

```
#!/usr/bin/perl  
My $PerlVortrag = „V.1.0“;  
Print „Go!“;
```

# *Reguläre Ausdrücke in Perl*

```
#!/usr/bin/perl
My $PerlVortrag = „V.1.0“;
Print „Go!“;
```

# Reguläre Ausdrücke in Perl

- Reguläre Ausdrücke sind in der Computerwelt allgegenwärtig (Textverarbeitung, Programmierung, Suchmaschinen)
- Regulärer Ausdruck = Schablone die auf einen Text passt oder nicht.
- In Perl benutzt zur Stringbehandlung
- Ist wie eine eigene Programmiersprache
- Dient nur zur Feststellung ob ein String passt oder nicht

## Definition:

**Ein Regulärer Ausdruck (oft auch als Suchmuster bezeichnet) ist eine Schablone, die auf einen gegebenen String passt oder nicht.**

```
#!/usr/bin/perl
My $PerlVortrag = „V.1.0“;
Print „Go!“;
```

# Erstes Code Beispiel

## Quelltext:

```
$_ = „Gesellschaft“;
if (/esel/){
    print „passt“;
}
else {
    print „passt nicht“;
}
Ausgabe: passt
```

- Der Variablen \$\_ wird der Wert „Gesellschaft“ zugewiesen
- Die if – Abfrage bezieht sich auf \$\_
- In Abhängigkeit vom Ergebnis der Abfrage ändert dies die Ausgabe auf der Konsole
- Interessanter Teil steht in der Bedingung der If – Anweisung
- Regulärer Ausdruck steht umfasst in Schrägstriche (/.../)

```
#!/usr/bin/perl
My $PerlVortrag = „V.1.0“;
Print „Go!“;
```

# Metazeichen/Escape Zeichen

- Metazeichen in Perl ist der Punkt (.)
- Symbol für ein beliebiges Zeichen ohne das Newline Zeichen (\n)
- Um einen String mit Punkt z.B. „3.1415“ zu finden benötigt man ein Escape Zeichen
- Escape Zeichen in Perl ist der Backslash (\), er nimmt Sonderzeichen die Bedeutung innerhalb eines Ausdrucks

## Beispiel:

Ausdruck /.est/

findet „Test“, „Fest“, „Pest“ usw.

Ausdruck /3\.1415/

findet „3.1415“

```
#!/usr/bin/perl
My $PerlVortrag = „V.1.0“;
Print „Go!“;
```

# Allgemeine Quantifier

- Benötigt, um eine bestimmte Anzahl  $X$  an Vorkommen eines Teilsuchstrings  $Y$  zu finden
- Haben allgemein die Form  
 $TeilsuchstringY\{untereGrenzeX_{min}, obereGrenzeX_{max}\}$
- Suche nach **mindestens**  $X_{min}$  Vorkommen in  $Y$  hat die Form  
 $/Y\{X_{min},\}/$
- Suche nach **genau**  $X$  Vorkommen von Teilsuchstring  $Y$  hat die Form  
 $/Y\{X\}/$

## Beispiel:

Ausdruck `/x{2,4}/`  
findet „xx“, „xxx“, „xxxx“  
aber auch „xxxxx“!)

Ausdruck `/x{2,}/`  
findet „xx“, „xxx“, „xxxx“...

Ausdruck `/x{2}/`  
findet „xx“

Ausdruck `/yx{2,3}/`  
findet „yxx“, „yxxx“

→ Quantifier bezieht sich nur auf vorangehendes Zeichen

```
#!/usr/bin/perl
My $PerlVortrag = „V.1.0“;
Print „Go!“;
```

# Einfache Quantifier

- Perl war ursprünglich Tool zur Vereinfachung von Arbeitsgängen für Systemadministratoren → Viele Abkürzungen für häufig verwendete Arbeitsgänge
- Einfache Quantifier sind Abkürzungen für Allgemeine Quantifier
- Sternchen(\*), Pluszeichen(+) und Fragezeichen (?) beziehen sich immer auf das vorhergehende Symbol (od. Symbolgruppe s. Mustergruppierung)

## Definition:

Das Sternchen (\*) steht für das Vorkommen von **keinem oder mehr** Zeichen und ist die Abkürzung für `/x{0,}/`.

Das Pluszeichen (+) steht für das Vorkommen von **mindestens einem** Zeichen und ist die Abkürzung für `/x{1,}/`.

Das Fragezeichen (?) steht für „**einmal oder keinmal**“ auftreten eines Zeichens und ist die Abkürzung für `/x{0,1}/`.

```
#!/usr/bin/perl
My $PerlVortrag = "V.1.0";
Print "Go!";
```

# Beispiele für Quantifier

Suchmuster	Suchstring	Ergebnis
/hal{2}oe\.e/	halloele	true
/ja*\.* / gefunden	Jaaa.....gefunden	true
/co+l/	cl	false
/co?l/	cl	true
/(Hallo)+\.{3}/	Hallo...	true

```
#!/usr/bin/perl
My $PerlVortrag = „V.1.0“;
Print „Go!“;
```

# Muster gruppieren

- Gleich dem Klammerungs Ausdruck in der Mathematik
- Dient zur Gruppierung von Teilsuchstrings
- Runde Klammern [()] werden zur Gruppierung verwendet
- Wird in Verbindung mit Quantifiern verwendet
- Wird außerdem zur Speicherung von „Zwischenergebnissen“ verwendet (später mehr)

## **Beispiel:**

Ausdruck /Hallo+/  
findet „Hallooooo“

Ausdruck /(Hallo)+/  
findet „HalloHalloHallo“

Ausdruck /((\?)(ok\?))?/  
findet „\ok?“, „ok?“, „“

→ Klammerung um beliebige Ausdrücke möglich

```
#!/usr/bin/perl
My $PerlVortrag = „V.1.0“;
Print „Go!“;
```

# Alternativen

- Werden verwendet, um eine Alternative zwischen zwei gleichwertigen Teilsuchstrings darzustellen.
- Entspricht dem logischen Oder
- Symbol ist die Pipe (|)
- Anders als Quantifier bezieht es sich nicht nur auf das vorhergehende Symbol, sondern auf alle vorhergehenden und nachfolgenden Wortzeichen (Präzedenz)

## Beispiel:

Ausdruck /Hallo|Ciao/

findet den Ausdruck „Hallo“ oder „Ciao“

## Achtung!

Ausdruck /Hallo|Ciao/

findet **nicht** „HallCiao“ oder „Halloiao“, die Alternative bezieht sich auf alle Wortzeichen vor und hinter dem Operator

```
#!/usr/bin/perl
My $PerlVortrag = „V.1.0“;
Print „Go!“;
```

# Fragen

1. Wie würde ein Muster aussehen müssen, welches eine beliebige Anzahl von Backslashes (\) gefolgt von einer beliebigen Anzahl von Sternchen (\*) finden soll?

`/\\*\\**/`

2. Wie würde ein Muster aussehen, welches genau zwei a gefolgt von mindestens zwei b finden soll?

`/aab+/, /a{2}b+/`

3. Welcher Teil des Regulären Ausdrucks `/h|hu|hun|hund/` trifft auf den String `hund` zu?

`h`, denn es wird von links nach rechts mit dem Regulären Ausdruck verglichen.

4. Welcher Teil des Regulären Ausdrucks `/hund|hun|hu|h/` trifft auf den String `„hund“` zu?

`hund`, denn es wird von links nach rechts mit dem Regulären Ausdruck verglichen.

```
#!/usr/bin/perl
My $PerlVortrag = "V.1.0";
Print "Go!";
```

# Zeichenklassen

- Eine Menge von Zeichen eingeschlossen in eckige Klammern ([ ]) steht für **ein** beliebiges Zeichen aus dieser Menge
- Zusammenhängende Bereiche werden mit einem Bindestrich (-) verkürzt  
/ABCDEF/ ⇔ /A-F/ , /0123456789/ ⇔ /0-9/
- Steht ein Caret (^) Symbol zu Beginn der Klasse, wird diese negiert. (Ein beliebiges Zeichen, aber keines aus der Menge)
- Sollen Caret (^) und Bindestrich (-) Teil der Menge sein verwendet man das Escape - Zeichen (\)

## Beispiel:

```
$_ "Der Jahr 2000 Bug"; //zu durchsuchender String
if (/Jahr [0-9]+ Bug| Fehler/) //Suchmuster passt auch auf Der Jahr 2000 Fehler
{
    print "passt";
}
```

Ausgabe: Passt

```
#!/usr/bin/perl
My $PerlVortrag = „V.1.0“;
Print „Go!“
```

# Abkürzungen in Zeichenklassen

- Manche Zeichenklasse werden so häufig verwendet, dass sie noch weiter verkürzt werden.
- Zeichenklasse [a-zA-Z0-9\_] wird als (**\w**) abgekürzt → Wortzeichen
- Zeichenklasse [0-9] wird als (**\d**) abgekürzt → Dezimalzeichen
- Zeichenklasse [\f\t\n\r ] wird als (**\s**) abgekürzt → Whitespace Zeichen
- Werden häufig in Verbindung mit Quantifiern (\*,+,?) verwendet

## Erklärung:

- *\f* steht für den Formularvorschub
- *\b* steht für den Tabulator
- *\n* steht für Neue Zeile
- *\r* steht für Wagenrücklauf

## Beispiel:

`/\s*/` prüft auf Vorkommen von Whitespace-Zeichen

Ausdruck `/\w+\s+\w+/`

findet „Hallo Ciao“ (Zwei Wörter getrennt durch ein Leerzeichen)

```
#!/usr/bin/perl
My $PerlVortrag = „V.1.0“;
Print „Go!“;
```

# Negieren von Zeichenklassen

- Zeichenklassen können auch negiert werden
- Hierzu kann das Caret (^) verwendet werden `/[^d]/` ⇔ alles außer Ziffern
- Im Allgemeinen verwendet man die entsprechenden Kapitale
- `\D` steht für ein Zeichen außer den Dezimalzeichen `[0-9]`
- `\W` steht für ein Zeichen außer den Wortzeichen `[a-zA-Z0-9_]`
- `\S` steht für ein Zeichen außer den Whitespace - Zeichen `[\f\t\n\r]`

## Erklärung:

1. `\d` steht für eine Ziffer
2. `\D` steht für ein Zeichen außer Ziffern
3. `\w` steht für ein Wort-Zeichen
4. `\W` steht für ein Zeichen außer Wort-Zeichen
5. `\s` steht für ein Whitespace-Zeichen
6. `\S` steht für Zeichen außer Whitespace-Zeichen

## Beispiel:

```
$_ = „Hallo-Hallo“;           //zu
durchsuchender String
if (/\w+[^s]\w+/)           //Suchmuster
    passt auf den gewünschten String
    {print „passt“;}
Ausgabe: Passt
```

```
#!/usr/bin/perl
My $PerlVortrag = „V.1.0“;
Print „Go!“;
```

# Fragen

1. Was kann man mittels der Zeichenklasse `/[\dA-Fa-f]+/` finden?

Hexadezimalzahlen

Worauf passt die Klasse `[\d\D]`?

Auf eine Ziffer oder eine Nichtziffer einschließlich des Newline Zeichens(`\n`)!

3. Wenn die Zeichenklasse `[\d\D]` auf eine Ziffer oder eine Nichtziffer passt, worauf passt dann `[^\d\D]`?

Passt auf alles das keine Zahl oder keine Nichtzahl ist – auf nichts.

4. Passt `(19|20|)\d\d\s?` auf den String „20“ und wenn ja warum?

Ja, als Alternative wird „“ ausgewählt und das Whitespace Zeichen kommt keinmal vor!

```
#!/usr/bin/perl
My $PerlVortrag = „V.1.0“;
Print „Go!“;
```

# Muster verankern

- Suchmuster werden immer im gesamten String gesucht. Möchte man das Suchmuster aber definitiv am Anfang oder am Ende stehen haben benötigt man Anker (^)(&)
- Caret (^) findet das Suchmuster wenn es am Anfang des Strings steht
- Kaufmannsund (&) findet das Suchmuster wenn es am Ende des Strings steht
- Anker können kombiniert werden
- Häufig verwendet um ganze Strings zu finden

## Beispiel:

```
Suchmuster m#^http://#
```

```
findet „http://www.www.de“
```

```
findet nicht „Seite http://www.www.de“
```

```
Suchmuster m#www.de&#
```

```
findet „http://www.www.de“
```

```
findet nicht „http://www.www.de/index.html“
```

```
Suchmuster m{^http://&}
```

```
findet „http://“
```

```
Suchmuster m{^\s*&}
```

```
findet leere Strings
```

```
#!/usr/bin/perl
My $PerlVortrag = "V.1.0";
Print "Go!";
```

# Wörter verankern

- Um innerhalb eines Suchstrings nach Wortgrenzen zu suchen verwendet man das Wordend Zeichen (`\b`), welches hier hört das Wort auf bedeutet
- Ein Wortende ist definiert als ein Zeichen `\W` (`[^\a-zA-z0-9_]`)
- Das Gegenteil des Wordend Zeichens `\B` bedeutet hier geht das Wort noch weiter
- Kann mit Musterankern kombiniert werden

## Beispiel:

```
Suchmuster /\bhallo\b/
findet „-hallo-“
findet nicht „ halloele“
```

```
Suchmuster /^tach\B/
findet „tachsche“
findet nicht „tach-“
```

```
Suchmuster /^hallo\b\s\&/
findet „hallo\t\t\n“
da & auch Newline Zeichen impliziert
```

```
#!/usr/bin/perl
My $PerlVortrag = "V.1.0";
print "Go
```

# Rückwärtsreferenz/Speichervariablen

- Wird bei einem Mustervergleich verlangt den soeben gefundenen Teilsuchstring noch einmal zu finden, so ist dies nicht einfach
- Eine Rückwärtsreferenz bezieht sich auf einen Speicher, der zu einem früheren Zeitpunkt der Mustererkennung gefüllt wurde.
- Hierfür besitzen die runden Klammern `()` die nötige Funktionalität, sie weisen die regexp Maschine an, den in ihnen gruppierten Teilsuchstring zu speichern
- Zugriff auf den gespeicherten Wert innerhalb der Mustererkennung erfolgt über `\n`
- `\n` bezieht sich auf die n-te öffnende Klammer innerhalb des regulären Ausdrucks
- Die n Rückwärtsreferenzen stehen auch unmittelbar nach Abschluss der Mustererkennung in Speichervariablen zur Verfügung
- Diese Variablen lassen sich über `$n` ansprechen

```
#!/usr/bin/perl
My $PerlVortrag = „V.1.0“;
print „Go“;
```

# Rückwärtsreferenz/Speichervariablen

## Beispiel:

```
/((Hallo|Ciao) (Welt)) \1/
    findet „Hallo Welt Hallo Welt“ oder aber „Ciao Welt Ciao Welt“

/((Hallo|Ciao) (Welt)) \2/
    findet „Hallo Welt Hallo“ oder „Ciao Welt Ciao“

/((Hallo|Ciao) (Welt)) \3/
    findet „Hallo Welt Welt“ oder „Ciao Welt Welt“
```

```
#!/usr/bin/perl -w

my $test = "\tHallo\tHallo";           #Stringzuweisung
if ($test =~ /(\s+)(\w+)\1(\2)/)      #Suchenach einem Leerzeichen, gefolgt
{                                       #von einem Wort und dann genau das gleiche
    #nochmal
    print "Die gefundenen Wörter lauten $1 $2 $3";
}
```

Ausgabe: Die gefundenen Wörter lauten \t Hallo Hallo

```
#!/usr/bin/perl
My $PerlVortrag = „V.1.0“;
Print „Go!“
```

# Automatische Speichervariablen

- Zusätzlich zu den `\n` Rückwärtsreferenzen und `$n` Speichervariablen legt Perl noch andere drei automatische Speichervariablen an, die im weiteren Programmverlauf nutzbar sind
- Die Variable `$`` (Backtick, nicht Hochkomma) speichert den Teil des ursprünglich zu durchsuchenden Strings, der vor dem durch das Muster gefundenen Teilstring steht.
- Die Variable `$&` enthält den Teilstring, der durch das Suchmuster identifiziert wurde oder, falls dieses nicht gefunden wurde, den Wert `undef`.
- Die Variable `$'` (Backtick, nicht Hochkomma) speichert den Teil des ursprünglich zu durchsuchenden Strings, der hinter dem durch das Muster gefundenen Teilstring steht.

```
#!/usr/bin/perl
My $PerlVortrag = „V.1.0“;
Print „Go!“;
```

# Codebeispiel Speichervariablen

## Beispiel:

```
1)#!/usr/bin/perl -w
2)while (<>)
3)    {
4)        chomp;
5)        if (/Testsuchmuster/)
6)            {
7)                print „Treffer: |$`<$&>$´|\n“;
8)            }
9)        else
10)           {
11)               print „Kein Treffer\n“;
12)           }
13)    }
```

Dieses Programm dient dem Testen von Mustern. es liest die Eingaben des Benutzers von der Konsole ein und führt einen Mustervergleich mit dem in Zeile 5 angegebenen regulären Ausdruck. Interessant ist hierbei nur Zeile 7. In dieser Zeile wird das gefundene Suchmuster in der Form „StringDavor<Suchmuster>StringDanach“ angezeigt.

```
#!/usr/bin/perl
My $PerlVortrag = „V.1.0“;
Print „Go!“;
```

# Präzedenz

1. Runde Klammern `[]`, die zum gruppieren und speichern von Teilen eines Musters verwendet werden, halten am stärksten zusammen.
2. Quantifier `*`, `+`, `?`, `{x,4}` hängen immer mit dem Element zusammen auf das sie folgen
3. Die Ankerzeichen `^`, `$`, `\b`, `\B` und Zeichenfolgen `.` Die Buchstaben halten genauso eng zusammen wie die Ankerzeichen mit denen sie in Verbindung stehen.
4. Auf der niedrigsten Stufe steht die Alternative `|`. Dies ist wichtig um Suchstrings der Form `/Hallo|Tach/` auch wirklich auf die String „Hallo“ oder „Tach“ zutreffen zu lassen. Wäre die Präzedenz der Alternative höher ließen sich mit dem obigen Beispiel die unsinnigen Strings „Halloach“ oder „HallTach“ finden.

```
#!/usr/bin/perl
My $PerlVortrag = „V.1.0“;
Print „Go!“;
```

# ***Mustervergleich mit m//***

- Es ist nicht zwingend notwendig einen Mustervergleich mit /.../ durchzuführen, es ergeben sich sogar manchmal falsche Ergebnisse
- Der Mustervergleich m// hilft hier. Schreibt man ein einleitendes m können die begrenzenden Schrägstriche durch andere Zeichen ersetzt werden
- Möglich für die Verwendung des m – „Operators“ sind jegliche Zeichen, die keinen Unterscheid zwischen der linken und der rechten Entsprechungen aufweisen (m!...!, m%...% usw.) genauso wie jegliche Art von Klammern ({...}, [...], (...))

## **Beispiel:**

```
Suchmuster /http:///
Findet „http:“, aber nicht das gewünschte „http://“
besser wäre die Verwendung des m-Operators
```

```
Suchmuster m#http://#
Findet „http://“
```

```
#!/usr/bin/perl
My $PerlVortrag = „V.1.0“;
Print „Go!“;
```

# Bindungsoperator =~

- Der Bindungsoperator (=`~`) weist perl an, einen Mustervergleich mit dem String rechts des Operators durchzuführen.

```
my $test = ‚Ein kleiner Test‘;
if ($test =~ /einer/)
{print „passt“;}
```

- Ein besonders häufig verwendetes Beispiel für den Zuweisungsoperator (=`~`) findet sich hier.

```
my $wert = „Abfragewert“;
$abfrage = (<STDIN> =~ /\b$wert\b/);
```

- Der Variablen `$abfrage` wird hierbei ein Boolescher Wert zugewiesen, die eigentliche Benutzereingabe wieder verworfen. Auf diese kommt es in diesem Falle auch gar nicht an, es geht darum, ob `^b$wert\b/`is Teil der Benutzereingabe `<STDIN>` ist.

```
#!/usr/bin/perl
My $PerlVortrag = "V.1.0";
Print "Go!";
```

# Flag- und Option Modifier

- Die Simulierung des Ignorierens der Groß- und Kleinschreibung unter Perl ist recht aufwendig. Um das Wort „Yes“ nicht case-sensitiv suchen zu lassen bräuchte man folgendes Muster `/[Yy][Ee][Ss]/`
- Einfacher geht dies mit dem `/i` Modifier, welcher wie alle Modifier hinter das Abgrenzungszeichen geschrieben wird
- Der `/s` Modifier verändert die Eigenschaften des Metazeichens Punkt (`.`) dahingehend, dass dieses nun auch für ein Newline Zeichen stehen kann und somit der Zeichenklasse `[\d\D]` gleicht
- Mehrere Modifier können kombiniert werden.

## Beispiel:

```
Suchmuster /hallo.hallo/is
Findet „ HaLlO\nhaLlO“
```

```
Suchmuster m{(hallo).\1}si
Findet „HALLo\nhallo“
```

```
my $suchmuster = „Hallo“;
$abfrage = (<STDIN> =~ /\b$suchmuster\b/is);
```

```
#!/usr/bin/perl
My $PerlVortrag = „V.1.0“;
Print „Go!“;
```

# Variableninterpolation

- Im vorhergehenden Beispiel wurde bewusst `$abfrage = (<STDIN> =~ /\b$suchmuster\b/si);` geschrieben.
- Variablen können in Suchmustern interpoliert werden ⇔ durch Ihre tatsächlichen Inhalt ersetzt werden
- Funktionsweise genau wie bei Strings, denn Strings in einfachen Quotas führen keine Variableninterpolation durch, Strings in doppelten Quotas hingegen interpolieren Variablen
- Damit Wortzeichen nach `$` nicht unsinnig als skalar Variable interpretiert werden verwendet man das Escape – Zeichen (`\`)

## Beispiel:

```
my $string1 = „a“*3           #$string1=aaa
$suchmuster = („Hallo“.$string1)*2;  #suchmuster = HalloaaaHalloaaa

$abfrage = (<STDIN> =~ /\b$suchmuster\b/is);
{...}
```

```
#!/usr/bin/perl
My $PerlVortrag = „V.1.0“;
print „Go!“;
```

## ***Ersetzen mittels s///***

- Die RegExp Maschine unter Perl kann zur String Verarbeitung benutzt werden und die Funktion „Suchen und Ersetzen“ mittels s/// anwenden.
- Syntax *s/\$VariableMitSuchmuster/Ersetzungstring/*
- Der erste Teil muss eine Variable sein, denn Perl kann nur etwas verändern das in „etwas“ enthalten ist.
- Per Standard ersetzt Perl immer nur das erste Vorkommen des Suchmusters, um dies zu ändern gibt es wiederum Modifier
- Der /g Modifier ersetzt alle Vorkommen des Musters durch den Ersetzungstring, die Modifier zum Ignorieren von Groß- und Kleinschreibungen (/i) sowie die Erweiterung des Metazeichens Punkt (.) (/s) sind ebenfalls verfügbar.
- Soll ein Wort komplett in Großbuchstaben oder Kleinbuchstaben umgewandelt werden, so verwendet man die \U (uppercase) und \L (lowercase) Modifier, bei einzelnen Buchstaben die \u und \l Modifier, die nur auf das nachfolgende Zeichen wirken.
- Um die Wirkung eines \U oder \L Modifiers frühzeitig zu beenden verwendet man den \E Modifier am Ende des gewünschten Wortes

```
#!/usr/bin/perl
My $PerlVortrag = „V.1.0“;
Print „Go!“;
```

# Codebeispiel s///

## Beispiel:

```
my $test = „Hallo du“;
$test =~ s/du/sie/;           #ersetze du durch Sie
print $test;
```

Ausgabe: Hallo sie

```
my $pfad = `www.www.de/www/`; #Benutzt Single Quotas,muss nicht interpolieren
$test = „http://$pfad“;       #Erstellt String
$test =~ s#http://##;        #lösche http:// aus URL
print $test;
```

Ausgabe: www.www.de/www/

```
my $test = „hALLO Welt“;
$test =~ s/(Hallo|Ciao)/\u\L$1/ig; #Schreibe den ersten Buchstaben groß, den Rest
                                     #klein, ersetze dabei alle Vorkommen und
                                     #ignoriere die Groß- und Kleinschreibung im
                                     #Suchmuster
```

Ausgabe: Hallo Welt

**Achtung:** Hierbei wird eine Rückwärtsreferenz verwendet

```
#!/usr/bin/perl
My $PerlVortrag = „V.1.0“;
Print „Go!“;
```

# Der Split Operator

- Der Split-Operator benutzt reguläre Ausdrücke, um einen String anhand eines vorher gewählten Trennzeichens in ein Array aufzuspalten. Die Syntax für diesen Operator lautet  
`@feld = split /Trennzeichen/ $string`
- Oftmals wird `split` verwendet, um einen Satz an seinen Whitespaces aufzusplitten und die Wörter in einem Array aufzufangen.
- **Achtung!**  
Folgen zwei der Trennsymbole direkt nacheinander so wird ein leerer Feldeintrag erzeugt.

## Beispiel:

```
@feld = split /:/, „abc:def:ghi:jkl“;
ergibt: ("abc","def","ghi","jkl")

@feld = split /:/:, „abc::ghi::jkl“;
ergibt: ("abc","","ghi","","jkl")

@feld = split /\s+/, „Hallo, dies ist ein Test“;
ergibt: ("Hallo,","dies","ist","ein","Test")
```

```
#!/usr/bin/perl
My $PerlVortrag = "V.1.0";
Print "Go!";
```

# Andere Modifier

- Perl kennt noch eine ganze Menge anderer Modifier
- Der für Leser des Quellcodes wichtigste ist der /x Modifier
- /x ignoriert Leerzeichen, Kommentare und Zeilenumbrüche in regulären Ausdrücken, so dass diese lesbarer geschrieben werden können.

s/(Hallo|Ciao)/\u\L\$1/ig wird mittels /x Modifier zu

```
s{
    (Hallo
     |
    Ciao)
}{
    \u\L
    $1
}ig
#ersetze
#das Wort Hallo
#Oder
#Das Wort Ciao
#schreibe den ersten Buchstaben groß, die anderen klein
#ersetze mit der Rückwärtsreferenzl
#Ersetze alles und ignoriere Groß- und Kleinschreibung
```

## Praktisches Beispiel:

Codebeispiel: Entfernen von C Kommentaren aus einer Datei

Suchmaske:

```
$/ = undef; $_ = <>;
```

```
s#/\[^\*][^\*]*\#+([\^/\*][^\*]*\#+)*/|("(\.|\.[^\"]*)"|'(\.|\.[^']*')'|.[^/"'\.\\]*)#$2#gs
print;
```

```
#!/usr/bin/perl
My $PerlVortrag = "V.1.0";
Print "Go!";
```

# Codebeispiel /x Modifier

## Beispiel:

```
s{
    /\*          ## Start of /* ... */ comment
    [^*]*\*+    ## Non-* followed by 1-or-more *'s
    (
        [^/*][^*]*\*+
    )*          ## 0-or-more things which don't start with /
              ## but do end with '*'
    /          ## End of /* ... */ comment

|           ## OR various things which aren't comments:

    (
        "        ## Start of " ... " string
        (
            \\.    ## Escaped char
        )
    )
|           ## OR
```

```
#!/usr/bin/perl
My $PerlVortrag = "V.1.0";
Print "Go!";
```

# Codebeispiel /x Modifier

## Beispiel:

```
[^"\\]      ## Non "\
)*)*
"          ## End of " ... " string
|         ## OR
'          ## Start of ' ... ' string
(
  \\.      ## Escaped char
|         ## OR
  [^'\\]   ## Non '\
)*)*
'          ## End of ' ... ' string
|         ## OR
.          ## Anything other char
[^/"'\\]* ## Chars which doesn't start a comment, string or escape
)
}{{$2}gxs;
```