Funktionale Konzepte in Java 8

Programmierparadigmen

Johannes Brauer

1. Dezember 2019

Ziele

- Erkennen von Zusammenhängen zwischen verschiedenen Programmierparadigmen
- Grundverständnis für die Nutzung funktionaler Konzepte in Java

Warum?

Einstiegsbeispiele

Finde Hamburg

Statt einer imperativen Lösung

```
    boolean found = false;
    for(String city : cities) {
      if(city.equals("Hamburg")) {
        found = true;
        break;
      }
    }

System.out.println("Found Hamburg?:" + found);
```

besser

```
System.out.println("Found Hamburg?:" + cities.contains("Hamburg"));
```

Aufgabe aus Teil 1:

Schreiben Sie Java-Code, der alle Preise aus einer Liste, größer als 20 sind, um 10% rabattiert und aufsummiert.

```
final List<BigDecimal> prices = Arrays.asList(
  new BigDecimal("10"), new BigDecimal("30"), new BigDecimal("17"),
  new BigDecimal("20"), new BigDecimal("15"), new BigDecimal("18"),
  new BigDecimal("45"), new BigDecimal("12"));

// Lösung für Java < 8
```

```
BigDecimal totalOfDiscountedPrices = BigDecimal.ZERO;

for(BigDecimal price : prices) {
  if(price.compareTo(BigDecimal.valueOf(20)) > 0)
    totalOfDiscountedPrices =
       totalOfDiscountedPrices.add(price.multiply(BigDecimal.valueOf(0.9)));
}

// Lösung für Java >= 8
final BigDecimal totalOfDiscountedPrices =
  prices.stream()
        .filter(price -> price.compareTo(BigDecimal.valueOf(20)) > 0)
        .map(price -> price.multiply(BigDecimal.valueOf(0.9)))
        .reduce(BigDecimal.ZERO, BigDecimal::add);
```

Registrierung eines *event listeners* in *Swing*

- Swing ist eine Java-Bibliothek für die Programmierung graphischer Benutzungsoberflächen.
- Um herauszufinden, was ein Benutzer getan hat, werden event listeners registriert.
- Benutzung einer anonymen, inneren Klasse, um die Betätigung einer Schaltfläche mit einem Verhalten zu verknüpfen:

```
button.addActionListener(new ActionListener()
                         { public void actionPerformed(ActionEvent event) {
                             System.out.println("button clicked");
    }
});
```

- Erzeugung eines neuen Objekts, das das `ActionListener`-Interface implementiert.
- Das Interface besitzt eine einzige Methode, `actionPerformed`, die durch Betätigung der Schaltfläche aktiviert wird.
- Anonyme innere classes wurden eingeführt, um es Java-Programmierern zu erleichtern, Verhalten zu repräsentieren und als Argumente weitergeben zu können.
- Dennoch wird im Beispiel immer noch viel „Boilerplate“-Code für eine einzige Zeile relevanten Codes verlangt.
- Der Code ist auch irreführend. Die Absicht ist ja nicht ein Objekt zu übergeben, sondern Verhalten.
- Abhilfe: Benutzung eines Lambda-Ausdrucks:

```
button.addActionListener(event -> System.out.println("button clicked"));
```

- **Frage:** Wo findet hier Typinferenz statt?

Durchsetzung von Programmierregeln

- Datenbankzugriffe müssen in einem gegebenen Umfeld unter Einhaltung bestimmter Regeln programmiert werden; z. B. muss eine bestimmte Form des Loggings durchgeführt werden.
- Code-Sequenzen wie die folgende müssen ständig wiederholt werden:

```
Transaction transaction = getFromTransactionFactory();

//... operation to run within the transaction ...

checkProgressAndCommitOrRollbackTransaction();
UpdateAuditTrail();
```

- bessere Variante:

```
runWithinTransaction((Transaction transaction) -> {
  //... operation to run within the transaction ...
});
```

Hinweise auf Literatur

- Java
  - Learning Java Functional Programming [[Ree15]]
  - Functional Programming in Java [[Sub14]]
- C#
  - Real-World Functional Programming: With Examples in F# and C# [[PS10]]
  - Functional C#: Uncover the secrets of functional programming using C# and change the way you approach your applications forever [[Ang16]]

Erste Erkenntnisse

- Vermeidung von expliziter Mutation: Mutation ist fehleranfällig und hinderlich für Parallelarbeit.
  (Why shared mutable state is the root of all evil.)
- Der Code wird ausdrucksstärker: Eine Preisliste soll rabattiert (`map`), gefiltert (`filter`) und aufsummiert (`reduce`) werden.
- Code im funktionalen Stil ist prägnanter: In der Regel wird weniger Code benötigt. Dabei geht es fast weniger um das Schreiben (geschieht nur einmal) als um das Lesen und Pflegen.
- Der funktionale Stil erlaubt eher zu formulieren, **was** passieren soll, als Details des **Wie** notieren zu müssen.
- Der funktionale Stil steht nicht im Widerspruch zur Objektorientierung:
  - Der eigentliche Paradigmenwechsel geschieht vom imperativen zum deklarativen Stil.

- In Java 8 können funktionale und objektorientierte Stile wirksam kombiniert werden:
  - Objektorientierung kann für die Modellierung realer Weltobjekte, ihrer Zustände und Beziehungen weiter genutzt werden.
  - Zusätzlich kann Verhalten und Datenverarbeitung durch Funktionen komponiert werden.
  - Das Verhalten muss nicht mehr zwingend an Objekte gebunden werden.

Sprachkonzepte

Lambda-Ausdrücke

- einige Beispiele bereits gesehen
- Lambda-Ausdrücke sind
  - anonyme Funktionen
  - akzeptieren Argumente
  - liefern Resultate
  - können auf außerhalb des Ausdrucks definierte zugreifen

Formen:

Lambda-Ausdruck	Bedeutung
()->System.out.println()	keine Parameter, erzeugt Effekt
x->System.out.println(x)	ein Parameter
x->2*x	ein Parameter, liefert Resultat
(x,y)->x+y	zwei Parameter, werden addiert
x->{int y = 2 x; return y;}	Code-Block aus mehreren Anweisungen

Hinweise zur Benutzung

siehe unten

Default-Methoden

- Eine Default-Methode ist eine Interface-Methode mit Implementierung.
- kein innerer Zusammenhang zu Lambda-Ausdrücken
- Einer Klasse, die ein Interface implementiert, stehen deren Default-Methoden zur Verfügung.

simples Beispiel

- Definition von Default-Methoden:

```
public interface Computable {
    public int compute();
    public default int doubleNumber(int num) {
        return 2*num;
    }
    public default int negateNumber(int num) {
       return -1*num;
} }
```

- Benutzung:

  1. Erzeugen einer Klasse, die das Interface (`Computable`) implementiert
  2. Erzeugen eines Exemplars der Klasse und Aktivierung der Default-Methode

  ```
  public class ComputeImpl implements Computable {
            @Override
            public int compute() {
   return 1; }

  ComputeImplcomputeImpl = new ComputeImpl();
     System.out.println(computeImpl.doubleNumber(2));
  ```

- Eine Implementierung der Methode `doubleNumber` ist in der Klasse `ComputeImpl` nicht erforderlich.
- Überschreiben ist jedoch zulässig.

Nutzen

- In Java 8 dürfen Interfaces statische Methoden und Default-Methoden besitzen.
- Hauptmotivation: Evolution von Interfaces; Interfaces aus der vor Java-8-Epoche können um Funktionen erweitert werden, ohne existierenden Code zu beeinflussen.
- Beispiel: Die `forEach`-Methode wurde als Default-Methode dem Interface `Iterable` aus dem `java.lang`-Package hinzugefügt.
- `forEach` akzeptiert einen Lambda-Ausdruck als Argument, der zu der `accept`-Methode des `Consumer`-Interface passt und führt ihn für jedes Element des zugrundeliegenden Behälters aus.
- Die Klasse `ArrayList` implementiert das `Iterable`-Interface:

  ```
  ArrayList<String> list = new ArrayList<>();
  list.add("Kartoffel"); list.add("Bohne"); list.add("Möhre");
  list.forEach(f->System.out.println(f));
  ```

Unterschied zu abstrakten Klassen

- Abstrakte Klassen können Exemplarvariablen besitzen, Interfaces nicht.
- Klassen können Verhalten (Default-Methoden) von mehreren Interfaces erben, aber nur von einer abstrakten Klasse.
- Die Regel, möglichst Interfaces gegenüber abstrakten Klassen zu bevorzugen, ist dank der Default-Methoden leichter zu befolgen.

Funktionale Interfaces

- Ein funktionales Interface
  - besitzt genau eine abstrakte Methode (Beispiel `Computable` s. .o),
  - darf Default-Methoden und
  - statische Methoden besitzen,
  - erleichtert die Nutzung von Lambda-Ausdrücken

Beispiele vordefinierter funktionaler Interfaces

`Consumer<T>` **Beschreibung** steht für eine Operation mit einem Parameter und ohne Reslutat, d. h. sinnvoll nur für Operationen mit Seiteneffekten

Abstrakte Methode `accept()`

Default-Methoden `andThen()`

Benutzungsbeispiel als Parameter der `forEach=Methode`

`Supplier<T>` **Beschreibung** eine Fabrik, die ein neues oder ein existierndes Objekt liefert

Abstrakte Methode `get()`

Default-Methoden –

Benutzungsbeispiele für die Erzeugung verzögerter, unendlicher Streams und als Parameter für die `orElseGet()`-Methode der Klasse `Optional`

`Predicate<T>` **Beschreibung** erlaubt die Prüfung, ob ein Argument eine Bedingung erfüllt

Abstrakte Methode `test()`

Default-Methoden `and()`, `negate()`, `or()`

Benutzungsbeispiel als Parameter für `Stream`-Methoden wie `filter()` und `anyMatch()`

`Function<T>` **Beschreibung** steht für eine Operation mit einem Argument und einem Resultat

Abstrakte Methode `apply()`

Default-Methoden `andThen(), compose()`

Benutzungsbeispiel als Parameter für die `Stream`-Methode `map()`

Methoden- und Konstruktorverweise

- ... eröffnen die Möglichkeit, an Stellen, wo Lambda-Ausdrücke erwartet werden, benannte Methoden oder Konstruktoren einzusetzen.
- Beispiele:
  - Die Sequenz

    ```
    Stream<Integer> stream = Stream.of(12, 52, 32, 74, 25);
           Stream
                .map(x -> x * 2)
                .forEach(x ->System.out.println(x));
    ```

    kann ersetzt werden durch

    ```
    Stream<Integer> stream = Stream.of(12, 52, 32, 74, 25);
    Stream
        .map(x -> x * 2)
        .forEach(System.out::println);
    ```

  - Zwei Methodenverweise

    ```
    stream.map(Math::sin.forEach(System.out::println);
    ```

Behälter

- Das bekannte `Collection`-Interface wurde um Methoden erweitert, die `Stream`-Objekte erzeugen:

  `stream` erzeugt einen sequentiel zu verarbeitenden `Stream`

  `parallelStream` erzeugt einen `Stream`, der parallel verarbeitet werden kann

- Beispiel für die Anwendung von `stream()` (Das `List`-Interface erweitert das `Collection`-Interface.)

  ```
  String names[] = {"Karl", "Rosa", "Klara", "Che"};
         List<String> list = Arrays.asList(names);
         Stream<String> stream = list.stream();
         stream.forEach(name ->System.out.println(name + " - "
             + name.length()));
  ```

- Hinweis: Da das `Collection`-Interface vom `Iterable`-Interface erbt, kann auf `List`-Objekte auch die `forEach`-Methode angewendet werden:

  ```
  list.forEach(name ->System.out.println(name + " - "
      + name.length()));
  ```

Zusammenspiel der Konzepte

Evolution der Iteration

- Die Mehrzahl aller Datenverarbeitungsprozesse beinhalten die Iteration über Behälter.

- Zum Beispiel über eine Liste mit Namen:

```
final List<String> studierende =
   Arrays.asList("Niklas", "Bogdan", "Markus", "Jonah", "Felina", "Finn");
```

- der „klassische" imperative Weg, wortreich und fehleranfällig:

```
for(int i = 0; i < studierende.size(); i++)
   { System.out.println(studierende.get(i));}
```

- schon besser, benutzt intern das `Iterator`-Interface mit den Methoden `hasNext()` und `next()`:

```
for(String name : studierende) { System.out.println(name);
}
```

- Beide Varianten benutzen externe Iteratoren.
- Dabei werden das Was und das Wie vermengt.
- Gründe für den Umstieg auf eine funktionale Variante:
  1. Die Schleifenkonstrukte sind inhärent sequentiell.
  2. Die Lösungen sind nicht polymorph. Die Liste wird an `for` „übergeben", anstatt sie als Argument an eine Methode weiterzureichen, die die Aufgabe erledigt.
  3. Damit ist das Abstraktionsniveau gering. Die Details der Iteration sollten ein für alle mal Bibliotheksroutinen überlassen werden.
- Das leistet die bereits behandelte `forEach`-Methode:

```
studierende.forEach(new Consumer<String>()
   { public void accept(final String name) {System.out.println(name); }
});
```

- der letzte Iterationsschritt: Ersetzen der anonymen inneren Klasse durch einen Lambda-Ausdruck und ohne explizite Typangabe:

```
studierende.forEach((name) -> System.out.println(name));
```

Transformation von Listen

- Angenommen, die Elemente der Namensliste sollen in Großbuchstaben verwandelt werden.
- Auf eine Lösung mit Mutation soll/muss verzichtet, weil
  - `String`-Objekte nicht änderbar sind,
  - die Originalliste erhalten bleiben soll und/oder
  - eine mit `Arrays.asList()` erzeugte Liste nicht änderbar ist.
- Eine Lösung unter Verwendung von `forEach` könnte so aussehen:

```
final List<String> uppercaseNames = new ArrayList<String>();
studierende.forEach(name -> uppercaseNames.add(name.toUpperCase()));
System.out.println(uppercaseNames);
```

- Der Vorteil von `forEach`: kann auf alle Behälterklassen angewendet werden.
- Der Nachteil: `forEach` liefert keinen Rückgabewert.
- Folge: leere Liste muss erzeugt werden, der die Elemente mit `add` hinzugefügt werden müsssen.
- Diese imperativen Überbleibsel können durch Verwendung der `map`-Methode beseitigt werden.
- Diese – wie andere klassiche Funktionen höherer Ordnung auch – ist nur auf Streams anwendbar.

```
studierende.stream()
           .map(name -> name.toUpperCase())
           .forEach(name -> System.out.print(name + " "));

// oder

studierende.stream()
           .map(String::toUpperCase)
           .forEach(name -> System.out.print(name + " "));
```

Weiterverarbeitung von Streams als Listen

- Die Sequenz

```
final List<String> startsWithF =
   studierende.stream()
              .filter(name -> name.startsWith("F"))
              .collect(Collectors.toList());
```

  liefert eine Liste aller Namen, die mit „F“ beginnen.
- Das `Stream`-Objekt, das die `filter`-Methode (wie auch `map`) als Resultat liefert, kann mit der `Stream`-Methode `collect` und der `toList`-Methode der Klasse `Collectors` in eine Liste verwandelt werden.
- Das ermöglicht z. B. folgende Ausgabe

```
System.out.println(String.format("Found %d names", startsWithF.size()));
```

Wiederverwendung von Lambda-Ausdrücken

- Ein Lambda-Ausdruck kann an eine Variable vom Typ `Predicate<T>` gebunden werden:

```
final Predicate<String> startsWithF = name -> name.startsWith("F");
final long countStudierendeStartF =
    studierende.stream()
         .filter(startsWithF).count();
```

- Um die Funktion für variable Anfangsbuchstaben umzugestalten, kann eine statische Methode benutzt werden:

  ```
  public static Predicate<String> checkIfStartsWith(final String letter) {
      return name -> name.startsWith(letter);
  }
  ```

  Benutzung:

  ```
  final long countStudierendeStartN =
     studierende.stream().filter(checkIfStartsWith("N")).count();
  ```

- Wenn statische Methode vermieden werden sollen, kann das `Function`-Interface verwendet werden:

  ```
  final Function<String, Predicate<String>> startsWithLetter =
    (String letter) -> {
      Predicate<String> checkStarts = (String name) -> name.startsWith(letter);
      return checkStarts;
  };
  ```

  Die Variable `startsWithLetter` wird an eine Funktion gebunden, die ein `String` erwartet und ein `Predicate` zurückgibt. Die Funktion ist äquivalent zur oben gezeigten statischen Methode. Die folgende Abkürzung ist erlaubt:

  ```
  final Function<String, Predicate<String>> startsWithLetter =
    (String letter) -> (String name) -> name.startsWith(letter);
  ```

  oder unter Verwendung von Java's Typinferenz:

  ```
  final Function<String, Predicate<String>> startsWithLetter =
    letter ->  name -> name.startsWith(letter);
  ```

- Benutzung der Funktion `startsWithLetter`

  ```
  final long countStudierendeStartN =
    studierende.stream()
           .filter(startsWithLetter.apply("N")).count();

  final long countStudierendeStartB =
    studierende.stream()
           .filter(startsWithLetter.apply("B")).count();
  ```

Flüssige Schnittstellen mit Lambda-Ausdrücken

- *Flüssige Schnittstelle* ist die Übersetzung des englischen Fachbegriffs *fluent interface*.
- Der Begriff ist wohl von Martin Fowler geprägt worden.
- Sie stellen eine Gestaltungsform für APIs dar, die zu gut lesbarem Code führt.

Beispiel

Das hier gezeigte Anwendungsbeispiel ist [Sub14] entnommen.

- Die herkömmliche Schnittstelle einer `Mailer`-Klasse, gekennzeichnet durch viele `void`-Methoden.

```
public class Mailer {
  public void from(final String address) { /*... */ }
  public void to(final String address)   { /*... */ }
  public void subject(final String line) { /*... */ }
  public void body(final String message) { /*... */ }
  public void send() { System.out.println("sending..."); }

//  könnte so benutzt werden:

Mailer mailer = new Mailer();
mailer.from("build@agiledeveloper.com");
mailer.to("venkats@agiledeveloper.com");
mailer.subject("build notification");
mailer.body("...your code sucks...");
mailer.send();
```

- Nachteile dieser Lösung
  - geschwätzig: `mailer` muss dauernd wiederholt werden
  - Verbleib des `Mailer`-Objekts unklar
- Erste Maßnahme: Verkettung von Methodenaufrufen

```
public class MailBuilder {
  public MailBuilder from(final String address) { /*... */; return this; }
  public MailBuilder to(final String address)   { /*... */; return this; }
  public MailBuilder subject(final String line) { /*... */; return this; }
  public MailBuilder body(final String message) { /*... */; return this; }
  public void send() { System.out.println("sending..."); }

  //... wird so benutzt:
  public static void main(final String[] args) {
    new MailBuilder()
      .from("build@agiledeveloper.com")
      .to("venkats@agiledeveloper.com")
```

```
        .subject("build notification")
        .body("...it sucks less...")
        .send();
    }
}
```

- Nutzung des Konstruktors stört noch den Lesefluss.
- Lebensdauer des `MailBuilder`-Objekts weiterhin problematisch.
- Zweite Maßnahme: Einsatz von Lambda-Ausdrücken und Privatisierung des Konstruktors

```
public class FluentMailer {
  private FluentMailer() {}

  public FluentMailer from(final String address) { /*... */; return this; }
  public FluentMailer to(final String address)   { /*... */; return this; }
  public FluentMailer subject(final String line) { /*... */; return this; }
  public FluentMailer body(final String message) { /*... */; return this; }

  public static void send(final Consumer<FluentMailer> block) {
    final FluentMailer mailer = new FluentMailer();
    block.accept(mailer);
    System.out.println("sending...");
  }
```

- Externe Erzeugung eines `FluentMailer`-Objekts nicht mehr möglich.
- `send`-Methode ist statisch und erwartet ein `Consumer`-Objekt (`block`) als Argument.
- Benutzung: Anstatt ein Exemplar zu erzeugen, wird die `send`-Methode aufgerufen.
- Diese erzeugt ein Exemplar und übergibt es an den Block:

```
public static void main(final String[] args) {
  FluentMailer.send(mailer ->
    mailer.from("build@agiledeveloper.com")
          .to("venkats@agiledeveloper.com")
          .subject("build notification")
          .body("...much better..."));
}
```

Fazit

- Nicht alles verwerfen, was Sie über Objektorientierung gelernt haben, aber nutzen Sie die funktionalen Möglichkeiten von Java 8:
  - Schreiben Sie deklarativen Code!

- Vermeiden Sie Zustandsänderungen!
- Vermeiden Sie Seiteneffekte!
- Bevorzugen Sie Ausdrücke gegenüber Anweisungen!
- Arbeiten Sie mit Funktionen höherer Ordnung!

Literaturverzeichnis

[Ang16] Wisnu Anggoro. *Functional C: Uncover the secrets of functional programming using C and change the way you approach your applications forever.* Packt Publishing, 2016.

[PS10] Tomas Petricek and Jon Skeet. *Real-World Functional Programming: With Examples in F and C.* Manning Publications, 2010.

[Ree15] Richard M Reese. *Learning Java Functional Programming.* Packt Publishing, 2015.

[Sub14] Venkat Subramaniam. *Functional Programming in Java: Harnessing the Power Of Java 8 Lambda Expressions.* Pragmatic Bookshelf, 2014.