Suchen in Texten

- Sie wissen wie in einem Text gesucht werden kann
- Sie wissen wie Suchmaschinen arbeiten
- Sie können mit Regex in Java umgehen

Finden eines Teilstrings in einem String

```
D r e i   r e i n e   R e i s e n d e
R e i s
  R e i s
          . . .
                    R e i s
```

```
r e i n
  r e i n
      . . .
          r e i n
```

```
int indexOf(String str, String pattern)
```

- liefert die Position, an der das Muster beginnt
- -1 falls das Muster nicht vorkommt

Brute-Force Suche in Java

```
static int indexOf(String str, String pattern) {
 for (int i = 0; i < str.length() - pattern.length() + 1; i++) {
   int k;
   for (k = 0;
        k < pattern.length() && str.charAt(i+k) == pattern.charAt(k);
        k++);
   if (k == pattern.length()) return i;
   }
 return -1;
}
```

- Muster wird an die Position i gesetzt
- Es wird mit dem String verglichen bis
  - Ende des Musters erreicht -> Erfolg
  - Nichtübereinstimmung
- Worst-Case Aufwand ist O(n*m)

Knuth-Morris-Pratt Algorithmus

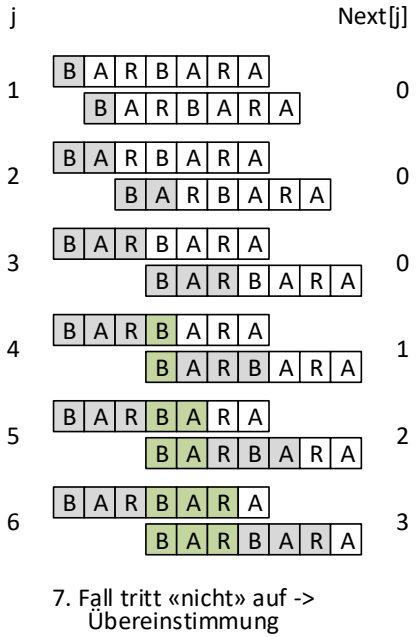
Idee: bei einem Nicht-Match das Pattern um mehr als eine Stelle verschieben, so dass allfällig auftretende Subpattern am Anfang des Patterns erkannt werden.

Bsp: Pattern = ***BARB***ARA und Text = …BAR***BARB***ARA…
Beim blauen B kommt es beim Brute Force-Verfahrung zum Abbruch, dabei haben wir eigentlich an dieser Stelle schon wieder das Subpattern BARB (kursiv & fett) am Anfang des Patterns erkannt.

- Ablauf Algorithmus in 2 Phasen:
  1. Im Pattern nach sich wiederholenden Subpattern suchen -> next-Tabelle (Startadresse in Pattern für weiteren Vergleich).
  2. Text gemäss der next-Tabelle durchsuchen.

Knuth-Morris-Pratt Algorithmus

1. Sich wiederholenden Subpattern suchen:
   - Das Pattern wird mit sich selbst verglichen. Vergleichspattern (grau hinterlegt) wird mit Länge 1 bis n gebildet.
   - Danach wird Vergleichspattern von ganz links nach rechst verschoben, bis alle überlappenden Zeichen übereinstimmen, oder keine Überlappung gefunden wurde (ist ein Teil des Vergleichspattern «Subpattern» des bisher verwendeten Patterns?)



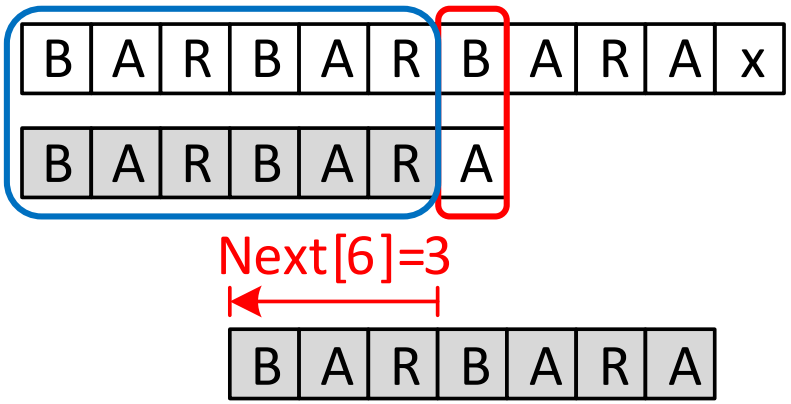
   - j: Anzahl erfolgreich verglichener Buchstaben
   - next[j]: um wieviel darf ich Pattern nach links verschieben, falls Buchstabe j + 1 abweicht.

j	1	2	3	4	5	6
Next[j]	0	0	0	1	2	3

Knuth-Morris-Pratt Algorithmus

2. Text gemäss der next-Tabelle durchsuchen:

j	1	2	3	4	5	6
Next[j]	0	0	0	1	2	3



Beim Vergleich des 7. Buchstabens (B <> A) des Patterns kommt es zur Abweichung.

Gemäss next-Tabelle darf für den weiteren Vergleich das Pattern um drei Positionen nach links verschoben werden und die Suche fortgesetzt werden…

Knuth-Morris-Pratt Algorithmus

- Sehr schlauer Algorithmus
- Laufzeit O(n+m), n = Länge Text, m = Länge Pattern

Aufgabe Knuth-Morris-Pratt Algorithmus:

Suchen Sie mit dem Algorithmus das Pattern nano im Text: nenananox

Invertierter Index

Motivation

- Wikipedia: Millionen englischer Artikel
- Query: Suche alle Artikel in denen das Wort «Twitter» vorkommt
- Wie macht man das?

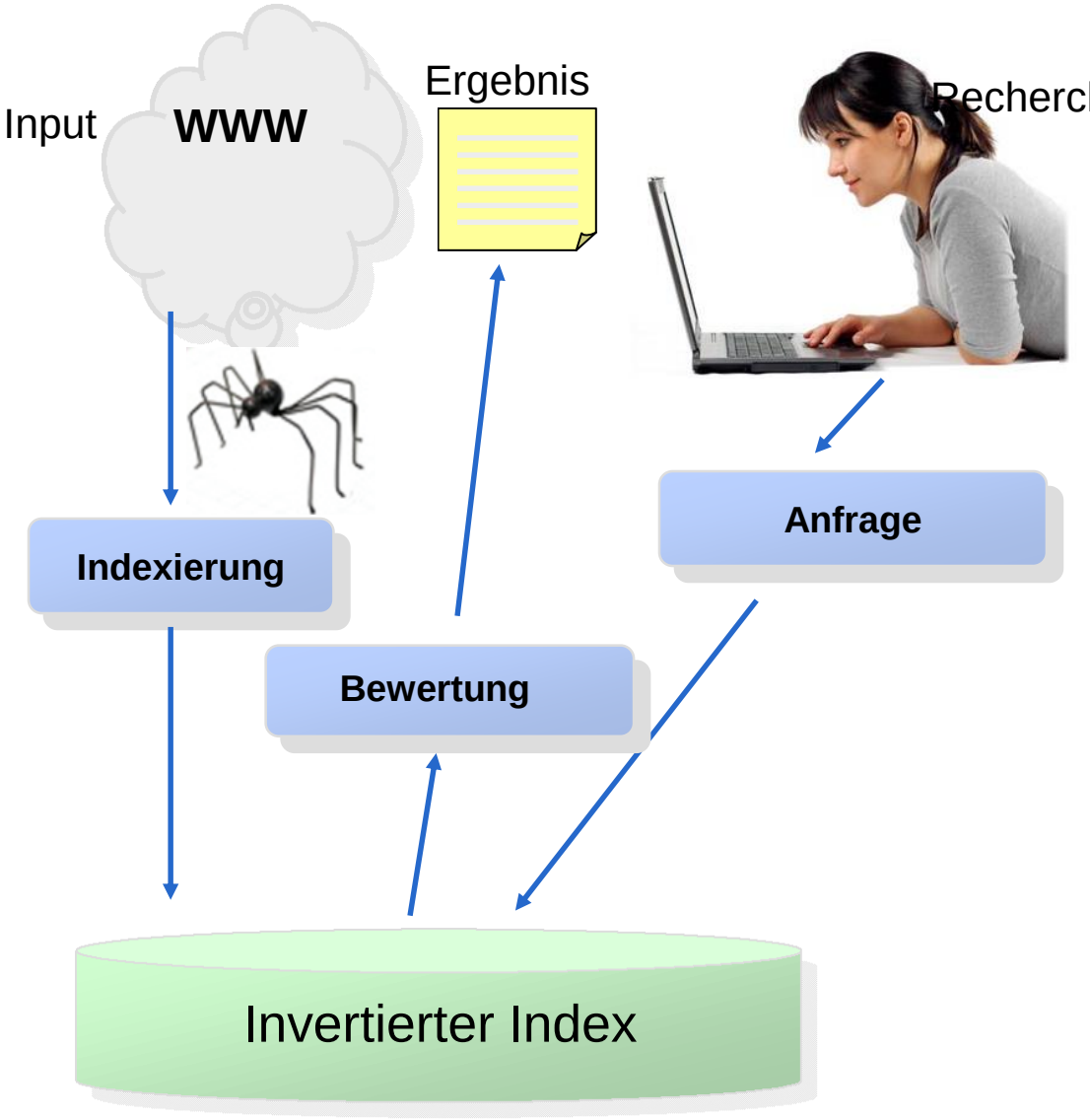
Invertierter Index

"…an **inverted index** is an index data structure storing a mapping from content, such as words or numbers, to its locations …in a document or a set of documents. " (wikipedia)

```
T[0] = "it is what it is"
T[1] = "what is it"
T[2] = "it is a banana"
```

```
"a": {(2, 2)}
"banana": {(2, 3)}
"is": {(0, 1), (0, 4), (1, 1), (2, 1)}
"it": {(0, 0), (0, 3), (1, 2), (2, 0)}
"what": {(0, 2), (1, 0)}
```

Elemente von Suchmaschinen



- **Web Roboter/Spider/ Crawler**
  - Durchlaufen regelmässig das Web nach neuen Informationen
- **Indexierung**
  - Aufbereitung von Dokumenten
  - Speicherung im Index / in der Datenbank der Suchmaschine
  - Dateisystem, das für die Suche geeignet ist
- **Retrievalsystem**
  - Suche im Index
  - Sortierung nach Relevanz
    - *Wo kommen die Suchbegriffe vor?*
    - *Wie oft kommen die Begriffe vor?*
    - *In welcher Reihenfolge?*
    - *Wie lang ist der Text?*
    - *Wie viele Links verweisen auf das Dokument?*

Ordnung, Verbesserungen?

Alle Dokumente in denen das Wort «twitter» vorkommt

- Performance?
- Verbesserungen?

Levenshtein Distanz
(Approximative Suche)

Levenshtein Distanz

Definition: Die **Levenshtein-Distanz** (auch: Editier-Distanz) von zwei Wörtern A und B ist die **minimale Anzahl** Operationen, um aus dem ersten Wort das zweite Wort zu machen.

Erlaubte "Operationen":

1. insert(c):
   Buchstaben 'c' an einer Position im ersten Wort einfügen
2. update(c->d):
   Buchstaben 'c' an einer Position im ersten Wort durch 'd' ersetzen
3. delete(c):
   Buchstaben 'c' an einer Position im ersten Wort entfernen

Einzelarbeit (2min):

1. Was ist die Levenshtein-Distanz von 'Haus' und 'Maus'?
2. Was ist die Levenshtein-Distanz von 'Saturday' und 'Sunday'?

Berechnung der Levenshtein Distanz

Gegeben zwei Wörter A und B der Länge n bzw m.

- Konsturiere eine Matrix D mit der Grösse $(n+1)x(m+1)$.
- D[i,j] gibt die Levenshtein-Distanz der **Präfixe** von A und B der Länge i bzw. j an.

B[j] zu A[i] von		***S***	***a***	t	u	r	d	a	y
	0	1	2	3	4	5	6	7	8
S	1	0	1	2	3	4	5	6	7
u	2	1	1	2	2	3	4	5	6
n	3	2	2	2	3	3	4	5	6
d	4	3	**3**	3	3	4	3	4	5
a	5	4	3	4	4	4	4	3	4
y	6	5	4	4	5	5	5	4	3

Beispiel:

A = 'Sunday', B = 'Saturday'

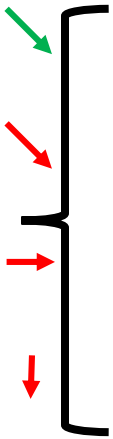
D[4,2] = 3, da 'Sa' und 'Sund' Distanz 3 haben.

Sund → Sund → Sand → Sad → Sa

0 1 1 1

Berechnung der Levenshtein Distanz

D[i,j] = min



D[i-1, j-1] + 0, falls A[i] = B[j]

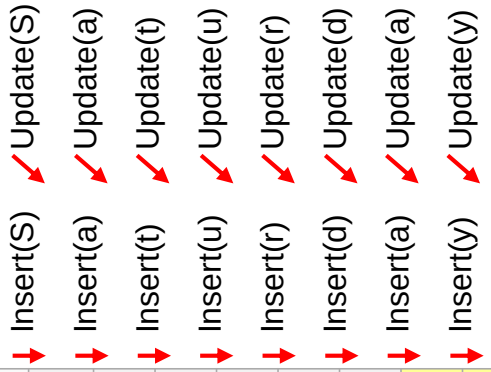
D[i-1, j-1] + 1, falls update(A[i]->B[j])

D[i, j-1] + 1, falls insert(B[j])

D[i-1, j] + 1, falls delete(A[i])

B[j] zu A[i] von		S	a	t	u	r	d	a	y
	0	1	2	3	4	5	6	7	8
S	1	0	1	2	3	4	5	6	7
u	2	1	1	2	2	3	4	5	6
n	3	2	2	2	3	3	4	5	6
d	4	3	3	3	3	4	3	4	5
a	5	4	3	4	4	4	4	3	4
y	6	5	4	4	5	5	5	4	3

Berechnung der Levenshtein Distanz



B[j] zu A[i] von		S	a	t	u	r	d	a	y
	0	1	2	3	4	5	6	7	8
S	1	0	1	2	3	4	5	6	7
u	2	1	1	2	2	3	4	5	6
n	3	2	2	2	3	3	4	5	6
d	4	3	3	3	3	4	3	4	5
a	5	4	3	4	4	4	4	3	4
y	6	5	4	4	5	5	5	4	3

B[0]=A[0]	TakeOver(S)	0
B[1]=a	Insert(a)	1
B[2]=t	Insert(t)	1
B[3]=A[1]	TakeOver(u)	0
B[4]=r	Update(r)	1
B[5]=A[3]	TakeOver(d)	0
B[6]=A[4]	TakeOver(a)	0
B[7]=A[5]	TakeOver(y)	0
Levenshtein-Distanz:		3

↓ Delete(S) ↘ TakeOver(S)

↓ Delete(u) ↘ TakeOver(u)

↓ Delete(n) ↘ TakeOver(n)

↓ Delete(d) ↘ TakeOver(d)

↓ Delete(a) ↘ TakeOver(a)

↓ Delete(y) ↘ TakeOver(y)

Berechnung der Levenshtein Distanz

Übung:

Berechnen Sie die Levenshtein Distance Matrix für die beiden Wörter. Wie gross ist die Levenshtein-Distanz?

B[j] zu A[i] von		W	O	R	L	D
W						
O						
R						
D						

Java Implementation

- Ein 2-dim Array "distance" um die minimale Distanz zwischenzuspeichern

```
public class LevenshteinDistance {
    private static int minimum(int a, int b, int c) {
      return Math.min(Math.min(a, b), c);
    }

    public static int computeLevenshteinDistance(String str1,String str2) {
      int[][] distance = new int[str1.length() + 1][str2.length() + 1];

      for (int i = 0; i <= str1.length(); i++) distance[i][0] = i;
      for (int j = 1; j <= str2.length(); j++) distance[0][j] = j;

      for (int i = 1; i <= str1.length(); i++)
         for (int j = 1; j <= str2.length(); j++) {
            int minEd = (str1.charAt(i - 1) == str2.charAt(j - 1)) ? 0 : 1;
            distance[i][j] = minimum(
                   distance[i - 1][j] + 1,
                   distance[i][j - 1] + 1,
                   distance[i - 1][j - 1]+ minEd);
         }
      return distance[str1.length()][str2.length()];
    }
}
```

Min von 3 Werten

all insert

all delete

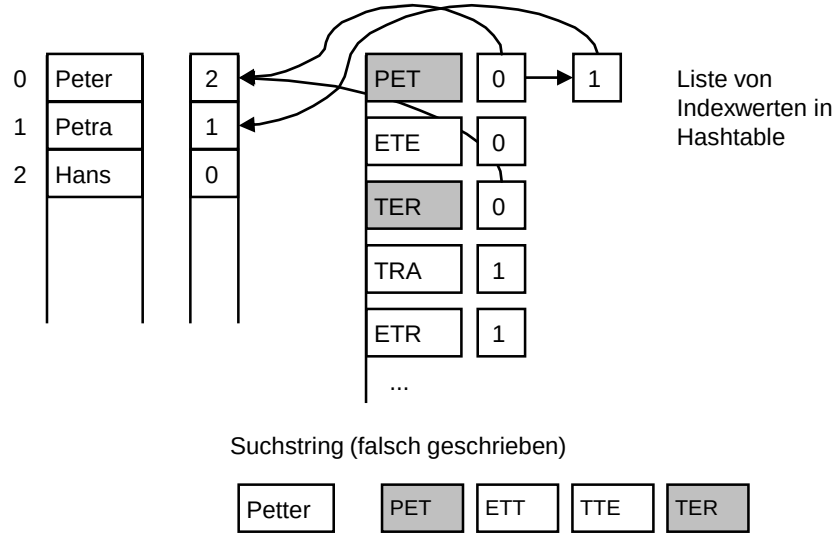
equal

substitute

Trigram Suche

Trigram Suche

- Fehlertolerante Suche (auch für Wortverdreher, z.B Vor- Nachname)
- Effizient für grosse Datenbestände
- Index -> Wort in 3-Buchstaben Gruppen unterteilt ,
  - z.B. sind das bei „Peter“, die 3-er Gruppen „PET“, “ETE“, “TER“.
  - Diese 3-er Gruppen werden für alle vorkommenden Worte gebildet und in Hashtabelle gespeichert
- Das zu suchende Wort wird ebenfalls in 3-er Gruppen zerlegt
  - das gesuchte Wort mit am meisten Übereinstimmungen wird genommen



Phonetische Suche

Phonetische Suche

- z.B. Soundex Phrasen nach ihrem Klang in englischer/deutscher Sprache.
- Wort besteht aus seinem ersten Buchstaben, gefolgt von drei Ziffern, z.B. "K523" (Soundex-Code)
  - Kurze Worte: mit 0 auffüllen; 0-werden ignoriert
  - Die Vokale A, E, I, O und U und die Konsonanten H, W und Y sind ausser beim ersten Zeichen zu ignorieren (in D auch ä,ö,ü).
  - Ziffern sind Konsonanten nach folgender Tabelle

Englisch

Ziffer	Repräsentierte Buchstaben
1	B, F, P, V
2	C, G, J, K, Q, S, X, Z
3	D, T
4	L
5	M, N
6	R

Deutsch

Ziffer	Repräsentierte Buchstaben
0	a, e, i, o, u, ä, ö, ü, y, j, H
1	b, p, f, v, w
2	c, g, k, q, x, s, z, ß
3	d, t
4	l
5	m, n
6	r
7	ch

- Beispiele:

Britney → BRTN → **B635**	bewährten → BRTN → **B635**
Spears → SPRS → **S162**	Superzicke → SPRZCK → S16222 → **S162**

Suchen nach Mustern

Reguläre Ausdrücke: Regex

- Zum Suchen von **definierten Mustern** in Texten d.h. Strings
  - ein bestimmter String: "ZHAW"
  - Muster kann "unscharf" definiert sein: z.B. IT13a, IT12a, IT12c
  - (Teil-)Muster kann sich wiederholen: 170.12.34.12
- Die meisten heutigen Programmiersprachen unterstützen die Suche nach Muster in Form von reguläre Ausdrücke (Regular Expressions oder kurz Regex)
- Regex ist unabhängig von Java definiert
- Java Klassenbibliothek definiert im Package java.util.regex
- die Klassen **Pattern** und **Matcher**

Definition des regulären Ausdrucks

- Zuerst muss der reguläre Ausdruck vorbereitet werden
- Die Klasse **Pattern**

```
Pattern pat = Pattern.compile("ZHAW");
```

- Das Muster kann im einfachsten Fall ein Textzeichen-String sein
- Alle Zeichen sind erlaubt ausser: `([{\^-$|]})?*+.`
  - Diese Zeichen müssen mit \ vorangestellt geschrieben werden
  - Vorsicht in Java String Konstante muss "\\" für "\" geschrieben werden
  - Beispiel: "wie geht's \\?"

Abfrage nach den gefundenen Stellen

- Ausgabe der gefundenen Stellen
- Die Klasse **`Matcher`**

```
Matcher matcher = pat.matcher("Willkommen an der ZHAW");
```

- Suche nächste Textstelle **`boolean find()`**
  - true falls gefunden

```
matcher.find();
```

- Gebe gefunden Teilstring zurück **`String group()`**

```
matcher.group(); // ZHAW
```

- gefundene Start und Endposition innerhalb String **`int start()`** und **`int end()`**

```
matcher.start(); //18
matcher.end();  //22
```

Vollständiges Beispiel

School of Engineering

```
import java.util.regex.*;


…
Pattern pat = Pattern.compile("ZHAW");
Matcher matcher = pat.matcher("Willkommen an der ZHAW");
while (matcher.find()) {
   String group = matcher.group();
   int start = matcher.start();
   int end = matcher.end();
   // do something
}
```

Platzhalter

- Oftmals wird nach unscharfen Muster gesucht, z.B. alle IT Klassen
- Es sind Platzhalter Zeichen erlaubt, die *Zeichenmengen* matchen
  - z.B. "." für beliebiges Zeichen "\d" für Zahl, ^\d für *keine* Zahl

Platzhalter	Beispiel	Bedeutung	Menge der gültigen Literale
.	a.b	Ein beliebiges Zeichen	aab, acb, aZb, a[b, ...
\d	\d\d	Digit[0-9]	78, 10
\D	\D	kein Digit	a, b , c
^	^\d	Negation	a, b , c
\s	\s	Leerzeichen (Blank,etc)	blank, tab, cr,
\S	\S	kein Leerzeichen	

- Aufgabe: Geben Sie das Suchmuster für beliebige IT Klassen an:
  IT10a, IT08b, IT09c

Eigene Zeichenmengen

Statt vordefinierte Zeichenmengen zu verwenden, können auch eigene definiert werden, diese werden in "[" "]" geklammert.

1. Aufzählung der Zeichen in der Zeichenmenge
   - ein Zeichen aus der Menge
   - z.B. "a", "b" oder "c" : [abc]
2. Bereiche
   - z.B. alle Kleinbuchstaben [a-z] alle Buchstaben [a-zA-Z]
3. Negation: Alle Zeichen ausser
   - z.B. [^a]
4. Aufgabe: Geben Sie das Suchmuster für beliebige IT Klassen an; es gäbe aber nur "a" bis "d"

Optional, Alternative und Wiederholung

- Optionale Teile: ?
  - wenn einzelner Buchstaben optional, z.B. ZHA?W -> ZHW oder ZHAW
- Alternative (Oder) |
  - wenn ein A oder B -> ZH(A|B)W -> ZHAW oder ZHBW
  - "natürliche" Verwendung von Klammern

Wiederholungen

Auch 0 mal erlaubt

1. Beliebig oft: *
   - eine Folge von Ziffern \d* -> _, 2,23,323,423,..
2. Mindestens einmal +
   - eine Folge von Ziffern aber mindest eine \d+ -> 3,34,234,…
3. Bestimmte Anzahl mal {n}
   - eine Folge von drei Ziffern \d{3} -> 341,241,123 ..
4. Mindestens, maximal Anzahl mal {n,m}
   - eine Folge von 1 bis 3 Ziffern \d{1,3} -> 1, 23, 124, ...

Zusammenfassung Metasymbole

- (Meta-)Sprache zur Beschreibung der Bildungsregeln von Sätzen
- Metasymbole: `( [ { \ ^ - $ | ] } ) ? * + .`

Metasymbol	Beispiel	Bedeutung	Menge der gültigen Literale
*	ax*b	0 oder mehrere x	ab, axb, axxb, axxxb, ...
+	ax+b	1 oder mehrere x	axb, axxb, axxxb, ...
?	ax?b	x optional	ab, axb
|	a|b	a oder b	a, b
()	x(a|b)x	Gruppierung	xax, xbx
.	a.b	Ein beliebiges Zeichen	aab, acb, aZb, a[b, ...
[]	[abc]x	1 Zeichen aus einer Menge	ax, bx, cx
[-]	[a-h]	Zeichenbereich	a,b,c, ..., h
\d	\d\d	Digit[0-9]	78, 10
\D	\D	kein Digit	a, b , c
^	^\d	Negation	a, b , c
\s	\s	Leerzeichen (Blank,etc)	blank, tab, cr,
\S	\S	kein Leerzeichen	

Übungsbeispiele

Regulärer Ausdruck	Gültigen Sätze
a?b+	
	ein, eine, einer
a(x|y)?b*	
	Binär-Zahlen
	Ganze Zahl

Weitere Methoden für Regex

Prüfe ob ganzer String einem Regex Muster entspricht:

```
boolean matches(String regexp);
```

```
String text = "Hallo Welt";
boolean passt;
passt = text.matches("H.*W.*");
passt = text.matches("H..o Wel?t");
passt = text.matches("H[alo]* W[elt]+");
passt = text.matches("Hal+o Welt.+");
```

- Aufgabe: Regex zum Prüfen ob ein String eine Int Zahl enthält

- Aufgabe: Regex zum Prüfen ob ein String eine IP Adresse enthält

… Weitere Methoden für Regex

```
String replaceAll(String regexp, String replaceStr);
```

```
String replaceFirst(String regexp, String replaceStr);
```

- Ersetzt im gegebenen String alle (bzw. den Ersten bei **`replaceFirst`**) Substrings, die **`regexp`** entsprechen, mit **`replaceStr`**

```
String new = text.replaceAll("l+", "LL");   // HaLLo WeLLt
```

```
String[] split(String regexp);
```

- Teilt den gegebenen String in mehrere Strings, **`regexp`** ist die Grenzmarke
- Das Resultat ist ein Array mit Teilstrings

```
String data = "4, 5, 6 2,8,, 100, 18"
String[] teile = data.split("[ ,]+");    // Menge der Zeichen " " und","
// 4 5 6 2 8 100 18
// teile[0] = "4", teile[1] = "5", …
```

Zusammenfassung

- Suche von Strings in Strings
- Suchmaschinen und Index
- Unscharfe Suche
- Suchen nach Mustern

Reguläre Ausdrücke – Lösung

Regulärer Ausdruck	Menge der gültigen Literale
`a?b+`	b, bb, bbb..., ab, abb, abbb...
eine?r?	`{ein, eine, einer}`
`a(x|y)?b*`	a, ax, ay, ab, axb, ayb, abb, axbb...
[_$a-zA-Z][_$a-zA-Z0-9]*	`Java-Bezeichner`
(0|-?[1-9][0-9]*)	`alle ganze Zahlen`