



Teil 2: Ruby-Vertiefung

... zu einer Auswahl wichtiger Aspekte
der Sprache



Reguläre Ausdrücke



Reguläre Ausdrücke



- sind ein typisches Leistungsmerkmal von Scriptsprachen
- bestehen aus Mustern (engl. "*patterns*") zum Suchen und Ersetzen in Zeichenketten ("*strings*")
- zerlegen Strings in verschiedene Abschnitte
- wirken im Normalfall auf Textzeilen, können aber auch auf mehrzeilige Strings angewendet werden.
- existieren schon seit Jahrzehnten in Unix und konzeptionell bereits seit den 1940er Jahren
- wurden durch Perl seit Anfang der 1990er Jahre populär
- beeinflussen eine Reihe von globalen Variablen
- wurden deshalb von Ruby objekt-orientiert gekapselt
- besitzen Optionen (!)
- werden von Ruby sowohl in traditioneller als auch in objekt-orientierter Weise angeboten
- ... sind einfach unverzichtbar!



Reguläre Ausdrücke



- Der *match*-Operator `=~`
 - Ein regulärer Ausdruck wird mittels des "*match*"-Operators `=~` zu einem String in Beziehung gesetzt (Reihenfolge beachten!):

```
str =~ re
```
 - Falls String und regulärer Ausdruck zueinander passen ("match"), liefert `str =~ re` den Index des ersten passenden Zeichens innerhalb `str` zurück, beginnend mit 0:

```
"0123456789" =~ /2/ # 2
```
 - Falls der Vergleich scheitert, ergibt der Ausdruck `nil`:

```
"0123456789" =~ /A/ # nil
```
 - Es ist zwar möglich, mit `=~` zwei Strings zu vergleichen; der zweite String wird dann implizit in einen regulären Ausdruck gewandelt:

```
"0123456789" =~ '2' # 2
```

Dieses Vorgehen ist aber nicht erwünscht und bewirkt eine Warnung!
 - `=~` bewirkt Seiteneffekte auf eine Reihe globaler Variablen. S.u.!



Reguläre Ausdrücke



- Reguläre Ausdrücke in Ruby
 - Ruby verwaltet reguläre Ausdrücke als Objekte einer eigenen Klasse "Regexp"

```
re = /2/ # Traditionelle Schreibweise
"0123456789" =~ re # 2
re.class # Regexp
```

```
re = Regexp.new('2') # alternativ, objekt-orientiert
"0123456789" =~ re # 2
re.class # Regexp
```

```
re = %r(2); ... # Erinnerung: Generalis. String
```

- Optionen regulärer Ausdrücke (Verwendung s.u.)
 - i case insensitive
 - o substitute once (bezogen auf Ersetzungsmuster #...)
 - m multiline mode ('.' passt dann auch auf \n, s.u.)
 - x extended mode (Aktivierung erweiterter Möglichkeiten)
 - Multibyte-Optionen n, e, s, u (z.B. für UTF-8), hier ignoriert.



Muster in regulären Ausdrücken



- Normale Zeichen
 - Alle "normalen" Zeichen (Sonderzeichen s.u.) suchen sich selbst:

```
"Dies ist ein Text" =~ /ein/ # 9 (Offset of match)
"Dies ist ein Text" =~ /kein/ # nil (no match)
```

- Wirkung des Match-Operators =~ hier wie Substring-Suche

- Sonderzeichen:

```
. | ( ) [ \ ^ { + $ * ?
```

- Sonderzeichen \ (backslash)

- Ähnlich wie bei Stringeingaben dient \ hier als *Escape*-Zeichen: Sonderzeichen werden durch *Escaping* (mit Präfix "\") zu normal suchbaren Zeichen:

```
"Rechne: 3 ** 4 = 81" =~ /\*\*/ # 10 (offset of '**')
```

```
winpath = 'C:\TEMP\sample.txt'
```

```
winpath =~ /\Temp/i # 2; mit Option i!
```



Muster in regulären Ausdrücken



- Sonderzeichen . (Punkt)
 - Der Punkt passt zu jedem beliebigen einzelnen Zeichen (*wildcard*).

```
"Beleg" =~ /e1.g/ # 1
"Belag" =~ /e1.g/ # 1
"bel*g" =~ /e1.g/ # 1
```

- Verliert Sonderbedeutung innerhalb eckiger Klammern [], s.u.
- Sonderfall Zeilenende-Zeichen (\n):

```
a = "Zeile1\nZeile2\nZeile3\n" # Multiline-Beispiel
```

```
a =~ /eile./ # 1
a =~ /eile../ # nil (. passt nicht zu \n!)
a =~ /eile../m # 1 (. passt nun zu \n)
```

```
re = Regexp.new('eile..', Regexp::MULTILINE)
a =~ re # 1 (analog)
```



Muster in regulären Ausdrücken



- Sonderzeichen ^ (Zeilenanfang); \A (String-Anfang)

```
"ein Fall" =~ /ei/ # 0
"kein Fall" =~ /ei/ # 1
"ein Fall" =~ /^ei/ # 0
"kein Fall" =~ /^ei/ # nil
"ein Fall" =~ /\Aei/ # 0
"kein Fall" =~ /\Aei/ # nil
```

- ^ und \A sind nur im Fall einzeliger Strings gleichwertig:

```
a = "eins\nzwei\ndrei\n"
a =~ /^zwei/ # 5
a =~ /\Azwei/ # nil
```



Muster in regulären Ausdrücken



- Sonderzeichen \$ (Zeilenende); \Z, \z (String-Ende)

"Fall" =~ /ll/	# 2
"Falle" =~ /ll/	# 2
"Fall" =~ /ll\$/	# 2
"Falle" =~ /ll\$/	# nil
"Fall" =~ /ll\Z/	# 2
"Falle" =~ /ll\z/	# nil

- \$ und \Z sind nur im Fall einzeliger Strings gleichwertig:

a = "eins\nzwei\ndrei\n"	
a =~ /zwei\$/	# 5
a =~ /zwei\z/	# nil

- \Z ignoriert \n am Stringende, \z nicht:

a =~ /drei\$/	# 10
a =~ /drei\Z/	# 10
a =~ /drei\z/	# nil
a =~ /drei.\z/	# nil
a =~ /drei.\z/m	# 10



Muster in regulären Ausdrücken



- Wiederholungsangaben mit Sonderzeichen {

- In geschweiften Klammern gibt man die minimale und maximale Wiederholungszahl für das davorstehende Muster an:

re = /abc{2,4}/	# 'ab' gefolgt von 2..4 mal 'c'
"abc" =~ re	# nil
"abcc" =~ re	# 2
"abccc" =~ re	# 2
"abcccc" =~ re	# 2
"abccccc" =~ re	# 2 (letztes 'c' stört nicht!)

- Kurzformen: {n} = {n,n}, {n,} = {n, ∞}

- Traditionelle Kurzformen (Sonderzeichen ?, +, *)

- ? = {0,1}, + = {1,}, * = {0, ∞}

"ab" =~ /bc*/	# 1
"abc" =~ /bc?d/	# nil
"abd" =~ /bc?d/	# 1
"abcd" =~ /bc+d/	# 1
"abccd" =~ /bc+d/	# 1
"abccd" =~ /bc?d/	# nil



Muster in regulären Ausdrücken



- **Wiederholungsangaben: Greedy vs. non-greedy**
 - Manchmal ist es entscheidend, zwischen der maximal und der minimal ausführbaren Wiederholungszahl zu unterscheiden:

Suche nach Zeichen zwischen whitespace

```
line = "Dies ist ein kurzer Satz."
```

"gierige" Variante (Normalfall)

```
line =~ /\s.*\s/ # Trefferbereich: "ist ein kurzer"
```

"nicht gierige" Variante

```
line =~ /\s.*?\s/ # Trefferbereich: "ist"
```

"nicht gierige" Variante, verallgemeinert

```
line =~ /\s.{5,}?\s/ # Trefferbereich: "ist ein"
```

- Unterscheide:

```
*    und *?
+    und +?
{n,} und {n,}?
```

- **Test:** Was ergibt (mit "line" wie oben definiert)

```
line =~ /A*/
```

Antwort: 0 (Jeder String enthält mind. 0-mal 'A!')



Muster in regulären Ausdrücken



- **Auswahl mit Sonderzeichen |**
 - "Oder"-Beziehungen sind einfach möglich mittels |:

```
"abc" =~ /c|d/ # 2 # c oder d
```

```
"abd" =~ /c|d/ # 2
```

```
"abe" =~ /c|d/ # nil
```

- | bindet nur schwach:

```
"abc" =~ /bc|d/ # 1 ("b", dann "c" oder "d")
```

```
"abd" =~ /bc|d/ # 2 ("bc" oder "d" !!)
```

```
"abe" =~ /bc|d/ # nil
```

```
"abd" =~ /bc|bd/ # 1 ("bc" oder "bd")
```

- **Gruppenbildung mit Sonderzeichen (**

```
"abc" =~ /b(c|d)/ # 1 ("b", dann "c" oder "d")
```

```
"abd" =~ /b(c|d)/ # 1
```

```
"abcabdabe" =~ /(ab.+)/ # 0
```

```
"abcde" =~ /b(c.)e/ # 1
```

- Seiteneffekt: \1 ... \9 (bzw. \$1 ... \$9) gesetzt (*backreferences*)



Muster in regulären Ausdrücken



- **Zeichenmengen mit Sonderzeichen [**
 - In eckigen Klammern listet man alle Zeichen, die an einer bestimmten Stelle zulässig sind:


```
"abdf" =~ /[cde]/ # 2 ('d' passt)
```
 - Auch Bereiche (bez. ASCII-Codierung) sind zulässig


```
/[01]/ # eine Binärziffer
/[0-7]/ # eine Oktalziffer
/[0-9a-fA-F]/ # mind. eine Hex-Ziffer
```
 - Suche nach '-' bzw. ']' möglich, wenn Zeichen am Anfang der Liste:


```
/[-+]/ # Vorzeichensuche
/[ ]>]/ # Suche nach schließenden Klammern
```
 - Sonderzeichen: In Zeichenmengen wie normale Zeichen (außer '\')


```
/[. | ( ) [ { + ^ $ * ? ] / # in [...] normal suchbar
```
 - Komplementbildung durch Voranstellen von '^'


```
/[^0-9]/ # Alles außer einer Ziffer
/[^ ^]/ # Alles außer '^'
```



Muster in regulären Ausdrücken



- **Spezielle Zeichen und Zeichenmengen**
 - \A, \Z, \z schon besprochen (Stringanfang / -ende)
 - \d, \D [0-9], [^0-9]
 - \n, \t, ... *newline, tab, ...* (vgl. Strings)
 - \000, ... \277 Suche über Oktalwert des Zeichens
 - \x00, ... \xff Suche über Hexadezimalwert des Zeichens
 - \s, \S [\t\n\r\f], [^...] (invers)
 - \b Backspace, wenn innerhalb [...], Wortgrenze sonst;
 - \B Treffer, wenn keine Wortgrenze

```
"einszwei zweidrei" =~ /zwei/ # 4
"einszwei zweidrei" =~ /\bzwei/ # 9
"einszwei zweidrei" =~ /\Bzwei/ # 4
```

 - \w, \W [0-9A-Z_a-z] (word char), [^...] (invers)
 - \G Stelle, an der der letzte Treffer endete (dokumentiert, aber ohne Beispiel. Noch zu klären!)



Reguläre Ausdrücke und globale Variablen



- Match-Operator bewirkt String-Zerlegung und Setzen bestimmter globaler Variablen:
 - \$` Teilstring vor dem Treffer
 - \$& Trefferbereich
 - \$' Teilstring nach dem Trefferbereich
 - \$1...\$9 Falls Gruppen (...) angegeben: Teilstring jeder Gruppe
 - \$~ Das erzeugt MatchData-Objekt (siehe Teil III)
- Beispiel zur Nutzung (aus dem *pickaxe*-Buch):

```
def showRE(a, re) # Trefferbereich hervorheben
  a =~ re ? "#{`}<<#{&}>>#{'}" : "no match"
end
```

```
showRE("yes | no", /\|/) # "yes <<|>> no"
```

```
a = "Eins zwei drei"
```

```
showRE(a, /ei/) # "Eins zw<<ei>> drei"
```

```
showRE(a, /ei/i) # "<<Ei>>ns zwei drei"
```



Reguläre Ausdrücke und *backreferences*



- Wirkung von Gruppenbildung mit runden Klammern (...)
 1. Wiederholungsangaben für ganze Gruppeninhalte möglich
 2. Globale Variablen \$1 ... \$9 sind danach mit Klammerwert gesetzt:

```
"aababcabcdabcde" =~ /(ab){2}/ # 1
```

```
line # "x = -5"
key, value = $1, $2 if line =~ /\s*(\w+)\s*=\s*([+-]?[d+])/
# key = "x", value = "-5"
```

3. Klammerwerte sind über \1, ..., \9 innerhalb des regulären Ausdrucks verfügbar ("*backreferences*").

– Beispiel 1: Suche nach doppelten Vokalen

```
"Woo sind hier doppelte Vokaale?" =~ /([aeiou])\1/i # 1
```

```
"Wo sind hier doppelte Vokaale?" =~ /([aeiou])\1/i # 25
```

– Beispiel 2: Suche nach doppelten Wörtern

```
"This has has been a word too much" =~ /(\w+)\s+\1/ # 5
```

– Beispiel 3: Flexible Suche nach *string delimiter*-Paaren

```
"He said 'Hello'" =~ /(['"]).*?\1/ # 8, $& == %('Hello')
```




Reguläre Ausdrücke II

Arbeiten mit regulären Ausdrücken



Suchen mit regulären Ausdrücken



- Beispiel: Zeilenfilter
 - Nur auf bestimmte Zeilen reagieren

```
comment_lines, unmatched_lines, dict = 0, 0, {}
while line = gets
  case line
  when /^#/                               # Kommentarzeilen zählen
    comment_lines += 1
  when /(\w+)\s*=\s*(\w.*)/               # key/value-Paar?
    dict[$1] = $2.strip
  when other_regex                         # usw.
    do_something_else
  else
    unmatched_lines += 1
  end
```



Suchen mit regulären Ausdrücken



- Beispiel: Intelligentes "Prompt"
 - Nur per Regex klar definierte Eingaben akzeptieren

```
def my_prompt( ptext, regex )
  loop do
    print ptext
    line = gets.chomp
    return $& if line =~ regex
    puts "Illegal value - please retry."
  end
end

age = my_prompt(
  "Enter age (0-129): ", /^d$|^d\d$|^1[012]\d$/
)
date_regex =
  /^20\d\d-((0[1-9])|(1[0-2]))-((0[1-9])|([12]\d)|(3[01]))$/
date = my_prompt("Enter current date as 20YY-MM-DD: ",
  date_regex)

# usw. / vgl. Demo!
puts "age = #{age}"
puts "date = #{date}"
```



Ersetzen mit regulären Ausdrücken



- Methoden `String#sub` & `String#gsub`
 - `sub`: Einmaliges Ersetzen (*substitution*)
 - `gsub`: Mehrmaliges Ersetzen (*global substitution*)
 - `sub!`, `gsub!`: Varianten, die den aktuellen String verändern.
- Einfacher Modus, mit Verwendung von *backreferences*:
 - `str.sub(regexp, replacement)`

```
"Hello, world".sub(/world/, "friends")
"Hello, friends"
"hello".sub(/([aeiou])/, '<1>') # "h<e>ll<o>"
"hello".gsub(/([aeiou])/, '<1>') # "h<e>ll<o>"
```

- Block-Modus, mit Verwendung der globalen Variablen:
 - `str.gsub(regexp) { |match| block }`

```
"Dies ist ein Satz".gsub(/\b\w/) {|c| c.upcase}
"Dies Ist Ein Satz"
"Dies ist ein Satz".gsub(/\b\w(.)/) {$1.upcase}
"Ies St In Atz"
```



```
def unescapeHTML( string ) # Beispiel aus pickaxe-Buch
  str = string.dup          # Auch für XML ideal geeignet
  str.gsub!(/&(.*?);/n) { # Löst char refs und
    match = $1.dup         # eingebaute entity refs auf!
    case match
    when /\Aamp\z/ni       then '&'
    when /\Aquot\z/ni      then '"'
    when /\Agt\z/ni        then '>'
    when /\Aalt\z/ni       then '<'
    when /\A#(\d+)\z/n     then Integer($1).chr
    when /\A#x([0-9a-f]+\z/ni then $1.hex.chr
    end
  }
  str
end
```

```
unescapeHTML("1&lt;2 &amp;&amp; 4&gt;3") # 1<2 && 4>3
unescapeHTML("&quot;A&quot;;=&#65;=&#x41;") # "A"=A=A
```



Reguläre Ausdrücke III

Das objekt-orientierte Interface
Die Klassen Regexp und MatchData



Reguläre Ausdrücke: Die Klasse Regexp



- Konstanten:

```
Regexp::IGNORECASE, Regexp::MULTILINE, Regexp::EXTENDED
```

- Klassenmethoden:

```
Regexp.compile, Regexp.new(pattern[,options[,lang]])
```

```
re = Regexp.new("abc\s*abc", Regexp::IGNORECASE, 'n')
# entspricht: re=/abc\s*abc/in
```

```
Regexp.last_match -> aMatchData
# entspricht $~
```

```
Regexp.quote, Regexp.escape(aString) -> aNewString
```

```
str = "ak2'$%'$%&$()" # Irgendwas!
str =~ %r#{Regexp.escape(str)} # Immer true !!
Regexp.escape('\*\*?{.}') # \\\*\*?{\}\. .
```

Verwenden Sie `Regexp.escape` bzw. `Regexp.quote`, wenn Sie nach Teilstrings suchen wollen, die Sonderzeichen enthalten könnten. Die Methode nimmt Ihnen die Sonderzeichenbehandlung einfach ab.



Reguläre Ausdrücke: Die Klasse Regexp



- Reguläre Methoden:

==

Vergleicht zwei Regexp-Objekte. Gleichheit gegeben bei Übereinstimmung in Suchmuster, Optionen und Zeichensatz:

```
/abc/ == /abc/ # true
/abc/ == /abc/i # false
/abc/u == /abc/n # false
```

===

Wird vom `case`-Statement verwendet. Wirkung dort wie `~`

=~

Vergleicht mit einem String. Siehe auch `String#=~`

```
/bc/i =~ "ABCD" # 1
```

~

Vergleicht mit `$_` - VERALTET; NICHT MEHR VERWENDEN!

```
# $_ = "Ein Test"
~/Te/ # 4
```



Reguläre Ausdrücke: Die Klasse Regexp



- Reguläre Methoden:

casefold? Liefert Status der Option IGNORECASE

```
/abc/.casefold? # false
/abc/i.casefold? # true
```

kcode Liefert aktuellen Character set code

```
/abc/.kcode # nil
/abc/n.kcode # "none"
/abc/u.kcode # "utf8"
```

match Liefert ein Matchdata-Objekt & setzt \$~ oder ergibt nil

```
/(.(\w)/.match( "abcd" ) # ergibt 0; $1 = "c"
"abcd" =~ /.(w)/ # analoge Wirkung
```

Vorteil von match: Jede Benutzung ergibt ein neues MatchData-Objekt, das das Ergebnis festhält. Siehe unten (MatchData).

source Liefert den Test-String des Ausdrucks.

```
re = Regexp.new("abc", Regexp::IGNORECASE, "u")
# ...
re.source # "abc"
```



Reguläre Ausdrücke: Die Klasse MatchData



- Anmerkungen

- Diese Klasse kapselt die Ergebnisse eines *match*-Operators, die gewöhnlich in globalen Variablen vorgehalten werden und stellt Methoden zum einfachen Zugriff auf diese Daten bereit.
- Durch Speicherung der *match*-Ergebnisse in separaten Objekten ist konfliktfreier Zugriff auf mehrere zurückliegende Mustervergleiche möglich.

- Konstanten, Klassenmethoden:

- Keine

- Typische Nutzung:

```
regex = Regexp.new( '\s*(\w+)\s*=\s*([-\+]?[d+]) ' )
if (md = regex.match( " x = -5 cm" ))
  x = md[1]; y = md[2] # md[i] wie $i, i=1..9
  units = md.post_match # wie $'
end
```



- Reguläre Methoden:

`[]` Für array-artigen Zugriff auf `$0`; `$1..$9`

`m[0]` # `$&` (ganzer Trefferbereich)

`m[1]` # `$1`

`m[2,3]` # `[$2, $3, $4]` (n, m: Start, Länge)

`m[1..3]` # `[$1, $2, $3]` (n..m: Indexbereich)

`begin` Offset des ersten Zeichens von Gruppe `i` (`$i`)

`end` Offset+1 des letzten Zeichens von Gruppe `i` (`$i`)

`md.begin(0)` # 0 (Offset von " x = ...")

`md.begin(1)` # 2 (Offset von "x")

`md.begin(2)` # 6 (Offset von "-5")

`md.end(0)` # 8 (Offset von "5" +1; vgl. `$&`)

`md.end(1)` # 3 (Offset von "x" +1)

`length, size` Analog zu Klasse `Array`

`md.length` # 3 (für `$0, $1, $2`)

`offset` Array mit `begin-` und `end-`Werten

`md.offset(0)` # [0, 8]

`md.offset(1)` # [2, 3]



- Reguläre Methoden:

`post_match` `$'` (String hinter dem Treffer)

`md.post_match` # ' cm'

`pre_match` `$`` (String vor dem Treffer)

`md.pre_match` # '' (hier leer)

`string` (konstanter) "match"-String

`md.string` # " x = -5 cm"

`to_a` Wandelt in echtes Array um

`md.to_a` # [" x = -5", "x", "-5"]

`to_s` `$&` (String der Trefferregion)

`md.to_s` # " x = -5"



Reguläre Ausdrücke IV

extended regular expressions
nach POSIX 1003.2



Extensions in regulären Ausdrücken



- **Grundmuster:**

(? . . .)

- Eingeleitet mit runder Klammer (und Fragezeichen ?
- Steuerzeichen direkt hinter dem Fragezeichen!
- Abgeschlossen mit runder Klammer)
- Nicht mit einfacher Gruppe (...) verwechseln!
- Gruppenwirkung besteht (z.B. für |), aber:
- Globale Variablen \$1, ..., sowie *backreferences* \1, ... werden nicht gesetzt!

- **(?# Kommentar)**

- Ermöglicht das Einfügen von Kommentartexten in reguläre Ausdrücke. Nützlich bei komplexen Fällen!



Extensions in regulären Ausdrücken



- `(?:re)`
 - Gruppenbildung ohne Erzeugung von `$1`, `\1` etc.

```
datum = "14.11.2003"
r1 = /(\d+) (-|\.) (\d+) (-|\.) (\d+)/
m1 = r1.match( datum )
m1.to_a      # ["14", ".", "11", ".", "2003"]
r2 = /(\d+) (?:-|\.) (\d+) (?:-|\.) (\d+)/
m2 = r2.match( datum )
m2.to_a      # ["14", "11", "2003"]
```

- `(?=re)`
 - Matchbedingung ohne "verbrauchende" Wirkung

```
showRE("1, 2, 3", /\d+,/)      # "<<1,>> 2, 3"
showRE("1, 2, 3", /\d+(?=,)/) # "<<1>>, 2, 3"
"1, 2, 3".scan(/\d+(?=,)/)    # ["1", "2"]
```



Extensions in regulären Ausdrücken



- `(?!re)`
 - Wie `(?=re)`, aber wahr, wenn `re` nicht zutrifft

```
showRE("1, 2, 3", /\d+(?!,)/) # "1, 2, <<3>>"
```

- `(?>re)`
 - Ermöglicht verschachtelte reguläre Ausdrücke. Diese können die Effizienz eines Ausdrucks erheblich steigern (Laufzeitoptimierung)!
Beispiel:

```
require "benchmark"
include Benchmark
# Zeichenfolge suchen: a, beliebig, b, beliebig, a
str = "a" + ("b"*5000) # Böswilliges Gegenbsp.!
bm(8) do |test|
  test.report("Normal: ") { str =~ /a.*b.*a/ }
  test.report("Versch.: ") { str =~ /a(?>.*b).*a/ }
end
```

- Ergebnis: Siehe online-Demo! Diskussion: Ursachen?



Extensions in regulären Ausdrücken



- `(?>re)` (Forts.)
 - Was ist bei "a.*b.*a" passiert beim Anwenden auf "abbbbb...bb"?
 - 1. "a" wird sofort gefunden
 - 2. ".*" ist die "gierige" Wiederholungsvariante - sie liest erst mal alles (und merkt es sich für eventuelles *backtracking*)
 - 3. Passt "b" ? Nein --> *backtracking* um ein Zeichen
 - 4. *backtracking* ok? Ja: Weiter mit (3), Nein: Suche erfolglos beenden.
 - 5. Zweites ".*" anwenden (auch hier: Rest einlesen & merken)
 - 6. Passt "a" ? Ja: Erfolg, Ende.
Nein: --> *backtracking*
 - 7. *backtracking* (2. Block) ok? Ja: (6) Nein: (3)
- Fazit:
 - Es gibt reguläre Ausdrücke mit Laufzeiten, die exponentiell sind in Bezug auf die Länge des untersuchten Strings! `(?>re)` vermeidet derartige Konstellationen durch Verhindern des *backtracking*.



Extensions in regulären Ausdrücken



- `(?imx)`
- `(?-imx)`
 - Ein- bzw. Ausschalten einer der Optionen i, m, x. Bei Verwendung innerhalb einer Gruppe bleibt die Wirkung auf diese Gruppe beschränkt!
- `(?imx:re)`
- `(?-imx:re)`
 - Ein- bzw. Ausschalten einer der Optionen i, m, x nur für *re*.



- **Zeichenmengen** nach POSIX 1003.2

- Benutzung: `[:name:]`
- Zulässige Namen:

```
:alnum:, :alpha:, :blank:, :cntrl:,  
:digit:, :graph:, :lower:, :print:,  
:punct:, :space:, :upper:, :xdigit:
```

- **Siehe auch:**

`regex(3)`, `regex(7)`, `wctype(3)`

- **Beispiel:**

```
ShowRE("abc12def", /[0-9]+/) # "abc<<12>>def"  
ShowRE("abc12def", /[\d]+/) # "abc<<12>>def"  
ShowRE("abc12def", /[[[:digit:]]+]/) # "abc<<12>>def"
```

- **Hinweis:**

- Zeichenmengen erfordern Verallgemeinerungen, wenn sie auf internationale Zeichensätze angewendet werden. Beispiel:
- Einige XML-Standards wie XML Schema und XPath verwenden verallgemeinerte reguläre Ausdrücke im Kontext von Unicode.



**Schließlich: Die folgenden POSIX-Standards sind
NICHT IN RUBY VERFÜGBAR:**

Zeichenfolgen wie Einzelzeichen behandeln

- Benutzung: `[[:chars.]]`
- Wirkung: Die in `[..]` eingeschlossene Zeichenfolge wird innerhalb `[]` wie ein Zeichen verwendet.
- **Beispiel** (wie es lauten könnte):

```
ShowRE("chchcc", /[[[:ch.]]*c/) # "<<chchc>>c"
```

Äquivalenzklassen bilden

- Benutzung: `[[:c=]]`
- Wirkung: Wenn z.B. `[[:ch.]]` definiert ist, sind `[[:c=]]` und `[[:h=]]` äquivalent. Wenn nicht, ist die Wirkung wie `[[:c.]]` bzw. `[[:h.]]`.
- **Beispiel** (wie es lauten könnte):

```
ShowRE("Ich mag Chemie", /[[[:ch.]].*[[[:c=]]]/i)  
# "I<<ch mag Ch>>emie"
```



In Ruby eingebaute Dinge

Scripting-Komfort
Voreingestellte globale Variablen
Vordefinierte globale Konstanten
Standardklassen
Standardmodule



Scripting-Komfort



Ruby besitzt zahlreiche **Programmoptionen**. Einige erleichtern das Schreiben einzelner, wirkungsvoller Kommandos:

Option -a

Wirkt wie `$F = $_.split`

Wird in Verbindung mit -n bzw. -p eingesetzt. Siehe Option -F

Option -n

Wirkt wie `while gets; ...; end` um den angegebenen Code:

```
# Wirkt wie "grep":  
$ ruby -n -e "print if /root/" /etc/passwd  
# Wirkt wie "cut":  
$ ruby -a -n -F':' -e "puts $F[6]" /etc/passwd
```

Option -p

Wirkt wie `while gets; ...; print; end` um den Code in -e:

```
# Wirkt wie "tr":  
$ echo $HOME | ruby -p -e "$_.tr! ("ehlw", "EHLW")  
/Local0/WErntgEs
```



Zugriff auf Umgebungsvariablen:

- Sehr einfach durch `ENV`
(hinter dem sich ein Singleton der Klasse Object verbirgt).
- ENV verhält sich weitgehend wie ein Hash, ist aber keiner!
- Bei Bedarf in Hash umwandelbar mit Methode `to_hash`
- Beispiele:

```
$ echo $HOME # Shell-Ausgabe
/local0/werntges
ruby -e 'puts ENV["HOME"]' # Analog in Ruby
/local0/werntges # Nun Ändern in Ruby:
ruby -e 'ENV["HOME"]="var/tmp"; puts ENV["HOME"]'
/var/tmp
$ echo $HOME # Wert im Elternprozess nicht geändert:
/local0/werntges
ENV["PATH"] += ":/my/bin" # Bequemes Arbeiten in Ruby
fork-Demo!
```



Zugriff auf Kommandozeilen-Optionen und -Parameter:

- Sehr einfach durch Array `ARGV`
- Alle von Ruby selbst nicht verbrauchten Werte finden sich in ARGV.
- Beispiele:

```
$ cat argv.rb
ARGV.each {|arg| puts arg}
$ argv.rb -x -f myfile
-x
-f
myfile
$ ruby -v argv.rb -v -x
ruby 1.8.0 (2003-08-04) [i686-linux]
-v
-x
```

- ACHTUNG - anders als in "C" ist ARGV[0] nicht das aktuelle Kommando (das findet sich in \$0), sondern das erste Argument!
- Hinweis: Bequeme Verarbeitung von Optionen mit **GetoptLong**



Automatisches Lesen von Quelldaten:

- Zeilenweises Einlesen aus einer Datei, mehreren Dateien oder stdin ist eine häufige Tätigkeit. Ruby bietet dafür einen Automatismus an:
- **ARGF** bietet Zugriff auf die (virtuelle) Konkatenierung aller Dateien, die per Kommandozeilen-Argument übergeben wurden.
- Sind keine angegeben, wird ARGF mit stdin identifiziert.
- **ARGF** (Synonym: **\$<**) ist ein spezielles Objekt mit Methoden wie Klasse File. Auch vorhanden: **fileno**, **filename**, **lineno**, **pos**.
- Methode **Kernel.gets** macht davon Gebrauch. Sie können also Textdateien einlesen & verarbeiten, ohne sie explizit zu öffnen und zu schließen:

```
# Wirkt wie "wc -l":  
$ ruby -e 'cw=0; while gets; cw+=1; end; puts cw' *.rb  
$ cat *.rb | ruby -e'cw=0; while gets; cw+=1; end; puts  
  cw'  
$ cat *.rb | wc -l # Sollte denselben Wert liefern
```



Globale Variablen

... teils aus der Perl-Tradition,
teils Ruby-eigene



- Darstellung
 - Die Darstellung erfolgt nach Sachgebieten
 - Jeder Eintrag beginnt mit einer (komma-separierten Liste synonymmer) Variablen, gefolgt vom
 - 1) Klassennamen des Objekts und
 - 2) (optional) in Klammern einer Default-Belegung.
 - Nächste Zeile: Erläuterungen, stichwortartig
 - Beispiel:


```
$>, $defout          IO ($stdout)
Ziel von Kernel#print / printf
```
- Inhalt
 - Erläuterungen sind sehr knapp. Primäres Ziel ist zu zeigen, was existiert.



- **Exception handling**
 - `$!` Exception
Das letzte Ausnahmeobjekt
 - `$@` Array
Stack backtrace;
vgl. `Exception#backtrace`
- **Separatoren**
 - `$/, $-0` String (newline)
Zeilentrennzeichen, wie von `gets`, `readline` verwendet.
 - `$\` String (nil)
Output record separator, wie von `print` und `IO#write` verw.
 - `$,` String (nil)
von `print` und `Array#join` zwischen Parametern verwendet
 - `$;, $-F` String (nil)
Trennzeichen für `split`
- **Input/Output**
 - `$stdin` IO (STDIN)
 - `$stdout` IO (STDOUT)
 - `$stderr` IO (STDERR)
 - Selbsterklärend
 - `$.` Fixnum
Nummer der zuletzt gelesenen Zeile der aktuellen Eingabedatei.
Entspricht `ARGF.lineno`
 - `$<` Object
Synonym von `ARGF`, s. dort
Ähnlich wie ein File-Objekt!
 - `$>`, `$defout` IO (\$stdout)
Ziel von `Kernel#print / printf`
 - `$FILENAME` String
entspricht `ARGF.filename`



Globale Variablen



• Laufzeitumgebung

- `$0` String
Name des gerade laufenden Ruby-Programms
- `$$` Fixnum
... und seine PID
- `$?` Fixnum
Exit-Status (errno) des zuletzt beendeten Prozesses
- `$:`, `$LOAD_PATH` Array
Die Verzeichnisse, in denen load und require Dateien erwarten.
- `$"` Array
Array mit Namen der von require geladenen Dateien.
- `$SAFE` Fixnum
Sicherheitsstufe (0), 0-4
- `__FILE__` String,
- `__LINE__` String
Name und aktuelle Zeilennummer der Quelltextdatei.

• Kommandozeile

- `$DEBUG` Object
true, falls -r oder --debug
- `$VERBOSE`, `$-v`, `$-w` Object
true, falls -v, -w oder --verbose gesetzt
- `$F` Array
erhält Ergebnis von split, wenn -a und (-p oder -n) gesetzt.
- `$_a` / `$_l` / `$_p` Object
true, wenn -a / -l / -p gesetzt
- `$_x`
Wert der Option -x
(0, a, d, F, i, K, l, p, v)
- `$*` Array
Synonym von ARGV



Globale Variablen



• Variablen, die von \$~ abhängen und auf die nicht zugewiesen werden darf:

- `$1`, `$2`, `$3`, ...
entspricht `$~[1]`, `$~[2]`, ...
- `$&` Letzter Trefferbereich
- `$`` String vor dem letzten Trefferbereich
- `$'` String hinter dem letzten Trefferbereich
- `$+` Inhalt der letzten Regex-Gruppe des letzten "match"

• Bemerkungen

- Alle o.g. Var sind String-Objekte.
- Alle Var. werden von einem erfolglosen `=~` auf `nil` gesetzt.
- Sonderfall, änderbar:
- `$=` Object (nil)
Auf `true` setzen: Entspricht `/.../i`

• Lokale Variablen

- `$_` Die zuletzt von `gets` oder `readline` eingelesene Zeile im aktuellen *scope*.
- `$~` Das zuletzt erzeugte MatchData-Objekt



Globale Konstanten



Globale Konstanten



- **Elementare Konstanten**
 - TRUE** Synonym von true
 - FALSE** Synonym von false
 - NIL** Synonym von nil
- **Spezielles**
 - DATA**
Gestattet das Einlesen von Zeilen hinter `__END__` im Quelltext, falls vorhanden.
 - TOPLEVEL_BINDING** Binding (binding-Konzept ausgelassen)
 - RUBY_PLATFORM** String
z.B. "i686-linux"
 - RUBY_RELEASE_DATE**
 - RUBY_VERSION** String
z.B. "1.8.0"
- **K. für die Kommandozeile**
 - ARGF, \$<** Object
ARGF besitzt ähnliche Methoden wie ein File-Objekt!
Lesen von ARGF bewirkt sequenzielles Lesen von allen Dateien, die als Kommandozeilen-Argumente angegeben wurden, Lesen von \$stdin sonst.
 - ARGV, \$*** Array
Array der Argumente der Kommandozeile
 - ENV** Object
Zum hash-artigen Zugriff auf Umgebungsvariablen.
 - STDIN, STDOUT, STDERR** IO
Die festen Standard-Stream als Konstanten, Default-Werte von \$stdin, \$stdout, \$stderr.



Standardklassen

Ruby's eingebaute Klassen:
Eine Übersicht



Die 32 Standardklassen: Übersicht



- **Numerische Klassen**
 - Bignum, Fixnum, Integer, Float, Numeric
- **I/O-Klassen**
 - Dir, IO, File, File::Stat
- **"Basistypen"**
 - Array, Hash, String, Range
- **Boole'sche Klassen**
 - TrueClass, FalseClass
- **Reguläre Ausdrücke**
 - Regexp, MatchData
- **OO-Klassen**
 - Binding, Class, Method, Module, Object, Symbol
- **Strukturbildung**
 - Struct, Struct::Tms
- **Code-Ausführung**
 - Binding, Continuation, Proc, Thread, ThreadGroup
- **Systemnahe Klassen**
 - Exception, Time



- **Comparable**
 - Vergleichsoperatoren
- **Enumerable**
- **Errno**
 - kapselt Fehlercodes des Betriebssystems
- **FileTest**
 - Alternative Methoden zu denen aus File::Stat
- **GC**
 - Garbage Collector
- **Kernel**
 - Zahlreiche Methoden rund um "libc"
- **Marshal**
 - Objektpersistenz
- **Math**
- **ObjectSpace**
 - Für GC
- **Process**
 - Verwaltung der OS-Prozesse



Array

Ausgewählte Beispiele

Hinweis zu Klasse Array:

mix-ins: Enumerable (16 Methoden)

Klassen-Methoden: 2

Normale Methoden: 61



Array: Ausgewählte Beispiele



- **Einfügen & Entfernen**
 - Am Anfang, mit **shift** & **unshift**:

```
a = [1, 2, 3, 4]
a.unshift(0)           # [0, 1, 2, 3, 4]
b = a.shift            # b==0, a==[1, 2, 3, 4]
```
 - Am Ende, mit **push** & **pop**:

```
a = [1, 2, 3, 4]
a.push( 5 )           # [1, 2, 3, 4, 5]
a << 6                # [1, 2, 3, 4, 5, 6]
b = a.pop             # b==6, a==[1, 2, 3, 4, 5]
```
 - Veränderungen an beliebiger Stelle, mit Index-Zugriffen:

```
a = [1, 2, 3, 4]
a[1, 2] = ["2", "3"]  # [1, "2", "3", 4]
a[1, 2] = 3           # [1, 3, 4]
a[1, 0] = 2           # [1, 2, 3, 4]
a.delete_at(2)        # 3; a==[1, 2, 4]
```



Array: Ausgewählte Beispiele



Stack-Beispiel

```
class Stack
  def initialize
    @store = []
  end
  def push(x)
    @store.push x
  end
  def pop
    @store.pop
  end
  def peek
    @store.last
  end
  def empty?
    @store.empty?
  end
end
```

Queue-Beispiel

```
class Queue
  def initialize
    @store = []
  end
  def enqueue(x)
    @store << x
  end
  def dequeue
    @store.shift
  end
  def peek
    @store.first
  end
  def empty?
    @store.empty?
  end
  def length
    @store.length
  end
end
```



Array: Ausgewählte Beispiele



- Mengen-Operationen mit Arrays

- Vereinigungs- und Schnittmenge bilden:

```
a = [1, 2, 3, 4, 5];   b = [3, 4, 5, 6, 7]
c = a | b             # [1, 2, 3, 4, 5, 6, 7]
d = a & b             # [3, 4, 5]
# Man beachte die Wirkung auf doppelte Einträge:
e = [1, 2, 2, 3, 4];   f = [2, 2, 3, 4, 5]
g = e & f             # [2, 3, 4]
```

- Bem.: Operator '+' entfernt Doppelte nicht!

- Differenzmenge bilden:

```
a = [1, 2, 3, 4, 5];   b = [4, 5, 6, 7]
c = a - b              # [1, 2, 3]
```



Array: Ausgewählte Beispiele



- Mengen-Operationen mit Arrays

- Element-Beziehung prüfen:

```
x = [1, 2, 3]
puts (x.include? 2) ? "Enthalten" : "Nicht enthalten"
# Prüfung, ob Array x ein Element 2 enthält
# Funktioniert, ist aber nicht die übliche Lesart.
```

- "Element-von" Operator ergänzen:

```
class Object
  def in( other )
    other.include? self
  end
end
x = [1, 2, 3]
puts (2.in x) ? "Enthalten" : "Nicht enthalten"
# Schon fast wie 2 ∈ x
```



- Mengen-Operationen mit Arrays
 - Teilmengen- und Obermengen-Prüfungen:

```
class Array
  def subset?( other )
    self.each {|x| return false unless other.include? x}
    true
  end
  def superset?( other )
    other.subset?( self )
  end
end
```

```
a = [1, 2, 3, 4, 5]; b = [2, 3]; c = [2, 3, 4, 5]
c.subset? a # false
b.subset? a # true
c.superset? b # true
```



- Mapping
 - Wenn jedes Element eines Arrays durch ein von ihm abgeleitetes ersetzt werden soll, spricht man auch von einer Abbildung (*map*).
 - Für tabellarische Mapping-Regeln bieten sich Hash-Tabellen an.
 - Für funktionale Mapping-Regeln gibt es die Array-Methode **collect** bzw. **map**, die ein gleichlanges Array zurückliefert:

```
[16, 17, 18, 19].collect do age
  age < 18 ? "minderjährig" : "volljährig"
end
# ["minderjährig", "minderjährig", "volljährig",
  "volljährig"]
```

```
Array(1..6).map {|n| n*n} # [1, 4, 9, 16, 25, 36]
```



- **"pack": Ein leistungsfähiges Perl-"Erbe"**
 - **pack** und **unpack** gestatten das Umcodieren von Arrays und Strings ("Einpacken von Datenstrukturen in kompakte Strings).
 - Die Möglichkeiten sind vielseitig und leistungsfähig, benötigen aber etwas Einarbeitung.
 - Sie gehören zum Repertoire der Skriptsprachen!
 - Details: Siehe Referenzhandbuch. Hier ein paar Anregungen:

```
a=["a", "b", "c"]; n=Array(65..70); u=Array(126..130)
```

```
a.pack('A3A3A3') # "a b c "
```

```
a.pack('a3a3a3') # "a\000\000b\000\000c\000\000"
```

```
n.pack('ccc') # "ABC"
```

```
n.pack('c*') # "ABCDEF"
```

```
# UTF-8 Codierung:
```

```
u.pack('U*') # "~\177\302\200\302\201\302\202"
```

```
# base-64 Codierung:
```

```
["bin string\001\002"].pack('m') # "YmluIHN0cmVudWEC\n"
```



Hash-Tabellen

Ausgewählte Beispiele

Hinweis zu Klasse Hash:

mix-ins: [Enumerable \(16 Methoden\)](#)

[Klassen-Methoden: 2](#)

[Normale Methoden: 38](#)



Hash: Ausgewählte Beispiele



- **Benutzung eines Defaults**
 - Zählen welches Wort wie oft erscheint, in einem Durchlauf und ohne *a priori* Wissen über mögliche Wörter:

```
h = Hash.new(0)      # Default: 0 (statt: nil)
word_list.each do |word|
  h[word] += 1      # h[word]==0 bei jedem neuen Wort!
end
```

```
puts "Worthäufigkeiten-Report"
# Sortiert nach Wörtern, alphabetisch:
h.keys.sort.each {|word| puts "#{word}\t #{h[word]}"}
# Sortiert nach Häufigkeiten, absteigend:
h.keys.sort {|a, b| h[b] <=> h[a]}.each do |word|
  puts "#{word}\t #{h[word]}"
end
```

- An der Tafel: Wie könnte man einen Defaultwert in Klasse Hash einbauen, wenn es ihn nicht schon gäbe?



Hash: Ausgewählte Beispiele



- **Regeln für key-Objekte**
 - Jedes Objekt kann *value* eines Hashes sein:

```
h["t1"] = Tree.new( some parameters )
h["t2"] = Tree.new( some other parameters )
h["t1"].class      # "Tree"
```

- Im Grunde kann jedes Objekt auch *key* eines Hashes sein, aber mit einer Einschränkung: Es darf seinen Wert nicht ändern!

```
a = ["a", "b"]; c = ["c", "d"]
h = {a => 100, c => 300}
h[a]      # 100
a[0] = "z" # a wird verändert!
h[a]      # nil
```

- **Ausweg: Methode "rehash"**

```
h.rehash      # [{"c", "d"} => 300, [{"z", "b"} => 100]
h[a]          # 100
```



- **Hintergrund zur Regel für key-Objekte: Object#hash**
 - Berechnet einen Fixnum-Wert für jedes beliebige Objekt
 - **Notwendige Eigenschaft:**

```
a.eql?(b) ==> a.hash == b.hash
```

(Haben zwei Objekte den gleichen Wert, müssen ihre Hash-Werte übereinstimmen)

- **Wirkung von Hash#rehash:**
 - Erneuerung der internen Tabelle der Hash-Werte!



Strings

Ausgewählte Beispiele

Hinweis zu Klasse String:

mix-ins: Enumerable (16 Methoden), Comparable (6)

Klassen-Methoden: 1

Normale Methoden: 75



- **Bereits vorgestellte Methoden**

```
+ , <<, concat
* , % , <=>, ==, ===
=~ , ~ ,
[], []=
each, each_line, each_byte
length, size, empty?
center, ljust, rjust
chop, chomp, strip
upcase, downcase, swapcase, capitalize
scan, sub, gsub
tr
crypt
succ
to_s, to_i, to_f,
split
```



- **unpack: Das Gegenstück zu Array#pack**

- unpack liefert grundsätzlich ein Array zurück.
- Die darin enthaltenen Objekte gehören zu Klassen, die sich aus dem Formatstring ergeben.
- Beispiel: Umwandlung UTF-8 in ISO-8859-1

```
def utf8_to_iso8859_1( utf8str )
  s, msb, lsb = '', 0, 0
  utf8str.unpack("U*").each do |i|
    if i & 0x80 == 0 then s << x
    elsif i <= 0xc3 && i >= 0xc0 then msb = i & 0x3
    elsif i >= 0x80 && i <= 0xbf
      lsb = i & 0x3f
      s << ((msb << 6) | lsb)
    else raise "Not an ISO-8859-1 char: #{i}"
    end
  end
end
```

```
utf8_to_iso8859_1("~/177\302\200\302\201\302\202")
# [126, 127, 128, 129, 130]
```



Structs

(mögliche spätere Ergänzung)



Aufruf von Methoden

Variable Argumentlisten
Übergabe von Code-Blöcken



- **Der allgemeine Methodenaufruf**
 - Abstrakte Darstellung, Zusammenfassung

```
[ receiver. ] name [ parameters ] [ block]
[ receiver:: ] name [ parameters ] [ block]

parameters <-- ( [param, ...] [, hashlist]
                [, *array] [, &aProc ] )

block          <-- { blockbody }
                  do blockbody end
```



- **Parameterübergabe *by reference***
 - Alle Parameter werden *by reference* übergeben:

```
def my_meth(a)
  a[2] = "set by my_meth"
end
x = [1, 2, 3]
x           # [1, 2, 3]
my_meth(x)  # "set by my_meth"
x           # [1, 2, "set by my_meth"] Geändert!
```

- Aber:

```
def my_meth2(a)
  a = a.succ
end
x = "abc"      # "abc"
my_meth2(x)    # "abd"
x              # "abc"      Nicht geändert!
my_meth2(1)    # 2
```

Diskussion der Ursachen
an der Tafel



- **Überschreiben von Methoden?**

- Ein (triviales) Beispiel aus C++ :

```
// Deklarationen für die Betragsbildung:  
// (man ignoriere typecast-Möglichkeiten)  
double abs(int);  
double abs(double);  
double abs(complex);  
double abs(char*); // Zahl als String  
// Drei Funktionen zu implementieren ...
```

- Überschreiben entfällt bei typenlosen Sprachen!

```
def abs(x)  
  return x.abs      if x.instance_of? Float  
  return x.abs.to_f if x.instance_of? Integer  
  return x.to_f.abs if x.instance_of? String  
  # usw. Könnte man sogar dynamisch erweitern.  
end
```



- **Hash-"Automatik"; variable Argumentzahl annehmen**

- Implizite Hash-Bildung am Ende einer normalen Parameterliste

```
def my_meth(a, b, c)  
  a # 1  
  b # 2  
  c # {3 => "str1", 4 => "str2", 5 => "str3"}  
end  
my_meth( 1, 2, 3 => "str1", 4 => "str2", 5 => "str3" )
```

- Explizite Annahme einer variablen Anzahl Argumente in ein Array:

```
def varargs( a, *rest )  
  "a=#{a} und Liste=#{rest.join('/')}"  
end  
varargs(1) # "a=1 und Liste="   
varargs(1,2) # "a=1 und Liste=2"   
varargs(1,2,3,4) # "a=1 und Liste=2/3/4"
```



Methoden aufrufen, allgemein



- **Variable Anzahl Argumente & Code-Blöcke übergeben**

- Variable Anzahl Argumente aus einem Array übergeben:

```
def my_meth( a, b, c, d, e )
  "Erhalten: #{a} #{b} #{c} #{d} #{e}"
end
my_meth(1, 2, 3, 4, 5)      # "Erhalten: 1 2 3 4 5"
my_meth(1, 2, 3, *['a', 'b']) # "Erhalten: 1 2 3 a b"
my_meth(*(10..14).to_a)    # "Erhalten: 10 11 12 13 14"
```

- Code-Blöcke als Parameter, Präfix &

```
class RegNode
  def each(&b)
    yield self      # Übergebenen Block ausführen
    @children.each { |node| node.each(&b) }
  end
end
```



Methoden aufrufen, allgemein



- **Dynamische Erkennung der Verwendung eines Blocks**

- Erinnerung an String#sub:

```
"test string".sub(/st/, "***")      # "te** **ring"
"test string".sub(/st/) {|s| '**'} # Blockform!
```

Frage: Wie realisiert man das??

- Implementierbar mit Methode Kernel#**block_given?**

```
def try
  if block_given?
    yield      # Block ausführen / Rückgabewert
  else
    "no block" # Default-Rückgabewert
  end
try          # "no block"
try{'hello'} # "hello"
try do "Hello" end # "Hello"
```



Mehr zu OO in Ruby

Das Nachrichtenaustauschprinzip
Schutz von Methoden
Entwurfsmuster (I)



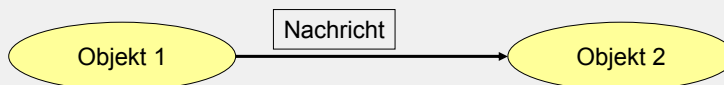
Das Nachrichtenaustauschprinzip



- Das Nachrichtenaustauschprinzip:

Objekte sind eigenständige Einheiten, deren Zusammenarbeit und Interaktionen mit Hilfe von Nachrichten bewerkstelligt wird, die sich die Objekte untereinander zusenden.

B. Oesterreich, Objektorientierte Softwareentwicklung



- Übliche Umsetzung:
 - Nachricht = Methodenaufruf
- Beispiel in Ruby:

```
# Teil einer Methode "fahren" der Klasse "Autofahrer":  
a = Car.new  
a.start_engine # Autofahrer (self) sendet Nachricht an a  
a.set_blinker("left") # weitere Nachricht, mit Param.
```



- Direkte Umsetzung des N.-Prinzips in Ruby:

```
# "Autofahrer"-Beispiel, einmal anders:  
a = Car.new  
# <self> sendet Nachricht "start_engine" an a  
a.send( :start_engine ) # Per Symbol  
a.send( "set_blinker", "left") # auch per Name!
```

- Auch Methoden selbst können in Ruby Objekte sein:

```
a = Car.new  
start = a.method( :start_engine )  
blink = a.method( "set_blinker" )  
# <self> sendet Nachrichten:  
start.call  
blink.call( "left" )
```

- Einzelheiten: Siehe Standardklasse "Method"



- **Zentrale Frage beim Thema Methodenschutz:**

- Wer darf welche Nachrichten an welche Objekte senden?

- **Implementierung**

- Methoden schützen & kapseln mit **Schlüsselwörtern**
`private, protected, public`
- Übernahme von Vorgehensweisen aus anderen OO-Sprachen!

- **public**

- Methoden sind standardmäßig "public", können also von jedem angewendet werden.

- **protected**

- "protected" Methoden können nur von Objekten derselben Klasse oder einer ihrer Unterklassen angewendet werden.

- **private**

- "private" Methoden können nur vom Objekt auf sich selbst angewendet werden.



- **Anwendung**
 - Erste Form:
Alle Methoden unterhalb eines der Schlüsselwörter unterliegen der jeweiligen Eingruppierung:

```
class ProtectionDemo
```

```
  def pub_meth1  
    # ...  
  end
```

```
  protected
```

```
  def prot_meth  
    # ...  
  end
```

```
private
```

```
  priv_meth  
    # ...  
  end
```

```
public
```

```
  def pub_meth2(x)  
    # ...  
  end
```

```
end # class ProtectionDemo
```

- **Bemerkungen:**
 - pub_meth1 ist "public" per Default.
 - Die anderen Methoden unterliegen der direkten Regelung ihrer Eingruppierung.



- **Anwendung**
 - Zweite Form:
Den Schlüsselwörtern folgen Listen der Symbole zu den jeweiligen Methoden:

```
class ProtectionDemo
```

```
  def pub_meth1  
    # ...  
  end
```

```
  def pub_meth2(x)  
    # ...  
  end
```

```
  def prot_meth  
    # ...  
  end
```

```
  def priv_meth  
    # ...  
  end
```

```
  public :pub_meth1,  
         :pub_meth2
```

```
  protected :prot_meth
```

```
  private :priv_meth
```

```
end # class ProtectionDemo
```

- **Bemerkungen:**
 - Die Methode "initialize" einer jeden Klasse ist automatisch "private".



- Beispiel "Kontoführung":

```
class Konten          # Vorgeschichte ausgelassen.
  private
    def belastung( konto, betrag )
      konto.wert -= betrag
    end
    def gutschrift( konto, betrag )
      konto.wert += betrag
    end
# Nur Methoden veröffentlichen, die vollständige
# Buchungen darstellen (Transaktionsschutz):
  public
    def spare( betrag )
      belastung( @gehaltskonto, betrag )
      gutschrift( @sparkonto, betrag )
    end
end
```



- Abstrakte Klassen:

- Klassen, die gemeinsame Methoden und Attribute für ihre Unterklassen enthalten, die aber selbst keine Exemplarbildung zulassen.
- Ruby-Beispiel: **Integer**
Diese Klasse besitzt keine (zugängliche) "new"-Methode:

```
# Normale Klassen:
String.methods.grep /new/ # ["new"]
Array.methods.grep /new/ # ["new"]

# "Abstrakte" Klasse:
Integer.methods.grep /new/ # []

# Aber auch:
Numeric.methods.grep /new/ # ["new"]
```



Entwurfsmuster (I)

(*design patterns*)



Entwurfsmuster



- Hintergrund:
 - Unabhängig von einer konkreten OO Programmiersprache ergeben sich bestimmte Standardsituationen immer wieder.
 - Sog. Entwurfsmuster (engl.: *design patterns*) entstanden, um diese Situationen sprachunabhängig in einem allgemeinen Sinne zu lösen (*high level building blocks*, bewährte Lösungsideen).
- Es gibt
 - **Architekturmuster** (für den Grobentwurf),
 - normale **Entwurfsmuster** (für den Feinentwurf) und
 - **Idiome** (sprachspezifische Lösungsbeschreibungen).
- Standardwerk zum Thema:
 - E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, 1995.



- **Beispiel 1: Besucher (*visitor*)**
 - Das Besucher-Muster stellt dar, wie für eine Menge von Objekten eine Operation iterativ ausgeführt werden kann, ohne dass die beteiligten Objekte wissen müssen, wer und wie über sie mit dieser Operation iteriert.
- **Realisierung in Ruby ?**
 - Geradezu idealtypisch - in Form des Standarditerators "**each**" !
- **Beispiel 2: Singleton**
 - Ein Singleton ist eine Klasse, von der es global nur ein einziges Exemplar gibt. Es wird aus normalen Klassen mittels Mechanismen gewonnen, die die Erzeugung weiterer Objekte verhindern.



- **Konzept** zur Realisierung von Singleton-Klassen in Ruby:
 - Die Klassenmethode "new" wird "private" erklärt.
 - Eine neue Klassenmethode tritt an ihre Stelle, etwa "create"
 - Zustandssteuerung per Klassenattribut
- **Beispiel:** Syslog (sollte es nur einmal geben)

```
class Syslog
  private_class_method :new
  @@log_hnd = nil

  def Syslog.create
    @@log_hnd = new unless @@log_hnd
    @@log_hnd
  end
  # ...
end
```

```
log1 = Syslog.create; log2 = Syslog.create # 2 Versuche
puts log1.id == log2.id # true (2 Ref. zum gleichen Objekt)
```



Das Entwurfsmuster "singleton"



- In Ruby eingebaute Realisierung von Singleton-Klassen:
 - Demonstriert am Beispiel Syslog:

```
require 'singleton'
```

```
class Syslog
  include Singleton
end
```

```
log1 = Syslog.instance
log2 = Syslog.instance    # 2 Versuche

puts log1.id == log2.id  # true
# 2 Referenzen zum gleichen Objekt!
```



Weitere Entwurfsmuster



- Ruby unterstützt zwei weitere, etwas komplexere Entwurfsmuster:
- **Delegate:**
 - Eine Alternative zur Mehrfachvererbung
 - Ein "delegator"-Objekt reicht Methoden, die es selbst nicht implementiert hat, weiter an assoziierte Objekte ("delegates").
 - Die "delegates" sind dabei keine Objekte aus einer Oberklasse des "delegators".
- **Observer** (auch: "publish/subscribe"):
 - Ein Mechanismus, mit dem ein Objekt eine Menge dritter Objekte über seine Zustandsänderungen informieren kann.
- Näheres dazu: Pickaxe book, ch. 25 (*OO design libraries*)



Stilfragen

The Ruby Way ...
(Sammlung, wächst noch)



Stilfragen



- Entschuldigen Sie, dass der Brief so lang geworden ist, ich hatte keine Zeit für einen kürzeren.

J. W. v. Goethe

- Vollkommenheit entsteht offenbar nicht dann, wenn man nichts mehr hinzuzufügen hat, sondern wenn man nichts mehr wegnehmen kann.

Antoine de St. Exupéry

- In der Kürze liegt die Würze.

(Volksmund)



Stilfragen



- Mache die Dinge so einfach wie möglich, aber nicht einfacher.
A. Einstein
- *When you say something in a small language, it comes out big. When you say something in a big language, it comes out small.*
L. Wall, über Perl
- *Don't write 200 lines of code when 10 will do.*
H. Fulton



Stilfragen



- *A smart servant should do a complex task with a few short commands.*
Y. Matsumoto ("Matz"),
über Computer und deren Sprachen
- *A program should follow the "Law of Least Astonishment".*
G. James, in
"The Tao of Programming"
- Die dümmersten Informatiker schreiben die dicksten Programme.
(Volksmund :-))



- *Ruby arose from the urge to create things that are useful and beautiful. Programs written in Ruby should spring from that same God-given source. That, to me, is the essence of the Ruby Way.*
H. Fulton, in "The Ruby Way"



Themenauswahl für Teil 3

Zur Diskussion
Vorschlag



Kursinhalte - Teil 3: Anwendungen



- Mehr zu Modulen, Namensräumen, Mixins
- **Extensions**
 - Das Zusammenspiel mit C/C++ Bibliotheken
 - Beispiel Win32, WinOLE
- Testen und Dokumentieren
 - RDoc und die Klasse "TestUnit"
- **Persistenz**
 - Objekte speichern & rekonstruieren; DBM; DBI-Grundlagen
- **GUI-Entwicklung mit Ruby**
 - FXRuby: Erste Schritte mit der portablen GUI-Bibliothek FOX
- Design patterns
 - Ruby's eingebaute Unterstützung:
(*Visitor, Singleton, Observer, Delegation*)
- **Ruby und das Internet**
 - cgi-Scripts, embedded Ruby
- **Distributed Ruby**



Termine für Teil 3 (Vorschlag)



- 24.11.
 - Abschluss von Kapitel 2 "Vertiefung"
- 1.12.
 - Persistenz; GUI (Anfang)
- 8.12.
 - GUI
- 15.12.
 - Extensions; Internet (I)
- 5.1.
 - Internet (II), Distributed Ruby