

Regular Expressions

kurz: RegEx
Pattern Matching

Wozu RegEx ?

- ▶ Entspricht/enthält Text ein bestimmtes **Muster**?
- ▶ Eingaben auf **Gültigkeit** prüfen
 - Telefonnummer: +43 (01) 123985
 - Email-Adresse: hansi.huber@quaxi.com
 - Kreditkartennummer
 - ...
- ▶ **Suchen & Ersetzen** in Texten
 - alle URLs
 - alle Maier, Mayr, Meir, ...
- ▶ Text **splitten**
 - Aufteilen nach einzelner Trennzeichen
 - Splitten mit Muster wie *xxx*

Wo gibt es RegEx ?

- ▶ Ist für alle gängigen **Programmiersprachen** implementiert

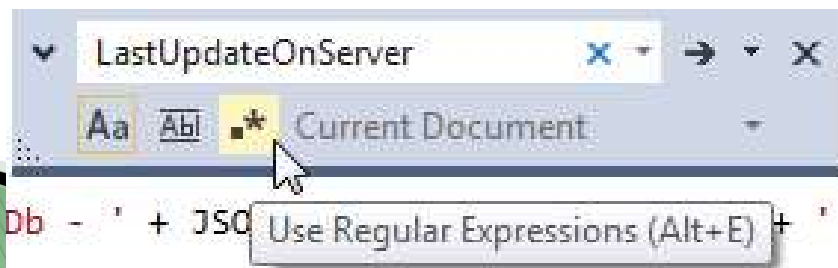
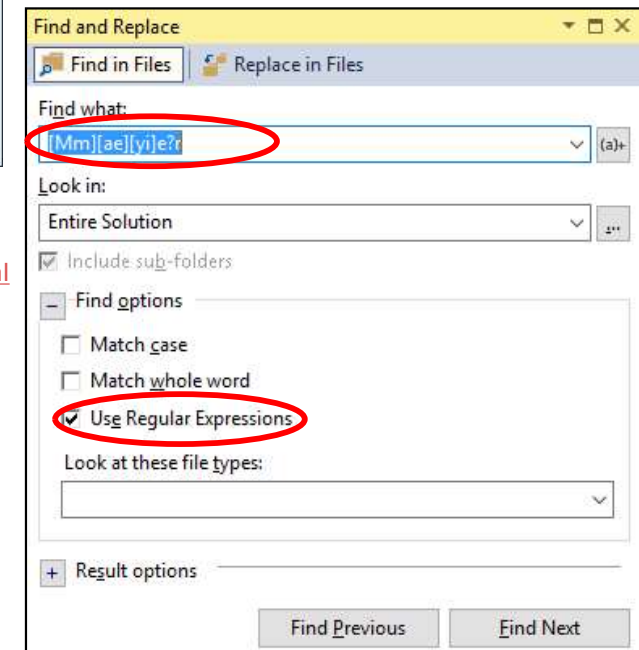
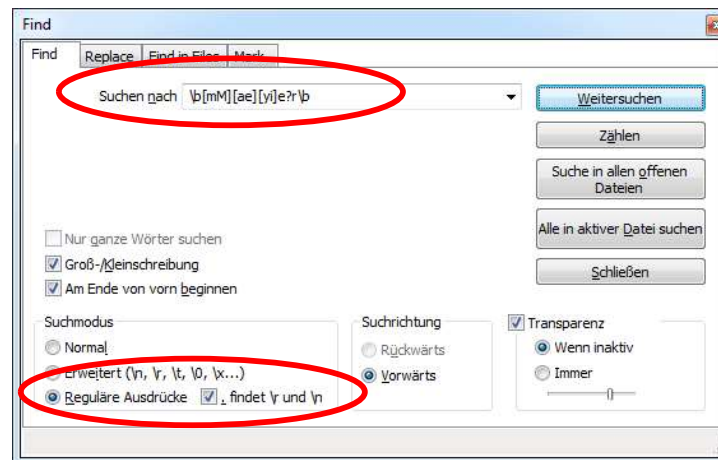
- C#
- Java
- Javascript
- ...

- ▶ **Texteditoren**

- Notepad++
- Word (mit Abstrichen)

<http://vlasovstudio.com/regent/documentation/Microsoft-Word-Wildcards-as-Regular-Expressions.html>

- ▶ **Visual Studio**



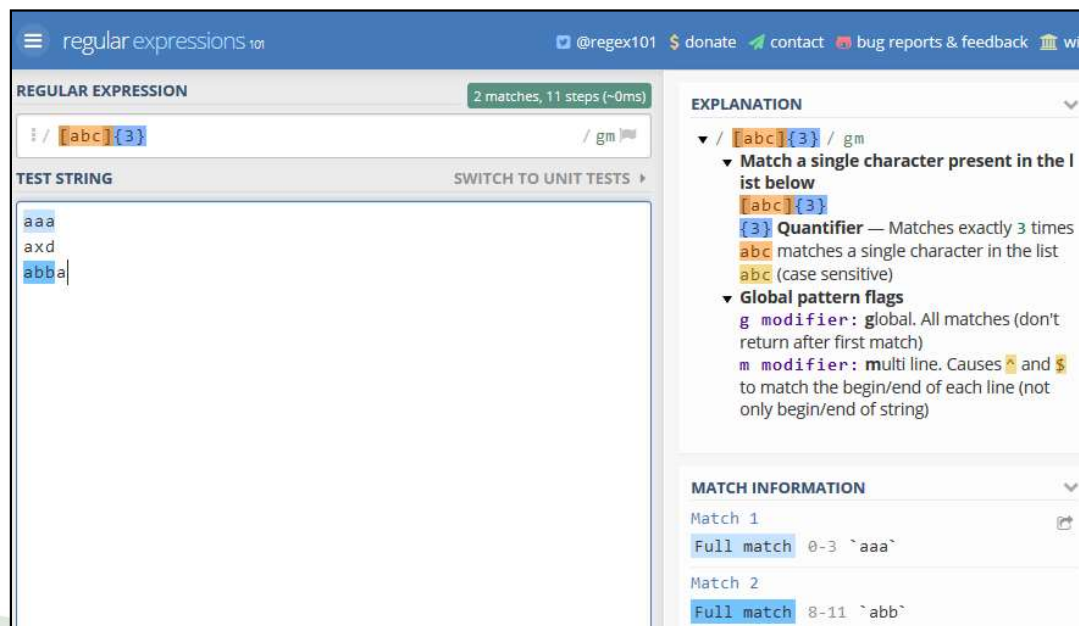
Testen von RegEx

► Offline Tool Espresso

- <http://www.ultrapico.com/Espresso.htm>
- muss installiert werden (➔ in der Schule nicht)

► Online Tools

- es gibt ziemlich viele
- z.B. <https://www.regex101.com/>



Zeichen

- ▶ Jedes Zeichen steht für sich selbst
- ▶ Außer Markup-chars: **\$ () + . * ? [\ ^ { |**
 - diese müssen mit **** maskiert werden
 - z.B.: **** oder **\?**
- ▶ Sonderzeichen
 - **\n**: newline
 - **\r**: carriage return
 - **\t**: Tabulator
- ▶ ASCII: auch als **\x12** möglich (HEX-Code)
 - Bsp: **\x41** steht für ‚A‘
- ▶ Unicode mit **\u1234**
 - ▶ Bsp: **\u0041** steht für ‚A‘

Zeichenklassen

- ▶ Oft braucht man nur bestimmte Buchstaben
 - → Zeichenklassen
 - werden mit **[]** markiert
 - mit **-** kann man Bereiche festlegen
 - **^** heißt „keines der Zeichen“
- ▶ Beispiele:
 - **[ax24y]**: eines dieser Zeichen
 - **[a-cx-z]**: a, b, c, x, y oder z
 - **[a-zA-Z0-9]**: beliebiger Buchstabe oder Ziffer
 - **[^aeiouAEIOU]**: kein Selbstlaut
- ▶ Sonderzeichen innerhalb Zeichenklassen:
 - **^** (Negation), **** (Escape), **-** (Bereich), **]** (Ende)
 - Beispiel: **[?\- (\\]** heißt: **?**, **-**, **(** oder ****

Zeichenklassen – Kürzel

- ▶ Bestimmte Zeichenklassen braucht man häufig
- ▶ ➔ es gibt Kürzel für einige Zeichenklassen

Kürzel	Beschreibung	entspricht
\d	Ziffer	[0-9]
\w	Wortzeichen	[a-zA-Z0-9_]
\s	Whitespace	[\t\r\n]
.	jedes Zeichen	

- ▶ \b ist eigentlich keine Zeichenklasse, sondern findet Position
- ▶ **Großbuchstaben** entsprechen der **Umkehrung**
 - ▶ Bsp: **\D** heißt „keine Ziffer“, also [^0-9]

Positionen

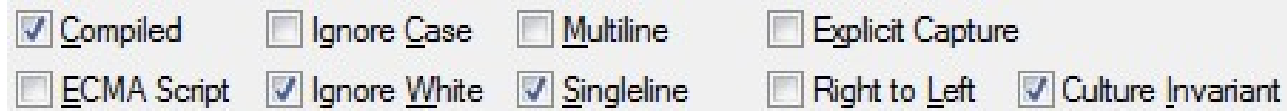
- ▶ Es gibt sogenannte „Anker“ für bestimmte Positionen
- ▶ Passen zu keinem Zeichen, sondern nur auf Position
- ▶ [^] = Textanfang
- ▶ ^{\$} = Textende
- ▶ ^{\b} = Wortgrenze

Varianten

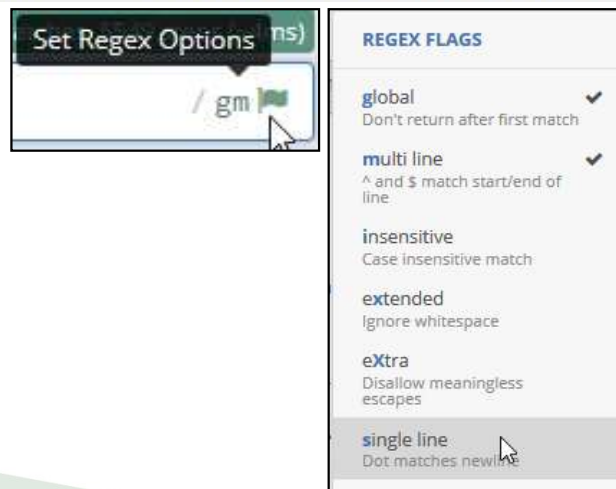
- ▶ Varianten können mit | angegeben werden
 - z.B. **Hansi | Susi**

Optionen

- ▶ Passt `.` zu Zeilenumbruch?
 - Option **Singleline**: true → ja (default ist false)
 - **Singleline beeinflusst also `.`**
- ▶ Ist Zeilenfang wie Textanfang bzw. Zeilenende wie Textende?
 - Option **Multiline**: true → ja (default ist false)
 - **Multiline beeinflusst also `^` und `$`**
- ▶ Diese Optionen sind nicht Teil des Pattern sondern müssen extra eingestellt werden.
- ▶ Bei Espresso im Tab „Design Mode“/“Characters“



- ▶ Bei regex101.com:



Häufigkeiten

- ▶ Ähnlich wie bei DTD kann man angeben, wie oft ein Zeichen vorkommen muss/darf
 - **{n}**: genau n Mal
 - **{n,m}**: mindestens n u. maximal m Mal
 - **{n, }**: mindestens n Mal
 - **+**: mindestens ein Mal, also **{1, }**
 - *****: beliebig oft (auch 0 Mal), also **{0, }**
 - **?**: 0 oder ein Mal, also **{0,1}**
- ▶ Beispiel Muster für Datum wie 9-10-2013
 - **\d{1,2}-\d{1,2}-\d{4}**
 - ginge auch so: **\d\d?-\d\d?-\d\d\d\d**

Übungen

- ▶ Mayr: alle Meier, Mair, Mayer,...
 - Mayrhuber bzw. Schildmair usw. ausschließen
- ▶ Smileys, wie z.B. ;-), :-), ;-o
- ▶ Binärzahlen mit 2 bis 16 Stellen
- ▶ Hauptwort mit genau 4 Buchstaben
- ▶ Wort großgeschrieben, das auf „en“ endet
- ▶ HTML-Anchor-Tags: ``
 - Noch offen: Wie bekomme ich Zugriff auf Link-Text??

greedy/lazy

- ▶ Regex arbeitet prinzipiell gierig (**greedy**)
- ▶ „Frisst“ also so viel wie möglich
- ▶ Beispiel: 1234123412341234
- ▶ Regex: **1\d*1**
- ▶ findet 1234123412341234
- ▶ **lazy** wäre: „Frisst“ so wenig wie möglich
- ▶ Zwei Möglichkeiten, Regex **lazy** zu machen:
 - Zeichen, das Ende markiert, von Suche ausnehmen
 - Bsp: **1[^1]*1** findet 1234123412341234
 - Zählzeichen lazy machen durch Anhängen von **?**
 - also: ***?**, **+?**, **??**, **{1,8}?**
 - Bsp: **1\d*?1** findet 1234123412341234

Gruppen


- ▶ Durch Klammern kann man Ausdrücke gruppieren
- ▶ Haben höchste Priorität bei Auswertung
- ▶ Bsp:
 - `\bSusi|Maria|Julia\b` heißt `\bSusi` od. `Maria` od. `Julia\b`, findet also Marianne
 - `\b(Susi|Maria|Julia)\b` heißt `\bSusi\b` od. `\bMaria\b` od. `\bJulia\b`
- ▶ Regex speichert beim Finden die Gruppen
- ▶ Index beginnt bei 1 !!!!!!!
- ▶ Beispiel: `\b(\d{4})-(\d{2})-(\d{2})\b`
 - Ergebnis bei 2013-09-11:

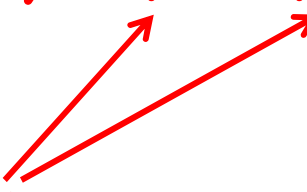
```
2013-09-11
├── 1: 2013
├── 2: 09
└── 3: 11
```

MATCH INFORMATION		
Match 1		
Full match	67-77	<code>^2018-10-08^</code>
Group 1.	67-71	<code>^2018^</code>
Group 2.	72-74	<code>^10^</code>
Group 3.	75-77	<code>^08^</code>

Gruppen – Referenzen

- ▶ Da Gruppen-Ergebnisse gespeichert werden
- ▶ ➔ kann darauf zugreifen
- ▶ Zugriff durch **\1**, **\2**,..., also mit **\n** wobei n die Gruppennummer ist
- ▶ Bsp.: Datum, wo Tag==Monat==Jahr ist, also z.B. 2012-12-12
- ▶ Regex: **\b\d{2}(\d{2})-\1-\1\b**

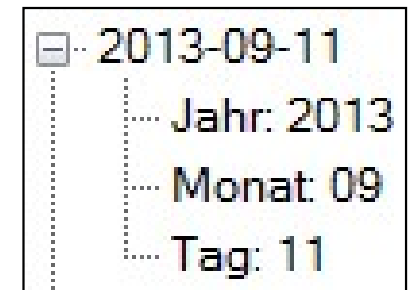

Gruppe **1**



- ▶ Nennt man auch **catching group** (einfangende Gruppe)

Gruppen – benannte Gruppen

- ▶ Man kann Gruppen auch benennen
- ▶ Schreibt **(?<myName>...)** oder **(? 'myName' ...)**
- ▶ Beispiel: Datum der Form 2013-09-10
- ▶ **\b(?<Jahr>\d{4})-(?<Monat>\d{2})-(?<Tag>\d{2})\b**
- ▶ Verwenden der Referenz:
 - **\k 'myName'** oder **\k<myName>**
- ▶ Beispiel von oben (2012-12-12)
- ▶ Regex: **\b\d{2}(?<Nr>\d{2})-\k'Nr'-\k'Nr'\b**
- ▶ Vorteil: fügt man neue Gruppen ein, ändert sich der Index der nachfolgenden Gruppen nicht



Gruppen – Performance

- ▶ benannte Gruppen kosten Performance
- ▶ Braucht man keine Referenz darauf, kann/soll man Speicherung verhindern
- ▶ Nennt man **non-catching group**
- ▶ Notation: `\b(?:Susi|Maria|Julia)\b`

Umfangreicheres Beispiel

- ▶ Regex für **Email-Adressen**
 - Text vor @ muss mit Buchstaben beginnen
 - vor @ beliebig viele Namen mit Punkt getrennt
 - hinter @ mindestens 1 Punkt
- ▶ Gültige Adressen:
 - susi.loewinger@dummy.at
 - hansi.huber@quaxi.com
 - hansi@quaxi.com
 - pauli.gruber-lehner@nixico.at
 - pauli.gruber-lehner@nixi.co.at
- ▶ Ungültige Adressen:
 - pauligruber-lehner@nixicoat
 - pauli.gruber-lehner@nixicoat
 - 123people@dummy.com
 - a..@nixi.at

Kurze Beispiele

- ▶ Dreifach-Buchstaben
- ▶ Doppel- oder Dreifachbuchstaben
- ▶ `<a>`-Tag mit Zugriff auf URL
- ▶ Zahlen der Form 7887

Umfangreicheres Beispiel 2 (HÜ)

- ▶ Regex für **HTML-Tags mit Attributen**
- ▶ **z.B.:** `<div class="abc xyz" id="divX" style="background-color:red; font-size:8pt;">`
- ▶ **Gruppen für**
 - Tagname
 - Attribut-Name
 - Attribut-Wert

Lookaround

- ▶ Normale matches „fressen“ die angegebenen Zeichen
- ▶ Manchmal möchte sich nur umschauchen, ohne die Position zu ändern → **Lookaround**
- ▶ Das geht nach vor und zurück:
 - **Lookahead**
 - **Lookbehind**
- ▶ Positiver Lookahead: **(?=...)**
- ▶ Negativer Lookahead: **(?!...)**
- ▶ Positiver Lookbehind: **(?<=...)**
- ▶ Negativer Lookbehind : **(?<!...)**

Lookahead

- ▶ Beispiel Positive Lookahead: 123 Euro
- ▶ `\d+(?= Euro)`
- ▶ Achtung:
 - der Cursor steht direkt hinter dem **3**er
 - der Text Euro wird nicht konsumiert
- ▶ Beispiel Negative Lookahead: 123 Dollar
- ▶ `\d+(?! Dollar)`
- ▶ Alles was nicht auf „Dollar“ endet, ist ein Match

Lookbehind

- ▶ Beispiel Positive Lookbehind: EUR123
- ▶ `(?<=EUR) \d{3}`
- ▶ Stellt sicher, dass an der aktuellen Position im String „EUR“ davor steht
- ▶ Dreistellige Zahl ist nur ein Match, wenn davor „EUR“ steht

- ▶ Beispiel Negative Lookbehind: USD123
- ▶ `(?<!EUR) \d{3}`
- ▶ Dreistellige Zahl ist nur dann ein Match, wenn davor nicht „EUR“ steht

Beispiel CamelCase

- ▶ Mit Lookarounds kann man Positionen finden
- ▶ Nennt man auch **Zero-Width Matches**
- ▶ Beispiel **Splitten** von „MyAwesomeMethod“
- ▶ Suche also Position mit
 - links Kleinbuchstabe → **(?<=[a-z])**
 - rechts Großbuchstabe → **(?=[A-Z])**
- ▶ → **(?<=[a-z])(?=[A-Z])**
- ▶ Achtung: Aufpassen mit Flag Ignore Case

```
var words = Regex.Split("MyAwesomeMethod", @"(?<=[a-z])(?=[A-Z])");  
foreach (var word in words)  
{  
    Console.WriteLine(word);  
}
```

My
Awesome
Method

Mehrere Bedingungen

- ▶ Lookaround sind vor allem dann nützlich, wenn mehrere Bedingungen zu prüfen sind
- ▶ **Lookarounds bewegen den Cursor nicht!!**
- ▶ Beispiel Passwort-Check:
 - zw. 6 und 10 Wortzeichen **`(?=\w{6,10}$)`**
 - Mind. 1 Kleinbuchstabe **`(?=[^a-z]*[a-z])`**
 - Mind. 1 Ziffer **`(?=\D*\d)`**
 - Mind. 3 Großbuchstaben **`(?=(?:[A-Z]*[A-Z]){3})`**
- ▶ **`^(?=\w{6,10}$)(?=[^a-z]*[a-z])(?=\D*\d)(?=(?:[A-Z]*[A-Z]){3}).*`**
- ▶ Allgemein:
 - n Bedingungen → n-1 Lookarounds