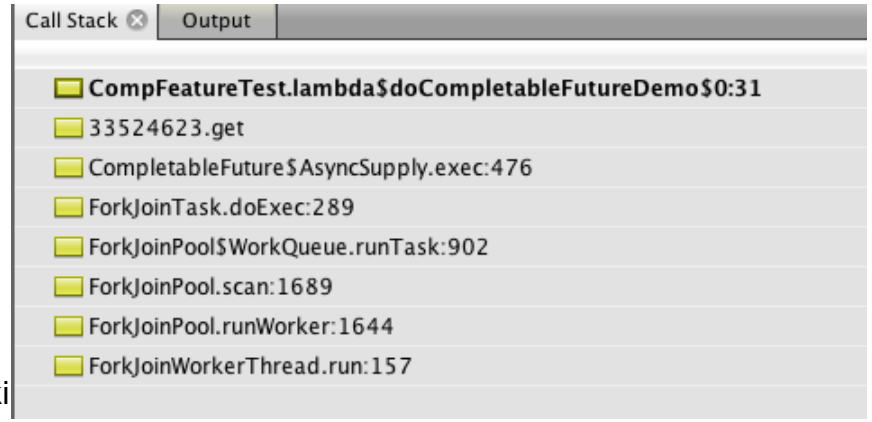
Produzierte Konsolenausgabe:

```
-> Now waiting for things to happen!
......1......2......The answer is 42..
-> Done.
```

Analyse vom Demo-Code

- Mit CompletableFutures (CF) kann also wie im Bsp. gesehen relativ einfach Code auf Hintergrund-Threads, d.h. asynchron zum main-Thread ausgeführt werden!
  - Wichtig zu sehen bei unserem Demo-Code: die 3 x `doBlockingWait(THREE_SECONDS)` werden NICHT auf dem main-Thread ausgeführt, sie blockieren also das GUI nicht. (Sonst würden in der Ausgabe zwischen `1`, `2` und `The answer is 42` keine Punkte stehen!)

```
-> Now waiting for things to happen!
......1......2......The answer is 42..
-> Done.
```



CF-Demo inkl. Debug-Mode

```
12      private static final int THREE_SECONDS = 3000;
13      private static final int HALF_SECOND = 500;
14
15      public static void main(String[] args) {
16          doCompletableFutureDemo();
17      }
18
19      public static void doBlockingWait(long waitingTimeMillis) {
20          try {
21              Thread.sleep(waitingTimeMillis); // my "long-lasting" ta
22          } catch (InterruptedException ie) {
23              // nop.
24          }
25      }
26
27      private static void doCompletableFutureDemo() {
            final CompletableFuture<String> longLastingTaskFuture
29                  = CompletableFuture.supplyAsync(() -> {
30                      doBlockingWait(THREE_SECONDS);
31                      System.out.print("1");
                        return "42";
```

```
-> Now waiting for things to happen!
......1......2......The answer is 42..
-> Done.
```

Call Stack | Output

- CompFeatureTest.lambda$doCompletableFutureDemo$0:31
- 33524623.get
- CompletableFuture$AsyncSupply.exec:476
- ForkJoinTask.doExec:289
- ForkJoinPool$WorkQueue.runTask:902
- ForkJoinPool.scan:1689
- ForkJoinPool.runWorker:1644
- ForkJoinWorkerThread.run:157

Verwendung von CompletableFutures

- In unserem Code-Beispiel verwenden wir asynchrone CompletableFutures (aCF) wie folgt:
  1. aCF erstellen & ausführen lassen
  2. danach weiteres aCF erstellen & ausführen lassen
  3. danach weiteren asynchronen Code ausführen lassen und abschliessen (d.h. kein neues aCF mehr erstellen)

- In der Folge schauen wir die dazu gebrauchten Methoden der Klasse CompletableFuture<T> an

1. Asynchrones CF erstellen & ausführen

```
final CompletableFuture<String> longLastingTaskFuture
    = CompletableFuture.supplyAsync(() -> {
        doBlockingWait(THREE_SECONDS);
        System.out.print("1");
        return "42";
    });
```

supplyAsync

```
public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier)
```

Returns a new CompletableFuture that is asynchronously completed by a task running in the `ForkJoinPool.commonPool()` with the value obtained by calling the given Supplier.

Type Parameters:

U - the function's return type

Parameters:

supplier - a function returning the value to be used to complete the returned CompletableFuture

Returns:

the new CompletableFuture

2. Danach weiteren asynchronen Code ausführen

```
final CompletableFuture<String> evenLongerLastingTaskFuture
        = longLastingTaskFuture.thenApplyAsync((String s) -> {
            doBlockingWait(THREE_SECONDS);
            System.out.print("2");
            return "The answer is " + s;
        });
```

thenApplyAsync

```
public <U> CompletableFuture<U> thenApplyAsync(Function<? super T,? extends U> fn)
```

Description copied from interface: CompletionStage

Returns a new CompletionStage that, when this stage completes normally, is executed using this stage's default asynchronous execution facility, with this stage's result as the argument to the supplied function. See the CompletionStage documentation for rules covering exceptional completion.

Specified by:

thenApplyAsync in interface CompletionStage<T>

Type Parameters:

U - the function's return type

Parameters:

fn - the function to use to compute the value of the returned CompletionStage

Returns:

the new CompletionStage

3. Danach mit weiterem async. Code abschliessen

```
evenLongerLastingTaskFuture.thenAcceptAsync((String s) -> {
        doBlockingWait(THREE_SECONDS);
        System.out.print(s);
});
```

thenAcceptAsync

```
public CompletableFuture<Void> thenAcceptAsync(Consumer<? super T> action)
```

Description copied from interface: CompletionStage

Returns a new CompletionStage that, when this stage completes normally, is executed using this stage's default asynchronous execution facility, with this stage's result as the argument to the supplied action. See the CompletionStage documentation for rules covering exceptional completion.

Specified by:

thenAcceptAsync in interface CompletionStage<T>

Parameters:

action - the action to perform before completing the returned CompletionStage

Returns:

the new CompletionStage

http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg

Fazit & Essenz

Höheren Stufe der Abstraktion: Vorteile

- Funktionale Programmierung bietet eine höhere Stufe der Abstraktion. Das hat folgende Vorteile:
  1. Ermutig und ermöglicht, Probleme anders zu kategorisieren und Gemeinsamkeiten zu erkennen
  2. Die Ausführung kann besser (automatisch) optimiert werden, z.B. durch Umordnung von Aufgaben
  3. Ermöglicht Lösungen, welche nicht möglich sind, wenn eine Entwicklerin tief in (herkömmlichen imperativen) Implementierungsdetails gefangen ist
     - z.B. Einfache Parallelisierung von stream-Operationen vs. eigene Implementierung von Multi-Thread-Code

Imperative vs. Functional Separation of Concerns

```
List<String> errors = new ArrayList<>();
int errorCount = 0;
File file = new File(fileName);
String line = file.readLine();
while (errorCount < 40 && line != null) {
    if (line.startsWith("ERROR")) {
        errors.add(line);
        errorCount++;
    }
    line = file.readLine();
}
```

```
List<String> errors =
    Files.lines(Paths.get(fileName))
        .filter(l -> l.startsWith("ERROR")
        .limit(40)
        .collect(toList());
```

Mario Fusco
@mariofusco

Follow

Imperative vs. Functional - Separation of Concerns

12:43 PM - 1 Mar 2015

905 704

Both algorithms do the same thing, they're probably equally fast and reasonable. Yet, one of the algorithms is much easier to write and read than the other. The difference lies in the fact that in imperative programming, different algorithmic requirements are spread throughout the code block, when in functional programming, each requirement has its own little line of code. Compare:

- Green: Error handling
- Blue: Stop criteria
- Red: IO operations
- Yellow: "Business logic"

⬅Anderes Code-Bsp.: **imperativ vs. funktional**

- Interessante Beobachtung in Bezug auf Separation of Concerns (SoC)
  - Siehe Text links unten...

...überzeugend, nicht? ☺

Quellen:
http://blog.jooq.org/2015/09/17/comparing-imperative-and-functional-algorithms-in-java-8/
https://twitter.com/mariofusco/status/571999216039542784/

Pure Functions

- Notwendige Eigenschaften für pure Funktionen:
  1. Produzieren keine (beobachtbaren) Seiteneffekte wie Zustandsänderungen, oder Ausgabe
  2. Produzieren für dieselben Argumente immer exakt dasselbe Resultat (sind also unabhängig von Zustand oder I/O)
- Es macht wenig Sinn, pure Funktionen zu verstecken (private), da sie keinen internen Zustand verändern
  - Pure Funktion kann nichts „kaputt“ machen
  - Pure Funktionen können problemlos wiederverwendet (Code re-use) und parallelisiert werden

Funktionale Programmierung: Zustandslosigkeit

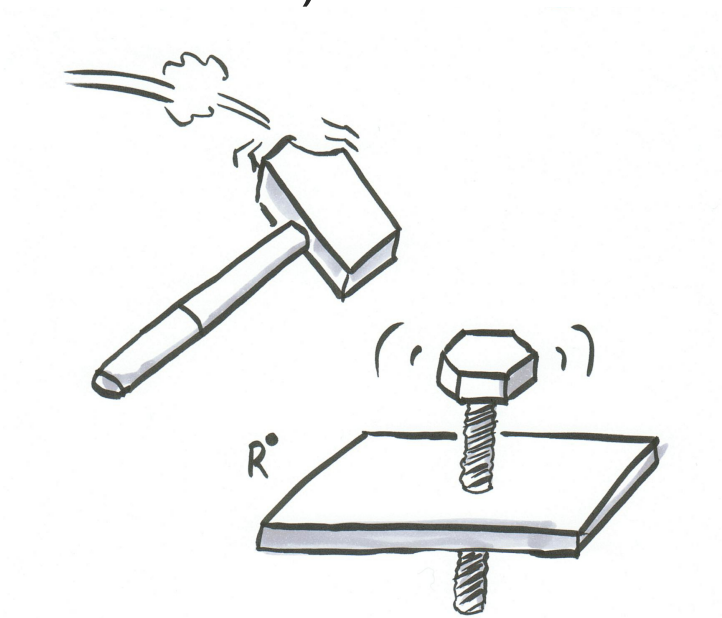
- Bei rein funktionaler Programmierung (wie z.B. in Scheme möglich) sind alle Funktionen pur
  - Es gibt keinen (globalen) Zustand
  - „Zustand“ existiert nur in Form vom Parametern und Rückgabewerten

- Zustandslosigkeit bietet diverse Vorteile, z.B. in Bezug auf Parallelisierbarkeit, Fehlertoleranz, und Robustheit

Abschluss: **Think Functional!** ☺

- Java bietet seit Version 8 besserer Unterstützung für diverse Konzepte der funktionaleren Programmierung
  - **Lambda-Ausdrücke** als „anonyme“ Funktionen mit „inline“-Möglichkeiten
  - **Streams** als „passende“ Datenstruktur für Funktionen höherer Ordnung (wie filter, map & reduce)

→Sprachmittel kennen & passend einsetzen!



http://blog.timoringlein.com/wp-content/uploads/2009/10/Hammer-und-Schraube.jpg