Propädeutikum
Programmierung in der Bioinformatik

Java – Reguläre Ausdrücke und PROSITE-Patterns

Thomas Mauermeier

04.12.2018

Ludwig-Maximilians-Universität München

Warum Reguläre Ausdrücke?

Regulärer Ausdruck

engl. **Reg**ular **Ex**pression ≈ Regex

≙ Muster (**Pattern**) mit dem man eine Menge anderer Strings beschreiben kann

Beispielpattern

`Amalien(str\.|straße)␣17`

Würde beschreiben:

- Amalienstr. 17
- Amalienstraße 17

Warum? → nächste Folie!

Quelle: https://imgs.xkcd.com/comics/regular_expressions.png

Anatomie eines Regex-Patterns

`Amalien(str\.|straße)␣17`

besteht aus:

Literals

- Symbole **ohne** Spezialbedeutung
- Meint: Wenn ich z.B. “A” schreibe meine ich auch dass der Computer hier “A” wörtlich verstehen soll

Metacharacters

- Symbole **mit** Spezialbedeutung
- Meint: Wenn ich z.B. “(...|...)” schreibe, sind das syntaktische Elemente die der Computer **nicht wörtlich** versteht

Aufschlüsselung des Beispielpatterns

`Amalien(str\.|straße)␣17`

Die beschriebenen Strings sollen...

- Mit `Amalien` beginnen

Aufschlüsselung des Beispielpatterns

`Amalien(str\.|straße)␣17`

Die beschriebenen Strings sollen...

- Mit `Amalien` beginnen
- Danach gibt es zwei Optionen wie der String weitergehen kann:

Aufschlüsselung des Beispielpatterns

`Amalien(str\.|straße)␣17`

Die beschriebenen Strings sollen...

- Mit `Amalien` beginnen
- Danach gibt es zwei Optionen wie der String weitergehen kann:
  - Entweder mit `str\.` oder `straße`
  - Warum `str\.` statt `str.`? → Punkt ist **Metacharacter**!
    ⇒ \ vor Punkt setzen = **Escaping**: Verändert die Bedeutung diverser Characters!

Aufschlüsselung des Beispielpatterns

`Amalien(str\.|straße)␣17`

Die beschriebenen Strings sollen...

- Mit `Amalien` beginnen
- Danach gibt es zwei Optionen wie der String weitergehen kann:
  - Entweder mit `str\.` oder `straße`
  - Warum `str\.` statt `str.`? → Punkt ist **Metacharacter**!
    ⇒ \ vor Punkt setzen = **Escaping**: Verändert die Bedeutung diverser Characters!
- und soll mit ␣17 enden

Metacharacters

`^`	Anker für Beginn der Zeile
`$`	Anker für Ende der Zeile
`\b`	Anker für Wortgrenze

`[ ]`	Klasse
`[^ ]`	Negation einer Klasse
`( )`	Block, Capturing Group
`|`	Alternativen, “oder”

`*`	Quantifier für 0 bis ∞-maliges Vorkommen
`+`	Quantifier für 1 bis ∞-maliges Vorkommen
`?`	Quantifier für 0 bis 1-maliges Vorkommen
`{n}`	Quantifier für **exakt** n-maliges Vorkommen
`{n,}`	Quantifier für n bis ∞-maliges Vorkommen
`{n,m}`	Quantifier für n bis m-maliges Vorkommen

Zeilenanker – ^ und $

`^Amalien(str\.|straße)␣17`

↪ Matcht nur am Zeilenanfang

```
1 Amalienstraße␣17
2 In␣der␣Amalienstraße␣17
3 Amalienstraße␣17␣ist␣ein␣Ort
4 Amalienstraße␣17istEinOrt
```

`Amalien(str\.|straße)␣17$`

↪ Matcht nur am Zeilenende

```
1 Amalienstraße␣17
2 In␣der␣Amalienstraße␣17
3 Amalienstraße␣17␣ist␣ein␣Ort
4 Amalienstraße␣17istEinOrt
```

❓ Was würde `^Amalien(str\.|straße)␣17$` im obigen Text matchen?

Zeilenanker – ^ und $

`^Amalien(str\.|straße) 17`

↪ Matcht nur am Zeilenanfang

```
1 Amalienstraße 17
2 In der Amalienstraße 17
3 Amalienstraße 17 ist ein Ort
4 Amalienstraße 17istEinOrt
```

`Amalien(str\.|straße) 17$`

↪ Matcht nur am Zeilenende

```
1 Amalienstraße 17
2 In der Amalienstraße 17
3 Amalienstraße 17 ist ein Ort
4 Amalienstraße 17istEinOrt
```

❓ Was würde `^Amalien(str\.|straße) 17$` im obigen Text matchen?

Zeilenanker – ^ und $

`^Amalien(str\.|straße) 17`

↪ Matcht nur am Zeilenanfang

```
1 Amalienstraße 17
2 In der Amalienstraße 17
3 Amalienstraße 17 ist ein Ort
4 Amalienstraße 17istEinOrt
```

`Amalien(str\.|straße) 17$`

↪ Matcht nur am Zeilenende

```
1 Amalienstraße 17
2 In der Amalienstraße 17
3 Amalienstraße 17 ist ein Ort
4 Amalienstraße 17istEinOrt
```

❓ Was würde `^Amalien(str\.|straße) 17$` im obigen Text matchen?

Zeilenanker – ^ und $

`^Amalien(str\.|straße) 17`

↪ Matcht nur am Zeilenanfang

```
1 Amalienstraße 17
2 In der Amalienstraße 17
3 Amalienstraße 17 ist ein Ort
4 Amalienstraße 17 istEinOrt
```

`Amalien(str\.|straße) 17$`

↪ Matcht nur am Zeilenende

```
1 Amalienstraße 17
2 In der Amalienstraße 17
3 Amalienstraße 17 ist ein Ort
4 Amalienstraße 17 istEinOrt
```

❓ Was würde `^Amalien(str\.|straße) 17$` im obigen Text matchen?

→ Nur erste Zeile: Zeilenanfang (^) muss vor Match, Ende ($) direkt dahinter

Word Boundary Anker – \b

\bcat\b

↪ Matcht nur wenn an Position von \b ein Wort endet

```
1 scatter␣cat␣catapult␣scat␣cat-foo
```

cat	matcht →	scatter␣cat␣catapult␣scat␣cat-foo
\bcat	→	scatter␣cat␣catapult␣scat␣cat-foo
cat\b	→	scatter␣cat␣catapult␣scat␣cat-foo

⚠ Ein Wort ist so definiert dass Sonderzeichen wie z.B. - als Boundary gelten!

Word Boundary Anker – \b

\bcat\b

↪ Matcht nur wenn an Position von \b ein Wort endet

```
1 scatter␣cat␣catapult␣scat␣cat-foo
```

	matcht	
cat	→	scatter␣cat␣catapult␣scat␣cat-foo
\bcat	→	scatter␣cat␣catapult␣scat␣cat-foo
cat\b	→	scatter␣cat␣catapult␣scat␣cat-foo

⚠ Ein Wort ist so definiert dass Sonderzeichen wie z.B. - als Boundary gelten!

Klassen – [ ]

`Am[aeou]lien(str\.|straße)␣17`

↪ Matcht exakt **ein** Symbol aus der Klasse

```
In␣der␣Amalienstr.␣17␣ist␣der␣CIP-Pool.
In␣der␣Amelienstr.␣17␣ist␣der␣CIP-Pool.
In␣der␣Amilienstr.␣17␣ist␣der␣CIP-Pool.
In␣der␣Amolienstr.␣17␣ist␣der␣CIP-Pool.
In␣der␣Amulienstr.␣17␣ist␣der␣CIP-Pool.
In␣der␣Amaeoulienstr.␣17␣ist␣der␣CIP-Pool.
```

Klassen – [ ]

`Am[aeou]lien(str\.|straße)␣17`

↪ Matcht exakt **ein** Symbol aus der Klasse

```
1 In␣der␣Amalienstr.␣17␣ist␣der␣CIP-Pool.
2 In␣der␣Amelienstr.␣17␣ist␣der␣CIP-Pool.
3 In␣der␣Amilienstr.␣17␣ist␣der␣CIP-Pool.
4 In␣der␣Amolienstr.␣17␣ist␣der␣CIP-Pool.
5 In␣der␣Amulienstr.␣17␣ist␣der␣CIP-Pool.
6 In␣der␣Amaeoulienstr.␣17␣ist␣der␣CIP-Pool.
```

Negation von Klassen – [^]

`Am[^aeou]lien(str\.|straße)␣17`

↪ Matcht exakt **ein** Symbol das **nicht** in Klasse ist

```
In der Amalienstr. 17 ist der CIP-Pool.
In der Amelienstr. 17 ist der CIP-Pool.
In der Amilienstr. 17 ist der CIP-Pool.
In der Amolienstr. 17 ist der CIP-Pool.
In der Amulienstr. 17 ist der CIP-Pool.
In der Amaeoulienstr. 17 ist der CIP-Pool.
```

⚠ ^ **in** einer Klasse (Negation) ≠ ^ **außerhalb** einer Klasse (Zeilenanker)

Negation von Klassen – [^]

`Am[^aeou]lien(str\.|straße)␣17`

↪ Matcht exakt **ein** Symbol das **nicht** in Klasse ist

```
In␣der␣Amalienstr.␣17␣ist␣der␣CIP-Pool.
In␣der␣Amelienstr.␣17␣ist␣der␣CIP-Pool.
In␣der␣Amilienstr.␣17␣ist␣der␣CIP-Pool.
In␣der␣Amolienstr.␣17␣ist␣der␣CIP-Pool.
In␣der␣Amulienstr.␣17␣ist␣der␣CIP-Pool.
In␣der␣Amaeoulienstr.␣17␣ist␣der␣CIP-Pool.
```

⚠ ^ **in** einer Klasse (Negation) ≠ ^ **außerhalb** einer Klasse (Zeilenanker)

Range – [X-Y]

`Amalien(str\.|straße) 17[a-d]`

↪ Kurzschreibweise in Klassen für eine (längere) Folge von Characters

```
In der Amalienstr. 17a ist der CIP-Pool.
In der Amalienstr. 17b ist der CIP-Pool.
In der Amalienstr. 17c ist der CIP-Pool.
In der Amalienstr. 17d ist der CIP-Pool.
In der Amalienstr. 17 a ist der CIP-Pool.
In der Amalienstr. 172 ist der CIP-Pool.
```

⚠ – **in** einer Klasse (Range) $\neq$ – **außerhalb** einer Klasse (Literal)

Range – [X-Y]

`Amalien(str\.|straße)␣17[a-d]`

↪ Kurzschreibweise in Klassen für eine (längere) Folge von Characters

```
1 In␣der␣Amalienstr.␣17a␣ist␣der␣CIP-Pool.
2 In␣der␣Amalienstr.␣17b␣ist␣der␣CIP-Pool.
3 In␣der␣Amalienstr.␣17c␣ist␣der␣CIP-Pool.
4 In␣der␣Amalienstr.␣17d␣ist␣der␣CIP-Pool.
5 In␣der␣Amalienstr.␣17␣a␣ist␣der␣CIP-Pool.
6 In␣der␣Amalienstr.␣172␣ist␣der␣CIP-Pool.
```

⚠ – **in** einer Klasse (Range) $\neq$ – **außerhalb** einer Klasse (Literal)

Vordefinierte Klassen

↪ Kurzschreibweisen für häufig verwendete Klassen

Klasse	Bedeutung
`.`	**Alles** außer Line Terminator, d.h. auch Whitespace
`\w`	Jeder Character aus einem **w**ort: `[a-zA-Z0-9]`
`\W`	Negation von `\w`: `[^\w]`
`\d`	Jeder Character aus einer Zahl (**d**igit): `[0-9]`
`\D`	Negation von `\d`: `[^\d]`
`\s`	Jeder Whitespace-Char: `[␣\t\n\x0B\f\r]`
`\S`	Negation von `\s`: `[^\s]`

⚠ `.` **in** einer Klasse (Literal) $\neq$ `.` **außerhalb** einer Klasse (Klasse)

Alternative (“oder”) – |

`str\.|straße`

↪ Zur Trennung von alternativen Pattern: *entweder ... oder ... oder ...*

```
Heute in der Amalienstr. 17
Gestern in der Amalienstraße 17
DieAmalienstraße 17
Amalien Straße
Amalien Str.
```

Alternative (“oder”) – |

`str\.|straße`

↪ Zur Trennung von alternativen Pattern: *entweder ... oder ... oder ...*

```
Heute in der Amalienstr. 17
Gestern in der Amalienstraße 17
DieAmalienstraße 17
Amalien Straße
Amalien Str.
```

Block bzw. Capturing Group – ( )

`(Amalien)(str\.|aße)␣17`

↪ Zur Gruppierung von Subpatterns und für **Backreferencing**

```
1 Amalienstr.␣17
```

Backreference auf...
Gruppe 0: `Amalienstr.␣17`
Gruppe 1: `Amalien`
Gruppe 2: `str.`

```
1 Amalienstraße␣17
```

Backreference auf...
Gruppe 0: `Amalienstraße␣17`
Gruppe 1: `Amalien`
Gruppe 2: `aße`

↪ Backreferencing erlaubt Zugriff auf Teile des gematchten Strings

Backreferencing **im Pattern** mittels: \1 für Gruppe 1, \2, für Gruppe 2 etc.

Quantifier

`Ama*lien(str\.|straße)␣17`

↪ Bezieht sich auf vorheriges Element im Pattern

```
1 Amlienstr.␣17
2 Amalienstr.␣17
3 Amaalienstr.␣17
4 Amaaalienstr.␣17
5 Amaaaalienstr.␣17
6 Amooolienstr.␣17
7 Amoooolienstr.␣17
```

Quantifier	Vorkommen
*	0 bis ∞-malig
+	1 bis ∞-malig
?	0 bis 1-malig
{n}	**exakt** n-malig
{n,}	n bis ∞-malig
{n,m}	n bis m-malig

Quantifier

`Ama*lien(str\.|straße)␣17`

↪ Bezieht sich auf vorheriges Element im Pattern

```
1 Amlienstr.␣17
2 Amalienstr.␣17
3 Amaalienstr.␣17
4 Amaaalienstr.␣17
5 Amaaaalienstr.␣17
6 Amoooolienstr.␣17
7 Amooooolienstr.␣17
```

Quantifier	Vorkommen
*	0 bis ∞-malig
+	1 bis ∞-malig
?	0 bis 1-malig
{n}	**exakt** n-malig
{n,}	n bis ∞-malig
{n,m}	n bis m-malig

Quantifier

`Am[ao]+lien(str\.|straße)␣17`

↪ Alle Quantifier funktionieren auch mit Klassen

```
1 Amlienstr.␣17
2 Amalienstr.␣17
3 Amaalienstr.␣17
4 Amaaalienstr.␣17
5 Amaaaalienstr.␣17
6 Amooolienstr.␣17
7 Amoooolienstr.␣17
```

Quantifier	Vorkommen
`*`	0 bis ∞-malig
`+`	1 bis ∞-malig
`?`	0 bis 1-malig
`{n}`	**exakt** n-malig
`{n,}`	n bis ∞-malig
`{n,m}`	n bis m-malig

Quantifier

`Am[ao]+lien(str\.|straße)␣17`

↪ Alle Quantifier funktionieren auch mit Klassen

```
1 Amlienstr.␣17
2 Amalienstr.␣17
3 Amaalienstr.␣17
4 Amaaalienstr.␣17
5 Amaaaalienstr.␣17
6 Amooolienstr.␣17
7 Amoooolienstr.␣17
```

Quantifier	Vorkommen
*	0 bis ∞-malig
+	1 bis ∞-malig
?	0 bis 1-malig
{n}	**exakt** n-malig
{n,}	n bis ∞-malig
{n,m}	n bis m-malig

Quantifier

`Am(aaa)?lien(str\.|straße)␣17`

↪ ... und auch mit Capturing Groups!

```
1 Amlienstr.␣17
2 Amalienstr.␣17
3 Amaalienstr.␣17
4 Amaaalienstr.␣17
5 Amaaaalienstr.␣17
6 Amooolienstr.␣17
7 Amoooolienstr.␣17
```

Quantifier	Vorkommen
`*`	0 bis ∞-malig
`+`	1 bis ∞-malig
`?`	0 bis 1-malig
`{n}`	**exakt** n-malig
`{n,}`	n bis ∞-malig
`{n,m}`	n bis m-malig

Quantifier

`Am(aaa)?lien(str\.|straße)␣17`

↪ ... und auch mit Capturing Groups!

```
1 Amlienstr.␣17
2 Amalienstr.␣17
3 Amaalienstr.␣17
4 Amaaalienstr.␣17
5 Amaaaalienstr.␣17
6 Amooolienstr.␣17
7 Amoooolienstr.␣17
```

Quantifier	Vorkommen
`*`	0 bis ∞-malig
`+`	1 bis ∞-malig
`?`	0 bis 1-malig
`{n}`	**exakt** n-malig
`{n,}`	n bis ∞-malig
`{n,m}`	n bis m-malig

Quantifier

```
Ama{3}lien(str\.|straße)␣17
```

```
1 Amlienstr.␣17
2 Amalienstr.␣17
3 Amaalienstr.␣17
4 Amaaalienstr.␣17
5 Amaaaalienstr.␣17
6 Amooolienstr.␣17
7 Amoooolienstr.␣17
```

Quantifier	Vorkommen
*	0 bis ∞-malig
+	1 bis ∞-malig
?	0 bis 1-malig
{n}	**exakt** n-malig
{n,}	n bis ∞-malig
{n,m}	n bis m-malig

Quantifier

Ama{3}lien(str\.|straße)␣17

```
1 Amlienstr.␣17
2 Amalienstr.␣17
3 Amaalienstr.␣17
4 Amaaalienstr.␣17
5 Amaaaalienstr.␣17
6 Amooolienstr.␣17
7 Amoooolienstr.␣17
```

Quantifier	Vorkommen
*	0 bis ∞-malig
+	1 bis ∞-malig
?	0 bis 1-malig
{n}	**exakt** n-malig
{n,}	n bis ∞-malig
{n,m}	n bis m-malig

Quantifier

```
Am(a|oo){2,}lien(str\.|straße)␣17
```

```
1 Amlienstr.␣17
2 Amalienstr.␣17
3 Amaalienstr.␣17
4 Amaaalienstr.␣17
5 Amaaaalienstr.␣17
6 Amooolienstr.␣17
7 Amoooolienstr.␣17
```

Quantifier	Vorkommen
*	0 bis ∞-malig
+	1 bis ∞-malig
?	0 bis 1-malig
{n}	**exakt** n-malig
{n,}	n bis ∞-malig
{n,m}	n bis m-malig

Quantifier

```
Am(a|oo){2,}lien(str\.|straße)␣17
```

```
1 Amlienstr.␣17
2 Amalienstr.␣17
3 Amaalienstr.␣17
4 Amaaalienstr.␣17
5 Amaaaalienstr.␣17
6 Amooolienstr.␣17
7 Amoooolienstr.␣17
```

Quantifier	Vorkommen
*	0 bis ∞-malig
+	1 bis ∞-malig
?	0 bis 1-malig
{n}	**exakt** n-malig
{n,}	n bis ∞-malig
{n,m}	n bis m-malig

Quantifier

`Ama{0,3}lien(str\.|straße)␣17`

```
1 Amlienstr.␣17
2 Amalienstr.␣17
3 Amaalienstr.␣17
4 Amaaalienstr.␣17
5 Amaaaalienstr.␣17
6 Amooolienstr.␣17
7 Amoooolienstr.␣17
```

Quantifier	Vorkommen
*	0 bis ∞-malig
+	1 bis ∞-malig
?	0 bis 1-malig
{n}	**exakt** n-malig
{n,}	n bis ∞-malig
{n,m}	n bis m-malig

Quantifier

`Ama{0,3}lien(str\.|straße)␣17`

```
1 Amlienstr.␣17
2 Amalienstr.␣17
3 Amaalienstr.␣17
4 Amaaalienstr.␣17
5 Amaaaalienstr.␣17
6 Amooolienstr.␣17
7 Amoooolienstr.␣17
```

Quantifier	Vorkommen
*	0 bis ∞-malig
+	1 bis ∞-malig
?	0 bis 1-malig
{n}	**exakt** n-malig
{n,}	n bis ∞-malig
{n,m}	n bis m-malig

Obacht!

- Es gibt fast immer mehrere Lösungen für ein bestimmtes Pattern:
  - `Amalien(str\.|straße)␣17` ≈ `Amalienstr(\.|aße)␣17`
- Patterns können/müssen je nach Anforderung präzise bis allgemein sein:
  - `\w+str(\.|aße)␣\d{2}` – allgemein
  - `Amalienstr(\.|aße)␣17` – präzise
- Jede Regex-Implementation (also: Java, Python, grep, etc.) hat ihren eigenen Dialekt. Syntax, Funktionsweise, Funktionsumfang können sich unterscheiden.
  - **Aber**: Wenn man das Konzept verstanden hat, sind die Ideen leicht übertragbar.
- Wird case-sensitive oder case-insensitive gematcht? (`Match` ≠ `match`)
  - Meist lässt sich die Defaulteinstellung durch setzen einer Flag ändern.

Regex in Java – Klassen Pattern und Matcher

Pattern

Klasse die Regex-Patterns repräsentiert, aber an sich noch nichts matchen kann

↪ **Beschreibt** was gematcht wird

Matcher

Klasse die Matches anhand eines Pattern-Objekts in einem String findet

↪ **Macht** das eigentliche Matching

Minimalbeispiel:

```java
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class RegexTest {
  public static void main(String[] args){
    // "a*b" ist das Pattern
    Pattern p = Pattern.compile("a*b");
    // in "aaaaab" wird gematcht
    Matcher m = p.matcher("aaaaab");
    System.out.println("Pattern in
        ↪ String: " + m.matches());
    // Ausgabe: "Pattern in String: true"
  }
}
```

Regex in Java – Klassen Pattern und Matcher

Erzeugung eines Patterns

`Pattern.compile(String regex)` nimmt ein Pattern als String an, kompiliert es, und gibt ein Pattern-Objekt zurück.

→ entspricht Konstruktor

Minimalbeispiel:

```java
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class RegexTest {
  public static void main(String[] args){
    // "a*b" ist das Pattern
    Pattern p = Pattern.compile("a*b");
    // in "aaaaab" wird gematcht
    Matcher m = p.matcher("aaaaab");
    System.out.println("Pattern in
        ↪ String: " + m.matches());
    // Ausgabe: "Pattern in String: true"
  }
}
```

Regex in Java – Klassen Pattern und Matcher

Erzeugung eines Matchers

Aufrufen der `Pattern`-Methode `p.matcher(String s)`, nimmt einen String auf dem gematcht werden soll an und gibt ein `Matcher`-Objekt zurück, das mit dem Pattern aus `p` arbeitet.

Minimalbeispiel:

```java
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class RegexTest {
  public static void main(String[] args){
    // "a*b" ist das Pattern
    Pattern p = Pattern.compile("a*b");
    // in "aaaaab" wird gematcht
    Matcher m = p.matcher("aaaaab");
    System.out.println("Pattern in
        ↪ String: " + m.matches());
    // Ausgabe: "Pattern in String: true"
  }
}
```

Regex in Java – Klassen Pattern und Matcher

Arbeiten mit dem Matcher

Das `Matcher`-Objekt bietet verschiedene Methoden an um in dem angegebenen String zu matchen. (Siehe Docs)

`m.matches()` returned z.B. ein `true` wenn das Pattern `a*b` den ganzen String `aaaaab` matcht.

Minimalbeispiel:

```java
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class RegexTest {
  public static void main(String[] args){
    // "a*b" ist das Pattern
    Pattern p = Pattern.compile("a*b");
    // in "aaaaab" wird gematcht
    Matcher m = p.matcher("aaaaab");
    System.out.println("Pattern in
        ↪ String: " + m.matches());
    // Ausgabe: "Pattern in String: true"
  }
}
```

Beispiel: Durchsuchen eines Files

```java
// Imports ausgelassen
public class RegexTest2 {
    public static void main(String[] args) {
        Pattern p = Pattern.compile("Amalien(str\\.|straße) 17");
        try (BufferedReader br = new BufferedReader(new
            ↪ FileReader("test.txt"))) {
            String line = null;
            Matcher m = null;
            while ((line = br.readLine()) != null) {
                m = p.matcher(line);
                while (m.find()) {
                  System.out.println("Match: " + m.group(0));
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Verwendete `Matcher`-Methoden

`find()`

Matcher versucht nächstes Vorkommen des Patterns zu finden
Returned `true` wenn gefunden, sonst `false`

`group(int n)`

Returned String mit Inhalt der *n*-ten Capturing Group des zuletzt durch `find()` gefundenen Matchs
Die 0-te Capturing Group ist immer der gesamte Match

```
5  // ...
6  String line = null;
7  Matcher m = null;
8  while ((line = br.readLine()) != null) {
9      m = p.matcher(line);
10     while (m.find()) {
11       System.out.println("Match: " +
             ↪ m.group(0));
12     }
13 }
14 // ...
```

Escaping in Java

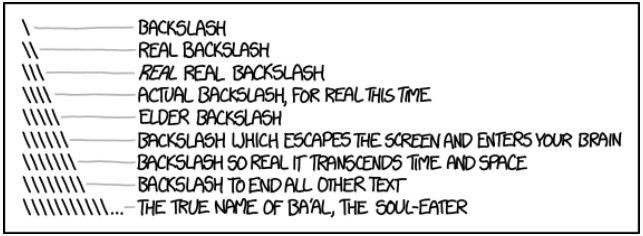
Wie erwähnt: \ vor einem Character verändert die Bedeutung, zB. \.

Problem: Java benutzt unabhängig von Regex intern schon Escaping

⇓

Lösung: Um Java klar zu machen, dass wir \. meinen, müssen wir erst \ an sich escapen: \\

Also: Java "erkennt" \. nur als \\.



Quelle: https://imgs.xkcd.com/comics/backslashes.png

PROSITE-Pattern

`C-x(2,4)-C-x(3)-[LIVMFYWC]-x(8)-H-x(3,5)-H`

$\hookrightarrow$ Art regulärer Ausdruck für ein Sequenzmotiv

- Sequenzmotiv: Muster einer gewissen Abfolge von Aminosäuren
- Hat andere Syntax: `https://prosite.expasy.org/prosuser.html#conv_pa`
- Lässt sich aber übersetzen $\rightarrow$ Übungsblatt

Weitere Informationen

- Regexr: Tool zum Testen von Patterns. **Nutzt es!**
  `https://regexr.com/`
- Oracle Regex Tutorial:
  `https://docs.oracle.com/javase/tutorial/essential/regex/index.html`
- Javadoc zur Pattern Klasse:
  `https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html`
- Javadoc zur Matcher Klasse:
  `https://docs.oracle.com/javase/8/docs/api/java/util/regex/Matcher.html`
- Buchempfehlung: Mastering Regular Expressions
  Kostenlos zu lesen als Ebook in der TUM-Bibliothek

Viel Spaß bei der Übung

Quelle: `https://imgs.xkcd.com/comics/perl_problems.png`