



## **PM2 Java: Reguläre Ausdrücke**



# Reguläre Ausdrücke

- **Definition, erste Beispiele**
- Zeichen in regulären Ausdrücken
- Vollständige Übereinstimmung
- Effizienz durch Kompilieren von Mustern
- Partielle Übereinstimmung
- Gierige (greedy) und bescheidene (reluctant) Operatoren
- Gruppen
- Gruppen und Rückverweise
- Wortgrenzen
- Anfang und Ende einer Zeichenfolge
- Einsammeln von Matches
- Zeichenketten zerlegen



# Reguläre Ausdrücke

- Reguläre Ausdrücke (**Regex**) beschreiben durch normale und gesondert interpretierte Zeichen (Sonderzeichen) Muster (**pattern**) in einem String.
- Mit regulären Ausdrücken lassen sich in Java folgende String-Manipulationen realisieren:
  - Passt eine String vollständig in ein Muster: **matches** (vollständiger Match)
  - Enthält ein String Teilstrings: **regionMatches** (Klasse **String**), **find** (Klasse **Matcher**) (partieller Match)
  - Teilstring-Ersetzung: **replaceAll**, **replaceFirst** (Klassen **String** und **Matcher**)
  - Zeichenketten-Zerlegung: **split** (Klasse **String**)



# Beispiele für reguläre Ausdrücke

Alle nachfolgenden Ausdrücke beschreiben Zeichenketten, die mit <i>aa</i> beginnen und auf <i>bb</i> enden		
Ausdruck	Matched mit	Erklärung
<i>"aa.*bb"</i>	<i>aabb</i> <i>aa(9&amp;Raxshgdhbb</i> <i>aa65bbb7...aaabbbbbb</i>	Dazwischen dürfen beliebige Zeichen stehen. Der Punkt "." steht für ein beliebiges Zeichen, * für keine oder beliebige Wiederholung.
<i>"aa\ . *bb"</i>	<i>aabb</i> <i>aa.....bb</i>	Dazwischen dürfen beliebig viele Punkte stehen.
<i>"aa.+bb"</i>	<i>aaXbb</i> <i>aa(9&amp;Raxshgdhbb</i> <i>aa65bbb7...aaabbbbbb</i>	Dazwischen muss mindestens ein Zeichen stehen.
<i>"aa.?bb"</i>	<i>aabb</i> <i>aaXbb</i>	Dazwischen muss genau ein oder kein Zeichen stehen.
<i>"aa.{2}bb"</i>	<i>aa.Pbb</i> <i>aaXYbb</i>	Dazwischen müssen genau zwei Zeichen stehen.
<i>"aa.{2,}bb"</i>	<i>aa.PZZZZZZbb</i> <i>aaP.bb</i> <i>aaPZbb</i>	Dazwischen müssen mindestens zwei Zeichen stehen.
<i>"aa.{1,3}bb"</i>	<i>aaZbb</i> <i>aaPZbb</i>	Dazwischen muss mindestens ein / dürfen maximal 3 Zeichen stehen.



# Reguläre Ausdrücke

- Definition, erste Beispiele
- ***Zeichen in regulären Ausdrücken***
- Vollständige Übereinstimmung
- Effizienz durch Kompilieren von Mustern
- Partielle Übereinstimmung
- Gierige (greedy) und bescheidene (reluctant) Operatoren
- Gruppen
- Gruppen und Rückverweise
- Wortgrenzen
- Anfang und Ende einer Zeichenfolge
- Einsammeln von Matches
- Zeichenketten zerlegen



# Zeichen in einem regulären Ausdruck

Ausdruck	Bedeutung
a	Zeichen a
\\	Zeichen \
\t	Tabulator
\n	Zeilenumbruch
\r	Zeilenvorschub
\u0085	Zeilenende
\u2028	Zeilenumbruch
\u2029	Paragraphumbruch
\. \? \* \+ \(\) \{ \}	die Zeichen . ? * + ( ) { }
.	beliebiges Zeichen
?,*,+	Quantifizierer
( ) { }	Sonderzeichen Gruppe / Quantifizierer

- In regulären Ausdrücken können alle Zeichen verwendet werden, die in „normalen“ Zeichenketten gebräuchlich sind.
- eine Reihe von Zeichen haben eine besondere Bedeutung in regulären Ausdrücken, z.B. die Zeichen **. ? \* + { } ( )**
- Sind diese Zeichen wörtlich gemeint, dann muss ihnen das Escape-Zeichen **\** (der Backslash) vorangestellt werden.



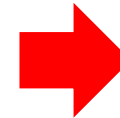
# Quantifizierer

- Quantifizierer beschreiben die Häufigkeit des Auftretens eines Zeichenmusters X. X steht hier für ein beliebiges Zeichenmuster.
  - **X?**: X kommt ein oder keinmal vor
  - **X+**: X kommt mindestens einmal vor
  - **X\***: X kommt keinmal oder mehrfach vor
  - **X{n}**: X muss genau n-mal vorkommen.
  - **X{n,}**: X kommt mindestens n-mal vor.
  - **X{n,m}**: X kommt mindestens n-, aber maximal m-mal vor.
- Sollen diese Zeichen wörtlich in einem regulären Ausdruck verwendet werden, so müssen sie escaped werden (durch **\\** oder ***Pattern.quote***)



# Quantifizierer

```
private static void quantifyDemo() {  
    p("Quantifizierer");  
    p(Pattern.matches("a", "aa"));  
    p(Pattern.matches("a", "b"));  
    p(Pattern.matches("a?", "aa"));  
    p(Pattern.matches("a+", "aa"));  
    p(Pattern.matches("a*", "aa"));  
    p(Pattern.matches("a*", "a*"));  
    p(Pattern.matches("a\\*", "a*"));  
    p(Pattern.matches(Pattern.quote("a*"), "a*"));  
    p(Pattern.matches("a{2}", "aa"));  
    p(Pattern.matches("a{3,}", "aa"));  
    p(Pattern.matches("a{1,4}", "aa"));  
}
```



false  
false  
false  
true  
true  
false  
true  
true  
true  
false  
true

→ Package allgemein Klasse `RegexDemo`





# Zeichenklassen

- Möchte man nicht einzelne Zeichen sondern einzelne Zeichen einer Kategorie beschreiben, z.B. einen Vokal, dann müssen **Zeichenklassen** verwendet werden.
- Zeichenklassen fassen die möglichen Zeichen in eckigen Klammern ein.
- Zeichen folgen ohne Trennzeichen aufeinander. Ein Leerzeichen in einer Zeichenklasse ist auch ein Zeichen
- Aufeinander folgende Zeichen sind durch **oder** verknüpft.
- Es ist möglich Zeichen durch Negation (^) auszuschließen. Die Negation bezieht sich auf alle Zeichen der Zeichenklasse.
- Zeichenbereiche werden mit - definiert
- Durch **&&** werden Schnittmengen von Zeichen dargestellt.



# Zeichenklassen

- Beispiele:
  - **[aeiou]** Vokal.
  - **[^aeiou]** ein Zeichen außer einem Vokal.
  - **[0-9a-fA-F]** eine Ziffer, ein Klein- oder ein Großbuchstabe
  - **[a-z&&[r-w]]** ein Zeichen im Bereiche r bis w
  - **[a-z&&[^r-w]]** ein Kleinbuchstabe außer r bis w.



# Zeichenklassen

```
private static void characterClassDemo() {  
    p("Zeichenklassen");  
    p(Pattern.matches("[aeiou]*", "Alle Neune"));  
    p(Pattern.matches("[aeiou]AW*", "Aale Wale"));  
    p(Pattern.matches("[0-9aeiou]*", "a0i7o6"));  
    // Beginnt das Wort mit b gefolgt von 7 Ziffern und endet es mit %  
    p("b012345%".matches("b[0-9]{7}%"));  
    // Email Adresses matchen  
    p(Pattern.matches("[^@]+@.+\\.\\.[^.]+" , "Birgit.Wendholt@haw.de"));  
}
```



Zeichenklassen  
false  
true  
true  
true  
true

→ Package allgemein Klasse `RegexDemo`



# Vordefinierte Zeichenklassen (vgl. *java.util.regex.Pattern*)

Zeichenklasse	Erklärung
.	Punkt steht für ein beliebiges Zeichen
\d	Ziffern: [0-9]
\D	keine Ziffer: [^0-9] bzw. [^\d]
\s	Leerzeichen: [ \t\n\x0B\f\r]
\S	kein Leerzeichen: [^\s]
\w	alphanumerisches Zeichen: [a-zA-Z0-9]
\W	Kein alphanumerisches Zeichen: [a-zA-Z0-9]
\p{Punct}	Punkt-Zeichen: !"#\$%&'()*+,-./:;<=>?@[\\]^_`{ }~



# Verwenden von Sonderzeichen in Strings

- Sonderzeichen müssen in Strings durch Verwendung eines Backslashes „escaped“ werden.
- Sollen die Zeichen `$ ^ . ? ( ) { } [ ]` nicht als Sonderzeichen sondern als einfaches Zeichen interpretiert werden, so müssen diese ebenfalls mit einem Backslash escaped werden.

	In String
<code>\d \D \s \S \w \W</code>	<code>"\\d" "\\D" "\\s" "\\S" "\\w" "\\W"</code>
<code>\p{Punct}</code>	<code>"\\p{Punct}"</code>
<code>\$ ^ . ? ( ) { } [ ]</code>	<code>"\\\$" "\\^" "\\." "\\?" "\\(" "\\["</code> etc.



## Vordefinierte Zeichenklassen

```
private static void characterClassPredefinedDemo() {  
    p("Vordefinierte Zeichenklassen");  
    // Enthält das Wort nur Ziffern oder ist es das leere Wort  
    p(Pattern.matches("\\d*", "1346"));  
    p(Pattern.matches("\\d*", ""));  
    p(Pattern.matches("\\d*", "aa1346"));  
    // Beginnt das Wort mit zwei beliebigen Zeichen und endet mit 0-n  
    // Ziffern  
    p(Pattern.matches("..\\d*", "aa1346"));  
    p(Pattern.matches("..\\d*", "aa1346"));  
    // Enthält das Wort keine Ziffern  
    p(Pattern.matches("\\D*", "aa1346"));  
    p(Pattern.matches("\\D*", "aa Wal! \\t voraus"));  
}
```

→ Package allgemein Klasse `RegexDemo`

*true*  
*true*  
*false*  
*true*  
*true*  
*false*  
*true*



# Logische Operatoren

- **XY**: X gefolgt von Y
- **X | Y**: entweder X oder Y
- **(X)**: Capturing Group, **()** bindet stärker als **/**

```
private static void logicalOpsDemo() {  
    p("Logische Verknüpfungen");  
    p(Pattern.matches("[a-z]{3}|[0-9]{3}.*", "abc 123xxxx"));  
    p(Pattern.matches("[a-z]{3}|[0-9]{3}.*", "123 abcyyyy"));  
    p(Pattern.matches("[a-z]{3}|[0-9]{2,3}.*", "12 abczz"));  
    p(Pattern.matches("[a-z]{3}[0-9]{3}.*", "abc 123____"));  
    p(Pattern.matches("[a-z]{3} [0-9]{3}.*", "abc 123äääääää"));  
    p(Pattern.matches("[a-z]{3} [0-9]{2,3}.*", "123 abciiii"));  
    p(Pattern.matches("[0-9]{2,3} [a-z]{3}.*", "12 abcuuuuuuuuuu"));  
    p(Pattern.matches("Mo|Di|Mi|Do|Fr", "Mo"));  
    p(Pattern.matches("M|Di|o|Fr", "Mo"));  
    p(Pattern.matches("(M|D)(i|o)|Fr", "Mo"));  
}
```

→ true  
true  
true  
false  
true  
false  
true  
true  
false  
true

→ Package allgemein Klasse `RegexDemo`



# Kleine Aufgaben

- Welcher reguläre Ausdruck beschreibt die digitale Uhrenanzeige. (Bsp.: 15:34)
- Welcher reguläre Ausdruck beschreibt die digitale Uhrenanzeige in 12-Stunden Darstellung. (Bsp.: 04:00 a.m. 10:34 p.m.)





# Reguläre Ausdrücke

- Definition, erste Beispiele
- Zeichen in regulären Ausdrücken
- ***Vollständige Übereinstimmung***
- Effizienz durch Kompilieren von Mustern
- Partielle Übereinstimmung
- Gierige (greedy) und bescheidene (reluctant) Operatoren
- Gruppen
- Gruppen und Rückverweise
- Wortgrenzen
- Anfang und Ende einer Zeichenfolge
- Einsammeln von Matches
- Zeichenketten zerlegen



## Vollständiges Patternmatching mit *matches*

- Die Methoden *matches* von *String*, *Matcher* und *Pattern* prüfen, ob ein String mit einem Muster vollständig übereinstimmt.

```
public static void main(String[] args) {  
    // prüft ob ein oder mehr Zeichen zwischen  
    // einfachen Quotes stehen  
    p( Pattern.matches( "'.*'", "'Moin Folks'" ) );  
    p( "'Moin Folks'".matches( "'.*'" ) );  
    p( Pattern.matches( "'.*'", "" ) );  
    p( Pattern.matches( "'.*'", "Moin Folks" ) );  
    p( Pattern.matches( "'.*'", "'Moin Folks'" ) );  
    Pattern pattern = Pattern.compile( "'.*'" );  
    Matcher matcher = pattern.matcher( "'Moin Folks'" );  
    p (matcher.matches());  
}
```



true  
true  
true  
false  
false  
false

→ Package allgemein Klasse *StringMatches*



# Reguläre Ausdrücke

- Definition, erste Beispiele
- Zeichen in regulären Ausdrücken
- Vollständige Übereinstimmung
- ***Effizienz durch Kompilieren von Mustern***
- Partielle Übereinstimmung
- Gierige (greedy) und bescheidene (reluctant) Operatoren
- Gruppen
- Gruppen und Rückverweise
- Wortgrenzen
- Anfang und Ende einer Zeichenfolge
- Einsammeln von Matches
- Zeichenketten zerlegen



# Effizientes Pattern-Matching

- Wird ein Pattern mehrfach angewendet, so sollte es kompiliert werden, um das teure Übersetzen nur einmal vornehmen zu müssen. (*[Pattern.compile\(regex\)](#)*)
- Auf der Basis der kompilierten Musters und einer zu vergleichende Zeichenkette wird ein Matcher-Objekt erzeugt
- Für den Mustervergleich wird die Methode *[matches](#)* auf dem Matcher-Objekt aufgerufen.

```
Pattern pattern = Pattern.compile("'.*');  
Matcher matcher = pattern.matcher("'Moin Folks");  
boolean doesMatch = matcher.matches();
```

- Die Methoden *[matches](#)* von *[String](#)* und *[Pattern](#)* kompilieren das Muster, erzeugen einen *[Matcher](#)* und rufen die Methode *[matches](#)* auf.
- Wird ein Muster nur einmal verwendet, dann empfiehlt sich die Verwendung der Methoden von *[Pattern](#)* und *[String](#)*



# Reguläre Ausdrücke

- Definition, erste Beispiele
- Zeichen in regulären Ausdrücken
- Vollständige Übereinstimmung
- Effizienz durch Kompilieren von Mustern
- ***Partielle Übereinstimmung***
- Gierige (greedy) und bescheidene (reluctant) Operatoren
- Gruppen
- Gruppen und Rückverweise
- Wortgrenzen
- Anfang und Ende einer Zeichenfolge
- Strings mit regulären Ausdrücken zerlegen
- Einsammeln von Matches



# Partielles Pattern-Matching auf Teilstrings

- Die Methode *find()* der Klasse *Matcher* findet alle Teilstrings in einem *String*, die mit einem regulären Ausdruck übereinstimmen.
- Die Methode *find()* merkt
  - den letzten übereinstimmenden Teilstring, der über *group()* erfragbar ist.
  - die Startposition zu diesem Teilstring, die über *start()* erfragbar ist.
  - die Endposition zu diesem Teilstring, die über *end()* erfragbar ist.



## Partielles Pattern-Matching auf Teilstrings

```
private static void greedyOperatorsDemo() {  
    p("Gierige Quantifizierer");  
    String str = "<b> Moin </b> Folks <b> Wake Up </b>";  
    Pattern pattern = Pattern.compile("<b>.*</b>");  
    Matcher matcher = pattern.matcher(str);  
  
    while (matcher.find())  
        p(matcher.group());  
    p("");  
}
```



Gierige Quantifizierer  
<b> Moin </b> Folks <b> Wake Up </b>

*Eigentlich hätten wir hier zwei Matchergebnisse für Teilstrings erwartet. Das Ergebnis ist hingegen der Eingabestring str. Das liegt an dem „gierigen“ Verhalten der Quantifizierer*

→ Package allgemein Klasse RegexDemo



# Reguläre Ausdrücke

- Definition, erste Beispiele
- Zeichen in regulären Ausdrücken
- Vollständige Übereinstimmung
- Effizienz durch Kompilieren von Mustern
- Partielle Übereinstimmung
- ***Gierige (greedy) und bescheidene (reluctant) Operatoren***
- Gruppen
- Gruppen und Rückverweise
- Wortgrenzen
- Anfang und Ende einer Zeichenfolge
- Einsammeln von Matches
- Zeichenketten zerlegen





# Gierige und nicht gierige Operationen

- Die Muster ***\*,?,+*** sind gierige (**greedy**) Operatoren, d.h. es wird immer versucht den **maximalen** String zu finden, der auf das Muster passt.
- Möchte man den **minimalen** String finden, müssen bescheidene (**reluctant**) Operatoren verwendet werden.
- Dies wird erreicht, in dem den gierigen Operatoren ein **?** nachgestellt wird.
  - ***X??***
  - ***X+?***
  - ***X\*?***
  - ***X{n}?***
  - ***X{n,}?***
  - ***X{n,m}?***



# Gierige und nicht gierige Operationen

- **Beispiel.** In dem vorhergehenden HTML Schnipsel sollen alle Vorkommen von fett hervorgehobenen Textelementen bestimmt werden.
- `<b> Moin </b> Folks <b> Wake Up </b>`
- Der gierige Quantifizierer aus dem Beispiel der vorausgehenden Folie versucht möglichst viele Zeichen zwischen `<b>` und `</b>` zu binden, so dass das Muster `.*` auf  
`Moin </b> Folks <b> Wake Up.`  
abgebildet wird.
- Der bescheidene `*?` Quantifizierer sucht nach dem ersten Auftreten von `</b>` nachdem `<b>` gelesen wurde und bindet so möglichst wenig Zeichen zwischen `<b>` und `</b>`.



## Teilstringmatch mit reluctant Operatoren

```
private static void reluctantOperatorsDemo2() {  
    p("Bescheidene Quantifizierer");  
    String str = "<b> Moin </b> Folks <b> Wake Up </b>";  
    Pattern pattern = Pattern.compile("<b>.*?</b>");  
    Matcher matcher = pattern.matcher(str);  
    // group() gibt den letzten Match zurück  
    // start() gibt die Anfangsposition des letzten matches  
    // end() die Endposition des letzten matches aus  
    while (matcher.find())  
        p(String.format("%1$s an Position [%2$d,%3$d]", matcher.group(),  
            matcher.start(), matcher.end()));  
    p("");  
}
```



Bescheidene Quantifizierer  
<b> Moin </b> an Position [0,13]  
<b> Wake Up </b> an Position  
[20,36]

→ Package allgemein Klasse RegexDemo



# Reguläre Ausdrücke

- Definition, erste Beispiele
- Zeichen in regulären Ausdrücken
- Vollständige Übereinstimmung
- Effizienz durch Kompilieren von Mustern
- Partielle Übereinstimmung
- Gierige (greedy) und bescheidene (reluctant) Operatoren
- **Gruppen**
- Gruppen und Rückverweise
- Wortgrenzen
- Anfang und Ende einer Zeichenfolge
- Einsammeln von Matches
- Zeichenketten zerlegen



# Gruppen (capturing groups)

- Eine ***capturing group*** umfasst alle Elemente, die zwischen einer geöffneten und einer geschlossenen runden Klammer in einem regulären Ausdruck stehen.
- Bsp.: in **((A) (B(C)))** sind die ***capturing groups***
  1. **((A) (B(C)))**
  2. **(A)**
  3. **(B (C))**
  4. **(C)**
- Bei geschachtelten Gruppen ergibt sich die Zählung der Gruppen von links Tiefe zuerst.
- Die Gruppe 0 ist immer der gesamte Match. Im obigen Beispiel sind Gruppe 0 und Gruppe 1 gleich.



# Gruppen und partielles Matching

- Beim partiellem Match merkt sich ein *Matcher* die Zeichensequenz, die Start und die Endposition der Gruppen eines Matches.
- Die Methoden *group(int index)*, *start(int index)* und *end(int index)* der Klasse *Matcher* liefern Zeichensequenz, Start und Endposition einer Gruppe.



## Gruppen und partielles Matching

```
private static void capturingGroups1() {  
    p("Gruppen eines Matches ausgeben");  
    // Teilketten, die mit mindestens einer Ziffer beginnen  
    // gefolgt von mindestens einem Buchstaben oder Punkt gefolgt von  
    // mindestens einer Ziffer  
  
    String regex = "((\\d+?)([a-zA-Z.]+(\\d+?)))";  
    String str = "123aaaaXX.klaa756";  
    Pattern pt = Pattern.compile(regex);  
    Matcher matcher = pt.matcher(str);  
  
    while (matcher.find()) {  
        p(matcher.group(0));  
        p(matcher.group(1));  
        p(matcher.group(2));  
        p(matcher.group(3));  
        p(matcher.group(4));  
    }  
}
```



```
123aaaaXX.klaa7  
123aaaaXX.klaa7  
123  
aaaaXX.klaa7  
7
```

→ Package allgemein Klasse RegexDemo



# Gruppen und partielles Matching

```
private static void capturingGroups1_1() {  
    p("Gruppen aller Matches ausgeben");  
    // Alle Teilketten, die mit mindestens einer Ziffer beginnen  
    // gefolgt von mindestens einem Buchstaben oder Punkt gefolgt von  
    // mindestens einer Ziffer  
  
    String regex = "((\\d+?)([a-zA-Z.]+(\\d+?)))";  
    String str = "123aaaaXX.klaa756";  
    Pattern pt = Pattern.compile(regex);  
    Matcher matcher = pt.matcher(str);  
  
    for (int i = 0;;) {  
        while (matcher.find(i)) {  
            p(matcher.group(0));  
            p(matcher.group(1));  
            p(matcher.group(2));  
            p(matcher.group(3));  
            p(matcher.group(4));  
            i = matcher.start() + 1;  
        }  
        break;  
    }  
}
```



Gruppen aller Matches ausgeben

123aaaaXX.klaa7

123aaaaXX.klaa7

123

aaaaXX.klaa7

7

23aaaaXX.klaa7

23aaaaXX.klaa7

23

aaaaXX.klaa7

7

3aaaaXX.klaa7

3aaaaXX.klaa7

3

aaaaXX.klaa7


7





## Gruppen (capturing groups)

```
private static void capturingGroups2() {  
    p("Gruppen eines Matches ausgeben");  
    // Teilketten, die mit genau Zeichen beginnen, gefolgt von einem  
    // Leerzeichen, gefolgt von genau 3 Zeichen, gefolgt von einem  
    // Leerzeichen gefolgt von genau drei Zeichen  
    String regex = "((.{2})\\s(.{3})\\s(.{3}))";  
    Pattern pt = Pattern.compile(regex);  
    Matcher matcher = pt  
        .matcher("--das-matched-nicht--Ab zwo Uhr--das-matched-nicht--");  
    while (matcher.find()) {  
        p(matcher.group(0));  
        p(matcher.group(1));  
        p(matcher.group(2));  
        p(matcher.group(3));  
        p(matcher.group(4));  
    }  
}
```



Ab zwo Uhr  
Ab zwo Uhr  
Ab  
zwo Uhr  
Uhr

→ Package allgemein Klasse RegexDemo



# Parsen einer URL aus einem HTML Tag

- Gegeben der HTML Tag  
`<a href="http://haw-hamburg.de" HAW >`
- **Aufgabe:** Es soll die URL aus dem HTML Tag geparkt werden.
- Das Muster dazu lautet:  
`"<a href=\\\"(.*)\\\".*?>"`
- `(.*)` matched mit der URL.
- Wir suchen mit einem **Matcher** nach der Übereinstimmung und geben dann die erste Gruppe aus.



## Parsen einer URL aus einem HTML Tag

```
private static void urlParse() {  
    p("Capturing Groups zum extrahieren von Teilzeichenketten");  
    String regex = "<a href=\"(.*)\".*?>";  
    Pattern pt = Pattern.compile(regex);  
    Matcher matcher = pt  
        .matcher("<a href=\"http://haw-hamburg.de\" HAW >");  
    while (matcher.find()) {  
        p(matcher.group(1));  
    }  
}
```



<http://haw-hamburg.de>

→ Package allgemein Klasse RegexDemo



# Non-capturing groups

- Manchmal benötigen wir die Gruppenklammern nur um eine Gruppe zu identifizieren, sind aber nicht am Inhalt der Gruppe interessiert.
- Dann verwenden wir non-capturing-groups, die mit „?“ eingeleitet werden.
- Non-capturing-groups werden bei der Zählung der Gruppen **nicht mitgezählt**.
- **Beispiel:** Sie sollen eine zusammenhängende Ziffernfolge aus einer Eingabe extrahieren. Die Zahl ist als ganze Zahl oder als Gleitkommazahl aufgebaut. In der Gruppe mit dem Dezimalkomma interessieren nur die Nachkommastellen und nicht das Komma. Daher ist diese Gruppe eine non-capturing group.

```
String eingabe1 = "Sie haben      1000000,99      € gewonnen.";
String eingabe2 = "Sie haben      1000000      € gewonnen.";
Pattern pt = Pattern.compile("(\\D+)\\s*(\\d*) (?:, (\\d*))?\\s*(\\.*)\\.");
```



# Reguläre Ausdrücke

- Definition, erste Beispiele
- Zeichen in regulären Ausdrücken
- Vollständige Übereinstimmung
- Effizienz durch Kompilieren von Mustern
- Partielle Übereinstimmung
- Gierige (greedy) und bescheidene (reluctant) Operatoren
- Gruppen
- ***Gruppen und Rückverweise***
- Wortgrenzen
- Anfang und Ende einer Zeichenfolge
- Einsammeln von Matches
- Zeichenketten zerlegen



# Gruppen und Rückverweise

- **Aufgabestellungen:**

1. Wir suchen in einer Folge von Zahlen, nach den Zahlen, die am Anfang und Ende die gleiche Ziffer haben.
  2. Wir suchen nach den Elementen, die zwischen zwei HTML Tags eingebettet sind.
  3. Wir vertauschen in einem String Vorname und Nachname.
- 
- ➔ Wir benötigen in den regulären Ausdrücken „Variablen“, um Zeichenfolgen zu referenzieren. Die Variablen sind in den regulären Ausdrücken die Gruppen, die über ihre Nummer eindeutig identifiziert werden können.



# Gruppen und Rückverweise

- Auf ***capturing groups*** kann man in einem regulären Ausdruck Bezug nehmen. Dies nennt man auch ***Rückverweise***.
- Dazu wird der Nummer der Gruppe ein Backslash vorangestellt. Also z.B. ***1*** um auf die erste Gruppe Bezug zu nehmen.
- In den ***replace*** Methoden von ***String*** und ***Matcher*** ist es ebenfalls möglich auf einzelne ***capturing groups*** Bezug zu nehmen.
- Hier wird dann der Nummer der ***capturing group*** ein Dollar Zeichen (***\$***) vorangestellt.

# 1. Zahlen mit gleichen Ziffern am Anfang und Ende



- In " 12121 34564 676786 " sollen alle Zahlen gefunden werden, die mit der gleichen Ziffer beginnen und enden.
- Die Leerzeichen markieren Anfang und Ende einer Zahl. **Muster:** `"\s+?"`.
- Die erste Ziffer fassen wir zu einer Gruppe zusammen: `"(1d)"`
- Auf die erste Ziffer können beliebig viele Ziffern folgen: `"1d*"`
- Die letzte Ziffer einer Zahl muss mit der ersten übereinstimmen. Dies erreichen wir durch Rückverweis auf die erste Gruppe, der mindestens ein Leerzeichen folgt: `"111\s+?"`
- Das vollständige Muster für den regulären Ausdruck: `"\s+?(1d)1d*111\s+?"`.
- Da wir an der Zahl interessiert sind, fassen wir den Ausdruck für die Zahl zu einer Gruppe zusammen. Dadurch wird die Gruppe 1 zu Gruppe 2: `"\s+?((1d)1d*112)\s+?"`





# 1. Zahlen mit gleichen Ziffern am Anfang und Ende

```
private static void capturingBackreferences() {  
    p("Rückverweise");  
    // Alle Zahlen, die mit der gleichen Ziffern beginnen und enden  
    String regex = "\\s+?((\\d)\\d*\\2)\\s+?";  
    Pattern pt = Pattern.compile(regex);  
    Matcher matcher = pt.matcher(" 12121  34564  676786 ");  
    while (matcher.find()) {  
        p(matcher.group(1));  
    }  
}
```



12121  
676786

→ Package allgemein Klasse RegexDemo



## 2. Elemente zwischen HTML Tags finden

- Wir suchen nach Elementen zwischen öffnenden und schließenden HTML Tags.
- **Bsp.:** Aus `<h1> Dies ist eine Überschrift </h1>` soll *Dies ist eine Überschrift* extrahiert werden.
- Wir benötigen ein Gruppe für den Namen des Tag, der in spitzen Klammern `<>` eingeschlossen sein muss. Der Name kann nur Buchstaben und Ziffern enthalten. Muster: `"<(llw+?)>,"`
- Durch einen Rückverweis auf die Gruppe (hier Gruppe 1 `"ll1"`), dem wir das Zeichen für dem schließenden Tag (`"/"`) voranstellen, beschreiben wird den zugehörigen schließenden Tag: `"</ll1>"`
- Dann fassen wir die Zeichen zwischen dem öffnenden und schließenden Tag zu einer weiteren Gruppe zusammen. Muster: `".*?"`
- Insgesamt: `"<(llw+?)>(.*)</ll1>"`



## 2. Elemente zwischen HTML Tags finden

```
private static void htmlTagParse() {  
    p("HTML Tag Parse");  
    String regex = "<(\\w+?)>(.*?)</\\1>";  
    Pattern pt = Pattern.compile(regex);  
    String htmlText = "<bf> Dies ist hervorgehoben </bf> Dies ist normaler Text "  
        + "<h1> Dies ist eine Überschrift</h1>";  
    Matcher matcher = pt.matcher(htmlText);  
    while (matcher.find()) {  
        p(matcher.group(0));  
        p(matcher.group(1));  
        p(matcher.group(2));  
    }  
}
```

*Von Interesse ist nur Gruppe 2  
Gruppe 0 und 1 werden zu  
Demonstrationszwecken mit  
ausgegeben.*



```
<bf> Dies ist hervorgehoben </bf>  
bf  
    Dies ist hervorgehoben  
<h1> Dies ist eine Überschrift</h1>  
h1  
    Dies ist eine Überschrift
```

→ Package allgemein Klasse RegexDemo



### 3. Nachname und Vorname vertauschen

- Gegeben die Zeichenkette: "**<Nachname>, <Vorname>**".
- **Aufgabe:** Es sollen **<Nachname>** und **<Vorname>** miteinander vertauscht werden und dabei das Komma eliminiert werden.
- **Beispiel:** Aus "**Hastig, Hugo**" wird "**Hugo Hastig**".
- **Lösung:**
  - Wir suchen in dem **String** nach zwei Zeichenfolgen, die nur Buchstaben ( "**\p{Alpha}+**" ) enthalten, von beliebig vielen Leerzeichen umgeben sein können ( "**\\s\***" ) und durch ein Komma getrennt sind und merken uns die Folgen in jeweils einer Gruppe:  
**"(\p{Alpha}+) \\s\*, \\s\*(\p{Alpha}+)"**
  - Wir vertauschen im **replaceAll** die erste ( "**\$1**" ) mit der zweiten Gruppe ( "**\$2**" ) und eliminieren das Komma.



### 3. Nachname und Vorname vertauschen

```
private static void changeNameSurname() {  
    p("Vertauschen von Name und Vorname");  
    String str = "Hastig, Hugo";  
    p(str);  
    Pattern pt = Pattern.compile("(\\p{Alpha}+)\\s*,\\s*(\\p{Alpha}+)");  
    Matcher match = pt.matcher(str);  
    p(match.replaceAll("$2 $1"));  
}
```



Hastig, Hugo  
Hugo Hastig

→ Package allgemein Klasse RegexDemo



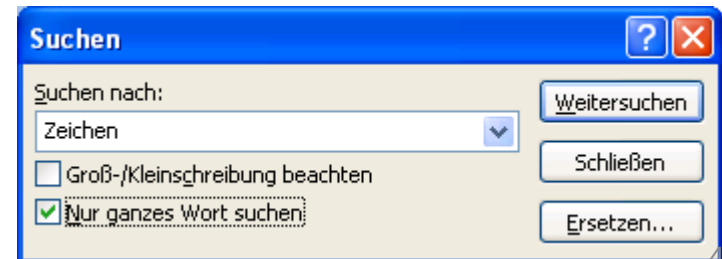
# Reguläre Ausdrücke

- Definition, erste Beispiele
- Zeichen in regulären Ausdrücken
- Vollständige Übereinstimmung
- Effizienz durch Kompilieren von Mustern
- Partielle Übereinstimmung
- Gierige (greedy) und bescheidene (reluctant) Operatoren
- Gruppen
- Gruppen und Rückverweise
- **Wortgrenzen**
- Anfang und Ende einer Zeichenfolge
- Einsammeln von Matches
- Zeichenketten zerlegen



# Wortgrenzen

- Wir betrachten erneut das Beispiel der Zahlenfolge: " **12121 34564 676786** "
- Um Wortgrenzen der Zahlen erkennen zu können, setzen wir voraus, dass diese von Leerzeichen umgeben sind: "**`\s+?(1\d)1\d*\s+?`**"
- Alternativ können wir Anfang und Ende eines Wortes mit Zeichen für Wortgrenzen beschreiben: **`1b`** ist das Zeichen, das eine Wortgrenze beschreibt (Anfang oder Ende). **`1B`** beschreibt eine Nicht-Wortgrenze.
- Mit Wortgrenzen lässt sich der reguläre Ausdruck umschreiben in: "**`1b(1\d)1\d*\s+111b`**"
- Wortgrenzen kommen bei der Textsuche in Editoren zum Einsatz, wenn z.B. nach ganzen Wörtern gesucht wird.





## Zahlen mit gleichen Ziffern an Wortgrenzen

```
private static void wordBoundaries() {  
    p("Wortgrenzen Rückverweise");  
    // Zahlen, die mit den gleichen Ziffern beginnen und enden  
    String regex = "\\b((\\d)\\d*\\2)\\b";  
    Pattern pt = Pattern.compile(regex);  
    Matcher matcher = pt.matcher("12121 34564 676786");  
    while (matcher.find()) {  
        p(matcher.group(1));  
    }  
}
```

*Wir benötigen jetzt keine  
Leerzeichen vor der ersten  
und nach der letzten Zahl*



12121  
676786

→ Package allgemein Klasse RegexDemo





# Kalendereinträge

- **Beispiel:** Sie sollen die Kalendereinträge für den **2.11.2015** aus einer Zeichenkette lesen. Einträge beginnen mit einem Datum, dem folgen von-bis Zeiten und eine Beschreibung. Alle Elemente sind durch Komma getrennt. Zeilen enden auf **"\n"**.

```
String calendar = "2.11.2015, 12:30, 13:30, Labor\n"  
+ "2.11.2015, 14:00, 16:00, Masterkurs\n"  
+ "12.11.2015, 8:15, 11:30, Praktikum\n"  
+ "nonsense 2.11.2015, 9:00, 15:00, Bremen\n";
```

- In dem Kalender befinden sich auch ein Eintrag für den 12. November.
- Suchen wir mit dem Ausdruck **"211.1111.201511s\*,(.\*)\n"**, dann werden **alle** Kalendereinträge gefunden, auch die **nonsense** Zeile.
- Ändern wir den regulären Ausdruck in **"11b211.1111.200911s\*,(.\*)\n"** dann werden nur noch alle Einträge für den 2.11.2015 gefunden, aber leider auch immer noch die **nonsense** Zeile.
- **Lösung:** Wir brauchen Sonderzeichen, um den Satzanfang zu beschreiben.



# Kalendereinträge

```
private static void wordBoundaries2() {  
    String calender = "2.11.2015, 12:30, 13:30, Labor\n"  
        + "2.11.2015, 14:00, 16:00, Masterkurs\n"  
        + "12.11.2015, 8:15, 11:30, Praktikum\n"  
        + "nonsense 2.11.2015, 9:00, 15:00, Bremen\n";  
    String regex1 = "2\\.11\\.2015\\s*,(.*?)\\n";  
    Matcher matcher = Pattern.compile(regex1).matcher(calender);  
    while (matcher.find()) {  
        p(matcher.group(1));  
    }  
    println();  
    String regex2 = "\\b2\\.11\\.2015\\s*,(.*?)\\n";  
    matcher = Pattern.compile(regex2).matcher(calender);  
    while (matcher.find()) {  
        p(matcher.group(1));  
    }  
}
```



12:30, 13:30, Labor  
14:00, 16:00 Masterkurs  
8:15, 11:30, Praktikum  
9:00, 15:00, Bremen

→ *Package allgemein Klasse RegexDemo*

12:30, 13:30, Labor  
14:00, 16:00 Masterkurs  
9:00, 15:00, Bremen



# Reguläre Ausdrücke

- Definition, erste Beispiele
- Zeichen in regulären Ausdrücken
- Vollständige Übereinstimmung
- Effizienz durch Kompilieren von Mustern
- Partielle Übereinstimmung
- Gierige (greedy) und bescheidene (reluctant) Operatoren
- Gruppen
- Gruppen und Rückverweise
- Wortgrenzen
- ***Anfang und Ende einer Zeichenfolge***
- Einsammeln von Matches
- Zeichenketten zerlegen



# Anfang (^) und Ende (\$) einer Zeichenfolge

- **Beispiel:** Sie lesen aus einer Komma-separierten Datei mit 4 Zeilen zeilenweise einzelne Kalendereinträge. Sie sollen die Einträge für den **2.11.2015** einsammeln.  
2.11.2015, 12:30, 13:30, Labor  
2.11.2015, 14:00, 16:00, Masterkurs  
12.11.2015, 8:15, 11:30, Praktikum  
nonsense 2.11.2015, 9:00, 15:00, Bremen
- Die Einträge für den 12.11. und die nonsense Zeile sollen nicht gelesen werden.
- Wenn wir ausnutzen, dass die Zeichenfolgen mit dem Datum beginnen und wir diesen Anfang (die Satzgrenze) mit ^ markieren, dann werden nur noch die für den 2.11. gültigen Einträge gelesen.
- Der Ausdruck ändert sich in: **"^211.1111.2015|ls\*,(.\*)|ls\*"**



## Anfang und Ende einer Zeichenfolge

```
private static void calendarFromFile() {  
    try (BufferedReader calReader = new BufferedReader(new FileReader(  
        new File("calendar"))));) {  
        String calEntry;  
        Pattern pattern = Pattern.compile("^2\\.11\\.2015\\s*,(.*)\\s*,.+?");  
        while ((calEntry = calReader.readLine()) != null) {  
            Matcher matcher = pattern.matcher(calEntry);  
            while (matcher.find()) {  
                p(matcher.group(1));  
            }  
        }  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```



12:30, 13:30, Labor  
14:00, 16:00 Masterkurs  
8:15, 11:30, Praktikum  
9:00, 15:00, Bremen

→ Package allgemein Klasse RegexDemo

12:30, 13:30, Labor  
14:00, 16:00 Masterkurs



# Sonderzeichen für Wort- und Zeichenketten-Gsrenzen

Zeichen	Bedeutung
\b	Wortgrenze Anfang/Ende eines Wortes
\B	keine Wortgrenze
^	Anfang einer Zeichenkette
\$	Ende einer Zeichenkette



# Reguläre Ausdrücke

- Definition, erste Beispiele
- Zeichen in regulären Ausdrücken
- Vollständige Übereinstimmung
- Effizienz durch Kompilieren von Mustern
- Partielle Übereinstimmung
- Gierige (greedy) und bescheidene (reluctant) Operatoren
- Gruppen
- Gruppen und Rückverweise
- Wortgrenzen
- Anfang und Ende einer Zeichenfolge
- ***Einsammeln von Matches***
- Zeichenketten zerlegen



# Einsammeln von Match Ergebnissen

- Manchmal benötigt man alle Ergebnisse, die das **Matcher** Objekt mit wiederholtem **find** einsammelt.
- Die Klasse **MatchResult** merkt sich **ein** Teilergebnis eines Matchvorganges.
- Ein **Matcher** Objekt wandelt mit der Methode **toMatchResult** ein Teilergebnis in ein **MatchResult** Objekt um.
- In **MatchResult** lässt sich dann das Ergebnis mit Hilfe der **group()**, **start()**, **end()** Methoden analysieren.
- Sammelt man nun diese **MatchResult** Objekte während eines Matchvorganges (**while-find** Schleife) ein, dann kann der gesamte Vorgang rekonstruiert werden.
- Im nachfolgenden Beispiel werden die Klassen und Interfaces **Iterable**, **List** und **ArrayList** verwendet. (Diese werden in einer der nächsten Vorlesung eingeführt, dann wird auch die Bedeutung der spitzen Klammern erklärt).





## Einsammeln von Match Ergebnissen

```
private static Iterable<MatchResult> findMatches( String pattern, CharSequence s )
{
    List<MatchResult> results = new ArrayList<MatchResult>();

    for ( Matcher m = Pattern.compile(pattern).matcher(s); m.find(); )
        results.add( m.toMatchResult() );

    return results;
}
```

---

```
String str = "<b> Moin </b> Folks <b> Wake Up </b>";
```

```
for (MatchResult mr : findMatches("<b>.*?</b>", str)) {
    p( mr.group() + " von " + mr.start() + " bis " + mr.end() );
}
```



```
<b> Moin </b> von 0 bis 13
<b> Wake Up </b> von 20 bis 36
```

→ Package allgemein Klasse CollectMatchResults



# Reguläre Ausdrücke

- Definition, erste Beispiele
- Zeichen in regulären Ausdrücken
- Vollständige Übereinstimmung
- Effizienz durch Kompilieren von Mustern
- Partielle Übereinstimmung
- Gierige (greedy) und bescheidene (reluctant) Operatoren
- Gruppen
- Gruppen und Rückverweise
- Wortgrenzen
- Anfang und Ende einer Zeichenfolge
- Einsammeln von Matches
- ***Zeichenketten zerlegen***



# Zeichenketten zerlegen

- Wenn wir Zeichenketten zerlegen unterscheiden wir **Token**, die Worte zwischen Trennzeichen, von den **Trennzeichen** (engl. delimiter).
- Eine Zerlegung entfernt die Trennzeichen aus der Original-Zeichenkette und liefert die Liste der Token zurück.
- Die Methode *split* der Klassen *String* und *Pattern* zerlegt Zeichenketten. Die Trennzeichen können über reguläre Ausdrücke beschrieben werden.
- Die Klasse *Scanner* verfügt über eine Reihe von Methoden, um Zeichenketten aus Dateien zu lesen und zu zerlegen. Auch hier können die Trennzeichen über reguläre Ausdrücke beschrieben werden.



# Strings mit *split* zerlegen

- Die Methode *split(String regex)* der Klasse String zerlegt einen String in Tokens zwischen den Delimitern (Begrenzern). Delimiter werden als reguläre Ausdrücke formuliert (*regex*), die *split* als Argument übergeben werden. Das Ergebnis von *split* ist ein Array mit *String* Objekten.
- **Beispiel 1:** Wir zerlegen eine Zeichenkette mit komma-separierten Token in Teilstrings (Delimiter ist *","*).
- **Beispiel 2:** Wir verwenden mehr als ein Trennzeichen für die Zerlegung einer Zeichenkette. Die Delimiter sind *","* und *"\$"*, die wir oder-verknüpfen: *"/|"\$"*
- **Beispiel 3:** Die Leerzeichen vor/nach den Trennzeichen aus Beispiel 1 sollen bei der Zerlegung mit entfernt werden.
- **Beispiel 4:** Hier sollen bei der Zerlegung die Leerzeichen aus Beispiel 2 entfernt werden.



## Strings mit *split* zerlegen

```
String str = "Eins $ one , zwei $ two , drei $ three";

String[] colonSep = str.split(",");
String[] multiSep = str.split(",|\\$");
String[] colonWithSpaceSep = str.split("\\s*,\\s*");
String[] multiWithSpaceSep = str.split("\\s*,\\s*|\\s*\\$\\s*");

p(Arrays.deepToString(colonSep));
p(Arrays.deepToString(multiSep));
p(Arrays.deepToString(colonWithSpaceSep));
p(Arrays.deepToString(multiWithSpaceSep));
```



```
[Eins $ one , zwei $ two , drei $ three]
[Eins , one , zwei , two , drei , three]
[Eins $ one, zwei $ two, drei $ three]
[Eins, one, zwei, two, drei, three]
```

→ Package zerlegung Klasse StringSplitDemo



# *CharSequence* mit *Pattern.split* zerlegen

- Gründe für die Verwendung von *Pattern.split*
  - Die Methode ist für den allgemeinen Typ *CharSequence* implementiert.
  - Es ist möglich auch *StringBuffer*, *StringBuilder* und *CharBuffer* Objekte ohne eine teure Umwandlung in einen *String* zu zerlegen.
  - Die Methode ist effizienter, wenn ein Delimiter-Muster mehrfach angewendet werden soll, da das Muster dann nur einmal kompiliert werden muss.



## *CharSequence* mit *Pattern.split* zerlegen

```
String str = "Eins $ one  , zwei $ two  , drei $ three";

Pattern colonSep = Pattern.compile(",");
Pattern multiSep = Pattern.compile(",|\\$");
Pattern colonWithSpaceSep = Pattern.compile("\\s*,\\s*");
Pattern multiWithSpaceSep = Pattern.compile("\\s*,\\s*|\\s*\\$\\s*");

p(Arrays.deepToString(colonSep.split(new StringBuilder(str))));
p(Arrays.deepToString(multiSep.split(str)));
p(Arrays.deepToString(colonWithSpaceSep.split(str)));
p(Arrays.deepToString(multiWithSpaceSep.split(new StringBuffer(str))));
```



```
[Eins $ one  , zwei $ two  , drei $ three]
[Eins , one  , zwei , two  , drei , three]
[Eins $ one, zwei $ two, drei $ three]
[Eins, one, zwei, two, drei, three]
```

→ Package zerlegung Klasse `StringSplitDemo`



# Zeichenketten mit *Scanner* zerlegen

- *Scanner* ist ein mächtiges Werkzeug um Zeichenketten und -ströme zu zerlegen.
- Mit den Konstruktoren lassen sich *Scanner* Objekte auf unterschiedlichen Zeichenquellen definieren. Die wichtigsten sind *String*, *File* und *InputStream* (z.B. *System.in* )
- Mit *hasNextLine* und *nextLine* lässt sich eine Datei zeilenweise lesen.
- *Scanner* iteriert über Zeichenquellen z.B. unter Verwendung der Methoden *hasNext* und *next*. *next* liefert das nächste Token. Die Zerlegung in Token erfolgt auf Basis des verwendeten Trennzeichen.
- Trennzeichen werden mit *useDelimiter(String regex)* gesetzt.
- *Scanner* hat desweiteren spezielle Methoden um Werte von Basisdatentypen zu lesen: *hasNext<BaseType>* und *next<BaseType>*. Bsp. (*hasNextInt()*, *nextInt()*)





# Zeichenketten mit *Scanner* zerlegen


- **Beispiel 1:** Trennzeichen sind Sequenzen von Leerzeichen
- **Beispiel 2:** Trennzeichen sind oder-verknüpfte Zeichen (**%,\$,&**), denen beliebig viele Leerzeichen vorangehen oder folgen können.
- Die Beispiele speichern die Ergebnisse in einer Liste von *MatchResult* Objekten (*ArrayList<MatchResult>*). (siehe auch Folien [55-56](#))
- Wenn *Scanner.next* erfolgreich war, dann wird das Match-Ergebnis gespeichert und kann mit der Methode *match* abgefragt werden.



## Beispiel 1

```
ArrayList<MatchResult> allMatches = new ArrayList<MatchResult>();  
// Beispiel 1: Sequenzen von Leerzeichen sind Trenner  
p("Beispiel 1");  
String str1 = "Ein    Tag    im Jahr    der    Schlange";  
Scanner scanner = new Scanner(str1);  
scanner.useDelimiter("\\s+");  
while (scanner.hasNext()){  
    scanner.next();  
    allMatches.add(scanner.match());  
}  
printMatches(allMatches);  
println();
```

### Beispiel 1



Ein	0-3
Tag	7-10
im	14-16
Jahr	17-21
der	24-27
Schlange	30-38

```
private static void printMatches(List<MatchResult> allMatches) {  
    for (MatchResult match : allMatches) {  
        p(String.format("%-12s %d-%d", match.group(), match.start(),  
            match.end()));  
    }  
}
```

→ Package zerlegung.scanner Klasse ScannerSplitStringDemo



## Beispiel 2

```
// Beispiel 2
String str2 = "b2345 %    Computer $    200 & Drucker $ 200";
p("Beispiel 2");
scanner = new Scanner(str2);
allMatches = new ArrayList<MatchResult>();
scanner.useDelimiter(Pattern.compile("\\s*(%|\\$|&)\\s*"));
while (scanner.hasNext()) {
    scanner.next();
    allMatches.add(scanner.match());
}
printMatches(allMatches);
```



Beispiel 2	
b2345	0-5
Computer	11-19
200	25-28
Drucker	32-39
200	42-45

```
private static void printMatches(List<MatchResult> allMatches) {
    for (MatchResult match : allMatches) {
        p(String.format("%-12s %d-%d", match.group(), match.start(),
            match.end()));
    }
}
```

→ Package zerlegung.scanner Klasse ScannerSplitStringDemo

# Dateien über *Scanner* einlesen und auswerten



- Beispiel:
  - Die Datei *a5datei* soll zeilenweise gelesen werden.
  - Leerzeilen sollen übersprungen werden.
  - Jede Zeile soll in ein Objekt der Klasse *Bestellung* übersetzt werden.
  - Eine *Bestellung* hat eine *Bestellnummer* und verwaltet eine Liste von Objekten der Klasse *Position*.
  - Führende Leerzeichen in der *Bestellnummer* müssen eliminiert werden.
  - Alle Bestellungen sollen in einer Liste eingesammelt werden.
  - Für alle Klassen ist eine *toString()* Methode zu schreiben.



## Inhalt von *a5datei*

```
b132568 % Joghurt $ 10000 & Kaese $ 6000
```

```
b132569% Motorroller $ 15 & Fahrraeder$ 500 & Blades $ 67899900
```

*Die Datei beginnt mit 2 Leerzeilen, gefolgt von einem Eintrag,  
gefolgt von einer Leerzeile, gefolgt von einem Eintrag*



## *a5datei* mit *Scanner* zeilenweise lesen

```
// a5datei liegt im Rootverzeichnis des Javaprojektes
try (Scanner scanner = new Scanner(new File("a5datei"))) {
    // Zeilen einlesen
    while (scanner.hasNextLine()) {
        String line = scanner.nextLine();
        p(line);
    }
}
```



```
b132568  %   Joghurt $ 10000 &   Kaese $ 6000
b132569%   Motorroller    $ 15 &   Fahrraeder$ 500 &       Blades $ 67899900
```

*Zeilenweises Auslesen  
reproduziert den Inhalt  
von a5datei.*



# Leerzeilen überspringen mit *Scanner skip*

- Die Methode *skip* der Klasse *Scanner* überspringt alle Zeichen, die mit einem bestimmten Muster übereinstimmen.
- Um Leerzeilen zu überspringen,
  - brauchen wir einen regulären Ausdruck für eine Leerzeile als Argument für *skip ("\\s\*")*, den wir aus Effizienzgründen kompilieren (*Pattern.compile("\\s\*")*).
  - müssen wir überprüfen, ob noch eine Zeile vorhanden ist (*scanner.hasNextLine()*)
  - müssen wir sicherstellen, dass nach dem Überspringen noch eine weitere Zeile gelesen werden kann (*&&scanner.skip(leerZeile).hasNextLine()*).
- Dann ist die nächste gelesene Zeile (*scanner.nextLine()*) sicher keine Leerzeile.



## Leerzeilen in *a5datei* überspringen

```
// a5datei liegt im Rootverzeichnis des Javaprojektes
try (Scanner scanner = new Scanner(new File("a5datei"))) {
    Pattern leerZeile = Pattern.compile("\\s*");
    // Zeilen einlesen und Leerzeilen überspringen
    while (scanner.hasNextLine()
        && scanner.skip(leerZeile).hasNextLine()) {
        String line = scanner.nextLine();
        p(line);
    }
}
```



b132568	%	Joghurt	\$ 10000	&	Kaese	\$ 6000	
b132569	%	Motorroller	\$ 15	&	Fahrraeder	\$ 500	Blades \$ 67899900

→ Package zerlegung.scanner Klasse ScanBestellungenEinfach





## Vorarbeiten: Klassen *Bestellung* / *Position*

```
public class Bestellung {  
    private String bestnr;  
    private ArrayList<Position> positionen= new ArrayList<Position>();  
  
    public Bestellung(String bestnr){  
        this.bestnr= bestnr;  
    }  
    public boolean add(Position p){  
        return positionen.add(p);  
    }  
    @Override  
    public String toString() {  
        return String.format("B(%1$s,%2$s)", bestnr, positionen);  
    }  
}
```

→ Package bestellung



## Vorarbeiten: Klasse *Bestellung*

```
public class Bestellung {  
    private String bestnr;  
    private ArrayList<Position> positionen= new ArrayList<Position>();  
  
    public Bestellung(String bestnr){  
        this.bestnr= bestnr;  
    }  
    public boolean add(Position p){  
        return positionen.add(p);  
    }  
    @Override  
    public String toString() {  
        return String.format("B(%1$s,%2$s)", bestnr, positionen);  
    }  
}
```

→ Package bestellung



## Vorarbeiten: Klasse *Position*

```
public class Position {
    private String name;
    private int menge;
    private String bemerkung;
    public Position(String name,int menge,String bemerkung) {
        this.name = name;
        this.menge = menge;
        this.bemerkung = bemerkung;
    }
    public Position(String name,int menge) {
        this(name,menge,"");
    }
    public String getName() {
        return name;
    }
    public int getMenge() {
        return menge;
    }
    public String getBemerkung(){
        return bemerkung;
    }
    public String toString() {
        return String.format("P(%1$s,%2$d,%3$s)", name,menge,bemerkung);
    }
}
```



## Mit *Scanner hasNextXXX, nextXXX* Token einer Zeile verarbeiten

- Für jede Zeile, die *Scanner.nextLine()* liest, benötigen wir wiederum einen Scanner, der die Token der Zeile liest (*lineScanner*).
- Token in den Zeilen werden begrenzt durch
  - die Zeichen %, \$, & (Muster: "(%|\\\$\\&)" )
  - das Endezeichen eines Strings \$ (Muster "(%|\\\$\\&|\$)").
- Diesen Zeichen können beliebig viele Leerzeichen vorausgehen und folgen
  - (Muster "\\s\*(%|\\\$\\&|\$)\\s\*").
- Das Muster kompilieren wir (Effizienz) und übergeben es als Trennzeichen an den *lineScanner*.

# Mit *Scanner hasNextXXX, nextXXX* Token einer Zeile verarbeiten



- Das erste Token einer Zeile entspricht der Bestellnummer.
- Die folgenden Token sind Paare aus Name und Menge für Positionen.
- Da die Nummer immer an zweiter Stelle steht, wissen wir,
  - wenn das nächste Token vom Typ *int* ist (*lineScanner.hasNextInt()*), dann hält die Variable *name* den Wert des vorausgehenden Token.
  - Hier können wir ein Objekt *Position* mit *name* und einem *int* Wert für die Menge (*lineScanner.nextInt()*) erzeugen.
- Nachdem eine Zeile abgearbeitet ist, fügen wir die *Bestellung* der Liste der Bestellungen hinzu.

→ Package zerlegung.scanner Klasse ScanBestellungenEinfach



```
List<Bestellung> bestellListe = new ArrayList<Bestellung>();
try (Scanner scanner = new Scanner(new File("a5datei"))) {
    Pattern leerZeile = Pattern.compile("\\s*");
    // Muster um die Token einer Zeile zu lesen
    Pattern delims = Pattern.compile("\\s*(%|\\$|&|\\$)\\s*");
    // Zeilen einlesen und Leerzeilen überspringen
    while (scanner.hasNextLine()
        && scanner.skip(leerZeile).hasNextLine()) {
        String line = scanner.nextLine();
        Scanner lineScanner = new Scanner(line);
        lineScanner.useDelimiter(delims);
        // Bestellnummer
        String bestnr = lineScanner.next();
        Bestellung best = new Bestellung(bestnr);
        // Positionen
        String name = "";
        while (lineScanner.hasNext()) {
            if (lineScanner.hasNextInt()) {
                best.add(new Position(name, lineScanner.nextInt()));
            } else {
                name = lineScanner.next();
            }
        }
        bestellListe.add(best);
    }
}
```

Package zerlegung.scanner  
Klasse ScanBestellungenEinfach



```
[B(b132568, [P(Joghurt, 10000), P(Kaese, 6000)]),
 B(b132569, [P(Motorroller, 15), P(Fahrraeder, 500), P(Blades, 67899900)])]
```



# Quellen

- Christian Ullenboom: **Java ist auch eine Insel**,  
[http://openbook.rheinwerk-verlag.de/javainsel/javainsel\\_04\\_009.html#dodtp33e02ca1-cb53-403e-92a2-dfe128dccc2](http://openbook.rheinwerk-verlag.de/javainsel/javainsel_04_009.html#dodtp33e02ca1-cb53-403e-92a2-dfe128dccc2) letzter Zugriff am 23.10.2015

