HOWTO zu den Perl-Klammerarten

(C) 2006-2023 T.Birnthaler/H.Gottschalk <howtos(at)ostc.de>
OSTC Open Source Training and Consulting GmbH
http://www.ostc.de

$Id: perl-parentheses-HOWTO.txt,v 1.16 2025/02/23 20:14:55 tsbirn Exp $

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

INHALTSVERZEICHNIS

1) Einführung
2) Klammerungs-Arten
2.1) Runde Klammern (...) --- parentheses
2.2) Geschweifte Klammern {...} --- braces
2.3) Eckige Klammern [...] --- brackets
2.4) Spitze Klammern <...> --- angle brackets
2.5) String, Regex, Liste, Parameter spezieller Operatoren
3) "Klammern" oder "Nicht Klammern" in Perl?
4) Formatierung

---

1) Einführung

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

2) Klammerungs-Arten

2.1) Runde Klammern (...) --- parentheses

Anwendungsgebiete dieser Klammerart sind:

  * Liste/Array/Hash-Definition
  * Auswertungs-Vorrang in Ausdruck ändern
  * Bedingung von if/while/until
  * Liste von foreach
  * Init/Bedingung/Inkrement bei for
  * Parameter eines Subroutinen-Aufrufs
  * Regex: Vorrang in Regex ändern
  * Regex: Merken von Teilmatches

Beispiele:

```
  ($a, $b, $c) = ( 'Tom', 'Hans', 'Rick' )            # Liste
  ( 'Tom', 'Hans', 'Rick' )                           # Liste
  ()                                                  # Leere Liste
  my @ostc = ( 'Tom', 'Hans', 'Rick' )                # Array definieren
  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 ($a <= 0) {...}                                 # Bedingung
  foreach (@liste) {...}                                # Liste
  for ($i = 0; $i < 100; ++$i) {...}                    # Liste

  &xyz("Hans", 20)                                      # Subroutine-Aufruf
  &xyz()                                                # Analog (keine Param.)

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

2.2) Geschweifte Klammern {...} --- braces

Anwendungsgebiete dieser Klammerart sind:

* Anonymen Hash definieren (Referenz)
* Hash-Element-Zugriff
* Variablen-Name klammern
* Anweisungs-Block (nach if, else, for, while, until)
* Eingeschachtelter Block (Gültigkeitsbereich)
* Anonyme Subroutine (Referenz)
* Subroutinen-Block
* eval-Block
* Regex: Wiederholungsfaktor n-m

Beispiele:

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

  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.

  if (...) { print "hier bin ich\n"; ... }              # Anweisungs-Block
  while (...) { print "hier bin ich\n"; ... }           # Anweisungs-Block
  until (...) { print "hier bin ich\n"; ... }           # Anweisungs-Block
  for (...; ...; ...) { print "hier bin ich\n"; ... }   # Anweisungs-Block
  foreach my $i (1 .. 10) { print "i ist $i\n"; ... }   # Anweisungs-Block

  {                                                     # Block aussen
      ...                                               # Gültigkeitsbereich 1
      {                                                 # Block innen
          ...                                           # Gültigkeitsbereich 2
      }                                                 # Block innen Ende
      ...                                               # Gültigkeitsbereich 1
  }                                                     # Block innen Ende

  my $proz = sub { print "Hallo hier bin ich\n" }       # Anonyme Subroutine (Ref)
  sub test                                              # Subroutinen-Block
  {
      print "Hallo\n"
  }

  eval { ... }                                          # Eval-Block (" " nötig!)
```

```
  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}
```

2.3) Eckige Klammern [...] --- brackets

Anwendungsgebiete dieser Klammerart sind:

* Anonymes Array definieren (Referenz)
* Array-Element-Zugriff
* Array-Slicing
* Regex: Zeichenmenge definieren

Beispiele:

```
  my $aref = [ 'Tom', 'Hans', 'Rick' ]                 # Anonymes Array (Ref)
  my @text = qw( abc def ghi )                         # Array definieren!

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

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

  @arr[0, -1] = @arr[1, 0]                             # Array-Slicing

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

2.4) Spitze Klammern <...> --- angle brackets

Anwendungsgebiete dieser Klammerart:

* HTML-Tags in Strings
* Vergleiche
* Bitoperation Shift
* Datei lesen/schreiben/anhängen
* STDIN lesen
* File-Globbing

Beispiele:

```
  print "<head>Titel</head>"                  # HTML-Tag

  if ($a < $b) { ... }                       # Vergleich "kleiner"
  if ($a <= $b) { ... }                      # Vergleich "kleiner gleich"
  if ($a > $b) { ... }                       # Vergleich "größer"
  if ($a >= $b) { ... }                      # Vergleich "größer gleich"

  0b01010101 << 4                            # Shift-left
  0b01010101 >> 4                            # Shift-right

  open(INP, "< datei.txt")                   # Einlesen (A)
  open(INP, "<", datei.txt)                  # Einlesen (B)
  open(INP, ">", datei.txt)                  # Schreiben
  open(INP, ">>", datei.txt)                 # Anhängen

  @zeilen = <INP>                            # Lesen (von Handle INP) (A)
  @zeilen = readline(INP)                    # Lesen (von Handle INP) (B)

  while (<STDIN>) { ... }                    # Lesen (von STDIN)
  while (<>) { ... }                         # Diamond-Operator (analog)
```

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

2.5) String, Regex, Liste, Parameter spezieller Operatoren

Als Begrenzungszeichen von String, Regex, Liste und den Parametern einiger spezieller Operatoren sind in Perl alle vier Klammerarten wählbar. Dann muss allerdings immer einer dieser Operatoren vorangestellt werden:

```
  "String"                                   # String (mit Substitution)
  'String'                                   # String (ohne Substitution)
  /Regex/                                    # Regex
  `Regex`                                    # Externes Kmdo ausführen

  qq{...}                                    # String (double quote)   "..."
  q{...}                                     # String (single quote)   '...'
  qw{...}                                    # Wortliste (quote word)  (...)
  qx{...}                                    # Execute (quote execute) `...`
  qr{...}                                    # Regex (quote regex)     /.../
  m{...}                                     # Regex (match)
  s{...}{...}                                # Regex (substitute)
  tr{...}{...}                               # tr-Emulation (translate)
  y{...}{...}                                # tr-Emulation (yank)
```

HINWEIS: Üblicherweise werden bei obigen speziellen Operatoren die geschweiften Klammern "{...}" als Begrenzer verwendet, es sind aber auch die anderen Klammerarten bzw. beliebige Begrenzerzeichen nutzbar. Sinnvollerweise wählt man als Begrenzerzeichen eines aus, das nicht im String oder Regex vorkommt.

```
  qq(...)                                    # String (double quote)   "..."
  qq[...]                                    # String (double quote)   "..."
  qq<...>                                    # String (double quote)   "..."
  qq@...@                                    # String (double quote)   "..."
  q|...|                                     # String (single quote)   "..."
  m/.../                                     # Regex (match)
  s(...)[...]                                # Regex (substitute)
```

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
```

HINWEIS: Nur in sehr wenigen Fällen sind Klammern wirklich notwendig, um den (falschen) Vorrang von Operatoren zu vermeiden, die Anzahl der an eine Funktion übergebenen Werte festzulegen oder den Aufruf einer (noch nicht definierten) eigenen Funktion anzudeuten. Aus Gründen der Übersichtlichkeit sollten die Parameter von Funktionen (wie in anderen Programmiersprachen auch) aber immer geklammert werden.

```
  sort $a, $b, $c, 5   -->   sort($a, $b, $c, 5)
  substr $a, 3, 5      -->   substr($a, 3, 5)
```

Sogenannten "Listen-Operatoren" wie "sort", "print", "..." sammeln möglichst viele Parameter auf ihrer rechten Seite. Soll dieses "Sammeln" begrenzt werden, dann muss geklammert werden:

```
  print      "z", "c", "b", "a", "\n"       # --> "zcba\n" (OK)
```

```
  print(     "z", "c", "b", "a"), "\n"      # --> "zcba"   ("\n" IGNORIERT)
  print sort "z", "c", "b", "a", "\n"       # --> "\nabcz" ("\n" MITSORTIERT)
  print(sort "z", "c", "b", "a", "\n")      # --> "\nabcz" ("\n" MITSORTIERT)
  print(sort "z", "c", "b", "a"), "\n"      # --> "abcz"   ("\n" IGNORIERT)
  print sort("z", "c", "b", "a"), "\n"      # --> "abcz\n" (OK)
  print(sort("z", "c", "b", "a"), "\n")     # --> "abcz\n" (OK)
```

Weiteres Beispiel:

```
  print substr "abcdef", 2, 3               # --> "cde"
  print substr "abcdef", 2, 3, "\n"         # --> Syntaxfehler
  print substr "abcdef", 2, 3 . "\n"        # --> "cde"   ("\n" fehlt + Warnung)
  print substr("abcdef", 2, 3), "\n"        # --> "cde\n" (OK)
  print(substr("abcdef", 2, 3) . "\n")      # --> "cde\n" (OK)
  print(substr("abcdef", 2, 3), "\n")       # --> "cde\n" (OK)
  print(substr("abcdef", 2, 3) . "\n")      # --> "cde\n" (OK)
```

TIPP: Man sollte sich für eine Art der Schreibweise entscheiden und diese dann durchgehend nutzen. Hier noch einige Beispiele für die Schreibweisen mit und ohne Klammerung:

```
  # Beides geht
  @arr = split / /, "abc def ghi"
  @arr = split(/ /, "abc def ghi")

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

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

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

4) Formatierung

Bei Klammern Leerzeichen so verteilen:

  * VOR öffnender Klammer ein Leerzeichen, danach nicht
  * NACH schließender Klammer ein Leerzeichen, davor nicht
  * Bei Listen/Array/Hash-Definition NACH öffnender + VOR schließender Klammer
  * Bei Array/Hash-Zugriff und Funktionsaufruf kein Leerzeichen um Klammern

Beispiel:

```
  if( $i == 10 ) ...                # unschön
  if ($i == 10) ...                 # OK

  ($a, $b) = ($a, $b)               # Vertauschen zweier Werte

  @arr  = ( 10, 20, 30 )            # Array-Definition
  %hash = ( "a" => 10, "b" => 20 )  # Hash-Definition

  $arr[1]                           # Array-Zugriff
  $hash{"abc"}                      # Hash-Zugriff
  &func(10.0, -4, "abc")            # Funktionsaufruf
```

(Zusätzliche) Klammern sind mehr zu tippen, tragen aber oft zum besseren Verständnis bei. (Zu) viele Klammern können allerdings auch das Verständnis erschweren. Dann hilft oft die Aufteilung einer Anweisung auf mehrere Zeilen und Einrückung gemäß der Klammerhierarchie:

```
  if ((($a) lt ($b)) or (($c) gt ($b)))      # Etwas zuviel geklammert ;-(
```

```
  if ($a lt $b or                           # Umbruch macht manches klarer
      $c gt $b)
```

Unübersichtlich:

```
  if (my ($a, $b, $c) = ($text =~ /^\s*(\w+)\s+(\w+)\s+(\d+)\s*$/))
  {
      ...
  }
```

Übersichtlich:

```
  if (my ($a, $b, $c) =
      ($text =~
         m{                 # match
              ^             # Zeilenanfang
                  \s*       # Leerraum (opt)
                  ( \w+ )   # Merken: Wort (muss)
                  \s+       # Leerraum (muss)
                  ( \w+ )   # Merken: Wort (muss)
                  s+        # Leerraum (muss)
                  ( \d+ )   # Merken: Zahl (muss)
                  \s*       # Leerraum (opt)
              $             # Zeilenende
          }x                # extended
      )
  ) {
      ...
  }
```