Reguläre Ausdrücke Objektorientiertes Design

Programmiermethodik 2

Zum Nachlesen:
Christian Ullenboom: Java ist auch eine Insel, Kapitel 4.7
http://openbook.rheinwerk-verlag.de/javainsel9/javainsel_04_007.htm

Wiederholung

- Innere Klassen
  - Mitgliedsklasse
  - Statische Innere Klasse
  - Lokale Klasse
  - Anonyme Innere Klasse
- Ereignisverarbeitung in JavaFX
  - Ereignisse und Event-Handler
  - Ereignis-Ursachen
  - Event-Handler-Umsetzungen

Ausblick

Agenda

- Reguläre Ausdrücke
    - Einführung
    - Reguläre Ausdrücke in Java
    - Syntax
- Objektorientiertes Design

Reguläre Ausdrücke

Was ist ein regulärer Ausdruck?

- ist Zeichenkette
- beschreibt Menge von Zeichenketten
- mit Hilfe von syntaktischer Regeln
- engl. regular expression

- Beispiel
  - [0-6]*
  - steht für die Menge aller Zeichenketten beliebiger Länge, die ausschließlich aus den Ziffern 0, 1, 2, 3, 4, 5, 6 besteht
  - z.B. 5143363

Einführung

- gibt es schon lange in der UNIX-Welt
- Unterstützung in einigen UNIX-Tools
    - awk
    - sed
    - grep
- Script-Sprache Perl setzte Standard
    - war/ist u.a. deswegen sehr beliebt

Umsetzung in Java

- reguläre Ausdrücke bewusst an Perl angelegt
- Java ist nicht Perl
- unter Java kurz: regex
- Package java.util.regex
- Java-Tutorial
    - *http://java.sun.com/docs/books/tutorial/essential/regex/intro.html*

Reguläre Ausdrücke in Java

- Reguläre Ausdrücke auch tiefer in Java verdrahtet
- Klassen String, StringBuffer und StringBuilder: Methoden wie z.B. indexOf(String): Auffinden eines Teilstrings.
- für viele einfache Dinge ausreichend
- aber für komplexe Dinge ungenügend

Wozu?

- beschreiben (komplexe) Eigenschaften von Strings
- Untersuchen, Editieren bzw. Manipulieren von Text und Daten
    - z.B: Aufbereiten von Tool-erzeugten ASCII-Ausgaben wie LOG-Dateien
- aber: eigene Syntax unabhängig von der Java-Syntax
    - an Perl angelehnt

Übung: Regulärer Ausdruck?

```
Scanner scanner = new Scanner(System.in);
System.out.print("Bitte regulären Ausdruck eingaben: ");
String regulaererAusdruck = scanner.nextLine();
System.out.print("Bitte zu durchsuchenden Text eingeben: ");
String suchText = scanner.nextLine();
scanner.close();
if (Pattern.matches(regulaererAusdruck, suchText)) {
  System.out.format("Text '%s' passt zu regulärem Ausdruck '%s'.",
      suchText, regulaererAusdruck);
} else {
  System.out.format("Text '%s' passt nicht zu regulärem Ausdruck
'%s'.",
    suchText, regulaererAusdruck);
}
```

Reguläre Ausdrücke in Java

Reguläre Ausdrücke in Java

- drei zentrale Klassen für den Umgang mit regulären Ausdrücken
  - Pattern
  - Matcher
  - PatternSyntaxException: unchecked Exception für Syntax-Fehler im Ausdruck

Pattern

- hat keinen öffentlich zugänglichen Konstruktor
- Objekt ist aufbereitete (kompilierte) Repräsentation des Suchargumentes
    - also eines regulären Ausdrucks
- Objekt wird mit statischer compile()-Methode erzeugt
- compile()-Methode benötigt Ausdruck als String
- Beispiel:

```
Pattern pattern = Pattern.compile("ab*");
```

Matcher

- hat keinen öffentlich zugänglichen Konstruktor
    - Instanzen können nur über Hilfsmethoden erzeugt werden
- Objekt ist aufbereitete Repräsentation des zu untersuchenden Textes - einem Test-String
- Beispiel

```
Pattern pattern = Pattern.compile("ab*");
Matcher matcher = pattern.matcher("aabbcc");
```

Matcher-Methoden

- Vorkommen
  - public boolean find()
  - sucht nächste Übereinstimmung mit dem Ausdruck im zu untersuchenden String
- Übereinstimmung
  - public boolean matches()
  - prüft ob der Ausdruck mit komplettem zu untersuchenden String übereinstimmt

Anwendung

- Überprüfe, ob der Ausdruck <regulaerer-ausdruck> den Text <text> beschreibt:

```
Pattern pattern = Pattern.compile( <regulaerer-ausdruck> );
Matcher matcher = p.matcher( <text> );
boolean passt = matcher.matches();
```

- Kurzform:

```
boolean passt = Pattern.matches(
    <regulaerer-ausdruck>, <text> );
```

- Beispiele:

```
Pattern.matches( "[0-6]*", "5143363"); // true
Pattern.matches( "[0-6]*", "773"); // false
```

Übung: Datum prüfen

- Der reguläre Ausdruck

```
"[1-3]{0,1}[1-9]\\.[1]{0,1}[1-9]\\."
```

- (als String) beschreibt Zeichenketten, die ein Datum in der Form TT.MM. darstellen.
- Schreiben Sie eine Methode istDatum(), die einen eine Zeichenkette als Argument bekommt und genau dann wahr zurückliefert, wenn die Zeichenkette ein Datum (TT.MM.) darstellt; ansonsten liefert die Methode falsch.
- Beispiele:

```
istDatum("28.5."); // true
istDatum("Heinz Müller"); // false
istDatum("53.2."); // false
```

Syntax

Zusammensetzung von Regulären Ausdrücken

- Reguläre bestehen aus:
  - Zeichen oder Literale (literals)
  - Metazeichen (meta character)
  - Zeichen-Gruppen (character classes)
  - Quantoren (quantifier)

Zeichen oder Literale

- (fast) alle Unicode-Zeichen
- Ausnahme: Metazeichen
- Beispiele:
    - 'a'
    - 'A'
    - 'Z'
    - '8'
    - ☠ (Unicode: "\u2620")

Metazeichen

- Zeichen mit einer besonderen Bedeutung
- erlaubte Metazeichen
    - ([{\^-$|]})?*+.
- Aufheben von Metazeichen (wenn man die Metazeichen als normale Zeichen behandeln will)
    - \ (Backslash) unmittelbar davor
    - Position zwischen \Q und \E

Beispiele

- Ausdruck \\
    - steht für Zeichen \
- Ausdruck \Q([{\^-$|]})?*+.\E
    - steht für Zeichenfolge ([{\^-$|]})?*+.

Eigene Zeichengruppen

- engl. character class
- ist keine Java-Klasse!
- wird gebildet durch […]
- Beispiele:

Syntax	Bedeutung	Ergebnis
[abcd]	Menge	a,b,c oder d
[^abcd]	Negation	alles außer a-d
[a-du-x]	Bereich	a, b, c, d, u, v, w, x
[a-d[u-x]]	Vereinigung	[a-du-x]
[a-e&&[bc]]	Schnitt	b oder c
[a-e&&[^bc]]	Subtraktion	a, d oder e
[a-e&&[^b-d]]	Subtraktion + Bereich	a oder e

Vordefinierte Zeichengruppen

.	ein beliebiges Zeichen
\d	eine Ziffer
\D	keine Ziffer
\s	Whitespace
\S	kein Whitespace
\w	ein „Wort-“Zeichen (bzw. [a-zA-Z_0-9])
\W	ein Zeichen der Unicode-Kategorie L
\p{<ID>}	ein Zeichen der Unicode-Kategorie <ID>

<ID>: L = Buchstaben, Lu = Großbuchstaben, Ll = Kleinbuchstaben, Nd = Ziffern, ...

Übung: a.\dx

- In welchen der Test-Strings a) - e) findet sich der reguläre Ausdruck

a.\dx

- a) otto
- b) 11x
- c) a1x
- d) a11x
- e) xaaaaaaaaaaa9x876543a21x

Quantoren

?	optionales, einmaliges Vorkommen (0- oder 1-mal)
*	optionales, beliebig häufiges Vorkommen (0-, 1-, 2-, ..., mal)
+	notwendiges, beliebig häufiges Vorkommen (1-, 2-, …, mal)
{n}	notwendiges n-maliges Vorkommen
{n,}	mindestens n-maliges Vorkommen
{n,m}	mindestens n-maliges, maximal m-maliges Vorkommen

Übung: `a*\d+1x`

- In welchen der Test-Strings a) - e) findet sich der reguläre Ausdruck

```
a*\d+1x
```

```
a)  otto
b)  11x
c)  a1x
d)  a11x
e)  xaaaaaaaaaaa9x876543a21x
```

Übung: `a*(b{1,2}c?)?d+`

- In welchen der Test-Strings a) - e) findet sich der reguläre Ausdruck

```
a*(b{1,2}c?)?d+
```

a) abcd
b) xabcdx
c) abd
d) acd
e) abbcdd
f) xaaaaaaaaadddzaady

Randbedingungen

- Anforderung, dass ein Match direkt am Rand zutreffen muss
  - (Zeilen-)Anfang: ^
  - (Zeilen-)Ende: $
  - Wort-Grenze: \b
  - nicht an der Wort-Grenze: \B

Randbedingungen

- Beispiel 1: Zeichenkette beginnt mit "`http://`": `^(http://)`

```
String regulaererAusdruck = "^http://";
Pattern pattern = Pattern.compile(regulaererAusdruck);
✓ Matcher matcherOk = pattern.matcher("http://www.heise.de");
✗ Matcher matcherNichtOk = pattern.matcher("xxxhttp://www.heise.de");
```

- Beispiel 2: Wort beginnt mit „He"

```
String regulaererAusdruck = "\\b(He)";
Pattern pattern = Pattern.compile(regulaererAusdruck);
✓ Matcher matcherOk = pattern.matcher("Jan Hein Klaas Pit");
✗ Matcher matcherNichtOk = pattern.matcher("JanHeinKlaasPit");
```

Capturing Groups

- fangen Gruppen von Zeichen ein um sie später gezielt weiter zu verwenden (engl. backreferenzes)
- im Ausdruck mit \<zaehler>
    - <zaehler> wird zum Durchnummerieren verwendet
    - später mit .group(<zaehler>)

Beispiele

- Ausdruck (\d\d)\1
  - passt für 1212
  - verfehlt 1234
  - (\d\d) steht dann für 12, \1 verwendet 12 wieder
- Ausdruck: (\d\d)(\d)\1\1\2
  - passt für 12312123
  - verfehlt 123123123123123
  - (\d\d) steht für 12 (Gruppe 1), (\d) steht für 3 (Gruppe 2)

Capturing Groups

- In welcher Reihenfolge werden die Gruppen abgelegt?
- Beispiel
  - ((X)(Y(Z)))
- Lösung (Regeln)
  - von außen nach innen
  - von links nach rechts
- im Beispiel
  - Gruppe 1: ((X)(Y(Z)))
  - Gruppe 2: (X)
  - Gruppe 3: (Y(Z))
  - Gruppe 4: (Z)

- Beispiel:
  - “122233"
  - ((\d)(\d\d\d(\d\d)))
- Liefert
  - 122233
  - 1
  - 22233
  - 33

Zugriff auf Gruppen in Matcher

- Gruppe (Gruppe 0 entspricht (regex))

  ```
  public String group (int gruppenNummer)
  ```

- liefert den Start-Index (erstes Zeichen) der Gruppe

  ```
  public int start (int gruppenNummer)
  ```

- liefert den Index des letzten Zeichen der Gruppe

  ```
  public int end (int gno)
  ```

- liefert die Anzahl Gruppen (ohne Gruppe 0)

  ```
  public int groupCount()
  ```

Beispiel

- Extraktion von Vorname und Nachname aus Namens-String
- Annahme: Trennung durch Leerzeichen „\\s"

```
Pattern pattern = Pattern.compile("([A-Za-z]+)\\s([A-Za-z]+)");
Matcher matcher = pattern.matcher("Philipp Jenke");
System.out.println(matcher.matches()); // Match prüfen
System.out.println(matcher.group(1)); // Vorname
System.out.println(matcher.group(2)); // Nachname
```

Übung: Digitalkamera

- Eine Digitalkamera speichert Bilder unter kryptischen Namen ab:
  - z.B. etwa Pmddnnnn.jpg
  - mit m für Monat (Hexadezimal-Darstellung)
  - mit d für Tag
  - mit n für laufende Foto-Nummer
- Format soll optimiert werden
  - foto_yyyy_mm_dd_nnnn.jpg
  - mit yyyy für aktuelles Jahr
- Beispiele
  - P8305065.jpg → foto_2015_08_30_5065.jpg
  - PB307123.jpg → foto_2015_11_30_7123.jpg
- Aufgabe
  - Schreiben Sie Java-Code zum Konvertieren eines Dateinamens.

Objektorientiertes Design

Zum Nachlesen

- Wikipedia: Prinzipien Objektorientierten Designs
- Brett McLaughlin et al.: Objektorientierte Analyse und Design von Kopf bis Fuß, O'Reilly
- Bernhard Lahres, Gregor Rayman: Objektorientierte Programmierung - Das umfassende Handbuch, Rheinwerk
- Reinhard Schiedermeier: Objektorientiertes Design (http://sol.cs.hm.edu/rs/)

SOLID

- Alpgemeingültige Prinzipien Objektorientierten Programmierens: SOLID
- **S**ingle Responsibility Principle
- **O**pen-Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

Single Responsibility Principle (SRP)

- Jede Klasse soll nur genau eine Verantwortung haben

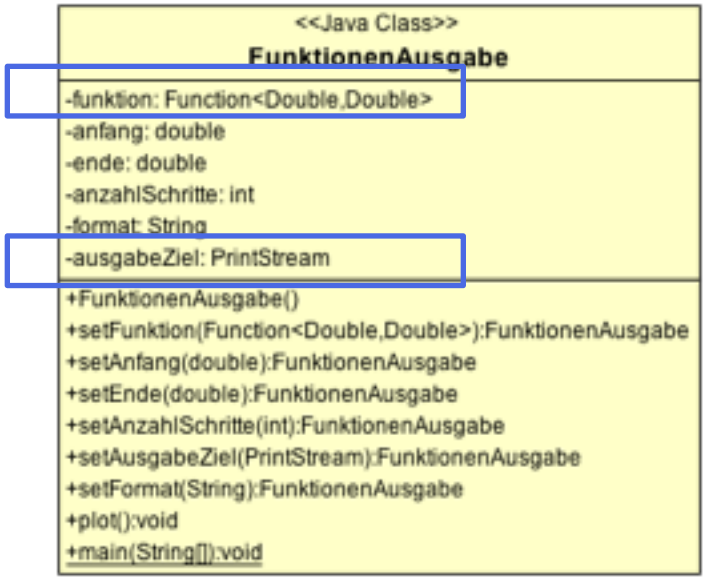
  „Es sollte nie mehr als einen Grund dafür geben, eine Klasse zu ändern."

  – Robert C. Martin: Agile Software Development: Principles, Patterns, and Practices

- Ziel:
  - eine Änderung der Anforderungen sollte nur Änderung einer Klasse auslösen

Beispiel SRP

- Gegeben: Klasse FunktionenAusgabe zum Plot von Funktionswerten



- Beinhaltet zwei Funktionalitäten
  - Funktion auswerten
  - Ausgabe erstellen
- Voneinander Unabhängige Änderungen, die Klasse betreffen
  - Funktion mit mehreren Parametern
  - Ausgabe als Grafik
- Besser: Trennung
  - Funktionsauswertung
  - Ausgabe

Open-Closed Principle (OCP)

- Idee:
  - Offen für Erweiterung, geschlossen für Veränderung
- Ziel:
  - Neue Anforderung erfordert keine Änderung bestehenden Codes, nur neuen Code
- Technische Umsetzung:
  - Offen für Erweiterung: Ableiten neuer Klassen
  - Geschlossen für Veränderung: Datenkapselung (data hiding)
  - häufig: Komposition

Beispiel: OCP

- Repräsentation eines Rechtecks

```
public class Rechteck {
  private double laenge, breite;
  public Rechteck(double laenge, double breite) {
    this.laenge = laenge;
  }
  public double getLaenge() {
    return laenge;
  }
  public double getBreite() {
    return breite;
  }
}
```

- Berechnen der gemeinsamen Fläche mehrerer Rechtecke:

```
double flaeche = 0;
for (Rechteck rechteck : rechtecke) {
  flaeche += rechteck.getLaenge() + rechteck.getBreite();
}
return flaeche;
```

Beispiel OCP

- Nun auch Kreise (gemeinsames Interface Form)

```
double flaeche = 0;
for (Form form : formen) {
  if (form instanceof Rechteck) {
    flaeche += ((Rechteck) form).getLaenge() +
        ((Rechteck) form).getBreite();
  } else if (form instanceof Kreis) {
    double radius = ((Kreis) form).getRadius();
    flaeche += radius * radius * Math.PI;
  }
}
return flaeche;
```

- Unschön:
  - Veränderung der gemeinsamen Flächenberechnung
  - Typecasts!

Beispiel OCP

- Besser: Erweiterung der Schnittstelle

```
public interface Form {
  public double flaecheBerechnen();
}
```

- und damit Vereinfachung des Berechnung

```
double flaeche = 0;
for (Form form : formen) {
  flaeche += form.flaecheBerechnen();
}
return flaeche;
```

- Offen für Erweiterung: neuen Firmenklassen verlangen keine Veränderung der gemeinsamen Funktionalität

Liskov Substitution Principle (LSP)

- Idee:
  - Objekte in einem Programm sollten sich mit Instancen ihrer (abgeleiteten) Untertypen ersetzen lassen, ohne die Korrektheit des Programs zu verändern

Beispiel LSP

```
public class Rechteck {
 protected double laenge, breite;
  public Rechteck(double laenge,
      double breite) {
    this.laenge = laenge;
  }
  public void setLaenge(double laenge) {
    this.laenge = laenge;
  }
  public void setBreite(double breite) {
    this.breite = breite;
  }
  public double getFlaeche() {
    return laenge * breite;
  }
}

public class Quadrat extends Rechteck {
  public Quadrat(double seitenlaenge) {
    super(seitenlaenge, seitenlaenge);
  }
  public void setLaenge(double laenge) {
    this.laenge = laenge;
    this.breite = laenge;
  }
  public void setBreite(double breite) {
    this.laenge = breite;
    this.breite = breite;
  }
}
```

- Berechnung des Flächeninhalts

```
Rechteck rechteck = ...
rechteck.setLaenge(2);
rechteck.setBreite(3);
System.out.println(rechteck.getFlaeche());
```

- Ergebnis 6 für

```
Rechteck rechteck = new Rechteck(1, 1);
```

- Ergebnis 9 für

```
Rechteck rechteck = new Quadrat(1);
```

- LSP verletzt

Interface Segregation Principle (ISP)

- Idee
  - Clients sollten nicht gezwungen werden, von Methoden abzuhängen, die sie nicht verwenden
- Maßnahmen:
  - Interfaces und ABCs beschränken auf Gruppen von Methoden, die logisch zusammenhängen
  - Aufteilung von Interfaces, die genau auf Clients passen

Beispiel ISP

- Interface java.util.Iterator<T>:

```
boolean hasNext()
T next()
default void remove()
default void forEachRemaining(Consumer<? super T>)
```

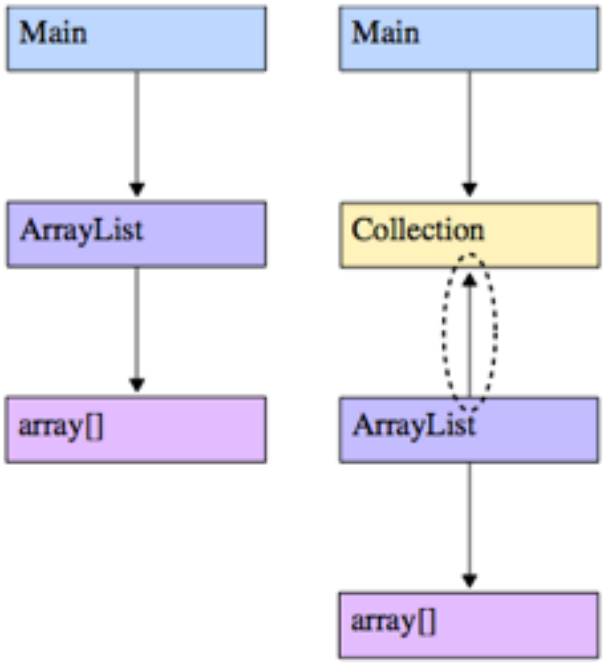
- Implementierungen für read-only-Container ⇒ kein remove
  - Ohne Redefinition: remove wirft UnsupportedOperationException
- besser:

```
interface ReadonlyIterator<T> {
  boolean hasNext();
  T next();
  default void forEachRemaining(Consumer<? super T>) {…}
}
interface ModifiableIterator<T> extends ReadonlyIterator<T>{
  default void remove() {…}
}
```

Dependency Inversion Principle (DIP)

- Idee
  - Module hoher Ebenen sollten nicht von Modulen niedriger Ebenen abhängen. Beide sollten von Abstraktionen abhängen.

- Main-Methode kennt interne Datenstruktur (links)
- Main-Methode und interne Daten beziehen sich auf eine Abstraktion (rechts)



Quelle: Reinhard Schiedermeier: http://sol.cs.hm.edu/rs/ood/slide0069.xhtml, 23.06.16

Design by Contract

- Entwurf gemäß Vertrag
- reibungslose Zusammenspiel einzelner Programmmodule durch die Definition formaler „Verträge“
- Schnittstellen
- Sicherstellung durch
    - Vorbedingungen
    - Nachbedingungen
    - Invarianten
- in Java nicht durch Programmiersprache unterstützt
    - außer: Erweiterungen (Bean Validation), externe Bibliotheken, z.B. Oval (http://oval.sourceforge.net/), Google Contacts for Java (https://github.com/nhatminhle/cofoja)
- aber: Asserts

Design By Contract

- Vorbedingungen
  - engl. precondition
  - Zusicherungen, die der Aufrufer einzuhalten hat
- Nachbedingungen
  - engl. postcondition
  - Zusicherungen, die der Aufgerufene einhalten wird
- Invarianten
  - engl. class invariants
  - Gesundheitszustand einer Klasse

Invarianten

- Aussage, die immer wahr ist
- Klasseninvarianten: Aussage über den Zustand eine Klasse
    - kann Werte von Objektvariablen beinhalten
    - muss vor Aufruf einer Methode gelten
    - muss nach Verlassen einer Methode gelten
    - darf innerhalb einer Methode zwischenzeitlich ungültig sein

- Beispiel: Wert alter muss immer >= 0 sein

```
public class Alter {
  /**
   * Aktuelles Alter, muss >= 0 sein.
   */
  private int alter;
  ...
}
```

Vorbedingungen: Pragmatischer Ansatz

- Vorbedingungen mittels Javadoc-Kommentar für jede Methode formulieren („Vorbedingung“)
- Vorbedingungen zu Beginn der Methode abprüfen
  - Assert
  - Exception

```
/**
 * Setzt das Alter.
 *
 * Vorbedingungen: Parameter alter >= 0<br>
 * Nachbedingungen: -
 *
 * @param alter
 *            Neuer Wert für das Alter.
*/
public void setAlter(int alter) {
  // entweder: Assert
  assert (alter >= 0);
  // oder: Exception
  if (alter < 0) {
    throw new IllegalArgumentException("Neuer Alter-Wert muss größer-gleich 0 sein.");
  }
  this.alter = alter;
}
```

Nachbedingungen: Pragmatischer Ansatz

- Nachbedingungen mittels Javadoc-Kommentar für jede Methode formulieren („Nachbedingung“)

```
/**
 * Bestimmt, ob es das Alter eines Erwachsenen ist.
 *
 * Vorbedingungen: -
 * Nachbedingungen: Liefert wahr, falls alter >= 18, sonst falsch.
 *
 * @return Wahr, wenn die Person erwachsen ist.
 */
public boolean istErwachsen() {
  return alter >= 18;
}
```

Invarianten: Pragmatischer Ansatz

- Überprüfen, wenn sich der Zustand einer Klasse ändert
    - Setter
    - Konstruktoren
    - Methoden, die Zustand verändern

- Überprüfen mit
    - entweder: Assertions
    - oder Exceptions

Übung:

- Gegeben ist die Klasse `Stapel`, die einen Stapel von int-Werten verwaltet:

```
public class Stapel {
  int[] stapel = new int[5];
  int anzahlElemente = 0;
}
```

- Gegeben Sie die Invarianten der Klasse an
- Schreiben Sie eine Methoden `hinzufuegen(int)`, die ein weiteres Element auf den Stapel legt. Geben Sie die Vor- und Nachbedingungen an und prüfen Sie diese falls möglich ab.

Zusammenfassung

- Reguläre Ausdrücke
    - Einführung
    - Reguläre Ausdrücke in Java
    - Syntax
- Objektorientiertes Design

Quellen

- Die Folien basieren auf einem Foliensatz von Michael Schäfers, Hochschule für Angewandte Wissenschaften (HAW) Hamburg
- Christian Ullenboom: Java ist auch eine Insel, Galileo Computing, 10. Auflage, 2011, Kapitel 4.7

  oder online

- *http://*openbook.galileocomputing.de/javainsel9/javainsel_04_007.htm, abgerufen am 28.04.2014