

HOWTO zu den Perl-Klammerarten

(C) 2006 T.Birnthaler/H.Gottschalk <howtos(at)ostc.de>

OSTC GmbH, <http://www.ostc.de>

\$Id: perl-parentheses-HOWTO.txt,v 1.8 2011-04-19 15:03:59 tsbirn Exp \$

Dieses Dokument beschreibt die 4 verschiedenen Klammerungs-Arten in Perl und ihre Anwendungsgebiete bzw. Bedeutungen.

Inhaltsverzeichnis

- 1) Einführung
- 2) Klammerungs-Arten
 - 2.1) (...) - parentheses
 - 2.2) {...} - braces
 - 2.3) [...] - brackets
 - 2.4) <...> - angle brackets
- 3) "Klammern" oder "Nicht Klammern" in Perl?

1) Einführung

Die Syntax von Perl ist nicht gerade als "einfach" zu bezeichnen, obwohl sie von ihrem Erfinder Larry Wall ganz bewusst so festgelegt wurde und keineswegs "unlogisch" ist. Insbesondere die vielen Sonderzeichen und vor allem die Klammer-Arten sind schwer zu verstehen. Hiermit soll anhand von Anwendungs-Beispielen der Einsatz und die Bedeutung der 4 verschiedenen Klammerungs-Arten in Perl erklärt werden.

2) Klammerungs-Arten

2.1) (...) - parentheses

Anwendungsgebiete von (...): Array, Liste, Hash, Vorrang ändern, Bedingung/Liste bei if/while/for/foreach/..., Parameter eines Subroutinen-Aufrufs, Regex-Klammerung bzw. Merken des Matches eines Regex-Teilausdrucks.

```
my @ostc = ( 'Tom', 'Hans', 'Rick' );           # Array definieren

($a, $b, $c) = ( 'Tom', 'Hans', 'Rick' );     # Liste
( 'Tom', 'Hans', 'Rick' );                   # Liste

my %pers = ( size => 1.82, weight => 0.1, ... ); # Hash definieren

$erg = ($a >= 50 and $a <= 100);              # Vorrang ändern
$erg = $a ** ($b * ($c + 1));                 # Vorrang ändern

if ($a eq 100) {...}                          # Bedingung
while (<>) {...}                               # Bedingung
until ($a <= 0) {...}                         # Bedingung
for ($i = 0; $i < 100; ++$i) {...}            # Liste
foreach (@liste) {...}                       # Liste

&xyz("Hans", 20);                             # Aufruf-Parameter

$zeile =~ /^(tom|rick|hans)$/;                 # Regex klammern
$zeile =~ /^(\d+)\s+(\d+)$/;                  # Regex merken
-> $1 $2                                       # autom. gefüllt!
```

2.2) {...} - braces

Anwendungsgebiete von {...}: Hash-Element-Zugriff, (anonymen) Hash definieren, Variablen-Name klammern, Block, (anonyme) Subroutine, Regex-Wiederholungsfaktor.

```
my %pers1 = ( size => 1.82, weight => 0.001, ... ); # Hash definieren
my $pers2 = { size => 1.82, weight => 0.001, ... }; # Anonymer Hash

my $size = $pers1{size};                       # Hash-Element-Zugr.
my $size = $pers2->{size};                     # Hash-Element-Zugr.

print "Gewicht: $pers1{weight}\n";             # Hash-Element-Zugr.
print "Gewicht: $pers1{'weight'}\n";          # Hash-Element-Zugr.
print "Gewicht: $pers2->{weight}\n";          # Hash-Element-Zugr.
print "Gewicht: $pers2->{'weight'}\n";         # Hash-Element-Zugr.
```

```

print "Der Wert ist ${wert}Euro\n";           # Variablen-Name kl.
${erg} = ${a} * ${b};                         # Variablen-Name kl.
%{erg} = %{a};                                 # Variablen-Name kl.
@{erg} = @{a};                                 # Variablen-Name kl.

foreach my $i (1 .. 10) { print "i ist $i\n"; } # Block

my $proz = sub { print "Hallo hier bin ich\n"; } # Anonyme Subroutine

sub test                                       # Subroutinen-Block
{
    print "Hallo\n";
}

if ($ip =~ /^d{1,3}\.d{1,3}\.d{1,3}\.d{1,3}$/) # Regex-Quantifizierer
# FALL A:   a* === a{0,}
# FALL B:   a+ === a{1,}
# FALL C:   a? === a{0,1}

eval { ... };                                 # Eval-Block (Strichpunkt!)

```

2.3) [...] - brackets

Anwendungsgebiete von [...]: Array-Element-Zugriff, anonymes Array, Regex-Zeichenmenge.

```

my @text = qw( abc def ghi );                 # Array definieren

my $inhalt = $text[1];                        # Array-Element

my $aref = [ 'Tom', 'Hans', 'Rick' ];        # Anonymes Array

print "Mitglied 1 ist @{$aref}[0]\n";        # Array-Element
print "Mitglied 2 ist @{$aref}[1]\n";        #
print "Mitglied 3 ist $aref->[2]\n";

if ($xy =~ /^[A-Z][a-z][0-9].*/ ) { ... }    # Regex-Zeichenmenge
if ($xy =~ /^[ABCDEFGHJKLMNPQRSTUVWXYZ]/ ) { ... } # Regex-Zeichenmenge

```

2.4) <...> - angle brackets

Anwendungsgebiete von <...>: HTML-Tags kennzeichnen, Vergleiche, Datei einlesen, STDIN lesen, File-Globbering.

```

print "<head>";                               # HTML-Tag

if ($a < $b) { ... }                          # Vergleich
if ($a > $b) { ... }                          # Vergleich

open(INP, "< datei.txt");                       # Einlesen (A)
open(INP, "<", datei.txt);                     # Einlesen (B)
@zeilen = <INP>;                              # Einlesen (A)
@zeilen = readline(INP);                      # Einlesen (B)
close(INP);

while (<STDIN>) { ... }                       # Einlesen (STDIN)
while (<>) { ... }                            # Diamond-Op.

foreach (</etc/*/*/*>) { ... }               # Globbing (A) (readdir)
foreach (glob "</etc/*/*/*>") { ... }        # Globbing (B) (readdir)

```

3) "Klammern" oder "Nicht Klammern" in Perl?

Perl erlaubt an vielen Stellen sowohl eine "Funktions"-Schreibweise (mit Klammern um die Argumente), z.B.:

```
@result = sort(@array);
```

als auch eine "Operator"-Schreibweise (ohne Klammern um die Argumente), z.B.:

```
@result = sort @array;
```

In ganz wenigen Fällen sind Klammern notwendig, um den (falschen) Vorrang von Operatoren zu vermeiden oder den Aufruf einer (noch nicht definierten) eigenen Funktion anzudeuten.

Die Klammern sind etwas mehr zu tippen, tragen aber zum besseren Verständnis bei. Man sollte sich für eine Art der Schreibweise entscheiden und diese dann durchgängig einsetzen. Hier noch einige Beispiele für die Schreibweisen mit

und ohne Klammerung:

```
# Beides geht (was ist besser?, keine Ahnung!)
@arr = split / /, "abc def ghi";
@arr = split(/ /, "abc def ghi");

open FILE, "<", datei.txt";
open(FILE, "<", datei.txt");

if ((($a) lt ($b)) or (($c) gt ($b)))      # etwas zuviel geklammert ;-(
if ($a lt $b or                            # Umbruch macht manches klarer
    $c gt $b)

# "print" und "printf" über Klammern unterscheiden
print "Dies ist Text\n";
printf("Dies ist Text mit Zahl %d\n", $zahl);

# Umgekehrt geht es natürlich auch ;- )
print("Dies ist Text\n");
printf "Dies ist Text mit Zahl %d\n", $zahl;
```