

Reguläre Ausdrücke

- Ein **regulärer Ausdruck** (engl. *regular expression*, Abk.: **RegExp** oder **Regex**) ist eine Zeichenkette, die der Beschreibung von Mengen von Zeichenketten mit Hilfe bestimmter syntaktischer Regeln dient.
- Reguläre Ausdrücke können als Filterkriterien in der Textsuche verwendet werden, indem der Text mit dem Muster des regulären Ausdrucks abgeglichen wird. (Pattern Matching). So ist es beispielsweise möglich, alle Wörter aus einer Wortliste (Array, Collection) herauszusuchen, die mit *S* beginnen und auf *D* enden – ohne die dazwischenliegenden Buchstaben oder deren Anzahl explizit vorgeben zu müssen.
- Mit einem einzigen regulären Ausdruck kann man eine Menge unterschiedlicher Texte beschreiben, die ähnlich aussehen, z.B.
 - Telefonnummern
 - Mail-Adressen
 - Datumsangaben
 - Uhrzeiten
- Reguläre Ausdrücke dienen dazu, solche Texte zu untersuchen, in ihre Einzelteile zu zerlegen oder zu verändern (suchen und ersetzen)

Anwendung:

1. Zeichenlitterale : einzelnes Zeichen oder Zeichenkette („a“, „al“)

Regex → a ;	String → „Hallo 3Al da draußen“	→ 3 „Matches“
Regex → al ;	String → „Hallo 3Al da draußen“	→ 1 Match

2. Metazeichen ... haben besondere Eigenschaften in Regex

[] () { } | ? + - * ^ \$ \ .

➔ Um die besonderen Eigenschaften aufzuheben, muss \ davor gestellt werden (in Java Strings \\ !!) → \[, \?,

3. Zeichenmengen : definiert mit [..] .. jedes Zeichen der Menge wird gefunden

Regex → [aeiou]	String → „Hallo 3Al da draußen“	→ 6 Matches
Regex → [aeiAEI]	String → „Hallo 3Al da draußen“	→ 6 Matches
Regex → [abc123]	String → „Hallo 3Al da draußen“	→ 4 Matches
Regex → [a-c1-3]	String → „Hallo 3Al da draußen“	→ 4 Matches

4. Sonderzeichen
(komplette Übersicht hier: [Java API Pattern](#))

.	(ein Punkt)	Jedes beliebige Zeichen
\d		Eine Ziffer → [0-9]
\D		Alles was KEINE Ziffer ist
\w		Buchstabe, Ziffer, Unterstrich
\W		KEIN Buchstabe, Ziffer, Unterstrich
\s		Leerraum (Abstand, Tabulator, Zeilenumbruch,..)
\S		KEIN Leerraum
^		Verneinung → [^0-9] → \D

Regex → **[a-c].[a-c]** String → „Data Match Pattern “ → 2 Matches
 Regex → **[\d]:[0-9]** String → „Ergebnis 4:2“ → 1 Match

5. Quantoren .. definieren die Häufigkeit des Vorkommens des vorher stehenden Zeichens (Zeichengruppe) fest:

?	Der voranstehende Ausdruck ist optional, er kann einmal vorkommen, muss es aber nicht, d. h. der Ausdruck kommt null- oder einmal vor.
+	Der voranstehende Ausdruck muss mindestens einmal vorkommen, darf aber auch mehrfach vorkommen
*	Der voranstehende Ausdruck darf beliebig oft (auch keinmal) vorkommen.
{n}	Der voranstehende Ausdruck muss exakt <i>n</i> -mal vorkommen.
{min,}	Der voranstehende Ausdruck muss mindestens <i>min</i> -mal vorkommen.
{min,max}	Der voranstehende Ausdruck muss mindestens <i>min</i> -mal und darf maximal <i>max</i> -mal vorkommen.
{0,max}	Der voranstehende Ausdruck darf maximal <i>max</i> -mal vorkommen

Beispiele:

1) Sportergebnisse:

a. Überprüfe **5:3**

- i. [0-9]:[0-9]
- ii. \d:\d

b. Überprüfe **57:49**

- i. [0-9]+:[0-9]+
- ii. [0-9]{1,}:[0-9]{1,}

c. Überprüfe **57:49 (23:20)**

- i. [0-9]+:[0-9]+ *\(\d+:\d+\)

d. Überprüfe **Team1 – Team2 57:49 (23:20)**

- i. \w{3,} *- *\w{3,} +[0-9]+:[0-9]+ *\(\d+:\d+\)

6. Verwendung von regulären Ausdrücken in Java (Strings)

Direkt auf String s angewendet:

- `s.matches(regex)`: überprüft, ob der String (komplett) dem Muster entspricht, das durch regex definiert ist.
- `s.replaceAll(regex, t)`: ersetzt alle Treffer von regex durch t
- `s.replaceFirst(regex, t)`: ersetzt ersten Treffer von regex durch t
- `s.split(regex)`: `split()` arbeitet generell mit regex
- `s.split(regex, count)`: nur count Elemente kommen ins array

7. Erweiterungen :

- ODER – Verknüpfung von Suchmustern mit `|`:

`"[1-9]{3}|bbb"` ... sucht nach 3 Zahlen ODER bbb

- Suche am ANFANG des Strings mit `^`:

`"^[1-9]{3}"` ... sucht nach 3 Zahlen am Anfang !!!

- Suche am ENDE des Strings mit `$`:

`"[1-9]{3}$"` ... sucht nach 3 Zahlen am Ende !!!

8. Verwendung von Pattern und Matcher:

```
String s = "16.12.1963";  
Pattern p = Pattern.compile("\\d{2}\\d{2}\\d{4}");  
Matcher m = p.matcher(s);
```

- m.matches(): überprüft auf vollständige Übereinstimmung
- m.find(): überprüft, ob (irgendwo) eine Übereinstimmung gefunden wird
 - if (m.find()) ... liefert true wenn gefunden
 - while (m.find()) ... iteriert über alle Fundstellen
 - mit m.group() kann ein übereinstimmender String(teil) ermittelt werden
- erg = m.replaceAll(neu) ... ersetzt jeden Treffer durch String neu
- erg = m.replaceFirst(neu) ... ersetzt ersten Treffer durch String neu

8. **Gruppen** : Durch Definition von Gruppen mit (und) wird nichts am Suchvorgang selbst verändert, allerdings lässt sich das Suchergebnis detaillierter darstellen.

- "\\d{2}\\\\.\\d{2}\\\\.\\d{4}" ... sucht Muster wie 16.12.1963, das Ergebnis wird als Gesamtheit betrachtet (nach m.find() liefert m.group() das gesamte Datum)
- "(\\d{2})\\. (\\d{2})\\. (\\d{4})" ... durch die runden Klammern wurden 3 Gruppen definiert, das Ergebnis kann als Gesamtheit betrachtet (nach m.find() liefert m.group() oder m.group(0) das gesamte Datum). Man kann aber auch auf die einzelnen Gruppen zugreifen :
 - i. m.group(1) liefert 16,
 - ii. m.group(2) liefert 12,
 - iii. m.group(3) liefert 1963
- Rückwärtsreferenzen in Gruppen:
"(\\d{2})\\. (\\d{1,2})\\. \\1" ... durch die runden Klammern wurden 2 Gruppen definiert, der dritte Teil ist eine Rückwärtsreferenz auf die erste Gruppe → dieser Teil muss für Übereinstimmung GLEICH der ersten Gruppe sein !!
 - i. Treffer z.B. für: „22.11.22“, aber nicht „22.11.21“

9. „gierige“ und „nicht gierige“ Ausdrücke

Beispiel: man möchte Ausdrücke der folgenden Form überprüfen:

beliebige Zeichen + 10 Ziffern + beliebige Zeichen

aa2bb1222333444cc5dd

folgender Regex wäre passend: "[a-z0-9]*([0-9]{10})[a-z0-9]*"

- i. m.group(1) liefert aa2bb,
- ii. m.group(2) liefert 1222333444,
- iii. m.group(3) liefert cc5dd

verändere nun den String zu : aa2bb12223334445cc5dd (nach 444 ein 5er !!)

- i. m.group(1) liefert aa2bb1,
- ii. m.group(2) liefert 2223334445,
- iii. m.group(3) liefert cc5dd

Erklärung: Der * Operator verhält sich „gierig“ ... er „schluckt“ so viele Zeichen wie möglich → es werden also die letzten 10 Ziffern aus der zweiten Gruppe „getroffen“. (weil die erste Gruppe möglichst viele Zeichen „schlucken“ möchte)
Mit dem Zusatzoperator ? kann ein „gieriger“ Operand zum Zurückhaltenden gemacht werden → ([a-z0-9]*?)

geänderter Regex: "`([a-z0-9]*)(\\d{10})([a-z0-9]*)`"

- i. m.group(1) liefert aa2bb,
- ii. m.group(2) liefert 1222333444,
- iii. m.group(3) liefert 5cc5dd