sidion

CLEAN CODE UND LAMBDAS – WAS WÜRDE UNCLE BOB WOHL DAZU SAGEN

21.06.2018

Matthias Koch

Zuhören. Analysieren. Beraten.

Agenda

1. Clean Lambdas
2. Lazy Logging
3. Lambdas @SonarQube
4. Functional Lenses – Dafür wurden Lambdas gemacht!

sidion

Agenda

1. **Clean Lambdas**
2. Lazy Logging
3. Lambdas @SonarQube
4. Functional Lenses – Dafür wurden Lambdas gemacht!

Zuhören. Analysieren. Beraten.

Ausgangslage

- Robert C. Martin veröffentlichte sein Standardwerk „Clean Code“ 2007.
- Lambdas finden Eingang in Java 8 mit Veröffentlichungstermin 18. März 2014.
- Was sagt die Clean Code Developer Initiative dazu? Fehlanzeige!

Was machen eigentlich Lambdas?

- Implementieren Functional / SAM-Type Interfaces

```
@FunctionalInterface
public interface <SAMTypeInterface> {

    public abstract T <SingleAbstract Method>(U) ;

    // Beliebig viele default und statische Methoden
}
```

Wo können Lambdas eigentlich auftreten?

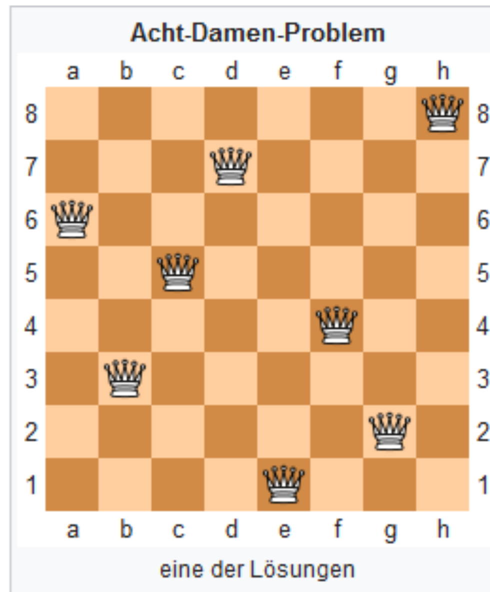
- Auf der rechten Seite einer Zuweisung an ein Functional Interface
- In einer Methode mit Rückgabewert vom Typ Functional Interface in der return-Anweisung
- Beim Methodenaufruf als Argument an einen Parameter vom Typ Functional Interface
- In einem Cast-Ausdruck
- Lambdas können anonyme innere Klassen ersetzen.

Beispiel

```
public static Stream<List<Pos>> famousFunction(int n) {

  return IntStream.rangeClosed(1, n)
    .mapToObj(i -> i)
    .sorted(Comparator.reverseOrder())
    .reduce( Stream.of(new ArrayList<Pos>()), (r, i) -> r.flatMap(qs -> IntStream.rangeClosed(1, n).mapToObj( j -> new Pos(j, i))
    .flatMap(p -> (qs.stream().allMatch( q -> (q.x != p.x && q.y != p.y && abs(q.x -p.x) != abs(q.y - p.y))) ?
    Stream.of(Stream.concat(qs.stream(), Stream.of(p))
    .collect(Collectors.toList())) : Stream.<List<Pos>> empty()))), Stream::concat);
}
```

Auflösung 1/2



Quelle: https://de.wikipedia.org/wiki/Damenproblem

Auflösung 2/2

```
public static Stream<List<Pos>> nQueens(int n) {

  return IntStream.rangeClosed(1, n)
    .mapToObj(i -> i)
    .sorted(Comparator.reverseOrder())
    .reduce( Stream.of(new ArrayList<Pos>()), (r, i) -> r.flatMap(qs -> IntStream.rangeClosed(1, n).mapToObj( j -> new Pos(j, i))
    .flatMap(p -> (qs.stream().allMatch( q -> (q.x != p.x && q.y != p.y && abs(q.x -p.x) != abs(q.y - p.y))) ?
    Stream.of(Stream.concat(qs.stream(), Stream.of(p))
    .collect(Collectors.toList())) : Stream.<List<Pos>> empty()))), Stream::concat);
}
```

Quelle: https://codereview.stackexchange.com/questions/78963/n-queens-functional

Ist kürzer wirklich besser?

```
Comparator<Integer> compare = (a, b) ->

      return  !Integer.compare(a, b);
```

```
Comparator<Integer> compare = compareInDescendigOrder(digit1, digit2);

Comparator<Integer> compareInDescendigOrder(digit1, digit2) {

       return (digit1, digit2) -> !Integer.compare(digit1, digit2);
}
```

Sprechende Bezeichner

- Namen sollen dem Code Kontext geben.
- Auch durch Lambdas darf kein Kontext wegfallen!
- Testbarkeit
- Wiederverwendbarkeit (DRY)

```
Comparator<Integer> compare = compareInDescendigOrder(digit1, digit2);

Comparator<Integer> compareInDescendigOrder(digit1, digit2) {

        return (digit1, digit2) -> !Integer.compare(digit1, digit2);
}
```

Komplexität 1/2

- Verschachtelte Lambdas sind meist schwer verständlich.
- (a, b) -> a + b
- Currying: f: a x b -> c äquivalent f‘: a -> (b -> c)
- (a,b) -> a + b äquivalent a-> (b -> a +b)

Komplexität 2/2

- Lambdas sollten kurz, prägnant und selbsterklärend sein.
- Parametertypen können vom Compiler deduziert werden und sollten daher weggelassen werden.
- Keine Klammern bei einem einzelnen Parameter
- Methodenreferenzen sollten verwendet werden, falls dies keinen Kontext vom Code nimmt.

Checked Exceptions 1/2

```
try {
    Predicate<Path> isEmptyFile = path -> Files.size( path ) == 0;
} catch ( IOException e ) { … }
```

Nee!

sondern nur:

```
Predicate<Path> isEmptyFile = path -> {
    try {
        return Files.size( path ) == 0;
    } catch ( IOException e ) {
        return false;
    }
};
```

Checked Exceptions 2/2

```
Public static boolean twoPredicates {
   Predicate<P1> firstPredicate = p1 -> {
      try { …
      } catch (Exception e ) { return false; }
   };

   Predicate<P2> secondPredicate = p2 -> {
      try { …
      } catch (Exception e ) { return false; }
   };
   return true;
 }
```

- Kann dazu führen, dass mehrere try/catch Blöcke in einer Methode auftreten.
- Clean Code sagt aber ein try/catch pro Methode!

Agenda

1. Clean Lambdas
2. **Lazy Logging**
3. Lambdas @SonarQube
4. Functional Lenses – Dafür wurden Lambdas gemacht!

Was macht eigentlich Lazy Logging?

- Seit Version 2.4 gibt es in log4j Java 8 Lazy Logging.
- Vermeidung von Guards um die logging Statements
- Keine Performanceeinbußen durch die Guards oder ggf. durch irrtümlich ausgeführten Debug-Code
- Wesentlich sauberer und kürzerer Code

Logging Code

- Gefährlich:

```
Logger.debug("Calling time consuming method {}", timeConsuming());
```

- Sicher, aber zu viel Code:

```
if (Logger.isDebugEnabled()) {
    Logger.debug("Calling time consuming method {}", timeConsuming());
}
```

- Verrückte Idee:

```
Logger.debug("Calling time consuming method {}", () -> timeConsuming());
```

Agenda

1. Clean Lambdas
2. Lazy Logging
3. **Lambdas @SonarQube**
4. Functional Lenses – Dafür wurden Lambdas gemacht!

Code im Sonarqube

CLEA N_LAMBDAS

23. April 2018 10:50 Version 1.0

Issues Measures Code Administration

Quality Gate Passed

Bugs & Vulnerabilities

Leak Period: since previous version
started vor 2 Stunden

0 A Bugs

0 A Vulnerabilities

0 New Bugs

0 New Vulnerabilities

Code Smells

0 A Debt

started vor 2 Stunden

0 Code Smells

0 New Debt

0 New Code Smells

Duplications

0.0% Duplications

0 Duplicated Blocks

— Duplications on New Code

177 Lines of Code

Java 177

Quality Gate
(Default) SonarQube way

Quality Profiles
(Java) Sonar way

Key
CLEAN_LAMBDAS

Events All

Quality Gate: Green (was Red)
23. April 2018

Version: 1.0
23. April 2018

Quality Gate: Red (was Green)
23. April 2018

SonarQube Regeln

@FunctionalInterface annotation should be used to flag Single Abstract Method interfaces

CodeSmell

Major

@Deprecated: Zu viele “false positives”

Noncompliant Code Example

```
public interface Changeable<T> {
    public void change(T o);
}
```

Compliant Solution

```
@FunctionalInterface
public interface Changeable<T> {
    public void change(T o);
}
```

SonarQube Regeln 2

Anonymous inner classes containing only one method should become lambdas

Code Smell

Major

Noncompliant Code Example

```
myCollection.map(new Mapper<String,String>() {
    public String map(String input) {
        return new StringBuilder(input).reverse().toString();
    }
});
```

Compliant Solution

```
myCollection.map(element -> new StringBuilder(element).reverse().toString());
```

SonarQube Regeln 3

Lambdas and anonymous classes should not have too many lines.

Code Smell
Major

Maximum allowed lines in an anonymous class/lambda
Default Value:
20

SonarQube Regeln 4

Lambdas should be replaced with method references:

Code Smell

Minor

Noncompliant Code Example

```
List list = new ArrayList();
list.add(0);
list.add(1);
list.forEach(n -> { System.out.println(n); });
```

Compliant Solution

```
List list = new ArrayList();
list.add(0);
list.add(1);
list.forEach(System.out::println);
```

SonarQube Regeln 5

Lamdbas containing only one statement should not nest this statement in a block:

Code Smell

Major

Noncompliant Code Example

```
x -> {System.out.println(x+1);}
(a, b) -> { return a+b; }
```

Compliant Solution

```
x -> System.out.println(x+1)
(a, b) -> a+b //For return statement, the return keyword should also be dropped
```

SonarQube Regeln 6

Parentheses should be removed from a single lambda input parameter when its type is inferred:

Code Smell

Minor

Noncompliant Code Example

```
(x) -> x * 2
```

Compliant Solution

```
x -> x * 2
```

SonarQube Regeln 7

Types should be used in lambdas:

Code Smell

Major

Noncompliant Code Example

```
Arrays.sort(rosterAsArray, (a, b) -> { // Noncompliant return
a.getBirthday().compareTo(b.getBirthday()); } );
```

Compliant Solution

```
Arrays.sort(rosterAsArray, (Person a, Person b) -> { return
a.getBirthday().compareTo(b.getBirthday()); } );
```

Exceptions

When the lambda has one or two parameters and does not have a block this rule will not fire up an issue as things are considered more readable in those cases.

```
stream.map((a, b) -> a.length); // compliant
```

Lambdas @SonarQube Fazit

- Regeln sind eher einfach „gestrickt“.
- Bei Verwendung eines einheitlichen Rulesets sehen Lambdas auch gleichartig aus.
- Komplexität ist ein zentraler Punkt und SonarQube spuckt Metriken zur Komplexität aus.
- Verleitet zu Aussagen, wie:
  „Wir haben 30% Code gespart.“
  „Unser Code ist wartbarer geworden!“
- Wenn duplizierter Code signifikant reduziert werde konnte, dann stimmt das auch.
- Aber was ist, wenn dies auf Kosten der Bezeichner geht?

Agenda

1. Clean Lambdas
2. Lazy Logging
3. Lambdas @SonarQube
4. **Functional Lenses – Dafür wurden Lambdas gemacht!**

Functional Lenses / Haskell Lenses

- Damit lassen sich Immutables modifizieren – ohne das die Immutables geändert werden müssen.
- Immutable Klassen sollen alle unverändert bleiben.
- Insbesondere keine Setter in diesen Klassen.
- Da das Objekt ggf. groß und tief geschachtelt sein kann, kommt eine komplette Neuerzeugung nicht in Frage.
- Keine Verwendung von Reflection um Objekte zu modifizieren.
- Lenses holen Bestandteile von Objekten in ein „Brennglas“. Dort kann man das Objekt dann mit Getter und Setter dekorieren.

Grafischer Ablauf

Interface Function

```
// Since Java 1.8
@FunctionalInterface
public interface Function<T, R> {

    R apply(T t);
...
  }
```

Interface BiFunction

```
// Since Java 1.8
@FunctionalInterface
public interface BiFunction<T, U, R> {

    R apply(T t, U u);
...
  }
```

Immutable Clown

```
public final class Clown {

  private final String    name;
  private final ClownType type;
  private final Hat       hat;

  public Clown(final String name, final ClownType type, final Hat hat) {
    this.name = name;
    this.type = type;
    this.hat = hat;
  }

  public String getName() {
    return name;
  }

  public ClownType getType() {
    return type;
  }

  public Hat getHat() {
    return hat;
  }

  @Override
  public String toString() {
    return "Clown [name=" + name + ", type=" + type + ", hat=" + hat + "]";
  }
}
```

Enum ClownType

```
public enum ClownType {

  Auguste, Blackface, Harlequin, Pierrot, Tramp, Whiteface
}
```

Immutable Hat

```
public final class Hat {

  private final Colour colour;
  private final Flower flower;

  public Hat(final Colour colour, final Flower flower) {
    this.colour = colour;
    this.flower = flower;
  }

  public Colour getColour() {
    return colour;
  }

  public Flower getFlower() {
    return flower;
  }

  @Override
  public String toString() {
    return "Hat [colour=" + colour + ", flower=" + flower + "]";
  }
```

Immutable Flower

```
public final class Flower {

  private final Color color;

  public Flower(final Color color) {

    this.color = color;
  }

  public Color getColor() {

    return color;
  }

  @Override
  public String toString() {

    return "Flower [color=" + color + "]";
  }
}
```

Lens Class

```
public class Lens<A, B> {

  private Function<A, B>      getter;
  private BiFunction<A, B, A> setter;

  public Lens(final Function<A, B> getter, final BiFunction<A, B, A> setter) {

    this.getter = getter;
    this.setter = setter;
  }

  compose…
  modify…
}
```

Test für Clown Lenses

```
@Test
public void testClownLenses() {

  Clown clown = new Clown("Zippo", ClownType.AUGUSTE, new Hat(Colour.RED, new
     Flower(Colour.YELLOW)));
  Logger.error(clown);

  final Lens<Clown, Hat> lensClownHat = new Lens<>(c -> c.getHat(), (c, h) -> new Clown(c.getName(),
    c.getType(), h));
  final Lens<Hat, Flower> lensHatFlower = new Lens<>(h -> h.getFlower(), (h, f) -> new
Hat(h.getColour(), f));
  final Lens<Flower, Colour> lensFlowerColour = new Lens<>(f -> f.getColour(), (f, c) -> new Flower(c));

  final Lens<Clown, Colour> changeFlowerColour =
    lensClownHat.compose(lensHatFlower).compose(lensFlowerColour);

  clown = changeFlowerColour.modify(dontCare -> Colour.BLUE).apply(clown);
  Logger.error(clown);
}
```

Modify

```
public Function<A, A> modify(final Function<B, B> modifier) {

  return (source) -> {

    final B extracted = getter.apply(source);
    final B transformed = modifier.apply(extracted);
    return setter.apply(source, transformed);
  };
}
```

Compose

```
public <C> Lens<A, C> compose(final Lens<B, C> tail) {

   return new Lens<>((final A a) -> tail.getter.apply(getter.apply(a)), (final A a, final C c) -> {

     final B b = getter.apply(a);
     final B modifiedB = tail.modify(source -> c).apply(b);
     return setter.apply(a, modifiedB);
   });
 }
```


Zuhören. Analysieren. Beraten.

sidion

Fazit

- Lambdas tangieren nur einen Teil der Clean-Code Regeln.
- Lambdas sind sehr mächtig, wollen also mit Bedacht eingesetzt werden.
- SonarQube kann uns leider nicht die ganze Qualitätssicherung für Lambda-Code abnehmen.
- Lambdas sind nicht nur „syntactic sugar“. Lambdas können in verschiedenen Kontexten zu sauberem Code führen.

DANKE!

FRAGEN?

sidion

Referenzen

- https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
- https://logging.apache.org/log4j/2.x/
- https://www.sonarqube.org/
- Robert C. Martin: Clean Code
- https://codereview.stackexchange.com/questions/78963/n-queens-functional

VIELEN DANK.

sidion

Heßbrühlstr. 15
70565 Stuttgart

Tel. + 49 711 490 109 – 0
Fax + 49 711 490 109 – 79

www.sidion.de

Zuhören. Analysieren. Beraten.