

Reguläre Ausdrücke

Carsten Gips (HSBI)

Unless otherwise noted, this work is licensed under CC BY-SA 4.0.

Gesucht ist ein Programm zum Extrahieren von Telefonnummern aus E-Mails.

=> **Wie geht das?**

Suchen in Strings

Gesucht ist ein Programm zum Extrahieren von Telefonnummern aus E-Mails.

=> **Wie geht das?**

030 - 123 456 789, 030-123456789, 030/123456789,
+49(30)123456-789, +49 (30) 123 456 - 789, ...

Definition Regulärer Ausdruck

Ein **regulärer Ausdruck** ist eine Zeichenkette, die zur Beschreibung von Zeichenketten dient.

Einfachste reguläre Ausdrücke

Zeichenkette	Beschreibt
<code>x</code>	"x"
<code>.</code>	ein beliebiges Zeichen
<code>\t</code>	Tabulator
<code>\n</code>	Newline
<code>\r</code>	Carriage-return
<code>\\</code>	Backslash

- `abc` => "abc"
- `A.B` => "AAB" oder "A2B" oder ...
- `a\\bc` => "a\bc"

Zeichenkette	Beschreibt
[abc]	“a” oder “b” oder “c”
[^abc]	alles außer “a”, “b” oder “c” (Negation)
[a-zA-Z]	alle Zeichen von “a” bis “z” und “A” bis “Z” (Range)
[a-z&&[def]]	“d”, “e” oder “f” (Schnitt)
[a-z&&[^bc]]	“a” bis “z”, außer “b” und “c”: [ad-z] (Subtraktion)
[a-z&&[^m-p]]	“a” bis “z”, außer “m” bis “p”: [a-lq-z] (Subtraktion)

- [abc] => “a” oder “b” oder “c”
- [a-c] => “a” oder “b” oder “c”
- [a-c][a-c] => “aa”, “ab”, “ac”, “ba”, “bb”, “bc”, “ca”, “cb” oder “cc”
- A[a-c] => “Aa”, “Ab” oder “Ac”

Vordefinierte Ausdrücke

Zeichenkette	Beschreibt
<code>^</code>	Zeilenanfang
<code>\$</code>	Zeilenende
<code>\d</code>	eine Ziffer: <code>[0-9]</code>
<code>\w</code>	beliebiges Wortzeichen: <code>[a-zA-Z_0-9]</code>
<code>\s</code>	Whitespace (Leerzeichen, Tabulator, Newline)
<code>\D</code>	jedes Zeichen außer Ziffern: <code>[^0-9]</code>
<code>\W</code>	jedes Zeichen außer Wortzeichen: <code>[^\w]</code>
<code>\S</code>	jedes Zeichen außer Whitespaces: <code>[^\s]</code>

- `\d\d\d\d\d` => "12345"
- `\w\wA` => "aaA", "a0A", "a_A", ...

- `java.lang.String`:

```
public String[] split(String regex)
public boolean matches(String regex)
```

Demo: `regexp.StringSplit`

Nutzung in Java

- `java.lang.String`:

```
public String[] split(String regex)
public boolean matches(String regex)
```

Demo: `regexp.StringSplit`

- `java.util.regex.Pattern`:

```
public static Pattern compile(String regex)
public Matcher matcher(CharSequence input)
```

- `java.util.regex.Matcher`:

```
public boolean find()
public boolean matches()
public int groupCount()
public String group(int group)
```

Unterschied zw. Finden und Matchen

- `Matcher#find`:

Regulärer Ausdruck muss im Suchstring **enthalten** sein.

=> Suche nach **erstem Vorkommen**

- `Matcher#matches`:

Regulärer Ausdruck muss auf **kompletten** Suchstring passen.

- Regulärer Ausdruck: `abc`, Suchstring: "blah blah abc blub"

- `Matcher#find`: erfolgreich

- `Matcher#matches`: kein Match - Suchstring entspricht nicht dem Muster

Zeichenkette	Beschreibt
<code>X?</code>	ein oder kein "X"
<code>X*</code>	beliebig viele "X" (inkl. kein "X")
<code>X+</code>	mindestens ein "X", ansonsten beliebig viele "X"
<code>X{n}</code>	exakt n Vorkommen von "X"
<code>X{n,}</code>	mindestens n Vorkommen von "X"
<code>X{n,m}</code>	zwischen n und m Vorkommen von "X"

- `\d{5}` \Rightarrow "12345"
- `-?\d+\.\d*` \Rightarrow ???

Interessante Effekte

```
Pattern p = Pattern.compile("A.*A");  
Matcher m = p.matcher("A 12 A 45 A");  
  
if (m.matches())  
    String result = m.group(); // ???
```

Nicht gierige Quantifizierung mit “?”

Zeichenkette	Beschreibt
<code>X*?</code>	non-greedy Variante von <code>X*</code>
<code>X+?</code>	non-greedy Variante von <code>X+</code>

- Suchstring “A 12 A 45 A”:
 - `A.*A` findet/passt auf “A 12 A 45 A”
 - `A.*?A`
 - findet “A 12 A”
 - passt auf “A 12 A 45 A” (!)

(Fangende) Gruppierungen

`Studi{2}` passt **nicht** auf “StudiStudi” (!)

(Fangende) Gruppierungen

`Studi{2}` passt **nicht** auf “StudiStudi” (!)

Zeichenkette	Beschreibt
<code>X Y</code>	X oder Y
<code>(C)</code>	Gruppierung

- `(A)(B(C))`
 - Gruppe 0: `ABC`
 - Gruppe 1: `A`
 - Gruppe 2: `BC`
 - Gruppe 3: `C`

(Fangende) Gruppierungen

`Studi{2}` passt **nicht** auf “StudiStudi” (!)

Zeichenkette	Beschreibt
<code>X Y</code>	X oder Y
<code>(C)</code>	Gruppierung

- `(A)(B(C))`
 - Gruppe 0: `ABC`
 - Gruppe 1: `A`
 - Gruppe 2: `BC`
 - Gruppe 3: `C`

`(Studi){2}` => “StudiStudi”

Gruppen und Backreferences

Matche zwei Ziffern, gefolgt von den selben zwei Ziffern

Gruppen und Backreferences

Matche zwei Ziffern, gefolgt von den selben zwei Ziffern

```
(\d\d)\1
```

- Verweis auf bereits gematchte Gruppen: `\num`

`num` Nummer der Gruppe (1 ... 9)

=> Verweist nicht auf regulären Ausdruck, sondern auf jeweiligen Match!

- Benennung der Gruppe: `(?<name>X)`

`X` ist regulärer Ausdruck für Gruppe, spitze Klammern wichtig

=> Backreference: `\k<name>`

Beispiel Gruppen und Backreferences

Regulärer Ausdruck: Namen einer Person matchen, wenn Vor- und Nachname identisch sind.

Beispiel Gruppen und Backreferences

Regulärer Ausdruck: Namen einer Person matchen, wenn Vor- und Nachname identisch sind.

Lösung: `([A-Z] [a-zA-Z]*)\s\1`

- RegExp: Zeichenketten, die andere Zeichenketten beschreiben
- `java.util.regex.Pattern` und `java.util.regex.Matcher`
- Unterschied zwischen `Matcher#find` und `Matcher#matches`!
- Quantifizierung ist möglich, aber **greedy** (Default)



Unless otherwise noted, this work is licensed under CC BY-SA 4.0.