Spurensuche im Datenwald

Tutorial

---

graphen, database, awk, markdown,

html, css, svg, javascript, php, postgresql

2016-10-15

freyketh@mail.md

Graph **18**

Das Haus vom Nikolaus **18**
Eckpunkte (Knoten) 18
Verbindungslinien (Kanten) 18
Grad 18
Mehrfachverbindung 18
Knoten, Kante, Richtung **19**
Knoten (Vertex) 19
Kante (Edge) 19
einfacher Graph 19
gerichtete Kante (Bogen, Arc) 19
gewichteter Graph (Attribute) 19
benannter Graph (Label) 19
Multigraph 19
Schlinge (Schleife) 20
Endknoten (Blatt) 20
Knotenzahl & Kantenzahl 20
Isolierter Knoten 20
Zyklischer Graph 20
Azyklischer Graph 20
Trivialer Zyklus 20
Baum 21
Spannbaum (Spanning Tree) 21
Label 21
Dichte (Density) 21
Ordnung 21
Größe 21
Komponenten und Brücken **22**
Zusammenhängender Graph 22
Unzusammenhängender Graph 22
Brücke 22
Kantenloser Graph (Nullgraph) 22
Vollständiger Graph 22
Unvollständiger Graph 22
Grad (Valenz) **23**
Quelle (Source Node) 23
Maximalgrad & Minimalgrad 23
Durchschnittsgrad 23
Senke (Sink Node) 23
Regulärer Graph 23
Isomorphie **24**
Isomorphe Graphen 24
Isomorphie mit Adjazenzmatrizen erkennen 25
Präsentation als Menge **26**
Directed Graph (Gerichteter Graph) 26
Undirected Graph (Ungerichteter Graph) 26
Undirected Multigraph (Ungerichteter Multigraph) 27
Kantenzug, Weg, Kreis **28**
Kantenzug, Kantenfolge 28
Weg 28
Kreis (Zyklus) 28

Masche (Dreieck) 28
Eulersche Tour 28
Tour 28
geschlossene Tour 28
Eulerscher Graph 29
Hamiltonscher Kreis 29
Hamiltonscher Graph 29
Vollständiger Graph 29
Unvollständiger Graph 29
Beziehungsformen (Adjazenz, Inzidenz) 30
Adjazenz-Matrix (Knoten-Knoten-Matrix) 30
Adjazenzliste 31
Inzidenzmatrix (Kanten-Knotenmatrix) 31
Datenstruktur und Repräsentation der Beziehungsform 32
Beispiel-1: ein gerichteter Multigraph als Liste und Matrix 32
Beispiel-2: ein ungerichteter Multigraph als Liste und Matrix 32
Beispiel-3: ein ungerichteter Multigraph als verkettete Liste 33
Beispiel-4: ein gerichteter Multigraph als verkettete Liste 33
Beispiel-5: ein ungerichteter Multigraph als Matrix 33
Beispiel-6: ein gerichteter Graph (Disgraph) als Matrix 34
Beispiel-7: ein ungerichteter Multigraph als Matrix 34
Beispiel-8: ein gerichteter Multigraph als verkettete Liste 34
Bipartition 35
Bipartiter Graph (Paar Graph) 35
Vollständig bipartiter Graph 36
Hypergraph 36
Travesierung (Traversal) 37
Breitensuche (Breadth-First Search, BFS) 37
Tiefensuche (Depth-First Search, DFS) 37
Baum 38
Spannbaum (Spanning Tree) 39
Gewurzelter Baum 39
Geordneter Wurzelbaum 40
Suchbaum 40
Binärbaum (Binary Tree) 40
AVL Baum 41
B-Tree 41

Linux: Bash 43

Shell 43
Command Line Interface (CLI) 43
Command Line Interpreter (CLI) 43
Command Line User Environment (CLOE) 43
Shell Familie 43
Parameter 44
Backslash im Input 44
Eingabeaufforderung (Prompt) 45
History 47
Optionen 49
Variablen 50
type 51
file 51
Keyboard Shortcuts 52
Undo & Cancel 52

Cursor 52
Prozesse 52
Completion 53
Löschen 53
Cut & Paste 53
Tauschen (swap) 53
Character Case 54
History 54
History mit Word Designators 54
touch 56
Parameter 56
Beispiele 57
Unterschied mtime, ctime, atime, relatime, noatime 58
tar (tape archive 59
Optionen 59
Beispiele 60
Link 61
I/O Redirection Operator 62
Standard Input/Output 62
Pipe 62
ls 63
File Classification 63
long list 63

Linux: awk 66

Trennzeichen (Separator) 66
Record (Datensatz) 66
Zeilentrennzeichen 66
Spaltentrennzeichen 66
Feld (Zeilenspalte) 68
awk Processing Cycle 69
Main Input Loop 69
Action 69
Shell-Aufruf 70
Parameter 70
Input per Pipe 71
Output an Pipe 71
ENVIRON (Gawk Environment-Variable) 71
Mehrere Trennzeichen (Multiple Delimiter) 72
Beispiel geburtsdaten.txt 72
Beispiel klammern.txt 72
awk Builtin-Variablen 73
Beispiel mitarbeiter.txt 74
Pattern-Action Statement 74
Regular Expression Operator 75
Externes awk-Skript 75
Remark 75
Pipe 76
Programmblock (Aktionsteil) 76
Variablen 77
Skalar 77
Array 78
Operatoren 80
Assignment Operator (Zuweisung) 80

Conditional Operator (Bedingung) 80
Regular Expression Operator 80
Ternary Operator (Conditional Assignment) 81
Vergleich Numeric mit String 81
Konkatenation (Verkettung) 82
Priorität und Assoziativität von Operatoren 83
Mehrere Programmblöcke 84
Bedingter Programmblock 84
Regular Expressions 85
Operatoren für einen Regular-Expression Vergleich 86
String-Funktionen 87
print 87
printf() 88
strftime() 89
sub() 91
gsub() 92
length() 92
index() 92
toupper() 92
tolower() 92
match() 93
split() 94
IGNORECASE 94
Mathematische Funktionen 95
strtonum() 95
Würfel-Funktion 95
ord() 95
Array-Funktionen 96
asort() 96
asorti() 96
delete 97
System-Funktionen 98
system() 98
systime() 98
Zeit-Funktionen 98
systime() 98
strftime() 98
mktime() 98
function() 99
Kommandozeilen-Argumente (ARGV, ARGC, ARGIND) 100
Kontrollstrukturen 101
if 101
for-Schleife 102
while-Schleife 103
do-while-Schleife 104
exit 104
next 105
nextfile 105
return 105
awk Einzeiler 106
Vorkommen pro Zeile zählen 106
Gesamt-Spaltenanzahl ermitteln 106
Zeige die Zeile, die die längste 1. Spalte hat 106

Zeige von jeder Zeile die letzte Spalte 106
Enferne doppelte Zeilen 106
Jede Zeile verdoppeln 107
Die 2. Spalte mit 10 multiplizieren 107
Tausche Spalte 1 mit Spalte 2 107
Zeilennummern am Zeilenanfang 107
Nur Zeilen mit mehr als 10 Zeichen ausgeben 107
Ohne Leerzeilen 107
getline 108
externes File 109
pipe 109
close() 110
Vergleich von Nawk und Gawk 111

Markdown 113

Zeilenumbruch 113
Möglichkeit 1: \n\n 113
Möglichkeit 2: \n\n 113
Möglichkeit 3: \\n 113
fett, kursiv, durchgestrichen 114
Inline-HTML 114
Überschrift (Headline) 115
Möglichkeit 1 (#) 115
Möglichkeit 2 (-) 115
Blockquote 116
Nested Blockqoute 116
Horizontale Linie 116
Liste 117
Unordered List 117
Ordered List (Numbered List) 117
Nested List 117
Block (Monospace) 119
Möglichkeit1: Tab 119
Möglichkeit2: ` 119
Bild 120
Title 120
Uniform Resource Locator (URL) 120
Möglichkeit 1: URL anzeigen 120
Möglichkeit 2: Bezeichnung anzeigen 121
Referenzierung 122
Tabelle 122
Escape Character (Fluchtsymbol) 123
markdown-Steuerzeichen 123

HTML 125

Änderungen zu den Vorgängerversionen 125
Doctype 125
Sprache (Language) 125
Namespace 125
CSS- und Javascript-Type 125
Semantische Strukturblöcke 125
Zeichensatz 126
Semantische Tags (aka *Innere Werte*) 127
<article> <section> 127

<header> 127
<nav> <menu> 128
<aside> 128
<figure> <figcaption> 128
<time> 128
<data> 128
Generische Container-Tags **128**
 128
 128
Textformatierende semantische Tags **129**
Typographische Tags **133**
<sub> 133
<sup> 133
<wbr> 134

 134
Geänderter und Gelöschter Text **134**
<ins> 134
Eingebetteter Inhalt **134**
Interaktive Elemente **138**
<details> <summary> 138
Link **139**
<link> 139
<a> 140
Textgruppe **141**
<p> 141
<hr> 141
 141
 141
Liste **142**
 142
 142
Definitionsliste **143**
Obsolete HTML-Tags **144**
Umdefinierte HTML-Tags **145**
Obsolete Attribute **145**
Obsolete Attribute als Tag-Tabelle **147**
Obsolete Attribute **149**
Html5 Tags **150**
Tabelle **151**
Formular **152**
Mess- und Fortschrittsbalken **153**
meter 153
progress 153
Favicon **154**
ico 154
png 154
Touch Icon 154
Empfehlung developers.google.com 155
Event Handler **156**

CSS **158**

Deklaration **158**
Shorthand Properties 158
Deklarationsblock (CSS Rule-Set) 158

Inline Style 158
Cascading (Prioritätskaskade) 159
Spezifitätsgewichtung von Selektoren (Selector Specificity Weight) 159
Eltern, Kinder und Nachfahren 160
Wurzel (root) 160
Eltern (parent) 160
Kind (child) 160
Nachfahre (Descendant) 160
Geschwister (Sibling) 161
Vererbung (Inheritance) 161
inherit 162
initial 162
unset 163
Value Processing 164
Declared Value 164
Cascaded Value 164
Specified Value 164
Computed Value 164
Used Value 164
Actual Value 164
Selektoren 165
Elementselektor 165
ID-Selektor 166
Klassen-Selektor 166
Attribut-Selektor 167
Universalselektor 168
Pseudoklassen-Selektor 168
Pseudoelement-Selektor 168
Kombinatoren 169
Gruppen-Kombinator 170
Child Combinator (Kind) 170
Descendant Combinator (Nachfahre) 171
Adjacent Sibling Combinator (angrenzendes Geschwister) 171
General Sibling Combinator (Geschwister) 172
Box-Modell 172
Block versus Inline 172
Block-Level-Elemente 174
Box-Abstände 174
Box-Gesamtbreite 175
Box Margin 176
Inline-Level Elemente 177
Box Display 178
Box Properties 179
Fluss und Positionierung 180
float 180
clear 181
position 184
display 187
Pseudoklassen 188
Interaktive Pseudoklassen 189
Struktur-Pseudoklassen 196
Negation-Pseudoklasse 202
Language-Pseudoklasse 202

Pseudoelemente 203
::selection 203
::before 203
::after 203
::first-letter 204
::first-line 204
Shorthand Properties 205
Sub-Properties top, right, bottom, left 205
Property border 206
Property background 206
Property font 206
Property list-style 207
Property outline 207
Property transition 207
Vendor Prefix 208
Längeneinheit (Unit) 209
absolute Größeneinheiten 209
relative Größeneinheiten 209
em 210
Viewport Unit 210
Html & CSS 210
Inline Style 210
<link> 210
@import 210
Farben (Colors) 211
additiver Farbraum 211
subtraktiver Farbraum 211
RGB-Werte 212
@media 216
Zeichensatz 216
externe Schrift 216
Liste 217
list-style-type 217
list-style-position 218
list-style-image 219
Counter 219
counter-increment 219
counter 220
counter-reset 220
Beispiel 220

Zeichen (Character) 222

Glyphe 222
Glyphenvarianten 222
Zeichensatz (Character Set) 222
Zeichensatztabelle (Codepage) 222
Kodierung (Encoding) 223
Code Point, Code Unit 223
Mojibake (文字化け) 223
ASCII 224
Versalie, Majuskel 224
Control Characters (Steuerzeichen) 224
Space (SP), Delete (DEL) 224
No Break Space (NBSP), Soft Hyphen (SHY) 225

Caret Notation 225
Escape Sequence (Fluchtsequenz) 225
ASCII C0 Control Characters (0 – 31) 226
ASCII C1 Control Characters (128 – 159) 227
ASCII Character Codes 228
NUL 229
Extended ASCII (8bit) 230
No Break Space (NBSP), Soft Hyphen (SHY) 230
Unterschiede zwischen ISO-8859-1 (Latin1) und ISO-8859-15 (Latin9) 230
ISO-8859-1 (Latin1), ISO-8859-15 (Latin9) 231
Double-Byte Character Set (DBCS) 232
Unicode 233
Glype 233
Control Characters 233
UTF Encoding 234
Byte-Order (Byte-Reihenfolge) 235
Byte-Order-Mark (BOM) 236
Whitespace 237
Character mit Whitespace-Flag 237
Character ohne Whitespace-Flag 238
Leerraum anzeigende Characters 238

Zeichensatztabellen (Code Pages) 240

ISO-8859-1 (Latin1) 240
ISO-8859-15 (Latin9) 240
DEC Multilingual Character Set 241
Win-1252 (Microsoft Windows Western European) 241
Codepage 437 (IBM-PC default) 242
Codepage 850 (IBM-PC: European) 242

Scalable Vektor Graphics (SVG) 244

Header 244
xml 244
doctype 244
<svg> 244
Attribute (ohne viewPort) 244
viewPort 245
viewBox 246
Koordinatensystem 249
Maßeinheiten 249
relative Einheiten 249
absolute Einheiten 249
Shape Elements (Formen) 250
Füll-Attribute 250
Stroke-Attribute (Konturlinie) 252
<rectangle> 253
<circle> 254
<line> 255
<elipse> 255
<polyline> 256
<polygon> 256
<path> 257
<text> <tspan> <textPath> 262
<image> 267

Cascading Style Sheets (CSS) 268
inline declaration 268
Rule-Set 268
externer Rule-Set 268
Gruppierungselemente 269
<symbol> 269
<defs> 270
<g> 270
<marker> 271
<use> 273
Transformation 274
translate 274
scale 275
rotate 276
skew 277

JavaScript 279

Geschichte 279
Netscape/Firefox JavaScript-Versionen 279
Core, DOM, BOM 279
Versionen von ECMA-262 280
Document Object Model (DOM) 280
Browser Object Model (BOM) 281
Literal, Identifier, Token, Expression, Statement 282
Literal 282
Variable, Konstante 284
Identifiers 284
Token 285
Expression (Ausdruck) 285
Statement 285
Dot Notation 286
Square Bracket Notation 286
Line Continuation Character 286
Source Code Remark 286
Deklaration 287
var 287
"use strict" 291
const 293
let 293
Initialisierung 296
Hello World 297
document.write() 297
alert() 297
console.log() 297
Inline Script mit Event-Handler 297
Pseudo-URL 297
Datentypen (Data Types) 298
Complex Data Types (Reference Values) 298
Primitive Data Types (Primitive Values) 298
String Funktionen 309
Objekt 310
Übersicht 310
Objekt-Erzeugung 312

Property (Objekt-Eigenschaft) 316
this 321
Konstruktor 328
with 331
Objekt Enumeration 332
In-build Objekte 333
In-build-Eigenschaften 338
Klasse 339
Array 342
Feldgröße 342
Feldwerte 342
Feldindex 342
Array-Erzeugung 342
Array Eigenschaften (Array Properties) 345
Array Methoden 346
Operatoren 363
Arithmetische Operatoren 363
Vergleichsoperatoren 366
Ternäre Operator (Ternary Operator) 369
Logische Operatoren (Logical Operators) 370
Bitwise Operators 372
Comma Operator 373
void Operator 374
typeof Operator 374
instanceof Operator 375
Zuweisungsoperator (Assignment Operator) 375
Destrukturierende Zuweisung (Destructuring Assignment) 377
Assoziativität (Associativity) 378
Präzedenz (Precedence) 378
String-Konkatenation 380
spread Operator 381
Control Flow (Flusskontrolle) 382
Conditional Statements (Bedingte Flusskontrolle) 382
Loops (Schleifen) 386
Funktion 393
Datentyp 393
Erzeugung 397
Immediately Invoked Function Expression (IIFE) 401
Parameter 403
Variable Scope 407
Funktion als Objekt-Eigenschaft 410
Aufruf (Invoke) 410
Function Scope (Script, Block, Closure) 416
Call-Stack & Call-Site 421
Scope (Gültigkeitsbereich) 422
Lexical Scope 422
Global Scope 424
Function Scope 424
Block Scope 425
Keywords 429
Obtrusive and Unobtrusive JavaScript 430
Obtrusive JavaScript 430
Unobtrusive JavaScript 430

Progressive Enhancement 431
JavaScript in HTML 432
Inline JavaScript 432
Pseudo-URL 432
Embedded JavaScript mit <script> 432

PHP 435

Literal, Identifier, Variable, Token, Expression 435
Literal 435
Variable 437
Identifiers 438
Konstante 439
Token 444
Expression (Ausdruck) 444
Statement 444
Source Code Remark 446
PHP Tag 447
Kurzform (Short Delimiter) 447
echo Tag 447
Deprecated Tags 447
Datentypen (Data Types) 448
Skalar (Primitive Value) 448
Composite Data Types 456
Wertzuweisung (Value Assignment) 457
Variable Inspecting (Debug) 459
Type Casting 462
Funktionen 462
Operatoren 471
Operatoren 478
Type Casting 478
Präzedenz (Precedence) 478
Scope (Gültigkeitsbereich) 480
Global-Scope 480
Static Scope 482
Control Flow (Flusskontrolle) 483
Conditional Statements (Bedingte Flusskontrolle) 483
Loops (Schleifen) 487
Label & goto 493
exit (die) 494
Klassen & Objekte 495
Klasse 495
Objekt 502
Cookie 504
Security 505
Mißbrauch (Abusement) 505

SQL & postgreSQL 507

Relation 507
Body 507
Relationsschema 507
Attribut (Head) 507
Attributwert 507
Tupel, Record Set 507

Domain 507
Heap 507
Clustered Index (SQL Server) 507
psql 508
Shell Parameter 508
Meta-Commands (Backslash Commands) 508
Login 508
Exit 508
Server-Version 508
Datenbank 509
SQL Notationsbeschreibung 509
Relation (Tabelle) 510
SQL Syntax 511
Verben 512
Whitespace 514
Komma (List Separator) 514
Remark (Bemerkung) 514
Case Sensitive 515
Literale 516
Boolean Literal 517
String Literal 518
NULL Literal 521
Exponent-Literal 521
Zahlen-Literal 522
Hexadezimal Literal 522
DATE Literal 523
TIME Literal 523
TIMESTAMP Literal 523
Expression (Ausdruck) 524
Single Expression 524
Compound Expression 524
Compound Expression mit Identifier 524
Funktion 524
Datentpyen 525
Numerics (Zahlen) 526
SELECT Blöcke 539
Kernblock 539
Neben-Block 540
End-Block 540
Operatoren 541
String Konkatenation 541
Vergleichsoperatoren 542
Präzedenz (Precedence) 549
Ergebnislimitierung 550
top 550
fetch first rows only 551
offset rows 552
limit 552
limit offset 552
Case When Then 555
Syntax-Variante 1 555
Syntax-Variante 2 555
IS NULL 559

Verbund (Join) 560
Join Operatoren 560
Cross Join (Kartesisches Produkt) 562
Inner Join 565
Outer Join 567
Natural Join 570
Aggregate Functions 575
count() 576
sum() 578
min() 579
max() 580
avg() 581
string_agg() 582
Subquery 583
Positionen 583
Nested Subquery 584
Korrelation 584
Transaktion 585
ACID 585
Isolationsphänomene 585
Isolation Level 586
Transaktionsverwaltung 587
Stored Procedures (PL/pgSQL) 588
DO 588
Code Block 588
Variable-Deklaration 589
Conditional Statements (Bedingte Flusskontrolle) 591
Remarks 597
Beispieltabellen 598

Datenbank 602

Datenbank-Modelle 602
Hierarchische Datenbank-Modell 602
Vernetztes Datenbank-Modell 604
Datenbank-Index 605
Buchindex 605
Datenbank-Index (Tabellenindex) 605
Tabelle 606
Clustered Index 606
B+Tree 607
Suchschritte 609
Baumgröße im B+Tree 611
B+Baum mit Compound-Primary-Index 612
Baumteilung im B+Tree 613
Page Split beim MS-SQL-Server 615
Index-Scan Operationen 616
Table Access By Index Rowid 616
Table Access Full (Full-Table-Scan) 616
Constraint *Primary Key* 618
Primary Key (Primärschlüssel) 618
Optimzier (Ausführungsplaner) 627
Funktionsbasierender Index (Oracle) 628
Computed Columns (Microsoft) 630

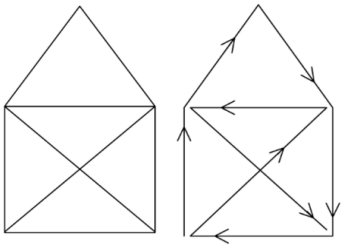
Parametisierte Abfragen (Bind-Variablen) 631
Prädikate 634
MySQL-Ausführungsplan 635
Stichwortverzeichnis 636
A-Z 636

Der Graph

Kapitel 1

Graph

Das Haus vom Nikolaus



»Das Haus vom Nikolaus« ist eine Knobel-Aufgabe für Kinder. Man soll Abbildung1 zeichnen, ohne den Stift abzusetzen und ohne eine Linie doppelt zu ziehen.
Man beginnt das Zeichnen der 8 Verbindungslinien links unten und sagt dabei ein Sprüchlein auf: »Das ist das Haus vom Ni – ko – laus. «
Ein Graph ist ein Gebilde (eine Struktur) und besteht aus Punkten und Verbindungslinien. Jede Kante (Verbindungslinie) verbindet genau 2 Ecken.

Eckpunkte (Knoten)

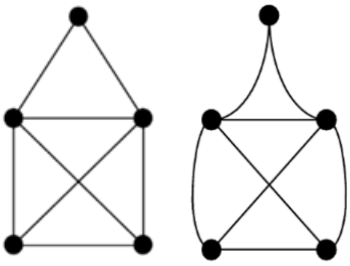
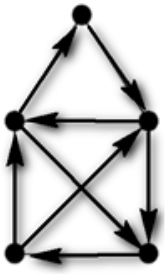


Abbildung3 ist das Haus als Graph. Die **Eckpunkte** werden meistens als Kreise dargestellt und auf Englisch **Vertex** (plural **Vertices)** genannt.
Das Haus hat 5 Ecken. Der Kreuzungspunkt in der Mitte zählt nicht als Ecke. Eine Ecke wird auch Knoten **(Node**) genannt.
Die Gesamtanzahl aller Knoten wird **Ordnung** gennant. Das Entfernen einer Ecke und seiner Kanten wird **Löschen** genannt.
Das Kreuzen von Verbindungslinien ergibt keinen Knoten.

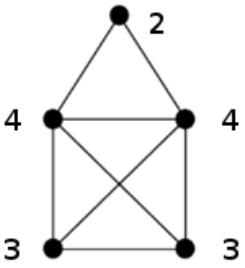
Verbindungslinien (Kanten)

Das Haus hat 8 **Verbindungslinien**. Diese Linien werden auch **Kanten** genannt (auf Englisch **Edge**). Ihre optische Form spielt keine Rolle.



Die Verbindungslinien müssen nicht zwingend als Gerade gezeichnet werden: siehe Abbildung4.
Kanten mit einer Richtungsinformation werden **gerichtete Kanten** (Bögen) genannt. Sie werden optisch als Richtungspfeil dargestellt.
Das Durchlaufen von aneinanderfolgenden Knoten und Kanten wird **Kantenzug** genannt. Ein Graph mit gewichteten Kanten wird **gerichteter Graph** genannt. Auf Englisch **Directed Graph (Digraph)** genant.
Die Länge des längsten Weges in einem Graphen nennt man **Umfang** des Graphen.
Bei einem **ebenen Graph** sind alle Kreuzungspunkte der Kanten ein Knoten. Das Haus ist kein ebener Graph.

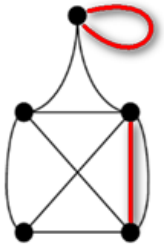
Grad



Der Grad ist die Anzahl der Verbindungen eines einzelnen Knoten. Beim Haus muß man den Kantenzug in einer Ecke mit dem Grad 3 beginnen. Eine Schlinge hat den Grad 2.

Beim einem Digraph ist der Eingangsgrad (**Indegree**) eines Knotens ist die Anzahl der gerichteten Kanten (Bögen), die zu dieser Ecke hinführen.
Der Ausgangsgrad (**Outdegree**) eines Knotens ist die Anzahl der gerichteten Kanten (Bögen), die von dieser Ecke wegführen.

Mehrfachverbindung



Einen Graphen mit parallelen Mehrfachkanten nennt man **Multigraph**.
Schleifen (Schlingen) sind Kanten, die einen Knoten mit sich selbst verbinden. Eine Schlinge hat den Grad 2.
Ein Graph ohne Schlingen und Mehrfachfachkanten wird einfacher Graph genannt.

Knoten, Kante, Richtung

Knoten (Vertex)

Der Punkt wird auch **Knoten** und **Ecke** (Eckpunkt) genannt. Auf Englisch **Vertex** (im Plural **Vertices**). Ein Punkt wird oft als Kreis und manchmal als Rechteck dargestellt.

Kante (Edge)

Die Linie wird auch **Kante** genannt (auf Englisch **Edge**). Durch das Verbinden mit einer Linie wird eine Beziehung zwischen den Knoten hergestellt. Wie beim Knoten, spielt das Aussehen der Linie keine Rolle (solange ein Pfeil weiterhin eine gerichtete Kante darstellt).

einfacher Graph

Ein einfacher (schlichter, gewöhnlicher) Graph enthält keine Mehrfachkanten und keine Schlingen. Im Beispiel rechts: Der Graph besteht aus 2 Knoten und einer Kante.

gerichtete Kante (Bogen, Arc)

Ist der Kante eine Orientierung (Richtung) zugewiesen, heißt sie Bogen und gerichtete Kante (auf Englisch **Arc**). Sie wird als Richtungspfeil dargestellt. Eine Kante ohne Richtung wird **ungerichtete Kante** genannt.

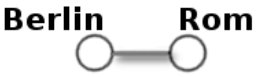
gerichteter Graph (Digraph)

Eine **gerichteter Graph** besteht nur aus **gerichteten Kanten (Bögen)**.
Auf Englisch **Directed Graph (Digraph)** genannt. Beispiel rechts: Der linke Punkt wird **Vorgänger (Predecessor)** genannt. Der rechte Punkt **Nachfolger (Successor)**. Eine **Senke** ist eine Ecke mit dem Ausgangsgrad 0.
Ein Digraph wird **stark zusammenhängend** genannt, wenn es von jeder Ecke zu jeder anderen Ecke einen gerichteten Weg gibt.

gewichteter Graph (Attribute)

Ein Graph, bei dem jeder Kante eine Bewertung (zB Zahl) zugeordnet worden ist. Im Beispiel hat die Kante das Attribut Länge, mit dem Attributwert 5. Im Gegensatz zu einer bloßen Bezeichnung (Label), ist das Gewicht entscheidend bei der Analyse des Graphen.

benannter Graph (Label)



Knoten und Kanten können mit Namen (Label) versehen sein. Man spricht man von einem benannten Graphen. Im Gegensatz zu einem gewichteten Graphen, spielen die Bezeichnungen keine Rolle bei der Analyse des Graphen.

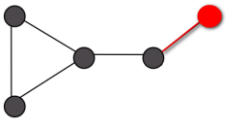
Multigraph

In einem Multigraph gibt es zwischen 2 Knoten mehr als eine Verbindung. Diese **Mehrfachkanten** werden **parallele Kanten** genannt. Auch eine Schlinge ist ein Multigraph (sie hat den Grad 2). Eine Mehrfachverbindung erzeugt einen geschlossenen Weg, also einen Kreis (Zyklus). Einen Kreis mit weniger als 3 Knoten wird **trivialer Zyklus** genannt.
Bei Mehrfachkanten spricht man auch über die Vielfachkeit der Kante.

Schlinge (Schleife)

Die Verbindungslinie kehrt in einer Eigenschleife zum Knoten zurück. Die Weglänge ist 1. Die Schlinge ist ein Multigraph und hat den Grad 2 (zwei Kanten). Die Schleife wird auf Englisch self-loop genannt. Eine Schleife erzeugt einen geschlossenen Weg, also einen Kreis (Zyklus). Einen Kreis mit weniger als 3 Knoten wird **trivialer Zyklus** genannt.

Endknoten (Blatt)



Ein Knoten mit Grad 1 wird Endknoten genannt (also ein Knoten mit nur einer Kante). Im Beispiel ist der rechte Punkt ein Blatt.

Knotenzahl & Kantenzahl

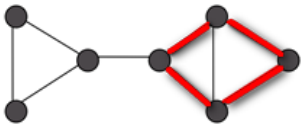
Die Knotenzahl ist die Anzahl aller Knoten, und die Kantenzahl die Anzahl aller Kanten eines Graphen.

Bei einem Multigraphen summiert man über die Vielfachheit der Kanten.

Isolierter Knoten

Ein Knoten ohne Verbindung (ohne Nachbarn) wird **Isolierter Knoten** genannt. Im Beispiel der mittlere Punkt.

Zyklischer Graph



Der Kreis wird auch Zyklus genannt, somit enthält ein zyklischer Graph mindestens einen Kreis. Ein Kreis ist ein geschlossener Weg, bei dem Startpunkt und Endpunkt identisch sind.

Ein Weg ist ein Kantenzug, bei dem jede Kante nur einmal verwendet werden darf.

Ein Graph ohne Kreis ist ein azyklischer Graph.

Azyklischer Graph

Ein Graph ohne Kreis ist ein azyklischer Graph.

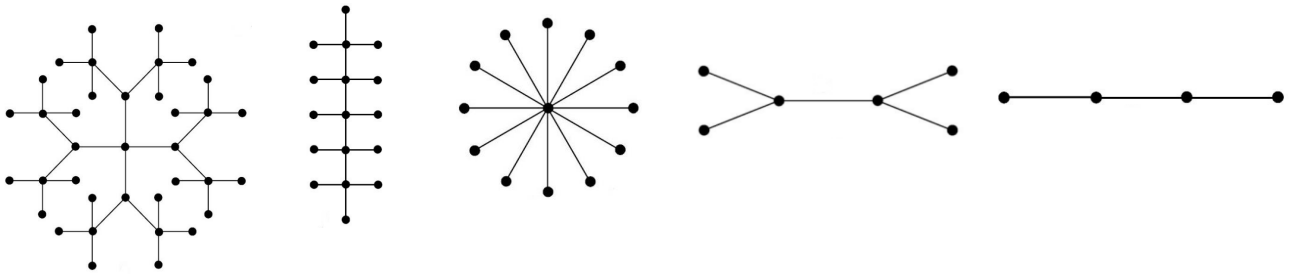
Ein Kreis ist ein geschlossener Weg, bei dem Startpunkt und Endpunkt identisch sind.

Trivialer Zyklus

Ein trivialer Zyklus ist ein Kreis mit weniger als drei Knoten. Zum Beispiel eine Schlinge oder eine Mehrfachverbindung.

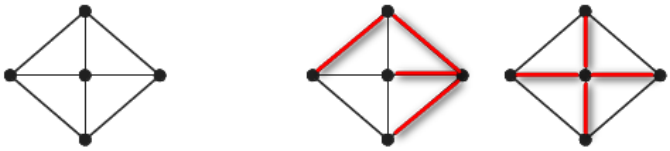
Baum

Ein Baum ist ein zusammenhängender Graph ohne Kreis. und Schlinge. Je zwei Knoten sind durch genau 1 Weg verbunden. Jeder Baum mit n Ecken hat genau n-1 Kanten. Bäume bestehen nur aus Brücken (können also kein Multigraph sein). Beim Entfernen einer Kante zerfällt der zusammenhängende Baum in Komponenten. 5 Beispiele:



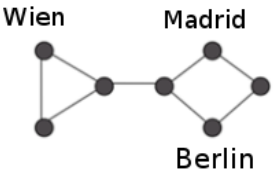
Spannbaum (Spanning Tree)

Der Spannbaum ist ein Teilgraph, der alle Knoten des Graphens enthält. Der Graph muss somit zusammenhängend sein.



Label

Ein Label ist die Bezeichnung einer Ecke oder Kante. Die Namen dürfen bei der des Graphen keine Rolle spielen (im Gegensatz zu einem gewichteten Graphen).



Analyse

Dichte (Density)

In einem einfachen Graphen stellt die **Kantendichte** das Verhältnis zwischen bestehender Kantenanzahl und möglicher Maximalanzahl dar.

Ein Graph ohne Kanten hat die Dichte 0. Ein vollständiger Graph hat die Dichte 1 (maximale Anzahl der Kanten).

Je geringer die Dichte, umso lichter der Graph.

Ordnung

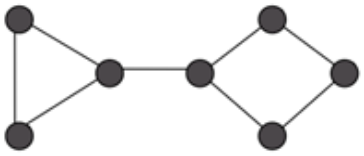
Die Ordnung eines Graphen ist die Anzahl aller Knoten des Graphen.

Größe

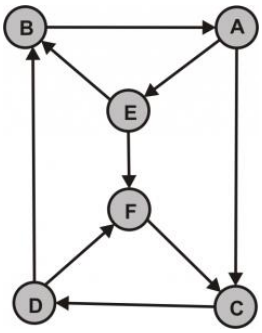
Die Größe eines Graphen ist die Anzahl aller Kanten des Graphen.

Komponenten und Brücken

Zusammenhängender Graph

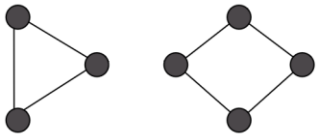


Ein Graph ist zusammenhängend, wenn es von jeder Ecke zu jeder anderen Ecke einen Kantenzug gibt.
Kann durch eine einzelne Kante der Graph zerfallen, wird diese Kante **Brückenkante** genannt.



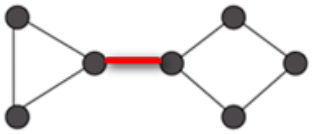
Ein gerichteter Graph **heißt stark zusammenhänged**, falls es für jedes Paar e,f von Ecken, sowohl einen Weg von e nach f, als auch von f nach e gibt. Ansonsten wird der Graph **schwach zusammenhänged genannt.**

Unzusammenhängender Graph



Ein Graph ist zusammenhängend, wenn für jedes Paar von Knoten ein Pfad existiert. Graphen die nicht zusammenhängend sind, können zumindest zusammenhängende Teile enthalten. Die Teilgraphen nennt man Komponenten.

Brücke

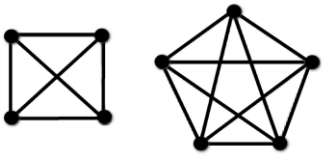


Wird nach dem Entfernen einer Kante aus einem zusammenhängenden Graph, ein unzusammenhängender, dann war diese Kante eine Brücke.
Bäume bestehen nur aus Brücken.

Kantenloser Graph (Nullgraph)

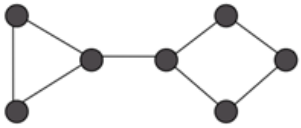
Ein kantenloser Graph besteht nur aus Ecken. Der unzusammenhängende Graph im Beispiel besteht aus 4 Komponenten. Ein Graph darf nicht nur aus Linien bestehen!

Vollständiger Graph



Ein Graph ist vollständig, wenn jede Ecke mit jeder anderen Ecke durch genau eine Kante verbunden ist.

Unvollständiger Graph



Unvollständige Graphen enthalten nicht alle maximal möglichen Kanten. Somit ist nicht jede Ecke mit jeder anderen Ecke durch eine Kante verbunden.

Grad (Valenz)

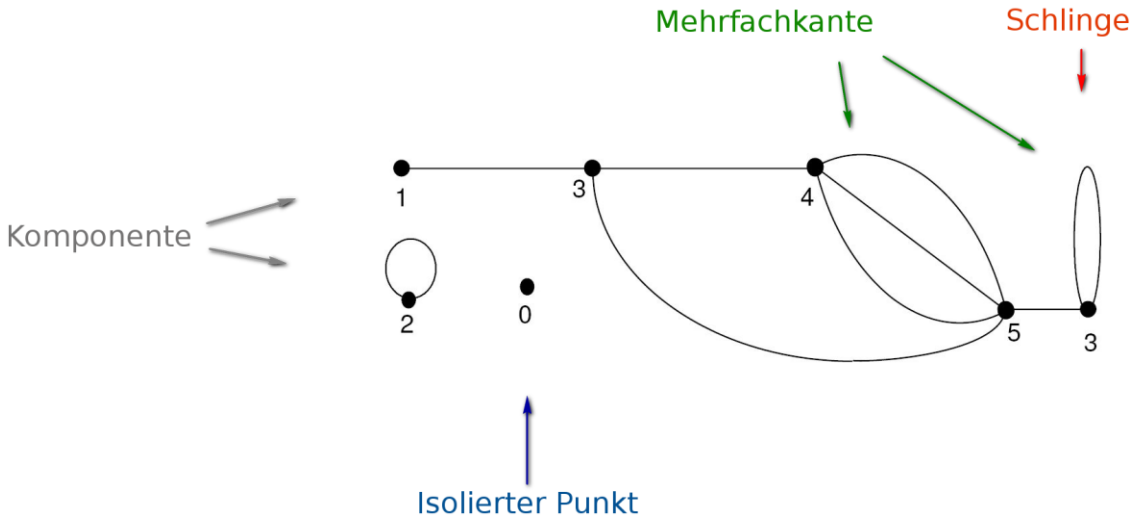
Der Grad ist die Anzahl der Verbindungen eines einzelnen Knoten. Beim Haus muß man den Kantenzug in einer Ecke mit dem Grad 3 beginnen. Eine Schlinge hat den Grad 2.

Bei einem Digraph:

Der Eingangsgrad (**Indegree**) eines Knotens ist die Anzahl der gerichteten Kanten, die zu dieser Ecke hinführen.

Der Ausgangsgrad (**Outdegree**) eines Knotens ist die Anzahl der gerichteten Kanten, die von dieser Ecke wegführen.

Das Beispiel ist ein unzusammenhängender Graph, der aus 3 Komponenten besteht.



Quelle (Source Node)

Eine Senke ist eine Ecke mit dem Eingangsgrad 0.

Maximalgrad & Minimalgrad

Der Maximalgrad ist der größte Knotengrad aller Knoten des Graphen.

Der Minimalgrad ist der kleinste Knotengrad aller Knoten des Graphen.

Durchschnittsgrad

Das arithmetische Mittel aller Eckengrade von G wird Durchschnittsgrad genannt.

Senke (Sink Node)

Eine Senke ist eine Ecke mit dem Ausgangsgrad 0.

Regulärer Graph

Hat jede Ecke den gleichen Grad, wird der Graph regulär genannt. Jede Ecke im Beispiel hat den Grad 2.

Besitzen alle seine Knoten Grad *k,* so bezeichnet man *G* als **k-regulär**. Einen 3-regulären Graphen nennt man **kubisch.**

Ein gerichteter Graph *G* heißt **regulär, wenn** alle seine Knoten den selben Eingangs- und Ausgangsgrad besitzen.

Besitzen alle seine Knoten Eingangs- und Ausgangsgrad *k,* so bezeichnet man *G* als k-regulär.

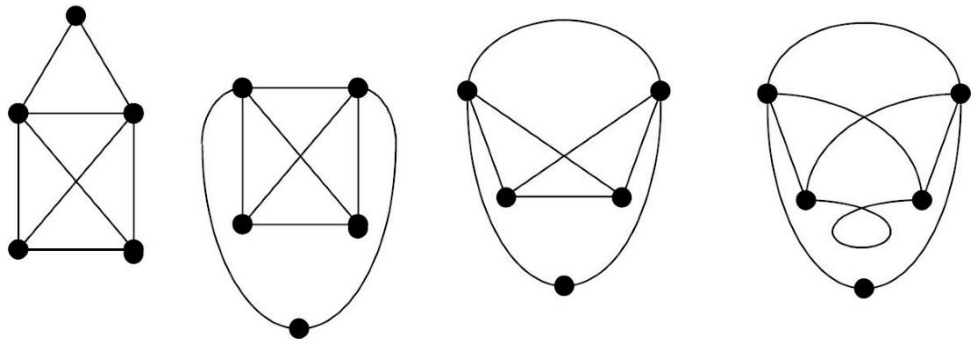
Isomorphie

Isomorphe Graphen

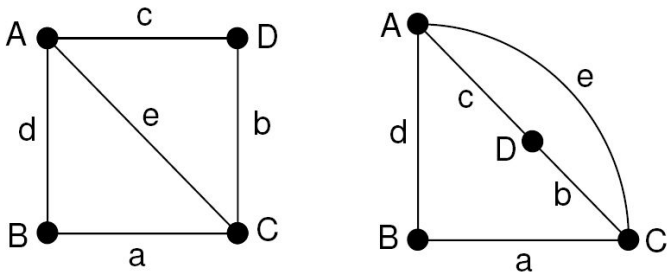
Zwei Graphen sind isomorph, wenn man den zweiten Graphen durch ein Verformen des ersten Graphen erzeugen kann. Seien G1 und G2 zwei Graphen:

Entspricht jeder Ecke von G1 eine Ecke von G2 und sind zwei Ecken von G2 genau dann verbunden, wenn die entsprechenden Ecken von G1 verbunden sind, wobei auch die Anzahl der Kanten übereinstimmen muss, so heißen die Graphen isomorph.

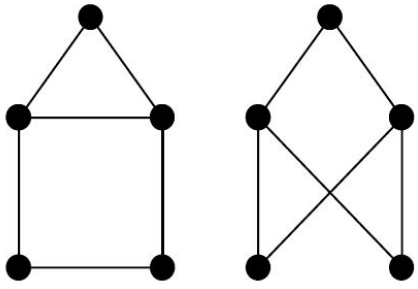
Alle 5 Abbildungen sind isomorph:



Auch diese beiden Graphen sind isomorph:



Ob 2 Graphen isomorph sind, lässt sich nicht an der Anzahl der Linien, Ecken und ihren jeweiligen Graden feststellen. Folgende 2 Graphen sind nicht isomorph, obwohl beide Graphen 5 Ecken, 6 Kanten, 3 Ecken mit Grad 2 und 2 Ecken mit Grad 3 haben:



Ein Graph wird plättbar (planar) genannt, wenn er sich als ebener Graph umzeichnen (verformen) lasst und somit zu ihm isomorph ist.

Zwei Graphen G1(E1, K1) und G2 (E2, K2) heißen isomorph, wenn es eine bijektive Abbildung $\phi : E1 \rightarrow E2$ und eine bijektive Abbildung $\psi : K1 \rightarrow K2$ gibt, die die folgende Eigenschaft hat: Wenn $K \in K1$ eine Kante zwischen den Ecken A und B von G1 ist, gilt: $\psi(k)$ ist eine Kante zwischen den Ecken $\varphi(A)$ und $\varphi(B)$.

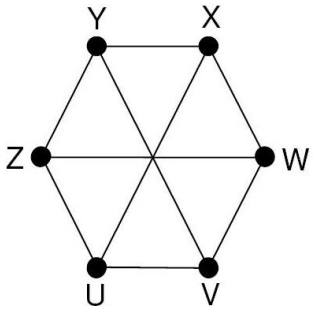
Isomorphie mit Adjazenzmatrizen erkennen

Aus einem Graphen lässt sich eine **quadratische Knoten-Knoten-Tabelle** erstellen.
Wenn die jeweilige Knoten-Knoten-Verbindung existiert, wird in die jeweilige Zelle eine 1 eingetragen (ansonsten eine Null). Diese Knoten-Knotenmatrix wird auch **Adjazenzmatrix** genannt.

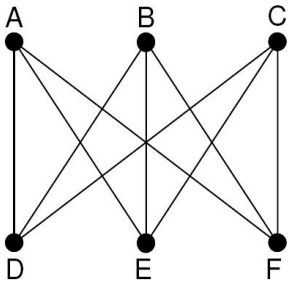
Adjazenz ist die Beziehung zwischen 2 Knoten.
Inzidenz ist die Beziehung zwischen einer Kante und einem Knoten.

Eine Adjazenzmatrix ist eine boolesche (n x n)-Matrix A = (aij).
Die Einträge aij beschreiben die Kanten von Knoten i zu Knoten j.
Besteht eine Verbindung zwischen den Knoten i und j, so ist aij gleich 1, sonst gleich 0.

In den beiden Graphen hat jede Ecke einen eindeutigen Namen (Label). Beide Graphen ergeben die gleiche Adjazenzmatrix. Somit sind beide Graphen **isomorph** (unter der Annahme, dass die Eckenbezeichnung keine Rolle spielt).

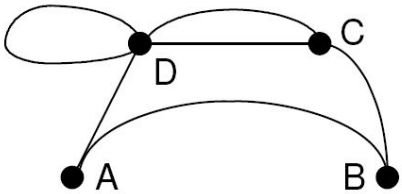


	U	V	W	X	Y	Z
U	0	1	0	1	0	1
V	1	0	1	0	1	0
W	0	1	0	1	0	1
X	1	0	1	0	1	0
Y	0	1	0	1	0	1
Z	1	0	1	0	1	0



	A	B	C	D	E	F
A	0	1	0	1	0	1
B	1	0	1	0	1	0
C	0	1	0	1	0	1
D	1	0	1	0	1	0
E	0	1	0	1	0	1
F	1	0	1	0	1	0

Bei einem gerichteten Multigraph wird die Anzahl der Linien eingetragen (also kein Bool mehr). Je nach Definition kann für eine Schlinge eine 1 oder eine 2 (=Grad) gezählt werden.



	A	B	C	D
A	0	1	0	1
B	1	0	1	0
C	0	1	0	2
D	1	0	2	1

Bei einem ungerichteten Digraph kann die Richtung mit -1 und +1 in der Matrix abgelegt werden.

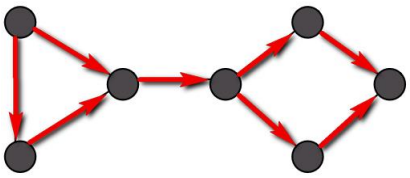
Präsentation als Menge

Ein Graph besteht aus Ecken (E) und Kanten (K). Somit ist ein Graph ein Paar von endlichen Mengen E und K.
Jedem Element von K ist eine zweielementige oder einelementige Teilmenge von E zugeordnet.
Die Elemente von E nennt man Ecken, die Elemente von K nennt man Kanten. E ist nicht die leere Menge.

Die Menge der Ecken darf nicht leer sein.
Die Menge der Kanten darf leer sein. Ein kantenloser Graph besteht aus lauter isolierten Ecken.

Directed Graph (Gerichteter Graph)

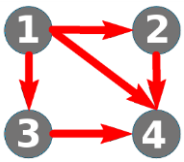
Ein gerichteter Graph besteht aus einer Menge *V* von Knoten (Vertices) und einer Menge *A* von **Arcs** (gerichtete Bögen).



Wobei ein Element a aus A einem geordneten Paar a=(i, j) von Knoten entspricht.
Man sagt: i ist Anfangsknoten und j ist Endknoten von a.
Ein Knoten i heißt Vorgänger des Knoten j, falls ein Pfeil (i, j) existiert.
Ein Knoten i heißt Nachfolger des Knoten j, falls ein Pfeil (j, i) existiert.
Vorgänger und Nachfolger werden als Nachbarn bezeichnet. (i, j) ist ein geordnetes Paar.

Beispiel

Ein gerichteter Graph G = (V , E) ist die Zusammensetzung einer Menge V (Vertices) von Knoten und einer Relation E ⊂ V × V von Kanten (Edges).

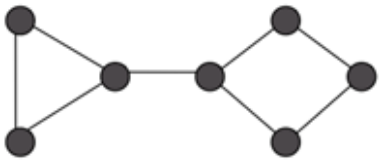


V={1,2,3,4}

E={(1,2), (1,3), (1,4), (2,4), (3,4)}

Undirected Graph (Ungerichteter Graph)

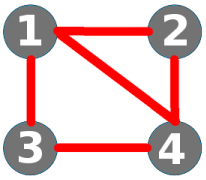
Ein **ungerichteter (gewöhnlicher) Graph** besteht aus **ungerichteten (gewöhnlichen) Kanten**. Richtungsinformationen sind nicht vorhanden.
Im Beispiel rechts: Der Graph besteht aus 7 Knoten und 8 Kanten.



Ein ungerichteter Graph besteht aus einer Menge V von Knoten (Vertices) und einer Menge *E* von Edges.
Wobei ein Element e aus E einer zwei-elementige Teilmenge e={i, j} von Knoten entspricht.
Man sagt: i und j sind die Endknoten der Kante e
{ i, j } ist ein ungeordnetes Knotenpaar.

Beispiel

Menge V von Knoten (Vertices) und einer Menge *E* von Edges:
V={1,2,3,4}



E={(1,2), (1,3), (1,4), (2,4), (3,4), (2,1), (3,1), (4,1), (4,2), (4,3)}

Undirected Multigraph (Ungerichteter Multigraph)

Ein Graph ist gegeben durch

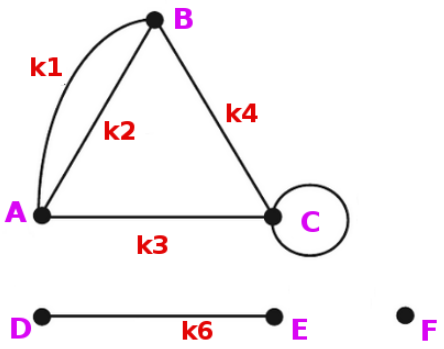
1) eine endliche Menge *E* (die Eckenmenge),
2) eine endliche Menge *K* (die Kantenmenge)
3) eine Abbildung $f : K \rightarrow F$, wobei *F* die Menge aller 2-elementigen Teilmengen von *E* ist.

Jede 2-elementige Teilmenge von *E* ist von der Form {x, y}, wobei x und y zwei verschiedene Elemente von *E* sind. Die Funktion f ordnet jedem Element k ∈ K ein Paar von Ecken zu.
Die beiden Ecken dieses Paares bezeichnen man als »durch die Kante k verbunden«.
Der Graph *G* ist das Tripel (*E, K, f*)

E = {A, B, C, D, E, F}

K = {k_1, k_2, k_3, k_4, k_5, k_6}

f: f(k_1) = (A,B)
f: f(k_2) = (A,B)
f: f(k_3) = (A,C)
f: f(k_4) = (B,C)
f: f(k_5) = (C,C)
f: f(k_6) = (D,E)

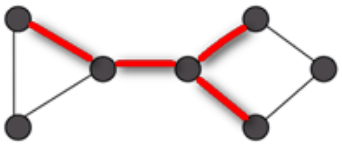


In der Mengenlehre heißen zwei Mengen A und B disjunkt oder elementfremd, falls sie kein gemeinsames Element besitzen.
A ={1 2 5} und *B* ={7 8 12} sind disjunkt.
A ={1 2 7} und *B* ={6 7 8 12} sind nicht disjunkt da sie die 7 gemeinsam besitzen

Kantenzug, Weg, Kreis

Es gibt für die Begriffe Weg und Kantenfolge keine einheitliche Definition.

Kantenzug, Kantenfolge



Die häufigste Definition: Der **Kantenzug** ist eine Folge von aneinandergrenzenden Knoten und Kanten (also eine **Kantenfolge**). Die Länge ist die Anzahl der Kanten. Jede Kante kann öfters berührt werden. Der **Kantenzug** kann geschlossen sein (Startpunkt ist auch der Endpunkt) oder offen sein. Auch eine einzelne Kante kann Kantenzug genannt werden.

Eine andere Definition erlaubt für die Kantenfolge keine doppelte Verwendung einer Kante.

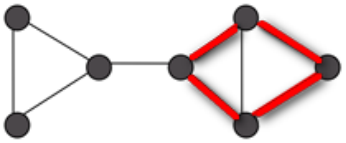
Weg

Eine häufige Definition von Weg: Ein Weg ist ein Kantenzug, bei dem jede Ecke nur einmal berührt wird. Somit kann sich ein Weg nie kreuzen. Die Weglänge ist die Anzahl der Kanten.

Ein geschlossener Weg ist ein Kreis (Startpunkt ist gleich Endpunkt).

Eine andere Definition bezieht sich auf die einmalige Verwendung einer Kante.

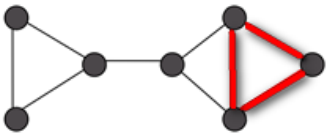
Kreis (Zyklus)



Der Kreis ist eine geschlossenen Kantenfolge, bei der jede Ecke nur einmal berührt wird. Oft auch geschlossener Weg genannt. Ein Graph mit einem Zyklus wird **Zyklischer Graph** genannt und ein Graph ohne Kreis ist ein **Azyklikscher Graph**.

Ein **trivialer Zyklus** ist ein Kreis mit weniger als 3 Knoten (Schlinge, Mehrfachverbindung).

Masche (Dreieck)



Eine Masche, auch Dreieck genannt, ist ein Zyklus ohne inneren Zweig.

Eulersche Tour

Ein Euler-Kantenzug führt durch jede Kante des Graphen genau einmal und endet an seinem Anfangspunkt.

Tour

Der Kantenzug enthält keine Kante doppelt. Eine Tour ist geschlossen, wenn der Startpunkt gleich dem Endpunkt ist (ansonsten ist die Tour offen).

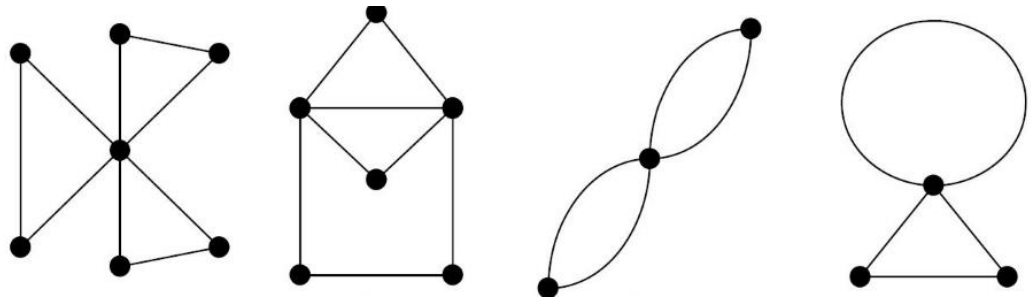
geschlossene Tour

Der Kantenzug enthält keine Kante doppelt und der Startpunkt ist gleich dem Endpunkt.

Eulerscher Graph

Ein Graph, der eine eulersche Tour enthält. Einen eulerschen Graphen kann man in einem Zug zeichnen (ohne den Stift abzusetzen), ohne eine Kante doppelt zu verwenden und mit dem letzten Zug zum Startpunkt zurückkehren.

Jede Ecke muss einen geraden Grad (2,4,6,8,...) haben.

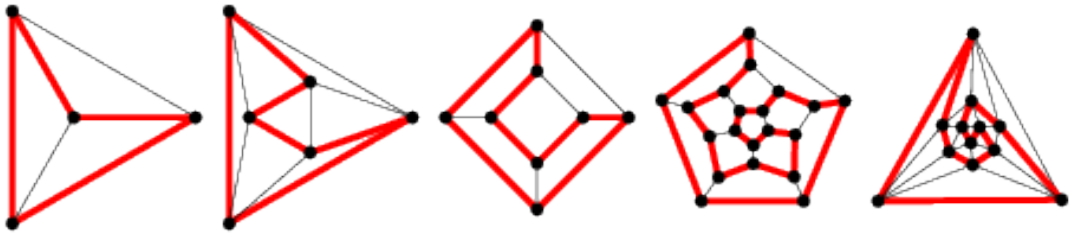


Hamiltonscher Kreis

Ein Kreis, der alle Ecke des Graphen enthält.

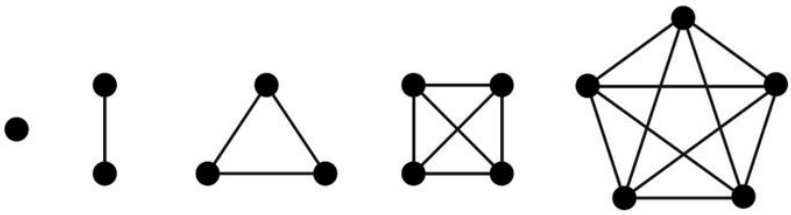
Hamiltonscher Graph

Ein Graph, der einen hamiltonschen Kreis enthält.



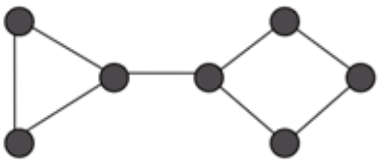
Vollständiger Graph

Ein Graph ist vollständig, wenn jede Ecke mit jeder anderen Ecke durch genau eine Kante verbunden ist.
Fünf Beispiele:



Unvollständiger Graph

Unvollständige Graphen enthalten nicht alle maximal möglichen Kanten. Somit ist nicht jede Ecke mit jeder anderen Ecke durch eine Kante verbunden.



Beziehungsformen (Adjazenz, Inzidenz)

Adjazenz liegt dann vor, wenn zwei Knoten über eine Kante miteinander verbunden sind. Somit bezeichnet Adjazenz die Beziehung zwischen gleichartigen Elementen eines Graphen. Mathematisch wird die Adjazenz mit der Adjazenzmatrix wiedergegeben.

Inzidenz ist die Beziehung zwischen einer Kante und einem Knoten. Das heißt eine Kante inzidiert mit ihrem Anfangs- und Endknoten.

Adjazenz-Matrix (Knoten-Knoten-Matrix)

Eine Adjazenzmatrix (Adjacency Matrix) ist eine boolesche (n x n)-Matrix A = (aij).
Die Einträge aij beschreiben die Kanten von Knoten i zu Knoten j.
Besteht eine Verbindung zwischen den Knoten i und j, so ist aij gleich 1, sonst gleich 0.

Dimension:

dim (A) = n * n mit n der Anzahl der Knoten

Die **Matrixelemente**:

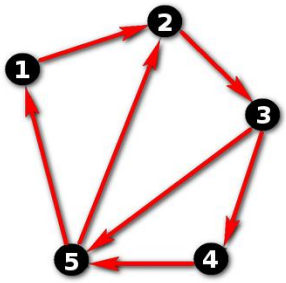
Ai,i = Anzahl der im Knoten i zusammentreffenden Kanten für i=j,

Ai,j = 1, wenn i,j eine Kante des Graphen darstellt
Ai,j = 0, wenn i,j keine Kante darstellt

Somit kann in dieser Matrix auf der Hauptdiagonalen abgelesen werden, wieviele Kanten in einem Knoten beginnen bzw. enden. In der Nebendiagonale sind die Knoten besetzt (zu- oder abgehende Knoten).

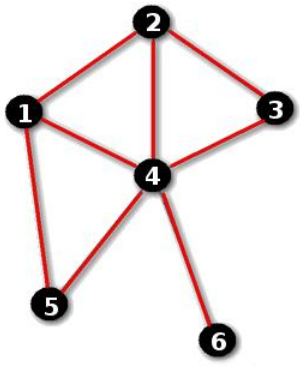
Als 1. Beispiel die Darstellung eines gerichteten Graphen in einer quadratischen (n,n) Matrix:

	1	2	3	4	5
1	0	1	0	0	0
2	0	0	1	0	0
3	0	0	0	1	1
4	0	0	0	0	1
5	1	1	0	0	0



Als 2. Beipiel ein ungerichteter Graph:

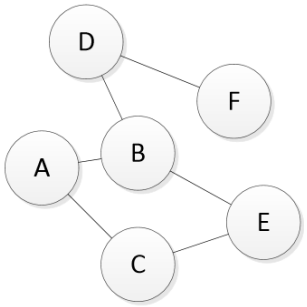
	1	2	3	4	5	6
1	0	1	0	1	1	0
2	1	0	1	1	0	0
3	0	1	0	1	0	0
4	1	1	1	0	1	1
5	1	0	0	1	0	0
6	0	0	0	1	0	0



Adjazenzliste

Für jeden Knoten gibt es eine Auflistung seiner verbundenen Knoten-Nachbarn:

A -> {B,C}
B -> {A,D,E}
C -> {A,E}
D -> {B,F}
E -> {B,C}
F -> {D}



Inzidenzmatrix (Kanten-Knotenmatrix)

Die Inzidenz (Incidence) ist die Beziehung zwischen einer Kante und einem Knoten. Endet eine Kante in einen Knoten, so sagt man, dass die Kante in den Knoten inzidiert. Ein Knoten heißt inzident zu einer Kante, wenn diese Kante zum Knoten führt.

Die Inzidenzmatrix B wird auch als **Kanten-Knotenmatrix** bezeichnet.

Dimension:

dim (B) = n * m n = Knotenanzahl, m = Kantenanzahl

Der Rang bei einem einfach zusammenhängenden Graph:

rang (B) = n - 1

Die Elemente der Matrix bei einem gerichtetem Graphen:

Bj,i = 1 der Bogen j vom Knoten i ausgeht
Bj,i = -1 der Bogen j im Knoten i endet
Bj,i = 0 kein Bogen vorhanden

Die Elemente der Matrix bei einem ungerichtetem Graphen:

Bj,i = 1, wenn die Kante j vom Knoten i ausgeht,
Bj,i = -1, wenn die Kante j im Knoten i endet und
Bj,i = 0, in allen übrigen Fällen.

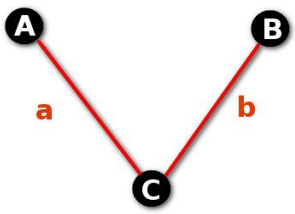
Als Beispiel ein ungerichteter Graph:

Die Inzidenzmatrix ist eine Kanten-Knotenmatrix: m * n Tabelle (Kanten-Anzahl * Knoten-Anzahl)

Möglichen Werte: 0 = keine Kante, 1 = Kante vorhanden

A ist inzident zu a
C ist inzident zu a
B ist inzident zu b
C ist inzident zu b

	A	B	C
a	1	0	1
b	0	1	1



Datenstruktur und Repräsentation der Beziehungsform

Für das Speichern eines Graphen müssen 3 Entscheidungen getroffen werden:

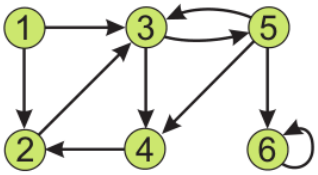
- als Beziehungsform die **Adjazenz** oder die **Inzidenz**
    a. **die Adjazenz** beschreibt die **Beziehung zwischen 2 Knoten**. Zwei Knoten heissen benachbart (adjazent), wenn eine Kante die beiden Knoten verbindet.
    b. Die **Inzidenz** ist die **Beziehung zwischen einer Kante und einem Knoten**.

- die **Repräsentationform**
    c. Liste
    d. Matrix

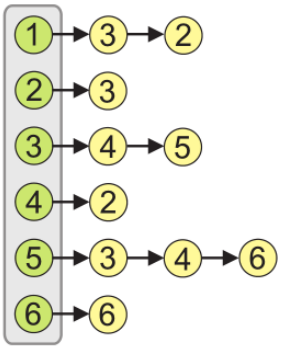
- **die Datenstruktur**
    e. bei einer **Auflistung** wird als Datenstruktur meistens eine **(doppelt) verkettete Liste** verwendet
    f. bei einer **Matrix** meistens ein **Array**

Beispiel-1: ein gerichteter Multigraph als Liste und Matrix

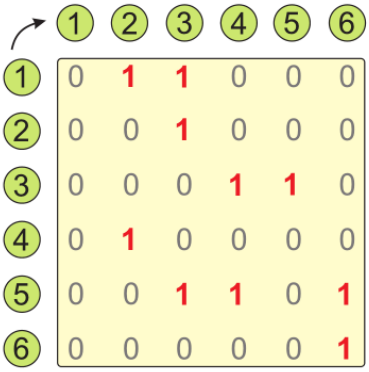
Ein gerichteter Multigraph:



Als **Adjazenz-Liste:**

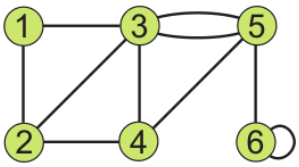


Und als **Adjazenzmatrix:**

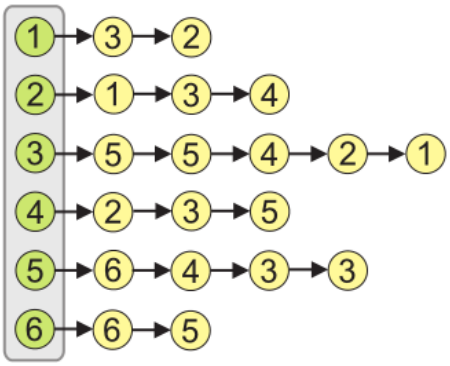


Beispiel-2: ein ungerichteter Multigraph als Liste und Matrix

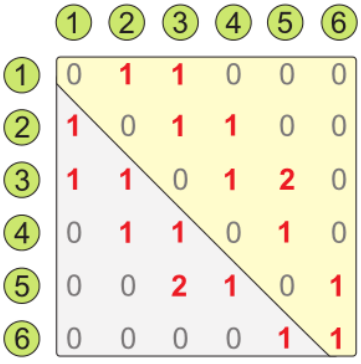
Ein gerichteter Multigraph:



Als **Adjazenz-Liste:**

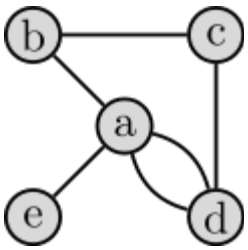


Und als **Adjazenzmatrix:**



Beispiel-3: ein ungerichteter Multigraph als verkettete Liste

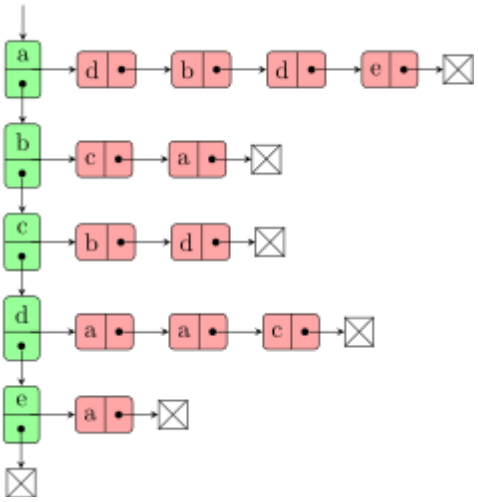
Graph



Adjazenzlisten

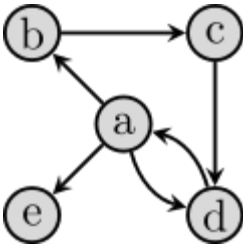
```
a: d, b, d, e
b: c, a
c: b, d
d: a, a, c
e: a
```

Datenstruktur



Beispiel-4: ein gerichteter Multigraph als verkettete Liste

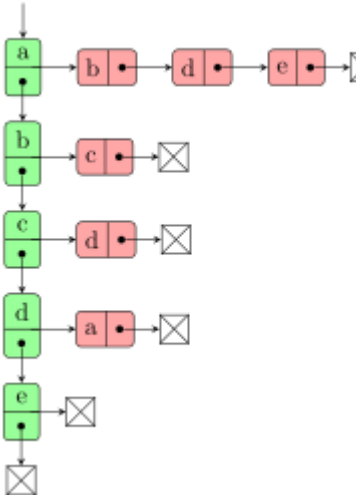
Graph



Adjazenzlisten

```
a: b, d, e
b: c
c: d
d: a
e:
```

Datenstruktur



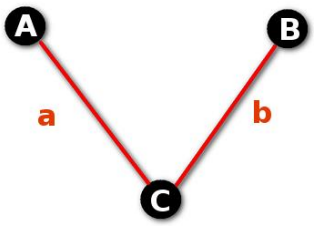
Beispiel-5: ein ungerichteter Multigraph als Matrix

Die **Adjazenzmatrix** ist eine **Knoten-Knotenmatrix**. n * n Tabelle (n=Knoten-Anzahl).

Möglichen Werte: 0 = keine Kante, 1 = Kante vorhanden

A,C sind adjazent A: C
B,C sind adjazent B: C
C,A sind adjazent C: A,B
C,B sind adjazent

	A	B	C
A	0	0	1
B	0	0	1
C	1	1	0



Beispiel-6: ein gerichteter Graph (Disgraph) als Matrix

Mögliche Werte-Variante 1: 1 = Vorgänger, -1 = Nachfolger, 0 = kein Bogen

Mögliche Werte-Variante 2: 1 = Bogen, 0 = kein Bogen

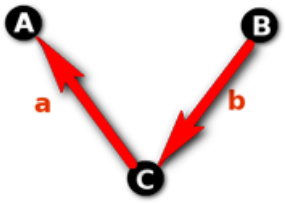
A ist Nachfolger von C

B ist Vorgänger von C

C ist Nachfolger von B

C ist Vorgänger von A

	A	B	C
A	0	0	-1
B	0	0	1
C	1	-1	0



Adjazenz-Auflistung (Nachbarschaftsliste)n:

B: C

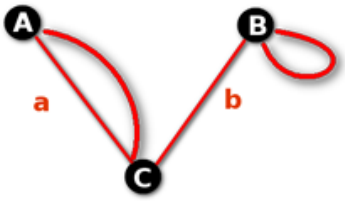
C: A

Beispiel-7: ein ungerichteter Multigraph als Matrix

Der Grad der Ecke wird eingetragen.

Die Schlinge hat den Grad 2 (siehe Abbildung von Beispiel-3).

	A	B	C
A	0	0	2
B	1	2	1
C	1	1	0



Beispiel-8: ein gerichteter Multigraph als verkettete Liste

Adjazenz-Listen:

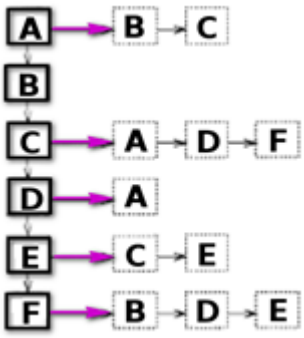
A: B,C

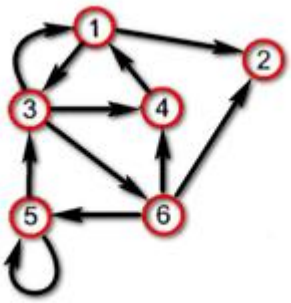
C: A,D,F

D: A

E: C,E

F: B,D,E





Bipartition

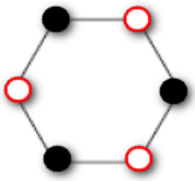
Bipartiter Graph (Paar Graph)

Ein Graph ist bipartit, wenn man seine Ecken so färben kann, dass jede Kante eine schwarze und eine weiße Ecke verbindet.
Die Ecken werden in zwei Klassen (Teilmengen) eingeteilt und meist E1 und E2 genannt. Die Kanten verbinden die Klassen miteinander. Möchte man die beiden Klassen herausheben, spricht man von einer Bipartition {E1, E2}.

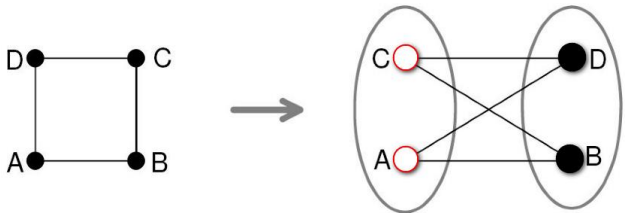
Der Graph im Beispiel 1 enthält 2 Arten von Ecken: Personen und Lebensmittel. Und nur zwischen Personen und Lebensmittel gibt es Kanten. Es ist nicht notwending, dass der Graph zusammenhängend ist (Kaffee ist ein isolierter Punkt).

Zum leichteren Verständnis kann Beispiel1 auch als bipartiter m-n Graph gennant werden, also bipartiter 5-8 Graph.

Beispiel 2 für einen bipartiten 3-3 Graphen:

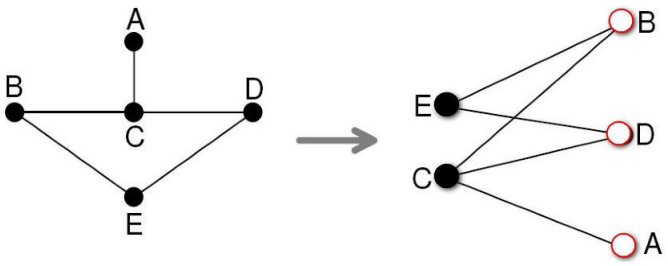


Beispiel 3 ein bipartites Quadrat. Zum besseren Erkennen verformt und abwechselnd gefärbt:



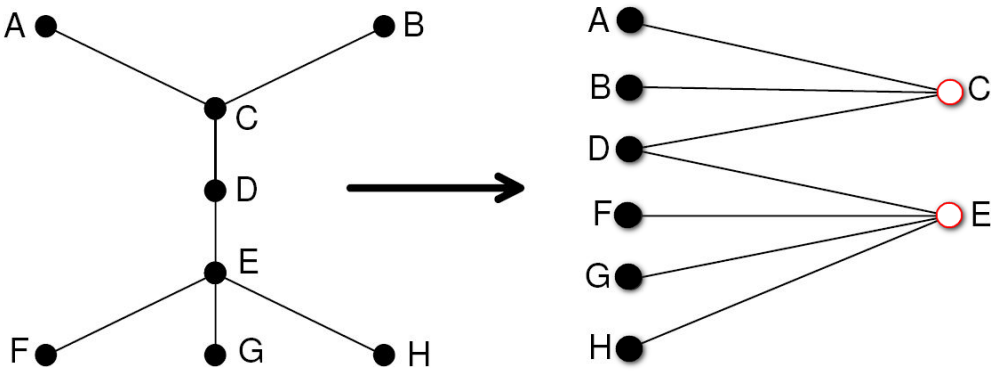
Kreise mit gerade Eckenanzahl sind bipartit.
Kreise mit ungerader Eckenanzahl sind nicht bipartit.

Beispiel 4, ein bipartiter 2-3 Graph:

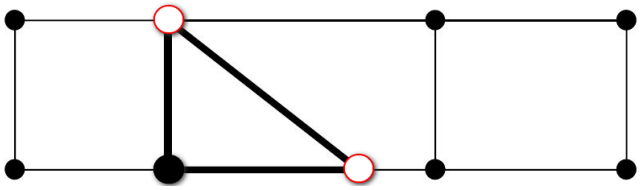


Bäume sind bipartit.

Beispiel 5, ein bipartiter 6-2 Baum:

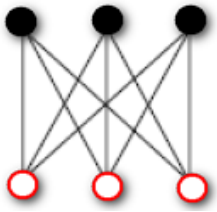


Ein Graph, der einen ungeraden Kreis enthält, kann nicht bipartit sein. Im Beispiel ein Dreieck (mit 3 Ecken):



Vollständig bipartiter Graph

Vollständig bipartit sind diese Graphen, wenn jede Ecke der einen Klasse mit jeder Ecke der anderen Klasse durch genau eine Kante verbunden ist.



Hypergraph

Bei Hypergraphen verbindet eine Kante mehr als 2 gleichzeitig. Sie sind schwer darstellbar: Bei dünnen Graphen (der Graph enthält nur wenig Kanten) zeichnet man eine Menge von Punkten, die den Knoten entsprechen. Die zu einer Hyperkante gehörigen Punkte werden dann durch eine geschlossene Linie umkreist, die somit die Teilmenge der zu ihr gehörenden Knoten innerhalb aller Knoten angibt.

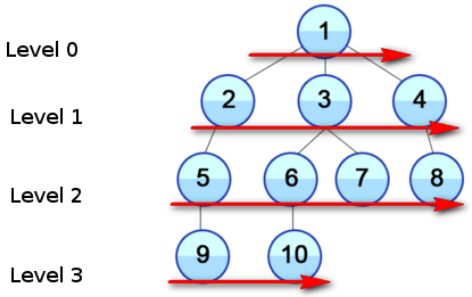
Travesierung (Traversal)

Das Travesieren ist das systematische Durchlaufen aller Knoten und Kanten eines Graphen.

Breitensuche (Breadth-First Search, BFS)

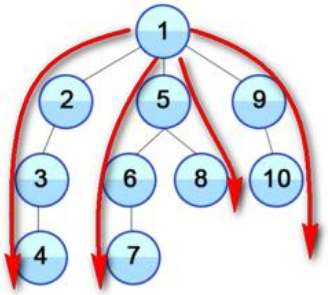
Besuche zuerst alle Knoten einer Ebene (alle Knoten mit gleicher Entfernung). Dann die nächste Ebene.

Als Beispiel ein Baum der Größe 10 und der Tiefe 4



Tiefensuche (Depth-First Search, DFS)

Folge jedem Pfad bis zur maximalen Tiefe. Als Beispiel ein Baum der Größe 10 und der Tiefe 4:



Baum

Ein Baum ist ein zusammenhängender Graph ohne Kreis. und Mehrfachverbindungen. Je zwei Knoten sind durch genau einen Weg verbunden. Die Kanten des Baumes werden Zweige (Branches) genannt und die Endknoten heißen Blätter.

Jeder Baum mit n Ecken hat genau n-1 Kanten.

Bäume bestehen nur aus Brücken (können also kein Multigraph sein). Beim Entfernen einer Kante zerfällt der zusammenhängende Baum in Komponenten.

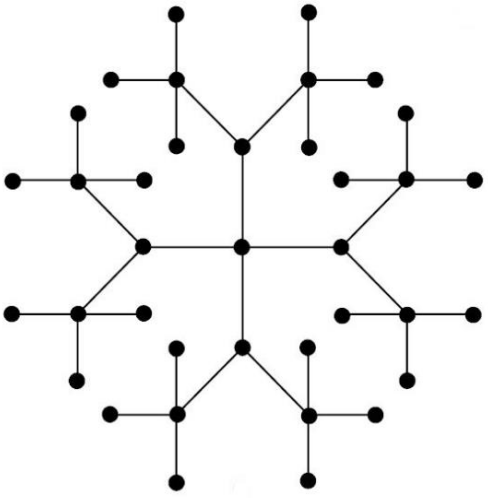
Eigenschaften eines Baums:

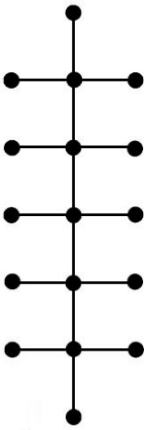
1) es gibt keinen Kreis (Zyklus), keine Schlinge und keine Mehrfachverbindungen
2) von jedem Knoten gibt es genau einen Weg zu einem anderen
3) G ist ein zusammenhängender Graph mit n-1 Kanten (n = Knotenanzahl).

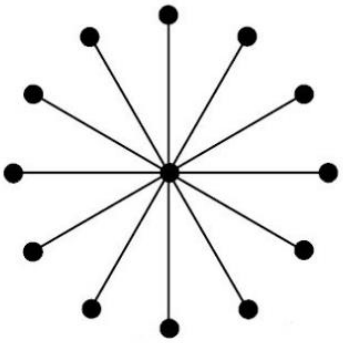
Ein Wald besteht aus einem oder mehreren Bäumen.

Ein Baum kann, muss aber nicht ein Wurzelbaum sein.

5 Beispiele für einen Baum:





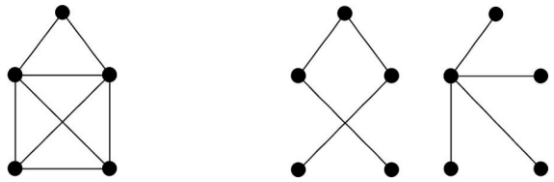


Kein Baum, da ein Kreis vorhanden ist:

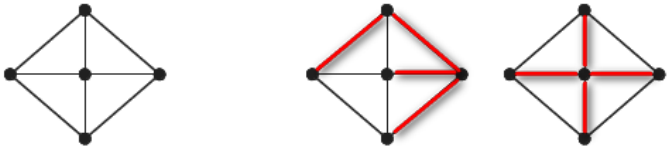
Spannbaum (Spanning Tree)

Der Spannbaum ist ein Teilgraph, der alle Knoten des Graphens enthält. Der Graph muss somit zusammenhängend sein.

Beispiel 1:



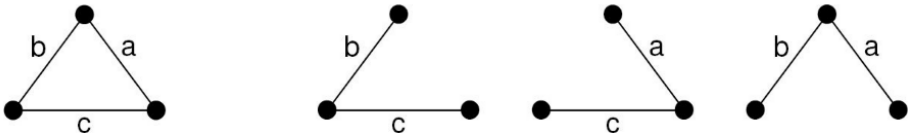
Beispiel 2:



Bei einem vollständigen Graph lässt sich die Anzahl der möglichen Spannbäume mit der Formel von Caley (Arthur Coley, 1821 bis 1895) errechnen.

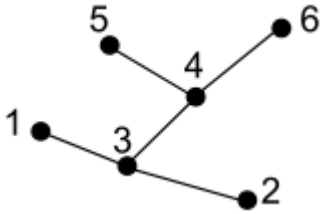
Die Anzahl möglicher Spannbäume bei einem vollständigen Graphen mit n Ecken: n^{n-2}

Beispiel 3, ein vollständiger Graph und seine 3 Spannbäume:

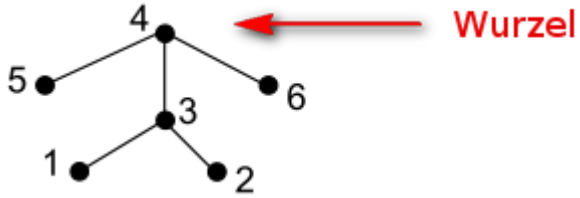


Gewurzelter Baum

Ein Baum ohne ausgezeichneter Wurzel:



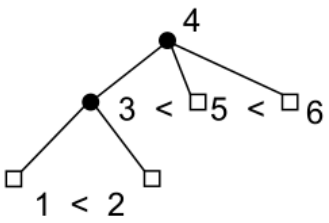
Punkt4 als Wurzel ausgezeichnet:



Die Kinder der Wurzel sind Punkt5, Punkt3 und Punkt6. Von diesen 3 Knoten sind Punkt5 und Punkt6 Endknoten (Blätter). Punkt3 wird innerer Knoten genannt, da er Kinder besitzt.

Geordneter Wurzelbaum

Liegt in den Knoten eine bestimmte Beziehung, zum Beispiel eine kleiner-gleich-Relation, dann wird der Wurzelbaum als geordnet betrachtet:



Suchbaum

Ein Suchbaum hat diese Eigenschaften:

1. Ein Suchbaum ist ein Baum, in dem ein Knoten als Wurzel ausgezeichnet ist.
2. Die Suchtiefe eines Knotens ist seine Weglänge zur Wurzel.
3. Die Suchtiefe eines Suchbaums ist seine maximale Weglänge (der längste Weg = der entfernteste Knoten).
   - ❖ Die Nachbarn eines Knoten, die eine größere Suchtiefe haben als dieser, heißen Kinder dieses Knoten.
   - ❖ Der (eindeutige) Nachbarn eines Knoten, der eine geringere Suchtiefe hat als dieser, heißt Elternteil dieses Knoten.
4. Ein Blatt ist ein Knoten mit Grad 1 (aber niemals die Wurzel!), also ein Knoten ohne Kinder.

Binärbaum (Binary Tree)

Ein Binärbaum (Binary Tree) ist ein Suchbaum, in dem jeder Knoten (Node) maximal zwei Kinder (Childs) hat.
Er ist ein geordneter Wurzelbaum der Ordnung 2 (maximal 2 Kindener). Bei mehr als maximal 2 Kindern wird von Vielwegbäumen gesprochen.

Wurzel (Root)

Der Startpunkt wird Wurzel (Root) genannt und befindet sich in Ebene (Level) 1.

Zweigknoten

Nach der Wurzel folgen die Knoten. Jeder Knoten (Node) kann 0 bis 2 Kinder haben:

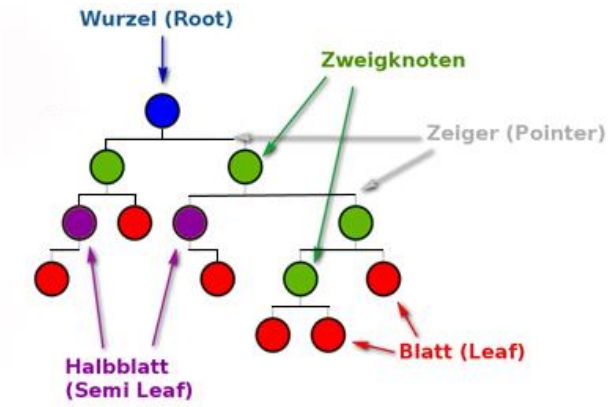
0 Kinder ein Knoten ohne Kinder wird Blatt (Leaf) genannt

1 Kind ein Knoten mit nur einem Kind wird Halbblatt (Semi Leaf) genannt
Halbblätter gibt es nur im Binärbaum und nicht zB im B-Tree (Balanced Tree)

2 Kinder ein Knoten mit 2 Kindern ist ein innerer Knoten und wird auch Zweigkonten (Branch Node) genannt.

Blatt (Leaf)

Ein Knoten ohne Kinder wird Blatt (Leaf) genannt. Ein Blatt ist die unterste und somit letzte Ebene des jeweiligen Astes. Bei einem **vollständigen Binärbaum**, hat jeder Blattknoten die gleiche Tiefe.



AVL Baum

Der AVL-Baum ist ein binärer Suchbaum mit der Zusatzeigenschaft, dass sich an jedem Knoten die Höhe der beiden Teilbäume um höchstens eins unterscheidet.

Diese Eigenschaft lässt seine Höhe nur logarithmisch mit der Zahl der Schlüssel wachsen und macht ihn zu.

einem balancierten binären Suchbaum

B-Tree

Wofür das B in B-Tree steht ist unbekannt. Am häufigsten steht es für Balanced Tree. Manchmal nach seinem Erfinder Rudolph Bayer (Bayer Tree).

Der B-Tree ist ein vollständig balanzierter binärer Baum. Siehe B+Tree Seite607 .

Linux: Bash

Kapitel 2

Linux: Bash

Shell

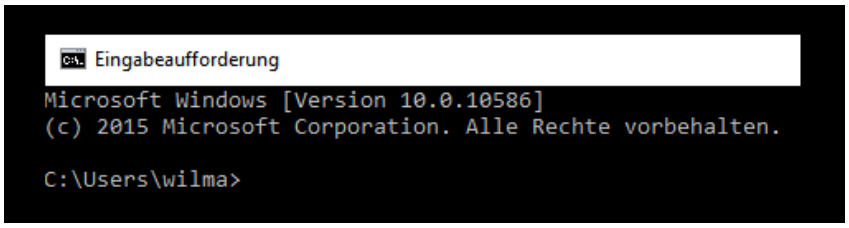
Eine Shell stellt ein Interface zu Programmen dar. Die Shell läuft üblicherweise im Textmodus (ohne GUI) und ist optimiert für die Eingabe auf einer Tastatur (Konsole). Das Interface ist nicht auf die Tastatur beschränkt, auch externe Programme können mit der Shell kommunzieren.

Mehrere Kommandos können zusammengefasst und als Batch-Datei abgelegt werden. Die meisten Shells sind programmierbar und bieten eine eigene Scriptsprache an.

Üblicherweisen werden die Anweisungen (per Skript oder Terminal) von einem Interpreter zeilenweise abgearbeitet.

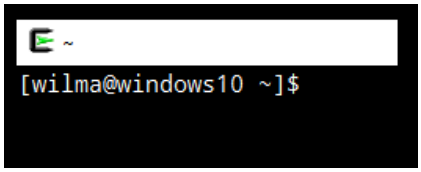
Die Windows-Shell:

```
C:\Users\wilma>
```



Die csh unter Cygwin:

```
[wilma@windows10 ~]$
```



Command Line Interface (CLI)

Die Eingabe und Ausführung von Shell-Kommandos erfolgt zeilenweise. Der Prompt zeigt die Stelle, an der mit der Eingabe begonnen werden kann. Dieser Teil einer Shell wird **Command Line Interface** (CLI) genannt. Die Command Line wird auch **Befehlszeile** genannt (Bereich neben dem Prompt, in dem Kommandos getippt werden können). Die Shell arbeitet nur im Textmodus. Es gibt keine GUI (Graphic User Interface), Icons und Panels.

Command Line Interpreter (CLI)

Das Verarbeiten des Inputs (Anweisungen, Schleifen, Variablen, Bedingungen, ...) wird der Shell vom **Command Line Interpreter** (CLI) durchgeführt.

Command Line User Environment (CLOE)

Die Shell im gesamten ist ein **Command Line User Environment** (CLOE).

Shell Familie

Die bekanntesten Shells:

dash Debian Almquist Shell.
Bei Debian die Standard-Shell für Shellskripte (sie ist kleiner und schneller als bash).

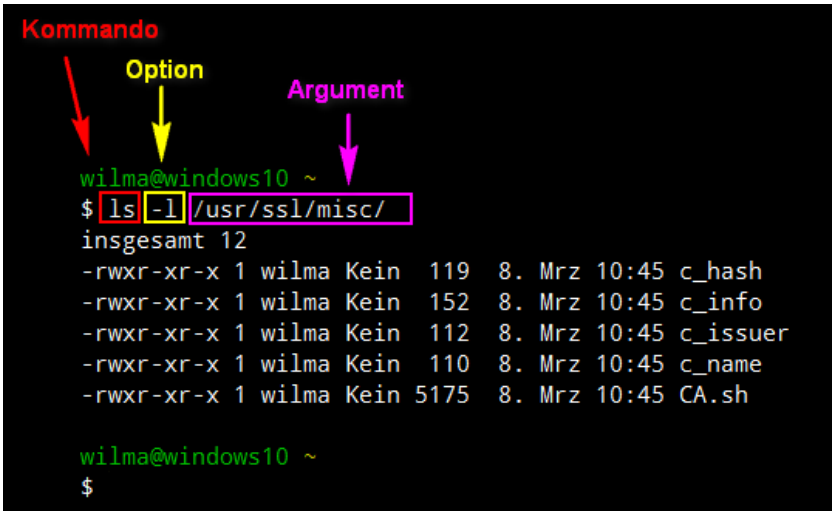
fish Friendly Interactive SHell
Syntax Highlighting; währenden dem Tippen Vorschläge und Korrekturen anzeigen;
in der Completition-Anzeige mit den Cursor-Tasten navigieren/auswählen; Skripte haben Modula-2 Syntax

csh C SHell Korn

ksh Korn SHell

Um fish zur Standard-Shell zu machen: `chsh -s /usr/bin/fish`
Bash als Standard-Shell: `chsh -s /bin/bash`

Parameter



Ein Programm (bzw Anweisung) kann mit optionalen Parameter gestartet werden. Parameter sind durch ein Leerzeichen getrennt.

Es gibt 2 Arten von Parameter:

- Optionen
- Argumente

Optionen werden ein Minus vorangestellt.

Beispiel:

```
ls -l /usr/ssl/misc
```

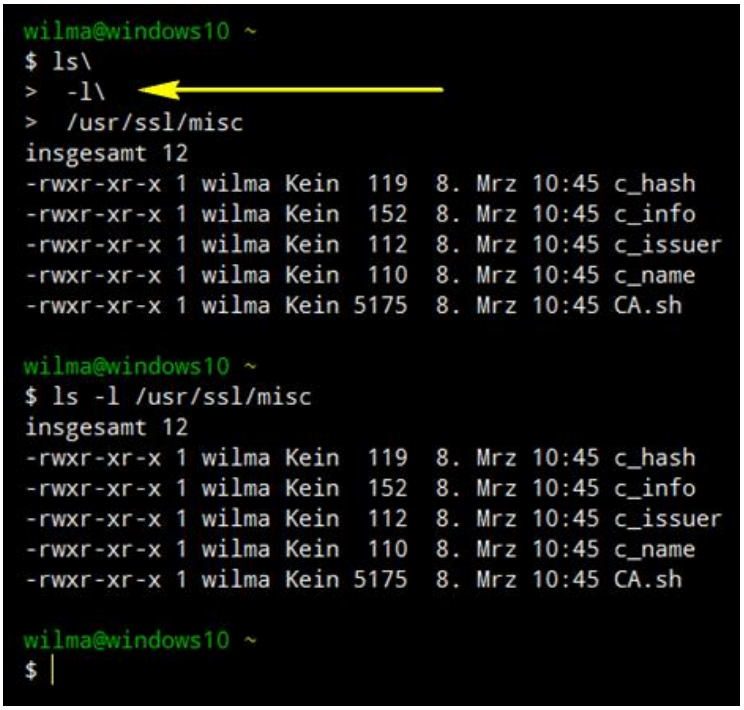
ls ist die Anweisung. **-l** ist die Option und **/usr/ssl/misc** ist das Argument.

Der Command Line Interpreter der Shell führt Zeile für Zeile aus. Dies gilt nicht nur für Eingabe über Console, sondern auch für Batch-Files.
Die Zeile wird in Wörter zerlegt (Trennzeichen ist das Leerzeichen). Alle Wörter sind für eine spätere Verwendung wieder aufrufbar (Ausnahme sind Sktipte).
Siehe Beispiel: Der Word Designator !-1:1 enthält das 2. Wort (Zählung beginnt bei 0) vom letzten Aufruf (-1).

```
wilma@windows10 ~
$ file !-1:2
file /usr/ssl/misc
/usr/ssl/misc: directory
```

Backslash im Input

Bei einem **backslash newline** wird eine zweite Prompt-Zeile erzeugt.
Das Promptzeichen ändert sich von einem $ auf ein >.
Die Shell wartet auf die Eingabe weiterer Anweisungen. Der Backslash kann an beliebiger Position gesetzt werden.



Beispiel:
Die ls Anweisung mit zwei Parameter:

```
wilma@windows10 ~
$ ls -l /usr/ssl/misc
```

Mit zwei backslash wurde der Input optisch auf drei Zeilen erweitert:

```
wilma@windows10 ~
$ ls\
>  -l\
>  /usr/ssl/misc
insgesamt 12
```

Eingabeaufforderung (Prompt)

Der Prompt zeigt genau auf die Stelle in der **command line** (Befehlszeile) einer Shell, an der Befehle eingegeben werden können. Der Inhalt und das Aussehen des Prompts hängt von der jeweiligen Shell und der Konfiguration ab.

Häufig sieht man die Raute #, das Dollarzeichen $, das Greater-then > und auf älteren System das Prozentzeichen %.

- # als Superuser eingeloggt
- $ normaler User (seltener ein %)
- > die Shell wartet auf die Eingabe weiterer Anweisungen. Die Formatierung ist in der Variable $PS2

Für gewöhnlich wird der Prompt mit zusätzlichen Informationen erweitert. Oft mit dem gegenwärtigen Username, dem Namen des Rechners und dem gegenwärtigen Pfad:

benutzername@*rechner*:~$

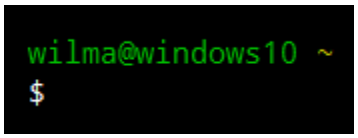
Die Tilde (~) steht für das Homeverzeichnis des jeweiligen Users.
Mit der Anweisung **pwd** (print working directory) wird das gegenwärtige Verzeichnis angezeigt.

Das Dollarzeichen steht für einen normalen User, die Raute (#) für den Superuser (root), und bei einem > wartet die Shell auf weitere Eingabe.

Zum Beispiel:

```
wilma@windows10 ~
$
```

wilma ist der Username, **windows10** ist der Name des Rechners.
Tilde gibt an, dass man sich gerade im Homeverzeichnis befindet.
zeigt einen Login als normaler User und die Stelle für die Texteingabe.



Die $

Der Input wird von der Shell bis zum ersten **unquoted newline** gelesen, interpretiert und sofort ausgeführt (newline = Zeilenumbruch = Enter-Taste).

Prompt-Konfiguration

In der Variable $PS1 ist der Standard-Prompt definiert. Um den Inhalt von $PS1 anzuzeigen:

```
tom@sanctum:~$ echo $PS1
\u@\h:\w\$
```

$PS1 unter Cygwin:

```
wilma@windows10 ~
$ echo $PS1
\[\e]0;\w\a\]\n\[\e[32m\]\u@\h \[\e[33m\]\w\[\e[0m\]\n\$
```

$PS2 unter Cygwin:

```
wilma@windows10 ~
$ echo $PS2
>
```

Escape-Sequenzen für Prompt

\a	ASCII Bell (Glockenton/Piep)
\d	date im Format "Weekday Month Date" Format. Zum Beispiel: Tue May 26
\D{format}	Format entspricht der Funktion strftime() bei Leerstring wird die **locale-specific** time verwendet.
\e	ASCII escape character (033)
\h	the hostname up to the first `.'
\H	the hostname
\j	the number of jobs currently managed by the shell
\l	the basename of the shell's terminal device name
\n	newline
\r	carriage return
\s	the name of the shell, the basename of $0 (the portion following the final slash)
\t	the current time in 24-hour HH:MM:SS format
\T	the current time in 12-hour HH:MM:SS format
\@	the current time in 12-hour am/pm format
\A	the current time in 24-hour HH:MM format
\u	the username of the current user
\v	the version of bash (e.g., 2.00)
\V	the release of bash, version + patch level (e.g., 2.00.0)
\w	the current working directory, with $HOME abbreviated with a tilde (uses the value of the PROMPT_DIRTRIM variable)
\W	the basename of the current working directory, with $HOME abbreviated with a tilde
\!	the history number of this command
\#	the command number of this command
\$	if the effective UID is 0, a #, otherwise a $
\nnn	the character corresponding to the octal number nnn
\\	a backslash
\[	begin a sequence of non-printing characters, which could be used to embed a terminal control sequence into the prompt
\]	end a sequence of non-printing characters

History

Mit der Anweisung history werden die letzten Eingaben angezeigt.
Die Konfiguration befindet sich in configuration **~/.bashrc** und in **/etc/bash.bashrc**).

Ohne Parameter werden alle Einträge im File history gezeigt:

```
wilma@windows10 /
$ history
```

Um die letzten 3 Einträge zu zeigen:

```
wilma@windows10 /
$ history 3
  157  ls -l /usr/sbin/backup
  158  file etc/profile
  159  history 3
```

Mit str+r kann nach einem String in der History gesucht werden. Die letzte Verwendung wird angezeigt.
Suche nach der letzten Verwendung von 168:

```
wilma@windows10 ~
(reverse-i-search)`168': whois 192.168.0.228
```

Über Word designators kann auf jede Zeile und jedes Wort der History zugegriffen werden. Diese Platzhalter werden vom CLI durch die echten Werte ersetzt.

Das Kommando **ls** ist das erste Wort, die Option **-l** ist das zweite Wort und der Pfad (Argument) das dritte Wort:

```
wilma@windows10 ~
$ ls -l /usr/ssl/misc
```

Der Word Designator **!-1:1** enthält das 2. Wort (Zählung beginnt bei 0) vom letzten Aufruf (-1), also vom letzten Eintrag in der History:

```
wilma@windows10 ~
$ file !-1:2
file /usr/ssl/misc
/usr/ssl/misc: directory
```

!! wiederholt die (gesamte) letzte Zeile:

```
wilma@windows10 ~
$ !!
touch blub.txt

wilma@windows10 ~
$ sudo !!
sudo touch blub.txt
```

Mit !! wird die letzten Anweisung wiederholt (identisch mit !-1):

```
wilma@windows10 /
$ !!
history 3
  158  file etc/profile
  159  history 3
  160  history 3
```

Die Einträge in der History sind nummeriert. Mit !*Nummer* kann dieser Eintrag noch einmal ausgeführt werden.

```
wilma@windows10 /
$ !158
file etc/profile
etc/profile: ASCII text
```

Die Einträge in der History sind nummeriert. Mit !-*Nummer* wird von hinten die Zählung begonnen (n Zeilen zurück).
Um die vorletzte Anweisung zu wiederholen:

```
wilma@windows10 /
$ !-2
```

Mit !*string* wird die letzte Anweisung wiederholt, die mit *string* **beginnt**.
Um die letzte Anweisung zu wiederholen, die mit fil beginnt:

```
wilma@windows10 /
$ !fil
file etc/profile
```

Um die letzte gzip-Anweisung zu wiederholen:

```
wilma@windows10 /
$ !gzip
```

Mit !?string wird die letzte Anweisung wiederholt, die string **enthält**.
Um die letzte Anweisung zu wiederholen, die den String **etc** enthält:

```
wilma@windows10 /etc/asciidoc
$ !?etc
cd /etc/asciidoc/
```

Mit !$ wird der letzte Parameter wiederholt (identisch mit !!$):

```
wilma@windows10 ~
$ touch -c foo.txt

wilma@windows10 ~
$ touch !$
touch foo.txt
```

Mit !* werden alle Parameter wiederholt:

```
wilma@windows10 ~
$ touch -c foo.txt

wilma@windows10 ~
$ touch !*
touch -c foo.txt
```

Mit !:*Nummer* wird der n. Parameter ausgegeben.
Die Zählung beinhaltet das Kommando und alle Parameter (identisch mit !!:*Nummer*)

```
wilma@windows10 ~
$ ls -l -i -s -a /etc/defaults/
```

Ergibt:

```
!:0 ist ls
!:1 ist -l
!:2 ist -i
!:3 ist -s
!:4 ist -a
!:5 ist /etc/defaults/
```

Optionen

-c	History löschen (c=clear)
-a	bisherige Session-Einträge jetzt in HISTFILE hinzufügen (a=append) (Und nicht erst bei Session-Ende.)
-d *Nummer*	einzelne Einträge in der History löschen. *Nummer* steht für den n. Eintrag in der Liste.
-n	Append the history lines not already read from the history file to the current history list. These are lines appended to the history file since the beginning of the current Bash session.
-r	Read the history file and append its contents to the history list.
-w	Write out the current history list to the history file.
-p	Perform history substitution on the args and display the result on the standard output, without storing the results in the history list.
-s	The args are added to the end of the history list as a single entry.

Variablen

HISTFILE Name der Datei, in der die Einträge gespeichert werden
Meistens ~/.bash_history

HISTSIZE maximale Einträge für die Session

- `0` History deaktiviert
- `n` History ist auf maximal n Einträge beschränkt

HISTCONTROL

`ignoreboth` ab jetzt gibt es jeden Eintrag maximal einmal (keine Duplikate mehr).
(keine Auswirkung auf die bestehenden Einträge).

`erasedups` bereits vorhandene Einträge werden immer wieder gelöscht
Nur immer der jeweils neueste (letzte) Eintrag ist sichtbar.

`ignorespace` Zeilen, die mit einem Leerzeichen beginnen, werden ignoriert

HISTFILZESIZE maximale Einträge im File ~/.bash_history
default ist 500

HISTIGNORE Einträge, die nicht geschrieben werden sollen.
Zum Beispiel nicht **ls** und **history**:

```
HISTIGNORE='ls:history
```

type

Type zeigt, ob die Anweisung ein internes Shell-Kommando oder ein eigenes Programm ist.

Kommando pwd (print working directory) ist eine interne Shell-Funktion:

```
wilma@windows10 ~
$ type pwd
pwd ist eine von der Shell mitgelieferte Funktion.
```

Mit der Option -t wird statt einem Satz, nur ein Wort zurückgeliefert. Mögliche Return-Werte:

1. alias shell alias
2. keyword reserved word
3. function shell function
4. builtin shell builtin
5. file disk file

Kommando pwd (print working directory) ist eine built-in Shell-Funktion:

```
wilma@windows10 ~
$ type -t pwd
builtin
```

Beispiel für ein Keyword:

```
wilma@windows10 ~
$ type if
if Ist ein reserviertes Schlüsselwort der Shell.
```

Beispiel für ein File:

```
wilma@windows10 ~
$ type ls
ls is hashed (/usr/bin/ls)
```

file

Mit der Anweisung file läßt sich die Art der Datei festellen.

Ein Plain-Text File:

```
$ file doc-base/copyright
doc-base/copyright: ASCII English text
```

Ein XML-Plain-Textfile:

```
$ file doc-base/install-docs.html
doc-base/install-docs.html: XML 1.0 document text
```

Ein Verzeichnis (mit der Endung .html)

```
$ file doc-base/doc-base.html
doc-base/doc-base.html/: directory
```

Eine komprimierte gzip-Datei:

```
$ file doc/doc-base/changelog.gz
doc-base/changelog.gz: gzip compressed data
```

Eine PDF-Datei:

```
$ file shared-mime-info/shared-mime-info-spec.pdf
shared-mime-info/shared-mime-info-spec.pdf: PDF document, version 1.4
```

Keyboard Shortcuts

Die default Tastenkombinationen der Bash (Emacs-Modus).

Um auf vi umzustellen: `$ set -o vi`

Auf Emacs umzustellen: `$ set -o emacs`

Konvention:

Alt + a	Alt-Taste drücken, halten und die a-Taste drücken die Alt-Taste wird auch **Meta Key** genannt.
smallword	ein Small Word besteht nur aus {a-z, A-Z, 0-9} /etc/defaults/etc/fish sind 8 Wörter
bigword	bei einem Big Word ist nur ein **Space Character** ein Trennzeichen. /etc/defaults/etc/fish ist ein 1 Wort

Undo & Cancel

Alt + r	Cancel the changes and put back the line as it was in the history (revert).
ctrl + _	Undo

Cursor

Ctrl + a	Zeilenanfang	identisch mit der Taste Pos1
Ctrl + e	Zeilenende	identisch mit der Taste Ende
Ctrl + p	Previous command	
Ctrl + n	Next command (Down arrow)	
Alt + b	1 smallword nach links	
Alt + f	1 smallword nach rechts	
Ctrl + f	1 Zeichen nach rechts	
Ctrl + b	1 Zeichen nach links	
Ctrl + xx	Toggle between the start of line and current cursor position	

Prozesse

Ctrl + C	Interrupt/Kill whatever you are running (SIGINT)
Ctrl + l	Clear the screen
Ctrl + s	Stop output to the screen (for long running verbose commands) Then use PgUp/PgDn for navigation
Ctrl + q	Allow output to the screen (if previously stopped using command above)
Ctrl + D	Send an EOF marker, unless disabled by an option, this will close the current shell (EXIT)
Ctrl + Z	Send the signal SIGTSTP to the current task, which suspends it. To return to it later enter fg 'process name' (foreground).

Completion

TAB Completion für Kommandos und Dateinamen.

Beispiel:

```
wilma@windows10 ~
$ ech⇤⇥
```

Da es bei **ech** nur eine Möglichkeit gibt, wird sofort ergänzt:

```
wilma@windows10 ~
$ echo
```

Gibt es keine eindeutige Möglichkeit, muss ein zweites mal der Tabulator gedrückt werden:

```
wilma@windows10 ~
$ ech⇤⇥⇤⇥
```

Erst nach dem zweiten Tab werden Vorschläge aufgelistet:

```
wilma@windows10 ~
$ ec
echo      echo    ecpg
```

Löschen

Ctrl + L Bildschirm löschen (gegenwärtige Zeile nicht mitlöschen)

Alt + Del lösche das Wort vor dem Cursor.

Alt + d lösche das Wort nach dem Cursor (Wort getrennt Leer- und Sonderzeichen wie / und -)

Ctrl + d Zeichen »unter« dem Cursor löschen

Ctrl + h Zeichen links vom Cursor löschen (Backspace)

Cut & Paste

Ctrl + w das Wort vor dem Cursor ausschneiden (kopieren & löschen)

Ctrl + k alles nach dem Cursor ausschneiden

Ctrl + u alles vor dem Cursor ausschneiden

ctrl + y das letzte Ausgeschnittene einfügen

Tauschen (swap)

Alt + t Swap current word with previous

Ctrl + t Swap the last two characters before the cursor (typo).

Esc + t Swap the last two words before the cursor.

Character Case

Alt + u UPPER capitalize every character from the cursor to the end of the current word.

Alt + l Lower the case of every character from the cursor to the end of the current word.

Alt + c Capitalize the character under the cursor and move to the end of the word.

History

Ctrl + r bereits beim Tippen wird in der History gesucht und entsprechende Vorschläge angezeigt

Ctrl + p Vorherige Anweisung in der History

Ctrl + n Nächste Anweisung in der History

Ctrl + s Zurück zur letzten Anweisung

Beware to not execute it from a terminal because this will also launch its XOFF.

Ctrl + o Anweisung ausführen, die per Ctrl+r oder Ctrl+s gefunden worden ist

Ctrl + g **History Searching Mode** verlassen

History mit Word Designators

!! letzte (ganze) Zeile wiederholen

```
wilma@windows10 ~
$ mkdir foo

wilma@windows10 ~
$ sudo !!
sudo mkdir foo
```

!abc die letzte Anweisung, die mit abc beginnt. Nach dem Enter wird die Zeile ausgeführt.

```
wilma@windows10 ~
$ !sud
sudo mkdir foo
```

!abc:p die letzte Anweisung, die mit abc beginnt.
Nach dem Enter wird die erzeugt Zeile nur angezeigt und nicht ausgeführt (p=print)

!$ von der letzten Zeile, **das letzte Wort** (identisch mit ALT+.)

Wird auch ausgeführt, wenn die letzte Anweisung aus nur einem Wort besteht.

```
wilma@windows10 ~
$ mkdir -v foo2
mkdir: Verzeichnis 'foo2' angelegt

wilma@windows10 ~
$ rmdir !$
rmdir foo2
```

ALT + . von der letzten Zeile, **das letzte Wort** (identisch mit !$)

!* **alle Parameter** von der letzten Zeile (also alles ohne dem 1. Wort).

Wenn die letzte Anweisung nur aus einem Kommando (also nur 1 Wort ohne Parameter) besteht, ist der Platzhalter leer

```
wilma@windows10 ~
$ mkdir -v foo3
mkdir: Verzeichnis 'foo3' angelegt

wilma@windows10 ~
$ rmdir !*
rmdir -v foo3
rmdir: Verzeichnis wird entfernt, 'foo3'
```

!^ von der letzten Zeile, der **erste Parameter** (also das 2. Wort).

```
wilma@windows10 ~
$ ls -l /etc/defaults/etc/fish/
insgesamt 4
-rw-r--r-- 1 wilma Kein 1885 30. Aug 2015  config.fish

wilma@windows10 ~
$ ls !^
ls -l
```

^abc^xyz die (ganze) letzte Anweisung wiederholen.

Ersetze vorher die Zeichenkette **abc** durch **xyz**. Es wird nur der 1. Treffer ersetzt:

```
wilma@windows10 ~
$ mkdir mamma

wilma@windows10 ~
$ ^a^i
mkdir mimma
```

touch

Verändert die Last-Accessed-Time und die Last-Modification-Time von Files (nicht die Last-Change-Time).

Wird die Anweisung bei nicht vorhandenen Files verwendet, werden diese neue angelegt. Wird keine bestimmte Zeit angegeben, wird die gegenwärtige System-Zeit verwendet.

```
touch [option] filename(s)
```

Einige Implementierungen ändern nur die Zeit der letzten Modifikation (nicht die Last Access-Zeit).

Unterschied von Modify und Change:

Modify	the last time the file was modified (content has been modified)
Change	the last time meta data of the file was changed (e.g. permissions)
Access	the last time the file was read

Parameter

-a ändert nur die Access Time

-r kopiert die Datumangaben von einem Referenz-File, auf andere Files

-c no-create (wenn eine Datei nicht existiert, wird keine neue Datei angelegt)

-d Zeitformat im human readable Format (Mächtigkeit hängt von der Implementierung ab).

Ändert Access Time und Modification Time.

-t Zeitformat als Kommazahl (WORD): [[CC]YY]MMDDhhmm[.ss]

```
CC  The first two digits of the year (the century).
YY  The second two digits of the year.
MM  The month of the year, from 01 to 12.
DD  the day of the month, from 01 to 31.
hh  The hour of the day, from 00 to 23.
mm  The minute of the hour, from 00 to 59.
SS  The second of the minute, from 00 to 61.
```

Ändert Access Time und Modification Time.

-m ändert nur die modification time

Beispiele

Nur mit Angabe einer Datei, wird die Access Time dieser Datei auf die gegenwärtige Zeit geändert.

```
touch foo.txt
```

Wenn die angegebene Datei foo.txt nicht existiert, wird die Datei angelegt.

Mit dem Parameter -c wird in so einem Fall keine neue Datei angelegt:

```
touch -c foo.txt
```

Die Dateien foo.txt und foo2.txt gleichzeitig ändern:

```
touch -c foo.txt foo2.txt
```

Nimm als Referenz die Zeit von der Datei foo.txt, und wende sie auf foo2.txt an:

```
touch -r foo.txt foo2.txt
```

Einige Implementierungen bieten als Parameter -F (forward) und –B (backward) an.

Datum von file6 minus 30 Sekunden und wende die Zeit auf file7 an:

```
touch -r file6 -B 30 file7
```

Mit -t und -d die Datei foo.txt auf den 31. Jänner 2006, 08:40 Uhr (und 26 Sekunden)

```
touch -t 200601310846.26 foo.txt
touch -d '2006-01-31 08:46:26' foo.txt
```

Mit -d auf den 1. Mai 2005, 10:22 Uhr und 22 Sekunden:

```
touch -d '1 May 2005 10:22' foo.txt
```

Wird nur eine Uhrzeit angegeben,wird der gegenwärtige Tag verwendet:

```
touch -d '14:24' file9
```

Ein rekursiver Aufruf ist nur über den Umweg von **find** möglich:

```
find . -exec touch {} +
```

Mit **ls** die Zeit der letzten Modifikation anzeigen:

```
ls -l foo.txt
```

Mit **stat** die drei Zeitangaben Access, Modify und Change anzeigen:

```
stat foo.txt
```

Mit **stroke** die Erstellzeit angeben:

```
stroke foo.txt
```

Unterschied mtime, ctime, atime, relatime, noatime

Access - the last time the file was read

Modify - the last time the file was modified (content has been modified)

Change - the last time meta data of the file was changed (e.g. permissions)

Writing to a file changes its **mtime** **and** **ctime**, while **reading** a file changes its **atime**.

As a result, on a POSIX-compliant system, reading a file causes a write, which has been criticized. This behaviour can usually be disabled by adding a mount option in /etc/fstab.

However, turning off atime updating breaks POSIX compliance, and some applications, notably the mutt mail reader (in some configurations), and some file usage watching utilities, notably tmpwatch. In the worst case, not updating atime can cause some backup programs to fail to back up a file.

Linux kernel developer Ingo Molnár called atime "perhaps the most stupid Unix design idea of all times," adding: "[T]hink about this a bit: 'For every file that is read from the disk, lets do a ... write to the disk! And, for every file that is already cached and which we read from the cache ... do a write to the disk!'" He further emphasized the performance impact thus:

Atime updates are by far the biggest IO performance deficiency that Linux has today. Getting rid of atime updates would give us more everyday Linux performance than all the pagecache speedups of the past 10 years, *combined*.

```
man mount
....
   relatime
          Update inode access times relative to modify or change time.
          Access time is only updated if the previous access time was ear-
          lier than the current modify or change time. (Similar to noat-
          ime, but doesn't break mutt or other applications that need to
          know if a file has been read since the last time it was modi-
          fied.)

          Since Linux 2.6.30, the kernel defaults to the behavior provided
          by this option (unless noatime was specified), and the stricta-
          time option is required to obtain traditional semantics. In
          addition, since Linux 2.6.30, the file's last access time is
          always updated if it is more than 1 day old.
```

tar (tape archive

Zum Packen und Entpacken von Archiven. Tar schreibt die zu sichernden Dateien sequenziell (hintereinander) in ein Backup-File. Diese Backup-Datei wird Archiv und **Tarball** genannt. Das sequenzielle Hinzufügen ist perfekt für Bandlaufwerke.

Optionen

-c create erzeugt ein neues Archiv

-x extract ein Archiv entpacken

-v verbose zeigt Informationen beim Packen/Entpacken an

-f following Name des Archivs

-z zip gzip Komprimierung

-j bzip2 bzip2 Komprimierung (etwas langsamer, aber effizienter als gzip)
Andere Bezeichunungen: tbz, tb2 und bz2

-Z compress Komprimierung

-d Dateien im Archiv und im Dateisystem miteinander vergleichen.

-J Archiv zusätzlich mit xz (de)komprimieren.

-k Das Überschreiben existierender Dateien beim Extrahieren aus einem Archiv verhindern.

-p Zugriffsrechte beim Extrahieren erhalten.

-r Dateien an ein bestehendes Archiv anhängen.

-t Inhalt eines Archivs anzeigen.

-u Nur Dateien anhängen, die jünger sind als ihre Archiv-Version.

-w Jede Aktion bestätigen.

-A Inhalt eines bestehenden Archivs an ein anderes Archiv anhängen.

-C Wechselt in das angegebene Verzeichnis. Das Archiv wird dann dort entpackt.

-M Mehrteiliges Archiv anlegen/anzeigen/extrahieren.

-L Medium wechseln, wenn ZAHL KBytes geschrieben sind.

-W Archiv nach dem Schreiben prüfen.

Beispiele

Erzeugt ein Archiv namens wilma.tar vom Verzeichnis cray:

```
$ tar cvf wilma.tar cray/
```

Erzeugt ein gzip-Archiv namens wilma.tar vom File cray.xml:

```
$ tar czvf wilma.tar cray.xml
```

Erzeugt ein bzip2-Archiv namens wilma.tar vom File cray.xml:

```
$ tar cjvf wilma.tar cray.xml
```

Entpacke das Archiv namens wilma.tar:

```
$ tar xf wilma.tar
```

Entpacke eine Datei, die mit bzip2 komprimiert wurde:

```
$ tar xjf wilma.bzip2
```

Ein Archiv mit dem Namen **archiv.tar** mit den Dateien **datei_1.txt** und allen Dateien vom Typ ***.pdf** anlegen:

```
$ tar -cf archiv.tar datei_1.txt *.pdf
```

Link

Ein File namens a anlegen:

```
wilma@windows10 ~/foo
$ touch a
```

ls -li zeigt nur einen Link (3. Spalte) auf dieses Datei (mit -i wird der Inode-Eintrag angezeigt).

```
wilma@windows10 ~/foo
$ ls -li
insgesamt 0
6755399441191961 -rw-r--r-- 1 wilma Kein 0 28. Mrz 11:08 a
```

Einen **Hard Link** namens b erzeugen:

```
wilma@windows10 ~/foo
$ ln a b
```

Beide Files haben den gleichen Inode-Eintrag (Anzahl der hard links hat sich auf 1 erhöht):

```
wilma@windows10 ~/foo
$ ls -li
insgesamt 0
6755399441191961 -rw-r--r-- 2 wilma Kein 0 28. Mrz 11:08 a
6755399441191961 -rw-r--r-- 2 wilma Kein 0 28. Mrz 11:08 b
```

Mit der Option -s wird ein **Soft Link** erzeugt:

```
wilma@windows10 ~/foo
$ ln -s a c
```

Die Datei c hat einen anderen Inode-Eintrag:

```
wilma@windows10 ~/foo
$ ls -li
insgesamt 1
 6755399441191961 -rw-r--r-- 2 wilma Kein 0 28. Mrz 11:08 a
 6755399441191961 -rw-r--r-- 2 wilma Kein 0 28. Mrz 11:08 b
23925373020491559 lrwxrwxrwx 1 wilma Kein 1 28. Mrz 11:16 c -> a
```

I/O Redirection Operator

Mit **I/O Redirection** ist das Weiterleiten/Umbiegen von Input (I) und Ouput (O) gemeint.

Standard Input/Output

Shell-Kommandos lesen Daten über **Standard Input**. Default ist die Tastatur (keyboard).

Mit dem Operator < wird der Standard Input angegeben.

Mit dem Operator > wird der Standard Ouput angegeben.

Für das Kommando **sort** den **Standard Input** von Keyboard auf das File mitarbeiter.txt ändern:

```
wilma@windows10 /
$ sort < mitarbeiter.txt
```

Der Output von sort (default das Display) in das File mitarbeiter_sortiert ändern:

```
wilma@windows10 /
$ sort < mitarbeiter.txt > sorted_mitarbeiter.txt
```

Die Reihenfolge der Operatoren spielt keine Rolle:

```
wilma@windows10 /
$ sort > sorted_mitarbeiter.txt < mitarbeiter.txt
```

Pipe

Mit dem Pipe Operator | wird der Standard-Ouput von einem Shell-Programm, zum Standard-Input von einem anderem Shell-Programm.

Das Ergebnis (Outpout) vom Kommando **ls** zum Pager **less** umleiten (less liest diesen Output als Input):

```
wilma@windows10 /
$ sort > ls -l | less
```

Den Output von **ls** zum Filter **head** (liest die ersten 10 Zeilen) schicken:

```
wilma@windows10 /
$ sort > ls -l | head
```

Den Output von du zum Filter sort (sortieren) schicken:

```
$ du /etc/ | sort -nr
```

Terminal Pager

ls

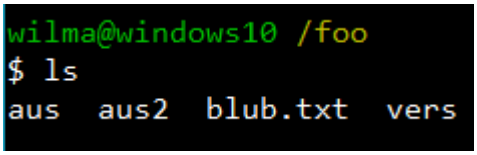
Anzeigen von File- und Verzeichnisinformationen.

Ohne Angabe von Parameter zeigt ls nur die Namen der Files und und die Namen der Verzeichnisse.

Versteckte Dateien (beginnen mit einem Punkt) werden nicht angezeigt. Um auch die versteckten Files zu sehen, muß ls mit dem Parameter -a aufgerufen werden (a = all files).

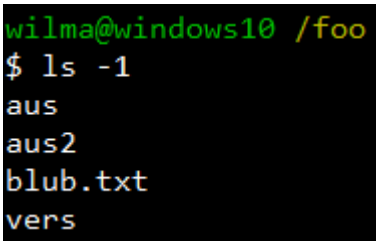
Ein Aufruf ohne Parameter:

```
wilma@windows10 /foo
$ ls
aus   aus2   blub.txt   vers
```



Mit der Option -1 werden die Namen untereinander angezeigt:

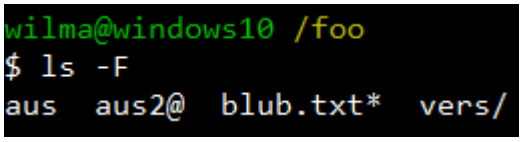
```
wilma@windows10 /foo
$ ls -1
aus
aus2
blub.txt
vers
```



File Classification

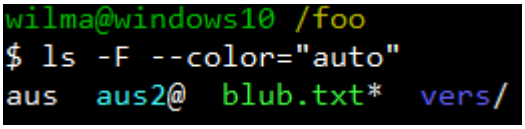
Mit der Option -F wird die File Classification angezeigt (* = ausführbar, / = Directory, @ = Link):

```
wilma@windows10 /foo
$ ls -F
aus   aus2@   blub.txt*   vers/
```



Mit der Option –-color=auto wird die File Classification farbig angezeigt:

```
wilma@windows10 /foo
$ ls -F –color="auto"
aus   aus2@   blub.txt*   vers/
```



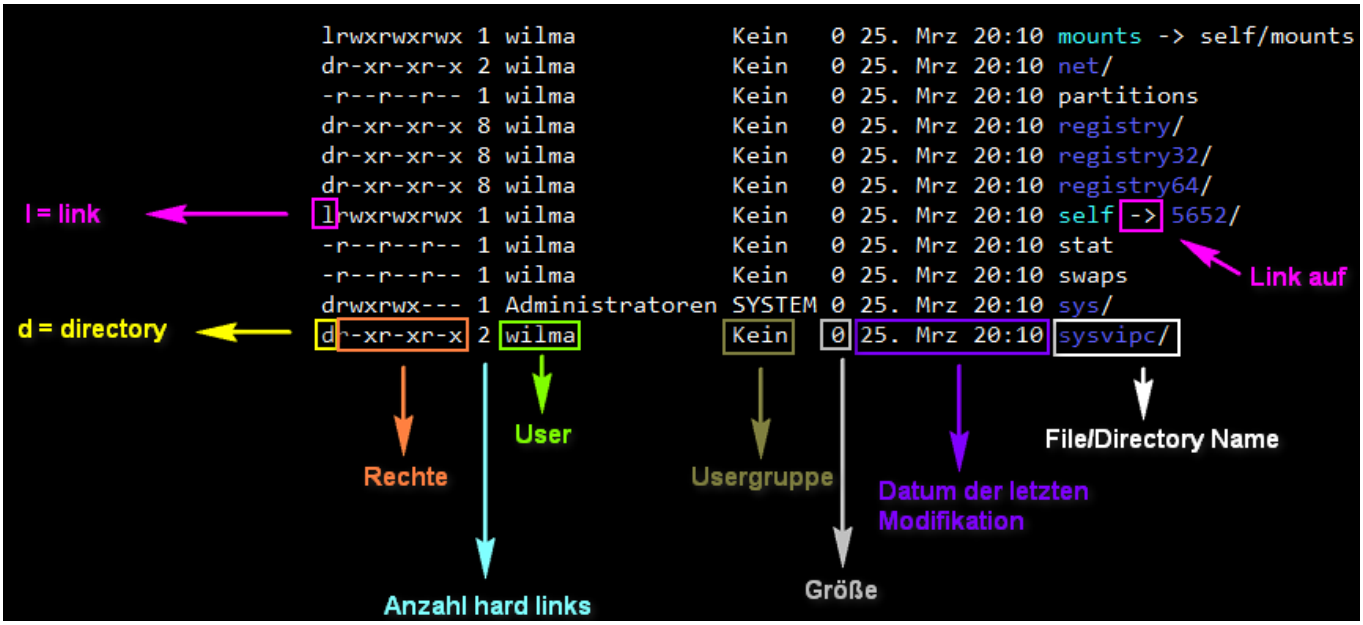
long list

Mit dem Parameter -l (long list) werden 7 Spalten angezeigt:

1. Rechte
   Das erste Zeichen: d = directory, l = link, - = Datei, s = socket
2. A nzahl der hard links, die auf den Eintrag zeigen
3. Name of owner
4. Name of group
5. Größe in bytes
6. Zeit der letzten Modifikation
7. Dateiname / Verzeichnisname

```
$ ls -l
```

Erzeugt diesen Output:



Linux: awk

Kapitel 3

Linux: awk

Das Command-Line-Tool awk ist ein zeilenbasierender Parser für Plaintext. Mit seiner eigenen Programmiersprache wird der Output erstellt.

Der Name awk kommt von den Initialen der Entwickler: Alfred V. Aho, Peter Weinberger und Brian W. Kernighan.

Die erste Version wurde 1977 in den AT&T Bell Laboratories entwickelt. 1985 wurde awk um **user-defined functions** und **regular expressions** erweitert. Die nächste Erweiterung fand 1989 statt.

Ab 1986 wurd **gawk** (GNU awk) von Paul Rubin, Jay Fenlason und John Woods entwickelt. Von 1988 bis 1989 wurde gawk um die neuen awk-Features erweitert (von David Trueman und Arnold Robbins).

1994 wurde Arnold Robbins der Hauptentwickler. 1997 wurde gawk um den **network access** erweitert (von Jürgen Kahrs).

Der **awk-level debugger** wurde 2011 von John Haque hinzugefügt.

Trennzeichen (Separator)

Awk verarbeitet den Input zeilenweise (Zeile für Zeile). Jede Zeile wird in Felder (Spalten) getrennt (mit Felder sind keine Arrays gemeint).

Record (Datensatz)

Eine eingelesene Textzeile wird auch Datensatz (Record) genannt.

Zeilentrennzeichen

Das default-Zeilen-Trennzeichen ist **newline**. Der Inhalt der Zeile ist in der Variable $0.

Die Variable **NR** enthält die gegenwärtige Zeile (Zeilennummer).

Mit der internen builtin (predefined) Variable RS kann das Zeilentrennzeichen geändert.

In diesem Beispiel wird das Zeilentrennzeichen auf einen Tag geändert:

```
BEGIN{
RS="\t"
}
{print $0}
```

In Gawk kann der Zeilentrenner auch \0 (Nul Character) sein. Damit wird das ganze Textfile als eine einzige Zeile gelesen. Andere awk-Implementierungen nehmen NUL als String-Terminator (c style).

Spaltentrennzeichen

Der default-Spaltentrenner ist Whitespace (Leerzeichen und Tab).

Die Variable **NF** enthält die Anzahl der Spalten (der gegenwärtigen Zeile).

Weiter builtin (predefined) Variablen siehe Seite 73.

Innerhalb einer Zeile sind **TAB** und **Space** (keine anderen Whitespace-Character) die Trennzeichen für Spalten.

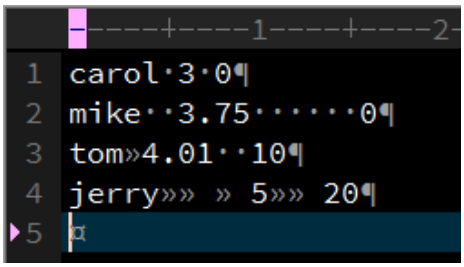
Aufeinanderfolgende Tabs und Spaces werden zu einem einzigen Trennzeichen zusammengefasst.

Um das Zusammenfassen zu deaktivieren, muss das Trennzeichen explizit gesetzt werden:

Der builtin-Variable FS wird ein Wert zugewiesen, in diesem Beispiel dem Tabulator (\t):

```
BEGIN{
FS="\t"
}
{print $0}
```

Beispiel für das automatische Zusammenfassen von Whitespace zu einem einzigen Trennzeichen:



In Zeile 2 werden 6 Leerzeichen (zwischen .75 und 0) zu einem Trennzeichen.

In Zeile 4 werden 3 Tabs und 2 Leerzeichen (zwischen jerry und 5) zu einem Trennzeichen.

Auf jede Spalte kann über eine Variable zugegriffen werden: $1 ist die erste Spalte, $2 die zweite und $n die n. Spalte. Die Spaltenanzahl darf in jeder Zeile unterschiedlich sein.

Mit der Option -F kann das Spaltentrennzeichen gesetzt werden. In diesem Beispiel auf einen Doppelpunkt:

```
-F":"
```

Spaltentrennzeichen auf einen Tab setzen (1 Tab = 1 Spalte):

```
-F"\t"
```

Das Trennzeichen kann auch durch ein Regex definiert werden:

```
-F"\t+"
```

Das Pluszeichen steht für: ein oder mehrere vom Zeichen davor.
Somit: Der Tab darf mehrmals hintereinander auftauchen (n Tabs = 1 Spalte)

Trennzeichen ist das Wort foo:

```
-F"foo"
```

Trennzeichen ist eine Ziffer:

```
-F"[0-9]"
```

Trennzeichen ist ein Semikolon oder ein Beistrich (beide Zeichen werden als Trenner verwendet):

```
-F"[;,]"
```

Siehe auch: Beispiel für **mehrere verschiedene Trennnzeichen** Seite 72

Trennzeichen per awk-Variable im BEGIN-Block definieren (Seite 73)

Feld (Zeilenspalte)

Eine Spalte wird auch Feld genannt. Zeilenfelder haben nichts mit Array-Feldern zu tun.

Um die 1. Spalte anzuzeigen:

```
print $1
```

Um die 3. Spalte anzuzeigen:

```
print $3
```

Um die Anzahl der vorhandenen Spalten anzuzeigen:

```
print NF
```

Um die letzte Spalte anzuzeigen:

```
print $NF
```

Um die 4. Spalte anzuzeigen:

```
x=4
print $x
```

Mit einer Schleife durch alle Spalten:

```
{
 for (i=1; i<= NF ; i++)
 {
 print "spalte " i " hat den wert: " $i
 }
}
```

Ergibt als Output (siehe mitarbeiter.txt Seite 73):

```
$ awk  -f skript.awk mitarbeiter.txt
spalte 1 hat den wert: Carol
spalte 2 hat den wert: 3
spalte 3 hat den wert: 0
spalte 1 hat den wert: Mike
spalte 2 hat den wert: 3.75
spalte 3 hat den wert: 0
spalte 1 hat den wert: Tom
spalte 2 hat den wert: 4.01
spalte 3 hat den wert: 10
spalte 4 hat den wert: Lehrling
spalte 1 hat den wert: Jerry
spalte 2 hat den wert: 5
spalte 3 hat den wert: 20
```

Lösche die 3. und 5. Spalte und zeige die Zeile an (awk ändert automatisch die Variable $0):

```
$3=""; $5=""; print;
```

awk Processing Cycle

Vor dem Main-Loop (Schritt 2 bis 3) wird einmal der BEGIN-Block und einmal der END-Block ausgeführt:

1. führe Anweisungen vom BEGIN-Block aus
2. Lese eine Zeile (bis \n) vom Input und trenne in Spalten (Spaltentrennzeichen mit Option -F bestimmen)
3. führe Anweisungen vom Main-Block aus. Falls weitere Inputzeilen vorhanden: Springe zu Punkt 2.
4. führe Anweisungen vom END-Block aus

Main Input Loop

Die Anweisung in den geschweiften Klammern ist der **Main Programm-Block**:

```
$ awk  '{print $1}'  mitarbeiter.txt
```

Awk loopt automatisch durch jede Zeile (Trennzeichen ist \n) im Input.

Ein **Main Programmblöcke** wird somit mehrmals für jede einzelne Zeile angewendet.
Wenn es mehrere **Zeilen im Main Programmblock** gibt, werden diese nacheinander ausgeführt.

Weiters gibt es auch einen **BEGIN-Programmblock** und einen **END-Programmblock** (diese werden nur einmal ausgeführt). Der BEGIN-Block einmalig vor dem Loop, und der END-Block einmalig nach dem Loop.

Action

Der Block mit den geschweiften Klammern ist die Action:

```
{print "Carol in 1. Spalte nicht gefunden"}
```

Aktionen können durch Bedingungen (Pattern) eingeschränkt werden.
Führe den Aktionsblock nur dann aus, wenn in der 1. Spalte der String Carol nicht gefunden wird.

```
$1 !~ /Carol/ {print "Carol in 1. Spalte nicht gefunden"}
```

`$1` in dieser Variable ist der Inhalt der 1. Spalte

`!~` der gesuchte String ist nicht in $1 enthalten

`/Carol/` der Search-Pattern (gesucht wird der String Carol)

Fixe Zeichenketten werden in doppelten Anführungszeichen gesetzt. Siehe auch: print (Seite 87)

```
wilma@windows10 /
$ awk  '{print "Mitarbeiter " $1 }'  mitarbeiter.txt
Mitarbeiter Carol
Mitarbeiter Mike
Mitarbeiter Tom
Mitarbeiter Jerry
```

Der Pattern **BEGIN** definiert den BEGIN-Block. Der Pattern **END** definiert den End-Block.

Siehe auch: Programmblock (Seite 76), Mehrere Programmblöcke (Seite 80)
Bedindigter Programmblock (Seite 84)
Pattern-Action Statement für den bedingten Programmblock (Seite 74)

Shell-Aufruf

Der Programmblock (alles zwischen den Anführungszeichen) als Argument:

awk 'awk-code' input(filenames)

```
$ awk   '$3==0 {print  $1}'   mitarbeiter.txt
```

Parameter

-F Spaltentrennzeichen setzen

-f Dateiname des awk-Skript

awk -f scriptfilename input(filenames)

-v mit -v können dem Skript Variablen(werte) übergeben werden: -v foo=123

Beispiel für -F

Setze Spaltentrennzeichen auf \t (Tab). Lese als Input das File mitarbeiter.txt (siehe mitarbeiter.txt Seite 73).

Zeige 1. und 3. Spalte.

```
wilma@windows10 /
$ awk -F"\t" '{ print $1 " hat " $3 " Stunden gearbeitet"}' mitarbeiter.txt
Carol hat 0 Stunden gearbeitet
Mike hat 0 Stunden gearbeitet
Tom hat 10 Stunden gearbeitet
Jerry hat 20 Stunden gearbeitet
```

Siehe auch: print (Seite 87)

Beispiel für -f

Lese den externen awk-Code aus der Datei skript.awk.

Der Input kommt aus mitarbeiter.txt (siehe mitarbeiter.txt Seite 73):

```
$ awk -f skript.awk mitarbeiter.txt
```

Beispiel für -v

Nach dem Parameter -v folgt die Variablenzuweisung (Gawk):

```
wilma@windows10 /
$ awk -v foo=55 -v foo2=66 -f skript2.awk
55
66
```

Das awk-Skript skript2.awk:

```
BEGIN{
print foo
print foo2
}
```

Input per Pipe

Input per Pipe von einem Shell-Programm:

```
Command |  awk 'awk-code'
```

```
ls -l    | awk '$3==0 {print  $1}'
```

Siehe auch: Pipe (Seite 76)

Output an Pipe

Mit dem Pipe-Operator | den Output von awk an ein Shell-Programm (zB sort) weiterleiten (umbiegen).

Den Awk-Output an das Shell-Programm sort übergeben:

```
wilma@windows10 /
$ awk -f skript.awk mitarbeiter.txt | sort
Carol   3       0
Jerry   5       20
Mike    3.75    0
Tom     4.01    10      Lehrling
```

Siehe auch: Pipe (Seite 76)

ENVIRON (Gawk Environment-Variable)

Die builtin-Variable ENVIRON (assoziative Array) von Gawk ist nicht vorhanden, wenn gawk mit --traditional' oder `--posix gestartet wird.

ENVIRON["AWKPATH"]

In einigen awk-Implementation muss bei Verwendung vom command-line-Parameter -f der Pfad angegeben werden (ausser das File ist im gegenwärtigen Verzeichnis).

Wird bei Gawk kein Pfad angegeben, sucht Gawk in den Pfaden, die in ENVIRON["AWKPATH"] angegeben sind. Trennzeichen ist der Beistrich.

Je nach System ist der default:

```
.:/usr/local/share/awk
```

oder:

```
.:/usr/share/awk
```

Mehrere Trennzeichen (Multiple Delimiter)

Soll nach mehreren (verschiedenen) Trennzeichen gesucht werden, müssen diese in eckigen Klammern stehen.

Beispiel geburtsdaten.txt

Die Plain-Text-Datei geburtsdaten.txt enthält 6 Zeilen.

```
Adams, Ansel;photographer;1902-02-20;1984-04-22
Asimov, Isaac;author;1920-01-02;1992-04-06
Janney, Allison;actress;1959-11-19
La Rue, Lash;actor;1917-06-15;1996-05-21
Sagan, Carl;astronomer/writer;1934-11-09;1996-12-20
Sharif, Omar;actor;1932-04-10
```

Für Nach- und Vornamen das Komma, für Beruf das Semikolon und für das Geburtsjahr das Minuszeichen:

```
wilma@windows10 /
$ awk -F'[;,-]' '{print $2, $1, "(" $3 ") was born in", $4 "."}' geburtsdaten.txt
 Ansel Adams (photographer) was born in 1902.
 Isaac Asimov (author) was born in 1920.
 Allison Janney (actress) was born in 1959.
 Lash La Rue (actor) was born in 1917.
 Carl Sagan (astronomer/writer) was born in 1934.
 Omar Sharif (actor) was born in 1932.
```

Siehe auch print Seite 87.

Beispiel klammern.txt

Die Plain-Text-Datei klammern.txt enthält 3 Zeilen.

```
123;abc[202];124(aa);44;<dd>[uuu]-[iii]-(999)
125;abc[203];124(bb);55;<ee>[vvv]-[jjj]-(888)
127;abc[204];124(cc);66;<ff>[www]-[kkk]-(777)
```

Um an die Werte in den runden Klammern zu gelangen: `()()`

Die Reihenfolge der runden Klammern spielt keine Rolle. Das Ergebnis ist identisch: `[()]`

```
wilma@windows10 /
$ awk -F '()()'   '{print "Spalte2: " $2 " / Spalte4: " $4}' klammern.txt
Spalte2: aa / Spalte4: 999
Spalte2: bb / Spalte4: 888
Spalte2: cc / Spalte4: 777
```

Für die Werte in eckigen Klammern: `[][]`

```
wilma@windows10 /
$ awk -F '[][]'   '{print "S2: " $2 " / S4: " $4 " / S6: " $6}' klammern.txt
S2: 202 / S4: uuu / S6: iii
S2: 203 / S4: vvv / S6: jjj
S2: 204 / S4: www / S6: kkk
```

Für die Werte in eckigen und runden Klammern: `[][()]`

```
wilma@windows10 /
$ awk -F '[][()]' '{print "S2: " $2 " / S4: " $4 " / S6: " $6 " / S8: " $8 " / S10: " $10}'
klammern.txt
S2: 202 / S4: aa / S6: uuu / S8: iii / S10: 999
S2: 203 / S4: bb / S6: vvv / S8: jjj / S10: 888
S2: 204 / S4: cc / S6: www / S8: kkk / S10: 777
```

awk Builtin-Variablen

Built-in (predefined) Variablen von awk:

$0 gesamter Zeileninhalt (ohne Zeilentrennzeichen)

$*n* *n*. Spalte. Zählung beginnt bei 1.

Wenn eine Spaltenvariable durch eine Zuweisung geändert wird, baut awk die gesamte Zeile $0 neu. Wenn die Variable $1="" auf Leerstring gesetzt wird, verschwindet sie auch in der Variable $0. Kommt ein Trennzeichen im neuen String vor, wird aber keine Spalte erzeugt: $1="Carol King" (keine neue Spalte, obwohl Space ein Trennzeichen ist).

NR gegenwärtige Zeilennummer. Zählung beginnt bei 1 (Number of the Record). Siehe Seite 66
Bei mehreren Files wird weitergezählt (falls gewünscht, muss NR manuell gesetzt werden).

NF Spaltenanzahl der gegenwärtigen Zeile (Number of Fields). Siehe Seite 66.

Wird per manueller Zuweisung der Wert von NF gesenkt, werden die überflüssigen Spalten in $0 von Gawk gelöscht. Dieses Verhalten ist von der awk-Implementierung abhängig.

FS Spaltentrennzeichen für den Input (Field Separator).
OFS Spaltentrennzeichen für den Output

RS Zeilentrennzeichen für den Input (Record Separator).
ORS Zeilentrennzeichen für den Ouput.

FILENAME Dateiname des gegenwärtig verarbeitenden Inputfile (hilfreich bei mehr als einem Inputfile). gibt es keine Datei, ist FILENAME -

CONVFMT steuert die Datentyp-Konvertierung. Default ist %.6g. Siehe Seite 81, 82 (gawk-Feature)

SUBSEP Trennzeichen bei Indexbezeichnungen von mehrdimensionalen Arrays. Siehe Seite 78
gawk/nawk-Feature

RSTART die Funktion match() setzt diese Variable. Siehe Seite 92
RSTART ist die Startposition vom gesuchten String (default ist 0). gawk/nawk-Feature

RLENGTH die Funktion match() setzt diese Variable. Siehe Seite 92
RLENGTH ist die Zeichenlänge vom gesuchten String (default ist -1). gawk/nawk-Feature

ARGV Array ARGV enthält die Argumente, mit der awk in der Shell aufgerufen worden ist.
gawk/nawk-Feature. Siehe Seite 95

ARGC die Anzahl der Argumente (also die Arraygröße von ARGV). gawk/nawk-Feature. Siehe Seite 95

ARGIND die nummerische Variable zeigt auf den jeweiligen Arrayindex von ARGV, um den gegenwärtigen Filename zu ermitteln (der gerade im Main-Loop verarbeitet wird). gawk-Feature.
Siehe auch FILENAME und Seite 95

Beispiel mitarbeiter.txt

Die Plain-Text-Datei mitarbeiter.txt enthält 4 Zeilen. Jede Zeile enthält 3 Spalten (die Zeile 3 hat 4 Spalten):
Name des Mitarbeiters, Stundenlohn und Stundenanzahl (Trennzeichen ist \t)

```
Carol  3  0
Mike3.75  0
Tom 4.01  10 Lehrling
Jerry  5  20
```

Um alle Mitarbeiter anzuzeigen:

```
wilma@windows10 /
$ awk  '{print $1}'  mitarbeiter.txt
Carol
Mike
Tom
Jerry
```

Pattern-Action Statement

Um den Output einzuschränken, wird ein **Pattern-Action Statement** benötigt. Der Pattern wird vor dem Aktionsteil (Action) gesetzt. Der Aktionsteils beginnt mit { und endet mit }. Eine Action ohne Pattern wird immer ausgeführt.
Als Beispiel alle Mitarbeiter, die nicht gearbeitet haben (siehe mitarbeiter.txt Seite 73):

```
wilma@windows10 /
$ awk  '$3==0 {print  $1}'  mitarbeiter.txt
Carol
Mike
```

`$3==0` ist der Pattern. `{print  $1}` ist die Action.

Um den Gesamt-Lohn zu berechnen:

```
wilma@windows10 /
$ awk  '$3  >  0 {  print  $1 " bekommt " $2  *  $3 " €"}'  mitarbeiter.txt
Tom bekommt 40.1 €
Jerry bekommt 100 €
```

$1 steht für die erste Spalte, $2 für die zweite Spalte und $3 für die dritte Spalte der jeweiligen Zeile.
Mit $3>0 werden nur Mitarbeiter angezeigt, die auch gearbeitet haben (mehr als 0 Stunden). Mit $2*$3 wird der Gesamtlohn berechnet (Stundensatz * Stundenanzahl).

Mit /foo/ wird nach dem String foo gesucht. Lohn nur für Jerry (in der Zeile muss der String **Jerry** vorkommen):

```
wilma@windows10 /
$ awk  '/Jerry/ {  print  $1 " bekommt " $2  *  $3 " €"}'  mitarbeiter.txt
Jerry bekommt 100 €
```

Lohn nur für Jerry und Carol anzeigen:

```
wilma@windows10 /
$  awk  '$1=="Jerry" || $1=="Carol" {print $1 " kriegt " $2*$3 " €"}'  mitarbeiter.txt
Carol kriegt 0 €
Jerry kriegt 100 €
```

Regular Expression Operator

~ Match operator String gefunden.

!~ No Match operator String nicht gefunden.

Suche den String **Carol**:

```
/Carol/ {print "Carol gefunden"}
```

Ist identisch mit:

```
$0 ~ /Carol/  {print "Carol gefunden"}
```

Beginnt zuerst der Suchausdruck, erscheint eine awk-Warnung:

```
/Carol/ ~ $0 {print "Carol gefunden"}
```

Output:

```
awk: skript.awk:1: Warnung: Regulärer Ausdruck links vom »~«- oder »!~«-Operator
```

In Verbindung mit einem Bool-Operator:

```
/Carol/ || /Tom/  {print "Carol oder Tom gefunden"} # sucht nach Carol oder Tom
```

ist identischt mit:

```
$0 ~ /Carol/ || /Tom/  {print "Carol oder Tom gefunden"}
```

Siehe auch: Programmblock (Seite 76)
Mehrere Programmblöcke (Seite 80)
Bedindigter Programmblock (Seite 8484)

Externes awk-Skript

Das gesamte Programm (alles zwischen den einfachen Anführungszeichen):

```
$  awk  '$1=="Jerry" || $1=="Carol" {print $1 " kriegt " $2*$3 " €"}'  mitarbeiter.txt
```

wird in ein File namens skript.awk ausgelagert:

```
$1=="Jerry" || $1=="Carol" {print $1 " kriegt " $2*$3 " €"}
```

In der Shell wird mit der Option -f der Filename angegeben:

```
wilma@windows10 /
$ awk -f skript.awk mitarbeiter.txt
Carol kriegt 0 €
Jerry kriegt 100
```

Siehe auch mitarbeiter.txt Seite 73.

Remark

Alles nach der Raute wird von awk nicht interpretitert. Der folgende Text (bis \n) kann als Bemerkung verwendet werden:

```
END {
# letzte Zeile
}
```

Pipe

Mit dem Pipe-Operator | den Output von einem Shell-Programm (zB ls) an awk weiterleiten (umbiegen):

```
ls -l | awk -f skript2.awk
```

Das Skript skript2.awk besteht aus einer Zeile:

```
{print "Zeile " NR ": " $0}
```

Ergibt diesen Output:

```
wilma@windows10 /var/log
$ ls -l  $* | awk -f /skript2.awk
Zeile 1: insgesamt 2816
Zeile 2: -rw-r--r-- 1 wilma Administratoren  405698 19. Mrz 18:51 setup.log
Zeile 3: -rw-r--r-- 1 wilma Administratoren 2472748 19. Mrz 18:51 setup.log.full
```

Mit dem Pipe-Operator | den Output von awk an ein Shell-Programm (zB sort) weiterleiten (umbiegen):

```
Den Awk-Output an das Shell-Programm sort übergeben:
wilma@windows10 /
$ awk -f skript.awk mitarbeiter.txt | sort
Carol   3       0
Jerry   5       20
Mike    3.75    0
Tom     4.01    10      Lehrling
```

Programmblock (Aktionsteil)

Blöcke sind gebündelte Anweisungen. Jeder Block beginnt mit einem { und endet mit einem }

Der Block ist ein Aktionsteil und wird deshalb auch Action genannt.

Wenn der Programmteil nur in bestimmten Situationen durchgeführt werden soll (bedingte Ausführung), wird ein **Pattern-Action Statement** vorangestellt. Nur wenn das Pattern erfüllt ist, wird der Progammblock ausgeführt.

Ein normaler Programmblock wird öfters (für jede einzelne Zeile) ausgeführt (durch den **Main Input Loop**).

Der BEGIN-Block wird nur einmal davor, und der End-Block wird nur einmal danach ausgeführt:

```
BEGIN {print "Alle Mitarbeiter:\n======Anfang===="}
{print "Mitarbeiter " $1}
END {print "======ENDE======="}
```

Ergibt diesen Output (siehe auch mitarbeiter.txt Seite 73):

```
wilma@windows10 /
$ awk -f skript.awk mitarbeiter.txt
Alle Mitarbeiter:
======Anfang====
Mitarbeiter Carol
Mitarbeiter Mike
Mitarbeiter Tom
Mitarbeiter Jerry
======ENDE=======
```

Siehe auch: Mehrere Programmblöcke (Seite 80)
Bedindigter Programmblock (Seite 84)
Pattern-Action Statement für den bedingten Programmblock (Seite 74)
awk Processing Cycle (Seite 67)

Variablen

Variablen benötigen keine Deklaration, keine Initialisierung und können ihren Datenyp (String bzw. Numeric) dynamisch ändern (siehe Seite 81 und Seite 82).

Die von awk vordefinierten Variablen (siehe Seite 73) können wie eigene Variablen behandelt werden.

awk kennt als Datentyp **Skalare** und **Arrays** (nicht in jeder awk-Implementierung).

Skalar

Ein **Skalar** kann immer nur einen einzigen Wert haben (im Gegensatz zu einem Array).

Es gibt 2 Arten von Skalaren (der Datentyp ändert sich dynamisch): Numeric und String

Wird eine String-Variable nicht initialisiert, erhält sie den default-Wert **Leerstring**. Wird eine Numeric-Variable nicht initialisiert, erhält sie den default-Wert 0.

Variablenbezeichnungen dürfen nur mit einem Buchstaben oder einem Unterstrich beginnen, und dürfen nur aus Buchstaben, Unterstrichen und Zahlen bestehen. Variablen in einer Funktion sind lokal.

Beispiel

Die Variable waehrung hat als Wert den String **Euro**. Die Variable x (number) zählt die Stunden, die Variable y (number) den Gesamt-Betrag:

```
BEGIN {waehrung=" Euro"; print "Alle Mitarbeiter:\n=Anfang="}
{
print "Mitarbeiter " $1; x = x + $2; y = y + $2*$3
}
END {
# letzte Zeile
print "=Ende=\nGesamt-Stunden: " x "\nGesamt-Auszahlung: " y waehrung
}
```

Der 1. Block ist der BEGIN-Block und wird nur einmal vor dem Loop ausgeführt.

Der 2. Block ist ein normaler Block und wird im Loop ausgeführt (für jede Zeile im Input).

Der 3. Block der der END-Block und wird nur einmal nach dem Loop ausgeführt.

Die Raute kennzeichnet einen **Remark** (Bemerkung): Alles nach der Raute wird von Awk ignoriert und somit auch nicht interpretiert/ausgeführt. Gibt es mehr als eine Anweisungen pro Zeile, müssen diese mit einem Semikolon getrennt werden.

Ergibt diesen Output (siehe auch mitarbeiter.txt Seite 73):

```
$ awk -f skript.awk mitarbeiter.txt
Alle Mitarbeiter:
=Anfang=
Mitarbeiter Carol
Mitarbeiter Mike
Mitarbeiter Tom
Mitarbeiter Jerry
=Ende=
Gesamt-Stunden: 15.76
Gesamt-Auszahlung: 140.1 Euro
```

Befinden sich mehrere Anweisungen in der gleichen Textzeile, müssen diese mit einem Semikolon getrennt werden. Aus:

```
print "1"
print "2"
```

wird:

```
print "1"; print "2"
```

Array

Arrays müssen nicht deklariert und explizit initialisiert werden.

Der Array-Variable foo wird an der Index-Position 0 der String **Carol** zugewiesen:

```
foo[0]="Carol";
```

Bei einem assoziativen Array ist der Index nicht nummerisch, sondern alphanummerisch:

```
foo["Carol"]="King";
```

Die Verwendung läßt sich auch mischen:

```
foo[0]="Luther";
foo["0"]="Carol";
foo["Carol"]="King";
print "foo: " foo[0] ", " foo["Carol"]
```

Ergibt als Output:

```
foo: Carol, King
```

Mehrdimensionales Array

Bei einem mehrdimensionalem Array sind die Indexbezeichnungen durch eckige Klammern getrennt:

```
person[0]["Nachname"]="King"
person[0]["Vorname"]="Carol"
print "Person1: " person[0]["Vorname"] ", " person[0]["Nachname"]
```

Ergibt diesen Output:

```
Person1: Carol, King
```

Alternativ sind die Indexbezeichnungen durch Beistriche getrennt:

```
  person[1,"Nachname"]="Luther"
  person[1,"Vorname"]="Martin"
print "Person2: " person[1,"Vorname"]  ", " person[1,"Nachname"]
```

Ergibt diesen Output:

```
Person2: Martin, Luther
```

SUBSEP

Intern wandelt awk ein mehrdimensionales Array in ein eindimensionales Array um. Die einzelnen Strings der Indexbezeichnungen werden also zu einem String umgewandelt. In diesem einzelnen String sind die eigentlichen Werte durch ein Steuerzeichen getrennt. Defaut für diesen Control-Character ist der **Field Seperator**. FS hat den ASCII-Wert 28 (0x1c hexadezimal und 034 oktal). Wenn Indexbezeichnungen ebensfalls FS verwenden, kann mit der awk-Variable SUBSEP ein anderer Control-Character als Trennzeichen definiert werden.

```
person[1,"Vorname"]="Carol"
print "Person2: " person[1,"Vorn\034ame"]
print "Person2: " person[1,"Vorname"]
```

Ergibt als Output:

```
Person2:
Person2: Carol
```

Mit FS ist hier nicht die builtin-Variable von awk gemeint!

Siehe auch **ASCII Control Characters** Seite 225.

in (Array-Operator)

Mit dem Operator in kann überprüft werden, ob ein bestimmter nummerischer Indexwert existiert:

indexwert in **arrayvariable**

Beispiel1: Gibt es den Indexwert **2** im Array **foo**:

```
if (2 in foo) {print "ja"} else {print "nein"}
```

Beispiel2:

```
BEGIN{
foo[0]="carol"
foo[1]="king"
if (1 in foo) {print "foo[1] gibt es"}
if (3 in foo) {print "foo[3] gibt es"} else {print "foo[3] gibt es nicht"}
}
```

Ergibt als Output:

```
foo[1] gibt es
foo[3] gibt es nicht
```

In Kombination mit einer for-Schleife:

Mit dem Keyword **in** in einer Schleife alle vorhandenen Arrayelemente einer nummerischen Array-Variable durchlaufen:

```
BEGIN{
foo[0]="a"; foo[1]="b"; foo[6]="d"; foo[2]="c"
for (indexwert in foo)
{print "indexwert " indexwert " = " foo[indexwert]}
}
```

Ergibt als Output:

```
indexwert 0 = a
indexwert 1 = b
indexwert 2 = c
indexwert 6 = d
```

delete

Mit dem Keyword **delete** kann ein einzelner Indexeintrag gelöscht werden:

```
delete arrayvariable [indexwert]
```

Ohne Angabe von einem Indexwert wird das gesamte Array geleert. Die Variable bleibt weiterhin ein Array.

Beispiel:

```
BEGIN{
foo[0]="carol"
foo[1]="king"
delete foo[1]  # delete foo  löscht die ganze Arrayvariable

if (1 in foo) {print "foo[1] gibt es"} else {print "foo[1] gibt es nicht"}
}
```

Ergibt als Output:

```
foo[1] gibt es nicht
```

Operatoren

Assignment Operator (Zuweisung)

Zuweisungsoperatoren:

=	Zuweisung.	Alle 3 Variablen sind 0:	`x = y = z = 0`
+=	Shortcut addition assignment	`foo = foo + 5` identisch mit:	`foo+=5`
-=	Shortcut subtraction assignment		
*=	Shortcut multiplication assignment		
/=	Shortcut division assignment		
%=	Shortcut modulo division assignment		
^=	Shortcut power (Potenz)		
**=	Non-Posix Gawk-Feature.	Identisch mit ^=	

Conditional Operator (Bedingung)

>	Is greater than	größer als
>=	Is greater than or equal to	größer oder gleich als
<	Is less than	kleiner als
<=	Is less than or equal to	kleiner oder gleich als
==	Is equal to	gleich
!=	Is not equal to	ungleich

Boolean Operators

&&	Logical AND	und	&& höhere Priorität als ||
||	Logical OR	oder	
!	Logical NOT	ungleich	

Regular Expression

~ Matches	Suche in 1. Spalte den String **Carol**: `$1 ~ /Carol/`
!~	Does not match

Überprüfe, ob in der Zeile die Wörter **King**, **Carol** und **tot** vorkommen:.

```
$0 ~ /King/ && /Carol/ && /tot/ {print "wahr: " $0;}
```

Und zusätzlich in der 1. Spalte nicht der String **am Leben** vorkommt:

```
$0 ~ /King/ && /Carol/ && /tot/  && $1 !~ /am Leben/  {print "wahr: " $0;}
```

Regular Expression Operator

~	Match operator	String gefunden.
!~	No Match operator	String nicht gefunden.

Ternary Operator (Conditional Assignment)

Es werden 3 Operanden benötigt. Der ternäre Operator besteht aus: ? :

Ist der Ausdruck vor dem ? true, wird der Ausdruck nach dem ? ausgeführt, ansonsten der Ausdruck nach dem :

Beispiel:

```
a=1;
b = (a) ? 2 : 3;  # a ist true, daher hat b den Wert 2)
b = (!a) ? 2 : 3;    # a ist false, daher hat b den Wert 3)
print b;
```

Vergleich Numeric mit String

Variablen benötigen keine Deklaration und können ihren Datenyp dynamisch ändern. Wird eine String-Variable nicht initialisiert, erhält sie den default-Wert Leerstring. Wird eine Numeric-Variable nicht initialisiert, erhält sie den default-Wert 0.

Wenn Vergleichsoperatoren zwei unterschiedlichen Datentypen vergleichen (zb **Numeric** mit **String**), findet eine Konvertierung statt. Die Art der Konvertierung wird mit der awk-Variable **CONVFMT** festgelegt.
Siehe auch awk-Variablen Seite 73.

Der String "1" wird als 1 interpretiert:

`1=="1"` true

Der String "+1" wird als "+" interpretiert:

`1=="+1"` false

Zahlenvergleich (Numeric Comparison):

`1==1` true

Der String "1" wird als 1 interpretiert und "2" als 2:

`"1" <= "2"` true

Der String "1a" wird als 1 interpretiert und "2" als 2:

`"1a" < "2"` true

Der String "a1" wird als "a" interpretiert:

`"a1" < 2` false

Zahlenvergleich (Numeric Comparison):

`1.0 < 2.4` false

Zahlenvergleich (Numeric Comparison):

`+3 > -2.4` true

Zahlenvergleich (.6 steht für 0.6 und -.8 für -0.8):

`-2.4 .6 > -.8` true

Konkatenation (Verkettung)

Strings werden mit einem Leerzeichen verbunden (konkatiniert):

```
{foo="Carol " "King"; print foo;}
```

Ergibt als Output:

```
Carol King
```

Bei zwei unterschiedlichen Datentypen (**Numeric** mit **String**), findet eine Konvertierung statt. Die Art der Konvertierung wird mit der awk-Variable **CONVFMT** festgelegt.

Siehe auch awk-Variablen Seite 73 und print Seite 84.

Variablen benötigen keine Deklaration und können ihren Datenyp dynamisch ändern.

Wird eine String-Variable nicht initialisiert, erhält sie den default-Wert **Leerstring**. Wird eine Numeric-Variable nicht initialisiert, erhält sie den default-Wert 0.

String + Zahl:

```
{foo="Carol " "King" + 9; print foo;}
{foo="Carol " "King"  9; print foo;}
{foo="Carol " "King" + 9a; print foo;}
{foo="Carol " "King"  9a; print foo;}
```

Ergibt immer als Output:

```
Carol King9
```

Konkatenation mit Variablen:

```
vorname = "Carol"; nachname = "King"; beschreibung = "cool"
text = "Mrs. " vorname " " nachname " ist " beschreibung
```

Ergibt als Output:

```
Mrs. Carol King ist cool
```

Aus Zahlen ein String und zurück:

```
print (2 5) + 3;
```

Ergibt als Output:

```
28
```

Aus `2` und `"k5"` wird `"2k5"`, und aus `"2k5"` wird 2:

```
print (2 "k5") + 3;
```

Ergibt als Output:

```
5
```

Bei einer nummerischen Operation wird aus dem nicht konvertierbaren String der Wert 0:

```
print (2 2) * "d2"
```

Ergibt als Output (aus "d2" wird 0):

```
0
```

Priorität und Assoziativität von Operatoren

Priorität	Operator		Assoziativität
1	()	Gruppierung	→
2	$	Feldoperator	→
3	++ --	Inkrement, Dekrement	→
4	^ **	Exponentiation	←
5	+ - !	Unäres Plus, Unäres Minus, logisches NOT	→
6	* / %	Multiplikation, Division, Modulo	→
7	+ -	Addition, Subtraktion	→
8		Stringverknüpfung	→
9	< <= == != > >= >> | |&	Vergleichsoperatoren und Umleitungen	→
10	~ !~	enthält, enthält nicht	→
11	in	Array-Mitgliedsoperator	→
12	&&	Logisches AND	→
13	||	Logisches OR	→
14	?:	Bedingungsoperator (Konditionaloperator)	←
15	= += -= *= /= %= ^=	Zuweisungen	←

→ links-assoziativ,

← rechts-assoziativ

Mehrere Programmblöcke

Sind mehrere Programmblöcke vorhanden, werden diese nacheinander ausgeführt:

```
{print "Mitarbeiter " $1;}
/Carol/ {print "   Carol gefunden"; dabei="ja"}
END {print "Carol ist dabei: " dabei}
```

Die ersten 2 Zeilen (der Main-Block) werden im Loop für jede Zeile des Inputs ausgeführt.

Der print in der zweiten Zeile wird nur bedingt durchgeführt, und zwar wenn der String Carol in der Textzeile gefunden wird. Die Bedingung ist ein Pattern-Action Statement. Siehe auch: Pattern-Action Statement (Seite 74)

Der zweite Block (END) wird nach dem Loop und somit nur einmal durchgeführt.

Ergibt diesen Output (siehe auch mitarbeiter.txt Seite 73):

```
$ awk -f skript.awk mitarbeiter.txt
Mitarbeiter Carol
   Carol gefunden
Mitarbeiter Mike
Mitarbeiter Tom
Mitarbeiter Jerry
Carol ist dabei: ja
```

Bedingter Programmblock

Abfrage auf Anzahl der Spalten mit der Variable NF:

```
$1=="Carol" {print "Mitarbeiter " $1 " hat " $2 " Stunden gearbeitet.";}
NF!=3 {print "Achtung! Mitarbeiter " $1 " hat statt 3 Spalten, " NF " Spalten ."}
```

Der Block wird durch das **Pattern-Action-Statement** NF!=3 bedingt (siehe Seite 74).

Ergibt diesen Output:

```
$ awk -f skript.awk mitarbeiter.txt
Mitarbeiter Carol hat 3 Stunden gearbeitet.
Achtung! Mitarbeiter Tom hat statt 3 Spalten, 4 Spalten .
```

Statt dem **Pattern-Action-Statement** kann die Bedingung auch im Programmblock (innerhalb der geschweiften Klammern) abgefragt werden.

Mit dem Keyworfd **if**:

```
$1=="Carol" {print "Mitarbeiter " $1 " hat " $2 " Stunden gearbeitet.";}

{
 if (NF!=3)
 {
 print "Achtung! Mitarbeiter " $1 " hat statt 3 Spalten, " NF " Spalten ."
 }
}
```

Ergibt diesen Output (siehe auch mitarbeiter.txt Seite 73):

```
wilma@windows10 /
$ awk -f skript.awk mitarbeiter.txt
Mitarbeiter Carol hat 3 Stunden gearbeitet.
Achtung! Mitarbeiter Tom hat statt 3 Spalten, 4 Spalten .
```

Regular Expressions

Search-Pattern-Bedingungen stehen in Slashes (/):

```
/f/          { print "Enthält den Buchstaben f" }
/foo/        { print "Enthält die Zeichenkette foo" }
```

Suche nach dem String **Carol** an beliebige Stelle in der Textzeile (siehe auch mitarbeiter.txt Seite 73):

```
/Carol/      {print "String Carol gefunden";}
```

Beispiele für Search-Pattern-Bedingungen mit **Regular Expressions Operatoren**:

```
/[0-9]+/     { print "Enthält Zahlen" }
/[A-Za-z]+/  { print "Enthält Buchstaben" }
/^$/         { print "Leere Zeile" }
```

Regular Expression Operatoren

^ Suche am Stringanfang nach dem String **foo** (der String muss mit **foo** beginnen):

```
^foo
```

Der ^ Operator wird auch Anker genannt, da er nur am Beginn von einem String übereinstimmt.

```
/^Carol/ {print "Carol am Anfang der Zeile gefunden";}
```

$ Suche am Stringende nach dem String **foo** (der String muss mit **foo** enden):

```
foo$
```

Der $ Operator wird auch Anker genannt, da er nur am Ende von einem String übereinstimmt.

```
/Lehrling$/ {print "Lehrling am Ende der Zeile gefunden";}
```

. Der Punkt steht für **ein** (einzelnes) beliebiges Zeichen (matches any single character).
Inklusive vom newline character.

Escape Sequence (Fluchtsequenz)

Damit Steuerzeichen nicht interpretiert werden, müssen sie escaped werden.
Dies geschieht, indem der Escape-Character Backslash (\) vorangestellt wird. Auf Englisch wird das **literal** (selbstbedeutendes Symbol) genannt. Der Escape-Character wird auf deutsch Fluchtsymbol genannt.
Ein escaped Backslash ist ein **Literal Backslash**.
Siehe auch **ASCII Control Characters** Seite 225.

Als Beispiel ein String der in doppelten Anführungzeichen steht:

```
"Er ist cool."
```

Kommt ein Anführungszeichen im String vor, muss dieses escaped werden:

```
"Er ist \"cool\"."
```

Das Beispiel in einer Action:

```
'{ print " Er ist \"cool\"." }'
```

Ergibt als Output:

```
Er ist cool.
```

Wenn kein Steuerzeichen escaped wird, erscheint eine Warnung:

```
{print "Carol \goder Tom gefunden"}
```

Ergibt diesen Output:

```
awk: skript.awk:1: Warnung: Fluchtsequenz »\g« wird wie ein normales »g« behandelt
```

Control Character	Beschreibung	ASCII-Wert	Kurzbeschreibung
\\	literal backslash		
\a	alert (bell)	7	BEL
\b	backspace	8	BS
\f	formfeed	12	FF
\n	newline (line feed)	10	LF
\r	carriage return	13	CR
\h	horizontal tab	9	HT
\v	vertical tab	11	VT
nnn	octal ascii value nnn		
\x*nn*	hexadecimal ascii value nn		

Operatoren für einen Regular-Expression Vergleich

Mit diesen Operatoren kann auf Erfüllung überprüft werden:

```
~     der Regex-Ausdruck wird gefunden (match)
!~    der Regex-Ausdruck wird nicht gefunden (no match)
```

Wenn in der 1. Spalte der String Carol vorkommt (siehe auch mitarbeiter.txt Seite 73):

```
$1 ~ /Carol/ {print "Carol in 1. Spalte gefunden"}
```

Wenn in der 1. Spalte der String Carol NICHT vorkommt:

```
$1 !~ /Carol/ {print "Carol in 1. Spalte nicht gefunden"}
```

String-Funktionen

print

Ein print ohne weitere Angaben, printet die gesamt Zeile:

```
{print}
```

Erzeugt als Output:

```
Carol   3       0
Mike    3.75    0
Tom     4.01    10      Lehrling
Jerry   5       20
```

Die ganze Zeile kann auch mit $0 geprintet werden:

```
{print $0}
```

Die ersten zwei Spalten (Spalte 1 mit der Variable $1, Spalte 2 mit $2) anzeigen:

```
{print  $1 $2}s
```

Erzeugt als Output:

```
Carol3
Mike3.75
Tom4.01
Jerry5
```

Zwischen $1 und $2 ein Leerzeichen einfügen:

```
{print  $1 " " $2}
```

Erzeugt als Output:

```
Carol 3
Mike 3.75
Tom 4.01
Jerry 5
```

Ein Komma zwischen zwei Variablen erzeugt ebenfalls ein Leerzeichen:

```
{print  $1 , $2}
```

Erzeugt als Output:

```
Carol 3
Mike 3.75
Tom 4.01
Jerry 5
```

Mit Zeichenketten:

```
{print "Mitarbeiter",$1,"hat ",$3,"Stunden (je ",$2,"euro):",$2*$3,"euro lohn" }
```

Erzeugt als Output:

```
Mitarbeiter Carol hat  0 Stunden (je  3 euro): 0 euro lohn
Mitarbeiter Mike hat  0 Stunden (je  3.75 euro): 0 euro lohn
Mitarbeiter Tom hat  10 Stunden (je  4.01 euro): 40.1 euro lohn
Mitarbeiter Jerry hat  20 Stunden (je  5 euro): 100 euro lohn
```

Siehe auch Konkatenation Seite 82.

printf()

Mit der Funktion printf() kann der Output formatiert werden. Das Statement hat die Form:

```
printf  (format,  value)
```

Im String format (1. Parameter) befindet sich der Output und Formatierungsangaben. Darin sind Platzhalter für die echten Werten (2. Parameter) enthalten:

```
%i        Integerzahl
%d        Integerzahl
%f        Fließkommazahl

%o        Oktalzahl
%x        Hexadezimalzahl
%c        ASCII-Number Character
%s        String
```

Der Platzhalter %s steht für einen String und erhält den Wert aus $1:

```
{ printf("Mitarbeiter: %s\n",  $1) }
```

Im Gegensatz zu print, muss ein Zeilenumbruch explizit mit \n gesetzt werden.
Der Output (siehe auch mitarbeiter.txt Seite 73):

```
Mitarbeiter: Mike
Mitarbeiter: Tom
Mitarbeiter: Jerry
```

strftime()

Die Funktion strftime erzeugt einen frei wählbaren Datumstring.

strftime(formatstring [, timestamp])

Der optionale zweite Parameter ist der UNIX-Zeitstempel (Anzahl der Sekunden seit 1970-01-01 00:00:00).

Gawk: %k %l %N %o %s %v

Kürzel	Bedeutung	von-bis		Beispiel
%S	**Sekunde**	00-69	Numeric	59
%M	**Minute**	00-59	Numeric	59
%H	**Stunde** wenn einstellig, mit 0 davor	00-23	Numeric	23
%k	Stunde wenn einstellig, mit Leerzeichen davor	0-23		3
%l	Stunde	1-12		
%I	Stunde (siehe %p AM/PM)	1-12	Numeric	12
%d	**Tag** des Monats wenn einstellig, mit 0 davor	01-31	Numeric	31
%e	Tag des Monats wenn einstellig, mit Leerzeichen davor		Numeric	1
%j	**Tag** des Jahres	001-366	Numeric	065
%u	**Wochentag** (Montag = 1)	1-7	Numeric	7
%w	Wochentag (Sonntag = 0)	0-6	Numeric	1
%A	Wochentag		String	Saturday
%a	Wochentag (kurze Schreibweise)		String	Sat
%W	Wochennummer (Montag als erster Tag der Woche)	00-53	Numeric	03
%U	Wochennummer (Sonntag erster Tag der Woche)	00-53	Numeric	53
%u	**Wochentag** (Montag = 1)	1-7	Numeric	7
%w	Wochentag (Sonntag = 0)	0-6	Numeric	1
%A	Wochentag		String	Saturday
%a	Wochentag (kurze Schreibweise)		String	Sat
%W	Wochennummer (Montag als erster Tag der Woche)	00-53	Numeric	03
%U	Wochennummer (Sonntag erster Tag der Woche)	00-53	Numeric	53

%m	**Monat**	01-12	Numeric	12
%B	Monatsname		String	January
%b	Monatsname (kurze Schreibweise)		String	Jan
%Y	**Jahreszahl** vierstellig		Numeric	2066
%y	Jahreszahl zweistellig	00-99	Numeric	24 (2066)
%C	Jahrhundert		Numeric	20
%t	Ein Tab-Character (\t)			
%p	lokale Entsprechung für eine 12-Stunden-Uhr (AM/PM)			
%r	lokale Entsprechung für eine 12-Stunden-Uhr %I:%M:%S %p			
%c	lokale Entsprechung für die Zeit- und Datumsdarstellung			Di, 12. Apr 2066 17:26:44
%x	Lokale Datumsdarstellung `%A %B %d %Y' in the "C" locale			02.12.2066
%X	Lokale Zeitdarstellung `%T' in the "C" locale			20:15:00
%%	Prozentzeichen-Literal			%
%R	%H:%M			18:02
%F	%Y-%m-%d ISO 8601			2064-12-20
%T	%H:%M:%S			18:02:40
%z	Zeitzone im Format +HHMM notwendig für RFC 822, RFC 1036			+0100
%Z	Zeitzonenname			CET

Beispiel mit nur einem Parameter, dem Formatstring:

```
print strftime("%Y-%m-%d %H:%M:%S")   # 2066-11-03 23:55:45
```

Beispiel mit dem zweiten Parameter, dem Zeitstempel:

```
jetzt=systime();                # gegenwärtige Zeit (timestamp)
                        # 3. Nov 2066
print strftime("%Y-%m-%d %H:%M:%S" , jetzt)# 2066-11-03 23:55:45
```

sub()

Ersetze im target **das erste Vorkommen** von regexp durch replacement.

sub (regexp, replacement [, target])

Ist kein target gegeben, so wird in $0 ersetzt. regexp kann ein String und ein regulärer sein.
Der Rückgabewert der Funktion ist die Anzahl der Ersetzungen.

Ersetze in der Variable target den String aa durch den String xx

```
target = "aa aa bb cc"
n = sub ("aa", "xx", target)
print "target nach Ersetzung: " target
print "Anzahl Ersetzungen: " n
```

Ergibt als Output:

```
target nach Ersetzung: xx aa bb cc
Anzahl Ersetzungen: 1
```

Um alle Vorkommen zu ersetzen, muss die Funktion gsub() verwendet werden. Siehe Seite 92

Wenn der 3. Parameter fehlt, wird $0 (ganze Zeile) verwendet: Ersetze in der ganzen Zeile den String **12** durch den String **66**:

```
{ sub("12","66"); print }
```

Ein Windows-File (\r\n) in ein Linux-File (\n) wandeln:
Ersetze am Zeilenende das \r (carriage return) durch einen Leerstring (das cr wird also entfernt):

```
{ sub(/\r$/,""); print }
```

Ein Linux-File (\n) in ein Windows-File (\r\n) wandeln:
Setze an das Zeilenende ein \r (durch das nachfolgende print wird jede Output-Zeile durch \r\n beendet).
Mit $ wird der Anker an das Zeilenende gesetzt (und ein Leerstring durch den String "\r" ersetzt):

```
{ sub(/$/,"\r"); print }
```

Left-Trim:
Entferne am Zeilenanfang Tabs und Spaces. Mit ^ wird der Anker an den Beginn gesetzt.
Mit `[ \t]+` werden Leerzeichen und Tabs gesucht. Das + steht für "**ein oder mehrere Tabs oder Leerzeichen**".

```
{ sub(/^[ \t]+/,""); print }
```

Right-Trim:
Entferne am Zeilenende Tabs und Spaces. Mit $ wird der Anker an das Zeilenende gesetzt.
Mit `[ \t]+` werden Leerzeichen und Tabs gesucht. Das + steht für "**ein oder mehrere Tabs oder Leerzeichen**".

```
{sub(/[ \t]+$/,""); print}
```

gsub()

Ersetze im target **alle Vorkommen** von regexp durch replacement.

sub (regexp, replacement [, target])

Ist kein target gegeben, so wird in $0 ersetzt. regexp kann ein String und ein regulärer Ausdruck sein.

Der Rückgabewert der Funktion ist die Anzahl der Ersetzungen.

Ersetze in der Variable target den String aa durch den String xx

```
target = "aa aa bb cc"
n = gsub ("aa", "xx", target)
print "target nach Ersetzung: " target
print "Anzahl Ersetzungen: "
```

Ergibt als Output:

```
target nach Ersetzung: xx xx bb cc
Anzahl Ersetzungen: 2
```

Soll nur das erste Vorkommen (statt alle) ersetzt werden, muss die Funktion sub() verwendet werden. Siehe Seite 88

length()

Die gawk-Funktion **length** gibt die Länge vom String zurück.

Zeilenlänge anzeigen:

```
{print "Zeile " NR " hat " length($0) " Zeichen"}
```

Wenn statt einem String, eine Zahl abgefragt wird:

```
{a=.5;  print length(a) " Zeichen"}  # 3 Zeichen
{a=0.5; print length(a) " Zeichen"}  # 3 Zeichen
{a=-0.5; print length(a) " Zeichen"} # 4 Zeichen
```

index()

Sucht einen String in einem anderen String, und gibt die Position (Index) vom ersten Suchtreffer zurück.

Die Suchrichtung ist von links nach rechts. Die Zählung beginnt bei 1. Bei keinem Treffer gibt die Funktion 0 zurück.

Suche die Zeichenkette King in der String-Variable a:

```
{a="Carol King ist tot."; print "Position von King: " index(a,"ist") } # Position ist 7
```

toupper()

Wandelt einen String in Großbuchstaben um (Rückgabewert):

```
BEGIN {
print toupper("BigMama"); # BIGMAMA
}
```

tolower()

Wandelt einen String in Kleinbuchstaben um (Rückgabewert):

```
BEGIN {
print tolower("BigMama"); # bigmama
}
```

Sie auch IGNORECASE Seite 94.

match()

Die Funktion sucht die Regular-Expression **regexp** im String **stringtarget**. Der Return-Wert ist die Position vom **longest, leftmost substring** (also das erste Vorkommen).

```
match(stringtarget, regexp [, ergebnisarray])
```

Bei keinem Treffer gibt die Funktion 0 zurück (gleichzeitig wird RSTART auf 0 und RLENGTH auf -1 gesetzt).

ergebnisarray ist eine gawk-Erweiterung. Sie enthält alle gefundenen Übereinstimmungen

Der Rückgabewert wird auch gleichzeitig in die built-in Variable RSTART geschrieben (default ist 0).
In der built-in Variable RLENGTH ist die Zeichenlänge des gefundenen Strings zu finden (default ist -1).

Beispiel:

```
BEGIN{
x="Mrs. Carol King";
if (match (x,/Carol/))
 {
    print "Carol an Position " match (x,/Carol/) " (=" RSTART ") gefunden";
    print "Laenge vom String: " RLENGTH
 }
}
```

Ergibt als Output:

```
Carol an Position 6 (=6) gefunden
Laenge vom String: 5
```

Ein Beispiel mit einem Ergebnis-Array (gesucht wird nach *Alice* oder *King*):

```
BEGIN{
x="Mrs. Alice King (Carolina)";
match (x,/Carol|King/,ergebnis)

 for (wert in ergebnis)
 {
    print "wert " wert " / " ergebnis[wert]
 }

}
```

Ergibt als Output:

```
wert 0start / 12
wert 0length / 4
wert 0 / King
```

split()

Die Funktion string trennt den String **targetstring**, anhand des optionalen **fieldseperator.** Der Rückgabewert ist die Größe des Arrays (höchste Indexnummer) . Der erste Teilstring ist in **ergebnisarray[1].**

```
split (targetstring, ergebnisarray [, fieldseperator])
```

Ohne Angabe vom 3. Parameter (fieldseperator) wird das Leerzeichen als Trennzeichen verwendet.

```
blub="aa**bb**cc**dd"
anz = split(blub,arr,"**")
for (i=1;i<=anz ;i++)
{
print "i von " anz ": " arr[i]
}
```

Ergibt als Output:

```
1 von 4: aa
2 von 4: bb
3 von 4: cc
4 von 4: dd
```

IGNORECASE

Mit der der builtin (predefined) Variable von Gawk kann die Berücksichtigung der Groß-Kleinschreibung verändert werden. Default ist 0.

IGNORECASE=1;	Groß-Kleinschreibung spielt keine Rolle	case insensitive
IGNORECASE=0;	Groß-Kleinschreibung berücksichtigen	case sensitive

IGNORECASE auf 0, also Groß-Kleinschreibung nicht ignorieren:

```
BEGIN{
$0="Mrs. Carol King";
IGNORECASE=0;
if (match ($0,/carol/)) { print "carol gefunden";  }
 else {print "carol nicht gefunden";}
}
```

Output:

```
carol nicht gefunden
```

IGNORECASE auf 1, also Groß-Kleinschreibung ignorieren:

```
BEGIN{
$0="Mrs. Carol King";
IGNORECASE=1;
if (match ($0,/carol/)) { print "carol gefunden";  }
 else {print "carol nicht gefunden";}
}
```

Output:

```
carol gefunden
```

Siehe auch toupper() Seite 92 und tolower() Seite 92.

Mathematische Funktionen

Funktion	Bedeutung/Rückgabewert
atan2(x,y)	Arcustangens von x/y in Radian
cos(x)	Cosinus in Radian
exp(x)	Exponentialfunktion
int(x)	Ganzzahliger Anteil von x (Truncate)
log(x)	Natürlicher Logarithmus zur Basis e
rand()	Pseudo-Zufallszahl zwischen 0 und 1
sin(x)	Sinus in Radian
sqrt(x)	Quadratwurzel
srand(x)	Setzt den Startwert für Zufallszahlen. Bei keiner Angabe wird die aktuelle Zeit verwendet

strtonum()

Die gawk-Funktion strtonum() wandelt einen Hex- oder Octalwert in einen Dezimalwert um.

```
BEGIN{
  hex="0xa9"
  oct="0251"
  print "hexadezimal: " hex
  print "octalo: " oct
  print "dezimal: " strtonum(hex) " (" strtonum(oct) ")"
}
```

Ergibt als Output:

```
hexadezimal: 0xa9
octalo: 0251
dezimal: 169 (169)
```

Würfel-Funktion

Die Funktion wuerfel() hat als Parameter die Anzahl der Würfelseiten:

```
function wuerfel(wuerfelseiten)
  {
  srand(systime());
  return 1 + int(rand() * wuerfelseiten)
  }
BEGIN{
  print wuerfel(6);
}
```

Möglicher Output:

```
1
```

ord()

Die gawk-Funktion strtonum() wandelt einen Hex- oder Octalwert in einen Dezimalwert um.

Array-Funktionen

Gawk bietet zwei Array-Funktionen an: asort() für das Sortieren nach den Arraydaten und asorti() für das Sortieren nach den Indexwertbezeichnungen. In beiden Funktionen wird das Array neu gebaut und neue Indexwerte verwegeben. Bei asorti() werden aus den Indexwertbezeichnung die Daten.

asort()

Die gawk Funktion asort() sortiert den Arrayinhalt (und baut das Array komplett neu auf).

Die Array-Indexwerte beginnen bei 1. Der Rückgabewert ist die Anzahl der Einträge (Arraygröße].

Der zweite Parameter ist optional: die Zielvariable (ebenfalls ein Array). Damit wird das Original-Array nicht zerstört.

```
anzahl = asort (arrayvariable [, Zielarrayvariable] )
```

Beispiel:

```
BEGIN{
        foo[9]=101
        foo[11]=550
        foo[7]=343
        foo[115]=221
        #a = asort(foo)
        #a = asorti(foo)
        print "eintraege: a"
        for (wert in foo)
        {
        s++
        print "Schleife " s " von " a " mit Indexwert " wert  " hat Wert " foo[wert]
        }
}
```

Ergibt als Output:

```
Schleife 1 von  mit Indexwert 7 hat Wert 343
Schleife 2 von  mit Indexwert 9 hat Wert 101
Schleife 3 von  mit Indexwert 11 hat Wert 550
Schleife 4 von  mit Indexwert 115 hat Wert 221
```

Mit aktiviertem asort() in Zeile 6:

```
Schleife 1 von 4 mit Indexwert 1 hat Wert 101
Schleife 2 von 4 mit Indexwert 2 hat Wert 221
Schleife 3 von 4 mit Indexwert 3 hat Wert 343
Schleife 4 von 4 mit Indexwert 4 hat Wert 550
```

asorti()

Die Array-Funkton asorti() sortiert die Indexwerte (und nicht die eigentlichen Werte). Die Sortierung erfolgt alphanummerisch. Darum kommt der Indexwert 100 vor 2.

Die Array-Indexwerte beginnen bei 1. Der Rückgabewert ist die Anzahl der Einträge (Arraygröße].

Der zweite Parameter ist optional: die Zielvariable (ebenfalls ein Array). Damit wird das Original-Array nicht zerstört.

Mit aktiviertem asorti() in Zeile 7 (siehe Code-Beispiel asort):

```
Schleife 1 von 4 mit Indexwert 1 hat Wert 11
Schleife 2 von 4 mit Indexwert 2 hat Wert 115
Schleife 3 von 4 mit Indexwert 3 hat Wert 7
Schleife 4 von 4 mit Indexwert 4 hat Wert 9
```

delete

Mit dem Keyword **delete** kann ein einzelner Indexeintrag gelöscht werden:

```
delete arrayvariable [indexwert]
```

Alernativ auch der Aufruf als Funktion:

```
delete (arrayvariable [indexwert])
```

Beispiel:

```
BEGIN{
foo[0]="carol"
foo[1]="king"
delete foo[1]  # delete foo  leer die ganze Arrayvariable

if (1 in foo) {print "foo[1] gibt es"} else {print "foo[1] gibt es nicht"}
}
```

Ergibt als Output:

```
foo[1] gibt es nicht
```

Ohne Angabe von einem Indexwert wird das gesamte Array geleert. Die Variable bleibt weiterhin ein Array und kann nicht als Skalar wiederverwendet werden:

```
delete arrayvariable
```

Alternativ auch der Aufruf als Funktion:

```
delete (arrayvariable)
```

Beispiel:

```
BEGIN{
foo[0]="carol"
foo[1]="king"
delete (foo)
if (1 in foo) {print "foo[1] gibt es"} else {print "foo[1] gibt es nicht"}
foo=3;
}
```

Eine Verwendung als Skalar erzeugt einen Error:

```
foo[1] gibt es nicht
awk: skript.awk:6: Fatal: Es wird versucht, das Feld »foo« in einem Skalarkontext zu verwenden
```

System-Funktionen

system()

Im einzigen Parameter ist die Shell-Anweisung enthalten. Der Rückgabewert ist der Exit-Status der Anweisung.

Mit unlink die Datei m2.txt löschen:

```
BEGIN{
ok=system ("unlink m2.txt");
print "ok " ok
}
```

Der Output:

```
ok 0
```

Wird das Skript noch einmal aufgerufen (Datei m2.txt wurde bereits erfolgreich gelöscht):

```
unlink: Entfernen (unlink) von 'm2.txt' nicht möglich: No such file or directory
ok 1
```

systime()

Die Funktion systime gibt die Anzahl der Sekunden seit 1970-01-01 00:00:00 (timestamp (Zeitstempel) zurück.

Keine Paramenter.

Zeit-Funktionen

systime()

Die Funktion systime gibt die Anzahl der Sekunden seit 1970-01-01 00:00:00 (timestamp (Zeitstempel) zurück.

Keine Paramenter.

strftime()

Rückgabewert ist ein frei wählbarer Datumstring. Parameter ist ein timestamp (Zeitstempel). Ohne Angabe des Parameter wird der gegenwärtige Zeitstempel genommen.

strftime(Formatstring, [timestamp [UTC]])

Siehe Seite 88.

mktime()

Die gawk-Funktion hat nur einen Parameter, ein Datumstring in diesem Format:

```
YYYY MM DD HH MM SS
```

Mit optionalen Daylight-Saving-Flag:

Der Rückgabewert ist ein UNIX-Timestamp (Anzahl der Sekunden seit 1970-01-01 00:00:00).

function()

Mit dem Keyword **function** wird eine eigene Funktion definiert. Nach dem Keyword muss sofort der Funktionsname folgen (ein Whitespace wird mit einem Error bestraft). In runden Klammern folgen optional Parameternamen:

```
function Funktionsname ([Variablename1, Variablename2, ...])
  {

  }
```

Funktionsdefinition ohne Parameter:

```
function sprich()
 {
 }
```

Funktionsdefinition mit einem Parameter:

```
function sprich(text)
       {
          print text
       }
BEGIN{
sprich("Hallo");
}
```

Ergibt als Output:

```
Hallo
```

Funktionsdefinition mit einem Rückgabewert (mit dem Keyword return):

```
function sprich(text)
       {
          return "Ich sprach " text
       }
BEGIN
 {
 print sprich("Hallo");
}
```

Funktionsdefinition mit mehreren Parameter:

```
function sprich(spruch, name, geschlecht) {
       if (geschlecht=="w") {anrede="Frau" }
          else {anrede="Herr";}
       return spruch " " anrede " " name;
       }

BEGIN{
print sprich("Hallo","Carol King", "w");
}
```

Ergibt als Output:

```
Hallo Frau Carol King
```

Kommandozeilen-Argumente (ARGV, ARGC, ARGIND)

Das built-in Array ARGV enthält die Argumente, mit der awk in der Shell aufgerufen worden ist.

Die Anzahl der Argumente (also die Arraygröße) ist in der built-in Variable ARGC (Numeric).

Die Option -f ist nicht dabei:

```
wilma@windows10 /
$ awk -f skript.awk mitarbeiter.txt klammern.txt
0: awk
1: mitarbeiter.txt
2: klammern.txt
```

Das dazugehörige Skript skript.awk:

```
BEGIN {
for (i=0;i<ARGC;i++)
 {
    print i ": " ARGV[i];
 }
}
```

Gawk besitzt die built-in Variable ARGIND (Numeric). Sie enthält den Indexwert des Array ARGV vom aktuellen Textfile.

Der Dateiname vom Array sollte identisch mit FILENAME sein:

```
print "Aktueller Dateiname im Main-Loop: " FILENAME " == " ARGV[ARGIND]
```

Erzeugt diesen Output:

```
$ awk -f skript.awk mitarbeiter.txt klammern.txt
Aktueller Dateiname im Main-Loop [1]: mitarbeiter.txt
Aktueller Dateiname im Main-Loop [2]: klammern.txt
```

Kontrollstrukturen

Kontrollstrukturen kontrollieren den Programmablauf:

1	Programmblock nur bedingt aufrufen	if-else- Verzweigungen
2	Programmblock wiederholen	for- und while-Schleife (Schleifenabbruch mit break)
3	Programmblock abbrechen	exit

Es gibt die Möglichkeit, Programmteile nur unter gewissen Bedingungen auszuführen oder Programmteile zu wiederholen.

if

Mit dem Keyword if wird ein Ausdruck auf true oder false überprüft:

```
if (Bedingung) Anweisung;
```

Ist der Ausdruck Bedingung erfüllt, wird die Anweisung ausgeführt (ansonsten wird die Anweisung übersprungen). Bei mehreren Anweisungen werden die Anweisungen in geschweiften Klammern gesetzt (zu einem Block zusammengefasst):

```
if (Bedingung)
{
Anweisung1
Anweisung2
}
```

Beispiel:

```
if (2+2==4) {print "2+2 ist 4"}
```

Siehe **Vergleichsoperatoren** (Conditional Operators) Seite 80.

Das Keyword else definiert einen **Alternativblock** (wenn der Ausdruck nicht erfüllt ist). Der else-Block wird nur dann ausgeführt, wenn die if-Bedingung false ist.

```
if (Bedingung) Anweisung;
else Anweisung;
```

Beispiel:

```
if (2+2==5) {print "2+2 ist 5"}
else {print "2+2 ist nicht 5"}
```

Mit dem Keyword else if können mehrere Abfragen nacheinander gestellt werden. Bei der ersten erfüllten Bedingung werden die restlichen else if nicht mehr abgefragt:

```
if (2+2==3) {print "2+2 ist 3"}
else if (2+2==4) {print "2+2 ist 4"}
else if (2+2==5) {print "2+2 ist 5"}
else {"print 2+2 ist irgendwas"}
```

Ergibt als Output:

```
2+2 ist 4
```

Siehe auch **ternärer Operator** Seite 81.

for-Schleife

```
BEGIN{
blub="aa**bb**cc**dd"
anz = split(blub,arr,"**")
for (i=1;i<=anz ;i++)
       {
       print i " von " anz ": " arr[i]
       }
}
```

Ergibt als Output:

```
1 von 4: aa
2 von 4: bb
3 von 4: cc
4 von 4: dd
```

break

Mit dem keyword break kann die Schleife abgebrochen werden:

```
BEGIN{
blub="aa**bb**cc**dd"
anz = split(blub,arr,"**")
for (i=1;i<=anz ;i++)
  {
       if (i>2) {break;}
       print i " von " anz ": " arr[i]
       }
}
```

Ergibt als Output:

```
1 von 4: aa
2 von 4: bb
```

continue

Mit dem Keyword continue zum Schleifenblock springen.

```
BEGIN{
for (p=4;p>=0;p--)   {
    if (p==2) {continue;}
    print "p ist " p
  }
}
```

Ergibt als Output:

```
p ist 4
p ist 3
p ist 1
p ist 0
```

in

Mit dem Keyword **in** in einer for-Schleife alle vorhandenen Arrayelemente einer nummerischen Array-Variable durchlaufen:

```
BEGIN{
foo[0]="a"; foo[1]="b"; foo[6]="d"; foo[2]="c";
for (indexwert in foo) {
print "indexwert " indexwert " = " foo[indexwert]}
}
```

Ergibt als Output:

```
indexwert 0 = a
indexwert 1 = b
indexwert 2 = c
indexwert 6 = d
```

while-Schleife

```
BEGIN{
p=4
while (p>0)
        {
                print "p ist " p
                p--
        }
}
```

Ergibt als Output:

```
p ist 4
p ist 3
p ist 2
p ist 1
```

break

Mit dem keyword break kann die Schleife abgebrochen werden:

```
BEGIN{
p=4
while (p>0) {
                if (p<3) {break}
                print "p ist " p
                p--
        }
}
```

Ergibt als Output:

```
p ist 4
p ist 3
```

continue

Mit dem Keyword continue zum Schleifenkopf springen:

```
BEGIN{
p=4
while (p>0)  {
    p--
    if (p==2) {continue;}
    print "p ist " p
  }
}
```

Ergibt als Output:

```
p ist 3  p ist 1   p ist 0
```

do-while-Schleife

Eine auf den Kopf gestellte while-Schleife:

```
BEGIN{
i = 0;
 do {
 ++i
 if (i==2) {continue;}
 print i;
 }
 while (i < 5)
}
```

Ergibt als Output:

```
1
3
4
5
```

exit

Das Keyword exit bricht den gegenwärtigen Programmblock ab, und springt zum END-Block. Ist kein END-Block vorhanden, wird das awk-Skript sofort beendet. Optional kann auch ein Wert (zB als Status-Code) mitgegeben werden.

exit [Ausdruck]

Der exit im BEGIN-Block bewirkt ein Springen zum END-Block (der Main-Loop wird nicht ausgeführt):

```
BEGIN{
print "anfang"
exit
}
{
print "main"
}
END {print "Ende"}
```

Ergibt diesen Output:

```
anfang
Ende
```

next

akw springt innerhalb vom Main-Loop zum nächsten Datensatz des gegenwärtigen Input.

Zeige nur Zeilen an, die nicht 4 Spalten besitzen:

```
{
if (NF != 4) {  next}
print $0
}
```

Innerhalb von einem BEGIN- oder END-Block wird nexty mit einem Error bestraft:

```
awk: skript.awk:2: error: »next« wird in BEGIN-Aktion verwendet
```

nextfile

awk springt zum nächsten Input-Textfile.

return

Die gegenwärtige Funktion wird beendet. Der Rückgabewert ist optional.

Das Keyword kann nur innerhalb einer Funktion verwendet werden:

```
awk: skript.awk:4: ^ »return« wird außerhalb einer Funktion verwendet
```

Siehe auch function() Seite 98.

Funktionsdefinition von sprich() mit 3 Übergabeparameter:

```
function sprich(spruch, name, geschlecht) {
       if (geschlecht=="w") {anrede="Frau" }
          else {anrede="Herr";}
       return spruch " " anrede " " name;
       }

BEGIN{
print sprich("Hallo","Carol King", "w");
}
```

Ergibt als Output:

```
Hallo Frau Carol King
```

awk Einzeiler

Vorkommen pro Zeile zählen

Zeige als Ergebnis, wie oft der String *Carol* in jeder Zeile vorkommt. Es gibt insgesamt 2 Blöcke.
Der Main-Loop-Block:

/Carol/ ist der Pattern, der nach dem String *Carol* sucht (in diesem Fall in jeder Zeile)

{m++} ist der dazugehörige Aktionsteil, der die Variable m hochzählt. Der Aktionsteil wird nur ausgeführt, wenn das Action-Pattern-Statement erfüllt ist.

Mit dem Keyword END startet der END-Block, der nur einmal ausgeführt wird und die Variable m ausgibt. Mit der Addition von 0 gibt es auch dann eine Anzeige (dynamische Typkonvertierung), wenn der String *Carol* nie gefunden worden ist und die Variable m ein Leerstring bleibt. Ein Leerstring + 0 erzeugt 0.

```
awk '/Carol/ { m++ }; END { print m+0 }'
```

Gesamt-Spaltenanzahl ermitteln

Zeige als Ergebnis, wie viele Spalten es im gesamten Textfile gibt.

Der Main-Loop-Block (in der builtin (predefined) Variable **NF** ist die Anzahl der vorhandenen Spalten):

es gibt kein Pattern, dh der Block `{gesamt=gesamt+NF}` wird immer für jede Zeile ausgeführt.

Mit dem Keyword END startet der END-Block. Der END-Block wird nur einmal ausgeführt und zeigt die Variable *gesamt* an (mit der Addition von 0 wird aus einem Leerstring eine 0).

```
awk '{ gesamt= gesamt + NF }; END { print gesamt+0 }'
```

Zeige die Zeile, die die längste 1. Spalte hat

Der Main-Loop-Block:

die Variable $1 enthält den String der 1. Spalte. $0 enthält die gesamt Zeile.
die Variable *maxzeile* die gegenwärtig längste Spalten-Zeile

Der END-Block zeigt die längste Spalte, gefolgt von der jeweiligen Zeile.

```
awk '$1 > max { max=$1; maxzeile=$0 }; END { print max, maxzeile }'
```

Zeige von jeder Zeile die letzte Spalte

Der Main-Loop-Block hat kein einschränkendes Pattern, dh der Block wird für jede Textzeile durchgeführt. Die Variable NF enthält die Anzahl der Spalten. $1 hat den Inhalt der 1. Spalte, $2 hat den Inhalt der 2. Spalte und $NF hat den Inhalt der letzten Spalte.

```
awk '{ print $NF }'
```

Enferne doppelte Zeilen

Das in im Action-Pattern !($0 in z) ist ein Array-Operator. Mit diesem Array-Operator wird überprüft, ob der Indexeintrag $0 (ganze Zeile) bereits existiert. Falls nein, wird er im Aktionteil mit z[$0] hinzugefügt. Das print zeigt die ganze Zeile an.

```
awk '!($0 in z) { z[$0]; print }'
```

Jede Zeile verdoppeln

Der Main-Loop besteht nur aus 2 print-Anweisungen.

Ein einzelnes print ist syntaktischer Zucker (Syntactical Sugar) für print $0.

```
awk '{print; print }'
```

Die 2. Spalte mit 10 multiplizieren

Der Main-Loop besteht nur aus 2 print-Anweisungen. Der Inhalt von $2 (2. Spalte) wird mit 10 mulitpliziert (awk erstellt nach der Zuweisung die Variable $0 neu). Ein einzelnes print ist syntaktischer Zucker für print $0 (in $0 ist die gesamte Zeile).

```
awk '{ $2*=10; print; }'
```

Tausche Spalte 1 mit Spalte 2

Der Inhalt von $2 (2. Spalte) wird in der Variable h zwischengespeichert. Ein einzelnes print ist syntaktischer Zucker (f syntactical sugar) für print $0 (in $0 ist die gesamte Zeile).

```
awk '{ h=$2; $2=$1; $1=h; print  }'
```

Zeilennummern am Zeilenanfang

Mit NR die gegenwärtige Zeilennummer voranstellen:

```
awk '{ print  NR ":", $0 }'
```

Nur Zeilen mit mehr als 10 Zeichen ausgeben

Mit der Funktion length() nur die Zeilen mit mehr als 10 Zeichen ausgeben:

```
awk '{ if (length($0)>10) {print} }'
```

Ohne Leerzeilen

Die gegenwärtige Zeile nur dann ausgeben, wenn die Anzahl der Spalten größer als 0 ist:

```
awk '{ if (NF>0) {print}  }'
```

getline

Mit der Anweisung getline kann aus unterschiedlichen Quellen eine Textzeile geholt werden.

Ohne weitere Angaben liest getline die nächste Datenzeile aus dem gegenwärtigen Input und legt sie in $0.

status = getline

Die Variable status kann einen dieser Rückgabewerte enthalten:

0	kein (weiterer) Datensatz vorhanden
-1	Error
1	ok

Beispiel:

```
{
 z++;
 print "Zeile " z " (NR ist " NR "): " $1;
 ok = getline ;
 if (ok) {
    print "Zeile " z " next (NR ist " NR "): " $1
    }
}
```

Der Output dazu (mitarbeiter.txt siehe Seite 73):

```
$ awk  -f skript.awk mitarbeiter.txt
Zeile 1 (NR ist 1): Carol
Zeile 1 next (NR ist 2): Mike
Zeile 2 (NR ist 3): Tom
Zeile 2 next (NR ist 4): Jerry
```

Statt in $0 kann der nächste Datensatz auch in einer Variable abgelegt werden:

status = getline *naechstezeile*

Diese Zeile gilt als abgeholt und wird vom Main-Loop nicht mehr gelesen.

```
{
 z++;
 print "Zeile " z " (NR ist " NR "): " $1;
 ok = getline nextzeile;
 if (ok) {
    print "Zeile " z " next (NR ist " NR "): " nextzeile
    }
}
```

Der Output dazu:

```
$ awk  -f skript.awk mitarbeiter.txt
Zeile 1 (NR ist 1): Carol
Zeile 1 next (NR ist 2): Mike           3.75    0
Zeile 2 (NR ist 3): Tom
Zeile 2 next (NR ist 4): Jerry  5       20
```

externes File

Mit < wird eine externe Quelle angegeben:

status = getline < "*filename*"

Ist der Filename ein "-" wird der nächste Datensatz vom Terminal geholt.

Im BEGIN-Block wird mit einer while-Schleife das ganze Textfile mitarbeiter.txt ausgelesen und die 1. Spalte angezeigt:

```
BEGIN{
while (getline < "mitarbeiter.txt" )
  {
     z++;
     print "Zeile " z " (NR ist " NR "): " $1;

  }
}
```

Der Output dazu:

```
$ awk  -f skript.awk mitarbeiter.txt
Zeile 1 (NR ist 0): Carol
Zeile 2 (NR ist 0): Mike
Zeile 3 (NR ist 0): Tom
Zeile 4 (NR ist 0): Jerry
```

pipe

In Kombination mit | wird das Ergebnis der Anweisung in $0 abgelegt.

Anweisung | getline

Ein Bespiel mit der Shell-Anweisung date:

```
BEGIN{
"date" | getline
print $0
close("date")
}
```

Der Output dazu:

```
$ awk  -f skript.awk mitarbeiter.txt
Do, 14. Apr 2066 15:46:01
```

close()

Die Funktion close() schließt geöffnete Verbindungen. Übergabeparameter ist der Filehandler.
Files und Pipes bleiben während der Laufzeit solange offen, bis sie explizit geschlossen werden.

Close liefert zwei mögliche Rückgabewerte:

Beispiel1

Mit getline die 1. Zeile vom Ergebnis der Shell-Anweisung (date) lesen:

```
BEGIN{
    "date" | getline
    if (!NF) {exit -1;}
    print "Datum: ", $0
    error=close("date")
if (error) {print "close ging schief"} else {print "close ok"}
}
```

Beispiel2

Mit print eine Zeichenkette in ein File schreiben. Wenn das File bereits existiert, wird der String dazugeschrieben:

```
BEGIN{
print "bin der inhalt" > "foo.txt"
error= close ("foo.txt")
if (error) {print "close ging schief"} else {print "close ok"}
       }
```

Beispiel3

Mit Print die Zeichenkette per Pipe an die Shell-Anweisung sort übergeben:

```
BEGIN{
 print "b\n\na\nx\n0" | "sort > sorted.txt"
 error = close ("sort > sorted.txt")
 print error
 if (error) {print "close ging schief"} else {print "close ok"}
}
```

Vergleich von Nawk und Gawk

		NAWK	GAWK
FNR	File Number of the Record (bei mehreren Files wird der Zähler nicht zurückgesetzt)	Ja	Ja
ARGC	Total number or arguments passed to awk script. Siehe Seite 95	Ja	Ja
ARGV	Array containing all awk script arguments. Siehe Seite 95	Ja	Ja
ARGIND	Index to ARGV to retrieve the current file name. Siehe Seite 95	Nein	Ja
SUBSEP	Subscript separator for array indexes. Siehe Seite 78	Ja	Ja
RSTART	Match function sets RSTART with the starting location of str1 in str2. Siehe Seite 92	Ja	Ja
RLENGTH	Match function sets RLENGTH with length of the str1. Siehe Seite 92	Ja	Ja
OFMT	Awk uses this to decide how to print values. Default is "%.6g"	Ja	Ja
ENVIRON	Array containing all environment variables and values	Nein	Ja
IGNORECASE	Default is 0 (case sensitive). When set to 1, it is case insensitive for string and reg-ex comparisons.	Nein	Ja
ERRNO	Contains error message of an I/O operation. e.g. while using getline function.	Nein	Ja
BINMODE n	Set binary mode for I/O. n can be 1 (input files), 2(output files), or 3(all files)	Nein	Ja
CONVFMT	The format used while converting number to string. Default ist meist "%.6g"	Nein	Ja
FIELDWIDTHS n	n is a space delimited number that indicates the column widths. If this is available, gawk uses this instead of FS.	Nein	Ja
LINT n	n can be a number. When n is a nonzero number (indicating true), gawk will displays fatal, invalid, or warning lint messages (same as –lint command line)	Nein	Ja
TEXTDOMAIN	This is used for internationalization.	Nein	Ja
sub(s1,s2,v)	In input string (var), str1 is replaced with str2, and output is in var. Siehe Seite 88	Ja	Ja
gsub()	Same as sub, but global. It does multiple substitutions on the same input string (var). Siehe Seite 92	Ja	Ja
match(s1,s2)	Returns positive number when str1 is present in str2. Siehe Seite 92	Ja	Ja
getline	Read next line from another input-file. Sets $0, NF Siehe Seite 108	Ja	Ja
toupper,tolower	Converts string to upper-case / to lower-case. Siehe Seite 92	Nein	Ja
|&	Two way communication between awk command and external process	Nein	Ja
systime()	Current time in epoch time. Combine with strftime. e.g. print strftime("%c",systime())	Nein	Ja

markdown

Kapitel 4

Markdown

Markdown dient zur einfachen Formatierung von Plain-Text. Sie ähnelt **Plain-Text Typesetting Systemen** wie troff und Lightweight Markup Languages (LML) wie HTML. Der Begriff Lightweight steht für eine leicht verständliche Auszeichnungssprache (verständlicher als XHTML/CSS).

Markdown wurde 2004 von John Gruber und Aaron Swartz entwickelt. Ihre Intention war eine **easy to write** und **easy to read** Formatierungssprache, die sich leicht in das HTML-Format konvertieren läßt.

Je nach Implementierung ist Markdown auch mit HTML-Anweisungen kombinierbar (inline HTML).

Zeilenumbruch

Es gibt zwei Möglichkeiten, um im Markdown-Output einen Zeilenumbruch zu erzeugen:

1. zwei Leerzeichen und ein Newline (\n) erzeugen als HTML ein
2. zwei Newlines (\n\n) erzeugen als HTML ein
3. manchmal (je nach Implementierung) erzeugt ein \\n ein

In einem Monospace-Block erzeugt bereits ein einzelnes Newline einen Zeilenumbruch (der Newline beendet den Block).

Möglichkeit 1: \n\n

Zwei Leerzeichen und ein Newline (\n) erzeugen für den HTML-Output ein

Möglichkeit 2: \n\n

Zwei Newlines (\n\n) erzeugen einen Zeilenumbruch. Für den HTML-Output wird ein   erzeugt.

```
Fehlt es mir

etwa an
Verrückten?
```

Erzeugt als Output:

```
Fehlt es mir
etwa an Verrückten?
```

Möglichkeit 3: \\n

Ein Backslash am Ende der Zeile und ein Newline (\n) erzeugen einen Zeilenumbruch.
Für den HTML-Output wird ein <br /> erzeugt.

Nicht in allen Implementierungen möglich.

```
Fehlt es mir\
etwa an
Verrückten?
```

Erzeugt als Output:

```
Fehlt es mir
etwa an Verrückten?
```

fett, kursiv, durchgestrichen

Für die Hervorhebung (Emphasis) von Buchstaben und Text, gibt es **keine einheitliche Definition.**

foo	kursiv (manchmal fett)	*italic*	foo
foo	fett	bold	foo
foo	kursiv (manchmal fett)	*italic*	foo
__foo__	fett	bold	foo
~~foo~~	durchgestrichen	~~strikethrough~~	<s>foo</s>
_f**oo**_	kursiv und fett	*foo*	

Die Tilde ist nicht in allen Implementierungen enthalten.

Beispiel:

```
Fehlt _**es**_ ~~uns~~ *mir* etwa an *Verrückten*, daß **ihr** diesen __Mann__ hergebracht habt?
```

Erzeugt:

```
Fehlt ***es*** ~~uns~~ *mir* etwa an *Verrückten*, daß **ihr** diesen **Mann** hergebracht habt?
```

Nach dem ersten Steuerzeichen darf kein Leerzeichen folgen.
Vor dem zweiten Steuerzeichen darf kein Leerzeichen stehen.

Inline-HTML

Kombiniert mit HTML:

```
<i>Fehlt</i> _**es**_ ~~uns~~ *mir* etwa <br>an *Verrückten*, daß **ihr** diesen __Mann__
hergebracht <b>habt</b>?
```

Erzeugt:

```
*Fehlt* ***es*** ~~uns~~ *mir* etwa
an *Verrückten*, daß **ihr** diesen **Mann** hergebracht **habt**?
```

Erzeugt als möglichen Output:

```
<p><i>Fehlt</i> <em><strong>es</strong></em> ~~uns~~ <em>mir</em> etwa <br>
an <em>Verrückten</em>, daß <strong>ihr</strong> diesen <strong>Mann</strong>
hergebracht <b>habt</b>?</p>
```

Überschrift (Headline)

Überschriften können auf 2 Möglichkeiten erzeugt werden: Mit Rauten und mit Gleichheitszeichen.

Möglichkeit 1 (#)

Die Anzahl der Rauten bestimmt die Schriftgröße. Je mehr Rauten (von 1 bis 5), umso kleiner die Schriftgröße.
Nach den Rauten folgt ein Leerzeichen, danach der Text der Überschrift. Rauten direkt nach dem Überschrift-Text werden igonoriert.

Die 6. Raute sollte nicht verwendet werden, da
kaum implementiert (obwohl von John Gruber spezifiziert).

##### Head 5	<h5>	Head 5
#### Head 4	<h4>	Head 4
### Head 3	<h3>	Head 3
## Head 2	<h2>	Head 2
# Head 1	<h1>	Head 1

Drei aufeinanderfolgende Rauten erzeugen eine <h3> Überschrift:

```
### bin eine Überschrift
und das Text
##### 2. Überschrift ####
#####Leerzeichen fehlt
```

erzeugt als Output:

bin eine Überschrift

und das Text

2. Überschrift

Leerzeichen fehlt

Möglichkeit 2 (-)

Der Text, der zur Überschrift werden soll, wird mit mindestens drei Gleichheitszeichen unterstrichen (entspricht etwa <h1>). Mindestens 3 Minuszeichen entsprechen etwa <h2>.
Space-Character sind in einigen Implementierungen erlaubt.

Head 1 <h1>Head 1</h1>
=====

Head 1

Head 2 <h2>Head 2</h2>

Head 2

Blockquote

Mit einem > in der ersten Spalte beginnt ein Blockquote. Er endet erst nach 2 aufeinanderfolgenden Newlines (\n\n).
Leerzeichen vor > sind erlaubt.
Ein Tab vor > erzeugt einen Monospace-Block.

```
Er sprach:
> Fehlt es mir etwa
an Verrückten, daß ihr diesen Mann hergebracht habt?
```

Erzeugt als Output:

```
Er sprach:

    > Fehlt es mir etwa
    an Verrückten, daß ihr diesen Mann hergebracht habt?
```

Und der mögliche HTML-Code dazu:

```
<p>Er sprach:</p><blockquote>
<p>Fehlt es mir etwa an Verrückten, daß ihr diesen Mann hergebracht habt?</p></blockquote>
```

Nested Blockqoute

Blockquotes dürfen mehrere Ebenen haben:

```
Er sprach:
> Fehlt es mir etwa
>> an Verrückten, daß ihr diesen Mann hergebracht habt?
```

Erzeugt als Output:

```
Er sprach:

    > Fehlt es mir etwa
    >> an Verrückten, daß ihr diesen Mann hergebracht habt?
```

Und der mögliche HTML-Code dazu:

```
<p>Er sprach:</p><blockquote><p>Fehlt es mir etwa </p>
<blockquote><p>an Verrückten, daß ihr diesen Mann hergebracht habt?</p></blockquote>
</blockquote>
```

Horizontale Linie

Ein Newline (\n), mindestens drei Sterne und ein weiteres Newline (\n) erzeugen eine horizontale Linie.

```
Text1
***
Text2
```

erzeugt diesen HTML-Code: `<p>Text1</p> <hr /> <p>Text2</p>`

Text1

Text2

Liste

Markdown kennt **Ordered Lists** (durchnummeriert) und **Unordered Lists**.

Unordered List

Um eine Auflistung zu beginnen, wird in die **erste Spalte ein *** und in die **zweite Spalte ein Leerzeichen** geschrieben (mit 1. Spalte ist das erste Zeichen in einer Zeile gemeint). In einigen Implementierungen ist vor dem ersten * ein Zeilenumbruch (\n) notwendig.

```
Test:
* foo
```

Der Erscheinungsform des Listenpunktes hängt von der Implementierung ab:

```
Test:
   • foo
```

Alternativ ist neben dem Stern oft auch ein Minus und ein Plus möglich:

```
Test:
* foo
+ foo2
- foo3
```

Erzeugt ein:

```
Test:
   • foo
   • foo2
   • foo3
```

Ordered List (Numbered List)

Um eine nummerierte Auflistung in der 1. Spalte zu starten, wird in die **erste Spalte eine Zahl** und in die **zweite Spalte ein Punkt** geschrieben. Die Zahlen haben keine Einfluss auf die Nummerierung (die wird automatisch von Markdown erstellt). Mit 1. Spalte ist das erste Zeichen in einer Zeile gemeint.
In einigen Implementierungen ist vor der ersten Zahl ein Newline (\n) notwendig.
In einigen Implementierungen beginnt die automatische Zählung bei 1, bei anderen ab der ersten Zahl.

```
Test:

2. foo2
3. foo3
3. foo3
0. foo0
```

Erzeugt als Output (Nummerierung beginnt bei 1):

```
Test:
   1. foo2
   2. foo3
   3. foo3
   4. foo0
```

Einige Implementierungen beginnen die Zählung ab der ersten Zahl. In diesem Beispiel ab 2:

```
Test:
   2. foo2
   3. foo3
   4. foo3
   5. foo0
```

Nested List

Eine zweite Ebene wird mit **zwei Leerzeichen und einem Stern** erzeugt (statt 2 Leerzeichen ist auch ein 1 Tab möglich):

```
+ foo
+ foo2
  + foo4
  + foo5
```

Ezeugt ein:

- foo
- foo2
  - ◊ foo3
  - ◊ foo4

Möglicher HTML-Output:

foo foo2
foo4
foo5

Alle weiteren Ebenen werden ebenfalls mit zwei Leerzeichen und einem * und erzeugt:

```
+ foo
+ foo2
  + foo4
  + foo5
    + foo6
    + foo7
```

Erzeugt ein:

- foo
- foo2
  - ◊ foo3
  - ◊ foo4
    - foo
    - foo2

Eine ordered Liste:

```
Test:

2. foo2
2. foo2
3. foo3
   8. foo8
   9. foo9
   9. foo9
```

Die Zählung ab 1 ist von der Implementierung abhängig. Möglicher Output:

```
Test:
   1. foo2
   2. foo2
   3. foo3
   1. foo8
   2. foo9
   3. foo9
```

Block (Monospace)

Es gibt 2 Möglichkeiten für Textblöcke (in Monospace-Font):

\n\t	statt einem Tab sind meist auch 4 Leerzeichen möglich	<code>
`	Text zwischen Backticks (auch über mehrere Zeilen)	<code>

Möglichkeit1: Tab

Ein **Newline** (\n) und ein **Tab** (\t) erzeugen einen Monospace-Textblock. Der folgende Text wird optisch als Block gekennzeichnet. Alternativ sind oft statt einem Tab, auch 4 Leerzeichen möglich.
Ein Zeilenumbruch (\n) beendet den Block (und erzeugt somit auch optisch einen Zeilenumbruch).

Im Block wird markdown nicht interpretiert.

```
Er sprach:

   Fehlt es mir etwa
   an Verrückten, daß ihr diesen **Mann** hergebracht habt?
ende
```

erzeugt als Output:

```
Er sprach:

    Fehlt es mir etwa
    an Verrückten, daß ihr diesen *Mann**
    hergebracht habt?

ende
```

Möglichkeit2: `

Schriftzeichen zwischen Backticks (`) werden als eigener Block dargestellt.

```
Fehlt es mir etwa
an Verrückten, daß ihr `diesen Mann hergebracht` habt?
```

erzeugt als Output:

```
Fehlt es mir etwa an Verrückten, daß ihr    diesen Mann hergebracht
```

Möglicher HTML-Code dazu:

```
<p>Fehlt es mir etwa an Verrückten, daß ihr <code>diesen Mann hergebracht</code> habt?</p>
```

Im Text dürfen nicht **zwei Newlines** (\n\n) oder **zwei Leerzeichen und ein Newline** sein.

Bild

Um ein Bild anzuzeigen, sind 3 bis 4 Anweisungen notwendig:

1. Ein Rufzeichen `!`
2. gefolgt vom optionalen Alternative-Text in eckigen Klammern `[foo]`
3. gefolgt von der url in runden Klammern `(foourl)`
4. optional: direkt nach der url folgt der Title `(foourl "foottle")`

Syntax-Beispiel: `![alttext](/path/to/img.jpg)`

Title

Nach der URL folgt ein Leerzeichen und der Title-Text in Anführungszeichen. Der Title-Text wird meist als Mouse-Over-Text angezeigt.

```
![bin alternativer text](http://www.heise.de/l.gif "bin title text")
```

Erzeugt als Output:

Wenn das Bild nicht geladen werden kann, wird der alternative Text angezeigt.

Uniform Resource Locator (URL)

Für eine anklickbare URL gibt es zwei Möglichkeiten: Entweder wird die URL selbst anklickbar oder die URL-Bezeichnung (und die URL wird ausgeblendet).

Möglichkeit 1: URL anzeigen

URL werden anklickbar/aufrufbar, wenn sie in spitzen Klammern stehen.

Diese Email-Adresse:

```
<mailto:foobar@foo.com>
```

Erzeugt als Output:

foobar@foo.com

Diese Webseite:

```
<http://www.heise.de>
```

Erzeugt als Output:

http://www.heise.de

Möglichkeit 2: Bezeichnung anzeigen

Nur die Bezeichnung ist zu sehen und ist anklickbar (die URL wird ausgeblendet).

Diese Email-Adresse:

```
[meine imehl](mailto:foobar@foo.com)
```

Erzeugt als Output: meine imehl

meine imehl

Dieses Beispiel:

```
[meine Webseite](http://www.heise.de)
```

Erzeugt als Output: meine Webseite

meine Webseite

Title

Ein Leerzeichen und ein Text in Anführungezeichen erzeugt einen Title (Mouseover-Text):

```
[meine Webseite](http://www.heise.de "bin der title")
```

Erzeugt als Output: meine Webseite

meine Webseite

Referenzierung

Für eine Referenzierung über eine ID, sind 2 Teile notwendig.

Teil 1 besteht aus 3 Anweisungen notwendig:

1. Ein Rufzeichen !
2. gefolgt vom optionalen Alternative-Text in eckigen Klammern [foo]
3. gefolgt von der ID in eckingen Klammern [fooid]

Syntax-Beispiel: `![alttext][id]`

Teil 2 besteht aus 3 bis 4 Anweisungen (Title ist optional):

1. ID in eckigen Klammern [fooid]
2. gefolgt von einem Doppelpunkt :
3. gefolgt von der URL foourl
4. gefolgt von einem Leerzeichen und dem Title-Text in Anführungszeichen "footitle"

Syntax-Beispiel: `[id]: /path/to/img.jpg "Title"`

Beispiel:

```
[alternative][binid]

[binid]: http://www.heise.de/i.gif "bin title text"
```

Erzeugt einen Link mit dem Linktext **alternative** und dem Mouseover-Text **bin title text**:

alternative

Der mögliche HTML-Code: `<a href="http://www.heise.de/i.gif" title="bin title text">alternative</a>`

Nur der alternative Text ist zu sehen (nicht das Bild).

Tabelle

Dieser markdown-Code:

```
| a | b | c | d | e |
|---|---|---|---|---|
|   | x |   |   | 2 |
|   |   |   | s | f |
| y |   | d2 |   | j3c |
```

Erzeugt diesen Output:

a	b	c	d	e
	x			2
			s	f
y		d2		j3c

Escape Character (Fluchtsymbol)

Escape Character ist ein Backslash. Der Escape-Character wird auf deutsch Fluchtsymbol genannt.
Siehe auch **ASCII Control Characters** Seite 225.

markdown-Steuerzeichen

Tab \t ein \t (in der ersten Spalte) erzeugt einen Monospace-Block

` Text zwischen zwei Backticks werden als Monospace-Block dargestellt.

* Text zwischen zwei Unterstrichen wird fett oder kursiv dargestellt
bei zwei Sternen wird der Text fett

_ Text zwischen zwei Unterstrichen wird oft kursiv dargestellt
bei zwei Unterstrichen wird der Text fett

* erzeugt eine Liste (wenn es, von Space und Tabs abgesehen, das erste Zeichen in der Zeile ist)
-
+

erzeugt (als erstes Zeichen) eine Überschrift, Überschrifttext folgt danach
oft ist vor dem Überschrifttext ein Leerzeichen notwendig

=== erzeugen (als erste Zeichen) eine Überschrift (Textzeile davor wird als Überschrift interpretiert)

--- erzeugen (als erste Zeichen) eine Überschrift (Textzeile davor wird als Überschrift interpretiert)

*** besteht eine Zeile nur aus drei Sternen, wird eine horizontale Linie gezeichnet.
Space-Character dürfen vor und nach den Sternen in der Zeile vorkommen.

< Text zwischen zwei spitzen Klammern sind eine URL
>

(Text zwischen zwei runden Klammern sind eine URL mit Bezeichner
)

[Text zwischen zwei eckigen Klammern sind der Bezeichner für eine URL
]

! ein Rufzeichen vor dem Bezeichner [ ] deklariert das Objekt der URL als Bild

| Tabellenzelle

HTML

Kapitel 5

HTML

Änderungen zu den Vorgängerversionen

Der default-Zeichensatz in html5 wurde UTF-8, einige meta-Deklarationen wurden unnötig und 8 semantische Tags wurden hinzugefügt.

Doctype

In HTML5 wurde der DOCTYPE deutlich verkürzt:

```
<!DOCTYPE html>
```

Zum Vergleich der DOCTYPE von `XHTML1 Strict`:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

Der DOCTYPE von `HTML4 Strict`:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
```

Im Gegensatz zu XHtml muss der Source Code nicht mehr XML konform sein (Tags müssen nicht zwingend geschlossen werden). Somit fällt auch die XML-Deklaration von XHTML weg:

```
<?xml version="1.0" encoding="UTF-8" ?>
```

Sprache (Language)

Die Textsprache wird mit dem Attribut `lang` im Tag `html` bestimmt.
Als Beispiel englisch:

```
<html lang="en">
```

Die Sprache über das http-Feld `Content-Language`:

```
Content-Language: en
```

Namespace

Eine explizite Namespace-Deklaration ist in html5 nicht mehr notwendig.
In xhtml über das Attribut xmlns:

```
<html xmlns="http://www.w3.org/1999/xhtml">
```

CSS- und Javascript-Type

Die Verwendung vom type-Attribut im Tag <script> und <style> ist nicht mehr notwendig.

In xhtml:

```
<style type="text/css">
<script type="text/javascript">
```

Semantische Strukturblöcke

html5 führt 8 neue semantische Tags ein: <article> <aside> <data> <header> <figure> <nav> <section> <time>

IE vor Version 9 kann unbekannte Elemente nicht per CSS stylen (Workaround: html5shiv).

Zeichensatz

Der HTML5 default-Zeichensatz (siehe W3) ist UTF-8.

Es gibt 3 Möglichkeiten für die Zeichensatz-Deklaration:

1) http-header
2) html-meta-tag
3) Byte-Order-Mark (siehe Seite 235) für UTF Encoding

Laut HTML5 soll ein **ISO-8859-1** (siehe Seite 231) Dokument als **Win1252** (siehe Seite 241) interpretiert werden.
Von der Verwendung von UTF-16 und UTF-32 wird abgeraten, UTF-7 ist explizit nicht erlaubt (siehe W3).

ISO-8859-1 und UTF-8 sind in den ersten 256 Zeichen identisch.

Eine Auflistung aller RFC2978 Character-Sets siehe IANA.

http-header

Im http1.1-Header wird der Zeichensatz über das Feld **Content-Type** bestimmt:

```
Content-Type: text/html; charset=UTF-8
```

Der Header kann zB durch PHP manipuliert werden, oder beim Apache durch diesen Eintrag im .htaccess:

```
AddType 'text/html; charset=UTF-8' html
```

Byte-Order-Mark

Laut HTML5 ist der **Byte-Order-Mark** (BOM) des Dokuments höherwertiger, als die Angabe im http-Header. Das ist hilfreich, wenn der Seitenautor keinen Einfluss auf die Servereinstellungen zur Zeichencodierung hat. Siehe W3. Siehe Byte-Order-Mark Seite 235.

Laut RFC 2718 soll der BOM nicht gesetzt werden, wenn die Zeichencodierung im http-Header UTF-16LE oder UTF-16BE ist. Nur bei UTF-16 (http-Header) soll ein BOM gesetzt werden.

Bei UTF-8 ist im Gegensatz zu den UTF-16-Codierungen der BOM nicht zwingend erforderlich, denn es gibt nur eine mögliche Reihenfolge der Bytes für ein bestimmtes Zeichen. Diese explizite UTF-8 Kennung wird **UTF-8 Signatur** genannt (siehe Seite 236).

meta Tag

Den Zeichensatz mit dem html-Tag meta und dem Attribut charset deklarieren:

```
<meta charset="utf-8">
```

Der Zeichensatz kann pro Dokument nur einmal gesetzt werden.

Zum Vergleich eine Deklaration für ein xml- oder XHtml-File:

```
<?xml version="1.0" encoding="utf-8"?>
```

Semantische Tags (aka *Innere Werte*)

 Tags sollten nur verwendet werden, wenn sich kein passender symantischer Tag finden lässt:

```
<div class="datum">  22.10.2066 </div>   <time>22.10.2066</time>        html5
<div class="main_content"> foo  </div>   <article> foo </article>      html5
<div class="kopfzeile"> foo  </div>      <header> foo </header>         html5
<div class="fusszeile"> foo  </div>      <footer> foo </header>         html5
<div class="navigation"> foo </div>      <nav> foo </nav>               html5
<div class="content"> foo    </div>      <section> foo </section>       html5
<div class="schaubild"> foo  </div>      <figure> foo </figure>         html5
<div class="blog_kommentar"> foo </div> <aside> foo </aside>           html5
```

Diese Tags haben eine semantische Bedeutung und bewirken eine optische Textformatierung:

```
<div class="text_klein"> foo </div>     <small> foo </small>                klein
<div class="kontaktadresse"> foo</div>  <address> foo </address>            kursiv
<div class="fachbegriff"> foo   </div>  <i> foo </i>                        kursiv
<div class="betonung"> foo   </div>     <em> foo </em>                      kursiv
<div class="wichtig"> foo    </div>     <strong> foo </strong>              fett
<div class="produkt_fett"> foo  </div>  <b> foo </b>                        fett
<div class="highlight"> foo  </div>     <mark> foo </mark>                  highlight
```

Der Tag ist ein generischer Container ohne semantische Bedeutung. Ein **General Flow Content Element**.

Eine Struktur-Information lässt nur über selbstsprechende Klassenbezeichnungen ableiten:

```
<div class="faq_head">
```

<article> <section>

Durch den Einsatz von den neuen html5-Tags <section> und <article> erkennt man inhaltlich zusammenhängende Blöcke. Für die Gliederung innerhalb eines article-Blocks sollte <section> verwendet (keine <article> Verschachtelung):

```
<article>
       <section> </section>
       <section> </section>
       <article> </article>
</article>
```

<header>

Das header Element leitet einen neuen semantischen Bereich ein und stellt meist eine Kopfzeile dar. Ein article-Block könnte ein Kopfzeile aus 2 Teilen haben: eine Überschrift und ein Datum. Die Kopfzeile wird mit dem neuen html5-Tag <header> Tag gekennzeichnet. Der Überschrift-Text wird mit <h1> und das Datum mit <h2> gekennzeichnet. Innerhalb von <h2> wurde das Datum noch zusätzlich mit dem semantischen Tag <time> versehen.

Der WHATWG-Tag <hgroup> wurde von W3 nicht in html5 übernommen (eine Gruppierung der Überschriften ist nicht notwendig).

```
<article>
    <header>
       <h1>Überschrift</h1> <h2><time>Datum</time></h2>
    </header>
</section>
```

Innerhalb von einem header-Element sollte kein footer- oder ein weiteres header-Element stehen.

<nav> <menu>

Für größere Link-Blöcke (Navigation) sollte der neue html5-Tag `<nav>` verwendent werden (für Ebook-Reader gedacht). Für Menü-Blöcke der umdefinierte Tag `menu`.

```
<nav>
  <h1>Hauptnavigation</h1>
  <ul> <li><a>über mich</a></li> <li><a>kontakt</a></li> </ul>
</nav>
```

<aside>

Für Inhalt, der nur teilweise zum Hauptinhalt der Seite gehört, sollte der neue html5-Tag `<aside>` verwendet werden (zB Kommentare zu einem Blog-Artikel).

<figure> <figcaption>

Für Inhalt, die den Hauptinhalt näher beschreiben und ergänzen, sollte der neue html5-Tag **<figure>** verwendet werden (Source Code, Bild, Video). Siehe auch Tabellen-Element `caption`.

```
<figure>
<img src="sonnenblumen.jpg">
<figcaption>van Goghs Sonnenblumen</figcaption>
</figure>
```

<time>

Für einen Text, der ein Datum oder eine Uhrzeit darstellt, sollte der neue html5-Tag `<time>` verwendet. Mit dem Attribut `datetime` kann ein Zeitstempel bestimmt werden:

```
<time datetime="2066-04-19 19:00">19. April 2066</time>
<time datetime="2066-04-19">mein Geburtstag</time>
<time>2066-04-19</time>
```

<data>

Für Text, der Zusatzinformationen für Parser bietet. Der Inhalt befindet sich im value Attribut und ist somit für Menschen nicht sichtbar. Beispiel:

```
<data value="produktid_859288">Rätsel Lexikon</data>
```

Generische Container-Tags

Generische Tags sind keinem bestimmten Zweck zugeordnet. Sie werden meist für das Gruppieren von Text verwendet (um sie mit CSS-Klassen zu formatieren).

Der `<span>` Tag ist ein **generic inline container** (Inline-Element). Er wird für Text verwendet, der per CSS (id oder class-Attribut) gestaltet wird. Ähnlich dem -Tag (dieser ist aber ein Block-Level-Element).

Der `<div>` Tag ist ein **Document Division Element** (Block-Level-Element). Er wird für Text verwendet, der per CSS (id oder class-Attribut) gestaltet wird. Ähnlich dem -Tag (dieser ist aber ein Inline-Element).

Textformatierende semantische Tags

Diese Tags sind textformatierend und haben eine semantische Bedeutung:

<abbr>

Der `<abbr>` Tag stellt Text oft unterstrichen (dottet) dar. Er steht für eine Abkürzung und bietet optional eine Beschreibung über das Attribut `title`.

Beispiel: `<abbr title="United States of America">USA</abbr>`

Output: USA

<address>

Der `<address>` Tag stellt Text kursiv dar. Er steht für Informationen über den Verfasser oder für eine Adresse.

Beispiel: `Fehlt es mir <address>etwa</address> an Verrückten`

Output:

Fehlt es mir
etwa
an Verrückten

<b>

Stellt Text fett dar. Er steht für einen optisch anders darzustellenden Text (ohne eine besondere Wichtigkeit zu implizieren).

Beispiel: `Fehlt es mir <b>etwa</b> an Verrückten`

Output: Fehlt es mir **etwa** an Verrückten

<bdo>

Der **Bi**directional **O**verride Tag bestimmet die Schreibrichtung der Zeichen.

Das Attribut `dir` hat als mögliche Werte:

ltr	left-to-right
rtl	right-to left
auto	Browsereinstellung

Beispiel:

```
<p><bdo dir="ltr">von links nach rechts</bdo></p>
<p><bdo dir="rtl">von rechts nach links</bdo></p>
```

Ergibt als Output:

von links nach rechts
sknil hcan sthcer nov

<blockquote>

Der `<blockquote>` Tag erzeugt einen Zeilenumbruch und rückt den Text nach rechts ein. Für längere Zitate.

Beispiel: `Fehlt es mir, <blockquote>etwa an Verrückten</blockquote>`

Output: Fehlt es mir,

> etwa an Verrückten

<cite>

Der <cite> Tag stellt Text meist kursiv dar. Für den Namen eines (Kunst-)werkes. Für ein längeres Zitat sollte man <blockquote> nehmen.

Beispiel: `Fehlt es mir <cite>etwa</cite> an Verrückten`

Output: Fehlt es mir *etwa* an Verrückten

<code>

Der <code> Tag stellt Text in Monospaced-Font dar. Er steht für einen Source Code (Programm-Code). Tabs (\t) und Zeilenumbrüche (\n) werden nur als ein Leerzeichen angezeigt. Sollen diese auch optisch umgesetzt werden, muss ein <pre> vorangestellt werden.

Siehe auch <samp> Seite 133, <pre> Seite 133, <kbd> Seite 133

Beispiel:

```
<code>
zeitungsente
 stopfgans
 wildhuhn
zaungast<br><i>brathenne</i><br>
</code>
```

Erzeugt als Output:

```
zeitungsente stopfgans wildhuhn zaungast
brathenne
```

<del>

Der <del> Tag stellt Text durchgestrichen dar. Ab html5 steht dieser für gelöschten Text. Für nicht mehr relevante oder nicht mehr gültige Daten, sollte <s> verwendet werden.

Beispiel: `Fehlt es mir, <del>etwa</del> an Verrückten`

Output: Fehlt es mir, ~~etwa~~ an Verrückten

<dfn>

Mit <dfn> kann ein Definitionsbegriff gekennzeichnet werden. Der Text wird kursiv angezeigt.

Als Beispiel HTML:

```
<dfn>HTML</dfn>
```

Output:

HTML

Mit dem Attribut `value` kann eine Definitionsbeschreibung hinzugefügt werden:

```
<dfn title="HyperText Markup Language">HTML</dfn>
```

Bei Kombination mit dem <abbr> Tag wandert das title-Attribut zum <abbr> Tag:

```
<dfn><abbr title="HyperText Markup Language">HTML</abbr></dfn>
```


Der Tag stellt Text meist kursiv dar. Für einen betonende Stelle in einer Rede (besondere Betonung).

Beispiel: `Fehlt es mir <em>etwa</em> an Verrückten`

Output: Fehlt es mir *etwa* an Verrückten

<i>

Der <i> Tag stellt Text kursiv dar. Ab html5 steht er für einen Textabschnitt, der vom **übrigen Inhalt** abgesetzt und **üblicherweise kursiv** dargestellt wird, ohne für eine spezielle Betonung oder Wichtigkeit zu stehen.

Beispiel: `Fehlt es mir <i>etwa</i> an Verrückten`

Output: Fehlt es mir *etwa* an Verrückten

<kbd>

Der <kbd> Tag stellt Text in Monospaced-Font dar. Er steht für einen Text, der vom User per Keyboard eingegeben werden muss. Tabs (\t) und Zeilenumbrüche (\n) werden optisch nicht umgesetzt. Html-Tags werden interpretiert.

Siehe auch <samp> Seite 133, <pre> Seite 133, <code> Seite 133

Beispiel:

```
Tippe ein:
<kbd>
zeitungsente
 stopfgans
 wildhuhn
zaungast<br><i>brathenne</i><br>
</kbd>
```

Erzeugt als Output:

Tippe ein: `zeitungsente stopfgans wildhuhn zaungast`
brathenne

Stellt Text fett dar. Er steht für einen besonders wichtigen Textabschnitt.

Beispiel: `Fehlt es mir <strong>etwa</strong> an Verrückten`

Output: Fehlt es mir **etwa** an Verrückten

<mark>

Mark stellt Text highlighted dar. Er steht für einen markierten Textabschnitt (ohne eine besondere Wichtigkeit zu implizieren).

Beispiel: `Fehlt es mir <mark>etwa</mark> an Verrückten`

Output: Fehlt es mir etwa an Verrückten

<pre>

Der <pre> Tag zeigt formatierten Plaintext (meist Source Code) in Monospaced-Font an. Im Gegensatz zu <code> und <samp> werden Zeilenumbrüche (\n) und Tabs (\t) optisch umgesetzt. Html-Tags werden interpretiert.
Siehe auch <samp> Seite 133, <code> Seite 133, <kbd> Seite 133

```
<pre>
zeitungsente
 stopfgans
 wildhuhn
zaungast<br><i>brathenne</i><br>
</pre>
```

Ergibt als Output:

```
zeitungsente
 stopfgans
 wildhuhn
zaungast
brathenne
```

<q>

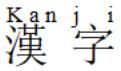
Der <q> Tag stellt Text in Anführungszeichen. Für ein kurzes Zitat innerhalb eines Text.
Beispiel: `Fehlt es mir <q>etwa</q> an Verrückten`
Output: Fehlt es mir "etwa" an Verrückten

<ruby> <rp> <rt>

Zeigt die Aussprache von ostasiatischen Zeichen an.
Beispiel:

```
<ruby>
  漢 <rp>(</rp><rt>Kan</rt><rp>)</rp>
  字 <rp>(</rp><rt>ji</rt><rp>)</rp>
</ruby>
```

Output:



<s>

Der <s> Tag stellt Text durchgestrichen dar. Ab html5 steht dieser Text für nicht mehr gültige oder nicht mehr relevante Daten dar. Für gelöschten Text sollte <del> verwendet werden.
Beispiel: `Fehlt es mir, <s>etwa</s> an Verrückten`
Output: Fehlt es mir, ~~etwa~~ an Verrückten

<samp>

Mit <samp> wird die Ausgabe von einem Computer (zb einer Shell) gekennzeichnet. Der Text wird in Monospaced-Font dargestellt. Tabs (\t) und Zeilenumbrüche (\n) werden optisch nicht umgesetzt. Html-Tags werden interpretiert.
Siehe auch <pre> Seite 133, <code> Seite 133, <kbd> Seite 133

Beispiel:

```
<samp>
zeitungsente
 stopfgans
 wildhuhn
zaungast<br><i>brathenne</i><br>
</samp>
```

Erzeugt als Output:

```
zeitungsente stopfgans wildhuhn zaungast
brathenne
```

<small>

Der <small> Tag stellt Text kleiner dar. Ab html5 steht dieser Text für das **Kleingedruckte** eines Dokuments (Ausschlussklauseln, Copyright-Hinweise).
Beispiel: `Fehlt es mir, <small>etwa</small> an Verrückten`
Output: Fehlt es mir, <small>etwa</small> an Verrückten

<var>

Der <va> Tag stellt Text kursiv dar. Er steht für eine Variable oder mathematischen Ausdruck.
Beispiel: `<var>x</var> = <var>y</var> + 2`
Output: $x = y + 2$

Typographische Tags

<sub>

Der <sub> Tag (Superscript) stellt Text niedergestellt dar. Er sollte in mathematischen Ausdrücken oder chemischen Formeln (H_2O) verwendet werden. In anderen Fällen sollte per CSS (vertical-align property) niedergestellt werden.

Beispiel: `H<sub>2</sub>O`

Ergibt als Output:

H_2O

<sup>

Der <sup> Tag (Superscript) stellt Text hochgestellt dar. Er sollte bei Potenzahlen oder französischen Abkürzungen (Mlle, Mme or Cie) verwendet werden. In anderen Fällen sollte per CSS (vertical-align property) hochgestellt werden.

Beispiel: `C<sup>ie</sup>`

Ergibt als Output:

Cie

<wbr>

Der `<wbr>` Tag kennzeichnet eine passende Stelle für einen eventuell notwendigen Zeilenumbruch. Im Gegensatz zu <br> wird kein Zeilenumbruch erzwungen. Entspricht dem Html-Entity `` (siehe auch ASCII Seite 225)

Beispiel Donaudampfschifffahrtsgesellschaft: `<p>Donaudampfschi fffahrtsgesellschaft</p>`

Der `<br>` Tag erzwingt einen Zeilenumbruch..

Beispiel: `Fehlt es mir etwa <br>an Verrückten`

Ergibt als Output:

> Fehlt es mir etwa
> an Verrückten

Geänderter und Gelöschter Text

<ins>

Markiert einen zum Dokument *hinzugefügten* Teil.

<del>

Markiert einen aus dem Dokument *entfernten* Teil.

Eingebetteter Inhalt

Markiert ein Bild.

<iframe>

Eine andere Html-Seite einbetten.

<embed>

Einbindungspunkt für externe Ressource, die meist keine Html-Seiten sind (zb Plugin).

<object> <param>

Mit dem Element `object` wird eine externe Datei eingebettet.

Attribute:

`data`	URL
`form`	ID des form Elementes
`height`	Höhe px
`width`	Breite px
`name`	Name
`type`	MIME-Typ
`usemap`	(Hash) Identifizierungsname des map-Elementes

Beispiel: `<object data="move.swf" type="application/x-shockwave-flash"></object>`

Mit dem Element `param` werden Parameter für ein `object` Element spezifiziert.

Attribute:

`name`	Names des Parameters
`value`	Wert des Parameters

Beispiel:

```
<object data="move.swf" type="application/x-shockwave-flash">
   <param name="foo" value="bar">
 </object>
```

<video> <source> <track>

Das Element `video` bettet ein Video ein. `track` bettet eine Untertitel-Datei ein. `source` gibt die URL an.

Attribute von `video`:

`src`	URL Alternative kann das `source` Element verwendet werden.
`poster`	URL vom Vorschaubild. default ist der 1. Frame vom Video.
`width`	Breite px
`heigth`	Höhe px
`muted`	Attributwert ist Bool (mit oder ohne Ton)
`autoplay`	Attributwert ist Bool
`loop`	Attributwert ist Bool
`controls`	Attributwert ist Bool (Kontrollelemente anzeigen)

Attribute von track:

`kind`	Art des Zusatz (zB subtitles)
`srclang`	Sprache (zb de, en)
`lang`	Bezeichnung

Beispiel:

```
<video src="videofile.ogg" controls autoplay poster="vorschau.jpg">
  Sorry, your browser doesn't support embedded videos.
 <track kind="subtitles" src="foo.en.vtt" srclang="en" label="English">
</video>
```

<audio> <source> <track>

Das Element `audio` bettet eine Audiodatei ein. `track` bettet eine Untertitel-Datei ein. `source` gibt die URL an.

Attribute von `audio`:

`src`	URL (alternativ kann das `source` Element verwendet werden)
`controls`	Attributwert ist Bool (Kontrollelemente anzeigen)
`width`	Breite px
`heigth`	Höhe px
`muted`	Attributwert ist Bool (mit oder ohne Ton)
`autoplay`	Attributwert ist Bool
`loop`	Attributwert ist Bool

Attribute von track:

`kind`	Art des Zusatz (zB subtitles)
`srclang`	Sprache (zb de, en)
`lang`	Bezeichnung

Beispiel:

```
<audio controls>
    Your browser does not support the audio element.
    <source src="foo.wav" type="audio/wav">
 <track kind="captions" src="foo.en.vtt" srclang="en" label="English">
</audio>
```

<canvas>

Steht für einen Bitmap-Bereich, der von Skripts verwendet werden kann.

Attribute:

`width`	Breite
`heigth`	Höhe
`id`	ID

Beispiel:

```
<canvas id="canvas" width="260" height="200">
  Dein Browser unterstützt das canvas Element nicht.
</canvas>
```

<svg>

Bettet **Scalable Vector Graphic** ein. Die 2D Vektorgrafik wird in einem XML-Format beschrieben.

Siehe SVG Seite 244.

Beispiel:

```
<svg>
   <path d="M 150,100  A 70 50  0 0 1  250,150" stroke="lime" fill="none" stroke-width="2"/>
</svg>
```

Erzeugt als Output:

<map> <area>

Definiert in Verbindung mit dem <area>-Element eine Image Map.

Interaktive Elemente

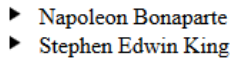
<details> <summary>

Das Element `details` markiert das anklickbare Kontrollelement ▶, mit dem der Benutzer zusätzliche Text-Informationen erhalten kann.
Das Element `summary` kennzeichnet eine Zusammenfassung für ein bestimmtes <details> Element.

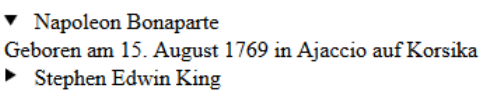
Beispiel:

```
<details>
    <summary>
    Napoleon Bonaparte
    </summary>
    Geboren am 15. August 1769 in Ajaccio auf Korsika
</details>
<details>
    <summary>
    Stephen Edwin King
    </summary>
    Geboren am 21. September 1947 in Portland
</details>
```

Erzeugt als Output:



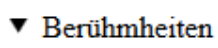
Nach dem Klick auf das 1. Kontrollelement ▶ klappt die Zusatzinformation auf.
Zum Zuklappen erscheint das Element ▼:



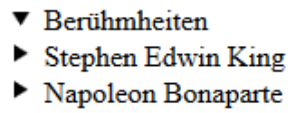
Die Struktur lässt sich auch verschachteln:

```
<details>
    <summary>Berühmheiten</summary>
   <details>
      <summary>Stephen Edwin King</summary>
      Geboren am 21. September 1947 in Portland
   </details>

   <details>
      <summary>Napoleon Bonaparte</summary>
      Geboren am 15. August 1769 in Ajaccio auf Korsika
   </details>
</details>
```

Ergibt als Output:



Nach dem Anklicken:



Link

Ein Link ist eine Verbindung (Beziehung) zu anderen Dateien. Für anklickbare Links wird das Element `a` verwendet, ansonsten das Element `link`.

<link>

Das Element link stellt eine Beziehung zu anderen Dateien her. Das Element darf nur im `head` verwendet werden.

Attribute:

`href`	Ziel einer Beziehung (href = HyperReference) ist ein URL (Uniform Resource Locator). Siehe wikipedia
`media`	Ausgabemedium (`print, screen, handheld`). Default ist `all`
`rel`	Typ der Beziehung (rel = relation)
`type`	MIME-Type der Ressource. Siehe Wikipedia und selfhtml-wiki

rel

Das Attribut `rel` definiert die Art der Beziehung. Mögliche Attributwerte:

`alternate`	alternative Seite (alternatives Dokument)
`appendix`	Anhang
`author`	Bezug zum Autor
`canonical`	primäre Ressource (falls das gegenwärtige Dokument eine Kopie ist). Suchmaschinen zeigen im Suchergebnis meist die Primärseite an.
`chapter`	Kapitel
`contents`	Inhaltsverzeichnis
`copyright`	Copyright
`bookmark`	Lesezeichen
`icon`	Browser-Icon. Siehe Seite 154. `<link rel="icon" sizes="192x192" href="favicon.png">`
`first`	erste Datei
`glossary`	Glossar
`help`	Hilfe
`index`	Stichwortverzeichnis
`last`	letzte Datei
`next`	nächste Seite. Je nach Einstellung laden Browser diese Seite im Voraus (prefetch).
`prev`	vorherige Seite
`search`	Suchfunktion
`section`	Abschnitt
`start`	Startseite
`shortcut icon`	Browser-Icon. Siehe Seite 154. `<link rel="shortcut icon" type="image/x-icon" href="/favicon.ico">`
`stylesheet`	CSS. Siehe auch Prioritätskaskade Seite 212. `<link rel="stylesheet" href="foo.css" type="text/css">`
`subsection`	Unterabschnitt
`up`	übergeordnete Kapitelseite

<a>

Das Element a hat 2 Einsatzmöglichkeiten:

1. einen Verweis (Link) kennzeichnen (auf beliebige Dateien)
2. einen Anker setzen

Link-Attribute:

href hyper reference. Attributwert ist ein URL (Uniform Ressource Locator)

Webseite:	`<a href="http://www.foo.com">foo</a>`
Email-Address:	`<a href="mailto:foo@foobar.com">foo@foobar.com</a>`
Telefonnummer:	`<a href="tel:+43123456789">+43 1234 567 89</a>`
Skype:	`<a href="skype:foo">foo</a>`
Email mit Subject:	`<a href="mailto:foo@foo.de?subject=frage">foo@foo.de</a>`
Email mit CC	`<a href="mailto:f1@foo.de?cc=f2@foo.de">f1@foo.de</a`
Auf Anker foo	`<a href="http://www.foo.com/index.html#foo">index</a`

hreflang Sprach-Code (siehe selfhtml-wiki).

de für deutsch, en für englisch

Beispiel:
```
<a href="foo.html" hreflang="de">Seite a</a>
```

tabindex die Reihenfolge bestimmen, in der die Links beim Drücken der Tab-Taste aufgerufen wird.
Die Sprung-Reihenfolge beginnt beim kleinsten Attributwert (Integer).
Aufgerufen wird der jeweilige Link durch Drücken der Enter-Taste.

Attributwert -1 nimmt den jeweiligen Link aus der Reihenfolge heraus (nicht fokusierbar)
Attributwert 0 ist der default.
Die Attributwerte müssen für das gesamte Dokument unique sein.

Beispiel:
```
<a href="#" tabindex="3">Seite a</a>
<a href="#" tabindex="2">Seite b</a>
<a href="#" tabindex="1">Seite c</a>
```

target Name des Browserfensters. Wenn das Fenster mit diesem Namen nicht (mehr) existiert, wird ein neues Fenster erstellt. Bei einem Frame, wird das jeweilige Framefenster fokusiert. Vordefinierte Konstanten:

_self	gleiches Fenster (default)
_blank	neues Fenster
_parent	Elternfenster
_top	oberstes Fenster

Beispiel:
```
<a href="foo.html" target="foobar">Seite foo</a>
```

title Tooltip-Text (erscheint beim Drüberfahren mit der Maus)

Beispiel:
```
<a href="foo.html" title="bin ein Hinweis">Seite foo</a>
```

download der Browser soll das Dokument nicht anzeigen, sondern als Download anbieten.

Beispiel:
```
<a href="foo.pdf" download>foo.pdf</a>
```

Beispiel:
```
<a href="foo.pdf" download="foo.pdf">foo.pdf</a>
```

accesskey bestimmt eine Tastenkombination, um den Link zu fokusieren (enter ruft den Link auf)
Beispiel Alt+f: `<a href="foo.pdf" accesskey="f">foo.pdf</a>`
Die Tastenkombination Alt+Shift+f ruft den Link sofort auf.

rel Art der Beziehung (Relation). Siehe Seite 139.

media Ausgabemedium (`print, screen, handheld`). Default ist `all`

Anker-Attribut:

id Mit dem Attribut `id` (String) wird ein Anker (Sprungmarke) definiert.

Die Verwendung vom Attribut `name` ist deprecated.

Der Ankername sollte kein Whitespace und Sonderzeichen enthalten.
Im Link wird dem Anker eine Raute vorangestellt.

Beispiel Definition: `<a id="foo" />`
Beispiel-Link: `<a href="index.html#foo">index</a>`

Textgruppe

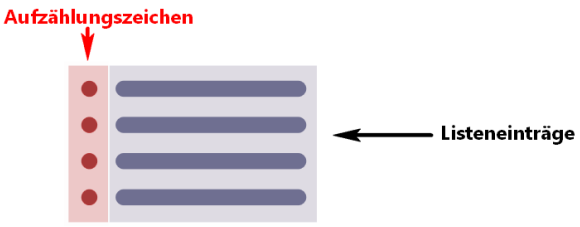
<p>
Markiert einen Absatz.

<hr>
Erzeugt eine horizontale Linie (um Texte thematisch zu trennen).

Das Block-Level-Element `div` bezeichnet ein allgemeines Container-Element ohne spezielle semantische Bedeutung. Wird oft zusammen mit class- oder id-Attributen verwendet, um es per Script oder CSS auswählen zu können.

Das Inline-Level-Element `span` markiert einen allgemeinen Textabschnitt. Wird oft zusammen mit class- oder id-Attributen verwendet, um es per Script oder CSS auswählen zu können.

Liste



`ol` und `ul` Listen bestehen aus 2 Elementen: Das `ul`/`ol` Element bestimmt die Art der Liste, das `li` Element markiert den Listeneintrag.

Bei einem `ul` ist das Aufzählungszeichen ein graphisches Symbol.

Bei einem `ol` (Ordered List) ist das Aufzählungszeichen eine (Alpha-)nummerische Sortierung.

Siehe HTML-Liste Seite 142. Siehe auch CSS-Counter Seite 219.

Der Tag <ul> kennzeichnet den Beginn einer **Unordered List**. steht für den jeweiligen Listeneintrag.

Siehe CSS-Formatierung siehe Seite 217.

```
<ul>
  <li>1</li>
  <li>2</li>
</ul>
```

Ergibt als Output:

- 1
- 2
-

Der Tag `<ol>` kennzeichnet den Beginn einer **Ordered List.** `<li>` steht für den jeweiligen Listeneintrag.

Siehe CSS-Formatierung siehe Seite 217.

Attribute:

`type`	die Art der (alpha)nummerischen Aufzählung. Startwert ist 1. Mit html3.2 eingeführt. Mit html4 deprecated. Mit html5 deprecated-Status aufgehoben. Mögliche Werte: `1 A a I i`
`start`	Startwert
`reversed`	Countdown (Startwert ist der höchste Wert). Mit html5 eingeführt.

Beispiel:

```
<ol type="I" start="8" reversed>
<li>8</li>
<li>7</li>
<li>6</li>
<li>5</li>
</ol>
```

Ergibt als Output:

```
VIII 8
 VII 7
  VI 6
   V 5
```

Definitionsliste

Mit <dt> wird ein zu erklärender Begriff gekennzeichnet. Mit <dd> die Begriffserklärung. Mit <dl> wird die Begriffsliste als Block gekennzeichnet. Als Beispiel die Begriffserklärung zu Zeitungsente:

```
<dl>
    <dt>Zeitungsente</dt>
  <dd>eine Falschmeldung in der Presse</dd>
</dl>
```

Erzeugt als Output:

Zeitungsente

eine Falschmeldung in der Presse

Ein Begriff kann mehrere Bedeutungen haben:

```
<dl>
    <dt>Ente</dt>
    <dd>Fluss in der Toskana</dd>
<dd>ein Tier aus der Familie der Entenvögel</dd>
</dl>
```

Erzeugt als Output:

Ente

Fluss in der Toskana
ein Tier aus der Familie der Entenvögel

Mehrere Begriffe mit einer Begriffsbeschreibung:

```
<dl>
    <dt>Mozilla Firefox</dt>
    <dt>Firefox</dt>
    <dd>Browser</dd>
</dl>
```

Erzeugt als Output:

Mozilla Firefox
Firefox

Browser

Obsolete HTML-Tags

Diese Tags sind deprecated, obsolet oder wurden nicht von W3C übernommen. Sie sollten nicht verwendet werden.

Tag	Beschreibung	html
<ACRONYM>	Zur semantischen Kennzeichnung eines Akronyms (aussprechbare Abkürzung wie zB NATO).	4
<APPLET>	Zum Inkludieren eines Javas-Applets. Ersatz: <object>, <embed>	3.2, 4
<BASEFONT>	Zur Definition von default-Schriftattributen (size, type, color). CSS-Ersatz: font-size, font-family, color	3.2, 4
<BGSOUND>	Automatisch startende Hintergrundmusik auf einer Webseite. Proprietärer Microsoft-Tag.	
<BIG>	Vergrößert die Schriftgröße. Wurde von HTML3.2 eingeführt. CSS-Ersatz: font-size: large;	3.2., 4
<BLINK>	Blinkender Text. Proprietärer Netscape-Tag.	
<CENTER>	Ein Block-level Element zum Zentrieren von Inline Elementen und von anderen Block-level-Elementen. CSS-Ersatz: text-align: center	
<COMMAND>	IE9 Tag. Markiert einen interaktiven Bereich (zB Button). Proprietärer Microsoft-Tag.	
<CONTENT>	Ist nur innerhalb von Shadow DOM als Insertion Point zu finden.	
<DIR>	Zur Kennzeichnung eines Directorys (Verzeichnis).	
<FONT>	Bestimmt die Schriftart (type), Schriftgröße (size) und die Schriftfarbe (color).CSS: font-size, font-family, color	3.2,, 4
<FRAME>	Zur Definition eines bestimmten Bereiches, in der eine andere HTML-Seite eingebettet wird (Definition eines Framefensters innerhalb eines Framesets). CSS-Ersatz: position: fixed	4
<FRAMESET>	Definition eines Framesets (der Tag enthält <FRAME>-Tags). Das Browserfenster wird in mehrere, voneinander unabhängige Bereiche (Framefenster) aufgeteilt. In jedem Framefenster wird eine eigene html-Seite geladen.	4
<HGROUP>	Wurde von WHATWG eingeführt. Zur Gruppierung von <h*1,2,3,4,5,6*> Tags.	
<ISINDEX>	Erzeugt ein einzeiliges Eingabefeld für eine serverseitige-Suche (Vorgänger vom <FORM>). Ersatz: <input type="search">	2, 3.2, 4
<LAYER>	Absolut oder relativ positionierbare, aus dem Standard-Textfluss herausgelöste Block-Bereiche. Netscape-Tag. <ilayer> für absolut oder relativ positionierbare, aus dem Standard-Textfluss herausgelöste Inline-Bereiche.	
<LISTING>	Der Text wird ohne dem Interpretieren von Tags (unverändert) in Monospaced-Font angezeigt. Der Browser soll mindestens 132 Zeichen pro Zeile anzeigen, ohne umzubrechen. Siehe auch <LISTING> und <PLAINTEXT>	2, 3.2
<MARQUEE>	Einzeilige Laufschrift. Ersatz: CSS-3-Eigenschaften overflow-style,marquee-direction, marquee-play-count, marquee-speed, marquee-style	
<MULTICOL>	Nur Netscape. Mehrspaltiger Textfluss, der Inhalt wird gleichmäßig auf mehrere Spalten verteilt. CSS-Ersatz: CSS3: overflow-style,marquee-direction, marquee-play-count, marquee-speed, marquee-style	
<NOEMBED>	Ein Fallback-Tag für Browser, die den <EMBED> Tag nicht unterstützen.	
<NOFRAME>	Alternativinhalt für Browser, die Frames nicht darstellen kann.	4
<PLAINTEXT>	Der Text wird (ohne dem Interpretieren von Tags) 1:1 in Monospaced-Font angezeigt. Siehe auch <LISTING> und <XMP>	2, 3.2
<SMALL>	Schriftgröße verkleinern. CSS-Ersatz: font-size: small	3.2, 4
<SPACER>	Von Netscape eingeführt. Soll wie transparente gif-Bilder einen Whitespace erzeugen.	
<STRIKE>	Durchgestrichener Text. CSS-Ersatz: text-decoration: line-through	2, 3.2
<TT>	Typewriter Text. Der Text wird in Monospaced-Font dargestellt. CSS-Ersatz: font-family: monospace	2, 3.2, 4
<XMP>	HTML Example Element (eXaMPle). Der Text wird ohne dem Interpretieren von Tags (unverändert) in Monospaced-Font angezeigt. Der Browser soll mindestens 80 Zeichen pro Zeile anzeigen, ohne umzubrechen. Siehe <listing>	2, 3.2

Umdefinierte HTML-Tags

Diese Tags waren deprecated und wurden von Html5 als neuer Tag umdefiniert.

Tag	Beschreibung	Version
<MENU>	Wurde von HTML2 eingeführt, von HTML4 auf deprecated gesetzt und von HTML5.1 als neuer Tag umdefiniert Wie bei wird eine unordered list angezeigt. Die einzelnen Items werden mit <menuitem> markiert und sollen eine Aktion auslösen (zb Buttons). <nav> sollte bei Navigationslinks verwendet werden.	2, 3.2, 4, 5.1
<S>	Der Text wird durchgestrichen dargestellt. Wurde von HTML3.2 eingeführt, von HTML4 auf deprecated gesetzt und von HTML5 als neuer Tag umdefiniert. Steht ab Html5 für Text, der nicht mehr aktuell oder gültig ist. Um gelöschten Text zu markieren, sollte <del> verwendet werden.	3.2, 4, 5
<U>	Stellt Text unterstrichen dar. Wurde von HTML3.2 eingeführt, von HTML4 auf deprecated gesetzt und von HTML5 als neuer Tag umdefiniert Steht ab Html5 für anders darzustellenden Text (zB um falsch geschriebene Wörter zu markieren).	

Obsolete Attribute

Diese Attribute sind deprecated, obsolet oder wurden nicht von W3C übernommen.

Attribut	Beschreibung	Version
abbr	markiert einen Hinweistext in <td> Elementen	4
axis	Liste von Kategorien, zu der eine Tabellenzelle gehört.	4
align	Ausrichtung von Text. Attributwerte: left, right, center, justify. CSS-Ersatz: text-align	3.2, 4
alink	Dokumentweite Farbe für aktive Verweise im <body>-Tag.CSS-Ersatz: a:hover, a:active	3.2, 4
archive	Liste mit Archivdateien für ein Java-Applet	3.2, 4
background	Hintergrundbild. CSS-Ersatz: background-color	3.2, 4
bgcolor	Hintergrundfarbe. CSS-Ersatz: background-color	3.2, 4
border	Rahmendicke bei Tabellen und Bildern. CSS-Ersatz: border	
cellpadding	Innenabstand von Zellen in einer Tabelle im <table>-Tag. CSS-Ersatz: padding	4
cellspacing	Abstand zwischen Tabzellen in einer Tabelle im <table>-Tag. CSS-Ersatz: border-spacing	4
char	Bestimmt in align="char" das Ausrichtungszeichen in Tabellenzellen	4
charoff	Bestimmt die Position des Ausrichtungszeichen an einem Dezimalzeichen in Tabellenzellen	4
charset	Bestimmt den Zeichensatz (RFC2978 Character-Sets siehe IANA) eines Link im <a>-Tag.	4
classid	Eindeutige Bezeichnung (ID) eines Objekts im <object>-Tag	4
compact	Bestimmt bei den Elementen <dir>,<dl> und <pre> eine schmälere Schrift. CSS:-Ersatz: letter-spacing	3.2, 4
clear	Aufhebung eines Elementeinflusses. Anwendung im -Tag. CSS-Ersatz: clear	3.2, 4
code	Klassendatei eines Java-Applets.	3.2, 4
codebase	Basis-URI für ein Java-Applet	3.2, 4
codetype	MimeType des Quelltexts der Datenquelle eines object-Elements	4
color	Schriftfarbe in den Elementen <basefont> und <font>.	3.2, 4
declare	das Objekt eines object-Elements soll geladen werden, aber nicht initialisiert werden	4

event	Führt den Code eines script-Elements aus, wenn ein bestimmtes Ereignis eintritt.	4
for	Im Zusammenhang mit dem script-Element: Script-Code für ein Element mit einem anzugebenden id-Namen.	4
face	Schriftart in den Elementen basefont oder font.	3.2, 4
frame	Bei Tabellenrahmen bestimmt dieses Attribut, welche Aussenramen angezeigt werden sollen.	4
frameborder	Rahmendicke von Framefenstern.	4
hspace	Horizontale Abstände (in Pixel) links und rechts von Elementen, die von anderen Elementen umflossen werden.	3.2, 4
link	Dokumentweite Farbe im <body>-Tag für Verweise zu noch nicht besuchten Verweisen.	3.2, 4
longdesc	URL-Adresse einer Quelle mit einer Langbeschreibung für Grafiken, Framefenster und Objekte.	4
marginheight	Anzahl der Pixel für den Abstand des Fensterinhalts zum oberen und unteren Fensterrand.	4
marginwidth	Anzahl Pixel für den Abstand des Fensterinhalts zum linken und rechten Fensterrand.	4
nohref	Bereich ohne Verweisziel in einem <area>-Element	3.2, 4
noresize	Kein Verändern der Framefenstergröße ermöglichen.	4
noshade	Flache Darstellung der Linien im <hr>-Tag.	4
nowrap	Zeilenumbruch in Tabellenzellen verhindern.	3.2, 4
profile	URL-Adresse für Meta-Informationen. Verwendbar im <head>-Tag.	4
prompt	Aufforderungstext beim <isindex>-Tag	2, 3.2, 4
rev	Zeichnet einen Link in den Elementen link, a und area als rückbezügliche Verknüpfung aus, komplementär zu zum rel-Attribut.	2, 3.2, 4
rules	Bestimmt die Gitternetzlinien einer Tabelle im <table>-Tag.	4
scroling	bestimmt, ob das Scrollen in einem Frame erlaubt sein soll.	4
scheme	gibt in einem <meta>-Element ein Schema an.	4
size	Schriftgöße in <basefont> und <font>. Im <input>-Tag weiterhin erlaubt (Html5).	3.2, 4
standby	Text der angezeigt wird, während das Objekt eines object-Elements geladen wird.	4
text	Dokumentenweite Textfarbe. Verwendbar im <body>-Element	3.2, 4
valuetype	Gibt in einem param-Element eines object-Elements an, wie der Initialisierungswert von value zu interpretieren ist. Mögliche Werte sind data, ref und object	4
version	Html-Version. Verwendbar im <html>-Tag	4
vlink	Dokumentweite Farbe für Verweise zu bereits besuchten Verweiszielen. Verwendbar im <body>-Tag. CSS-Ersatz: a:visited	3.2, 4
vspace	Vertikale Abstände (in Pixel) oben und unten von Elementen, die von anderen Elementen umflossen werden. CSS-Ersatz: margin-top, margin-bottom	3.2, 4

Obsolete Attribute als Tag-Tabelle

Diese Attribute sind deprecated, obsolet oder wurden nicht von W3C übernommen.

Tag	Attribut	
<a>	name	Falls das Attribut doch vorhanden ist, muss der Attributwert mit dem Attributwert von id identisch sein. <a name="foo" id="foo" Alternative ist id.
<body>	alink background bgcolor link marginbottom marginheight marginleft marginright margintop marginwidth text vlink	
 	clear	
<caption>	align	
<col>	align char charoff valign width	
	align	
<dl>	compact	
<embed>	align hspace vspace	
<hr>	align color noshade sizewidth	
<h*n*>	align	
<iframe>	align allowtransparency frameborder hspace marginheight marginwidth scrolling vspace	
<input>	align hspace vspace	
<img>	border	Falls das Attribut doch vorhanden ist, muss der Attributwert 0 sein.
	align hspace vspace	
<input>	maxlength size	Das Attribut nicht mehr verwenden, wenn das Attribut type im Nummernstatus ist.
<legend>	align	
	type value	
<type>		
<object>	align border hspace vspace	
	compact	
<p>	align	
<pre>	width	
<script>	language	Falls das Attribut doch vorhanden ist, muss der Attributwert den String JavaScript enthalten. Alternative ist type
<table>	align border background bgcolor bordercolor cellpadding cellspacing frame rules width	Falls das Attribut border doch vorhanden ist, sollte es nur den Wert 0 oder 1 haben (als Fallback für ältere Browser).

<tbody>	align background char charoff valign	
<td>	background bgcolor char charoff height nowrap valign width	
<th>	background bgcolor char charoff height nowrap valign width	
<thead>	align background char charoff valign	
<tfoot>	align background char charoff valign	
<tr>	background bgcolor char charoff valign	
	compact type	

Obsolete Attribute

Diese Attribute sind deprecated, obsolet oder wurden nicht von W3C übernommen.

Tag	Beschreibung	
above	Name eines zuvor definierten Layers. Dieser soll bei Überlappung über dem aktuellen Layer angeordnet werden. CSS-Ersatz: z-index	Netscape
behavior	Im marquee-Element die Art, wie sich ein Lauftext verhalten soll. Übliche Angabe: behavior="alternate" (hin- und herlaufender Text).	
below	Name eines zuvor definierten Layers. Dieser soll bei Überlappung unter dem aktuellen Layer angeordnet werden.	Netscape
bgproperties	In Verbindung mit dem obsoleten Attribut background im <body>-Tag: fixes Hintergrundbild, das nicht mitscrollt.	Microsoft
bordercolor	Rahmenfarbe einer Tabelle.	Microsoft
bordercolordark	3D-Rahmen einer Tabelle: dunkle Farbe.	Microsoft
bordercolorlight	3D-Rahmen einer Tabelle: helle Farbe.	Microsoft
clip	Anzeigebereich eines Layers beschneiden.	Netscape
controls	Im img-Element und in Verbindung mit dem dynsrc-Attribut die Angabe, dass Bedienelemente für ein Video angezeigt werden sollen. Standalone-Attribut.	Microsoft
direction	Im marquee-Element die Richtung eines Lauftexts.	Microsoft
framespacing	Abstand von Framefenstern voneinander - framespacing="0" bewirkt nahtlose Frames ohne sichtbare Rahmen.	Microsoft
dynsrc	Funktioniert das img-Element in ein Element für Videos um. Wird anstelle eines src-Attributs verwendet, um ein Video anzuzeigen.	Microsoft
gutter	Spaltenabstand bei mehrspaltigem Textfluss mit dem multicol-Element.	Netscape
internal	Browser-Grafik in einem img-Element referenzieren.	Netscape
left	Linksposition eines Layers	Netscape
leftmargin	Abstand des Seiteninhalts vom linken und rechten Rand des Fensters. <body>-Tag.	Microsoft
loop	Endlosschleife für das Abspielen eines Videos oder eines Sound-Tracks im <embed>-Tag.	Netscape
lowsrc	Grafiken in zwei Versionen laden: <img lowsrc="low.gif" src="high.jpg">	Netscape
scrollamount	Im marquee-Element ein von zwei Aspekten der Laufgeschwindigkeit: Pixelanzahl zwischen den Scroll-Zuständen.	Microsoft
scrolldelay	Im marquee-Element ein von zwei Aspekten der Laufgeschwindigkeit: Verzögerung zwischen zwei Scroll-Zuständen in Millisekunden.	Microsoft
start	Im img-Element und in Verbindung mit dem dynsrc-Attribut der Startauslöser zum Abspielen eines Videos. Üblich ist die Angabe start="mouseover	Microsoft
top	Layer oben positionieren	Microsoft
topmargin	Abstand des Seiteninhaltes vom oberen und unteren Rand des Fensters im <body>-Tag.	Microsoft
wrap	Umbruchkontrolle bei textarea-Element: mögliche Angaben sind virtual (scheinbarer Zeilenumbruch), physical (tatsächliches Einfügen von Zeilenumbruchzeichen) oder off (kein Zeilenumbruch).	Netscape
z-index	Absolute Schichtposition bei Layern, die sich überlappen.	Netscape

Html5 Tags

In Html5 hinzugefügt:

<address>	Textblock für eine Adresse.	Semantik
<article>	Beschreibt einen eigenständigen Textblock, der inhaltlich eigenständig sein kann.	Semantik
<aside>	Randbemerkung.	Semantik
<audio>	Markiert eine Tondatei oder einen Audiostream.	
<bdi>	Markiert Text, der vom umgebenden Inhalt zum Zweck der bidirektionalen Formatierung (z.B. arabischer Text innerhalb von deutschsprachigen Inhalten) isoliert werden soll. Hiermit kann ein Textabschnitt mit einer unterschiedlichen oder unbekannten Textrichtung gekennzeichnet werden.	
<canvas>	Ein bestimmter Bitmap-Bereich für Graphik-Skripte.	
<data>	Für Menschen nicht sichtbare Metainformationen.	
<embed>	Steht für einen Einbindungspunkt für externe Ressourcen.	
<figcaption>	Beschriftung der Abbildung	
<figure>	Abbildung	Semantik
<footer>	Fußteil der Seite oder eines Abschnitts.	Semantik
<header>	Kopfteil der Seite oder eines Abschnitts.	Semantik
<KEYGEN>	Wird in Kombination mit <FORM> für Webbasierende Zertifikatmanagementsystem verwendet.	
<main>	Der Hauptinhalt einer Seite (darf nur einmal vorkommen).	
<mark>	Steht für Text, der aus Referenzgründen hervorgehoben wird, d.h. der in anderem Kontext von Bedeutung ist.	
<math>	kennzeichnet eine mathematische Formel	
<nav>	Beschreibt einen Abschnitt, der Navigationslinks enthält.	Semantik
<rp>	Wird zusammen mit dem Element <ruby> verwendet, um Ruby-Text mit Klammern zu umgeben, die angezeigt werden, wenn das Benutzerprogramm (Browser) keine Ruby-Annotationen unterstützt.	
<rt>	Bezeichnet den Text einer Ruby-Annotation.	
<ruby>	Bezeichnet einen Textteil mit Ruby-Annotationen. Dies sind kurze Aussprachetipps und andere Hinweise, die hauptsächlich für ostasiatische Typografie verwendet werden.	
<section>	Beschreibt einen Abschnitt eines Dokuments (bzw. Seite).	Semantik
<source>	Ermöglicht es Autoren, alternative Medienressourcen (z.B. verschiedene Audio- oder Videoformate) für Medienelemente wie <video> oder <audio> anzugeben.	
<svg>	Um eine SVG-Vektorgraphik einzubetten.	
<time>	Für Zeitangaben.	Semantik
<track>	für zusätzliche Medienspuren (z.B. Untertitel) für <video> oder <audio>	
<video>	Steht für eine Videodatei und die dazugehörigen Audiodateien, sowie die für das Abspielen nötigen Kontrollelemente.	
<wbr>	Hiermit kann die Gelegenheit für einen Zeilenumbruch gekennzeichnet werden, mit dem die Lesbarkeit verbessert werden kann, wenn der Text auf mehrere Zeilen verteilt wird.	

Tabelle

`<table>`	Markiert eine Tabelle.
`<thead>`	Gruppe der Tabellenzeilen, die die Beschriftungen der Tabellenspalten enthalten.
`<th>`	eine Tabellenzeile als Beschriftung
`<tbody>`	Steht für die Spalten, die die eigentlichen Daten einer Tabelle enthalten.
`<caption>`	Kennzeichnet die Beschriftung (Titel) einer Tabelle. Siehe auch Element figcaption Seite 128.
`<tr>`	eine Zeile mit Tabellenzellen.
`<td>`	eine einzelne Tabellenzelle.
`<tfoot>`	Gruppe der Tabellenzeilen, die die Zusammenfassungen der Tabellenspalten enthalten.
`<col>`	Steht für eine Tabellenspalte.
`<colgroup>`	Steht für eine Gruppe aus einer oder mehreren Tabellenspalten.

Formular

`<form>`	Markiert ein Formular. Formulare bestehen aus einer Reihe von Kontrollelementen, deren Werte zur weiteren Verarbeitung an einen Server übertragen werden.
`<fieldset>`	Steht für eine Gruppe von Kontrollelementen.
`<legend>`	Kennzeichnet eine Beschriftung für ein <fieldset>-Element.
`<label>`	Kennzeichnet die Beschriftung für ein Formular-Kontrollelement (z.B. Texteingabefelder).
`<input>`	Steht für ein Feld für Benutzereingaben eines bestimmten Typs. Der Typ (Radiobutton, Ankreuzfeld, Texteingabe, etc.) wird anhand des type-Attributs angegeben.
`<button>`	Markiert einen Button.
`<select>`	Kennzeichnet ein Kontrollelement, mit dem aus einer Reihe von Optionen ausgewählt werden kann.
`<datalist>`	Steht für eine Sammlung vordefinierter Optionen für andere Kontrollelemente.
`<optgroup>`	Steht für eine Reihe logisch gruppierter Auswahloptionen.
`<option>`	Steht für eine Auswahloption innerhalb eines <select>-Elements, oder einen Vorschlag innerhalb eines <datalist>-Elements.
`<textarea>`	Markiert ein Element für mehrzeilige Texteingaben
`<output>`	Markiert das Ergebnis einer Berechnung.

```
<form oninput="result.value=parseInt(a.value)+parseInt(b.value)">
    <input type="range" name="b" value="50" /> +
    <input type="number" name="a" value="10" /> =
    <output name="result"></output>
</form>
```

Mess- und Fortschrittsbalken

meter

Mit dem Element `meter` wird eine Messskala erstellt. Siehe W3.

Attribute:

`min` Mindestwert der Skala (default ist 0)

`max` Maximalwert der Skala (default ist 1)

`value` darzustellender Wert (default ist 0)

Optionale Attribute:

`low` Balken ist rot, wenn `value<low`

`high` Balken ist gelb, wenn `low<=value<high`

`optimum` Balken ist grün, wenn `value>=optimum` .

Beispiel:

```
<meter min="0" max="100" value="25">25 / 100 </meter> <br>
<meter min="0" max="1" value="0.25">0.25 / 1 </meter> <br>
<meter min="0" max="100" value="24" low="25" high="75" optimum="90">24 / 100</meter><br>
<meter min="0" max="100" value="25" low="25" high="75" optimum="90">25 / 100 </meter><br>
<meter min="0" max="100" value="90" low="25" high="75" optimum="90">90 / 100 </meter><br>
```

Ergibt als Output:

meter 25 / 100

meter 0.25 / 1

meter 24 / 100

meter 25 / 100

meter 90 / 100

progress

Mit dem Element `progress` wird eine Fortschrittsanzeige (Balken) erstellt.

Attribute:

`value` darzustellender Wert (default ist 0).
Ist das Attribut nicht vorhanden, wird ein animierter Balken dargestellt

`max` Maximalwert

Beispiel:

```
22 / 100 <progress max="100" value="22">22 / 100</progress>
 - /   - <progress max="100" >-</progress>
```

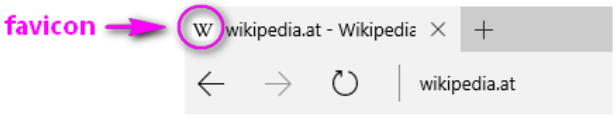
Erzeugt diesen Output:

22 / 100

- / -

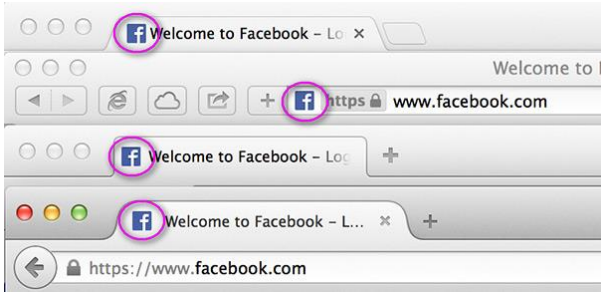
Favicon

Favicons sind Icons, die vom Browser meist neben der URL positioniert werden. Siehe Element `link` Seite 139.
Für jedes externe File wird ein eigener http-Request benötigt.



ico

Das ursprüngliche Grafikformat war ico (siehe Wikipedia). Mögliche Auflösungen waren 16x16 und 32x32 Pixel. Das Favicon lag meist im root-Verzeichnis.



Beispiel:

```
<link rel="shortcut icon" type="image/x-icon"
href="/favicon.ico">
```

png

Icons im Grafikformat png (siehe Wikipedia) haben meist eine Größe von 32x32 und 48x48 Pixel. Empfohlen wird 192x192, da der Browser automatisch auf die optimale Größe skaliert.

Beispiele:

```
<link rel="icon" sizes="32x32" href="/favicon.png" type="image/png" >

<link rel="icon" sizes="192x192" href="favicon.png">
```

Touch Icon

Das Touch-Icon wird nicht im Browser, sondern auf der Startseite (Homescreen) des Gerätes angezeigt.

apple-touch-icon

Icon-Größen:

`76 × 76`	iPad and iPad Mini models with a 1× display
`120 × 120`	iPhone 4s, iPhone 5, iPhone 6 (2× display)
`152 × 152`	iPad and iPad Mini models with a 2× display
`180 × 180`	iPhone 6 Plus (3× display).

Beispiel:

```
<link rel="apple-touch-icon" sizes="180x180" href="favicon1.png">
<link rel="apple-touch-icon" sizes="152x152" href="favicon2.png ">
<link rel="apple-touch-icon" sizes="120x120" href="favicon3.png ">
<link rel="apple-touch-icon" sizes="76x76" href="favicon4.png ">
<link rel="apple-touch-icon" href="favicon.png">

<link rel="apple-touch-icon-precomposed" sizes="76x76" href="favicon4-precomposed.png">
```

Laut devolper.apple.com:

The icon that is the most appropriate size for the device is used. If no sizes attribute is set, the element's size defaults to 60 x 60.

If there is no icon that matches the recommended size for the device, the smallest icon larger than the recommended size is used. If there are no icons larger than the recommended size, the largest icon is used.

Safari on iOS 7 doesn't add effects to icons. Older versions of Safari will not add effects for icon files named with the `-precomposed.png` suffix

Micorosoft TileImage

Für Windows8 beträgt die Größe 144x144 Pixel. `TileColor` ist die Hintergrundfarbe der Kachel.

```
<meta name="msapplication-TileColor" content="#2b5797">
<meta name="msapplication-TileImage" content="/mstile-144x144.png">
```

Ab Windows8.1 und IE11 sucht der IE im root-Verzeichnis nach der Datei browserconfig.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<browserconfig>
  <msapplication>
    <tile>
      <square70x70logo src="/mstile-70x70.png"/>
      <square150x150logo src="/mstile-150x150.png"/>
      <square310x310logo src="/mstile-310x310.png"/>
      <wide310x150logo src="/mstile-310x150.png"/>
      <TileColor>#2b5797</TileColor>
    </tile>
  </msapplication>
</browserconfig>
```

Kachelgrößen und empfohlene Bildgrößen laut Microsoft:

Kachelgröße	Standardkachelabmessungen	Mindestbildgröße	Empfohlene Bildgröße
Klein	70 x 70	56 x 56	128 x 128
Mittel	150 x 150	120 x 120	270 x 270
Breit	310 x 150	248 x 120	558 x 270
Groß	310 x 310	248 x 248	558 x 558

Die maximale Bildgröße für alle Kacheln ist 1024 x 1024.

Empfehlung developers.google.com

```
<!-- icon in the highest resolution we need it for.
Chrome and Opera uses icon.png, which is scaled to the necessary size by the device. To prevent
automatic scaling, you can also provide additional sizes by specifying the sizes attribute. -->
<link rel="icon" sizes="192x192" href="icon.png">

<!-- reuse same icon for Safari -->
<link rel="apple-touch-icon" href="ios-icon.png">

<!-- multiple icons for IE -->
<meta name="msapplication-square310x310logo" content="icon_largetile.png">

<!-- multiple icons for Safari -->
<link rel="apple-touch-icon" href="touch-icon-iphone.png">
<link rel="apple-touch-icon" sizes="76x76" href="touch-icon-ipad.png">
<link rel="apple-touch-icon" sizes="120x120" href="touch-icon-iphone-retina.png">
<link rel="apple-touch-icon" sizes="152x152" href="touch-icon-ipad-retina.png">

<!-- multiple icons for Windows -->
<meta name="msapplication-square70x70logo" content="icon_smalltile.png">
<meta name="msapplication-square150x150logo" content="icon_mediumtile.png">
<meta name="msapplication-wide310x150logo" content="icon_widetile.png">
```

Event Handler

Diese Attribute sind **Event Handlers** für JavaScript. Der Attributwert enthält JavaScript, der aufgerufen wird, wenn das Event erfüllt ist. Das Einfügen von JavaScript in ein HTML-Attribut wird **Inline JavaScript** genannt.

`onabort`	The user aborted loading the page
`onblur`	The user left the object
`onchange`	The user changed the object
`onclick`	The user clicked an object
`onerror`	The script encountered an error
`onfocus`	The user made an object active
`onload`	The object finished loading
`onmouseover`	The cursor moved over an object
`onmouseout`	The cursor moved off an object
`onselect`	The user selected the contents of an object
`onsubmit`	The user submitted a form
`onunload`	The user left the page

Als Beispiel das Event-Attribut `onclick`;

```
<a href="#" onclick=" alert('Ade, schnöde Welt'); ">klick mich</a>
```

Inline-JavaScript sollte nur dann verwendet werden, wenn der Author keinen Zugriff auf das eingebettete oder externe JavaScript (siehe Seite 432) besitzt. Siehe auch Inline Style auf Seite 210.

CSS

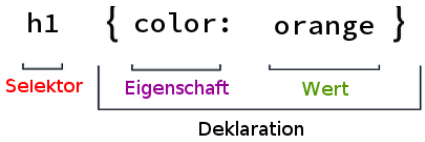
Kapitel 6

CSS

Deklaration

Die Syntax eines CSS Rule-Set:

```
selector{
property: value;
}
```



Beispiel:

```
h1 {
color: orange;
}
```

h1 ist der Selektor, *color* ist Property (Eigenschaft) und **orange** ist Value (Wert) der Eigenschaft.

Ein property-value-Paar bildet eine Deklaration (auch Deklarationspaar genannt). Mehrere Paare sind ein Deklarationsblock.

Value kann eine Größenangabe, eine Konstante, ein Farbwert, ein Bild oder das Ergebnis einer Funktion sein.

Eine (relative) Größenangabe:	`font-size:2em`
Eine vordefinierte Konstante:	`font-color:red; align:center;`
Ein Farbwert (RGB hex):	`font-color:#990000`
Ein Bild:	`background: url(http://www.foo.com/foo.png)`
Berechnete Größenangabe mit calc():	`width: calc(100%/2 - 2*1em - 2*1px)`

Shorthand Properties

Einer Eigenschaft können auch mehrere Werte (multiple values) gleichzeitig zugewiesen werden:

```
h1 {
border: 1px dotted red;
}
```

Siehe **Shorthand Properties** Seite 204.

Deklarationsblock (CSS Rule-Set)

Ein CSS-Rule-Set kann mehr als eine Deklaration haben:

```
h1 {
border: 1px;
border: dotted;
border: red;
}
```

Inline Style

Mit dem Attribut style wird das Element direkt und ohne Selektor formatiert:

```
<h1 style="border: 1px dotted red;">blub</h1>
```

In der Spezifitätsgewichtung von Selektoren (siehe Seite 159) erhält Inline Style 1000 Punkte.

Inline Style sollte nur dann verwendet werden, wenn der Author keinen Zugriff auf die CSS-Rule-Sets besitzt.

Cascading (Prioritätskaskade)

Styles können drei Urheber, und somit gleichzeitig aus drei Quellen stammen (siehe auch W3 und W3):

Author	der Schreiber der CSS
User	Benutzer des Browser
User Agent	der Browser

In absteigender Priorität (5 = höchste Priorität) gibt es folgende 5 Stufen:

5. User styles, die mit !important gekennzeichnet worden sind
4. Author styles, die mit !important gekennzeichnet worden sind
3. Author styles
2. User styles
1. User agent styles

Innerhalb einer Stufe wird die Priorität anhand der Selektoren-Spezifitätsgewichtung ermittelt (siehe Seite 159).

Spezifitätsgewichtung von Selektoren (Selector Specificity Weight))

Wenn zwei gegensätzliche Formatierungsangaben die gleiche Kaskaden-Priorität besitzen (siehe **Cascading** Seite 212), zieht die Spezifitätsgewichtung der Selektoren (Selector Specificity Weight).

ID-Selektoren	jeweils 100 Punkte (höchstes Gewicht)	#
Klassen-Selektoren	jeweils 10 Punkte (auch Pseudo-Klassen)	.
Element-Selektoren	jeweils nur 1 Punkt (auch Pseudo-Elemente)	

Die Anzahl der Selektoren (multipliziert mit ihren Punkten) ergibt die Spezifitätsgewichtung.
CSS in einem style-Tag haben 1000 Punkte und siegen immer über eine Selektorengewichtung.
Universal Selector `*` (siehe Seite 168), **Combinators** `+ > ~` (siehe Seite 169) und **negation pseudo-class** `:not()` beeinflussen nicht die Gewichtung.
Absteigende Wertigkeit der Gewichtungsstrings:

I-K-E		
2-0-0	`#foo1 #foo2`	2 ID-Selektoren
1-2-0	`.foo1 .foo2 #foo3`	2 Klassen-Selektoren, 1 ID-Selektor
1-1-1	`.foo1 a#foo2`	1 Element, 1 Klasse, 1 ID
1-0-1	`a#foo`	1 Element, 1 ID
1-0-0	`#foo`	1 ID
0-2-2	`p.foo1 a.foo2`	2 Elemente, 2 Klassen
0-2-0	`.foo1 .foo2`	2 Klassen
0-1-2	`p a.foo`	2 Elemente, 1 Klasse
0-1-1	`a.foo`	1 Element, 1 Klasse
0-1-0	`.foo`	1 Klasse
0-0-2	`p a`	2 Elemente
0-0-1	`a`	1 Element

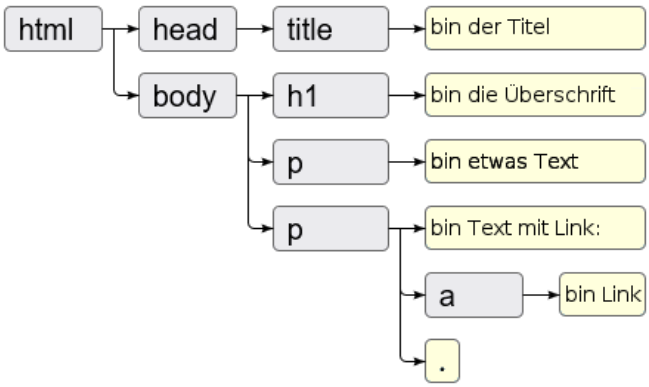
Bei Gleichstand gewinnt der Letztgenannte. CSS in einem style-Tag (Inline Style) siegt über jede Selektorengewichtung und hat immer eine höhere Priorität.

Eltern, Kinder und Nachfahren

Einige Formatanweisungen (CSS-Properties) für ein HTML-Element gelten auch für die (meisten) Nachfahren. Anders gesagt: untergeordnete Kind-Elemente erben Eigenschaften ihrer Elternelemente.
Nicht vererbt werden zB Farbanweisungen in Links (<a>) und die Eigenschaft border.

Diese Beispiel (siehe color:red im body):

```
<html>
    <head><title>bin der Titel</title>
</head>
    <body style="color:red">
       <h1>bin die Überschrift</h1>
       <p>bin etwas Text</p>
    <p>bin Text mit Link:
       <a href="">bin Link</a>.</p>
  </body>
 </html>
```



Erzeugt diesen Output:

bin die Überschrift

bin etwas Text
bin Text mit Link: bin Link.

Wurzel (root)

Das Element <html> ist das root element (Stammelement) und hat keine Vorfahren.

Eltern (parent)

Besitzt ein Element mindestens ein Kind (child) wird es **parent element** genannt. Ein Kind ist ein direkter Nachfahre.
Beispiel:

<html> ist Elter von <head> und <body>

<body> ist Elter von <h1>

Kind (child)

Ein direkter Nachfahre wird Kind genannt.
Siehe auch **Child Combinator** Seite 170 .
Beispiel:

<head> und <body> sind Kinder von <html>.

Der <title> Tag ist das Kind von <head>.

Nachfahre (Descendant)

Alle Elemente (Html-Tags) innerhalb eines bestimmten Elements, sind seine Nachfahren.
Siehe auch **Descendant Combinator** Seite 170 .
Beispiel: alle Tags zwischen <html> und </html> sind die Nachfahren von <html>.

Geschwister (Sibling)

Elemente, die nebeneinander in der gleichen Baumebene stehen (gleiches Eltern-Element), werden Geschwister genannt.

Für Geschwister siehe **General Sibling Combinator** Seite 172.

Für angrenzende Geschwister siehe **Adjacent Sibling Combinator** Seite 171.

Beispiel: `<body>` ist ein Geschwister von `<head>`.

Der `<h1>` Tag ist ein Geschwister von `<p>`.

Dieses Beispiel:

```
<style>
h1  {
color: white;
background-color: black;
}
</style>
<h1> blub1 </h1>
<h1> <b> blub2 </b> </h1>
<h1> <b> <i> blub3 </i> </b> </h1>
<b> blub4 </b>s.
```

erzeugt diesen Output:

blub1
blub2
blub3
blub4

Vererbung (Inheritance)

Einige CSS-Eigenschaften eines HTML-Elementes vererben sich auf die Nachfahren des Elementes (siehe W3).

Diese automatische Vererbung kann aufgehoben werden durch:

1) Keyword inherit (siehe Seite 162)
   explizite Vererbung (bei Eigenschaften ohne automatischer Vererbung)

2) Keyword initial (siehe Seite 162)
   Vererbung abschalten (bei Eigenschaften mit automatischer Vererbung)

3) Selektoren (siehe Seite 165)

4) Kombinatoren (siehe Seite 169)
   Für Kind-Elemente siehe **Child Combinator** Seite 170. Für Nachfahren-Elemente siehe **Descendant Combinator** Seite 171.
   Für Geschwister siehe **General Sibling Combinator** Seite 172. Für angrenzende Geschwister siehe **Adjacent Sibling Combinator** Seite 171.

5) Prioritätskaskade (Cascading) (siehe Seite 212)

6) Keyword unset (siehe Seite 163)

Für Value Processing siehe Seite 164.

inherit

Bei Eigenschaften, die nicht automatisch vererbt werden, kann die Vererbung durch das Keyword `inherit` (Property Value) erzwungen werden.

Mit dem Keyword `initial` (Property Value) kann die automatische Vererbung abgeschaltet werden.

Das Element `h1` soll die Schriftgröße vom Elternelement erben:

```
<style>
   h1 {font-size:inherit; color:red;}
</style>

<b> blub <h1> foobar </h1> </b>
```

Ergibt als Output:

blub

foobar

initial

Die Vererbung kann durch das Keyword `initial` (Property Value) abgeschaltet werden.

Das Element `b` soll nicht die Schriftgröße vom Elternelement erben:

```
<style>
i {color:red;}
b {color:initial; }
</style>
<i> blub <b> foobar </b> </i>
```

Erzeugt diesen Output:

blub foobar

`!important` im Elternelement hat keine Wirkung auf das Kind:

```
<style>
   i {color:red !important;}
   b {color:initial; }
</style>
<i> blub <b> foobar </b> </i>
```

Ergibt als Output:

blub

foobar

unset

Der Wert `unset` (Keyword) macht eine bestehende Formatierung rückgängig.

Die Spezifitätsgewichtung der Selektoren (siehe Seite 159) muss mindestens gleich groß sein.

Mit `reset` wird `background-color` zurückgesetzt:

```
<style>
   b { background-color: yellow; }
   b { background-color: unset; }
</style>
<b>foobar</b>
```

Ergibt als Output:

foobar

all

Die Eigenschaft (Property) `all` steht für alle Eigenschaften. Wird `all` auf `unset` gesetzt, werden alle Eigenschaften zurückgesetzt. Die Spezifitätsgewichtung der Selektoren (siehe Seite 159) muss mindestens gleich groß sein.

```
<style>
b { background-color: yellow; font-style:italic;}
b { all: unset; }
</style>
<b>foobar</b>
```

Ergibt als Output:

foobar

Im 2. Beispiel ist die Spezifität der älteren Formatierung höher und der `reset` zieht nicht:

```
<style>
html b { background-color: yellow; font-style:italic;}
 b { all: unset; }
</style>
<b>foobar</b>
```

Ergibt als Output:

foobar

Value Processing

Die Reihenfolge zum Ermitteln des Endwertes einer Eigenschaft (Property):

1) alle Declared Values aus allen CSS-Quellen ermitteln
2) den Cascaded Value in der Kaskade ermitteln
3) Specified Value durch Vererbung und Spezifizierung
4) Computed Value durch Umwandlung der relativen Einheiten
5) Used Value durch Berechnung der relativen Einheiten, die vom Rendern abhängig sind
6) Actual Value durch Anpassung an das jeweilige Ausgabemedium

Siehe auch W3 und W3.

Declared Value

Alle deklarierten Formatanweisungen sammeln. Quellen sind User-CSS, Author-CSS und User-Agent-CSS.

Cascaded Value

Für jede einzelne Deklaration wird die Priorität durch die Kaskade (siehe Seite 212) ermittelt.

Specified Value

Der Cascaded Value wird durch Vererbung (siehe Seite 161), Selektoren (Seite 165) und Kombinatoren (siehe Seite 169) näher spezifiziert (siehe Spezifitätsgewichtung der Selektoren Seite 159).

Der Wert vor der Überprüfung der Vererbung, wird **Initial Value** genannt (siehe initial Seite 162). Der vererbte Wert wird **Inherit Value** (siehe Seite 162) bezeichnet.

Computed Value

Aus Keywords und relativen Längeneinheiten wird der Computed Value (berechneter Wert) ermittelt. Für den Computed Value ist kein Rendern der Seite notwendig. Sind Nachfahren vorhanden, wird der Computed Value vom Elternelement an seine Nachfahren vererbt.
Beispiele:

das Keyword small wird durch die Pixel-Schriftgröße ersetzt.
die relative Längeneinheit em wird durch px ersetzt.
aus einer relativen URL wird eine absolute URL

Used Value

Ist die Längeneinheit abhängig vom Rendern der Seite (relativ zu anderen Objekten der Seite), muss der Used Value berechnet werden.

Beispiel: width: auto
break-before

Actual Value

Abhängig vom Ausgabemedium muss optional aus dem Computed Value der **Actual Value** (tatsächliche Wert) berechnet werden.

Selektoren

Ein Selektor (selector) grenzt die Anzahl der möglichen Ziele für eine Formatierungsanweisung ein.

Es gibt 4 Hauptarten von Selektoren

	Beispiel	
Element-Selektor:	`p {color:red;}`	Seite 165
Klassen-Selektor:	`.foo {color:red;}`	Seite 166
ID-Selektor.	`#foo {color:red;}`	Seite 166
Attribut-Selektor:	`h1[title]`	Seite 167

Und 2 davon abgeleitete Unterarten:

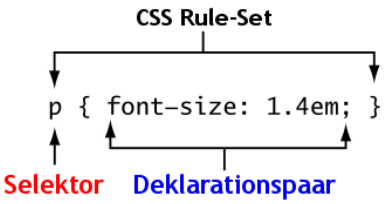
Pseudo-Element:	`::after`	Seite 202
Pseudo-Klasse:	`:enabled`	Seite 188

Pseudoelemente müssen ab CSS3 mit 2 Doppelpunkten beginnen (davor war die Notation nur mit einem Doppelpunkt).

Es ist immer nur ein Pseudoelement pro Selektor möglich. Selektoren-Gewichtung siehe Seite 159.

Jedes CSS-Rule-Set beginnt zwingend mit mindestens einem Selektor.
Gefolgt vom CSS-Block, der in geschweiften Klammern steht:

```
selector {
Eigenschaft: Wert;
}
```



Die Eigenschaft (Property) und der Eigenschaftswert (Value) bilden ein Deklarationspaar.
Im Beispiel: die Formatanweisung `font-size:1.4em` gilt nur für das Element `<p>`.

Elementselektor

Ein Elementselektor beziehen sich auf ein bestimmtes Html-Tag (Html-Element).
In diesem Beispiel bezieht sich die `font-size` auf das Element `<h1>` (h1 ist somit ein Element-Selektor).
`font-size` ist die Eigenschaft (property) und `120%` ist der Wert (value) der Eigenschaft:

```
h1 {
font-size: 120%;
}
```

In der Spezifitätsgewichtung von Selektoren (Seite 159) erhält der Elementselektor 1 Punkt.

ID-Selektor

Ein ID-Selektor bezieht sich auf eine bestimmte ID eines id-Attributes (also auf einen bestimmten Attributwert vom Attribut `id`).

Als Selektor wird der ID immer eine **Raute** vorangestellt.

In diesem Beispiel ist foo die ID:

```
<style>
#foo {
color: white;
background-color: black;
}
</style>
<h1 id="foo">blub</h1>
<h1 id="foo2">blub2</h1>
```

In der Spezifitätsgewichtung von Selektoren (Seite 159) erhält der ID-Selektor 100 Punkte. Die höchste Punkteanzahl, da mit der ID-Selektion am genauesten selektiert wird (**most specific**).

Die ID sollte in der Seite eindeutig sein (nur einmal, bei genau einem Tag). Wenn man mehr als nur einen ganz bestimmten (einzelnen) Tag formatieren möchte, sollte man den Klassen-Selektor verwenden.

Klassen-Selektor

Ein Klassen-Selektor bezieht sich auf eine bestimmte Klassenbezeichnung eines class-Attributes (also auf einen bestimmten Attributwert vom Attribut `class`).

Als Selektor wird der Klassenbezeichnung immer ein Punkt vorangestellt.

In diesem Beispiel ist foo die Klasse. Sie wird in in zwei `<h1>` Tags verwendet:

```
<style>
.foo {
color: white;
background-color: black;
}
</style>
<h1 class="foo">blub</h1>
<h1 class="foo">blub2</h1>
<h1 class="foo3">blub3</h1>
```

Im Gegensatz zum ID-Selektor darf sich die Klasse auf mehrere Tags beziehen.

In der Spezifitätsgewichtung von Selektoren (siehe Seite 159) erhält der Klassenselektor 10 Punkte.

Im Stringwert vom Attribut `class` können mehrere Klassenbezeichnungen kombiniert werden:

```
<h1 class="foo1 foo2 foo3">blub3</h1>
```

Der Attributwert enthält die Klassenbezeichnungen `foo1`, `foo2` und `foo3`.

Attribut-Selektor

Ein Attribut-Selektor bezieht sich auf ein Attribut von einem Element. Die einfachste Form sucht nur Elemente, die das angegebene Attribut besitzen:

```
[Attribut]
```

Einschränkung auf ein bestimmtes Element:

```
Element[Attribut]
```

Selektiere `h1` Elemente mit dem Attribut `title`:

```
<style>
h1[title] {
color: white;background-color: black;
}
</style>
<h1> blub1 </h1>
<h1 title="foobar"> blub2 </h1>      <!-- hier besitzt h1 das Attribut title -->
```

Erzeugt als Output:

blub1

blub2

bestimmter Attributwert

Selektiert Elemente mit einem bestimmten Attribut und einem bestimmten Attributwert:

```
Element[Attribut="Wert"]
```

Selektiere `type`-Attribute (in `input` Elementen), deren Attributwerte `submit` sind:

```
input[type="submit"]
```

Attributwert mit bestimmtem String beginnend

Selektiert Elemente, deren Attributwerte mit der Zeichenkette des angegebenen Wertes anfangen.

```
Element[Attribut^="Wert"]
```

Selektiere `href`-Attribute (in `a` Elementen), deren Attributwerte mit `http` beginnen:

```
a[href^="http"]
```

Attributwert mit bestimmtem String endend

Selektiert Elemente, deren Attributwerte mit der Zeichenkette des angegebenen Wertes enden.

```
Element[Attribut$="Wert"]
```

Selektiere `href`-Attribute (in `a` Elementen), deren Attributwerte mit `.pdf` enden:

```
a[href$=".pdf"]
```

Attributwert mit bestimmtem String

Selektiert Elemente, bei denen die Zeichenkette im Attributwert vorkommt.

```
Element[Attribut*="Wert"]
```

Selektiere alle Links zu Wikipedia:

```
a[href*="wikipedia.org"]
```

Leerzeichengetrennte Teilattributwerte

Selektiert Elemente in Attributwerten, deren Teilwerte mit Leerzeichen getrennt sind.

```
Element[Attribut~="Wert"]
```

Suche nach dem Attribut `class`, dessen Klassenbezeichnungen die Bezeichnung `foo` enthält:

```
[class~="foo"]
```

Der Selektor zieht in diesem Beispiel:

```
<b class="test foo blub">
```

Sprachcode-Muster

Sucht nach RFC3282 Sprachcode-Muster.

Suche nach dem Attribut lang, dessen Attributwert `de` ist, oder mit `de-` beginnt (de-ch, de-at).

```
[lang|="de"]
```

Siehe auch Pseudoklasse :lang() Seite 202.

Universalselektor

Der Universalselektor * grenzt nicht ein, sondern bezieht sich auf alle Elemente.

Diese font-size-Angabe bezieht sich auf alle Elemente: 168

```
* {font-size: 120%;}
```

Aufgerufenes Ankerziel in allen Elementen grün färben:

```
*:target {  background-color:#009900;}
```

Steht der Universalselektor am Beginn, kann er auch weggelassen werden:

```
:target {  background-color:#009900;}
```

Jeder direkte Nachfahre (Kind) vom Element body:

```
body > * {  background-color:#009900;}
```

Pseudoklassen-Selektor

Mit Pseudoklassen lassen sich dynamische und interaktive Elemente selektieren. Siehe Pseudoklassen Seite 188.

Pseudoelement-Selektor

Mit Pseudoelemente lässt sich auf Fake-Elemente selektieren (sind nicht im DOM). Siehe Pseudoelemente Seite 202.

Kombinatoren

Ein Kombinator (Combinator) stellt eine Beziehung zwischen 2 Selektoren her. Der Gruppen-Kombinator erweitert das Ziel der Formatierungsanweisungen, die restlichen Kombinatoren grenzen die Zielgruppe ein.

Selektor1 *Kombinator* *Selektor2*

Ziel eingrenzen

Beim **Child Combinator >** zieht b nur dann, wenn b der direkte Nachfahre (Kind) von h1 ist: Seite 170

```
<h1>  >  <b>
```

Beim **Descendant Combinator** zieht <b> nur dann, wenn <b> ein Nachfahre von h1: Seite 171

```
<h1>  Leerzeichen <b>
```

Beim **General sibling combinator ~** zieht <b> nur dann, wenn <b> ein Geschwister von h1:

```
<h1>  ~  <b>
```

Beim **Adjacent sibling combinator +** zieht b nur dann, wenn b ein angrenzendes Geschwister von h1 ist:

```
<h1>  +  <b>
```

Ziel erweitern

Beim **Gruppen-Kombinator ,** bezieht sich die Formatanweisung auf beide Elemente (h1 und b):

```
<h1>  ,  <b>
```

Kombinatoren verschachteln

`b` zieht nur dann, wenn `b` das Kind von `i` ist, und `i` das Kind von `h1` ist:

```
<h1>  >  <i>  >  <b>
```

a Elemente, die in einem div Element mit dem Attribut `id="foo"` stehen:

```
div#foo a
```

alle p Elemente mit der Klasse `foo`:

```
p.foo
```

alle em Elemente, die sich innerhalb eine h1 Elementes befinden (h1 sind die Nachfahren):

```
h1 em
```

Gruppen-Kombinator

Mit dem Gruppen-Kombinator , (Komma) kann eine Formatanweisung mehreren Selektoren gleichzeitig zugewiesen werden.

Aus

```
h1 {font-size: 120%;}
h2 {font-size: 120%;}
```

wird:

```
h1, h2 {font-size: 120%;}
```

Child Combinator (Kind)

Der Child Combinator ist das > (größer-als Zeichen). Die Formatierung zieht nur dann, wenn das 2. Element der direkte Nachfahre (Kind) vom 1. Element ist.

Für das 1. Kind (Erstgeborener) muss die Pseudo-Klasse :first-child pseudo-class verwendet werden.

In diesem Beispiel ist das Element <b> das Kind von <h1>:

```
<style>
h1 > b {
color: white;
background-color: black;
}
</style>
<h1> blub1 </h1>
<h1> <b> blub2 </b> </h1>       <!-- nur hier ist b das Kind von h1 -->
<h1> <i> <b>blub3</b> </i> </h1>
<h1> </h1>  <b>blub4</b>
```

Ergibt als Output:

blub1

blub2

blub3

blub4

Descendant Combinator (Nachfahre)

Der **Descendant Combinator** ist das *Leerzeichen* (Space). Die Formatierung zieht nur dann, wenn das 2. Element ein Nachfahre (arbitrary descendant) vom 1. Element ist (direkte und ferne Nachfahren). Somit muss sich das 2. Element noch vor dem End-Tag des 1. Elements befinden.
Zwischen den Teilselektoren darf sich nur Whitespace befinden.

In diesem Beispiel muss das Element <b> ein Nachfahre von <h1> sein.

```
<style>
p  b {
color: white;
background-color: red;
}
</style>

<p> blub1 </p>
<p> <b>        blub2 </b> </p>              <!--  ist der direkte Nachfahre von p -->
<p> <i>   <b>  blub3 </b> </i> </p>         <!--  ist ein Nachfahre von p -->
<p> </p>  <b>  blub4 </b>
```

Ergibt als Output:

blub1

blub2

blub3

blub4

Adjacent Sibling Combinator (angrenzendes Geschwister)

Der **Adjacent Sibling Combinator** ist + (Plus). Die Formatierung zieht nur dann, wenn das 2. Element ein direkt nachfolgender Nachbar (direkt angrenzendes Geschwister) vom 1. Element ist.

Beispiel:

```
<style>
p + b {
    color: white;
    background-color: black;
 }
</style>

<p> blub1 </p>
<p> <b> blub2 </b> </p>
<p> <b> blub3 </b></p>
<p>  </p>
<b> blub4 </b>              <!-- b ist ein direkt angrenzendes Geschwister von p -->
<b> blub5 </b>
```

Ergibt diesen Output:

blub1

blub2

blub3

blub4 **blub5**

General Sibling Combinator (Geschwister)

Der **General sibling combinator** ist ~ (Tilde). Die Formatierung zieht nur dann, wenn das 2. Element ein Nachbar (Geschwister) vom 1. Element ist.

Beispiel:

```
<style>
h1  ~  b {
color: white;
background-color: black;
}
</style>
<body>
<h1> blub1 </h1>
<h1> <i> <b> blub2 </b> </i></h1>
<h1><i> blub5 </i> </h1>
<br>
<b> blub6 </b>   <!--  b ein Nachbar (Geschwister) auf der gleichen Ebene von h1 -->
</body>
```

Ergibt diesen Output:

blub1

blub2

blub3

blub4 blub5

Box-Modell

Jedes Element (Html-Tag) im `<body>` stellt ein Rechteck mit mindestens 4 Ecken dar. Es gibt **Block-Level-Elemente** und **Inline Elemente**. Siehe auch mozilla.org.

Block versus Inline

Innerhalb des Textflusses kann man zwischen Block- und Inline-Elementen unterscheiden.

1) **Block-Elemente**

   Ein Block-Element erzeugt im Textfluss einen neuen Absatz. Block-Elemente nehmen so viel Breite wie möglich (wenn keine width Angabe vorhanden), und so so viel Höhe wie notwendig.

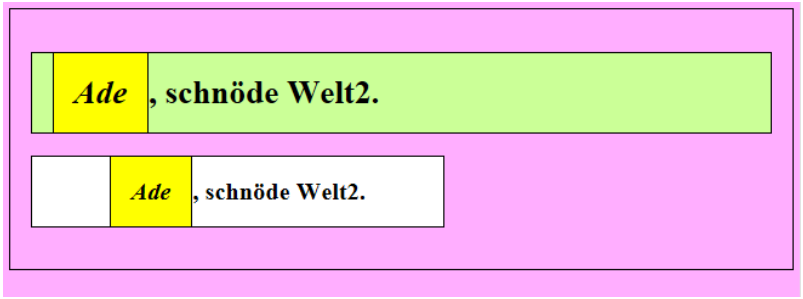
2) **Inline-Elemente**

   Sie nehmen nur so viel Breite und Höhe wie notwendig. Sie erzeugen keinen Absatz.
   width-Angaben haben keine Wirkung.

Dieses Beispiel:

```
<style>

body { border: solid 1px black;
       padding: 20px;
       background-color: #ffaeff; /* rosa */
 }

h1 {border: solid 1px black;
    padding: 20px;
    background-color: #cbff97; /* grün */
}

h2 {border: solid 1px black;
    padding: 20px;
    background-color: #ffffff; /* weiß */
    width: 50%;
    text-align: center;
 }

i { border: solid 1px black;
    padding: 20px;
 background-color: #ffff00; /* gelb */
 }

</style>
<h1> <i>Ade</i>, schnöde Welt2.</h1>
<h2> <i>Ade</i>, schnöde Welt2.</h2>
```

Ergibt diesen Output:



Das Block-Level-Element `<body>` (rosa Fläche) erstreckt sich über die gesamte Fensterbreite.
Das Block-Level-Element `<h1>` (grüne Fläche) erstreckt sich über die gesamte Breite.
Das Inline-Level-Element `<i>` (gelbe Fläche) erstreckt sich nur über die benötigte Breite.

Das Block-Level-Element `<h2>` (weiße Fläche) erstreckt sich nicht über die gesamte Breite, da `width:50%`

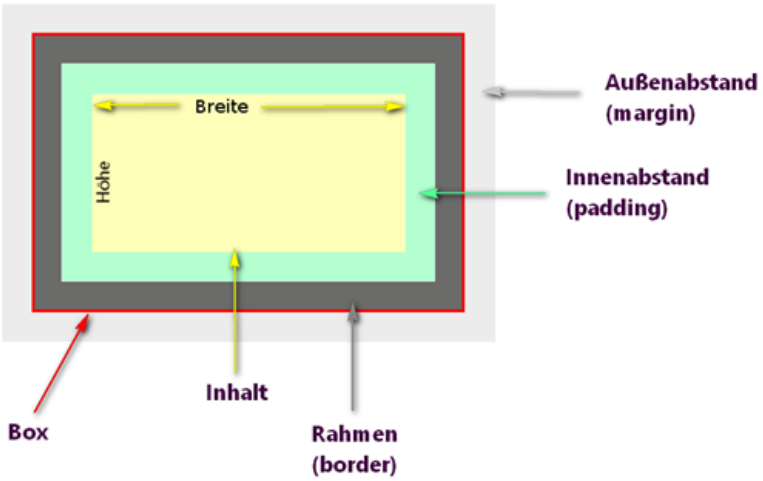
Aus jedem Element ein Block-Level-Element: `display: block;`
Aus jedem Element ein Inline-Level-Element: `display: inline;`

Block-Level-Elemente

Folgende Html-Tags (Elemente) sind Block-Level-Elemente. Sie lösen einen Zeilenumbruch aus. Und nehmen so viel Breite wie möglich, und so so viel Höhe wie notwendig. Siehe auch **Block versus Inline** Seite 172 .

```
<address>       <article>     <aside>
<blockqoute>    <canvas>      <div>
<dl>            <fiedset>     <figcaption>
<figure>        <footer>      <form>
<h1>,<h2>,…     <header>      <hgroup>
<hr>            <li>          <main>
<nav>           <noscript>    <ol>
<output>        <p>           <pre>
<section>       <table>       <td>
<tfoot>         <ul>          <video>
```

Box-Abstände



Bei jeder **Block-Level-Box** lassen sich diese Abstände bestimmen:

`margin`	Außenabstand
`border`	Rahmen (dunkelgrau)
`padding`	Innenabstand (türkis)
`width`	Breites des Inhalts (gelb)
`heigth`	Höhe des Inhalts (gelb)

Siehe Längeneinheiten Seite 209. Mit diesen Sub-Properties lassen sich `margin` und `border` detaillierter definieren:

`top`	oben
`right`	rechts
`bottom`	unten
`left`	links

Der Außenabstand bezieht sich auf andere Box-Elemente und auf die Entfernung zum Seitenrand.
Dieses Beispiel:

```
<style>
body { border: solid 1px black;
    padding: 20px;
}
h1 {
    border-top: solid 10px black;
    border-right: solid 40px red;
    border-bottom: solid 60px blue;
    border-left: solid 20px yellow;
       padding: 20px;
    background-color: green;
}
</style><h1> <i>Ade</i>, schnöde Welt.</h1>
```

Erzeugt diesen Output:

Box-Gesamtbreite

Die width-Angabe bestimmt die innere Content-Breite (exklusive Rahmen und Innenabstand) von Block-Level-Elementen. Der Internet Explorer nahm ursprünglich width für die Gesamtbreite. Dieses Verhalten wurde per CSS3 als eigene Eigenschaft (box-sizing: border-box) nachgereicht.
Bei Inline-Elementen wird die width-Angabe ignoriert.

box-sizing: content-box

Diese Methode ist das default-Verhalten. Eine explizite Angabe ist nicht notwendig.
Die Gesamtbreite einer Box ergibt sich aus:

Rahmen +	Innenabstand	+ Innenbreite	+ Innenabstand	+ Rahmen
border +	padding	+ width	+ padding	+ border

Siehe auch Box-Abstände Seite 174.
Dieses Beispiel:

```
<style>
h1 {
 margin: 15px;                    /* Außenabstand */

 border-left: solid 10px green;   /* Rahmen links */
 padding: 20px;                   /* Innenabstand links+rechts (20+20) */
 width: 200px;                    /* Innenbreite */
 border-right: solid 60px red;    /* Rahmen rechts */
 box-sizing: content-box;         /* diese Angabe ist nicht notwendig, da default */

 background-color: blue;          /* 310 = 10 + 20 + 200 + 20 +60 */
}
</style>
<h1>x</h1>
<section style="width:325px; background-color: black">x</section>
```

Ergibt diesen Output:

box-sizing: border-box

Durch die Angabe von `box-sizing: border-box` und `width: 310px` wird der identische Output erzeugt.

```
h1 {
 margin: 15px;                    /* Außenabstand */
 border-left: solid 10px green;   /* Rahmen links */
 padding: 20px;                   /* Innenabstand links+rechts (20+20) */
 width: 310px;                    /* Gesamtbreite, da box-sizing: border-box */
 border-right: solid 60px red;    /* Rahmen rechts */

 background-color: blue;
 box-sizing: border-box;
}
</style>
<h1>x</h1>
<section style="width:325px; background-color: black">x</section>
```

Box Margin

Bei Block-Level-Elementen bestimmt die Eigenschaft `margin` den Abstand zu den äußeren Elementen und zum Seitenrand.
Bei Inline-Elementen bestimmt `margin` nur den horizontalen Abstand.

Beispiel ohne width

Als Beispiel drei Block-Level-Elemente:

```
<style type="text/css">
h1 {color:white;height:2em}
</style>
<h1 style=" background-color:red;">h1</h1>
<h1 style=" background-color:blue;">h2</h1>
<h1 style=" background-color:green;">h3</h1>
```

Ergibt als Output:

positiver Margin-Wert

Ein positiver Margin-Wert beim 2. Block-Level-Element vergrößert den Außenabstand zu den anderen Elementen und zum Seitenrand (h2 entfernt sich runter von h1. Und h3 rückt ebenfalls runter):

```
<h1 style=" background-color:blue;margin:1.3em">h2</h1>
```

Ergibt als Output:

negativer Margin-Wert

Ein negativer Margin-Wert beim 2. Block-Level-Element verkleinert den Außenabstand. h2 schiebt sich über h1.
Der Text von h2 verschwindet links vom Seitenrand. h3 schiebt sich über h2 (Überlappung):

```
<h1 style=" background-color:blue;margin:-1.3em">h2</h1>
```

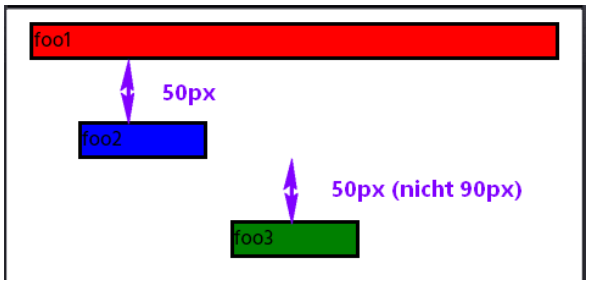
Ergibt als Output:

Beispiel mit width

Die Angabe von **width** verhindert, dass das blaue Block-Level-Element über die gesamte Breite reicht. Die zusätzliche Angabe von **auto** bei der Eigenschaft `margin` zentriert das Element (roter Balken).

```
<style>
.foo1 {
 margin: 10px;
 background-color:red;
}
.foo2 {
  margin: 50px;
 width: 100px;
 background-color:blue;
}
.foo3 {
  margin: 40px auto;
 width: 100px;
 background-color:green;
}
</style>
<p class="foo1">foo1</div>  <p class="foo2">foo2</div>  <p class="foo3">foo3</div>
```

Bei überlappenden Margin-Bereichen, wird der größere Margin-Abstand verwendet:



Inline-Level Elemente

Die Angabe margin hat nur horizontal eine Wirkung.

Keine Wirkung hat width und height: Inline-Elemente nehmen immer nur so viel Breite, wie sie benötigen.

Die Höhe wird von der Schriftgröße und der Zeilenhöhe bestimmt.

Eingebettete Bilder sind **Replaced Inline Elements** und befinden sich an der Baseline.

Beispiel:

```
<span style="background-color:#ffff66;">Fehlt es mir etwa an Verrückten,<br>daß ihr diesen Mann
hergebracht habt, damit er gegen mich
<b style="border:solid">tobe</b>? Sollte der in mein Haus kommen?
<img src="fruehlingsblume.jpg" height="50">
Fehlt es mir etwa an Verrückten, daß ihr diesen Mann hergebracht habt,
<b style="margin:80px;border:solid">damit er gegen mich tobe</b>? Sollte der in mein Haus
kommen? </span>
```

Ergibt diesen Output:

Fehlt es mir etwa an Verrückten,
daß ihr diesen Mann hergebracht habt, damit er gegen mich **tobe**? Sollte der in mein Haus kommen? Fehlt es mir etwa an Verrückten, daß ihr diesen Mann hergebracht habt,
damit er gegen mich tobe ? Sollte der in mein Haus kommen?

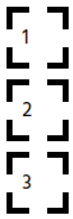
Box Display

Mit der Eigenschaft `display` kann aus jedem Html-Element ein **Block-Level** (`block`) oder ein **Inline-Element** (`inline`) gemacht werden. Mit dem Wert `none` wird die Box vom Browser nicht gerendert. Mit `visibility:hidden` wird sie auf unsichtbar gestellt (und verbraucht optisch keinen Platz).

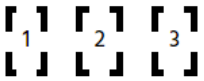
Das div Element ist ein Block-Level-Element:

```
<style type="text/css">

div {
       width: 1.5em; height: 1.5em;
       margin: 0.5em; padding: 0.5em;
       border: 5px dashed black;
       /* display: inline */
 }

</style>

<div>1</div> <div>2</div> <div>3</div>
```

Ergibt als Output:



Wird die 6. Zeile eingeschaltet, wird aus dem div-Element ein Inline-Element und bewirkt somit keinen Zeilenumbruch mehr:



Die Blöcke sind schmäler, da width-Angaben bei Inline-Elementen nicht greifen.

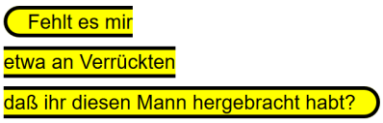
Box Properties

box-decoration-break

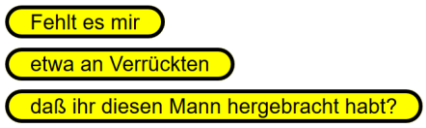
Die CSS3-Eigenschaft `box-decoration-break` steuert das Verhalten bei einem Zeilenumbruch in einem Inline-Element.

```
<style type="text/css">

.foo {
        background:yellow;
        padding: 0em 1em;
        border-radius: 16px;
        border-style: solid;
        line-height: 2;
       /* box-decoration-break: clone; */
    }
</style>
<span class="foo">Fehlt es mir<br>etwa an Verrückten<br>
daß ihr diesen Mann hergebracht habt?</span>
```

Beim Wert `slice` (Firefox default) wird die Box einfach abgeschnitten:



Beim Wert `clone` wird die linke Seite kopiert und gespiegelt (IE-Edge Verhalten):



height

Die Eigenschaft `height` bestimmt die Höhe eines Elementes. Siehe Längeneinheiten Seite 209.

Mögliche Werte:

auto die Höhe hängt vom Inhalt ab (so hoch wie notwendig)

Wert

Prozentwert

width

Die Eigenschaft `width` bestimmt die Breite eines Elementes. Siehe Längeneinheiten Seite 209.

Mögliche Werte:

auto die Breite hängt vom Inhalt ab (so hoch wie notwendig)

Wert

Prozentwert

Fluss und Positionierung

Der normale (default) Seitenfluss für Inline-Elemente: Positioniere ein Element nach dem anderen, von links oben nach rechts oben (aneinanderreihend). Beginne eine neue Zeile, wenn der Seitenrand erreicht wird.
Block-Level-Elemente lösen zusätzlich vor dem Positionieren einen Zeilenumbruch aus.

Dieser Normal Flow kann für jedes einzelne Element geändert werden. Bei `float:right` schwimmt das Element nach rechts. Die nachfolgenden Elemente bleiben im Normal Flow und positionieren sich links von diesem Element.

Mit `clear:left` kann für jedes einzelne Element bestimmt werden, ob beim Platzieren der Platz links vom Element frei sein muss.
Bei `clear:right` muss beim Platzieren der Platz rechts frei sein.

float

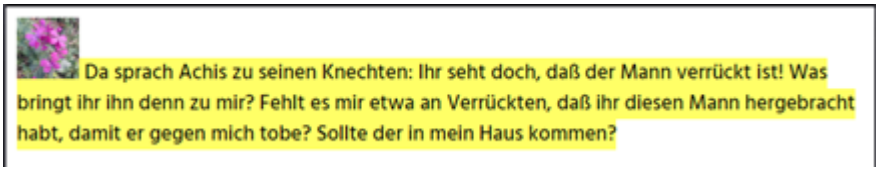
Die Eigenschaft `float` bestimmt, ob und wie ein Element innerhalb von seinem Parent-Element schwimmen soll.
Bei float:right schwimmt das Element nach rechts. Die nachfolgenden Elemente bleiben im Normal Flow und positionieren sich links von diesem Element.
Mögliche Werte:

`left`	Element soll nach links schwimmen
`right`	Element soll nach rechts schwimmen
`none`	default. Normal Flow.

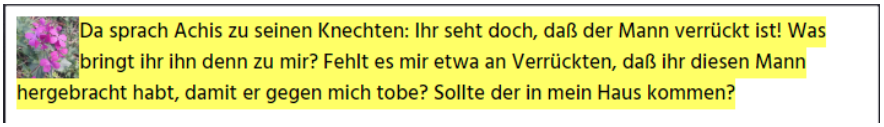
Dem Bild wird `float:none` zugewiesen:

```
<style>
.blume {
  float: none;
}
</style>
<span style="background-color:#ffff66;"><img src="f.jpg" class="blume">Da sprach Achis zu seinen Knechten: Ihr seht doch, daß der Mann verrückt ist! Was bringt ihr ihn denn zu mir? Fehlt es mir etwa an Verrückten, daß ihr diesen Mann hergebracht habt, damit er gegen mich tobe? Sollte der in mein Haus kommen? </span>
```

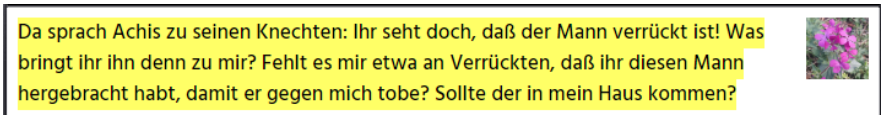
Bei `float:none` ist der Output:



Bei einer Änderung auf `float:left` rutscht der Text rauf und das Bild bleibt links:



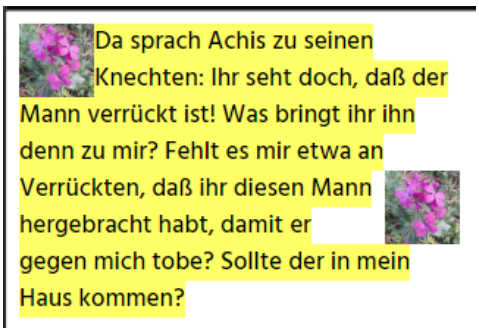
Bei einer Änderung auf `float:right` rutscht der Text rauf und das Bild nach rechts:



Ein Beispiel mit 2 Bildern in einem section-Element:

```
<style>
    .blume {
       float:left;
       height:50px;
    }

    .blume2 {
       float:right;
       height:50px;
    }
</style>

<section style="width:300px">
<span style="background-color:#ffff66;"><img src="f.jpg" class="blume">Da sprach Achis zu seinen
Knechten: Ihr seht doch, daß der Mann verrückt ist! Was bringt ihr ihn denn zu mir? Fehlt es mir
etwa an Verrückten, <img src="f.jpg" class="blume2">daß ihr diesen Mann hergebracht habt, damit
er gegen mich tobe?</span>
</section>
```

Ergibt als Output:



clear

Die Eigenschaft `clear` bestimmt, ob beim Platzieren der Platz auf der Seite frei sein muss. Clear wirkt nur bei Block-Level-Elemente.

Mögliche Werte:

`none`	es dürfen sich auf beiden Seiten fließende Elemente befinden
`left`	linke Seite muss frei sein
`right`	rechte Seite muss frei sein
`both`	beide Seiten müssen beim Positionieren frei sein.

Der grüne Block befindet sich im Normal Flow und positioniert sich links oben:

```
.green {
     background-color:  green;
}
.box { width: 150px;  height: 150px;}
<div class="box green"></div>
```

Bei `float:right` positioniert sich der grüne Block rechts:

```
.green {
         background-color:  green;
        float:right;
    }

.box { width: 150px;  height: 150px;}
```

Eine zweite (rote) nachfolgende Box positioniert sich links:

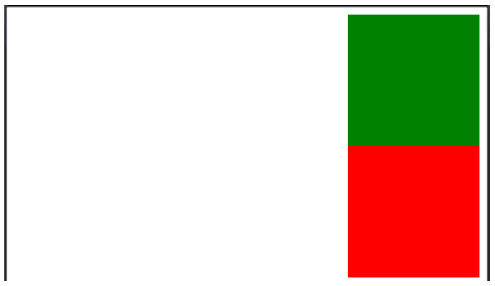
```
.green {
         background-color:  green;
         float:right;
    }

.red {
          background-color: red;
    }
.box { width: 150px;  height: 150px;}
<div class="box green"></div>
<div class="box red"></div>
```

Der roten Box wird auch `float:right` zugewiesen:

```
.green {
       background-color:  green;
       float:right;
 }

.red {
       background-color: red;
       float:right;
 }

.box { width: 150px;  height: 150px;}
<div class="box green"></div>
<div class="box red"></div>
```

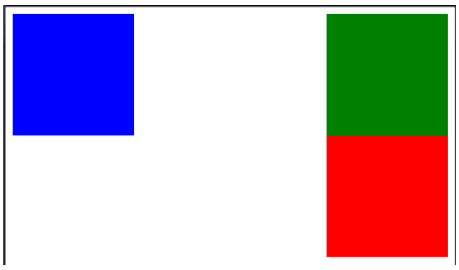
Der roten Box wird neben `float:right` auch `clear:right` zugewiesen. Beim Positionieren der roten Box muss der Platz rechts frei sein. Da sich rechts die grüne Box befindet, rutscht die rote Box runter.

```
.green {
    background-color:  green;
    float:right;
 }

.red {
    background-color: red;
    float:right;
    clear:right;
 }

.box { width: 150px;  height: 150px;}
<div class="box green"></div>
<div class="box red"></div>
```



Eine nachfolgende 3. Box (blau) befindet sich im Normal Flow und positioniert sich links oben.

```
.green {
    background-color:  green;
    float:right;
 }

.red {
     background-color: red;
     float:right;
     clear:right;
 }

.blue {
    background-color: blue;
 }

.box { width: 150px;  height: 150px;}

<div class="box green"></div>
<div class="box red"></div>
<div class="box blue"></div>
```



position

Die Eigenschaft `position` bestimmt die Art der Positionierung.

Mögliche Werte:

`absolute` **absolute Positionierung.**

Das Element wird anhand jenes Vorfahrenelementes (ancestor) positioniert, das nicht `static` ist.
Gibt es solch ein Element nicht, wird es anhand des Fensters positioniert.
Element ist außerhalb des Normal Flow.

`fixed` **fixierte Positionierung**

Positionierung relativ zum Fensterrand. Das Element bleibt auch beim Scrollen an dieser Stelle.

`relative` **relative Positionierung**

Das Element richtet sich an der Stelle aus, an der es bei `static` stehen würde.
Element ist innerhalb des Normal Flow (darum wird bei einer Verschiebung per `top/left/bottom/right` die frei gewordene Stelle nicht von anderen Elementen beansprucht).

`static` **statische Positionierung**

Keine besondere Positionierung (Grundeinstellung).
Element ist innerhalb des Normal Flow.

`inherit` (vererbte) Positionierung vom Vorfahren

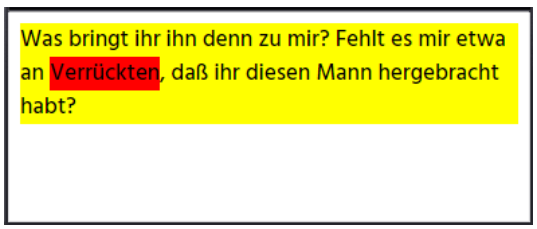
Zusätzlich sind die 4 Abstandsangaben möglich: `top, right, bottom, left`

Beispiel 1

Der Wert `static` ist der default für die Eigenschaft `position`. Zusätzliche Angaben wie `top` und `left` haben keine Wirkung.

```
<style>

.foo {
      position:static;
      background-color:red;
 }

section {background:yellow;}

</style>
<section>
Was bringt ihr ihn denn zu mir? Fehlt es mir etwa <br>an
<span class="foo">Verrückten</span>, daß ihr diesen Mann hergebracht habt?
</section>
```

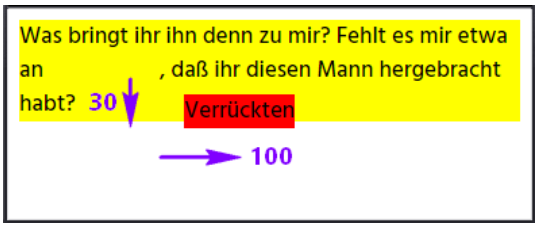
Ergibt als Output:



Die Eigenschaft wird auf `relative` geändert:

```
position:relative; top:30px; left:100px;
```

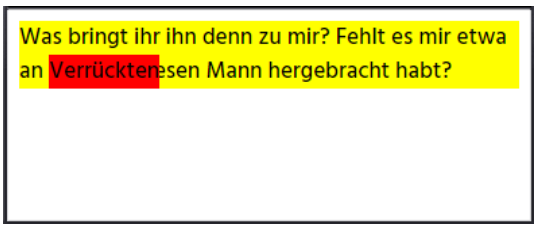
Die rote Box wird um 30px nach unten und um 100px nach rechts verschoben:



Die Eigenschaft wird auf `absolute` geändert:

```
position:absolute;
```

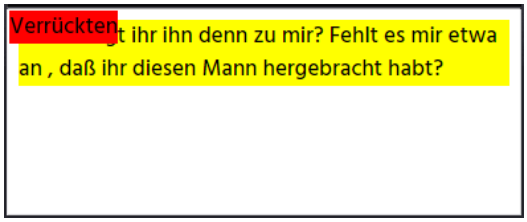
Damit wird die rote Box aus dem Flow genommen:



`Top` und `left` auf 1px:

```
position:absolute; top:1px; left:1px;
```

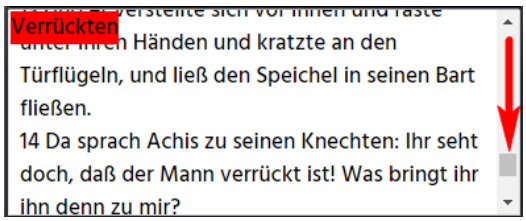
bewirkt eine Positionierung in die linke obere Ecke:



Eine Änderung auf `fixed` bewirkt eine Fixierung:

```
position:fixed; top:1px; left:1px;
```

Die rote Box bleibt auch beim Scrollen in der linken oberen Ecke:



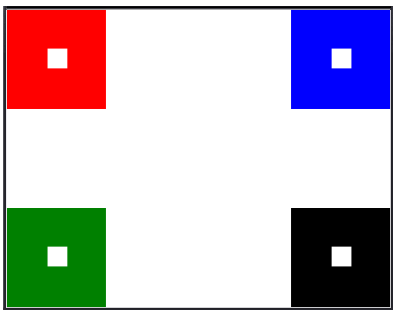
Beispiel 2

4 Boxen werden mit `absolute` in die jeweiligen Ecken gezeichnet.

Mit `absolute` wird in jede Box eine kleine weiße Box gezeichnet.

```
<style>

   .box { position: absolute; width: 100px; height: 100px; }

   .weiss { background: white; position: absolute; top: 39%; left: 41%; width: 20%; height: 20%; }

</style>

<div style="background: red; top: 0; left: 0;" class="box">
   <div class="weiss"></div>
</div>

<div  style="background: blue; top: 0; right: 0;" class="box">
   <div class="weiss"></div>
</div>

<div  style="background: green; bottom: 0; left: 0;" class="box">
   <div class="weiss"></div>
</div>

<div  style="background: black; bottom: 0; right: 0;" class="box">
   <div class="weiss"></div>
</div>
```

Ergibt als Output:



display

Die Eigenschaft `display` bestimmt, ob das Element als Block, Inline oder List-Item dargestellt worden soll.

`block`	Element als Block-Level-Element darstellen
`inline`	Element als Inline-Level-Element darstellen
`list-item`	Element als Listeneintrag darstellen
`none`	Element ausblenden

Ab CSS2:

`compact`	Darstellung als kompakte Box
`inline-table`	Darstellung als Inline-Tabelle
`marker`	Darstellung als Listensymbol
`table`	Darstellung als Tabelle
`run-in`	Darstellung als eingefahrene Box
`table-caption`	Darstellung als Tabellenüberschrift
`table-cell`	Darstellung als Tabellenzelle
`table-column`	Darstellung als Tabellenzeile
`table-column-group`	Darstellung als Zeilengruppe einer Tabelle
`table-footer-group`	Darstellung als Fußzeile(n) einer Tabelle
`table-header-group`	Darstellung als Kopfzeile(n) einer Tabelle
`table-row`	Darstellung als Tabellenspalte
`table-row-group`	Darstellung als Spaltengruppe einer Tabelle

Pseudoklassen

Eine Pseudoklasse bezieht sich auf ein vorhandenes Element, das auf eine bestimmte gegenwärtige Eigenschaft abgefragt wird. Also auf eine Struktur, die in einem besonderen Zustand ist.
Pseudoelemente sind Fake Elemente, die nicht im DOM enthalten sind. Siehe auch Mozilla.org.

Interaktive Pseudoklassen

`:link`	selektiere Links (<a>), die noch nicht besucht worden sind	Seite 189
`:visited`	selektiere Links (<a>), die bereits besucht worden sind	Seite 189
`:active`	Element, das gerade angeklickt wird	Seite 190
`:target`	wenn ein Anker aufgerufen worden ist, diesen selektieren	Seite 189
`:checked`	Elemente, die angehakt worden sind (entweder per default oder vom User)	Seite 191
`:disabled`	deaktivierte (disabled) Formular-Elemente	Seite 191
`:enabled`	aktivierte (enabled) Formular-Elemente	Seite 192
`:indeterminate`	Formular-Elemente mit unbestimmten Zustand selektiert	Seite 192
`:focus`	das gerade fokusierte Elemente (meist Formularfelder)	Seite 192
`:hover`	Element, über dem sich gerade der Zeiger befindet	Seite 193
`:in-range`	input Elemente, dessen value-Werte innerhalb des min-Bereiches liegen	Seite 193
`:out-of-range`	input Elemente, dessen value-Werte ausserhalb des min-Bereiches liegen	Seite 193
`:valid`	Formularfelder, deren Eingaben als valid (gültig) eingestuft werden	Seite 194
`:invalid`	Formularfelder, deren Eingaben als invalid (ungültig) eingestuft werden	Seite 194
`:required`	Formular-Elemente, die das Attribut `required` besitzen	Seite 195
`:read-only`	Formular-Elemente, die das Attribut readonly besitzen	Seite 195
`:read-write`	beschreibbare Elemente, die nicht das readonly-Attribut besitzen	Seite 195

Struktur-Pseudoklassen

`:first-child`	selektiert alle Erstgeborene (dann auf gewünschtes Element prüfen)	Seite 196
`:last-child`	selektiert alle Letztgeborenen (dann auf gewünschtes Element prüfen)	Seite 196
`:first-of-type`	alle Kinder mit dem gewünschten Elementtyp und dann den Erstgeborenen	Seite 197
`:last-of-type`	alle Kinder mit dem gewünschten Elementtyp und dann den Letztgeborenen	Seite 197
`:only-child`	alle Elemente, die Einzelkinder sind	Seite 198
`:nth-child`	alle Kinder	Seite 198
`:nth-of-type`	selektiert alle Geschwister mit dem gewünschtem Element	Seite 200
`:root`	Root-Element	Seite 201
`:empty`	alle Elemente ohne Kinder (Whitespace gilt als Füllung)	Seite 202

Negation-Pseudoklasse

`:not(`*foo*`)`	alle Elemente, bis auf das angegebe Element *foo*	Seite 202

Language-Pseudoklasse

:lang (foo)	alle Elemente mit dem Attribut `lang` und dem Attributwert `foo`	Seite 202

Interaktive Pseudoklassen

Mit **Interaction Pseudo-Classes** können Elemente abgefragt werden, die durch Interaktion vom User verändert werden können. Die Eigenschaften der Elemente ändern sich somit dynamisch (**Dynamic Pseudo-Classes).**

Empfohlene Reihenfolge der Zuweisung:

:link :visited :hover :focus :active

:link

Mit der CSS1 Pseudoklasse `:link` werden Links (<a>) selektiert, die noch nicht besucht worden sind.
Beispiel:

```
a:link {color: red}                    <!—- färbe a rot, wenn der link nicht aufgerufen worden ist -->
```

:visited

Mit der CSS1 Pseudoklasse `:visited` werden Links (<a>) selektiert, die bereits besucht worden sind.
Beispiel:

```
a:visited {color: black}               /* färbe a schwarz, wenn der link aufgerufen worden ist */
```

:any-link

Mit der Pseudoklasse `:any-link` werden alle Links selektiert. Keine Browserunterstützung.
Beispiel:

```
a:any-link {color: black}              /* färbe a immer schwarz */
```

:target

Mit der Pseudoklasse `:target` (ab CSS3) wird überprüft, ob ein Anker verwendet (aufgerufen) worden ist.
Falls ja, wird das Element, in dem sich die Anker-ID befindet, von der Pseudoklasse selektiert.

```
<style type="text/css">

/*  färbe rot, wenn ein anker aufgerufen worden ist, und der Anker in einem p Element liegt */

   p:target { background-color:#990000;   }


/* färbe schwarz, wenn ein anker aufgerufen worden ist
   (egal in welchem Zielelement sich der Anker befindet) */

   *:target {  background-color:#009900;  }


   p { background-color:#000099; color:#ffffff;}

</style>

<a href="#foo1" />blub1</a>                        /* link auf foo1 */

<a href="#foo2" />blub2</a>                        /* link auf foo2 */

<a href="#foo3" />blub3</a>                        /* link auf foo3 */


<p id="foo1">blub1</p>                             /* anker foo1 */

<p id="foo2">blub2</p>                             /* anker foo2 */

<h1 id="foo3">blub3</h1>                           /* anker foo3 */
```

Link blub2 mit Anker-ID foo2 wurde geklickt:

:active

Mit der CSS1 Pseudoklasse `:active` werden Links (<a>) selektiert, die gerade aktiv sind (gerade geklickt werden). Ab CSS2 können alle aktiven Elemente selektiert werden.

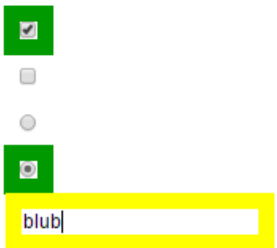
Beispiel:

```
a:active {color: red}                /* färbe rot, wenn der link gerade gedrückt wird */

*:active {color: red}                /* färbe rot, wenn ein Element gerade aktiv ist */
```

Dieses Beispiel:

```
<style type="text/css">
 input:checked { outline: 12px solid #009900;}
 input:active { outline: 12px solid #ffff00;}
 input {margin:20px;display:block;}
</style>

<input type="checkbox" value="4" checked >
<input type="checkbox" value="3">
<input type="radio" name="a" value="1">
<input type="radio" name="a" value="2"  checked>
<input type="input" >
```

Ergibt diesen Output, wenn gerade `<input type="text">` gedrückt wird:



:checked

Mit der Pseudoklasse `:checked` werden Elemente selektiert, die angehakt worden sind (entweder per default oder vom User). Mögliches Element ist `<input>` mit den Attributen `radio` und `checkbox`.

Beispiel:

```
<style type="text/css">
   input:checked { outline: 12px solid #0000ff}
   input[type="checkbox"]:checked {outline: 12px solid #ff04ff}
   input {margin:20px;display:block;}
</style>
<input type="checkbox" value="4" checked >                    <!-- checkbox checked-->
<input type="checkbox" value="3">
<input type="radio" name="a" value="1">
<input type="radio" name="a" value="2" checked>               <!-- radio checked -->
```

Ergibt diesen Output:

:disabled

Mit der Pseudoklasse `:disabled` werden deaktivierte (disabled) Formular-Elemente selektiert.

```
<style type="text/css">
  input:disabled { outline: 12px solid #ffff00;}
  input {margin:20px;display:block;}
</style>
<input type="input" value="blub" disabled >                   <!-- deaktiviert -->
```

Ergibt diesen Output:

:enabled

Mit der Pseudoklasse `:enabled` werden aktivierte (enabled) Formular-Elemente selektiert.

```
<style type="text/css">

    input:disabled { outline: 12px solid #ffff00;}

    input:enabled { outline: 12px solid #00ff00;}

    input {margin:20px;display:block;}

</style>

<input type="input" value="blub1" disabled >

<input type="input" value="blub2" >
```

Ergibt als Output (1. Feld disabled):

:indeterminate

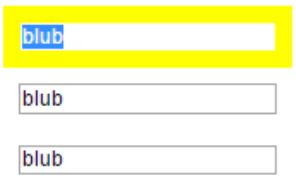
Mit der Pseudoklasse `:indeterminate` werden Formular-Elemente mit unbestimmten Zustand selektiert.

:focus

Mit der CSS2 Pseudoklasse `:focus` wird das gerade fokusierte Element (meist ein Formularfeld) selektiert.

```
<style type="text/css">

    :focus { outline: 12px solid #ffff00;}

    input {margin:20px;display:block;}

</style>


<input type="input" value="blub" >

<input type="input" value="blub" >

<input type="input" value="blub"  >
```

Ergibt diesen Output (1. Formularfeld wurde fokusiert):



:hover

Mit der CSS2 Pseudoklasse :hover wird das Element fokusiert, über dem sich gerade der Zeiger befindet.

```
<style type="text/css">

    body >*:hover { outline: 12px solid #ffff00;}

    input {margin:20px;display:block;}

</style>

<input type="input" value="blub1" >

<input type="input" value="blub2" >
```

Ergibt diesen Output (Mauszeiger befindet sich über das 2. Eingabefeld):

:in-range

Mit den beiden Pseudoklassen :inrange und :out-of-range werden <input> Elemente überprüft, die als Attribut/Attributwert type="number" und die Attribute min und max besitzen.

:in-range selektiert die input Elemente, dessen value-Werte innerhalb des erlaubten min-max-Bereiches liegen.
:out-of-range selektiert die input Elemente, dessen value-Werte außerhalb des erlaubten min-max-Bereiches liegen.

```
<style>

input:in-range {  background-color: green;    }

input:out-of-range {
                        background-color: red;
                        border: 2px solid red;
    }

input { color:white;}

</style>

<input id="foo1" type="number" placeholder="1 bis 5" min="1" max="5" value="3">

<input id="foo2" type="number" placeholder="6 bis 10" min="6" max="10" value="0">
```

Ergibt diesen Output:

:out-of-range

Siehe Pseudoklasse :inrange Seite 193.

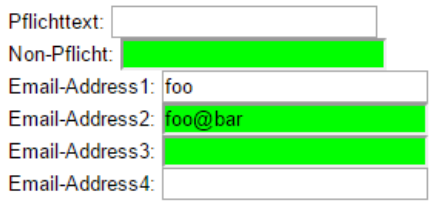
:valid

Die Pseudoklasse `:valid` selektiert Formularfelder, deren Eingaben vom Browser als valid (gültig) eingestuft werden.

Beispiel:

```
<style type="text/css">
input:valid {
  background-color: lime;
}
</style>

Pflichttext: <input type="text"  required> <br>
Non-Pflicht: <input type="text"> <br>
Email-Address1: <input type="email" value="foo"> <br>
Email-Address2: <input type="email" value="foo@bar"> <br>
Email-Address3: <input type="email" value=""> <br>
Email-Address4: <input type="email" value="" required> <br>
```

Ergibt als Output:



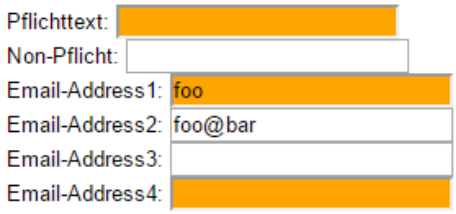
:invalid

Die Pseudoklasse `:invalid` selektiert Formularfelder, deren Eingaben vom Browser als invalid (ungültig) eingestuft werden.

Beispiel:

```
<style type="text/css">
input:invalid {
  background-color: orange;
}
</style>

Pflichttext: <input type="text"  required> <br>
Non-Pflicht: <input type="text"> <br>
Email-Address1: <input type="email" value="foo"> <br>
Email-Address2: <input type="email" value="foo@bar"> <br>
Email-Address3: <input type="email" value=""> <br>
Email-Address4: <input type="email" value="" required> <br>
```

Ergibt als Output:



:required

Die Pseudoklasse `:valid` selektiert Formular-Elemente, die das Attribut `required` besitzen.

```
<style type="text/css">
    input:required {  background-color: yellow;  }
</style>

Pflichttext: <input type="text"  required> <br>

Text: <input type="text">
```

Ergibt als Output:

Pflichttext:
Text:

:read-only

Die Pseudoklasse `:read-only` selektiert Formular-Elemente, die das Attribut readonly besitzen.

```
<style type="text/css">
    input:read-only {   background-color: yellow;   }
</style>

Text: <input type="text" value="blub1"> <br>
Readonly-Text: <input type="text" readonly value="blub2"> <br>
Disabled-Text: <input type="text" disabled value="blub3"> <br>
```

Ergibt als Output:

Text: blub1
Readonly-Text: blub2
Disabled-Text: blub3

:read-write

Die Pseudoklasse `:read-write` selektiert beschreibbare Elemente, die nicht das readonly-Attribut besitzen.

Disabled Formularfelder werden als beschreibbar betrachtet.

```
<style type="text/css">
 input:read-write {  background-color: yellow;  }
</style>

Text: <input type="text" value="blub1"> <br>

Readonly-Text: <input type="text" readonly value="blub2"> <br>

Disabled-Text: <input type="text" disabled value="blub3"> <br>
```

Ergibt als Output:

Text: blub1
Readonly-Text: blub2
Disabled-Text: blub3

Struktur-Pseudoklassen

Mit Struktur-Pseudoklassen lassen sich DOM-Eigenschaften selektieren.

:first-child

Die Pseudoklasse selektiert alle Erstgeborene (und überprüfe dann, ob vom gewünschte Element).

Färbe alle Erstgeborene vom Element b:

```
<style type="text/css">

    b:first-child {background-color:red;  text-decoration: underline;}

</style>

<p>
    <b>abc1</b>                     <!-- erstes kind und ein b element -->
    <b>def1</b>
</p>

<p>
    <i>abc2</i>                     <!-- erstes kind, aber kein b element -->
    <b>def2</b>
</p>
```

Ergibt als Output:

abc1 def1
abc2 def2

:last-child

Die Pseudoklasse selektiert alle Letztgeborenen (und überprüfe dann, ob vom gewünschte Element).

Färbe alle Letztgeborene vom Element b:

```
<style type="text/css">

    b:last-child {background-color:red;  text-decoration: underline overline;}

</style>

<p>
    <b>abc1</b>
    <b>def1</b>         <!-- letztgeborenes und ein b element -->
</p>

<p>
    <b>def2</b>
    <i>abc2</i>         <!-- letztgeborenes, aber kein b element -->
</p>
```

Ergibt als Output:

abc1 **def1**
def2 *abc2*

:first-of-type

Selektiert alle Kinder mit dem gewünschten Elementtyp und nimmt dann den Erstgeborenen.

```
<style type="text/css">

    b:first-of-type {background-color:red;  text-decoration: underline overline;}

</style>

<p>
    <i>111</i>
    <b>222</b> <!-- erstes b element -->
    <b>333</b>
    <i>444</i>
</p>
```

Ergibt als Output:

111 222 333 *444*

:last-of-type

Selektiert alle Kinder mit dem gewünschten Elementtyp und nimmt dann den Letztgeborenen.

```
<style type="text/css">

    b:last-of-type {background-color:red;  text-decoration: underline overline;}

</style>

<p>
    <i>111</i>
    <b>222</b>
    <b>333</b>          <!-- letztes b element -->
    <i>444</i>
</p>
```

Ergibt als Output:

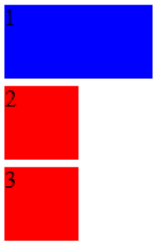
111 222 333 *444*

:only-child

Selektiere alle Elemente, die Einzelkinder sind.

```
<style>
      div   {
               background: red;
               height: 50px; width: 50px; margin:5px;
            }

      :only-child {
               background: blue;
               width:100px;
            }
</style>

<section>
      <div>1</div>                              <!-- Einzelkind -->
</section>

<section>
      <div>2</div><div>3</div>
</section>
```

Ergibt als Output:



:nth-child (Kinder)

Die **nth-child**-Pseudoklasse selektiert alle Kinder und wendet danach optional die Formatierung an (optional, da es nicht der gewünschte Elementtyp sein muß).

Die **nth-of_type**-Pseudoklasse (Seite 200) selektiert alle Geschwister mit dem gewünschtem Element.

Für Unterschied zwischen :nth-of-type und :nth-child siehe Seite 201.

Die Pseudoklasse besitzt 4 Syntaxmöglichkeiten (Parameter-Varianten):

1)	nth-child (*a* n+*b*)	jedes a. Element beginnend beim b. Element
2)	nth-child (-*a*n+*b*)	jedes a. Element vom ersten Element bis zum b. Element
3)	nth-child(*keyword*)	Keyword ist *odd* (gerade Indexwert) oder *even* (ungerader Indexwert).
4)	nth-child(*n*)	n. Element

Varianten 5 und 6 sind kombinierte `:nth-cild` Anweisungen.

Indexzähler beginnt bei 1.

Variante 1 – Ab dem b. Element

Selektiere Elemente, die (an+b) Geschwister haben:

- a jedes a. Elemente
- b beginnend beim b. Element (diese Startposition wird ebenfalls selektiert)

Beispiel:

```
<section>
<b> 1 </b><b> 2 </b><b> 3 </b><b> 4 </b><b> 5 </b><b> 6 </b><b> 7 </b>
<b> 8 </b><b> 9 </b><b> A </b><b> B </b><b> C </b><b> D </b><b> E </b><b> F </b>
</section>
```

Färbe jedes Element gelb, beginnend beim 3. Element (dieses 3. Element wird auch gelb):

```
b:nth-child(1n+3) {background-color:#ffff00;}
```

1 2 3 4 5 6 7 8 9 A B C D E F

Der 1 vor dem n kann bei "jedes Element" auch weggelassen werden:

```
b:nth-child(n+3) {background-color:#ffff00;}
```

Färbe jedes 3. Element gelb, beginnend beim 2. Element (dieses 2. wird ebenfalls selektiert und gelb):

```
b:nth-child(3n+2) {background-color:#ffff00;}
```

1 2 3 4 5 6 7 8 9 A B C D E F

Variante 2 – Ab dem 1. Element

Selektiere Elemente, die (-an+b) Geschwister haben:

- a jedes a. Elemente
- b vom ersten Element bis zum b. Element (inklusive b. Element)

Beispiel:

```
<section>
<b> 1 </b><b> 2 </b><b> 3 </b><b> 4 </b><b> 5 </b><b> 6 </b><b> 7 </b>
<b> 8 </b><b> 9 </b><b> A </b><b> B </b><b> C </b><b> D </b><b> E </b><b> F </b>
</section>
```

Färbe alle Elemente bis zum 5. Element (fünfte Element ist inklusive):

```
b:nth-child(-1n+5) {background-color:#ffff00;}
```

1 2 3 4 5 6 7 8 9 A B C D E F

Färbe jedes 2. Element. Vom ersten bis zum 8. Elemente (achte Element inklusive):

```
b:nth-child(-2n+8) {background-color:#ffff00;}
```

1 2 3 4 5 6 7 8 9 A B C D E F

Variante 3 – Odd/Even

Selektiere Elemente, die einen ungeraden Zähler (1, 3, 5, ...) haben:

```
li:nth-child(odd) {background-color: green; }
```

Selektiere Elemente, die einen geraden Zähler (2, 4, 6, ...) haben:

```
li:nth-child(even) {background-color: green; }
```

Variante 4 – Einzelzugriff

Auf jedes Kind-Element einzeln zugreifen.

Das erste li Element grün, das fünfte li Element rot färben:

```
li:nth-child(1) {background-color: green; }
li:nth-child(5) { background-color: red; }
```

Variante 5 – Generic Child Range

Auf einen von-bis-Bereich zugreifen.

Selektiere vom 4. bis zum 8. Element:

```
b:nth-child(n+4):nth-child(-n+8) {background-color:#ffff00;}
```

1 2 3 4 5 6 7 8 9 A B C D E F

Variante 6 – Advanced Generic Child Range

In einem von-bis-Bereich nur auf die odd/even zugreifen.

Selektiere vom 2. bis zum 9. Element:

```
nth-child(n+2):nth-child(odd):nth-child(-n+9)
```

1 2 3 4 5 6 7 8 9 A B C D E F

:nth-of-type (Geschwister)

Syntax und Parameter entsprechen :nth-child (siehe Seite 160).

Die **nth-of_type-**Pseudoklasse selektiert alle Geschwister mit dem gewünschtem Element.

Die **nth-child-**Pseudoklasse (siehe Seite 160) selektiert alle Kinder und wendet danach optional die Formatierung an (optional, da es nicht der gewünschte Elementtyp sein muß).

Unterschied zwischen :nth-of-type und :nth-child siehe Seite 201.

:nth-of-type versus :nth-child

Die nth-child-Pseudoklasse selektiert alle Kinder wendet danach optional die Formatierung an (optional, da es nicht der gewünschte Elementtyp sein muß).

Die nth-of_type-Pseudoklasse selektiert alle Kinder mit dem gewünschtem Element.

Beispiel:

```
<style>
    b:nth-child(2) {background-color:red;}
    b:nth-of-type(2) {background-color:yellow;}
</style>

<section>
  <blub>blubber</blub><b> 1 </b><b> 2 </b><b> 3 </b><b> 4 </b><b> 5 </b><b> 6 </b><b> 7 </b>
  <b> 8 </b><b> 9 </b><b> A </b><b> B </b><b> C </b><b> D </b><b> E </b><b> F </b>
<section>
```

Ergibt als Output:

blubber 1 2 3 4 5 6 7 8 9 A B C D E F

Ein zweites <blub>blubber</blub> wird vorne hinzugefügt:

```
<style>
   b:nth-child(2) {background-color:red;}
   b:nth-of-type(2) {background-color:yellow;}
</style>

<section>
   <blub>blubber</blub><blub>blubber</blub><b> 1 </b><b> 2 </b><b> 3 </b><b> 4 </b>
   <b> 5 </b><b> 6 </b><b> 7 </b>
   <b> 8 </b><b> 9 </b><b> A </b><b> B </b><b> C </b><b> D </b><b> E </b><b> F </b>
<section>
```

Ergibt als Output:

blubberblubber 1 2 3 4 5 6 7 8 9 A B C D E F

:root

Selektiert das Wurzel-Element. Sollte in einem HTML-File das html Element sein.

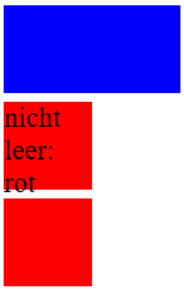
:empty

Die Pseudoklasse `:empty` selektiert alle Elemente ohne Kinder (Whitespace gilt als Füllung).

Färbe Elemente blau, wenn sie keine Kinder enthalten:

```
<style>
 .box {
    background: red;
    height: 50px; width: 50px; margin:20px;
 }

 .box:empty {
    background: blue;
    width:100px;
 }
</style>
<div class="box"><!-- leer: blau--></div>
<div class="box">nicht leer: rot</div>
<div class="box"> <!-- nicht leer, da leerzeichen vorhanden: rot --> </div>
```

Egibt als Output:



Negation-Pseudoklasse

Mit der Negationspseudoklasse `:not(`*foo*`)` werden alle Elemente bis auf das angegebe Element (*foo*) selektiert.

Als Beispiel alle Elemente bis auf b rot färben:

```
<style type="text/css">
 body > *:not(b) {color:#e600e6; background-color:red;}
</style>
<i>abc</i> <b>def</b> <u>ghi</u>
```

Ergibt diesen Output:

abc def ghi

Language-Pseudoklasse

Mit der Sprachcode-Pseudoklasse `lang(`*foo*`)` werden alle Elemente mit dem Attribut `lang` und dem Attributwert `foo` selektiert.

Alle Elemente mit dem Attribut `lang` und dem Attributswert `de` (inklusive de-at, de-ch, de) rotfärben:

```
<style type="text/css">
 body > *:lang(de) {color:#e600e6; background-color:red;}
</style>
<i lang="de-at">abc</i> <b>def</b> <u>ghi</u>
```

Ergibt diesen Output:

abc def ghi

Siehe auch Attribut-Selektor Seite 168.

Pseudoelemente

Pseudoelemente (Pseudo Elements) sind Fake-Elemente, die nicht im DOM enthalten sind.
Eine Pseudoklasse bezieht sich auf ein vorhandenes Element, das auf eine bestimmte gegenwärtige Eigenschaft abgefragt wird.

`::before`	zusätzlicher Text am Beginn des Contenttextes hinzufügen	Seite 203
`::after`	zusätzlicher Text am Ende des Contenttextes hinzufügen	Seite 203
`::first-letter`	erstes Zeichen eines Textcontents in einem Block-Level-Element	Seite 204
`::first-line`	erste (optische) Textzeile eines Block-Level-Elementes	Seite 204
`::selection`	Farben des vom User markierten Textes definieren	Seite 203

Pseudoelemente müssen ab CSS3 mit 2 Doppelpunkten beginnen (davor war die Notation nur mit einem Doppelpunkt).
Siehe auch Mozilla.org.

::selection

Zur Bestimmung der Schrift- und Hintergrundfarbe (color, background-color) für eine vom User markierte Textstelle (zB per Maus selektiert).
Rote Schrift auf schwarzem Hintergrund:

```
p.normal::selection {
 background:black;
 color:red;
}
```

Für Mozilla muss `-moz-selection` verwendet werden.

::before

Das Pseudoelement `::after` erweitert mit der Eigenschaft `content` den Textcontent beim angegebenen Element. Dieser zusätzliche Text wird an den Anfang des Contents hinzugefügt und kann mit CSS formatiert werden. Darin enthaltene html-Tags werden nicht interpretiert.

```
<style type="text/css">
b::before{content:'B';background-color:lime;}
b::after{content:'D';background-color:red;}
b:hover::after{content:'hover';background-color:yellow;}
</style>
A<b>C</b>E
```

Erzeugt als Output: ABCDE

::after

Das Pseudoelement `::after` erweitert mit der Eigenschaft `content` den Textcontent beim angegebenen Element. Dieser zusätzliche Text wird an das Ende des Contents hinzugefügt und kann mit CSS formatiert werden. Darin enthaltene html-Tags werden nicht interpretiert.

```
<style type="text/css">
b::before{content:'B';background-color:lime;}
b::after{content:'D';background-color:red;}
b:hover::after{content:'hover';background-color:yellow;}
</style>
A<b>C</b>E
```

Erzeugt als Output: ABCDE

::first-letter

Das Pseudoelement `::first-letter` bezieht in einem Block-Level-Element auf das erste Zeichen vom Textcontent.

Ist das erste Zeichen ein Anführungszeichen (oder Stern/Raute) wird dieses und der erste Buchstabe selektiert.

```
<style type="text/css">
p::first-letter {background-color:yellow;text-decoration:underline; }
h6::before{content:'x';}
h6::first-letter {background-color:yellow}
</style>

<p>blub</p>
<p>-blub</p>
<p>_blub</p>
<p>"blub</p>
<p>""blub</p>
<p>'blub</p>
<p>*blub</p>
<p>#blub</p>
<h6>blub</h6>
```

Ergibt als Output:

blub
-blub
_blub
"blub
""blub
'blub
*blub
#blub
xblub

::first-line

Das Pseudoelement `::first-line` bezieht sich auf die (optisch) erste Textzeile einer Box.

Beispiel:

```
<style type="text/css">
p {width:90px;border:solid;}
p::first-line {background-color:yellow;text-decoration:underline;}
</style>

<p>Fehlt es mir etwa an Verrückten, daß ihr diesen Mann hergebracht habt, damit er gegen mich
tobe? Sollte der in mein Haus kommen?</p>
```

Ergibt als Output:

Fehlt es mir etwa an Verrückten, daß ihr diesen Mann hergebracht habt, damit er gegen mich tobe? Sollte der in mein Haus kommen?

Shorthand Properties

Mit Shorthand Properties können mehrere Deklarationspaare in einem einzigen Paar untergebracht werden.

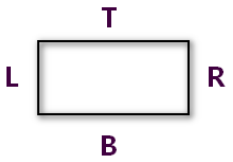
Sub-Properties top, right, bottom, left

Als Beispiel vier Deklarationspaare für padding (die 4 Sub-Properties *top, right, bottom left*):

```
p {padding-top: 1.5em;}
p {padding-right: .6em;}
p {padding-bottom: 1.7em;}
p {padding-left: 3em;}
```

4 Werte Shorthand

Als Shorthand fallen die Sub-Properties Bezeichnungen weg, und die Values werden mit dem Main-Property *padding* zu einem einzigen Deklarationspaar zusammengefasst:



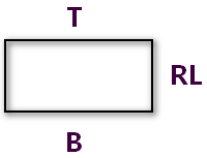

```
p {padding: 1.5em .6em 1.7em 3em;}
```

Die Reihenfolge der Angaben ist Top. Right, Bottom, Left.

Eselsbrücke: *Starting at 12 o'clock will keep you out of T R o u B L e*

3 Werte Shorthand

Bei 3 Werten ist die Reihenfolge T RL B:

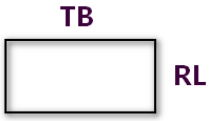



```
p {padding: 1.5em .6em 3em;}
```

R=L.

2 Werte Shorthand

Bei 2 Werten ist die Reihenfolge TB RL:




```
p {padding: 1.5em .6em;}
```

T=B. R=L.

1 Wert Shorthand

Bei nur einem Wert ist T=B=R=L:

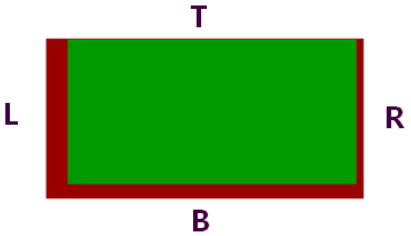
```
p {padding: 1.5em}
```

Beispiel border-width

```
border-top-width:1px;
border-right-width:10px;
border-bottom-width:20px;
border-left-width:30px;
```

4 Werte Shorthand

Aus dem Beispiel mit 4 Paaren, wird ein einziges Deklarationspaar:




```
border-width: 1px 10px 20px 30px;
```

Die Reihenfolge der Angaben ist TRBL: Top, Right, Bottom, Left.

Property border

Die Properties *border-width, border-style* und *border-color* als 3 einzelne Deklarationspaare:

```
border-width:     1px;          Width
border-style:     solid;        Style
border-color:     #000;         Color
```

Als Shorthand fallen die Sub-Properties Bezeichnungen weg, und die Values werden mit dem Main-Property *border* zu einem einzigen Deklarationspaar zusammengefasst:

```
border:1px solid #000;
```

Die Reihenfolge spielt keine Rolle, da der Browser anhand den Values die dazugehörigen Properties erkennt.
Die offizielle W3C-Reihenfolge ist WSC: Width Style Color

Property background

Fünf Sub-Properties als 5 einzelne Deklarationspaare:

```
background-color:     #f00;          Color
background-image:     url(b.gif);    Image
background-repeat:      no-repeat;   Repeat
background-attachment: fixed;        Attachment
background-position:    0 0;         Position
```

Als Shorthand fallen die Sub-Properties Bezeichnungen weg, und die Values werden mit dem Main-Property *background* zu einem einzigen Deklarationspaar zusammengefasst:

```
background: #f00 url(b.gif) no-repeat fixed 0 0;
```

Die offizielle W3C-Reihenfolge ist CIRAP.

Property font

Sechs Sub-Properties als 6 einzelne Deklarationspaare:

```
font-style:  italic;                    Style
font-variant:  small-caps;              Variant
font-weight: bold;                      Weight
font-size:   1em;                       Size
line-height: 140%;                      Height
font-family: "Lucida Grande",sans-serif; Family
```

Als Shorthand fallen die Sub-Properties Bezeichnungen weg, und die Values werden mit dem Main-Property *font* zu einem einzigen Deklarationspaar zusammengefasst:

```
font: italic  small-caps  bold  1em/140%  "Lucida Grande",sans-serif;
```

Die offizielle W3C-Reihenfolge ist SVWSHF.

Property list-style

Drei Sub-Properties als 3 einzelne Deklarationspaare:

```
list-style-type:     square;          Type
list-style-position:inside;           Position
list-style-image:    url(foo.gif);    Image
```

Als Shorthand fallen die Sub-Properties Bezeichnungen weg, und die Values werden mit dem Main-Property *list-style* zu einem einzigen Deklarationspaar zusammengefasst:

```
list-style:  square inside url(foo.gif);
```

Property outline

Drei Sub-Properties als 3 einzelne Deklarationspaare:

```
outline-color: #f00;          Color
outline-style: solid;         Style
outline-width: 2px;           Width
```

Als Shorthand fallen die Sub-Properties Bezeichnungen weg, und die Values werden mit dem Main-Property *outline* zu einem einzigen Deklarationspaar zusammengefasst:

```
outline: #f00 solid 2px;
```

Property transition

Vier Sub-Properties als 4 einzelne Deklarationspaare:

```
transition-property:        opacity;          Color
transition-duration:        3s;               Style
transition-delay:           ease-in-out;      Width
transition-timing-function:   1s;             Width
```

Als Shorthand fallen die Sub-Properties Bezeichnungen weg, und die Values werden mit dem Main-Property *transition* zu einem einzigen Deklarationspaar zusammengefasst:

```
transition: opacity 3s ease-in-out 1s
```

Vendor Prefix

Der Browser-Prefix erlaubt eine indivuelle Browserformatierung einer Eigenschaft.

-khtml-	Konqueror browser
-ms-	Microsoft
-moz-	Mozilla (Gecko browsers)
-o-	Opera
-webkit-	Safari und Chrome

Als Beispiel die Eigenschaft box-shadow:

```
-webkit-box-shadow: 10px 10px 5px black;
-moz-box-shadow: 10px 10px 5px black;
-ms-box-shadow: 10px 10px 5px black;
-o-box-shadow: 10px 10px 5px black;
box-shadow: 10px 10px 5px black;
```

Längeneinheit (Unit)

Mit der Längenangabe werden Abstand, Größe (Breite, Höhe) und Liniendicke bechrieben.
Es gibt zwei Arten der Längenangabe: die relative und die absolute Länge (bzw Größe).

absolute Größeneinheiten

Die absoluten Größeneinheiten sollten dem Druck vorbehalten sein (1cm Schriftgröße auf Kinoleinwand = 1cm auf Handy):

in Inch (1 inch = 1 Zoll = 2.54cm)

pt Point (Punkt)

mm

pc Pica. Typografische Maßeinheit. 1 Pica entspricht 12 Punkt.

px Pixel (abhängig von der Pixeldichte des Ausgabegeräts, relativ also von Ausgabegerät zu Ausgabegerät, absolut dagegen auf ein und das selbe Ausgabegerät bezogen)

relative Größeneinheiten

Die relativen Größeneinheiten für den Screen vorbehalten sein:

em bezieht sich auf die Schriftgröße des Elements

Ausnahme: wenn die font-size-Eigenschaft (also die Schriftgröße selbst) mit dieser Maßangabe versehen wird, bezieht sie sich auf die Schriftgröße des Elternelements.

px relativ von Ausgabegerät zu Ausgabegerät

% je nach CSS-Eigenschaft relativ zur elementeigenen Größe, oder zu der des Elternelements, oder zu einem allgemeineren Kontext.

ch relativ zur Breite von 0 (Ziffer Null)

rem relativ zur Schriftgröße vom root element

vw relativ zu 1% der Fensterbreite (Viewport width)

vh relativ zu 1% der Fensterhöhe (Viewport heigth)

ex bezogen auf die Höhe des Kleinbuchstaben x in diesem Element

Ausnahme: wenn die font-size-Eigenschaft (also die Schriftgröße selbst) mit dieser Maßangabe versehen wird, bezieht sie sich auf die Höhe des Kleinbuchstaben x im Elternelement.

W3: In the past, CSS required that implementations display absolute units correctly even on computer screens. But as the number of incorrect implementations outnumbered correct ones and the situation didn't seem to improve, CSS abandoned that requirement in 2011. Currently, absolute units must work correctly only on printed output and on high-resolution devices. CSS doesn't define what high resolution means.

w3schools: Pixels (px) are relative to the viewing device. For low-dpi devices, 1px is one device pixel (dot) of the display. For printers and high resolution screens 1px implies multiple device pixels.

em

1em entspricht der gegenwärtigen Schriftgröße. 2em verdoppelt die Schriftgröße.

```
text-indent: 1.5em;
margin: 1em;
font-size: 0.9em
```

Viewport Unit

Der Viewport ist das Browserfenster. Wird das Fenster verkleinert, verringern sich entsprechend die Viewport-Größenangaben.

vw	1/100 der viewport width
vh	1/100 der viewport height
vmin	1/100 des Viewport kleineren Seite
vmax	1/100 des Viewport größeren Seite

Html & CSS

Es gibt 3 Arten CSS zu inkludieren:

Inline Style

Mit dem Attribut `style` direkt in einem Html-Element.

Diese Methode hat die höchste Spezifitätsgewichtung (siehe Seite 159):

```
<p style="color: red;">Another paragraph</p>
```

Inline Style sollte nur dann verwendet werden, wenn der Author keinen Zugriff auf die CSS-Rule-Sets besitzt.

Siehe auch **Inline Javascript** auf Seite 432.

HTML, CSS und JavaScript sollten getrennt sein und im Idealfall in seperaten Files liegen. Dies erleichtert die Trennung von Inhalt (HTML), Optik (CSS) und Interaktion (JavaScript). Siehe **Unobtrusive JavaScript** auf Seite 430.

<link>

Ein externes CSS-File mit Element <link> inkludieren (für jedes externe File wird ein eigener http-Request benötigt):

```
<link href="styles.css" rel="style sheet"
```

@import

Ein externes CSS-File per @import inkludieren. (innerhalb von einem <style> Element).

Das style Element kann sich in einem CSS- oder Html-File befinden.

```
<style> @import url('styles.css'); </style>
```

Ab CSS2 muss nicht mehr die url-Funktion verwendet werden:

```
<style> @import 'styles.css' </style>
```

Ab CSS2 kann ein Mediatyp (siehe Seite 216) hinzugefügt werden (default ist `all`):

```
@import url("pda.css") handheld;
@import "speak.css" aural;
@import "print.css" print;
```

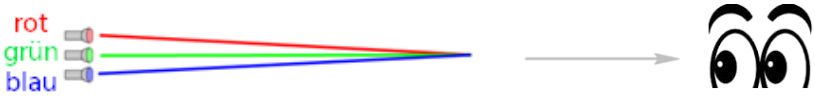
Für jedes externe File wird ein eigener http-Request benötigt.

Farben (Colors)

CSS-Farbangaben beziehen sich immer auf den **additiven RGB-Farbraum** (siehe Wikipedia).

additiver Farbraum

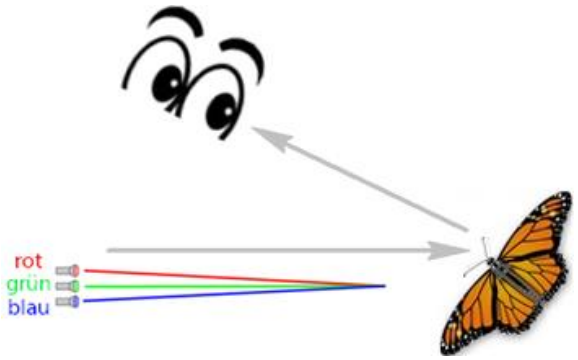
In einem additiven Farbraum strahlt das betrachtete Medium Licht aus. Das Objekt ist also selbstleuchtend (zB Sonne, Lampe und Glühwürmchen).
Jede Farbe wird durch das Mischen von Rot, Grün und Blau erzeugt. Diese drei Farben werden **Lichtfarben** genannt.
Siehe RGB-Notation Seite 212.



Sind die drei Grundfarben gleichmässig verteilt, entsteht Weiß.
Gelb (Yellow) entsteht durch gleichmässige Mischung von **Rot und Grün.**
Türkis (Cyan) entsteht durch gleichmässige Mischung von **Grün und Blau**.
Purpur (Magenta) ensteht durch gleichmässige Mischung von **Blau und Rot**.

subtraktiver Farbraum

In einem **subtraktiven Farbsystem** (siehe Wikipedia) wird das betrachtete Medium (Buch, Wand, Blume) angeleuchtet.
Die Farbe entsteht durch die reflektierenden Farbanteile. Und somit indirekt durch das Wegfallen von Farbanteilen.
Dies geschieht duch **Remission (**Absorption) oder T**ransmission** (Farbfilter).



Es werden also, ausgehend von Weiß als Grundfarbe, Farbanteile weggenommen (durch Absorption und Filterung). Für das subtraktive Farbsystem wird der CMY-Farbraum verwendet. Beim Druck werden die durchsichtigen Farbschichten Cyan, Magenta und Gelb (Yellow) auf das Papier aufgetragen.
Für ein besseres Schwarz wird dieses Modell durch ein explizites Hinzufügen von Schwarz erweitert: CMYK (K steht für den Schwarzanteil). Das K in der Abkürzung stammt von der Verwendung einer **Schlüsselplatte** (Key plate) im Druck.

Wird weißes Licht durch einen **Rotfilter** geschickt (Rot wird subtrahiert), entsteht **Cyan.**
Wird weißes Licht durch einen **Grünfilter** geschickt, entsteht **Magenta.**
Wird weißes Licht durch einen **Blaufilter** geschickt, entsteht **Yellow.**

Wird weißes Licht durch einen **Grün- und Blaufilter** geschickt, entsteht **Rot.**
Wird weißes Licht durch einen **Rot- und Blaufilter** geschickt, entsteht **Grün.**
Wird weißes Licht durch einen **Grün- und Rotfilter** geschickt, entsteht **Blau.**

RGB-Werte

Die Werte für die drei einzelnen Farben Rot, Grün und Blau können **absolut** (jeweils zwischen 0 und 255) oder **prozentual** angegeben werden. Siehe **additiver Farbraum** auf Seite 211.

Bei absoluten Angaben ergeben sich 256^3 Möglichkeiten, und somit 16.777.216 Millionen Farben.

Die absoluten Werte können dezimal (Zahlenbasis 10) oder hexadezimal (Zahlenbasis 16) sein.

Alle 4 Angaben erzeugen die gleiche Farbe (red):

Farbangabe hexadzimal: `color: #ff0000;`
Farbangabe dezimal: `color: rgb(255,0,0);`
Farbangabe prozentual: `color: rgb(100%,0%,0%);`
Farbangabe mit Keyword: `color: red;`

Basic Color Keywords

Seit CSS1 gibt es 16 vordefinierte Basic-Keywords. Mit CSS2.1 kam als 17. Farbe orange.

	Color Keyword	Hexadezimal	Dezimal
	black	#000000	0,0,0
	silver	#C0C0C0	192,192,192
	gray	#808080	128,128,128
	white	#FFFFFF	255,255,255
	maroon	#800000	128,0,0
	red	#FF0000	255,0,0
	orange	#ffa500	255,165,0
	purple	#800080	128,0,128
	fuchsia	#FF00FF	255,0,255
	green	#008000	0,128,0
	lime	#00FF00	0,255,0
	olive	#808000	128,128,0
	yellow	#FFFF00	255,255,0
	navy	#000080	0,0,128
	blue	#0000FF	0,0,255
	teal	#008080	0,128,128
	aqua	#00FFFF	0,255,255

Extended Color Keywords

	Color Keyword	Hexadezimal	Dezimal
	aliceblue	#f0f8ff	240,248,255
	antiquewhite	#faebd7	250,235,215
	aqua	#00ffff	0,255,255
	aquamarine	#7fffd4	127,255,212
	azure	#f0ffff	240,255,255
	beige	#f5f5dc	245,245,220
	bisque	#ffe4c4	255,228,196
	black	#000000	0,0,0
	blanchedalmond	#ffebcd	255,235,205
	blue	#0000ff	0,0,255
	blueviolet	#8a2be2	138,43,226
	brown	#a52a2a	165,42,42
	burlywood	#deb887	222,184,135
	cadetblue	#5f9ea0	95,158,160
	chartreuse	#7fff00	127,255,0
	chocolate	#d2691e	210,105,30
	coral	#ff7f50	255,127,80
	cornflowerblue	#6495ed	100,149,237
	cornsilk	#fff8dc	255,248,220
	crimson	#dc143c	220,20,60
	cyan	#00ffff	0,255,255
	darkblue	#00008b	0,0,139
	darkcyan	#008b8b	0,139,139
	darkgoldenrod	#b8860b	184,134,11
	darkgray	#a9a9a9	169,169,169
	darkgreen	#006400	0,100,0
	darkgrey	#a9a9a9	169,169,169
	darkkhaki	#bdb76b	189,183,107
	darkmagenta	#8b008b	139,0,139
	darkolivegreen	#556b2f	85,107,47
	darkorange	#ff8c00	255,140,0
	darkorchid	#9932cc	153,50,204
	darkred	#8b0000	139,0,0
	darksalmon	#e9967a	233,150,122
	darkseagreen	#8fbc8f	143,188,143
	darkslateblue	#483d8b	72,61,139
	darkslategray	#2f4f4f	47,79,79
	darkturquoise	#00ced1	0,206,209
	darkviolet	#9400d3	148,0,211
	deeppink	#ff1493	255,20,147
	deepskyblue	#00bfff	0,191,255
	dimgray	#696969	105,105,105
	dodgerblue	#1e90ff	30,144,255
	firebrick	#b22222	178,34,34
	floralwhite	#fffaf0	255,250,240
	forestgreen	#228b22	34,139,34
	fuchsia	#ff00ff	255,0,255
	gainsboro	#dcdcdc	220,220,220
	ghostwhite	#f8f8ff	248,248,255
	gold	#ffd700	255,215,0

	goldenrod	#daa520	218,165,32
	green	#008000	0,128,0
	greenyellow	#adff2f	173,255,47
	grey	#808080	128,128,128
	honeydew	#f0fff0	240,255,240
	hotpink	#ff69b4	255,105,180
	indianred	#cd5c5c	205,92,92
	indigo	#4b0082	75,0,130
	ivory	#fffff0	255,255,240
	khaki	#f0e68c	240,230,140
	lavender	#e6e6fa	230,230,250
	lavenderblush	#fff0f5	255,240,245
	lawngreen	#7cfc00	124,252,0
	lemonchiffon	#fffacd	255,250,205
	lightblue	#add8e6	173,216,230
	lightcoral	#f08080	240,128,128
	lightcyan	#e0ffff	224,255,255
	lightgoldenrodyellow	#fafad2	250,250,210
	lightgray	#d3d3d3	211,211,211
	lightgreen	#90ee90	144,238,144
	lightgrey	#d3d3d3	211,211,211
	lightpink	#ffb6c1	255,182,193
	lightsalmon	#ffa07a	255,160,122
	lightseagreen	#20b2aa	32,178,170
	lightskyblue	#87cefa	135,206,250
	lightslategray	#778899	119,136,153
	lightsteelblue	#b0c4de	176,196,222
	lightyellow	#ffffe0	255,255,224
	lime	#00ff00	0,255,0
	limegreen	#32cd32	50,205,50
	linen	#faf0e6	250,240,230
	magenta	#ff00ff	255,0,255
	maroon	#800000	128,0,0
	mediumaquamarine	#66cdaa	102,205,170
	mediumblue	#0000cd	0,0,205
	mediumorchid	#ba55d3	186,85,211
	mediumpurple	#9370db	147,112,219
	mediumseagreen	#3cb371	60,179,113
	mediumslateblue	#7b68ee	123,104,238
	mediumspringgreen	#00fa9a	0,250,154
	mediumturquoise	#48d1cc	72,209,204
	mediumvioletred	#c71585	199,21,133
	midnightblue	#191970	25,25,112
	mintcream	#f5fffa	245,255,250
	mistyrose	#ffe4e1	255,228,225
	moccasin	#ffe4b5	255,228,181
	navajowhite	#ffdead	255,222,173
	navy	#000080	0,0,128
	oldlace	#fdf5e6	253,245,230
	olive	#808000	128,128,0
	olivedrab	#6b8e23	107,142,35
	orange	#ffa500	255,165,0
	orangered	#ff4500	255,69,0
	orchid	#da70d6	218,112,214

	palegoldenrod	#eee8aa	238,232,170
	palegreen	#98fb98	152,251,152
	paleturquoise	#afeeee	175,238,238
	palevioletred	#db7093	219,112,147
	papayawhip	#ffefd5	255,239,213
	peachpuff	#ffdab9	255,218,185
	peru	#cd853f	205,133,63
	pink	#ffc0cb	255,192,203
	plum	#dda0dd	221,160,221
	powderblue	#b0e0e6	176,224,230
	purple	#800080	128,0,128
	red	#ff0000	255,0,0
	rosybrown	#bc8f8f	188,143,143
	royalblue	#4169e1	65,105,225
	saddlebrown	#8b4513	139,69,19
	salmon	#fa8072	250,128,114
	sandybrown	#f4a460	244,164,96
	seagreen	#2e8b57	46,139,87
	seashell	#fff5ee	255,245,238
	sienna	#a0522d	160,82,45
	silver	#c0c0c0	192,192,192
	skyblue	#87ceeb	135,206,235
	slateblue	#6a5acd	106,90,205
	slategray	#708090	112,128,144
	snow	#fffafa	255,250,250
	springgreen	#00ff7f	0,255,127
	steelblue	#4682b4	70,130,180
	tan	#d2b48c	210,180,140
	teal	#008080	0,128,128
	thistle	#d8bfd8	216,191,216
	tomato	#ff6347	255,99,71
	turquoise	#40e0d0	64,224,208
	violet	#ee82ee	238,130,238
	wheat	#f5deb3	245,222,179
	white	#ffffff	255,255,255
	whitesmoke	#f5f5f5	245,245,245
	yellow	#ffff00	255,255,0
	yellowgreen	#9acd32	154,205,50

@media

Mit @media-Aufruf wird eine Formatierung nur für ein bestimmtes Ausgabegerät definiert.

Syntax:

```
@media mediatyp
        {
        CSS-Rule-Set
        }
```

Mediatypen

all	für alle Ausgabegeräte.
aural	für aurale Ausgabegeräte (vorlesender Browser).
braille	für Blindenschriftfähige Ausgabegeräte.
embossed	für Drucker, die Blindenschrift ausgeben können.
handheld	Ausgabegeräte mit kleinem Display
print	für Drucker.
projection	für projezierte Präsentationen (z.B. Projektoren, Polylux, etc.).
screen	allgemein für alle (Computer-)Bildschirme.
tty	für Ausgabegeräte mit feststehendem Symbolgitter/Schriftart ("fixed-pitch character grid")
tv	für Fernseh-Bildschirme.

Beispiel:

```
@media print
{
 body {color: black; background-color: white}
}
```

Zeichensatz

Mit @charset wird der Zeichensatz bestimmt. Eine Auflistung aller RFC2978 Character-Sets siehe IANA.

Beispiel:

```
@charset "ISO-8859-1"
```

externe Schrift

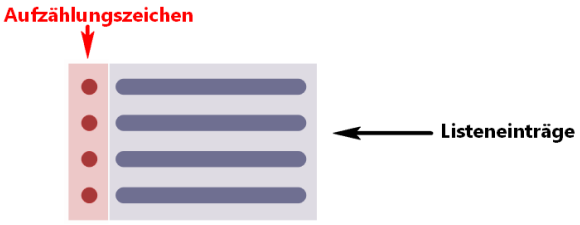
Mit @font-face eine externe Schrift inkludiert.

Beispiel:

```
@font-face
{
 src: url("http://www.foo.com/foobar.eot");
 font-family: 'meine Schrift';
}
p {font-family: 'meine Schrift';}
```

Liste

Listen bestehen aus 2 Elementen: Das `ul` oder `ol` Element bestimmt die Art der Liste, das `li` Element markiert den Listeneintrag.



Bei einem ul ist das Aufzählungszeichen ein graphisches Symbol.

Bei einem ol (Ordered List) ist das Aufzählungszeichen eine (Alpha-)nummerische Sortierung.

Siehe HTML-Liste Seite 142.

Siehe auch CSS-Counter Seite 219.

list-style-type

Bei sortierten Listen (`ol` siehe Seite 142) zur Definition der (alpha-)nummerischen Sortierung. Bei Aufzählungslisten (`ul` siehe Seite 142) zur Definition des Aufzählungszeichen (Bullet). Siehe auch mozilla.org.

Eigenschaftswerte bei ul

`disc`	•	gefüllter Kreis als Bullet (default)
`circle`	○	leerer Kreis als Bullet
`square`	■	quadratisches Bullet
none		kein Bullet
`diamond`	◆	
`check`	✓	
`box`	□	

Eigenschaftswerte bei ol

`decimal`	Nummerierung 1., 2., 3., 4.
`decimal-leading-zero`	Nummerierung 01., 02., 03., 04.
`lower-roman`	Nummerierung i., ii., iii., iv.
`upper-roman`	Nummerierung I., II., III., IV.
`lower-alpha`	Nummerierung a., b., c., d.
`lower-latin`	Nummerierung a., b., c., d.
`upper-alpha`	Nummerierung A., B., C., D.
`upper-latin`	Nummerierung A., B., C., D.
none	keine Nummerierung
`binary`	Nummerierung als Binärzahlen 0, 1, 10, 11, 100, 101, ...
`ower-hexadecimal`	Nummerierung als Hexadezimalzahlen, 0, 1, 2, ... , 9, a, b, ... , f, 10 ...
`upper-hexadecimal`	Nummerierung als Hexadezimalzahlen, 0, 1, 2, ... , 9, A, B, ... , F, 10 ...
`super-decimal`	Nummerierung in hochgestellten Dezimalzahlen
`asterisks`	Nummerierung mit Asterisks, *, **, ***, ...
`footnotes`	Nummerierung als Fußnoten, Zeichen 2A, 2051, 2020, 2021 (*, ⁂, †, ‡)

Beispiel:

```
<style>
    ol
    {
       list-style-type:lower-alpha;
    }
</style>
<ol>
 <li>1</li>
 <li>2</li>
    <ol>
       <li>1</li>
       <li>2</li>
    </ol>
 <li>3</li>
</ol>
```

Erzeugt als Output:

a. 1
b. 2
    a. 1
    b. 2
c. 3

list-style-position

Bestimmt die Position des Aufzählungszeichen. Siehe auch mozilla.org.

Eigenschaftswerte:

`inside`	im rechten Block beim li Content
`outside`	im linken Block (default)

Beispiel für outside:

```
<style>
ul
    {
          list-style-position:outside;
          background-color:yellow;
    }
li { background-color:lime;}
</style>
<ul>
   <li>Fehlt es mir etwa an Verrückten</li>
</ol>
```

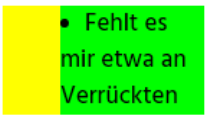
Ergibt als Output:

- Fehlt es mir etwa an Verrückten

Beispiel für inside:

```
<style>
ul
    {
        list-style-position:outside;
        background-color:yellow;
    }
li { background-color:lime;}
</style>
<ul>
   <li>Fehlt es mir etwa an Verrückten</li>
</ol>
```

Ergibt als Output:



list-style-image

Bestimmt ein Bild als Aufzählungszeichen bei einem `ul` Element. Siehe mozilla.org.

```
<style>
ul { list-style-image: url("stern.jpg") }
</style>
<ul>
    <li>1</li>
    <li>2</li>
</ul>
```

Ergibt als Output:

Counter

counter-increment

Mit dem Keyword `counter-increment` kann eine Variable um einen bestimmten Interger-Wert erhöht und gesenkt werden. Dies geschieht bei jeder Verwendung des CSS-Deklarationblockes. Der Gültigkeitsbereich der Variable ist das Element und seine Nachfahren. Siehe W3.

Erhöhe foo um 1: `h6 { counter-increment: foo 1; }`

Erhöhe foo um 3: `h6 { counter-increment: foo 3; }`

Senke foo um -3: `h6 { counter-increment: foo -3; }`

Das Keyword kann pro Deklarationsblock nur einmal verwendet werden.
Bei Mehrfach-Verwendung wird nur die letzte ausgeführt.

counter

Mit der CSS-Funktion `counter()` kann die Variable abgerufen und optional formatiert werden:

counter (*Variable* [,*FormatType*])

Beispiel:

```
h6::before {
  content: 'foo ist ' counter(foo) ': ';
}
```

FormatType entspricht den Werten für `list-style-type` (siehe Seite 217).

Beispiel: counter(foo,decimal-leading-zero)

FormatType in Anführungszeichen wird als String interpretiert und der Variable angehängt (String-Konkatenation).

Beispiel: counter(foo," (Wert von foo1)")

Pseudoelement `::before` siehe Seite 203.

counter-reset

Mit dem Keyword `counter-reset` kann eine Variable auf einen bestimmten Integer-Wert (default ist 0) gesetzt werden.

Setze foo auf 0: `h6 { counter-reset: foo; }`

Setze foo auf 3: `h6 { counter-reset: foo 3; }`

Setze foo auf -5:`h6 { counter-reset: foo -5; }`

Beispiel

```
<style>
b { counter-reset: foo 1; }              /* foo auf 1 setzen */
h6 { counter-increment: foo 1;}            /* foo=foo+1 */
h6::before {
  content: 'foo ist ' counter(foo,decimal-leading-zero) ': ' ;
}

</style>
 <h6>blub1</h6>
<h6>blub2</h6>
<h6>blub3</h6>
<b>reset</b>
<h6>blub4</h6>
```

Ergibt als Output:

foo ist 01: blub1

foo ist 02: blub2

foo ist 03: blub3

reset

foo ist 02: blub4

Zeichen

Kapitel 7

Zeichen (Character)

Ein Zeichen ist abstrakt und wird erst durch eine Glyphe optisch dargestellt.

Glyphe

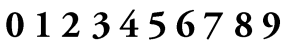
Die Glyphe stellt optisch ein Zeichen (Buchstabe/Symbol) dar. Zu einem Zeichen kann es mehrere Glyphen geben.
Als Beispiel einige Glyphen des Zeichen A:

Glyphenvarianten

Ähnliche Glyphen werden Glyphenvarianten genannt. Als Beispiel der Buchstabe A in der Schriftart Garamond und GaramondBold:

Zeichensatz (Character Set)

Ein Zeichensatz ist eine Menge von Zeichen (Gesamtheit der zur Verfügung stehenden Zeichen). Als Beispiel ein Ziffernzeichensatz von 0 bis 9, optisch dargestellt durch die Glyphen von GaramondBold:



Besitzt der Zeichensatz (charset) eine fixe Reihenfolge der Zeichen, kann diese für eine Kodierung und Erstellung einer Zeichensatztabelle verwendet werden. Eine Kodierung kann auch ohne Verwendung von bits erfolgen, zB rot13.
Die Verwendung von **Code Points** ist für einen Zeichensatz nicht zwingend notwendig.

Zeichensatztabelle (Codepage)

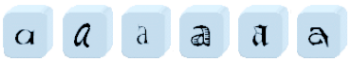
Jedem Zeichen in einem Zeichensatz ein eindeutiger nummerischer Wert (ID) zugewiesen. Solch eine Zuordnung wird Kodierung (Encode) genannt. Eine Auflistung aller RFC2978 Character-Sets siehe IANA. Als Beispiel US-ASCII:

```
65     A
97     a
42     *
160    (Leerzeichen)
```

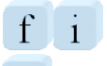
Wird für diese ID eine andere Zeichensatztabelle verwendet, kann **Mojibake** (Zeichensalat) entstehen (siehe Seite 223).

Die Glyphe einer Schriftart (und nicht der Zeichensatz), bestimmt das konkrete Aussehen des Symbols. Für ein Zeichen gibt es meist mehrere Glyphen.

Das Zeichen (Character) a:

hat zB folgende Glyphen: 
Diese Glyphen haben keinen eigenen **Code Point.**

Mehrere Zeichen (Character Sequence) können zu einer Glyphe zusammengefasst werden.

Die Zeichenkette fi 

als eine einzige Glyphe:
Diese Glyphen haben ihren eigenen **Code Point.**

Kodierung (Encoding)

Die Kodierung von Zeichen besteht aus 2 Teilen:

1) **Code Point**
   In europäischen Sprachen ist jedem Zeichen eine fixe und eindeutige ID (Position) in einer Zeichensatztabelle zugeordnet. In ASCII hat der Buchstabe A (großes A) den **Code Point** 65 (dezimal).

2) Bit-Kodierung des Code Point
   a) fixe Bit-Länge
      ZB besteht in der einfachen Variante von ASCII jedes Zeichen aus 7bit. Das ergibt eine maximale Zeichensatzgröße von 2^7 (128 Zeichen). In der extenden ASCII-Version besteht jedes Zeichen aus 8bit, also maximal 2^8 (256) Zeichen. Siehe **extended ASCII** Seite 230.
   b) unterschiedliche Bit-Länge
      In UTF-8 bestehen Zeichen aus unterschiedlichen Anzahl von Bits (1 bis 4 Bytes pro Zeichen).

Code Point, Code Unit

Ein Code Point ist eindeutiger nummerischer Wert, der einem Character (Zeichen) in einer Character Table (Zeichentabelle) entspricht. In ASCII hat der Buchstabe A (großes A) den **Code Point** 65 (dezimal).

Code Unit ist die binäre Repräsentation des Code Points und hängt vom **Character Encoding Scheme** ab.
Als Beispiel der Buchstabe A in ASCII: `0100 0001`
Eine Auflistung aller RFC2978 Character-Sets siehe IANA.

Mojibake (文字化け)

Wenn nicht die richtige (ursprüngliche) Zeichensatztabelle für einen Text verwendet wird, kann **Mojibake** entstehen. Ein Zeichensalat, dessen Ursache eine falsche Zeichensatztabelle ist.
Der Begriff 文字化け (Mojibake) wurde erstmal in Japan verwendet und bedeutet "geänderte Zeichen" (meist im negativen Sinne).

Manchmal wird der Begriff auch verwendet, wenn die Ursache für den Buchstabensalat eine unpassende Schriftart ist.

Beispiel 1:
Japanisch in UTF-8: 文字化け
Die gleiche Bytefolge in ISO-8859-1: æ–‡å——åŒ–ã• ‘

Beispiel 2:
Ein isländische Wort in UTF-8: þjóðlöð
Die gleiche Bytefolge in ISO-8859-1: Ã¾jÃ³Ã°lÃ¶Ã°

Beispiel 3:
Arabisch in UTF-8: الإعلان العالمى لحقوق الإنسان
In ISO-8859-1: ï»¿Ø§Ù"Ø¥Ø¹Ù"Ø§Ù†Ø§Ù"Ø¹Ø§Ù"Ù…Ù‰ Ù"ØÙ‚Ù‚ Ø§Ù"Ø¥Ù†Ø³Ø§Ù†
In ISO-8859-2: ď˙ťžŘ§Ů�ŘĽŘšŮ�Ř§Ů Ř§Ů�ŘšŘ§Ů�ŮŮ Ů�ŘŮ�ŮŮ Ř§Ů�ŘĽŮ�ŘŕŘ§Ů
In ISO-8859-5: яЛПиЇй�иЅиЙй�иЇй иЇй�иЙиЇй�й�й й�ий�й�й иЇй�иЅй�иГиЇй

ASCII

Die erste Version von American Standard Code for Information Interchange (ASCII) erschien 1963. Sie ist eine 7bit-Zeichensatzkodierung, die 2^7 (128) Zeichen enthält (von 0 bis 127).

Die zweite Version von Zeichensatztabelle (Codepage) ist von 1967 (USAS X3.4-1967): Einige Steuerzeichen fielen weg, das ESC wurde von Position 126 auf Position 27 verschoben. Auf Position 126 kam neu dazu die Tilde. Zwischen 97 und 122 wurden Kleinbuchstaben hinzugefügt.

1968 bekam `LF` auch die alternative Bedeutung **New Line** (`NL`).

7bit-ASCII:

Dezimal	Hex	
000-031	00-1F	*C0 Control Characters*
032	20	*Space (SP)*
033-047	20-2F	`!"#$%&'()*+,-./`
048-063	30-3F	`0123456789:;<=>?`
064-095	40-5F	`@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`
096-126	60-7E	`` `abcdefghijlkmnopqrstuvwxyz{|}~ ``
127	7F	*Delete (DEL)*

8bit-ASCII:

128-159	80-9F	*C1 Control Characters*
160	A0	*No Break Space (NBSP): erzwungenes Leerzeichen (kam mit ISO-8859-1)*
161-254	A1-FE	*94 additional displayable characters*
255	FF	Soft Hyphen (SHY): bedingtes (weiches) Trennzeichen kam mit ISO-8859-1)

Versalie, Majuskel

Großbuchstaben werden in der Typographie Majuskel genannt.
Kleinbuchstaben werden Minuskel und Gemeine genannt. Sie wurden 1967 dem ASCII-Zeichensatz hinzugefügt (USAS X3.4-1967) und liegen zwischen Code-Point 97 und 127

Control Characters (Steuerzeichen)

Die ersten 32 Zeichen (Position 0 bis 31) sind Steuerzeichen (control characters). Dieser Code-Point-Block wird **C0 Control Characters** geannt. Mit ANSI X3.64 kamen mit **C1 Control Characters** (Position 128 bis 159) ein weiterer Block hinzu.

Space (SP), Delete (DEL)

Space (SP) auf Position 128 und Delete (DEL) auf Position 129 werden sowohl als **Control Character**, als auch **non-printing graphic character** betrachtet.

No Break Space (NBSP), Soft Hyphen (SHY)

Mit ISO-8859-1 wurde auf Position 160 **No Break Space (NBSP)** eingeführt (erzwungenes Leerzeichen) und **Soft-Hyphen (SHY)** auf Position 255 eingeführt. Sie werden manchmal zu den Control Characters dazugezählt.

NBSP erzwingt ein Leerzeichen (an dieser Stelle darf die Zeile nicht umgebrochen werden). **html-Entity**:

SHY ist ein weiches Trennzeichen. Es kennzeichnet eine optimale Position (Sollbruchstelle) für einen eventuell notwendigen Zeilenumbruch. Falls das Wort getrennt wird, erscheint ein Trennstrich. Wird das Wort nicht getrennt, bleibt das weiche Trennzeichen für den Menschen unsichtbar.Sie html-Tag <wb> (Seite 134) **html-Entity**:

Caret Notation

Die Caret Notation ermöglicht die Eingabe von Steuerzeichen per Tastatur. Für komplette Liste Siehe **ASCII Control Characters** Seite 225. Die Notation besteht aus 2 Teilen, dem ^ und (meist) einem Buchstaben:

^ *Buchstabe*

Das 0. Zeichen (NUL) ist:

^@

Das 1. Zeichen (SOH) ist (A=1):

^A

Das 2. Zeichen (STX) ist (B=1):

^B

Das 26. Zeichen (SUB) ist (Z=26):

^Z

Das 27. Zeichen (ESC) ist:

^?

Escape Sequence (Fluchtsequenz)

In einem Textfile werden Steuerzeichen über einen Escape Character (Fluchtsymbol) maskiert. Das Fluchtsymbol ist der Backslash: \

Die Escape Sequenz besteht aus 2 Teilen, dem \ und (meist) einem Buchstaben:

\ *Buchstabe*

Zum Beispiel:

\t	Tab
\n	New Line (Form Feed)
\a	Bell (Glockenton, Piep)

Für alle Escape Sequenzen siehe Seite 226, 6 Tabellenspalte.

ASCII C0 Control Characters (0 – 31)

dec	octal	hex	ctrl+	Symbol	Escape	carret	Beschreibung
0	000	00		NUL	\0	^@	Null char (a filler character. doesn't affect the information content). Siehe Seite 229.
1	001	01	A	SOH		^A	Start of Heading (Indicates the beginning of a heading in a transmission)
2	002	02	B	STX		^B	Start of Text (indicates the beginning of a text)
3	003	03	C	ETX		^C	End of Text (Terminates a text in a transmission.)
4	004	04	D	EOT		^D	End of Transmission (even used as an End-Of-File control in a Unix shell session)
5	005	05	E	ENQ		^E	Enquiry (Requests a response from a remote station)
6	006	06	F	ACK		^F	Acknowledgment (An affirmative response. Transmitted as a response to the sender.)
7	007	07	G	BEL	\a	^G	Bell (calls for human attention. BEL may control alarm or attention devices)
8	010	08	H	BS	\b	^H	Back Space (non-destructive (!): Moves one character position backwards)
9	011	09	I	HT	\t	^I	Horizontal Tab (to the next pre-determined character position)
10	012	0A	J	LF	\n	^J	Line Feed (New Line) Move down or move to start of next line (nl)
11	013	0B	K	VT	\t	^K	Vertical Tab (Advances to the same character position on the next pre-determined line.)
12	014	0C	L	FF	\f	^L	Form Feed (Advances to the next form or page)
13	015	0D	M	CR	\r	^M	Carriage Return (Moves to the line home position)
14	016	0E	N	SO		^N	Shift Out / X-On (Used to extend the character set)
15	017	0F	O	SI		^O	Shift In / X-Off (Used in conjunction with SO.)
16	020	10	P	DLE		^P	Data Line Escape (DLE is the "escape" character for transmission control.)
17	021	11	Q	DC1		^Q	Device Control 1 (oft. XON) (Intended to turn on or start an ancillary device)
18	022	12	R	DC2		^R	Device Control 2 (Intended for turning on or starting an ancillary device)
19	023	13	S	DC3		^S	Device Control 3 (oft. XOFF) Intended for turning off or stopping
20	024	14	T	DC4		^T	Device Control 4 (Intended to turn off, stop or interrupt an ancillary device)
21	025	15	U	NAK		^U	Negative Acknowledgement (to indicate inability to receive a message).
22	026	16	V	SYN		^V	Synchronous Idle (Used as "time-fill" in synchronous transmission. during an idle condition)
23	027	17	W	ETB		^W	End of Transmit Block (when data is divided into blocks for transmission)
24	030	18	X	CAN		^X	Cancel (Indicates that data is in error or should be disregarded)
25	031	19	Y	EM		^Y	End of Medium (only suitable for paper/ magnetic tape to say "no more data")
26	032	1A	Z	SUB		^Z	Substitute (Used in place of an invalid or erroneous character)
27	033	1B	[	ESC	\e	^?	Escape
28	034	1C	\	FS			Field Separator (FS) sieh SUBSEP Seite 78
29	035	1D	]	GS			Group Separator
30	036	1E	^	RS			Record Separator (RS)
31	037	1F	_	US			Unit Separator
32	040	20		SP			Space
127	177	7F		DEL			Delete

Space (SP) und Delete (DEL) werden sowohl als **Control Character**, als auch **non-printing graphic character** betrachtet.

ASCII C1 Control Characters (128 – 159)

dezimal	octal	hex	Esc+	Symbol	Beschreibung
128	000	80		PAD	unassigned, "Padding Character" (A reserved control code)
129	001	01	A	HOP	unassigned, "High Octet Preset" (A reserved control code)
130	002	02	B	BPH	Break Permitted Here (A point where a line break may occur)
131	003	03	C	NBH	No Break Here (A point where a line break may not occur)
132	004	04	D	IND	Index (Moves to the next line keeping the current horizontal position)
133	005	05	E	NEL	Next Line (Moves to the first position of the next line. Or to line home or line limit position)
134	006	06	F	SSA	Start of Selected Area (Starts a string of character positions whose contents can be transmitted.)
135	007	07	G	ESA	End of Selected Area. Ends a string of character positions (started by SPA)
136	010	08	H	HTS	Horizontal Tabulation Set (Sets a tab stop at the active position)
137	011	09	I	HTJ	Horizontal Tabulation with Justification (Moves text to the following tab stop)
138	012	0A	J	VTS	Vertical Tabulation Set (Sets a vertical tab stop at the active line)
139	013	0B	K	PLD	Partial Line Down (Moves down to start subscripts. Subscripts end at the next PLU)
140	014	0C	L	PLU	Partial Line Up (Moves up to start superscripts. Superscripts end at the next PLD)
141	015	0D	M	RI	Reverse Index (Moves to the previous line keeping the current horizontal position)
142	016	0E	N	SS2	Single Shift Two (extend the char set. Next character will be from the currently chosen G2 set.)
143	017	0F	O	SS3	Single Shift Three (extend the character set. Next Char will be from the currently chosen G3 set.)
144	020	10	P	DCS	Device Control String (Starts a device control string. ST ends the string)
145	021	11	Q	PU1	Private Use One (Reserved for private use, no standardized meaning)
146	022	12	R	PU2	Private Use Two (Reserved for private use, no standardized meaning)
147	023	13	S	STS	Set Transmit State (Notifies that data is ready for transfer from a device)
148	024	14	T	CCH	Cancel Character (Ignore the preceding graphic character (and CCH itself too)
149	025	15	U	MW	Message Waiting (Sets a message waiting indicator in the receiving device)
150	026	16	V	SPA	Start of Guarded Protected Area (EPA will end the string)
151	027	97	W	EPA	End of Guarded Protected Area (Ends the area started by SPA)
152	030	98	X	SOS	Start of String (Starts a control string. The string ends at ST. It cannot contain a SOS)
153	031	99	Y	SGCI	unassigned, "Single Graphic Character Introducer"
154	032	9A	Z	SCI	Single Character Introducer
155	033	9B	[	CSI	Control Sequence Introducer (Starts a control sequence)
156	034	9C	\	ST	String Terminator (Closes a string opened by APC, DCS, OSC, PM or SOS)
157	035	9D	]	OSC	Operating System Command (Starts an operating system control string. The string ends at ST)
158	036	9E	^	PM	Privacy Message (Starts a privacy message. ST will end the message)
159	037	9F	_	APC	Application Program Command (Starts an APC string. ST will end the command.)
160	240	A0		NBSP	No-Break Space. A space for use when a line break is to be prevented. html-Entity: Taste: Ctrl+Shift+SPACE
173	255	AD		SHY	Soft Hyphen. Indicates an intraword break point for use when a word must be broken across lines. The visual rendering either is a hyphen. html-Entity: Taste: Ctrl+-

ASCII Character Codes

DEC	OCT	HEX	Symbol	HTML-Entity	Beschreibung	Description
32	040	20			Leerzeichen	Space
33	041	21	!		Rufzeichen, Ausrufungszeichen	Exclamation mark
34	042	22	"	"	doppeltes Anführungszeichen	Double quotes (or speech marks)
35	043	23	#		Raute, Doppelkreuz, Nummernzeichen	Number Sign, Hash, Hash Mark
36	044	24	$		Dollarzeichen	Dollar Sign
37	045	25	%		Prozentzeichen	Procenttecken
38	046	26	&	&	Kaufmännisches Und	Ampersand
39	047	27	'		einfaches Anführungszeichen	Single quote
40	050	28	(		Öffnende runde Klammer	Open parenthesis (or open bracket)
41	051	29	)		Schließende runde Klammer	Close parenthesis (or close bracket)
42	052	2A	*		Stern	Asterisk
43	053	2B	+		Plus	Plus
44	054	2C	,		Komma	Comma
45	055	2D	-		Hyphen	
46	056	2E	.		Punkt	Period, Dot, Full stop
47	057	2F	/		Schrägstrich	Slash or divide
48	060	30	0		Null	Zero
49	061	31	1		Eins	One
50	062	32	2		Zwei	Two
51	063	33	3		Drei	Three
52	064	34	4		Vier	Four
53	065	35	5		Fünf	Five
54	066	36	6		Sechs	Six
55	067	37	7		Sieben	Seven
56	070	38	8		Acht	Eight
57	071	39	9		Neun	Nine
58	072	3A	:		Doppelpunkt	Colon
59	073	3B	;		Strichpunkt	Semicolon
60	074	3C	<	<	Kleinerzeichen	Less than (or open angled bracket)
61	075	3D	=		Gleich	Equals
62	076	3E	>	>	Größerzeichen	Greater than (or close angled bracket)
63	077	3F	?		Fragezeichen	Question mark
64	100	40	@		Klammeraffe	At symbol
65	101	41	A			Uppercase A
66	102	42	B			Uppercase B
67	103	43	C			Uppercase C
68	104	44	D			Uppercase D
69	105	45	E			Uppercase E
70	106	46	F			Uppercase F
71	107	47	G			Uppercase G
72	110	48	H			Uppercase H
73	111	49	I			Uppercase I
74	112	4A	J			Uppercase J
75	113	4B	K			Uppercase K
76	114	4C	L			Uppercase L
77	115	4D	M			Uppercase M
78	116	4E	N			Uppercase N
79	117	4F	O			Uppercase O
80	120	50	P			Uppercase P
81	121	51	Q			Uppercase Q
82	122	52	R			Uppercase R
83	123	53	S			Uppercase S
84	124	54	T			Uppercase T
85	125	55	U			Uppercase U
86	126	56	V			Uppercase V
87	127	57	W			Uppercase W
88	130	58	X			Uppercase X
89	131	59	Y			Uppercase Y
90	132	5A	Z			Uppercase Z

DEC	OCT	HEX	Symbol	HTML Entity	Description	
91	133	5B	[		Öffnende eckige Klammer	Opening bracket
92	134	5C	\		Umgekehrter Schrägstrich	Backslash
93	135	5D	]		Schließende eckige Klammer	Closing bracket
94	136	5E	^			Caret - circumflex
95	137	5F	_		Unterstrich	Underscore
96	140	60	`			Grave accent
97	141	61	a			Lowercase a
98	142	62	b			Lowercase b
99	143	63	c			Lowercase c
100	144	64	d			Lowercase d
101	145	65	e			Lowercase e
102	146	66	f			Lowercase f
103	147	67	g			Lowercase g
104	150	68	h			Lowercase h
105	151	69	i			Lowercase i
106	152	6A	j			Lowercase j
107	153	6B	k			Lowercase k
108	154	6C	l			Lowercase l
109	155	6D	m			Lowercase m
110	156	6E	n			Lowercase n
111	157	6F	o			Lowercase o
112	160	70	p			Lowercase p
113	161	71	q			Lowercase q
114	162	72	r			Lowercase r
115	163	73	s			Lowercase s
116	164	74	t			Lowercase t
117	165	75	u			Lowercase u
118	166	76	v			Lowercase v
119	167	77	w			Lowercase w
120	170	78	x			Lowercase x
121	171	79	y			Lowercase y
122	172	7A	z			Lowercase z
123	173	7B	{		Öffnende geschweifte Klammer	Opening brace
124	174	7C	|		Senkrechter Strich	Vertical bar
125	175	7D	}		Schließende geschweifte Klammer	Closing brace
126	176	7E	~			Equivalency sign - tilde
127	177	7F			Tilde	Delete

NUL

Das Nullzeichen (Null char) ist ein Füllzeichen (filler character). Es hat keine Funktion und wird deshalb in Situationen eingesetzt, in denen Bytes zwingend vorhanden sein müssen. Ähnlich dem Assembler-Befehl NOP (No Operation). das ebenfalls keine Funktion besitzt.

In C kennzeichnet `NUL` das Ende einer Zeichenkette.

Extended ASCII (8bit)

Die erweiterte ASCII-Version verwendet 8bit (2^8), also 256 Zeichen. Die zusätzlichen Zeichen befinden sich von Position 128 bis Position 255. Die 16 Erweiterungen der ISO-8859 Familie:

ISO/IEC 8859			
8859-1	Latin-1	Westeuropäisch	
8859-2	Latin-2	Mitteleuropäisch	Tschechisch, Ungarisch, Polnisch, Kroatisch, Rumänisch, Slovenisch, Serbisch (latin)
8859-3	Latin-3	Südeuropäisch	Maltesisch, Esperanto, Türkisch (vor der Einführung von Latin5)
8859-4	Latin-4	Nordeuropäisch	Estnisch, Lettisch, Littauisch, Grönländisch, Samisch (Lappisch)
8859-5		Kyrillisch	Bulgarisch, Russisch, Serbisch, Mazedonisch, Ukrainisch
8859-6		Arabisch	
8859-7		Griechisch	
8859-8		Hebräisch	
8859-9	Latin 5	Türkisch	Inuit, Isländisch, Tükisch
8859-10	Latin 6	Nordisch	
8859-11		Thai	
8859-12		nie verabschiedet	Indisch
8859-13	Latin 7	Baltisch	Litauisch, Lettisch
8859-14	Latin 8	Keltisch	Gällisch, Walisisch
8859-15	Latin 9	Westeuropäisch	(Codename Latin0)
8859-16	Latin 10	Südosteuropäisch	

Die deutschen Umlaute und ß sind in jeder Latin-Version enthalten.
ISO-8859-1 und UTF-8 sind in den ersten 256 Zeichen identisch.
Laut HTML5 soll ein ISO-8859-1 Dokument als Win1252 interpretiert werden.

No Break Space (NBSP), Soft Hyphen (SHY)

Mit ISO-8859-1 wurde auf Position 160 **No Break Space (NBSP)** eingeführt (erzwungenes Leerzeichen) und **Soft-Hyphen (SHY)** auf Position 255 eingeführt. Sie werden manchmal zu den Control Characters dazugezählt.

NBSP erzwingt ein Leerzeichen (an dieser Stelle darf die Zeile nicht umgebrochen werden). **Html-Entity**:
Taste: Ctrl+Shift+SPACE

SHY ist ein weiches Trennzeichen. Es kennzeichnet eine optimale Position (Sollbruchstelle) für einen eventuell notwendigen Zeilenumbruch. Falls das Wort getrennt wird, erscheint ein Trennstrich. Wird das Wort nicht getrennt, bleibt das weiche Trennzeichen für den Menschen unsichtbar. **Html-Entity**:

Taste: AltGr+- Taste bei CP850: Alt+240 Taste bei CP1252: Alt+0173

Unterschiede zwischen ISO-8859-1 (Latin1) und ISO-8859-15 (Latin9)

Acht Zeichen wurden durch neue ersetzt:

Dec	Hex	ISO-8859-1		ISO-8859-15
164	A4	¤	Currency Sign	€ Euro Currency Sign
166	A6	¦	Broken Bar	Š Latin Capital Letter S With Caron
168	A8	¨	Diaeresis	š Latin Small Letter S With Caron
180	B4	´	Acute Accent	Ž Latin Capital Letter Z With Caron
184	B8	¸	Cedilla	ž Latin Small Letter Z With Caron
188	BC	¼	Vulgar Fraction One Quarter	Œ Latin Capital Ligature OE
189	BD	½	Vulgar Fraction One Half	œ Latin Small Ligature OE
190	Be	¾	Vulgar Fraction Three Quarters	Ÿ Latin Capital Letter Y With Diaeresis

Durch das Hinzufügen von französischen Zeichen ist ISO-8859-15 häufig auf UNIX-Systemen in Frankreich zu finden.

ISO-8859-1 (Latin1), ISO-8859-15 (Latin9)

Laut HTML5 soll ein ISO-8859-1 (Latin1) Dokument als Win-1252 (CP-1252) interpretiert werden. Win-1252 hat die Steuerzeichen (128-159)durch druckbare Zeichen ersetzt. Siehe Win1252-Zeichensatztabelle Seite 241.
ISO-8859-1 und UTF-8 sind in den ersten 256 Zeichen identisch.

ISO-8859-1

DEC	Symbol	HTML-Entity		
160				no break space
161	¡	¡		inverted exclamation mark
162	¢	¢		cent
163	£	£		pound
164	¤	¤		currency
165	¥	¥		yen
166	¦	¦	Unterbrochener senkrechter Strich	broken vertical bar
167	§	§		section
168	¨	¨		spacing diaeresis
169	©	©		copyright
170	ª	ª		feminine ordinal indicator
171	«	«	Deutsch elegant öffnend	angle quotation mark (left)
172	¬	¬		negation
173				soft hyphen
174	®	®		registered trademark
175	¯	¯		spacing macron
176	°	°		degree
177	±	±	Ordinalzeichen (feminin)	plus-or-minus
178	²	²		superscript 2
179	³	³		superscript 3
180	´	´		spacing acute
181	µ	µ		micro
182	¶	¶		paragraph
183	·	·		middle dot
184	¸	¸		spacing cedilla
185	¹	¹		superscript 1
186	º	º	Ordinalzeichen (maskulin)	masculine ordinal indicator
187	»	»	Deutsch elegant öffnend	angle quotation mark (right)
188	¼	¼		fraction ¼
189	½	½		fraction ½
190	¾	¾		fraction ¾
191	¿	¿		inverted question mark
192	À	À		capital a, grave accent
193	Á	Á		capital a, acute accent
194	Â	Â		capital a, circumflex accent
195	Ã	Ã		capital a, tilde
196	Ä	Ä		capital a, umlaut mark
197	Å	Å		capital a, ring
198	Æ	Æ		capital ae
199	Ç	Ç		capital c, cedilla
200	È	È		capital e, grave accent
201	É	É		capital e, acute accent
202	Ê	Ê		capital e, circumflex accent
203	Ë	Ë		capital e, umlaut mark
204	Ì	Ì		capital i, grave accent
205	Í	Í		capital i, acute accent
206	Î	Î		capital i, circumflex accent
207	Ï	Ï		capital i, umlaut mark
208	Ð	ETH;		capital eth, Icelandic
209	Ñ	Ñ		capital n, tilde
210	Ò	Ò		capital o, grave accent
211	Ó	Ó		capital o, acute accent
212	Ô	Ô		capital o, circumflex accent

ISO-8859-15

€	€	
Š	Capital Letter S With Caron	
š	Small Letter S With Caron	
Ž	Capital Letter Z With Caron	
ž	Small Letter Z With Caron	
Œ	Capital Ligature OE	Œ
œ	Small Ligature OE	œ
Ÿ	Capital Letter Y Diaeresis	Ÿ

201	Õ	Õ		capital o, tilde
214	Ö	Ö		capital o, umlaut mark
215	×	×	Malzeichen	multiplication
216	Ø	Ø		capital o, slash
217	Ù	Ù		capital u, grave accent
218	Ú	Ú		capital u, acute accent
219	Û	Û		capital u, circumflex accent
220	Ü	Ü		capital u, umlaut mark
221	Ý	Ý		capital y, acute accent
222	Þ	Þ		capital THORN, Icelandic
223	ß	ß		small sharp s, German
224	à	à		small a, grave accent
225	á	á		small a, acute accent
226	â	â		small a, circumflex accent
227	ã	ã		small a, tilde
228	ä	ä		small a, umlaut mark
229	å	å		small a, ring
230	æ	æ		small ae
231	ç	ç		small c, cedilla
232	è	è		small e, grave accent
233	é	é		small e, acute accent
234	ê	ê		small e, circumflex accent
235	ë	euml;		small e, umlaut mark
236	ì	ì		small i, grave accent
237	í	í		small i, acute accent
238	î	î		small i, circumflex accent
239	ï	ï		small i, umlaut mark
240	ð	ð		small eth, Icelandic
241	ñ	ñ		small n, tilde
242	ò	ò		small o, grave accent
243	ó	ó		small o, acute accent
244	ô	ô		small o, circumflex accent
245	õ	õ		small o, tilde
246	ö	ö		small o, umlaut mark
247	÷	÷	Geteiltzeichen	division
248	ø	ø		small o, slash
248	ù	ù		small u, grave accent
250	ú	ú		small u, acute accent
251	û	û		small u, circumflex accent
252	ü	ü		small u, umlaut mark
253	ý	yacute;		small y, acute accent
254	þ	þ		small thorn, Icelandic
255	ÿ	ÿ		small y, umlaut mark

Double-Byte Character Set (DBCS)

Das Double-Byte Character Set ist eine Zeichensatzkodierung (character endcoding), in der jedes Zeichen (inklusive Steuerzeichen) aus 2 Bytes besteht. Als Beispiel das koreanische KS X 1001.

DBCS ist ein **Fixed-Width Encoding** Format (im Gegensat zu zu **Variable-width Encoding** Formaten wie UTF-8, Shift JIS, GB2312, Big5, EUC-JP und EUC-KR.

Variable-width Encoding Format mit unterschiedlicher Byteanzahl wird Multibyte-Encoding genannt. Diese charsets werden Multibyte Character Sets (MBCS) genannt.

Unicode

Ab 1987 begannen Joseph D. Becker (Xerox), Lee Collins (Xerox) und Mark Davis (Apple) mit der Entwicklung von Unicode. Der Begriff Unicode wurde von Becker geprägt und zum ersten Mal 1988 in seinem Artikel Unicode88 erwähnt. 1991 wurde mit Version 1.0 die erste offizielle Version veröffentlicht. Bis zum Jahr 1995 war Unicode ein 16bit **Fixed-width Encoding** Format.

Mit Version 2.0 aus dem Jahr 1996 wurde **Unicode Transformation Format** (UTF) eingeführt. Und damit auch ein Variable-width Encoding (Multibyte Encoding).

UTF-8 hat ein **Variable-width Encoding** (Multibyte Encoding): Es benötigt pro Zeichen zwischen 1 und 4 Bytes.
UTF-16 benötigt zwei oder vier Bytes und UTF-32 benötigt immer 4 Bytes. UTF-32 hat somit ein **Fixed-width Encoding.**

Glype

Die Glyphe einer Schriftart (und nicht der Zeichensatz), bestimmt das konkrete Aussehen des Symbols. Für ein Zeichen gibt es meist mehrere Glyphen.

Das Zeichen (Character) a:

hat zB folgende Glyphen:
Diese Glyphen haben keinen eigenen **Unicode Code Point.**

Glypen-Merge

Mehrere Zeichen (Character Sequence) können zu einer Glyphe zusammengefasst werden.

Die Zeichenkette fi

als eine einzige Glyphe:
Diese Glyph hat ihren eigenen **Unicode Code Point.**

Kontextabhängige Glyphen

Im Arabischen können Glyphen kontextabhängig sein und völlig unterschiedlich aussehen.

Dieses arabischen Zeichen:

hat diese kontextabhängige Glyphen:

Jede dieser Glyphen hat jeweils ihren eigenen **Unicode Code Point.**

Kontextabhängiger Glyph-Split

In Tamil muss der Glyph von einem Character kontextabhängig in 2 Teile getrennt werden.

Dieser Character in der Sprache Tamil:

hat kontextabhängig einen geteilten Glyphen:

Die beiden Glyphen haben jeweils ihren eigenen **Unicode Code Point.**

Control Characters

The Unicode Control Characters decken ab:

U+0000—U+001F	C0 controls	Siehe Seite 226
U+007F	delete	
U+0080—U+009F	C1 controls	Siehe Seite 227

UTF Encoding

Bis zum Jahr 1995 war Unicode ein 16bit **Fixed-width Encoding** Format. Mit Version 2.0 aus dem Jahr 1996 wurde **Unicode Transformation Format** (UTF) eingeführt. Und damit auch ein Variable-width Encoding (Multibyte Encoding).

UTF-8 hat ein **Variable-width Encoding** (Multibyte Encoding): Es benötigt pro Zeichen zwischen 1 und 4 Bytes.
UTF-16 benötigt zwei oder vier Bytes und UTF-32 benötigt immer 4 Bytes.
UTF-32 hat somit ein Fixed-width Encoding**:**

UTF-8	1, 2, 3, 4	Bytes	Variable-width Encoding (Multibyte Encoding)
UTF-16	2, 4	Bytes	Variable-width Encoding (Multibyte Encoding)
UTF-32	4	Bytes	Fixed-width Encoding

Siehe auch Byte-Order (Seite 235) und Byte-Order-Mark (Seite 235).

Unicode codespace

Der offizielle Code Point Bereich liegt von 0 bis 10FFFF16 (1 112 064) und gilt auch für UTF-32. Dadurch ist eine problemlose Konvertierung zwischen UTF-16 und UTF-32 möglich.

UTF-8 Encoding

8-Bit UCS Transformation Formatn (UCS = Universal Character Set):

1. Byte	2. Byte	3. Byte	4. Byte	Anzahl freier Bits	Mögliche Zeichen
0xxxxxxx				7	127
110xxxxx	**10**xxxxxx			(5+6)=11	2 047
1110xxxx	**10**xxxxxx	**10**xxxxxx		(4+6+6)=16	65 535
11110xxx	**1**0xxxxxx	**10**xxxxxx	**10**xxxxxx	(3+6+6+6)=21	1 114 111

Bei jedem einzelnen Byte läßt sich das Format an den Anfang-Bits erkennen (ohne die Nachbar-Bytes zu kennen).
Die Zeichenkette lässt sich somit sowohl von vorne, als auch von hinten lesen und interpretieren.
Anders als zum Beispiel bei GBK, das nur von vorne gelesen werden kann.

Beispiele für eine Konvertierung von 16-bit Unicode nach UTF-8:

16-bit Unicode	UTF-8 Sequence
x0001	01
x007F	7F
x0080	C2 80
07FF	DF BF
0800	E0 A0 80
FFFF	EF BF BF
010000	F0 90 80 80
10FFFF	F4 8F BF BF

Byte-Order (Byte-Reihenfolge)

Die Bytes eines **Code Points** (siehe Seite 223) können von links nach rechts, oder von rechts nach links geschrieben werden. Diese Unterscheidung nennt man Byte-Order (Byte-Reihenfolge).

Als Beispiel der Hexwert `0x44332211` (dezimal 1144201745). Das »little end byte« (the lowest or least significant byte) ist 0x11, das »big end byte« ist 0x44.

Bei **Big-Endian** steht das »big, most significant byte« zu Beginn (großes Ende zuerst):

```
44 33 22 11
```

Im **Little-Endian** Format steht das »little end byte« am Beginn und das »big end byte« am Ende:

```
11 22 33 44
```

Das seltene **Middle Endian** gibt es in zwei Varianten:

```
22 11 44 33
```

Und:

```
33 44 11 22
```

Intel/AMD x86, Altera Nios, Atmel AVR, digital VAX und digital Alpha arbeiten im Little-Endian Format.

IBM Mainframes, Motorola 6800/68k , MIPS, Sun SPARC, System-z, PowerPC, TMS9900, Atmel AVR32 arbeiten im Big-Endian Format.
Einige Rechner lassen sich zwischen little-Endian und Big-Endian umschalten (einige PowerPC, MIPS, ARM, Xscale). Dies wird **Mixed Endian** und manchmal auch **Bi-Endian** genannt.

In TCP/IP wird Big-Endian verwendet.

Im Deutschen werden die Zahlen von 13 bis 99 in little-endian ausgesprochen: »Ein-und-Zwanzig«. Die Eins als weniger wertige Stelle wird zuerst gesprochen.
Die Zahl 1230 wird in Big-Endian ausgesprochen: »Eintausend-zweihundert-dreißig«. Die 1 hat die Wertigkeit 1000, die 2 die Wertigkeit 100 hat und die 3 die Wertigkeit 10.

Danny Cohen prägte den Begriff »Little and Big Endian« 1980. Die Bezeichnungen **Big Endian** und **Little Endian** stammen aus dem Roman »Gullivers Reisen« von Jonathan Swift (1726).
Big-Endian beschreibt eine der beiden politischen Fraktionen, die ihre Eier vom größeren Ende aus schält (der »primitive Weg«). Sie rebellieren gegen den Lilliputanerkönig, da dieser von seinen Untergebenen (die Little Endians) erwartet, dass sie ihre Eier vom kleineren Ende aus öffnen.
Mit diesen Begriffen parodierte Swift den jahrhundertelangen Zwist zwischen der katholischen und der anglikanischen Kirche in Großbritannien.
Cohen verwendete Endian im Aprilscherz »On holy wars and a plea for peace (IEN 137)«.

Byte-Order-Mark (BOM)

Am Beginn eines UTF-Textfiles kann ein Byte-Order-Mark (BOM) gesetzt werden. Er ist zwischen 2 und 4 Byte lang:

`EF BB BF`	UTF-8 Signatur
`FF FE`	UTF-16 BOM, little endian
`FE FF`	UTF-16 BOM, big endian
`FF FE 00 00`	UTF-32 BOM, little endian
`00 00 FE FF`	UTF-32 BOM, big-endian

Einige Textapplikation erkennen am BOM die verwendete Byte-Kodierung des Plaintextfiles.
Siehe Byte-Order Seite 235.

Laut HTML5 ist der BOM des Dokuments höherwertiger, als die Angabe im http-Header. Das ist hilfreich, wenn der Seitenautor keinen Einfluss auf die Servereinstellungen zur Zeichencodierung hat.

Gemäß RFC 2718 und dem Unicode-Standard soll man kein BOM am Seitenanfang verwenden, wenn man die Zeichencodierung per HTTP als "UTF-16LE" oder "UTF-16BE" angibt. Nur wenn für die Seite per HTTP der IANA-Zeichencodierungs-Bezeichner "UTF-16" angegeben ist, ist ein BOM richtig.

UTF-8 Signatur

Bei UTF-8 ist im Gegensatz zu den UTF-16-Codierungen der BOM nicht zwingend erforderlich, denn es gibt nur eine mögliche Reihenfolge der Bytes für ein bestimmtes Zeichen. Diese explizite UTF-8 Kennung wird UTF-8 Signatur gennant.

Zero width no-break space

Der Code Point U+FEFF ist auch das Zeichen zero width no-break space. Ein Leerraum (Space) ohne Breite (Zero Width) und den Text nicht unterbricht (no-Break). Exkurs: **Zero width Space (U+200B)** bietet einen Sollbruchstelle für einen Zeilenumbruch an (wenn dieser unbedingt notwendig ist). Entspricht dem html-Tag <wbr>.

Beispiele

Textfile als ISO-8859-1 (also ohne BOM) mit dem Zeichen ö. Der ASCII Code Point ist 65 (hexadezimal F6).

```
   11110110                                          F6
```

UTF-8 ohne BOM:

```
   11000011 10110110                                 C3 B6
```

UTF-8 mit BOM:

```
   11101111 10111011 10111111 11000011 10110110      EF BB BF C3 B6
```

UTF-16 Little Endian ohne BOM:

```
   11000011 10110110                                 C3 B6
```

UTF-16 Little Endian mit BOM

```
   11111111 11111110 11110110 00000000               FE FF F6 00
```

UTF-16 Big Endian ohne BOM

```
   00000000 11110110                                 00 F6
```

UTF-16 Big Endian mit BOM:

```
   11111110 11111111 00000000 11110110               FE FF 00 F6
```

Whitespace

25 Character haben den Whitespace-Flag auf ja, 6 Character auf nein und 5 Chars stellen Whitespace graphisch dar.

Character mit Whitespace-Flag

Unicode		Dec			Beschreibung
U+0009	char tabulation	9		HT	
U+000A	line feed	10
LF	
U+000B	line tabulation	11		VT	
U+000C	form feed	12		FF	
U+000D	carriage return	13		CR	
U+0020	space	32		SP	Leerzeichen
U+0085	next line	133		NEL	
U+00A0	no-break space	160		NBS	erzwungenes Leerzeichen
U+1680	ogham space mark	5760			Used for interword separation in Ogham text. Normally a vertical line in vertical text or a horizontal line in horizontal text, but may also be a blank space in "stemless" fonts. Requires an Ogham font.
U+2000	en quad	8192			Width of one en. U+2002 is canonically equivalent to this character; U+2002 is preferred.
U+2001	em quad	8193			"mutton quad". Width of one em. U+2003 is canonically equivalent to this character; U+2003 is preferred.
U+2002	en space	8194			"nut". Width of one en. U+2000 En Quad is canonically equivalent to this character; U+2002 is preferred.
U+2003	em space	8195			"mutton". Width of one em. U+2001 Em Quad is canonically equivalent to this character; U+2003 is preferred.
U+2004	three-per-em space	8196			"thick space". One third of an em wide.
U+2005	four-per-em space	8197			"mid space". One fourth of an em wide.
U+2006	six-per-em space	8198			One sixth of an em wide. In computer typography, sometimes equated to U+2009.
U+2007	figure space	8199			Figure space. In fonts with monospaced digits, equal to the width of one digit.
U+2008	punctuation space	8200			As wide as the narrow punctuation in a font, i.e. the advance width of the period or comma.
U+2009	thin space	8201			One-fifth (sometimes one-sixth) of an em wide. Recommended for use as a thousands separator for measures made with SI units. Unlike U+2002 to U+2008, its width may get adjusted in typesetting.[4] Html Entity:
U+200A	hair space	8202			Thinner than a thin space.
U+2028	line separator	8232			
U+2029	paragraph separator	8233			
U+202F	narrow no-break space	8239			Narrow no-break space. Similar in function to U+00A0 No-Break Space. When used with Mongolian, its width is usually one third of the normal space; in other context, its width sometimes resembles that of the Thin Space (U+2009).
U+205F	medium mathematical space	8287			MMSP. Four-eighteenths of an em. In mathematical typography, the widths of spaces are usually given in integral multiples of an eighteenth of an em, and 4/18 em may be used in several situations, for example between the **a** and the **+** and between the **+** and the **b** in **a + b**.
U+3000	ideographic space	12288			As wide as a CJK character cell (fullwidth).

Character ohne Whitespace-Flag

6 Character mit dem Unicode-Flag Whitespace auf nein.

U+180E	mongolian vowel separator	6158	MVS	A narrow space character, used in Mongolian to cause the final two characters of a word to take on different shapes. It is no longer classified as space character (i.e. in Zs category) in Unicode 6.3.0, even though it was in previous versions of the standard.
U+200B	zero width space	8203	ZWSP	zero-width space. Used to indicate word boundaries to text processing systems when using scripts that do not use explicit spacing. It is similar to the soft hyphen, with the difference that the latter is used to indicate syllable boundaries, and should display a visible hyphen when the line breaks at it. Html Entity:
U+200C	zero width non-joiner	8204	ZWNJ	When placed between two characters that would otherwise be connected, a ZWNJ causes them to be printed in their final and initial forms, respectively. Html Entity: ‌
U+200D	zero width joiner	8205	ZWJ	When placed between two characters that would otherwise not be connected, a ZWJ causes them to be printed in their connected forms. Html Entity: ‍
U+2060	word joiner	8288	WJ	Similar to U+200B, but not a point at which a line may be broken. Html Entity:
U+FEFF	zero width non-breaking space	65279		

Leerraum anzeigende Characters

U+00B7	Middle dot	183		·	Interpunct Named entity: ·
U+237D	Shouldered open box	9085		⍽	Used to indicate a NBSP
U+2420	Symbol for space	9248		␠	
U+2422	Blank symbol	9250		␢	
U+2423	Open box	9251		␣	Used in a textbook 1985 on Modula-2, a programming language where space codes require explicit indication. Also used in the keypad silkscreening[n 1] of the Texas Instruments' TI-8x series of graphing calculators. Named entity: ␣

Zeichensatztabellen

Kapitel 8

Zeichensatztabellen (Code Pages)

ISO-8859-1 (Latin1)

1	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0x0	*nul*	*stx*	*sot*	*etx*	*eot*	*enq*	*ack*	*bel*	*bs*	*ht*	*lf*	*vt*	*ff*	*cr*	*so*	*si*
0x1	*dle*	*dc1*	*dc2*	*dc3*	*dc4*	*nak*	*syn*	*etb*	*can*	*em*	*sub*	*esc*	*fs*	*gs*	*rs*	*us*
0x2	sp	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
0x3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0x4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0x5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0x6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0x7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	*del*
0x8	*pad*	*hop*	*bph*	*nbh*	*ind*	*nel*	*ssa*	*esa*	*hts*	*htj*	*vts*	*pld*	*plu*	*ri*	*ss2*	*ss3*
0x9	*dcs*	*pu1*	*pu2*	*sts*	*cch*	*mw*	*spa*	*epa*	*sos*	*sgci*	*sci*	*csi*	*st*	*osc*	*pm*	*apc*
0xa	*nbsp*	¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	*shy*	®	¯
0xb	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
0xc	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
0xd	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
0xe	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
0xf	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

ISO-8859-15 (Latin9)

9	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0x0	*nul*	*stx*	*sot*	*etx*	*eot*	*enq*	*ack*	*bel*	*bs*	*ht*	*lf*	*vt*	*ff*	*cr*	*so*	*si*
0x1	*dle*	*dc1*	*dc2*	*dc3*	*dc4*	*nak*	*syn*	*etb*	*can*	*em*	*sub*	*esc*	*fs*	*gs*	*rs*	*us*
0x2	sp	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
0x3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0x4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0x5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0x6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0x7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	*del*
0x8	*pad*	*hop*	*bph*	*nbh*	*ind*	*nel*	*ssa*	*esa*	*hts*	*htj*	*vts*	*pld*	*plu*	*ri*	*ss2*	*ss3*
0x9	*dcs*	*pu1*	*pu2*	*sts*	*cch*	*mw*	*spa*	*epa*	*sos*	*sgci*	*sci*	*csi*	*st*	*osc*	*pm*	*apc*
0xa	*nbsp*	¡	¢	£	€	¥	Š	§	š	©	ª	«	¬	*shy*	®	¯
0xb	°	±	²	³	Ž	µ	¶	·	ž	¹	º	»	Œ	œ	Ÿ	¿
0xc	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
0xd	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
0xe	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
0xf	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

DEC Multilingual Character Set

D	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0x0	*nul*	*stx*	*sot*	*etx*	*eot*	*enq*	*ack*	*bel*	*bs*	*ht*	*lf*	*vt*	*ff*	*cr*	*so*	*si*
0x1	*dle*	*dc1*	*dc2*	*dc3*	*dc4*	*nak*	*syn*	*etb*	*can*	*em*	*sub*	*esc*	*fs*	*gs*	*rs*	*us*
0x2	sp	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
0x3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0x4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0x5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0x6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0x7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	*del*
0x8																
0x9																
0xa		¡	¢	£		¥		§	¤	©	ª	«				
0xb	°	±	²	³		µ	¶	·		¹	º	»	¼	½		¿
0xc	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
0xd		Ñ	Ò	Ó	Ô	Õ	Ö	Œ	Ø	Ù	Ú	Û	Ü	Ÿ		ß
0xe	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
0xf		ñ	ò	ó	ô	õ	ö	œ	ø	ù	ú	û	ü	ÿ		

Win-1252 (Microsoft Windows Western European)

#	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0x0	*nul*	*stx*	*sot*	*etx*	*eot*	*enq*	*ack*	*bel*	*bs*	*ht*	*lf*	*vt*	*ff*	*cr*	*so*	*si*
0x1	*dle*	*dc1*	*dc2*	*dc3*	*dc4*	*nak*	*syn*	*etb*	*can*	*em*	*sub*	*esc*	*fs*	*gs*	*rs*	*us*
0x2	sp	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
0x3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0x4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0x5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0x6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0x7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	*del*
0x8	€		,	ƒ	„	…	†	‡	ˆ	‰	Š	‹	Œ		Ž	
0x9		‘	’	“	”	•	–	—	˜	™	š	›	œ		ž	Ÿ
0xa	*nbsp*	¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	*shy*	®	¯
0xb	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
0xc	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
0xd	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
0xe	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
0xf	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Codepage 437 (IBM-PC default)

*	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0x0	*nul*	*stx*	*sot*	*etx*	*eot*	*enq*	*ack*	*bel*	*bs*	*ht*	*lf*	*vt*	*ff*	*cr*	*so*	*si*
0x1	*dle*	*dc1*	*dc2*	*dc3*	*dc4*	*nak*	*syn*	*etb*	*can*	*em*	*sub*	*esc*	*fs*	*gs*	*rs*	*us*
0x2	sp	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
0x3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0x4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0x5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0x6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0x7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	*del*
0x8	Ç	ü	é	â	ä	à	å	ç	ê	ë	è	ï	î	ì	Ä	Å
0x9	É	æ	Æ	ô	ö	ò	û	ù	ÿ	Ö	Ü	¢	£	¥	₧	ƒ
0xa	á	í	ó	ú	ñ	Ñ	ª	º	¿	⌐	¬	½	¼	¡	«	»
0xb	░	▒	▓	│	┤	╡	╢	╖	╕	╣	║	╗	╝	╜	╛	┐
0xc	└	┴	┬	├	─	┼	╞	╟	╚	╔	╩	╦	╠	═	╬	╧
0xd	╨	╤	╥	╙	╘	╒	╓	╫	╪	┘	┌	█	▄	▌	▐	▀
0xe	α	ß	Γ	π	Σ	σ	µ	τ	Φ	Θ	Ω	δ	∞	φ	ε	∩
0xf	≡	±	≥	≤	⌠	⌡	÷	≈	°	∙	·	√	ⁿ	²	■	*nbsp*

Codepage 850 (IBM-PC: European)

*w	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0x0	*nul*	*stx*	*sot*	*etx*	*eot*	*enq*	*ack*	*bel*	*bs*	*ht*	*lf*	*vt*	*ff*	*cr*	*so*	*si*
0x1	*dle*	*dc1*	*dc2*	*dc3*	*dc4*	*nak*	*syn*	*etb*	*can*	*em*	*sub*	*esc*	*fs*	*gs*	*rs*	*us*
0x2	sp	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
0x3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0x4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0x5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0x6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0x7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	*del*
0x8	Ç	ü	é	â	ä	à	å	ç	ê	ë	è	ï	î	ì	Ä	Å
0x9	É	æ	Æ	ô	ö	ò	û	ù	ÿ	Ö	Ü	ø	£	Ø	×	ƒ
0xa	á	í	ó	ú	ñ	Ñ	ª	º	¿	®	¬	½	¼	¡	«	»
0xb	░	▒	▓	│	┤	Á	Â	À	©	╣	║	╗	╝	¢	¥	┐
0xc	└	┴	┬	├	─	┼	ã	Ã	╚	╔	╩	╦	╠	═	╬	¤
0xd	ð	Ð	Ê	Ë	È	ı	Í	Î	Ï	┘	┌	█	▄	¦	Ì	▀
0xe	Ó	ß	Ô	Ò	õ	Õ	µ	þ	Þ	Ú	Û	Ù	ý	Ý	¯	´
0xf	*shy*	±	‗	¾	¶	§	÷	¸	°	¨	·	¹	³	²	■	*nbsp*

SVG

Kapitel 9

Scalable Vektor Graphics (SVG)

Scalable Vektor Graphics (SVG) ist eine xml-basierende Auszeichnungssprache. Sie beschreibt zweidimensionale, skalierbare Grafiken. gzip komprimierte SVG-Dateien haben die Dateiendung `gsvg`. Mime-Type ist `image/svg+xml`

Header

Der Header besteht aus der xml-Deklaration und dem doctype. Ab SVG1.2 fällt doctype weg.

xml

Der xml-Header:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
```

Siehe auch Byte-Order (Seite 235), Byte-Order-Mark (Seite 235) und UTF-Encoding (Seite 234).

doctype

Der `doctype` für SVG1.0:

```
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN" "http://www.w3.org/TR/SVG/DTD/svg10.dtd">
```

Der `doctype` für SVG1.1:

```
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN" "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
```

Für SVG1.2 hat W3 kein dtd veröffentlicht. Die SVG-Version wird mit dem Attribut `version` deklariert.

<svg>

Das Element `svg` ist ein Gruppierungselement, dass den eigentlichen Dokumenteninhalt markiert. In tiny-Profilen darf es nur ein svg-Element im Dokument geben.
In der Vollversion von SVG1.1 dürfen im root-svg weitere svg-Elemente eingebettet sein.

Painter's-Modell

Gerendert wird nach dem Painter's-Modell: Jedes Objekt wird der Reihe nach aufgetragen (gemalt). Nachfolgende Objekte werden über bereits bestehende Objekte gezeichnet. Die Farbe muss nicht deckend sein, die Transparenz wird durch die Eigenschaft `opacity` (siehe Seite 251) bestimmt.

Attribute (ohne viewPort)

`xmlns` Attributwert: `"http://www.w3.org/2000/svg"`

`version` Verwendete SVG-Version. Mögliche Attribut-Werte:

- `x1.0`
- `x1.1`
- `x1.2`

`baseProfile` Mögliche Attributwerte:

- `tiny` Attributwert ist nur möglich bei SVG-Version 1.1 und 1.2
  Tiny deckt einen Teilbereich von SVG ab, der für leistungsschwache Hardware geeignet ist
- `basic` Attributwert ist nur möglich bei SVG-Version 1.1
- `full` voller SVG-Funktionsumfang

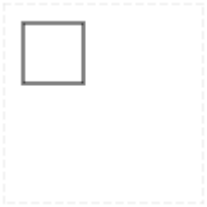
viewPort

Das Element svg definiert auch die Breite (`width`) und die Höhe (`height`) einer Zeichenfläche. Objekte, die über die Leinwand (canvas) hinausragen, werden abgeschnitten. Default von `width` und `height` sind `100%`.

Beispiel für einen viewPort mit der Breite `100px` und Höhe `100px`:

```
    <svg width="100" height="100" >
    <rect x="0" y="0" width="100" height="100"  stroke-dasharray="4 2"  stroke-width="1"
stroke="#e2e2e2" fill="none"/>
    <path  fill="none" stroke="black" d="M10,10 L 40,10 40,40 10,40 Z"/>
    </svg>
```

Ergibt als Output:



Beispiel mit zu kleinem viewPort (Breite `30`):

```
    <svg width="30" height="100" >
     <rect x="0" y="0" width="100" height="100"  stroke-dasharray="4 2"  stroke-width="1"
    stroke="#e2e2e2" fill="none"/>
     <path  fill="none" stroke="black" d="M10,10 L 40,10 40,40 10,40 Z"/>
    </svg>
```

Ergibt als Output:

Innerhalb von einem `<svg>` Element lassen sich weitere svg-Elemente einbetten (verschachteln).

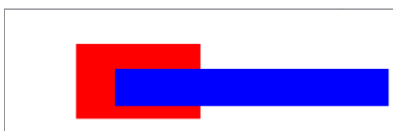
Ihr Ursprung ist dann relativ zum Elternelement.

Das baseProfile `tiny` erlaubt keine Verschachtelung.

Beispiel:

```
    <svg  width="900" height="900">
    <rect x="50" y="20" width="100" height="60" fill="red"/>

     <svg x="80" y="10" width="300" height="900">
       <rect x="1" y="30" width="220" height="30" fill="blue" />
     </svg>
    </svg>
```

Ergibt als Output:



Ohne Verschiebung von x 80 und y 10:

Die Positionsangaben haben keine Auswirkung, wenn das svg Element das Wurzelelement (root) ist.

Das svg Element mit x,y-Angaben muss ein Nachfahre von einem svg (Eltern-)Element sein.

Die Attribute x und y sind animierbar.

viewBox

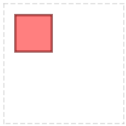
Das Attribut `viewBox` bestimmt einen Ausschnitt der Leinwand. Attributwert ist ein String.
Im String befindet sich entweder der Wert `none`, oder 4 Werte (durch Leerzeichen getrennt).

Die vier Attributwerte in folgender Reihenfolge:

`x`	x-Koordinate. Minuswert verschiebt nach links, Pluswert nach rechts
`y`	y-Koordinate. Minuswert verschiebt nach oben, Pluswert nach unten
`width`	Breite
`heigth`	Höhe

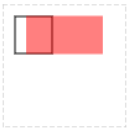
Ist die Breite oder Höhe unterschiedlich zum viewPort, wird ein benutzerdefiniertes Koordinatensystem erzeugt und automatisch auf die viewPort-Werte skaliert.

Beispiel 1: Breite und Höhe sind identisch mit dem viewPort. Es findet keine Skalierung statt.



```
<svg width="100" height="100" viewBox="0 0 100 100">
   <svg width="100" height="100" viewBox="0 0 100 100">
```

Beispiel 2: viewBox-Breite ist kleiner als die viewPort-Breite. Wenn das Attribut `preserveAspectRatio` den Wert `none` hat, wird die ViewBox der Breite nach gestreckt.



```
<svg width="100" height="100" viewBox="0 0 100 100">
   <svg preserveAspectRatio="none" width="100" height="100" viewBox="0 0 50 100">
```

Beispiel 3: viewBox-Breite ist größer als die viewPort-Breite. Wenn das Attribut `preserveAspectRatio` den Wert `none` hat, wird die ViewBox der Breite nach gestaucht.

```
<svg width="100" height="100" viewBox="0 0 100 100">
   <svg preserveAspectRatio="none" width="100" height="100" viewBox="0 0 150 100">
```

preserveAspectRatio

Das Attribut `preserveAspectRatio` bestimmt:

1. ob bei nur einer Wert-Abweichung (Breite oder Höhe) vom viewPort,
   das Seitenverhältnis trotzdem bewahrt werden soll. Attributwert **none** erlaubt Stauchen (stretch) und Strecken (shrink)
2. die Ausrichtung (alignment) der Zeichnung (x,y) innerhalb des viewBox-Fensters (nur in Kombination mit `clip` oder meet)
3. ob bei einem zu kleinen Fenster die Zeichnungen abgeschnitten werden soll (**clip**) oder ob die Größe der Zeichnung automatisch angespasst werden soll (**meet**). Nur in Kombination mit x,y.

Ohne Verwendung von `preserveAspectRatio` wird das gleiche Seitenverhältnis erzwungen. Die Verwendung von `none` schaltet das **uniform scaling** ab:

```
preserveAspectRatio = 'none'
```

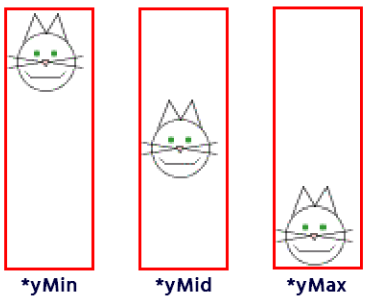
Von `none` abgesehen, ist der Attributwert von `preserveAspectRatio` ein String, der 2 Parameter enthält (durch Leerzeichen getrennt):

```
preserveAspectRatio = "Alignment MeetOrClip"
```

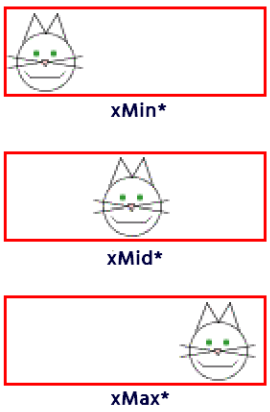
Alignment besteht aus der Kombination von **x** mit **Min**/**Mid**/**Max** und **y** mit **Min**/**Mid**/**Max**

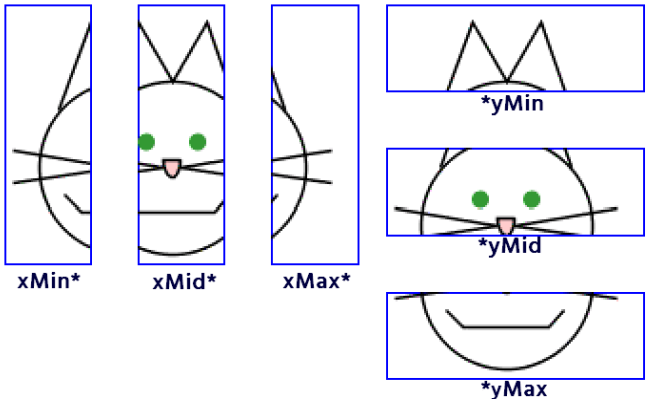
MeetOrClip ist entweder **meet** (auto-fit) oder **clip** (wenn notwendig, Zeichnung abschneiden)

Bei meet:



Bei clip:

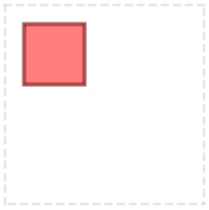




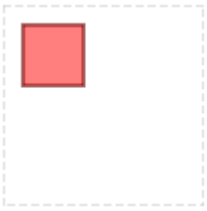
viewBox Beispiele

```
<svg width="100" height="100">
       <path  fill="none" stroke="black" d="M10,10 L 40,10 40,40 10,40 Z"/>
       <svg  width="100" height="100" viewBox="0 0 100 100">
       <path fill="red" stroke="red" d="M10,10 L 40,10 40,40 10,40 Z" opacity="0.5" />
 </svg>
```

viewBox="0 0 50 100"

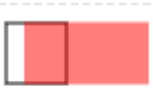


mit preserveAspectRatio="none"



viewBox="0 0 50 100"

mit preserveAspectRatio="none"

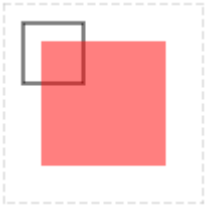


viewBox="0 0 100 50"

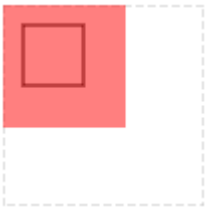
mit preserveAspectRatio="none"

viewBox="0 0 50 50"

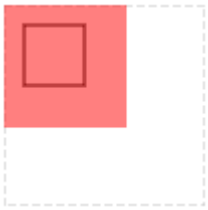
mit preserveAspectRatio="none"



viewBox="10 10 50 50"



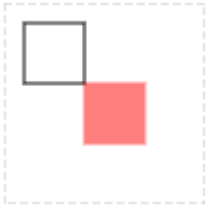
mit preserveAspectRatio="none"



viewBox="10 10 100 100"

viewBox="-10 10 100 100"

viewBox="-30 -30 100 100"



Koordinatensystem

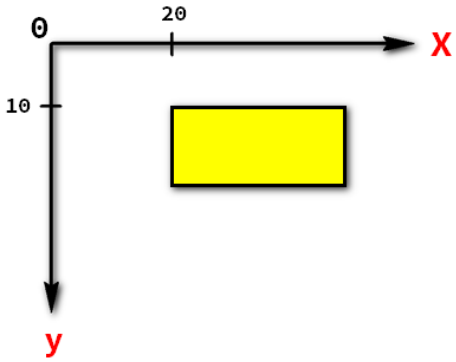
Die linke obere Ecke ist der Ursprung. Die x-Achse verläuft nach rechts, die y-Achse nach unten.

Default-Größeneinheit ist Pixel (siehe Maßeinheiten Seite 249).

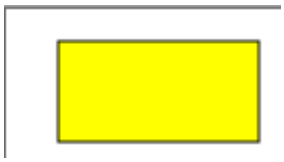
Das Koordinatensystem wird durch Transformation (Seite 274), viewPort (Seite 245) und viewBox (Seite 246) verändert.

Beispiel:

```
<svg width="300" height="200">
    <rect x="20" y="10" width="100" height="50"
    style="fill:yellow;stroke:black" />
</svg>
```



Ergibt als Output:



Maßeinheiten

SVG kennt relative und absolute Größenangaben. Default ist Pixel.

relative Einheiten

px	relativ zur Bildschirmauflösung und benutzerdefinierten Koordinatensystem
em	relativ zur gegenwärtig verwendeten Schriftgröße (meistens Buchstabe a)
ex	relativ zur Höhe des Buchstaben x der verwendeten Schrift
%	Prozentwert relativ zum benutzerdefinierten Koordinatensystem

absolute Einheiten

in	inch (1 inch = 2.54 cm)
cm	Zentimeter
mm	Millimeter
pt	Points (72. Teil von einem Inch)
pc	Pica (1 Pica = 1/6 inch)

Shape Elements (Formen)

SVG kennt 6 Form-Elemente:

`circle`	Kreis, Seite 254
`line`	Linie, Seite 255
`elipse`	Elipse, Seite 255
`rectangle`	Rechteck, Seite 254
`polyline`	Mehrfachlinie, Seite 256
`polygon`	Mehrfachlinie, Seite 256
`path`	Pfad mit Mehrfachanweisungen (Linien und Kurven). Seite 257

- `L` Linie, Seite 258
- `H` horizontale Linie, Seite 258
- `V` vertikale Linie, Seite 258
- `Z` zurück zum Startpunkt, Seite 259
- `Q` quadratische Bézierkurve (T bei einer 2. folgenden Bézierkurve), Seite 259
- `C` kubische Bézierkurve (S bei einer 2. folgenden Bézierkurve), Seite 260
- `A` elliptische Kurve, Seite 262

Drei Füll-Attribute:

`fill`	Farbwert, Seite 250
`fill-opacity`	Transparenz, Seite 251
`fill-rule`	Füllverhalten bei Kontur-Überschneidung, Seite 251

Sieben Kontur-Attribute:

`stroke`	Linienfarbe der Kontur, Seite 252
`stroke-dasharray`	Linienmuster, Seite 252
`stroke-linecap`	Linienende, Seite 252
`stroke-opacity`	Linientransparenz, Seite 252
`stroke-linejoin`	Linienbiegung, Seite 252
`stroke-miterlimit`	Linienbiegung miter limit, Seite 253
`stroke-width`	Liniendicke der Kontur (Pfad liegt in der Mitte), Seite 252

Die Attribute können auch als CSS definiert werden (Seite 268).

Füll-Attribute

Attribut `fill` für die Farbe, `fill-opacity` für die Transparenz und `fill-rule` für das Füllverhalten bei Konturüberschneidung.

fill

Attributwert vom Attribut `fill` ist ein Farb-Keyword oder ein RGB-Wert (siehe Seite 212). Für die Füllung werden Start- und Endpunkt verbunden (somit werden auch Formen gefüllt, die nicht mit `stroke` geschlossen wurden).
Beispiel:

```
<polyline points="0,20 20,20 20,40" fill="blue" stroke="red" stroke-width="3" />
```

Ergibt als Output:

fill-opacity

Der Attributwert bestimmt die Transparenz der Füllung. Datentyp ist float.

Wertebereich geht von 0 bis 1.

Bei 0 ist die Füllung völlig transparent, bei 0.5 zu 50% transparent und bei 1 deckend.

Beispiel:

```
<ellipse cx="40" cy="40" rx="30" ry="20" style="fill: fuchsia; opacity:1"" />
<ellipse cx="60" cy="40" rx="30" ry="20" style="fill: blue; opacity:0.5" />
<ellipse cx="50" cy="60" rx="30" ry="20" style="fill: lime; opacity:0.5" />

<circle cx="150" cy="40" r="10" style="fill: fuchsia;" />
<circle cx="180" cy="40" r="10" style="fill: blue;" />
<circle cx="165" cy="60" r="10" style="fill: lime;" />
```

Ergibt als Output:

fill-rule

Der Attributwert bestimmt das Füll-Verhalten, wenn sich die Formpfade des Elements kreuzen

nonzero

Nimm einen beliebigen Punkt in der Füllfläche. Denke dir von diesem Punkt einen Strahl (Gerade) in eine beliebige Richtung. Trifft der Strahl auf eine Konturlinie, wird überprüft, von welcher Richtung die Konturlinie den Strahl kreuzt. Bei links nach rechts, addiere um 1. Bei rechts nach links, subtrahiere um 1.
Der Strahl darf nicht durch Schittpunkte gehen.

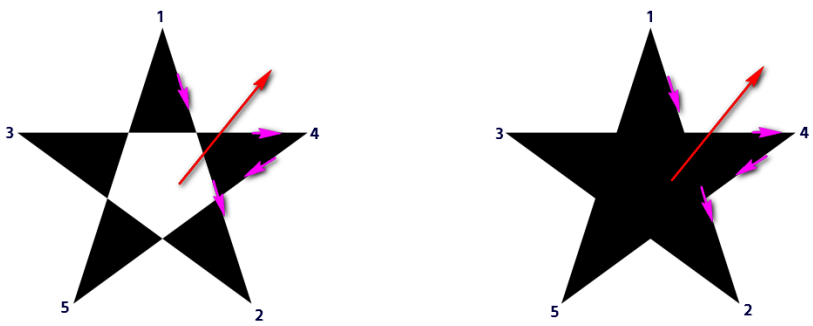
evenodd

Nimm einen beliebigen Punkt in der Füllfläche. Denke dir von diesem Punkt einen Strahl (Gerade) in eine beliebige Richtung. Trifft der Strahl auf eine Konturlinie, addiere um 1. Ist die Zahl gerade, dann befindet sich der Punkt ausserhalb der Fläche.

Beispiel:

```
<svg>
<path d="M 250,75 L 323,301 131,161 369,161 177,301 z" fill-rule="evenodd" />
 <svg x="400">
    <path d="M 250,75 L 323,301 131,161 369,161 177,301 z" fill-rule="nonzero" />
 </svg>
</svg>
```

Ergibt als Output:



Stroke-Attribute (Konturlinie)

stroke Attributwert ist ein Farb-Keyword oder ein RGB-Wert (siehe RGB-Werte Seite 212).

stroke-width Liniendicke (Datentyp ist Integer).

Default-Wert ist 1.

Stroke verläuft mittig vom Pfad.

stroke-dasharray Attributwert besteht aus einer Sequenz (String) von Zahlenpaaren:

Linienlänge-Pixel Leerabstand-Pixel

Der String bestimmt das Muster für gestrichelte/gepunktete Linien.

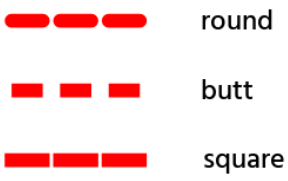
Der Wert **5** 3 **1** 3 bedeutet:

5 Pixel Strich, 3 Pixel Abstand, **1** Pixel Strich, 3 Pixel Abstand.

Beispiel: 9 Pixel Linie, 5 Pixel Leerabstand:

```
stroke-dasharray: 9 5;
```

Ergibt als Output:

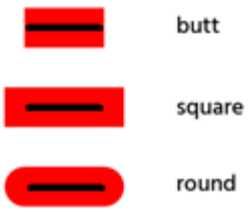


stroke-linecap Form des Linienendes (schwarz ist der Pfad).

Bei `butt` wird das Linienende nicht verlängert.

Bei `round` wird das Ende um die halbe Strichbreite erweitert und abgerundet.

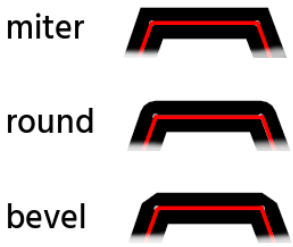
Bei `square` wird das Ende um die halbe Strichbreite erweitert.



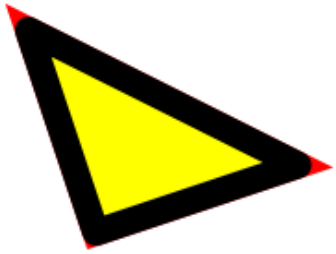
stroke-opacity Transparenz der Linie. Datentyp ist float. Wertebereich von 0 bis 1.

`0` ist völlig transparent, `1` ist deckend (default).

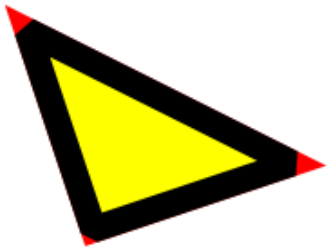
stroke-linejoin Form vom Polygon Bend. Drei mögliche Werte: `miter`, `round` und `bevel`.



Vergleich `miter` (rot) und `round` (black):



Vergleich `miter` (rot) und `bevel` (black):



stroke-miterlimit Max ratio of miter point to line thickness (Datentyp ist Integer). Default ist 4.

<rectangle>

Das Element `rectangle` zeichnet ein Rechteck.

Attribute:

x	x-Koordinate
y	x-Koordinate
width	Breite
heigth	Höhe
rx	Ecke (optional)
ry	Ecke (optional)

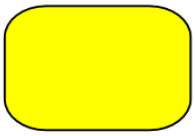
`stroke`-Attribute siehe Seite 252.

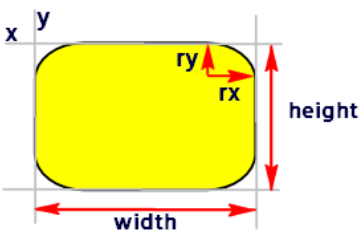
`marker`-Attribute siehe Seite 271

Beispiel:

```
<rectangle x="100" y="10" width="90"  height="60" rx="20" ry="15" stroke="black" fill="yellow" />
```

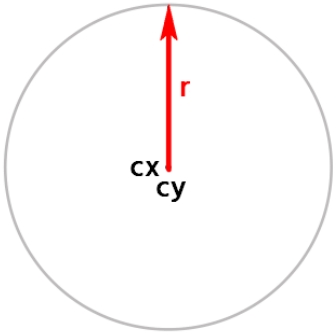
Erzeugt als Output:





<circle>

Das Element `circle` zeichnet einen Kreis.



Attribute:

`cx`	x-Koordinate (default ist Pixel)
`cy`	y-Koordinate
`r`	Radius

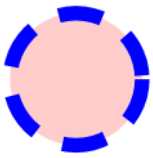
`stroke`-Attribute siehe Seite 252.

`marker`-Attribute siehe Seite 271

Beispiel:

```
<circle cx="10" cy="5" r="9" fill="red" stroke-dasharray="6 4"
   stroke="blue" fill-opacity="0.2" stroke-width="2" />

<circle cx="10" cy="30" r="9" fill="green" stroke-dasharray="6 4" stroke="blue" fill-opacity="1"
   stroke-width="2" stroke-linecap="round" />

<circle cx="10" cy="45" r="9" fill="yellow" stroke-dasharray="6 4" stroke="blue"
   fill-opacity="0.5" stroke-width="2" stroke-opacity="0.5" stroke-linecap="round" />
```

Ergibt diesen Output:



Der Abschluss der Konturlinien ist `butt`.

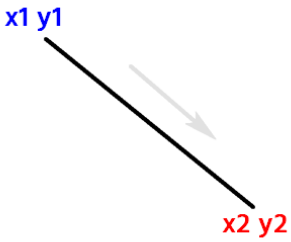
Der Abschluss der Konturlinien ist `round`.
Die Konturlinien haben 6 Pixel Länge (mit jeweils 4 Pixel Leerabstand).

Die Füllfarbe (gelb) und die Konturlinien (blau) haben eine Transparenz von 50%.

<line>

Das Element `line` zeichnet eine Line und hat 4 Attribute:

x1	x-Koordinate vom Startpunkt
y1	y-Koordinate vom Startpunkt
x2	x-Koordinate vom Endpunkt
y2	y-Koordinate vom Endpunkt



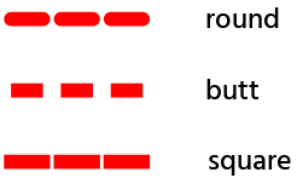
stroke-Attribute siehe Seite 252.

marker-Attribute siehe Seite 271

Beispiel:

```
<line x1="10" y1="5" x2="80" y2="50" stroke-width="10"
stroke-dasharray="7 4" stroke-linecap="round" stroke="red"/>
```

Ergibt diesen Output:

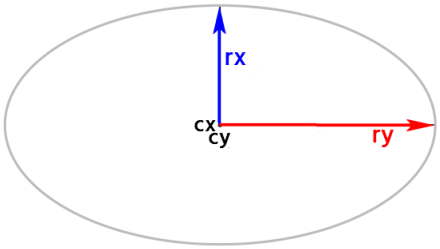


<elipse>

Das Element `elipse` zeichnet eine Elipse.

Attribut:

cx	x-Koordinate (default ist Pixel)
cy	y-Koordinate
rx	x-Radius
ry	y-Radius



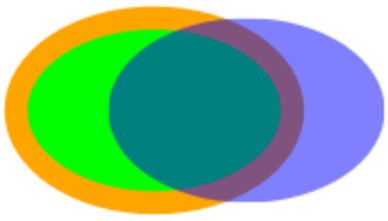
stroke-Attribute siehe Seite 252.

marker-Attribute siehe Seite 271

Beispiel:

```
<ellipse cx="40" cy="40" rx="30" ry="20" style="stroke:orange; stroke-width: 5; fill: lime;" />

<ellipse cx="60" cy="40" rx="30" ry="20" style="stroke:red; stroke-width: 0; fill: blue; opacity:0.5"
/>
```

Ergibt als Output:



Die orange Konturlinie hat eine Dicke von 5 Pixel (blau 0 Pixel).

Die blaue Elipse hat eine Transparenz von 50% (orange 0%).

<polyline>

Das Element polyline verbindet alle angegebenen Punkte. Im Gegensatz zu polygon, wird der letzte Punkt mit dem Startpunkt nicht verbunden.

Attribute:

points	Datentyp ist String. Die Koordinaten-Paare, die verbunden werden sollen, sind durch ein Leerzeichen getrennt.	
stroke-linejoin	bestimmt die Form vom **Polygon Bend**	
	fill	fill-Attribute siehe Seite 250
	stroke	stroke-Attribute siehe Seite 252.
	marker	marker-Attribute siehe Seite 271

Beispiel:

```
<polyline points="10,2  60,2  35,52"
        style="stroke:red; stroke-width: 3; fill: lime;"/>
```

Ergibt diesen Output:

<polygon>

Das Element polygon verbindet alle angegebenen Punkte. Im Gegensatz zu polyline, wird der letzte Punkt mit dem Startpunkt verbunden.

Attribute:

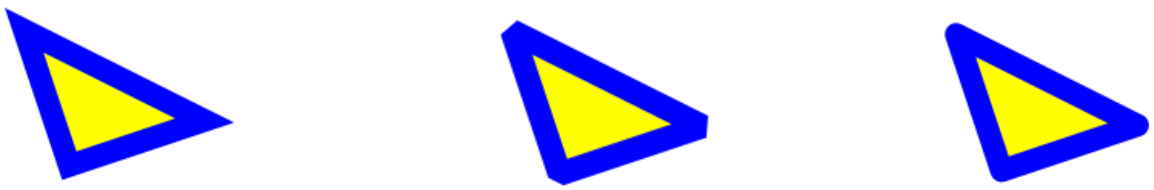
points	Datentyp ist String. Die Koordinaten-Paare, die verbunden werden sollen, sind durch ein Leerzeichen getrennt.	
stroke-linejoin	bestimmt die Form vom **Polygon Bend**	
	fill	fill-Attribute siehe Seite 250
	stroke	stroke-Attribute siehe Seite 252.
	marker	marker-Attribute siehe Seite 271

Beispiel 1:

```
<polygon points="0,0 10,30 40,20"
style="stroke: blue; fill: yellow; stroke-width:5; stroke-linejoin:miter" />
```

Mit dem Attribut stroke-linejoin (siehe Seite 252) wird die Form vom **Polygon Bend** bestimmt:

miter **bevel** **round**



<path>

Mit dem Element `path` können Linien und Kurven gezeichnet werden.

Attribut:

`d`	Attributwert ist ein String. Im String befinden sich beliebig viele Draw Anweisungen
`marker-start`	Positioniere Marker am Beginn des Pfades (siehe Seite 271)
`marker-mid`	Positioniere Marker in der Mitte (Ecken) des Pfades (siehe Seite 271)
`marker-mid`	Positioniere Marker am Ende des Pfades (siehe Seite 271)

Draw-Anweisungen

Linien

L	Linie ziehen nach	Zwei Attribute:	`x,y`	siehe Seite 258
M	Move to (Pfadcursor verschieben)	Zwei Attribute:	`x,y`	siehe Seite 257
H	Horizontale Linie zeichen	Ein Attribut:	`x`	siehe Seite 258
V	Vertikale Linie zeichnen	Ein Attribut:	`y`	siehe Seite 258
Z	gerade Linie zurück zum Startpunkt			siehe Seite 259

Kurven

A	Elliptischer Kurvenbogen Sieben Attribute: `rx,ry,xrot,lf,sf,x,y`	siehe Seite 262
C	Kubische Bézierkurve Sechs Attribute: `c1x,cy1y  c2x,c2y  x,y`	siehe Seite 260
Q	Quadratische Bézierkurve Vier Attribute: `cx,cy  x,y`	siehe Seite 259

Draw-Anweisungen als Großbuchstaben erwarten absolute Positionsangaben.
Die gleichen Draw-Anweisungen als Kleinbuchstaben erwarten relative Positionsangaben.

Als Trennzeichen für Attribute kann das Leerzeichen und das Komma verwendet werden.

M (Move Cursor to)

Die Draw-Anweisung `M` setzt an die angegebene x,y-Koordiante den Zeichenstift an.

```
path d="M10,10 H 20 M 30,10 H 40 M 50,10 H 60" stroke="#000" stroke-width="2" fill="red">
```

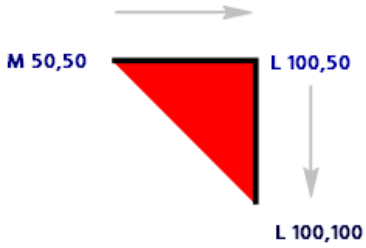
Ergibt als Output:

L (Linie)

Die Draw-Anweisung L zeichnet eine Linie nach x,y. Am Pfad-Beginn und Ende können Marker gesetzt werden (Siehe Seite 271).

```
<path d="M50,50 L 100,50 L 100,100" stroke="#000" stroke-width="2" fill="red">
```

Ergibt als Output:



Wird L (oder andere Draw-Anweisungen) wiederholt ausgeführt, muss L nur einmal angegeben werden:

```
<path d="M50,50 L 100,50 L 100,100" stroke="#000" stroke-width="2" fill="red">
```

Ergibt:

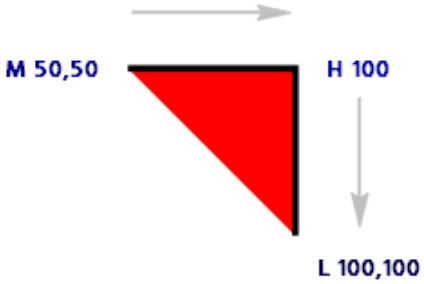
```
<path d="M50,50 L 100,50 100,100" stroke="#000" stroke-width="2" fill="red">
```

H (horizontale Linie)

Die Draw-Anweisung H zeichnet eine horizontale Linie nach x. Am Pfad-Beginn und Ende können Marker gesetzt werden (Siehe Seite 271).

```
<path d="M50,50 H 100 L 100,100" stroke="#000" stroke-width="2" fill="red">
```

Ergibt als Output:

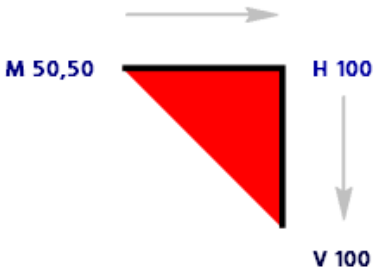


V (verticale Linie)

Die Draw-Anweisung V zeichnet eine horizontale Linie nach x. Am Pfad-Beginn und Ende können Marker gesetzt werden (Siehe Seite 271).

```
<path d="M50,50 H 100 V 100" stroke="#000" stroke-width="2" fill="red">
```

Ergibt als Output:

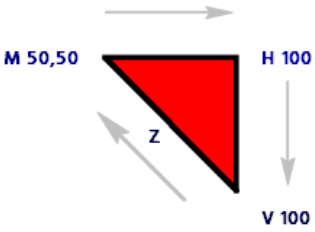


Z (zurück zum Startpunkt)

Die Draw-Anweisung Z zeichnet eine Line zurück zum Startpunkt (close path).

```
<path d="M50,50 H 100 V 100 Z" stroke="#000" stroke-width="2" fill="red">
```

Ergibt als Output:



Q (quadratische Bézierkurve)

Die Draw-Anweisung Q zeichnet eine quadratische Bézierkurve.

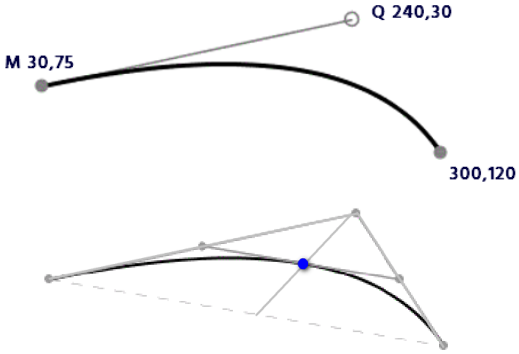
Vier Attribute: `cx,cy  x,y`

cx,cy ist der Control-Point. Der **Inflection Point** ist der Punkt, an dem die Kurve die Richtung ändert.

Beispiel 1:

```
<path d="M30,75 Q 240,30 300,120" stroke="black" stroke-width="2" fill="red">
```

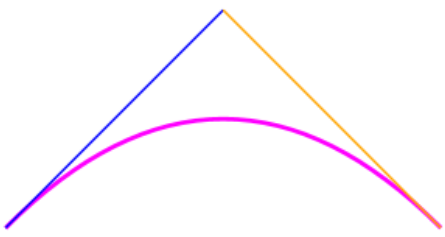
Erzeugt diesen Output:

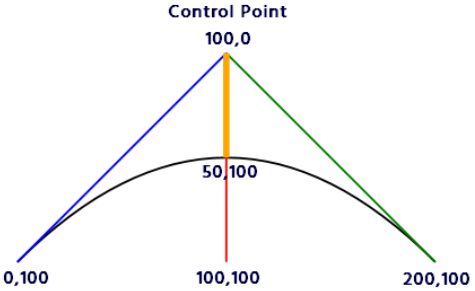


Beispiel 2:

```
<path d="M 0,100 Q 100 0, 200 100" stroke-width="2" fill="white" stroke="fuchsia" />

<line x1="0" y1="100" x2="100" y2="00" stroke="blue"  stroke-width="1" />
<line x1="100" y1="0" x2="200" y2="100" stroke="orange"  stroke-width="1" />
```

Erzeugt diesen Output:





T

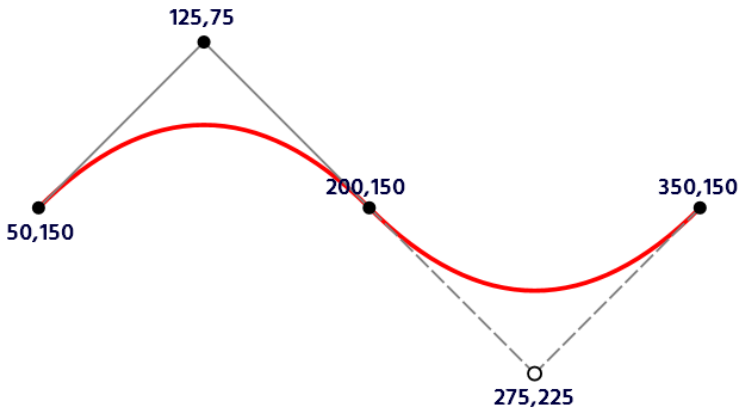
Nach einer Q Anweisung kann eine T Anweisung folgen. T hängt eine 2. Bezierkurve an die erste Bezierkurve an. Diese Anweisung hat nur 1 Koordinatenpunkt: x,y des Endpunktes.

Der erste Kontrollpunkt wird automatisch aus dem letzten Kontrollpunkt der direkt zuvor festgelegten Bézier-Kurve generiert: ein Spiegelpunkt.

Beispiel:

```
<path d="M 50,150 Q 125,75 200,150  T 350,150" />
```

Ergibt als Output:



C (kubische Bézierkurve)

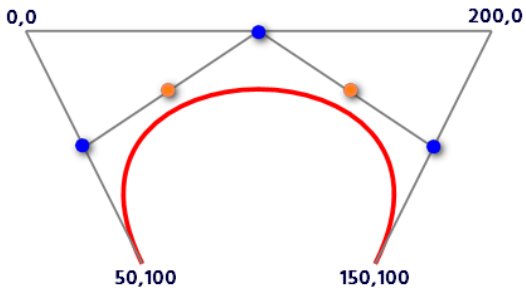
Die Draw-Anweisung C zeichnet eine kubische Bezierkurve.
Sechs Attribute: **c1x,c1y c2x,c2y x,y**

c1 ist Control-Point 1 und c2 ist Control-Point 2.

Beispiel 1:

```
<path d="M 50,100 C 0,00 200,0 150,100" stroke="red" stroke-width="2" fill="white" />
```

Ergibt als Output:



Beispiel 2:

```
<path d="M 40,50 C 110,10 40,10 110,50" stroke="red" stroke-width="2" fill="white" />
```

Beispiel 3:

```
<path d="M 40,50 C 110,10 40,90 110,50" stroke="red" stroke-width="2" fill="white" />
```

Beispiel 4:

```
<path d="M 40,50 C 60,10 90,90 110,50" stroke="red" stroke-width="2" fill="white" />
```

S

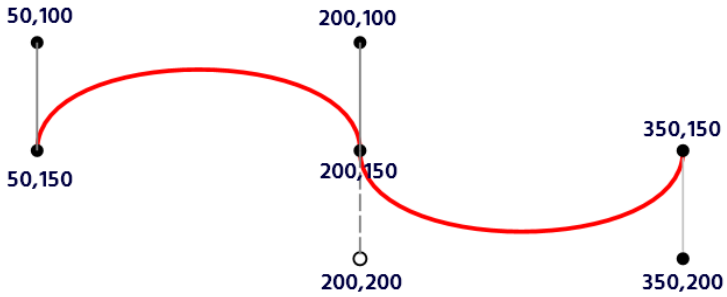
Nach einer C Anweisung kann eine S Anweisung folgen. S hängt eine 2. Bezierkurve an die erste Bezierkurve an. Diese Anweisung hat nur 2 Koordinatenpunkte: x,y des zweiten Control-Point und x,y des Endpunktes.

Der erste Kontrollpunkt wird automatisch aus dem letzten Kontrollpunkt der direkt zuvor festgelegten Bézier-Kurve generiert: ein Spiegelpunkt.

Beispiel:

```
<path d="M 50,150  C  50,100 200,100 200,150  S 350,200 350,150" fill="none" stroke="red"/>
```

Ergibt als Output:



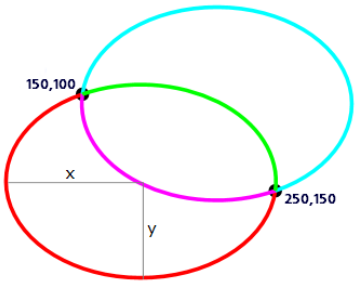
A (elliptischer Kurvenbogen)

Die Anweisung A hat sieben Attribute:

rx	Radius der x-Achse der Ellipse	
ry	Radius der y-Achse der Ellipse	
xrot	die Rotation der x-Achse der Ellipse in Grad (0 = Rotation)	
lf	large-arc-flag:	0 für den kurzen Weg um die Ellipse 1 für den langen Weg um die Ellipse
sf	Sweep-flag:	0 für Zeichnung entgegen den Uhrzeigersinn 1 für Zeichnung mit dem Uhrzeigersinn
x	x-Koordinate des Endpunktes	
y	y-Koordinate des Endpunktes	

Beispiel: langer Weg *gegen Uhrzeigersinn*:

```
<path d="M 150,100  A 70 50 0 1 0 250,150"  stroke="red"
fill="none" stroke-width="2"/>
```



langer Weg *im Uhrzeigersinn*:

```
<path d="M 150,100  A 70 50  0 1 1  250,150" stroke="aqua" fill="none" stroke-width="2"/>
```

kurzer Weg *gegen Uhrzeigersinn*:

```
<path d="M 150,100  A 70 50  0 0 0  250,150" stroke="fuchsia" fill="none" stroke-width="2" />
```

kurzer Weg *im Uhrzeigersinn*:

```
<path d="M 150,100  A 70 50  0 0 1  250,150" stroke="lime" fill="none" stroke-width="2"/>
```

<text> <tspan> <textPath>

Mit dem Element `text` wird Text und `tspan`-Elemente gruppiert, formatiert und angezeigt.

Die Elemente `tspan` und textPath sind Gruppierungselemente innerhalb vom Element `text` (zum Formatieren von einzelnen Abschnitten eines Textes). Siehe auch CSS (Seite 268).

SVG1.2 ist nur zeilenorientiert (also keine mehrzeiligen Absätze).

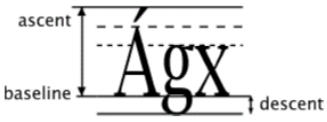
Attribute von `text` und `tspan`:

`x`	x-Koordinate vom Startpunkt (linke obere Ecke der Textbox)
`y`	x-Koordinate vom Startpunkt (linke obere Ecke der Textbox)
`dx`	verschiebe um dx horizontal (relative Position)
`dy`	verschiebe um dy vertikal (relative Position)

`font-weight`	Mögliche Attributwerte (fett):
	`bold`
	`normal`

`font-style`	Mögliche Attributwerte (kursiv):
	`italic`
	`normal`

`text-decoration`	Mögliche Attributwerte (unterstrichen, oberstrichen, durchgestrichen):
	`none`
	`underline`
	`overline`
	`line-through`

`word-spacing`	Wortabstand (Leerraum). Mögliche Attributwerte: `normal` *value* (*value* wird zum default Wortabstand dazugezählt)
`letter-spacing`	Zeichenabstand (Leerraum). Mögliche Attributwerte: `normal` *value* (*value* wird zum default Zeichenabstand dazugezählt)
`font-size`	Schriftgröße ist Distanz von Baseline zu Baseline: 
`stroke`	Schriftfarbe (siehe RGB-Wert Seite 212)

`font-family`	Schriftfamilie. (je nach Renderer auch Schriftart). Mögliche Werte:
	`serif`
	`sans-serif`
	`monospace`

`textLength`	autofit in die angegebene Länge

`lengthAdjust`	bei Verwendung von `textLength`, die autofit-Methode:
	`spacing`
	`spacingAndGlyphs`

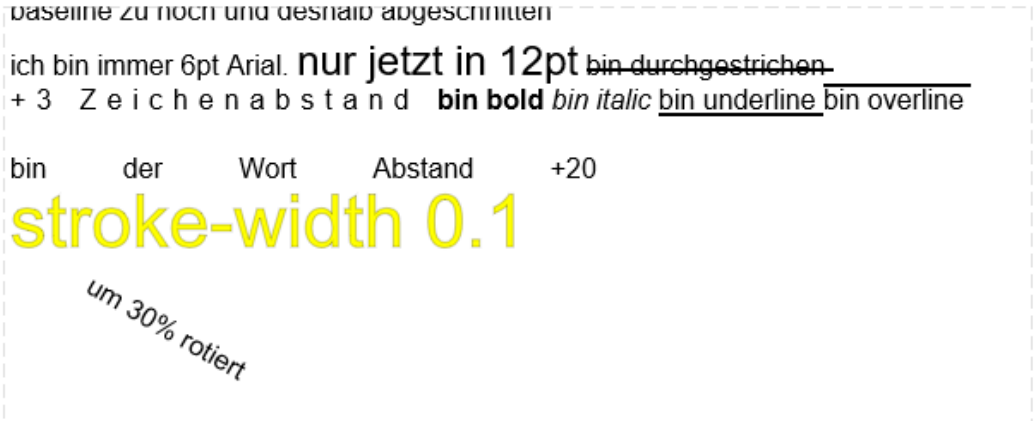
`text-anchor`	Ausrichtung. Mögliche Werte (linksbündig, zentriert, rechtsbündig):
	`start`
	`middle`
	`end`

`baseline-shift`	Hochstellung, Niederstellung
	`super`
	`sub`

Beispiel 1:

```
<svg width="300" height="300" >
   <defs>
      <style type="text/css">
    text {font-size:6pt; font-family:Arial;  }
   </style>
    </defs>

   <text x="2" y="4"> baseline zu hoch und deshalb abgeschnitten   </text>

   <text x="2" y="20">
   ich bin immer 6pt Arial. <tspan font-size="12">nur jetzt in 12pt</tspan>
   <tspan text-decoration="line-through"> bin durchgestrichen </tspan>
   </text>

   <text x="2" y="30" >
   <tspan letter-spacing="3"> +3 Zeichenabstand </tspan>
   <tspan font-weight="bold"> bin bold </tspan>
   <tspan font-style="italic"> bin italic </tspan>
   <tspan text-decoration="underline"> bin underline </tspan>
   <tspan text-decoration="overline"> bin overline </tspan>
   </text>

   <text x="2" y="50">
   <tspan word-spacing="20"> bin der Wort Abstand +20</tspan>
   </text>

   <text x="2" y="90" transform="rotate(30) translate(60,-30)">
   um 30% rotiert
   </text>

   <text x="2" y="70" style="font-size:16pt;stroke-width: 0.1;stroke:black; fill:yellow">
   stroke-width 0.1
   </text>
 </svg>
```

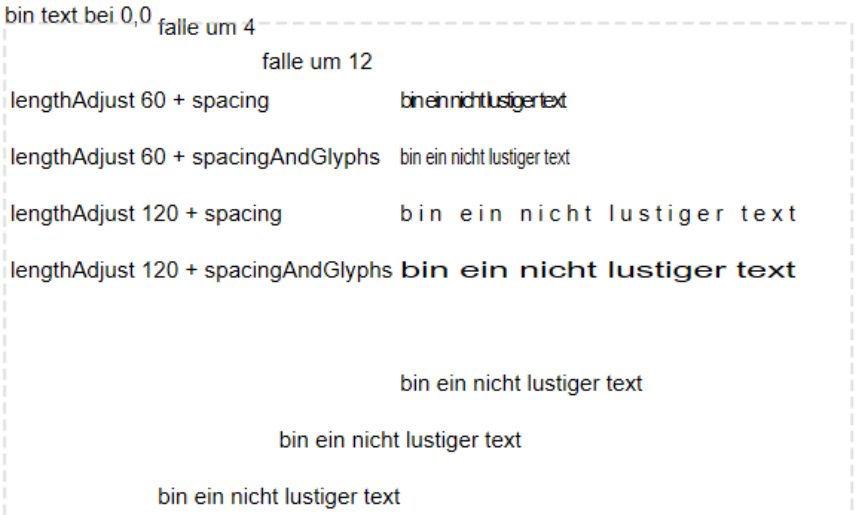
Ergibt als Output:



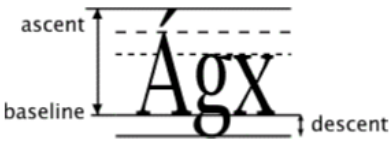
Beispiel 2:

```
<svg width="300" height="300" >
   <defs>
      <style type="text/css">
       text {font-size:6pt; font-family:Arial;  }
      </style>
    </defs>

   <text x="0" y="0" >
   bin text bei 0,0
   <tspan dy="4">falle um 4</tspan>
   <tspan dy="12">falle um 12</tspan>
   </text>

   <text x="2" y="30"> lengthAdjust 60 + spacing </text>
   <text x="2" y="50"> lengthAdjust 60 + spacingAndGlyphs </text>
   <text x="2" y="70"> lengthAdjust 120 + spacing </text>
   <text x="2" y="90"> lengthAdjust 120 + spacingAndGlyphs </text>

   <text x="140" y="30" textLength="60" lengthAdjust="spacing">
   bin ein nicht lustiger text </text>

   <text x="140" y="50" textLength="60" lengthAdjust="spacingAndGlyphs">
   bin ein nicht lustiger text</text>

   <text x="140" y="70" textLength="140" lengthAdjust="spacing">
   bin ein nicht lustiger text</text>

   <text x="140" y="90" textLength="140" lengthAdjust="spacingAndGlyphs">
   bin ein nicht lustiger text</text>


   <text x="140" y="130" text-anchor="start">bin ein nicht lustiger text</text>

   <text x="140" y="150" text-anchor="middle">bin ein nicht lustiger text</text>

   <text x="140" y="170" text-anchor="end">bin ein nicht lustiger text</text>
</svg>
```

Ergibt als Output:



Beispiel 3: textPath



Bei Text innerhalb des Elementes `textPath`, läuft die Baseline der Textzeile einen Pfad entlang.

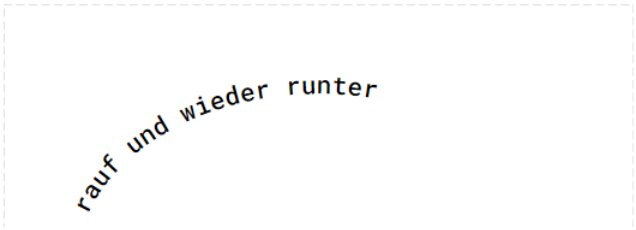
Mit dem Attribut `xlink:href` ruft `textPath` die id vom Pfad auf.

Mit dem Attribut `startOffset` kann die Position auf dem Pfad bestimmt werden.

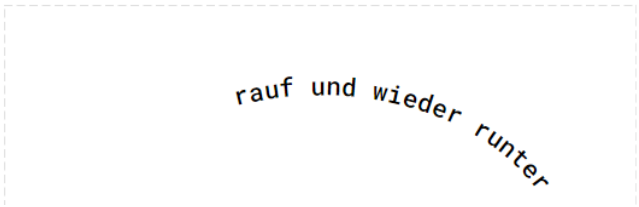
Beispiel:

```
<svg width="300" height="300" >
    <defs>
     <path id="foo" d="M 40,100  A 70 50  0 1 1  260,150" />
    </defs>

    <text style="font-size: 12;font-family:monospace">
    <textPath xlink:href="#foo">
    rauf und wieder runter
    </textPath>
    </text>
</svg>
```

Ergibt als Output:



Beispiel mit `startOffset="90"`:



<image>

Mit dem Element image wird ein externes Bild angezeigt.

Mögliche Attribute:

x	x-Koordinate
y	y-Koordinate
width	Breite des Bildes
height	Höhe des Bildes
xlink:href	URL
viewBox	siehe viewBox Seite 246
preserveAspectRatio	siehe preserveAspectRatio Seite 246
type	Mimetype

Beispiel:

```
<symbol style="stroke-dasharray: 4 2;"  id="foo" >
   <circle cx="40" cy="50"  r="40" stroke="blue" stroke-width="3" fill="red" opacity="0.5" />
   <circle cx="40" cy="70"  r="10" stroke="black" stroke-width="3" fill="black" opacity="0.5" />
</symbol>

      <image x="140" y="0" xlink:href="ninja.png" width="120" height="148" />
      <use xlink:href="#foo" x="80"/>
      <use xlink:href="#foo" x="230" />
```

Ergibt als Output:

Cascading Style Sheets (CSS)

Es gibt drei Orte für CSS-Deklarationen (siehe Deklaration Seite 158):

inline declaration

In einem Tag mit dem Attribut `style`.

Beispiel:

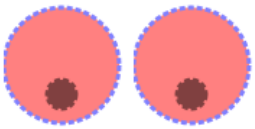
```
<path d="M 150,100 A 70 50 0 1 1 250,150" style="stroke:gray; stroke-width:1px;" />
```

Rule-Set

Als CSS-Rule-Sets mit Selektoren (Seite 165) und Kombinatoren (Seite 169).

```
<defs>
   <style type="text/css">
     .cssfoo {stroke-dasharray: 4 2;}
    </style>
 </defs>


<symbol class="cssfoo"  id="foo" >
   <circle cx="40" cy="50"  r="40" stroke="blue" stroke-width="3" fill="red" opacity="0.5" />
   <circle cx="40" cy="70"  r="10" stroke="black" stroke-width="3" fill="black" opacity="0.5" />
</symbol>

<use xlink:href="#foo" x="90"/>
<use xlink:href="#foo" x="180" />
```

Ergibt als Output:



externer Rule-Set

Als Beispiel das externe File `foo.css`:

```
<?xml-stylesheet type="text/css" href="foo.css"?>
```

Gruppierungselemente

Gruppierungselemente sind Container, die mehrere Elemente zusammenfassen.

	Nachfahren erben			
	das Inline-CSS	**Gruppen-ID**	**viewBox**	**wird sofort gerendert**
symbol	ja	ja	ja	nein
g	ja	ja	nein	ja
defs	nein	nein	nein	nein
marker	ja	ja	ja	nein

Der Aufruf geschieht mit dem Element use (siehe Seite 273)

Das Element marker kann nur in einem path Element aufgerufen werden.

<symbol>

Mit dem Element symbol wird eine Gruppe von Elementen markiert (Container-Element). Die Elemente innerhalb von symbol werden nicht gerendert (im Unterschied zum Element g, siehe Seite 270).

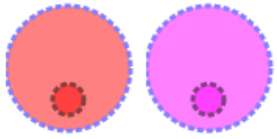
Mögliche Attribute:

id	Mit dem Attribut id wird dem Symbol eine eindeutige ID zugewiesen. Über diese ID lassen sich die gruppierten Elemente aufrufen und ausführen. Der Aufruf geschieht mit dem Element use (siehe Seite 273) und dem href Attribut (bei href ist Der Attributwert die Raute, gefolgt von der ID).
style	Inline-CSS im Element symbol vererbt sich auf die Nachfahren-Elemente.
viewBox	siehe Seite 246
preserveAspectRatio	siehe Seite 246)

Beispiel:

```
<symbol  style="stroke-dasharray: 4 2;" id="foo">
   <circle cx="40" cy="50"  r="40" stroke="blue" stroke-width="3"  opacity="0.5" />
     <circle cx="40" cy="70"  r="10" stroke="black" stroke-width="3"  opacity="0.5" />
</symbol>

<use xlink:href="#foo" x="90" style="fill:red" />
<use xlink:href="#foo" x="180" style="fill:fuchsia" />
```

Ergibt als Output:



<defs>

Mit dem Element defs wird eine Gruppe von Elementen markiert (Container-Element). Die Elemente innerhalb von defs werden nicht gerendert. Im Unterschied zu g und symbol ist weder Inline-CSS, noch eine Gruppen-ID möglich.

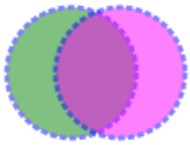
Im Element defs können auch die Elemente g, marker und symbol eingebettet werden.

Der Aufruf geschieht mit dem Element use (siehe Seite 273) und dem href Attribut.
Der Attributwert ist die Raute, gefolgt von der ID.

Beispiel:

```
<defs>
   <circle cx="40" cy="50"  id="foo" r="40" stroke="blue"
      stroke-width="3"  opacity="0.5" stroke-dasharray="4 2" />
</defs>

<use xlink:href="#foo" x="60" style="fill:green" />
<use xlink:href="#foo" x="90" style="fill:fuchsia" />
```

Ergibt diesen Output:



<g>

Mit dem Element g wird eine Gruppe von Elementen markiert (Container-Element). Die Elemente innerhalb von g werden gerendert (im Unterschied zum Element symbol).

Mögliche Attribute:

id	Mit dem Attribut id wird dem Element eine eindeutige ID zugewiesen. Über diese ID lassen sich die gruppierten Elemente aufrufen und ausführen. Der Aufruf geschieht mit dem Element use (siehe Seite 273) und dem href Attribut (bei href ist Der Attributwert die Raute, gefolgt von der ID).
style	Inline-CSS im Element symbol vererbt sich auf die Nachfahren-Elemente.

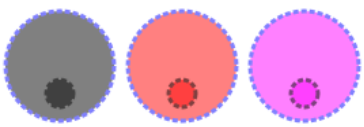
Inline-CSS im Element g vererbt sich auf die Nachfahren-Elemente.

Das g Element kann im Element defs eingebettet werden (und wird damit nicht mehr automatisch gerendert).

Beispiel:

```
<g  style="stroke-dasharray: 4 2;" id="foo">
 <circle cx="40" cy="50"  r="40" stroke="blue" stroke-width="3"  opacity="0.5" />
 <circle cx="40" cy="70"  r="10" stroke="black" stroke-width="3"  opacity="0.5" />
</g>

<use xlink:href="#foo" x="90" style="fill:red" />
<use xlink:href="#foo" x="180" style="fill:fuchsia" />
```

Ergibt als Output:



<marker>

Mit dem Element `marker` wird eine Gruppe von Elementen markiert (Container-Element).
Hauptverwendungszweck ist das Positionieren von Richtungspfeilen und Knotenpunkten.

Die Elemente innerhalb von `marker` werden nicht gerendert. Marker können am Beginn und am Ende eines Pfades (`path` Element) positioniert werden.

Mögliche Attribute im Element `marker`:

`id`	Mit dem Attribut `id` wird dem Marker eine eindeutige ID zugewiesen. Über diese ID lassen sich die gruppierten Elemente aufrufen und ausführen. Der Aufruf geschieht im Element `path` mit den Attributen `marker-start`, `marker-mid` und `marker-end`.
`marker-width`	Breite
`marker-height`	Höhe
`refX`	x-Koordinate (optional)
`refY`	y-Koordinate (optional)
`style`	Inline-CSS im Element `marker` vererbt sich auf die Nachfahren-Elemente.
`viewBox`	siehe Seite 246
`preserveAspectRatio`	siehe Seite 246)
`orientation`	Attributwert ist `auto` das Objekt wird in die Pfadrichtung gedreht (zB Pfeilspitzen für Richtungsangaben)

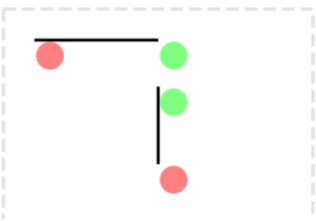
Mögliche Attribute im Element `path`:

`marker-start`	Positioniere Marker am Beginn des Pfades
`marker-mid`	Positioniere Marker in der Mitte des Pfades An jeder Ecke. Und an jeder Stelle, an der mit `M` gesprungen worden ist.
`marker-end`	Positioniere Marker am Ende des Pfades
`marker`	Positioniere Marker an allen Stellen (Shorthand für start, mid und end)

Beispiel 1a (ohne `refX`, `refY`):

```
<svg width="100" height="100"
<marker id="foo1" markerWidth="50" markerHeight="50" >
   <circle cx="5" cy="5" r="4" style="fill: red; stroke: red; opacity:0.5;"/>
</marker>

<marker id="foo2" markerWidth="50" markerHeight="50" >
   <circle cx="5" cy="5" r="4" style="fill: lime; stroke: lime; opacity:0.5;"/>
</marker>

<path d="M 10 10 50 10 M 50 25 L 50 50 " fill="none" style="marker-start: url(#foo1); marker-mid:
url(#foo2); marker-end: url(#foo1); stroke: black; "/>
</svg>
```

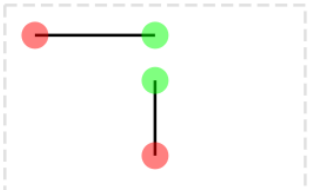
Ergibt als Output:



Beispiel 1b (mit `refX, refY`):

```
<svg width="100" height="100"
<marker id="foo1" markerWidth="50" markerHeight="50" >
   <circle cx="5" cy="5" r="4" style="fill: red; stroke: red; opacity:0.5;"/>
</marker>

<marker id="foo2" markerWidth="50" markerHeight="50" >
   <circle cx="5" cy="5" r="4" style="fill: lime; stroke: lime; opacity:0.5;"/>
</marker>

<path d="M 10 10 50 10 M 50 25 L 50 50 " fill="none" style="marker-start: url(#foo1); marker-mid:
url(#foo2); marker-end: url(#foo1); stroke: black; "/>
</svg>
```

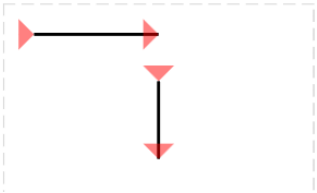
Ergibt als Output:



Beispiel 2 (mit `orientation="auto"`):

```
<svg width="100" height="100" >

   <marker id="foo1" markerWidth="5" markerHeight="10" refX="5" refY="5" orient="auto">
       <path d="M 0 0 5 5 0 10 Z" style="fill: red; opacity: 0.5"/>
   </marker>

   <path d="M 10 10 50 10 M 50 25 L 50 50 " fill="none" style="marker-start: url(#foo1);
     marker-mid: url(#foo1); marker-end: url(#foo1); stroke: black; "/>
</svg>
```

Ergibt als Output:



Shorthand

Das Attribut `marker` ist ein Shorthand für `marker-start, marker-mid, marker-end`

Beispiel:

```
marker-start: url(#foo1); marker-mid: url(#foo1); marker-end: url(#foo1)
```

als Shorthand:

```
marker: url(#foo1);
```

<use>

Mit dem Element `use` wird ein Element (oder eine Elementgruppe) mit einer bestimmten ID aufgerufen und ausgeführt.

Die Referenzierung auf das gewünschte Element, geschieht über das Attribut `xlink:href`

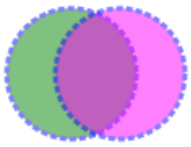
Mögliche Attribute:

`x`	x-Koordinate
`y`	y-Koordinate
`style`	siehe CSS Seite 268
`xlink:href`	Referenzierung über das Attribut `id` (bzw Attributwert) vom gewünschten Element. Auch per URL: `xlink:href="extern.svg#foo"`

Beispiel:

```
<defs>
   <circle cx="40" cy="50"  id="foo" r="40" stroke="blue"
      stroke-width="3"  opacity="0.5" stroke-dasharray="4 2" />
</defs>

<use xlink:href="#foo" x="60" style="fill:green" />
<use xlink:href="#foo" x="90" style="fill:fuchsia" />
```

Ergibt diesen Output:



Transformation

Das Attribut `transform` hat als Attributwert einen String.

Im String sind folgende Funktionen möglich (bei mehreren durch Leerzeichen getrennt):

`translate ()`	verschiebt das Koordinatengitter des Objekts zur Position `x,y` (siehe Seite 274)
`scale ()`	ändert die Größe des Koordinatengitter des Objekt (siehe Seite 274). Parameter ist der Skalierungsfaktor. Ein Wert über 1 vergrößert, ein Wert unter 1 verkleinert.
`rotate()`	rotiert das Koordinatengitter des Objekts (siehe Seite 276).
`skew()`	neigt das Koordinatengitter des Objekts (siehe Seite 277).

translate

Die Funktion `translate()` verschiebt das Koordinatengitter des jeweiligen Objekts.

Mögliche Funktionsparameter:

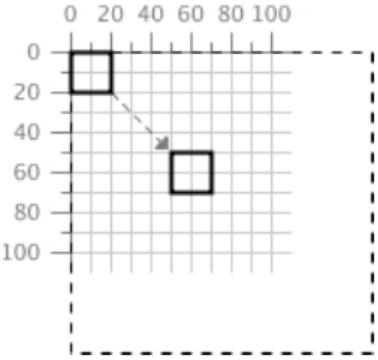
x	verschiebt das Koordinatengitter um x
y	verschiebt das Koordinatengitter um y

Beispiel:

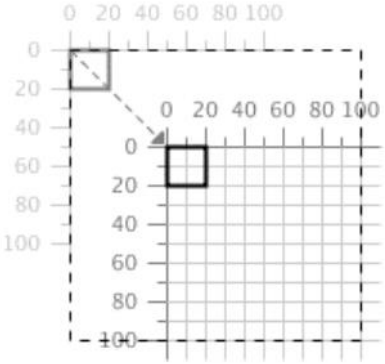
```
<svg width="100" height="100"
<symbol  style="stroke-dasharray: 4 2;" id="foo">
   <rect x="0" y="0" width="20" height="20" />
</symbol>

<rect x="0" y="0" width="100" height="100" stroke="silver"
   stroke-width="1" stroke-dasharray="4 2" fill="none" />
<use xlink:href="#foo" x="0" y="0" style="fill:fuchsia" />
<use xlink:href="#foo" x="0" y="0" transform="translate(50,50)"  style="fill:red" />
</svg>
```

Ergibt als Output:

Optisch verschiebt sich das Objekt von 0,0 nach 50,50:



Intern wird das Objekt-Gitter verschoben:



scale

Die Funktion `scale()` verändert die Größe des Koordinatengitters eines Objekts. Der Übergabe-Parameter ist der Skalierungsfaktor. Ein Wert über 1 vergrößert, ein Wert unter 1 verkleinert.

Scale mit 2 Parameter:

`scale (x,y)` x ist der Skalierungsfaktor für die x-Achse. y ist der Skalierungsfaktor für die y-Achse

Scale mit 1 Parameter:

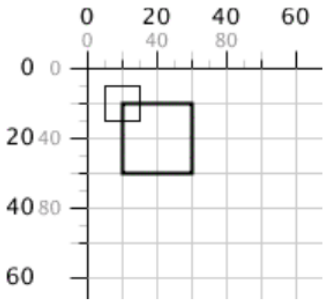
`scale (v)` v ist der Skalierungsfaktor für die x- und y-Achse

Beispiel1:

```
<svg width="100" height="100"
<symbol  style="stroke-dasharray: 4 2;" id="foo">
    <rect x="10" y="10" width="20" height="20" />
</symbol>
<rect x="0" y="0" width="100" height="100"
   stroke="silver" stroke-width="1" stroke-dasharray="4 2" fill="none" />

<use xlink:href="#foo" x="0" y="0" style="fill:fuchsia" />
<use xlink:href="#foo" x="0" y="0" transform="scale(2)"  style="fill:red" />
</svg>
```

Ergibt als Output:

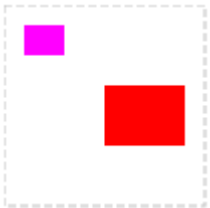
Vergrößerung des Koordinatensystem:

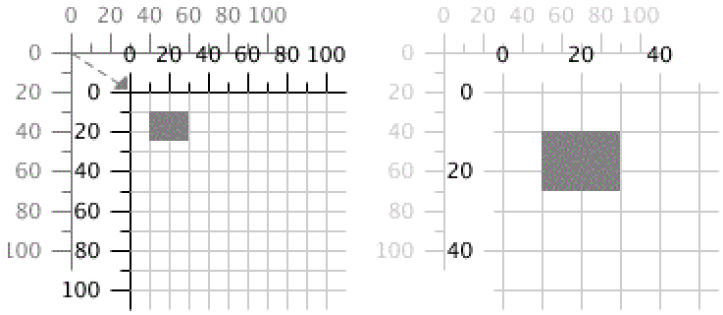


Beispiel2:

```
<symbol  style="stroke-dasharray: 4 2;" id="foo">
    <rect x="10" y="10" height="15" width="20" />
</symbol>
<rect x="0" y="0" width="100" height="100" stroke="silver"
   stroke-width="1" stroke-dasharray="4 2" fill="none" />

<use xlink:href="#foo" x="0" y="0" style="fill:fuchsia" />
<use xlink:href="#foo" x="0" y="0" transform="translate(30,20) scale(2)"  style="fill:red" />
```

Ergibt als Output:





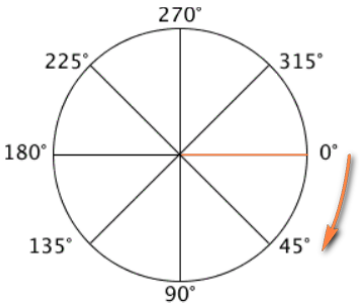
scale around a point

Um ein Objekt um *centerX, centerY* um den Faktor *factor* zu skalieren:

```
translate(-centerX*(factor-1), -centerY*(factor-1))
scale(factor)
```

rotate

Die Funktion `rotate()` rotiert das Koordinatengitter des Objekts (im Uhrzeigersinn).



Mit 1 Parameter:

`rotate(g)` rotiert das Koordinatensystem des Objekts um *g* Grad
Drehpunkt liegt bei 0,0

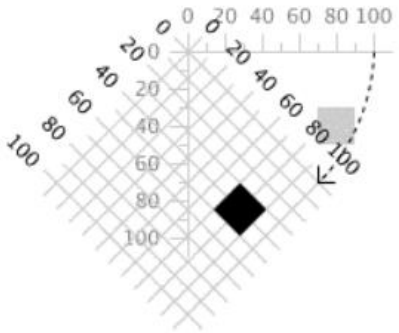
Mit 2 Parameter:

`rotate(g,x,y)` rotiert das Koordinatensystem des Objekts um *g* Grad.
Drehpunkt liegt bei Position *x,y* (im Objekt links oben)

Beispiel1:

```
<symbol style="stroke-dasharray: 4 2;" id="foo">
    <rect x="70" y="30" width="20" height="20" />
</symbol>
<rect x="0" y="0" width="100" height="100" stroke="silver" stroke-width="1"
   stroke-dasharray="4 2" fill="none" />
<use xlink:href="#foo" x="0" y="0" style="fill:fuchsia" />
<use xlink:href="#foo" x="0" y="0" transform="rotate(45)"  style="fill:red" />
```

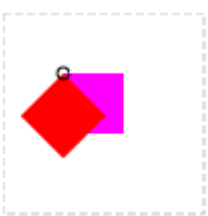
Erzeugt als Output:



Beispiel2:

```
<svg width="100" height="100"
<symbol style="stroke-dasharray: 4 2;" id="foo">
   <rect x="30" y="30" width="30" height="30" />
</symbol>
<rect x="0" y="0" width="100" height="100" stroke="silver"
   stroke-width="1" stroke-dasharray="4 2" fill="none" />

<use xlink:href="#foo" style="fill:fuchsia" />
<use xlink:href="#foo" transform="rotate(45,30,30)"  style="fill:red" />

<circle cy="30" cx="30" r="3" fill="none" stroke="black" />
</svg>
```

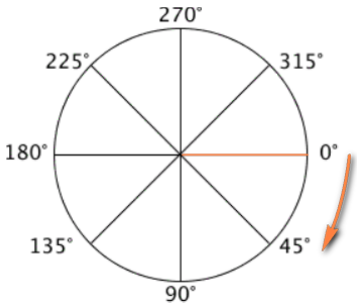
Ergibt als Output:



skew

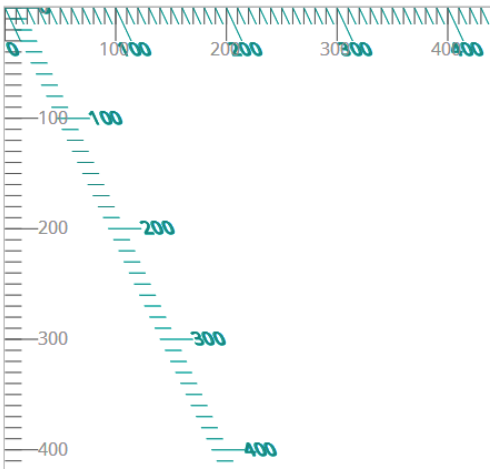
Die Funktion `skewX(g)` neigt die x-Achse des Koordinatengitters des Objekts um *g* Grad (im Uhrzeigersinn). Die Funktion `skewY(g)` neigt die y-Achse des Koordinatengitters des Objekts um *g* Grad (im Uhrzeigersinn).

Beispiel:




```
<svg width="100" height="100"
   <symbol transform="" style="stroke-dasharray: 4 2;" id="foo">
        <rect x="0" y="0" width="30" height="30" opacity="0.5" />
   </symbol>
   <rect x="0" y="0" width="100" height="100" stroke="silver"
        stroke-width="1" stroke-dasharray="4 2" fill="none" opacity="0.5"/>

   <use xlink:href="#foo" style="fill:black" />
   <use xlink:href="#foo" transform="skewX(30)"  style="fill:red" />
</svg>
```

Erzeugt als Output:



JavaScript

Kapitel 10

JavaScript

Geschichte

NCSA Mosaic wurde 1993 auf der Universität von Illinois entwickelt und war der erste Webbrowser mit einer GUI (Graphical User Interface). 1994 wurde von der Firma **Netscape** der Browser **Netscape Navigator** veröffentlicht und wurde bis 1996 der Marktführer (75% Marktanteil).

Die Sprache **Mocha** wurde im Frühjahr 1995 vom Netscape Entwickler Brendan Eich entwickelt. Im September 1995 wurde Mocha in **LiveScript** umbenannt.

Im Winter 1995 ging Netscape eine Kooperation mit der Firma **Sun** ein, um mit **LiveScript** auch Java-Applets steuern zu können. Die dazu notwendige Schnittstelle wurde von Netscape **LiveConnect** getauft.
Um von der Popularität der neuen Sprache Java (und den Java-Applets) zu profitieren, erwarb Netscape im Dezember 1995 eine **trademark license** von Sun, um LiveScript in **JavaScript** umbennen zu dürfen.
1996 wurde JavaScript 1.0 erstmals mit dem Browser Netscape Navigator **2** veröffentlicht.

Als Konkurenzprodukt veröffentlichte Microsoft 1996 mit dem Browser **Internet Explorer 3** die Sprache **JScript** (eine VBScript-Portierung, die in etwa den Umfang von JavaScript 1.0 abdeckte). Der Beginn des Browser Krieges.
Im Oktober 1997 kam der **Internet Explorer 4** heraus, der etwa den Sprachumfang von JavaScript 1.1 abdeckte.

Die Fähigkeit, mit CSS und JavaScript eine HTML-Seite zur Laufzeit dynamisch zu verändern, wurde DHTML (Dynamic HTML genannt). DHTML für **Netscape** war in vielen Teilen nicht kompatibel für den **Internet Explorer**.

Um JavaScript zu standardisieren, erarbeitete Netscape, gemeinsam mit der **European Computer Manufacturers Association,** den Script-Standard ECMA-262, der im Juni 1997 veröffentlicht wurde. Dieser wurde im April 1998 als ISO-Norm **ISO/IEC 16262:1998** übernommen.

Netscape/Firefox JavaScript-Versionen

1996	Netscape Navigator 2.0	JavaScript 1.0 (DOM Level 0, Zugriff auf Formularfelder)
1996	Netscape Navigator 3.0	JavaScript 1.1 (Position und Sichtbarkeit von Bildern ändern)
1997	Netscape Communicator 4.0	JavaScript 1.2 (CSS ändern, Position und Sichtbarkeit von Elementen)
1998	Netscape Communicator 4.5	JavaScript 1.3
2000	Netscape Navigator 6	JavaScript 1.5 (mit DOM Level 1)
2006	Mozilla 1.0	JavaScript 1.5 (ECMA-252 Version 3)
2005	Firefox 1.5	JavaScript 1.6
2006	Firefox 2.0	JavaScript 1.7
2007	Firefox 3.0	JavaScript 1.8
2009	Firefox 3.5	JavaScript 1.8 Updates
2010	Firefox 3.6	JavaScript 1.8.1
2010	Firefox 4	JavaScript 1.8.5. Vollständiges ECMAScript 5

Core, DOM, BOM

JavaScript besteht aus 3 Teilen:

1) ECMAScript (Core)
   Syntax, Keywords, Operatoren, Objekte
2) Document Object Model (DOM)
   Zugriff auf die Baum-Struktur des Dokumentes
3) Browser Object Model (BOM)
   Zugriff auf Browser-Komponenten (Cookies, PopUp, Fenster öffnen/verschieben/schliessen, ...)

Versionen von ECMA-262

Der Kern (Core) von JavaScript ist ECMAScript (ECMA-262). Es beschreibt: Syntax, Keywords, Operatoren und Objekte.
Die Versionen von ECMA-262:

ECMA-Version 1	1997	
ECMA-Version 2	1998	Kompatibilität zu ISO/IEC 16262
ECMA-Version 3	1999	regular expressions, exception handling
ECMA-Version 4		nicht veröffentlicht. optional static typing, packages and namespaces, generators & iterators, JSON, class-based object-oriented programming
ECMA-Version 5	2009	Mit **strict mode** wurde eine erweiterte Fehlerprüfung hinzugefügt. getter and setter properties, JSON
ECMA-Version 5.1	2011	Entspricht ISO/IEC 16262:2011, Version 3
ECMA-Version 6 (**Harmony**)	2015	iterators, for-of Schleifen, generator expressions, arrow functions, binary data, typed arrays, collections (maps, sets and weak maps), promises, reflection, proxies
ECMA-Version 7	2016	Array.prototype.includes, Exponentiation Operator **

Document Object Model (DOM)

Das Document Object Model (DOM) ist ein Application Programming Interface (API) für XML und HTML.
Die Definition wurde in IDL (Interface Definition Language) geschrieben.
Siehe DOM-Level auf mozilla.org.

DOM Level 1

DOM Level 1 wurde vom W3C im Oktober 1998 veröffentlich. Es besteht aus 2 Teilen:

1. **DOM1 Core**
   erlaubt den Zugriff auf die Struktur eines XML-Dokumentes
2. **DOM1 HTML**
   eine Erweiterung für HTML-spezifische Objekte

DOM Level 2

DOM Level 2 besteht aus 6 Teilen:

1. **DOM2 Core**
   der Kern wurde erweitert. Unter anderem um `getElementById`
2. **DOM2 HTML**
   wurde um die Möglichkeiten von DOM2-Core erweitert. Zum Beispiel um die Eigenschaft `contentDocument`
3. **DOM2 Views**
4. **DOM2 Events**
   Führt die Konzepte von Event Flow, Capture, Bubbling und Cancelation ein.
   Unter anderem die Methoden `addEventListener` und `handleEvent`
   Interfaces wie `EventTarget`, `EventListener`, `Event`, `DocumentEvent`, `MouseEvent`, `MutationEvent`
5. **DOM2 Style**
   dynamischer Zugriff auf CSS (Cascading Style Sheets)

6 **DOM2 Traversal and Range**

DOM2-Traversal-Interfaces wie `NodeIterator` and `TreeWalker` erlauben die Travesierung des Dokumentes.
DOM2-Range erlaubt das dynamische Erzeugen, Ändern und Löschen von Bereichen in einem Dokument.

DOM Level 3

DOM Level 3 besteht aus 5 Teilen:

1 **DOM3 Core**

der Kern wurde erweitert. Unter anderem um `adoptNode`, `strictErrorChecking`, `textContent`

2 **DOM3 Load and Save**

ermöglicht das Laden von einem XML-Dokument in ein DOM-Dokument. Und die Serialisierung eines DOM-Dokumentes in ein XML-Dokument.

3 DOM3 Validation

Validierung des Dokumentes.

4 **DOM3 Events**

Keyboard-Events wurden hinzugefügt.

5 **DOM3 XPATH**

Zugriff auf das DOM über XPATH

DOM Level 4

DOM Level 3 wurde 2015 vom W3C veröffentlicht.

Browser Object Model (BOM)

Das BOM wurde erstmals mit HTML5 in einen offiziellen Standard aufgenommen.
Es beinhaltet:

1) das Verschieben, Verkleinern, Vergrößern und Schließen von Fenstern
2) das Objekt `navigator` enthält Informationen über den Browser
3) das Objekt `screen` enthält die Auflösung des Fensters
4) Cookie Verwaltung
5) das Objekt `XMLHttpRequest`

Literal, Identifier, Token, Expression, Statement

Literale sind unveränderliche Werte, die für sich selbst stehen. **Variablen** stehen für änderbare Werte. **Konstanten** stehen für unveränderliche Werte. **Identifiers** sind Variablebezeichnungen. **Token** sind kleinstmögliche Einzelteile in der Syntax einer Programmiersprache.

Literal

Ein Literal ist ein fixer (konstanter) Wert, der für sich selbst steht (also keine Variable) und somit immutable (unveränderlich) ist.

String-Literal

Ein String-Literal steht in Anführungszeichen (doppelte oder einfache) und beinhaltet Zeichen:

Ziffern, Buchstaben
Control Characters (siehe **ASCII C0** Seite 225 und **ASCII C1** Seite 227)

Beispiel für ein String-Literal: `"bin ein literal"`

Multi-Line

Backslash erlaubt Zeilenumbrüche in einem String-Literal (im Source Code). Siehe l**ine continuation character** Seite 286).
Beispiele:

```
console.log (typeof "33");          // string
console.log (typeof "bin text");    // string
console.log (typeof "bin \
text");                             // string
```

Escape Sequence (Fluchtsequenz)

Damit Steuerzeichen in einem String-Literal nicht interpretiert werden, müssen sie escaped werden. Dies geschieht, indem der **Escape-Character** Backslash (\) vorangestellt wird. Der Escape-Character wird auf deutsch **Fluchtsymbol** genannt. Ein escaped Backslash (\\) ist ein **Literal Backslash** (der Backslash steht für sich selbst).
Siehe auch **ASCII C0** auf Seite 226, und **ASCII C1 auf** Seite 227

Als Beispiel ein String der in doppelten Anführungzeichen steht:

```
"Er ist cool."
```

Kommt ein Anführungszeichen im String vor, muss dieses escaped werden:

```
"Er ist \"cool\"."
```

Ergibt als Output:

```
Er ist cool.
```

Control Character	Beschreibung	ASCII-Wert	
\\	literal backslash		
\a	alert (bell)	07	BEL
\b	backspace	08	BS
\f	formfeed	12	FF
\n	newline (line feed)	10	LF
\r	carriage return	13	CR
\h	horizontal tab	09	HT
\v	vertical tab	11	VT
\u*nnnn*	hexadecimal unicode value nnnn		
\x*nn*	hexadecimal ascii value nn		

Anführungszeichen

Drei Arten von Anführungszeichen sind möglich:

"	doppeltes Anführungszeichen (Double Quotes)	`console.log("x ist 3. ");`
'	einfaches Anführungszeichen (Single Quotes)	`console.log('x ist 3.');`
`	backtick. Ab ES6 für **Template Strings**	`var x=3; console.log(`x ist ${x}.`)`

Number Literal

Ein **Number Literal** ist eine Zahl. Siehe Datentyp `number` auf Seite 299 .

Beispiel für ein Number-Literal: `123.456`

```
console.log (typeof 3);            // number
console.log (typeof 3.3);          // number
```

Boolean Literal

Ein **Boolean Literal** hat nur 2 mögliche Werte: `true` und `false`. Siehe auch Datentyp `boolean` auf Seite 299 .

Beispiel:

```
var foo=true; console.log(foo +' ' + typeof foo);          // true boolean
var foo=false; console.log(foo +' ' + typeof foo);         // false boolean
```

Exponent Literal

Der Buchstabe **e** ist der Exponent Literal. Er verschiebt das Komma in einer Zahl nach rechts oder nach links.

Die Zahl nach e bestimmt die Anzahl der Kommastellen.

Beispiel1:

```
var y = 12.34e-1            // Komma um eine Stelle nach links

console.log (y)
```

Ergibt als Output:

```
1.234
```

Beispiel2:

```
var y = 12.34e+1            // Komma um eine Stelle nach rechts

console.log (y)
```

Ergibt als Output:

```
123.4
```

Variable, Konstante

Variablen beinhalten Werte ("*Variablen stehen für einen Wert*"). Der Variablename wird **Identifier** genannt.
Eine Sonderform von Variablen sind Konstanten. **Konstante**n können nach ihrer Initialisierung (Wertzuweisung) ihren Wert nicht mehr ändern (siehe Initialisierung Seite 296).

Welche Werte eine Variable aufnehmen kann, hängt vom **Datentyp** (Seite 298) der Variable ab.

Vor der erstmaligen Verwendung einer Variable, muss sie deklariert (Seite 287) werden. Dies geschieht durch die Keywords `let`, `const` und `var`.

Wird bei der Deklaration auch ein Wert zugewiesen, wird dies **Initialisierung** (Seite 296) genannt. Dieser Erstwert wird auch **Initial Value** genannt.

Identifiers

Identifier Names sind Bezeichnungen von Variablen (Konstante, Objekte und Funktion).

1. Namen dürfen **beginnen** mit:

   Buchstaben

   Dollarzeichen (siehe Seite 228). Bei Verwendung von jQuery, beziehen sich diese auf **wrapped objects**.

   Unterstrich (siehe Seite 229)
2. Namen dürfen Ziffern beinhalten.
3. Namen düfen **kein reserviertes Wort** sein (siehe Keywords Seite 429).

Konstanten werden meist **großgeschrieben** (alle in der Variable vorkommenden Buchstaben).
Klassen beginnen meist mit einem Großbuchstaben.
JavaScript ist case-sensitive: `foo` und `Foo` sind 2 unterschiedliche Variablen.

Camel case

Bei **Camel Case** werden die einzelnen Wörter (einer Variablebezeichnung) ohne Trennzeichen zusammengeschrieben.
Der erste Buchstabe eines Wortes wird großgeschrieben (dadurch ergeben sich optisch Kamelhöcker).
Beispiele:

```
currentCounterFlag=0; whitePlayerStatus=2;
```

Snake case

Bei **Snake Case** werden die einzelnen Wörter (einer Variablebezeichnung) mit dem Unterstrich getrennt (der Unterstrich erinnert optisch an eine Schlange).
Beispiel:

```
current_counter_flag=0; white_player_status=2;
```

Token

Token sind kleinstmögliche Einzelteile in der Syntax einer Programmiersprache:

1) Keyword
   Beispiel: `while`
2) Identifier
   Beispiel: `binVariable`
3) String-Literal
   Beispiel: `"bin text"`
4) Number-Literal:
   Beispiel: `123.456`
5) Seperator Character `.,;(){}[]`
   Beispiel: `{`
6) Operator
   Beispiel: `+`

Expression (Ausdruck)

Eine Expression erzeugt einen Wert. Kann eine einzelne Variable oder ein einzelnes Literal sein. Oder eine Kombination aus Literalen, Operatoren, Variablen und Rückgabewerten von Funktionen sein.

Für Expressions bei **bedingter Flusskontrolle** siehe Seite 382. Für Datentyp `boolean` siehte Seite 304.

Beispiele:

```
9
"bin ein text"
true
3+4
5*x
meineFunktion()-7+x
```

Statement

Ein Statement ist eine Anweisung, die eine Aktion durchführt.

Als Beispiel eine Zuweisung: `foo = 9 +1`
`9+1` ist eine Expression (als Summe ergibt die Expression den Wert `11`, als Boolean den Wert `true`). Mit dem Zuweisungsoperator (Gleichheitszeichen) wird der Variable der Wert 11 zugewiesen.

Semikolon

Bei mehr als einem Statement pro Textzeile, müssen Statements durch Semikolons getrennt werden.

Beispiel: `foo1 = 9+1; foo2= 7+1;`

Das Semikolon am Ende einer Zeile (oder am Ende eines Blockes) ist optional:
Beispiele:

```
foo = 9 +1
if (foo) {foo2=1; foo3=2}
```

Block Statements (Compound Statements)

Mehrere Statements werden mit geschweiften Klammern zu einem Block (**Compound Statements**) zusammengefasst.

Sie werden bei der Flusskontrolle verwendet (siehe Seite 380).

Beispiel: `if (foo) {foo2=1; foo3=2;}`

Dot Notation

Die Zugriffsyntax auf ein Objekt erfolgt über die **Punkt-Notation** (Dot Notation). Beispiel:

```
document.write("bin ein Text")
```

`document` ist das Objekt, `write` ist die Funktion und `"bin ein Text"` ist der Wert, der an die Funktion übergeben wird.

Square Bracket Notation

Der Zugriff auf eine Eigenschaft (Property) über eckige Klammern (wie bei einem assoziative Array):

```
var maus = {fell: "braun", id: 32, "magKaese": true}
console.log('Mag Käse: ' + maus.magKaese);          // Dot Notation
console.log('Mag Käse: ' + maus["magKaese"]);       // Square Bracket Notation
```

Siehe auch **Dot Notation** auf Seite 316.

Line Continuation Character

Der **Line Continuation Character** ist der backslash. Er erlaubt Zeilenumbrüche in einem String (im Source Code).

Beispiel:

```
document.
body.bgColor='#\
990000';
```

Das Beispiel ohne Backslash ist fehlerhaft und erzeugt einen Error:

```
document.
body.bgColor='#
990000';
```

Fehlermeldung in der Konsole: `Uncaught SyntaxError: Invalid or unexpected token`

Source Code Remark

Mit // wird ein einzeiliger Kommentar im Source Code hinzugefügt, mit /* und */ ein mehrzeiliger.

single line comment

Ein // (2 forward slashes) ermöglicht einen einzeiligen Remark (Comment) im SourceCode.

Beispiel:

```
foo++;   // erhöhe die Variable um 1
```

multi-line comment

Ein /* und */ ermöglicht eine mehrzeilige Quelltext-Bemerkung.

Beispiel:

```
/* bin ein
mehrzeiliger Remark */
```

Commenting Out

Das Umwandeln von Statements zu Remarks, wird **Commenting Out** genannt:

Beispiel:

```
// foo=foo+1;
foo++;
```

Deklaration

Die Deklaration von Variablen ist optional. Bei Verwendung von `"use strict"` ist die Deklaration verpflichtend. Das Keyword für eine Deklaration ist **`var`**. Ab ECMA-Script 6 gibt es für Block-Variablen das Keyword **`let`** und **`const`**. Deklarationen sollten am Beginn des jeweiligen Scope stehen und vorallem vor der ersten Verwendung der Variable.

var

Mit dem Keyword **`var`** wird eine Variable deklariert. Als Beispiel eine Variable namens `foo`:

```
var foo;
```

Ohne gleichzeitiger Wertzuweisung (Initialisierung), hat die Variable den Wert `undefined`.
Somit ist eine Variable-Deklaration ohne Wertzuweisung identisch mit:

```
var foo=undefined;
```

Mit nur einem `var` können auch mehrere Variablen deklariert werden (mit einem Beistrich getrennt):

```
var foo1, foo2, foo3;
```

Value Assignment

Wird bei der Deklaration gleichzeitig ein Wert zugewiesen (**Value Assignment**), wird dies **Initialisierung** (Seite 296) genannt. Der zugewiesene Wert (bei der Erstzuweisung) wird **Initial Value** genannt.

Mehrfach-Deklaration

Beispiel: eine reine Deklaration und zwei Deklarationen mit Wertzuweisung (Initialisierung, siehe Seite 296):

```
var foo0, foo1=2, foo2="bin text";

console.log(foo0 + ' ' + typeof (foo0));                    // undefined       undefined
console.log(foo1 + ' ' + typeof (foo1));                    // 2               number
console.log(foo2 + ' ' + typeof (foo2));                    // bin text        string
```

Ergibt als Output:

```
undefined  undefined
2  number
bin text  string
```

Wiederholte Deklaration

Eine erneute Deklaration (der gleichen Variablen) führt zu keinem Fehler (im Gegensatz zu `let`, siehe Seite 293).
Beispiel:

```
var foo;
var foo;
```

Declaration Hoisting

Bei der Deklaration mit `let` und `const` muss die Deklaration zwingend vor der erstmaligen Verwendung der Variable gesetzt werden:

```
x=1                                          // x is not defined
let x;
console.log(x)
```

Erzeugt als Output:

```
Uncaught ReferenceError: x is not defined
```

Bei der `var`-Deklaration geschieht dies automatisch und wird **Declaration Hoisting** genannt: Eine bestehende Deklaration wird an den Anfang des Scope verschoben (also an den Anfang des jeweiligen Programmblockes).

Beispiel1: Aus diesem Beispiel:

```
"use strict"
x=1
var x;
console.log(x)
```

wird intern beim Programmstart:

```
var x
x=1
console.log(x)
```

Beispiel2: Bei einer Initialisierung wird nur die Deklaration verschoben (die Zuweisung bleibt):

```
x=1
var x=2;
console.log(x)
```

wird intern beim Programmstart:

```
var x
x=1
x=2
console.log(x)
```

Beispiel3: **Declaration Hoisting** innerhalb einer Funktion (Seite 399 für Function Declaration Hoisting):

```
function foo ()
   {
      y++
      var y              // var ist nur im function Scope gültig
   }
foo ()
```

wird intern beim Programmstart:

```
function foo ()
   {
      var y              // die Deklaration wird an den Anfang des gegenwärtigen Scope verschoben
      y++
   }
foo ()
```

Beispiel4: **Kein Declaration Hoisting** bei `let` und `const`:

```
function foo ()
   {
      y++
      let y
   }
foo ()
```

ergibt intern beim Programmstart keine Änderung und einen Compiler-Error:

```
function foo ()
   {
      y++                          // Uncaught ReferenceError: y is not defined
      let y
   }
foo ()
```

Scope von var

Ist die `var` Variable im äußersten Scope (Global Scope) deklariert, ist sie im gesamten Programm verfügbar.

Ist die `var` Variable in einer Funktion deklariert, ist sie nur in der Funktion gültig (inklusive aller Unterblöcken und anderen Funktionen).

Scope siehe Seite 422, **Function Scope** siehe Seite 407.

Beispiel1: `var` und `let` im Global Scope deklariert (überall sichtbar)

```
var x=1                                          // Global Scope
let y=2                                          // Global Scope

 {
    console.log ('1 x: ' + typeof x)             // beide im Unterblock sichtbar
    console.log ('1 y: ' + typeof y)
 }

function foo () {                                // beide in der Funktion sichtbar
    console.log ('2 x: ' + typeof x)
    console.log ('2 y: ' + typeof y)
 }

foo ()
```

Beispiel2: `var` und `let` in einem Unterblock deklariert (für `let` Seite 293)

```
 {
    var x=1                                      // Global Scope
    let y=2                                      // Block Scope (Seite 425)

    console.log ('1 x: ' + typeof x)             // number
    console.log ('1 y: ' + typeof y)             // number
 }

function foo () {
    console.log ('2 x: ' + typeof x)             // number
    console.log ('2 y: ' + typeof y)             // undefined
 }

foo ()

console.log ('3 x: ' + typeof x)                 // number
console.log ('3 y: ' + typeof y)                 // undefined
```

Beispiel3: `var` und `let` in einer Funktion deklariert (für `let` Seite 293)

```
function foo ()
 {
    var x=1                                          // Function Scope (Seite 407)
    let y=2

    console.log ('2 x: ' + typeof x)                 // number
    console.log ('2 y: ' + typeof y)                 // number
 }

foo ()

console.log ('3 x: ' + typeof x)                     // undefined
console.log ('3 y: ' + typeof y)                     // undefined
```

"use strict"

Dieses Literal sollte am Beginn des jeweiliges Scope stehen.
Zum Beispiel: Ist der Einzeiler innerhalb einer Funktion, dann gilt `'use strict'` nur für diese Funktion.

Das Pragma ist ein Hinweis für den Compiler:

1) Bei `"use strict"` kann das Keyword `with` (Seite 428) nicht mehr verwendet werden.
2) Variable-Deklarationen sind bei Verwendung von `"use strict"` nicht mehr optional.
3) Identifier für Funktionsnamen und Funktionsparameter dürfen nicht `eval` oder `arguments` lauten
4) Identifier der Funktionsparameter müssen eindeutig sein (darf jeweils nur einmal vorkommen). Siehe Seite 403
5) der Versuch eine read-only Objekt-Eigenschaft zu ändern, erzeugt einen Fehler

Variable Deklaration

Deklarationen sind bei Verwendung von `"use strict"` nicht mehr optional. Die Verwendung einer Variable ohne Deklaration, erzeugt einen Fehler. Der Einzeiler muss zu Beginn des Scope gesetzt sein.

Für Deklaration Seite 287.

Beispiel1: x ohne Deklaration

```
"use strict"
x=1
console.log(x)
```

Ergibt als Output:

```
Uncaught ReferenceError: x is not defined
```

Beispiel2: x mit Deklaration

```
'use strict'
var x
x=1
console.log(x)
```

Ergibt als Output:

```
1
```

read-only Objekt-Eigenschaft

Bei `"use strict"` erzeugt der Versuch, eine read-only Objekt-Eigenschaft zu ändern, einen Fehler.

Beispiel: die Eigenschaft `rasse` vom Objekt `Hund` ist nicht veränderbar

```
var hund = {}

Object.defineProperty(hund, "rasse", { writable: false,  value: "Dackel" }   )

hund.rasse = "Schäfer"
```

Erzeugt als Output:

```
Cannot assign to read only property 'rasse' of object '#<Object>'
```

eval()

Bei "use strict" bleiben intern deklarierte Variablen im eval Scope.

Beispiel 1a: ohne "use strict" gelangt die Variable x in den äußeren Scope

```
eval("var x = 10;");

console.log (x)
```

Erzeugt als Output:

```
10
```

Beispiel 1b: mit "use strict" bleibt die Variable x im eval Scope

```
"use strict"

eval("var x = 10;");

console.log (x)
```

Ergibt als Output einen Error:

```
Uncaught ReferenceError: x is not defined
```

Beispiel 1c:

```
"use strict"

var wert = eval("var x = 10;");

console.log (wert)

console.log (typeof x)
```

Erzeugt als Output:

```
10
undefined
undefined
```

this coercion

Wenn this den Wert undefined oder null hat, wird bei "use strict" nicht mit **Call-Site** vom **Global Execution Context** verbunden (siehe Seite 321).

Beispiel 1a:

```
window.color = "red";
function Farbe() {
     console.log (this.color);
   }
Farbe();                          // red
```

Beispiel 1b:

```
"use strict"
window.color = "red";
function Farbe() {
    console.log (this.color);     // Uncaught TypeError: Cannot read property 'color' of undefined
   }

Farbe();
```

const

Mit dem Keyword **const** (ab ECMA-Script 6) wird eine Konstante (im **Block Scope**) definiert:

```
const FOO=4;
```

Für gewöhnlich werden Konstanten großgeschrieben (also FOO, statt foo).

Konstanten sind nur in dem Programmblock gültig (sichtbar), in dem sie deklariert worden sind.
Siehe let Seite 293 und **Block Scope** Seite 425.

Bei const muss gleichzeitig eine Wert-Initialisierung stattfinden (im Gegensatz zu var und let). Siehe auch Initialisierung Seite 296.
Beispiel:

```
const FOO;
```

Ergibt als Fehler (Chrome): Uncaught SyntaxError: Missing initializer in const declaration

let

Ab ECMA-Script 6 (Harmony) gibt es das Keyword let.
Sein Gültigkeitsbereich liegt innerhalb des { } Blockes (inklusive allen Unterblöcken und Funktionen), in dem es definiert worden ist. Der Kontext (Schleife, if-else,...) des Blockes spielt keine Rolle.

Siehe const Seite 293 und **Scope** Seite 422.

Beispiel1: let ist in Unterblöcken sichtbar

```
var x=1
let y=2

 {
    console.log ('1 x: ' + typeof x)              // beide im Unterblock sichtbar
    console.log ('1 y: ' + typeof y)
 }

function foo () {                                 // beide in der Funktion sichtbar
    console.log ('2 x: ' + typeof x)
    console.log ('2 y: ' + typeof y)
 }

foo ()
```

Beispiel2: let in einem Unterblock deklariert (nur im Unterblock, aber nicht im Nebenblock sichtbar)

```
 {
    var x=1                                       // Global Scope
    let y=2                                       // Block Scope (Seite 425)

    console.log ('1 x: ' + typeof x)              // number
    console.log ('1 y: ' + typeof y)              // number
 }

function foo () {
    console.log ('2 x: ' + typeof x)              // number
    console.log ('2 y: ' + typeof y)              // undefined
 }

foo ()

console.log ('3 x: ' + typeof x)                  // number
console.log ('3 y: ' + typeof y)                  // undefined
```

Beispiel3: let in einer Funktion deklariert (nur in der Funktion sichtbar)

```
function foo ()
 {
    var x=1                                          // Function Scope (Seite 407)
    let y=2

    console.log ('2 x: ' + typeof x)                 // number
    console.log ('2 y: ' + typeof y)                 // number
 }

foo ()

console.log ('3 x: ' + typeof x)                     // undefined
console.log ('3 y: ' + typeof y)                     // undefined
```

Beispiel4a: Variable Masking (Seite 427) mit var und let

```
var foo=2;                              // erste Deklaration (im Global Scope)
 {
    let foo=3;                          // zweite Deklaration (im Block Scope)
                                        // das lokal foo überlagert das globale foo
    console.log('drin ' + foo);
 }

console.log('aussen ' + foo);           // das globale foo hat weiterhin den Wert 2
```

Ergibt als Output:

```
drin 3
aussen 2
```

Beispiel4b: das Beispiel mit var (Global Scope):

```
var foo=2;                                  // erste Deklaration (im Global Scope)
 {
    console.log('drin1 ' + foo);
    var foo=3;                              // zweite Deklaration (im Global Scope)
    console.log('drin2 ' + foo);
 }

console.log('aussen ' + foo);               // foo ist auch ausserhalb vom Block 3
```

Ergibt als Output:

```
drin1 2
drin2 3
aussen 3
```

Im Gegensatz zu var, darf mit let die gleiche Variable nicht noch einmal deklariert werden (zumindest nicht im gleichen Block).

Dieses Beispiel erzeugt einen Fehler: `{ let foo=1; let foo=4;}`

Dieses Beispiel erzeugt keinen Fehler: `{ let foo=1;} {let foo=4;}`

Value Assignment (Initialisierung)

Wird bei der Deklaration gleichzeitig ein Wert zugewiesen (**Value Assignment**), wird dies **Initialisierung** (Seite 296) genannt. Der zugewiesene Wert (bei der Erstzuweisung) wird **Initial Value** genannt.

Beispiel:

```
let x           // Deklaration
let y=1         // Deklaration und Initialisierung mit dem Wert 1 (Initial Value ist 1)
```

Keine Declaration Hoisting

Eine Variable darf in ihrem Block nicht vor ihrer `let`-Deklarierung verwendet werden.
Dieses Beispiel erzeugt einen Fehler:

```
{
  foo=0;
  let foo;
}
```

Ergibt als Output:

```
Uncaught ReferenceError: foo is not defined
```

Im Gegensatz zu `var` wird bei `let` und `const` kein **Declaration Hoisting** durchgeführt (eine automatische Deklaration am Anfang des Scopes, siehe Seite 288).

Temporal Dead Zone (TDZ)

Bei Deklarationen mit `let` und `const` wird von JavaScript kein **Declaration Hoisting** (Seite 288) durchgeführt (im Gegensatz zu `var`).

Der Bereich vom Scope Anfang bis zur Deklaration wird **Temporal Dead Zone** genannt, da die Variable in diesem Bereich nicht verwendet werden kann.

Beispiel1: die Abfrage mit dem `typeof` Operator (vor der Deklaration)

```
console.log(typeof foo)                    // Temporal Dead Zone

const foo=1
```

Ergibt als Output:

```
Uncaught ReferenceError: foo is not defined
```

Beispiel2: bei `var` wird von JavaScript das **Declaration Hoisting** (Seite 288) durchgeführt

```
console.log(typeof foo)

var foo=1
```

Ergibt als Output:

```
 undefined
```

Initialisierung

Die Erstzuweisung eines Wertes (**Initial Value**) wird Initialiserung genannt. Der Zuweisungsoperator (**Assignment Operator**) ist das Gleichheitszeichen. Die Wertzuweisung wird **Value Assignment** genannt.

Die Initialisierung sollte bereits bei der Deklaration (siehe Seite 287) stattfinden.

Number (Zahl)

Als Beispiel der Variable `foo` den Number-Literal `666` zuweisen (Literal siehe Seite 282):

```
let foo=666;
```

Der Konstante `foo` den Number-Literal `666` zuweisen:

```
const FOO=666;
```

String (Zeichenkette)

Der Variable `foo` den String-Literal `bin ein Text` zuweisen (Literal siehe Seite 282):

```
let foo='bin ein Text';
```

Der Konstante `foo` den String-Literal `bin ein Text` zuweisen:

```
const FOO='bin ein Text';
```

Anführungszeichen

Drei Arten von Anführungszeichen sind möglich:

"	doppeltes Anführungszeichen	single quotes
'	einfaches Anführungszeichen	double quotes
`	backtick. Ab ES6 für **Template Strings**	**backticks**

Multi-Line

Backslash erlaubt Zeilenumbrüche in einem String-Literal (im Source Code). Siehe **line continuation character** Seite 286).
Beispiele:

```
console.log (typeof "33");          // string
console.log (typeof "bin text");    // string
console.log (typeof "bin \
text");                             // string
```

Escape Sequence (Fluchtsequenz)

Damit Steuerzeichen in einem String-Literal nicht interpretiert werden, müssen sie escaped werden. Dies geschieht, indem der **Escape-Character** Backslash (\) vorangestellt wird. Der Escape-Character wird auf deutsch **Fluchtsymbol** genannt. Ein escaped Backslash (\\) ist ein **Literal Backslash** (der Backslash steht für sich selbst).
Siehe auch **ASCII C0** auf Seite 226, und **ASCII C1 auf** Seite 227

Als Beispiel ein String der in doppelten Anführungzeichen steht:

```
"Er ist cool."
```

Kommt ein Anführungszeichen im String vor, muss dieses escaped werden:

```
"Er ist \"cool\"."
```

Ergibt als Output:

```
Er ist cool.
```

Hello World

document.write()

Ein **Hello World!** mit `<script>` (siehe **Embedded Script** Seite 432) und `document.write()`:

```
<!DOCTYPE html>
<html>
<head> <title>Hello World mit document.write</title> </head>
<body>
 <script>
      document.write("Ade, schnöde Welt.");
 </script>
</body></html>
```

alert()

Ein **Hello World!** mit `<script>` (siehe **Embedded Script** Seite 432) und `alert()`:

```
<!DOCTYPE html>
<html>
<head> <title>Hello World</title> </head>
<body>
 <script>
      alert("Ade, schnöde Welt.");
 </script>
</body></html>
```

console.log()

Ein **Hello World!** mit `<script>` (siehe **Embedded Script** Seite 432) und `console.log()`:

```
<!DOCTYPE html>
<html>
<head> <title>Hello World</title> </head>
<body>
 <script>
      console.log("Ade, schnöde Welt.");
 </script>
</body></html>
```

Inline Script mit Event-Handler

Ein **Hello World!** mit dem Attribut `onclick` (siehe **Inline Script** Seite 432):

```
<!DOCTYPE html>
<html>
<head> <title>Hello World</title> </head>
<body>
 <script>
      <a href="#" onclick=" alert('Ade, schnöde Welt'); ">klick mich</a>
 </script>
</body></html>
```

Inline-JavaScript sollte nur dann verwendet werden, wenn der Author keinen Zugriff auf das eingebettete oder externe JavaScript besitzt. Siehe auch **Inline Style** auf Seite 210. Siehe auch **unobtrusive JavaScript** Seite 430.

Pseudo-URL

```
<!DOCTYPE html>
<html>
<head> <title>Hello World</title> </head>
<body>
    <a href="javascript:alert('Ade, schnöde Welt');">klick mich</a>
</body></html>
```

Datentypen (Data Types)

JavaScript ist ein eine **loosely (weakly) typed language**. Bei der **Deklaration** (siehe Seite 287) ist eine Angabe des Datentyps nicht möglich.

Complex Data Types (Reference Values)

Zu den **Complex Data Types** zählen **Arrays** (Seite 342), **Funktionen** (Seite 393) und **Objekte** (Seite 310).

Array:

```
let foo = [1,2,3]
console.log (typeof foo)              // Datentyp object     Seite 342
```

Funktion:

```
function foo () {}
console.log (typeof foo)              // Datentyp function  Seite 393      (ist aber auch ein Objekt)
```

Objekt:

```
let foo = {a:'1', b:'2', c:'3'}
console.log (typeof foo)              // Datentyp object     Seite 310)
```

Reference Values werden im Heap gespeichert.
Bei einem **Variable Copy** (siehe Seite 375) wird die Referenz kopiert (und nicht die Werte).
Die Referenz ist ein Zeiger, der auf die Daten im Heap zeigt.

Primitive Data Types (Primitive Values)

JavaScript kennt 6 **Primitive Data Types.** Sie sind Skalare und können somit immer nur einen Wert enthalten (im Unterschied zu Objekten (Seite 310), die mehrere Werte gleichzeitig enthalten können).

`number`	Zahl (Seite 299). Für das Number-Objekt siehe Seite 334. Werte, die auch vom Datentyp `number` sind: `NaN`, `Infinity`, `-Infinity`
`string`	Zeichenkette (Seite 306). Für das String-Objekt Seite 333.
`boolean`	logisches wahr (`true`) oder falsch (`false`). Siehe Seite 304. Nur diese 2 Werte sind möglich. Für Boolean-Objekt Seite 334.
`undefined`	steht für *nicht zuwiesener Datentyp* (zB bei einer Deklaration ohne Initialisierung). Nur dieser 1 Wert ist möglich. Seite 305.
`null`	steht für *nicht vorhandenen* Nur dieser 1 Wert ist möglich. Seite 305.
`symbol`	Seite 305.

Primitive Values werden im Stack gespeichert.
Bei einem **Variable Copy** (siehe Seite 375) werden die Werte kopiert .

number

JavaScript kennt für Zahlen nur den Datentyp `number`. Intern wird jede Zahl als **double-precision floating-point number** abgelegt (64bit IEEE 754). Für **bitwise operators** (Seite 372) wird aus **double** ein 32bit **integer** big endian. Siehe **endian** Seite 235.

Die Konstante `Number.MIN_SAFE_INTEGER` (`-9007199254740991`) enthält den Integer-Minimalwert und `Number.MAX_SAFE_INTEGER` (`9007199254740991`) den Maximalwert.

Ebenfalls vom Datentyp `number` sind diese Werte:

```
NaN
Infinity
-Infinity
```

Beispiel1:

```
let foo=123.456;
console.log(foo);
console.log(typeof foo);
```

Ergibt als Output:

```
number
```

Beispiel2:

```
let foo=123;
console.log(foo);
console.log(typeof foo);
```

Ergibt als Output:

```
number
```

Floating Point Number Comparison

Die Kombination aus loose typing und 64bit IEEE 754 ergibt Unschärfen, die vor allem bei Überprüfung auf Gleichheit zu Problemen führt. Zb ergibt 0.1 + 0.2 nicht 0.3

Beispiel: `0.1 + 0.2` ergibt nicht `0.3`

```
if (0.1 + 0.2 != 0.3)
   {
      console.log ('0.1 + 0.2  ist ungleich  0.3');
   }
console.log (0.1 + 0.2);      // 0.30000000000000004
```

Ergibt als Output:

```
0.1 + 0.2  ist ungleich  0.3
0.2
0.30000000000000004
```

Zahlenbasis

Werte können dezimal (Zahlenbasis 10), hexadezimal (Zahlenbasis 16) oder oktal (Zahlenbasis 8) sein:

```
foo=255;                    // dezimal
foo=0xff;                   // hexadezimal beginng mit 0x
foo=0o377;                  // octal beginnt mit 0o
```

Die Funktione `parseInt()` konvertiert eine Zahl (Datentyp `string`) von beliebiger Zahlenbasis (2 bis 36), nach Zahlenbasis 10 (siehe Seite 303).

Beispiel:

```
var foo=parseInt("ff",16)              // Zahlenbasis 16
console.log (foo)

var foo=parseInt("377",8)              // Zahlenbasis 8
console.log (foo)


var foo=parseInt("11111111",2)         // Zahlenbasis 2
console.log (foo)
```

Ergibt als Output:

```
255
255
255
```

number NaN

Der Wert `NaN` steht für **Not a Number** und ist vom Datentyp `number`. Überprüfung auf Wert `NaN` siehe Seite 301.

Beispiel1:

```
let foo=0/0;
console.log(foo);
console.log(typeof foo);
```

Ergibt als Output:

```
NaN
number
```

Beispiel2:

```
let foo=Number("bin text");
console.log(foo);
console.log(typeof foo);
```

Ergibt als Output:

```
NaN
number
```

number Infinity

Der Wert `Infinity` steht für unendlich und ist vom Datentyp `number`. Siehe auch Multiplikator Seite 364.

In PHP nennt sich der Wert INF und ist vom Datentyp double.

Beispiel:

```
foo1=Math.pow(10,1000);
foo2=Math.log(0);
console.log(foo1 + ' = ' + typeof foo1);
console.log(foo2 + ' = ' + typeof foo2);
```

Ergibt als Output:

```
Infinity = number
-Infinity = number
```

Der Wert `Infinity` minus `Infinity` ergibt `NaN`.

Beispiel:

```
console.log(typeof (Infinity-Infinity))
console.log(Infinity-Infinity)
```

Ergibt als Output:

```
number
Nan
```

isNaN()

Nur mit der Funktion `isNaN()` oder `Object.is()` (Seite 367) kann überprüft werden, ob eine Variable `NaN` ist:

```
foo=0/0;
console.log( NaN===NaN )            // Übereinstimmung nicht erkannt
console.log( foo===NaN )            // Übereinstimmung nicht erkannt
console.log( isNaN(foo) )                       // Übereinstimmung erkannt
console.log( Object.is(foo,NaN) )               // Übereinstimmung erkannt
```

Ergibt als Output:

```
false
false
true
true
```

Number()

Die Funktion `Number()` wandelt den übergebenen Wert (`string` oder `boolean`) in eine Zahl um.

Syntax:

```
Number (Parameter);
```

Der Boolean-Wert `true` ergibt 1, und `false` ergibt 0:

```
foo=Number(true);
console.log(foo + ' ' + typeof foo);   // 1 number

foo=Number(false);
console.log(foo + ' ' + typeof foo);   // 0 number
```

Ein String, der nur aus Ziffern besteht, wird erfolgreich umgewandelt:

```
foo=Number("32");
console.log(foo + ' ' + typeof foo);      // 32 number
```

Ein String, der nicht nur aus Ziffern besteht, wird zum Wert `NaN` (vom Datentyp `number`):

```
foo=Number("3a");
console.log(foo + ' ' + typeof foo);      // NaN number
```

Sind die ersten Zeichen (in einem String) Nullen, dann werden diese weggeworfen:

```
foo=Number("000000032");
console.log(foo + ' ' + typeof foo);      // 32 number
```

Ist das erste Zeichen in einem String ein Plus, dann wird dieses weggeworfen:

```
foo=Number("+32");
console.log(foo + ' ' + typeof foo);      // 32 number
```

parseInt()

Die Funktion `parseInt()` wandelt die übergebene Zahl (`number` oder `string`) in eine Ganzzahl (Datentyp `number`) um. Intern ist die Ganzzahl weiterhin als **double-precision floating-point number** abgelegt (64bit IEEE 754).

Syntax:

```
parseInt (Parameter [, Zahlenbasis]);
```

Das Argument kann als Zahlenbasis 2 bis 36 haben (konvertiert wird immer nach Zahlenbasis 10).
Default für den Zahlenbasis-Parameter ist 10. Beispiele für Zahlenbasis-Konvertierung siehe Seite 300.

Beispiel1 - Wandelt `1.7` in `1`:

```
foo=parseInt(1.7,10);
console.log(foo + ' ' + typeof foo);      // 1 number
```

Beispiel2 - Wandelt eine Kommazahl, die sich in einem String befindet, in eine Ganzzahl:

```
foo=parseInt("1.7",10);
console.log(foo + ' ' + typeof foo);   // 1 number
```

Beispiel3: Wirft ab dem ersten Buchstaben in einem String alle Zeichen weg:

```
foo=parseInt("7a6",10);
console.log(foo + ' ' + typeof foo);   // 7 number
```

Beispiel4: Ist das erste Zeichen in einem String ein Pluszeichen, dann wird dieses weggeworfen:

```
foo=parseInt("+33",10);
console.log(foo + ' ' + typeof foo);   // 33 number
```

parseFloat()

Die Funktion `parseFloat()` wandelt den übergebenen String in eine (Komma-)Zahl um (Datentyp `number`).
Intern ist die Zahl als **double-precision floating-point number** abgelegt (64bit IEEE 754).

Syntax:

```
parseFloat (Parameter);
```

Beispiel1 - Wandelt eine Kommazahl in einen String:

```
foo=parseFloat("1.7");
console.log(foo + ' ' + typeof foo);   // 1.7 number
```

Beispiel2: Wirft ab dem ersten Buchstaben in einem String alle Zeichen weg:

```
foo=parseFloat("7a6");
console.log(foo + ' ' + typeof foo);   // 7 number
```

Beispiel3: Ist das erste Zeichen in einem String ein Pluszeichen, dann wird dieses weggeworfen:

```
foo=parseFloat("+44");
console.log(foo + ' ' + typeof foo);   // 44 number
```

boolean

Der Datentyp `boolean` hat nur 2 mögliche Werte:

```
true
false
```

Diese Werte ergeben `false`:

```
NaN                                                    In PHP ergibt NAN den Wert true
undefined
0
-0
""
''
false
```

Diese Werte ergeben `true`:

```
Infinity
"0"
"NaN"
true
8.3
```

Beispiel:

```
var foo=Infinity;
if (foo) console.log(foo + ' ist true. ' + typeof foo)
```

Ergibt als Output:

```
foo ist true. number
```

truthy, falsy

Wenn ein Wert (bzw Expression) zu `true` konvertiert werden kann, wird er **truthy** (siehe mozilla.org) genannt. Wenn ein Wert zu `false` konvertiert werden kann, wird er **falsy** genannt (siehe mozilla.org).

Siehe auch Logische Operatoren (Seite 370).

Boolean als Objekt

Das Boolean Objekt ist ein Objekt-Wrapper für einen booleschen Wert.

Syntax, um ein Objekt mit dem Konstruktor `Boolean` zu erzeugen:

```
new Boolean([value])
```

Ist der optionale Parameter `0`, `null`, `false`, `NaN`, `undefined` oder ein Leerstring(""), so besitzt das Objekt den Wert `false`. Alle anderen Werte, zB ein Objekt oder die Zeichenkette (string) "false", erzeugen ein Objekt mit dem Wert true.

Ein Boolean-Objekt darf nicht anstelle eines primitiven Booleanwertes verwendet werden.

Beispiel1:

```
var foo = new Boolean(false);

if (foo) {
      console.log('expression ist true. type: ' + typeof foo + ' foo: ' + foo)
   }
```

Ergibt als Output:

```
expression ist true. type: object foo: false
```

Beispiel2: Boolean-Objekt wird automatisch in `number` umgewandelt (`true` ist 1, `false` ist 0)

```
var foo = new Boolean (true)

if (1 === 1) { console.log('true ergibt 1'); }
```

Ergibt als Output:

```
 true ergibt 1
```

Boolean() Funktion

Die Funktion wertet eine Expression aus. Der Rückgabewert ist das Ergebnis der Auswertung und somit `true` oder `false`.

```
Boolean([Expression])
```

Beispiel:

```
console.log( Boolean(false));
console.log( Boolean("false"));
console.log( Boolean(3-3));
```

Ergibt als Output:

```
false
true
false
```

undefined

Der Datentyp `undefined` hat nur einen möglichen Wert: `undefined`
Nach der Deklaration und vor der Initialisierung, hat eine Variable den Datentyp `undefined`.
Beispiel:

```
var foo;
console.log(typeof foo);
```

null

Der Datentyp `null` hat nur einen möglichen Wert: `null`
`null` steht für einen nicht existierenden Wert.

Aus historischen Gründen gibt `typeof` bei `null` ein `object` zurück (obwohl `null` ein eigener Datentyp ist).

symbol

Mit ES6 wurde der Datentyp `symbol` eingeführt. Ein Symbol stellt ein eindeutiges Token dar.
Symbols werden mit dem Konstruktor `Symbol()` erstellt (und nicht mit dem Keyword `new`).

```
const BLAU = Symbol();
const ROT = Symbol ("Blutrot");
```

string

Der Datentyp `string` besteht aus Zeichen (Ziffern, Buchstaben und Sonderzeichen).
Siehe **String-Literal** auf Seite 282, **Escape Sequence** auf Seite 282, **Zeichensatz-Kodierung** auf Seite 223 und **UTF-Encoding** auf Seite 234.

Für **String-Funktionen** siehe Seite 309.

Multi-Line

Backslash erlaubt Zeilenumbrüche in einem String-Literal (im Source Code). Siehe l**ine continuation character** Seite 286).
Beispiele:

```
console.log (typeof "33");          // string
console.log (typeof "bin text");    // string
console.log (typeof "bin \
text");                             // string
```

Escape Sequence (Fluchtsequenz)

Damit Steuerzeichen in einem String-Literal nicht interpretiert werden, müssen sie escaped werden. Dies geschieht, indem der **Escape-Character** Backslash (\) vorangestellt wird. Der Escape-Character wird auf deutsch **Fluchtsymbol** genannt. Ein escaped Backslash (\\) ist ein **Literal Backslash** (der Backslash steht für sich selbst).
Siehe auch **ASCII C0** auf Seite 226, und **ASCII C1 auf** Seite 227

Als Beispiel ein String der in doppelten Anführungzeichen steht:

```
"Er ist cool."
```

Kommt ein Anführungszeichen im String vor, muss dieses escaped werden:

```
"Er ist \"cool\"."
```

Ergibt als Output:

```
Er ist cool.
```

Anführungszeichen

Drei Arten von Anführungszeichen sind möglich:

"	doppeltes Anführungszeichen	single quotes
'	einfaches Anführungszeichen	double quotes
`	backtick. Ab ES6 für **Template Strings**	**backticks**

toString() Methode

Die Datentypen `object` (inklusive Array, Seite 361), `number`, `string`, `boolean` besitzen die Methode `toString()`.

Beispiel1: Zahl

```
var foo=9.4;
var foostring=foo.toString();

console.log(typeof foo);
console.log(foostring + ' / type: '+typeof foostring);
```

Ergibt als Output:

```
number
9.4 / type: string
```

Beispiel2: Array

```
var foo=Array.of(10,4);
var foostring=foo.toString();
console.log(foostring);
console.log('type: ' + typeof foostring);
```

Ergibt als Output:

```
10,4
type: string
```

Beispiel3: Boolean

```
var foo=true;
var foostring=foo.toString();

console.log(typeof foo);
console.log(foostring + ' / type: '+typeof foostring);
```

Ergibt als Output:

```
boolean
true / type: string
```

String-Vergleich

String-Comparison ist case sensitive. Siehe Vergleichsoperatoren Seite 366:

```
var foo = Boolean('carol'=='Carol)
console.log(foo);
```

Ergibt als Output:

```
false
```

Für einen **case insensitive Vergleich** wird entweder die Methode `toLowerCase()` oder `toUpperCase()` verwendet:

```
console.log( 'A'.toLowerCase() == 'a' );
console.log( 'a'.toUpperCase() == 'A' );
```

Ergibt als Output:

```
true
true
```

Bei **Relationaloperatoren** (Seite 367) werden die ASCII-Werte (Seite 231) verglichen.

```
console.log('a' > 'A');         // true
console.log('a' > 'b');         // false
console.log('a' > '1');         // true
console.log('&' < '1');         // true
```

Ergibt als Output:

```
true
false
true
true
```

Bei **unterschiedlichen Datentypen** wird (für den Vergleich) der Datentyp automatisch geändert:

```
console.log('1' == '1');
console.log('1' == 1);
```

Ergibt als Output:

```
true
true
```

Nur mit dem **Strict Equality Operator** (Seite 367) kann auch auf Gleichheit des Datetyps überprüft werden:

```
console.log('1' === '1');
console.log('1' === 1);
```

Ergibt als Output:

```
true
false
```

String Funktionen

```
String.fromCharCode()        Code-Points in Zeichen umwandeln
String.fromCodePoint()       Code-Points in Zeichen umwandeln
```

String.fromCharCode()

Übergabewerte sind 1 bis n Code-Points (siehe Seite 223)

Rückgabewert ist der String, der aus den einzelnen Zeichen gebildet wird

Syntax:

```
String.fromCharCode(num1 [,numN])
```

Beispiel:

```
var foo = String.fromCharCode(81, 171, 565, 0x1D306);
console.log(foo)
```

Ergibt als Output:

```
Q«ɳ 팆
```

String.fromCodePoint ()

Übergabewerte sind 1 bis n Code-Points (siehe Seite 223)

Rückgabewert ist der String, der aus den einzelnen Zeichen gebildet wird

Exception bei einem ungültigen Wert: `RangeError: argument is not a valid code point`

Syntax:

```
String.fromCodePoint (num1 [,numN])
```

Beispiel:

```
var foo = String.fromCodePoint (81,171, 565, 466, 667);
console.log(foo)
```

Ergibt als Output:

```
Q«ɳǒɕ
```

Objekt

Ein Objekt ist ein Datentyp, der mehr als einen Wert (gleichzeitig) enthalten kann. Es ist somit kein Skalar.
Diese 3 Datentypen sind Objekte:

Objekte, Funktionen, Arrays	sind	**Complex Data Types**, siehe Seite 298 Funktionen Seite 393 Array Seite 342

Diese 6 Datentypen sind Skalare:

Number, String, Boolean, Symbol, null, undefined	sind	**Primitive Data Type** siehe Seite 298

Übersicht

Ein Objekt ist eine **Dynamic Collection of Properties** (eine Sammlung von Eigenschaften, die sich jederzeit ändern lassen). Eine Objekt-Eigenschaft besteht aus dem Eigenschaftsnamen (**Key Name**), dem Wert (**Value**) und dem **Property Descriptor** (Seite 319).

Der Wert einer Eigenschaft (Property) kann von jedem Datentyp (Seite 298) sein.

Der Zugriff auf eine Eigenschaft (Property) erfolgt über einen **Unique Key**. Der Key ist ein Identifier (Seite 284) vom Datenstyp string (Seite 306) oder symbol (Seite 305).

Die Zugriff-Notation (Seite 316) ist entweder Objektname.Eigenschaftsname oder Objektname[Eigenschaftsname]

JavaScript ist kennt keine Klassen (Object-based Language). Die Vererbung von Objekten ist prototype-based (ein Funktionsobjekt dient als Konstruktor).

Property

Die Inhalte eines Objektes werden **Properties** (manchmal auch **Members)** genannt, siehe Seite 316.
Ein **Property** besteht aus:

1)	Keyname (Identifier)	`string` oder `symbol`	für Identifier Seite 284
2)	und einem Wert	beliebiger Datentyp (auch Funktionen)	für Objekt-Funktionen Seite 317 für Objekte Seite 317
3)	einem Descriptor	Datentyp Object (enumerable, writable,...)	Seite 319

Property Descriptors

Der **Property Descriptor** ist ein Objekt, das eine einzelne Eigenschaft um zusätzliche Attribute (Beschreibungen) erweitert.
Der Descriptor ist entweder ein **Date Descriptor** oder ein **Accessor Descriptor** (Seite 319).
Ein **Data Descriptor** enthält: `configurable, enumerable, value, writable`
Ein **Accessor Descriptors** enthält: `configurable, enumerable, get, set`

Zugriff-Notation

Für den Zugriff auf ein Property gibt es 2 Möglichkeiten:

1)	Dot Notation	`katze.magMaeuse=true;`	**.**	Seite 316
2)	Square Bracket Notation	`katze["magMaeuse"]=true`	**[ ]**	Seite 316 (Member Access Operator)

Property löschen

Mit dem Keyword `delete` kann eine Eigenschaft gelöscht werden (Seite 318).

Syntax: `delete Objektname.Eigenschaftsname`

Objekt-Erzeugung

Ein Objekt kann auf 3 Arten erzeugt werden (Seite 312):

1)	mit einem **Objekt-Literal**	`{ Eigenschaftsname : Eigenschaftswert }`	Seite 312
2)	mit dem Objekt-Konstruktor **Object**	`new Object()`	Seite 313
3)	mit anderen Konstruktoren	`new Date()`	Seite 314

Objekt Enumeration

Für die Enumeration aller Eigenschaften (ohne Eigenschaften vom Datentyp `symbol`) eines Objektes gibt es 2 Möglichkeiten (Seite 332):

1. die `for in` Schleife
2. `Object.keys`

Objekt und Array im Vergleich

Es gibt zwei Unterschiede zwischen einem Objekt und einem Array:

1. Der Zugriff-Index bei einem Array ist vom Datentyp `Number`
   Der Zugriff-Index (Key) bei einem Objekt ist ein `String` oder ein `Symbol`.

2. Arrays sind **Ordered**. Das Feldelement mit dem Index 0, kommt immer vor dem Feldelement mit dem Index 1
   Also `foo[0]` immer vor `foo[1]`
   Objekt-Eigenschaften sind **Non-Ordered**. `obj.a` muss nicht vor `obj.b` kommen

Objekt-Erzeugung

Ein Objekt kann auf 3 Arten erzeugt werden:

1) mit einem **Objekt-Literal** { *Eigenschaftsname* : *Eigenschaftswert* }

2) mit dem Objekt-Konstruktor **Object** new Object()

3) mit einem anderen, bereits bestehenden Konstruktur (zB das built-in Objekt Date)

Zum Vergleich: Arrays werden mit einem **Array-Literal** (Seite 342) erzeugt, und Regular Expressions mit einem **Regular-Expression-Literal**.

Methode 1 - Objekt Literal

Ein **Objekt-Literal** besteht aus dem Property Name, einem Doppelpunkt und dem Property Value:

{ *Eigenschaftsname* : *Eigenschaftswert* }

Das **Objekt-Literal** ist eingeschlossen in geschweiften Klammern.
Ist mehr als ein Literal vorhanden, müssen diese durch ein Komma getrennt werden (Kommaseparierte Liste).

Beginnt der Eigenschaftsname mit einer Ziffer, oder enthält Leer- und Sonderzeichen, muss der Name in Anführungszeichen stehen. Siehe **Identifier** auf Seite 284.

Beispiel1:

Als Beispiel das Objekt `maus`, mit den Eigenschaften `fell`, `id` und `magKaese`:

```
var maus = {fell: "braun", id: 32, "magKaese": true}
```

Um die Lesbarkeit zu erhöhen, mit Leerzeilen getrennt:

```
var maus = {
      fell: "braun",
      id: 32,
      "magKaese": true
   }
```

Beispiel2: leeres Objekt

Als Beispiel wird das Objekt `katze` ohne Eigenschaften erzeugt:

```
var katze = {}
```

Nachträglich eine Eigenschaften hinzufügen:

```
katze.magMaeuse=true;                  // Dot Notation
katze["magKaese"]=false;               // Member Access Operator
```

Beispiel3: Eigenschaftsnamen mit Sonderzeichen (in einfachen und doppelten Anführungszeichen)

```
var maus = {
 fell: "braun",
 id: 32,
 "mag Käse": true,                  // Name enthält Leerzeichen und Umlaut
 'mag Katzen': false,               // einfaches Anführungszeichen
 '!@#$%^&*': true,
 '08/15 Maus': true                 // Name beginnt mit einer Ziffer
 }
```

Beispiel4: Eigenschaftsname entspricht einem Keyword (Seite 429)

```
var maus = {
 'break': false,                // Keyword in einfachen Anführungszeichen
 "continue": true               // Keyword in doppelten Anführungszeichen
 }
```

Beispiel5: Eigenschaftsattribute `enumerable`, `writable`, `configurable` (Seite 429)

```
var maus = {saeugetier: 'ja'}

Object.defineProperty( maus, 'tierart', {
    value: 'Nager',
    enumerable: false,
    writable: false,
    configurable: true
 });

var info=Object.getOwnPropertyDescriptor( maus, 'tierart' );

console.log (info)
console.log (maus.tierart)
```

Ergibt als Output:

```
Object {value: "Nager", writable: false, enumerable: false, configurable: true}
Nager
```

Methode 2 - Object()

Mit dem Keyword `new` und dem Built-in Objekt-Konstruktor `Object()` das Objekt `katze` erzeugen:

```
var katze = new Object ()
```

`katze` ist eine Instanz vom Konstruktor `Object()`

Sind keine Argumente vorhanden, können die Klammern entfallen:

```
var katze = new Object
```

Bei Objekt `katze` die Eigenschaft `magMaeuse` auf `true` setzen (und Eigenschaft `magKaese` auf `false`):

```
var katze = new Object ();
katze.magMaeuse=true;               // Dot Notation
katze["magKaese"]=false;            // Member Access Operator

var foo='magMilch'                  // String-Variable foo den Wert magMilch zuweisen
katze[foo]=true;                    // Objekt-Eigenschaft magMilch auf true setzen


console.log(katze.magMaeuse)
console.log(katze.magKaese)

console.log(katze.magMilch)   // Objekt-Eigenschaft magMilch direkt auslesen
console.log(katze[foo])       // Objekt-Eigenschaft magMilch indirekt über Variable foo auslesen
```

Ergibt als Output:

```
true
false
```

Methode 3 – Konstruktor

Der Konstruktor ist ein Funktion-Objekt. Beim Erstellen eines neues Objektes wird mit dem Keyword `new` eine Kopie vom Konstruktur erzeugt. Diese Kopie wird Instanz genannt.

Diese Vorgehensweise wird **Prototyp-basierende** Vererbung genannt: das bereits bestehende Objekt ist der Prototyp (siehe OOP Seite Fehler! Textmarke nicht definiert.). Das neu kopierte Objekt ist der Nachbau vom Prototyp.

```
var katze = new Objektname ();
```

Beispiel1: der Konstruktor ist das Built-in Funktion-Objekt `Date()`

```
var zeit = new Date()       // das Objekt zeit ist eine Instanz vom Konstruktur Date()

console.log (typeof zeit)             // Variable zeit hat den Datentyp object

console.log (zeit instanceof Date)    // Operator instanceof siehe Seite 375
```

Ergibt als Output:

```
object
true
```

Beispiel2: das Built-in Objekt `Boolean` (siehe Seite 334)

Der Konstruktur `Boolean` erzeugt ein Boolean-Objekt. Der Übergabewert ist der Value (`true` oder `false`) vom neuen Objekt.

Die Methode `valueOf()` gibt den Inhalt vom neuen Objekt als `boolean` zurück.

Beispiel:

```
var foo = new Boolean(true);            // Erzeuge mit der Konstruktur-Funktion Boolean ein Objekt namens foo
                                        // foo ist eine Instanz von Boolean()

console.log (foo)

console.log (typeof foo)                    // foo hat den Datentyp object

console.log (foo instanceof Boolean)        // true, da foo eine Instanz vom Objekt Boolean ist

console.log (foo.valueOf())                 // der Inhalt des Objekts ist  true und vom Datentyp Boolean
```

Ergibt als Output:

```
Boolean {[[PrimitiveValue]]: true}
object
true
true
```

Beispiel3: der Konstruktur ist eine eigene Funktion namens `Hund`

```
function Hund() {
this.rasse = 'Dackel';
}

var Hund2 = new Hund();                 // erzeuge das Objekt Hund2 anhand vom Konstruktur Hund
                                        // Hund2 ist eine Instanz von Hund

console.log (Hund2.rasse)

console.log (Hund2 instanceof Hund)         // Operator instanceof siehe Seite 375
```

Ergibt als Output:

```
Dackel
true
```

Methode 4 – Object.create()

Die Methode Object.create() erzeugt ein neues Objekt anhand eines Prototypen und bestimmten Properties.

Syntax: Object.create(proto[, propertiesObject])

Property (Objekt-Eigenschaft)

Die Inhalte eines Objektes werden Properties (oder Members) genannt. Inbuild-Eigenschaften siehe Seite 338.
Ein **Property** besteht aus **2 Teilen**:

1) einem **Namen (Key)** — Der Key ist entweder vom Datentyp `string` oder `symbol`
`symbol`-Keys werden bei `for…in` Schleifen nicht berücksichtigt.
2) einem **Wert** — Beliebiger Datentyp (auch ein Objekt).

Zugriff auf eine Objekt-Eigenschaft

Es gibt 2 Arten, um auf eine Eigenschaft zuzugreifen:

1) **Dot Notation** (Punkt) — `katze.magMaeuse=true;`
2) mit dem **Member Access Operator** [ ] — `katze["magMaeuse"]=true;`

Für gemischte Verwendung siehe Seite 316 (Objekt als Eigenschaft): `tiere.tierheim["ort"]`
Verwendung bei Methoden siehe Seite 317

Dot Notation

Bei der Punkt-Notation wird der Objektname und der Eigenschaftsname durch einen Punkt getrennt.
Beispiel:

```
var foo = {katze: 'Pussycat'}
foo.katze="Pussy"

console.log(typeof foo)
console.log(typeof foo.katze)          // Datentyp string
console.log(foo.katze)
```

Ergibt als Output:

```
object
string
Pussy
```

Member Access Operator

Bei der **Square Bracket Notation** steht der Identifier in Anführungszeichen und eckigen Klammern (**Member Access Operator**). Syntax:

```
Objektname.["Eigenschaftsname"]
```

Beispiel:

```
var foo = {katze: 'Pussycat'}
foo["katze"]="Pussy";

console.log(typeof foo)                  // object
console.log(typeof foo.katze)            // string
console.log(foo.katze)                   // Pussy
```

Ergibt als Output:

```
object
string
Pussy
```

Bei einem Zugriff auf die Eigenschaft über eine String-Variable, kann nur der **Member Access Operator** verwendet werden (ohne Anführungszeichen):

```
var katze = new Object ();

var foo='magMilch'                          // der Variable foo den Wert magMilch zuweisen

katze[foo]=true;                            // der Objekt-Eigenschaft magMilch den Wert true zuweisen

console.log(katze.magMilch)                 // direkter Zugriff auf die Eigenschaft magMilch
console.log ('mag Milch: ' + katze[foo]);   // Zugriff über die String-Variable foo
```

Ist der Key vom Datentyp `symbol`, fallen die Anführungszeichen weg:

```
var foo = {};

var HUND = Symbol();

foo[HUND] = "Bello";

console.log (foo[HUND])
```

Property ist ein Objekt

Der Wert einer Eigenschaft kann auch vom Datentyp `object` sein.

Beispiel: die Eigenschaft `tierheim` ist vom Datentyp `object`

```
var tiere = {
    katze: "Pussy",
    tierheim: {name: "Arche Noah", ort: "Honigbergen"},
    alter: 2
    }

console.log (tiere.katze)
console.log (tiere.alter)
console.log (tiere.tierheim.name)               // Dot Notation
console.log (tiere.tierheim["ort"])             // Square Bracket Notation
```

Ergibt als Output:

```
Pussy
2
arche noah
Honigbergen
```

Property ist eine Funktion

Eine Eigenschaft kann auch eine Funktion sein. Bei Objekten werden Funktionen auch Methoden genannt.

Siehe Funktionen Seite 393.

Beispiel1: die Eigenschaft `sprich` als Funktion

```
var tiere = {
    katze: "Pussy",
    tierheim: {name: "arche noah", ort: "Zuckerberg"},
    alter: 2,
    sprich: function () {return "miau";}
    }
console.log (tiere.sprich())
```

Ergibt als Output:

```
miau
```

Beispiel2: die Eigenschaft `sprich` als Funktion mit Übergabewert

```
var tiere = {
    katze: "Pussy",
    tierheim: {name: "arche noah", ort: "Zuckerberg"},
    alter: 2,
    sprich: function (sag) {return sag;}
    }
console.log (tiere.sprich('miau'))
```

Ergibt als Output:

```
miau
```

Beispiel3: Funktionsaufruf in der **Square Bracket Notation**

```
var hund = {
    rasse: 'Dackel',
    sprich: function () {return 'Kläff'    }
 }
console.log (hund['sprich']() )
```

Ergibt als Output:

```
Kläff
```

Beispiel4: Funktionsaufruf (Square Bracket Notation) mit Übergabewert

```
var hund = {
    rasse: 'Dackel',
    sprich: function (sag) {return sag     }
 }
console.log (hund['sprich']('Wuff') )
```

Ergibt als Output:

```
Wuff
```

delete (Eigenschaft löschen)

Mit dem Keyword `delete` kann eine Eigenschaft gelöscht werden.

Syntax: `delete Objektname.Eigenschaftsname`

Beispiel:

```
var hund = {
    rasse: 'Dackel',
 }
console.log (hund['rasse'])

delete hund['rasse']

console.log (hund['rasse'])
```

Ergibt als Output:

```
Dackel
undefined
```

Property Descriptors

`Object.defineProperty()` kann eine neue Eigenschaft hinzufügen und bestehende Eigenschaften verändern. Rückgabewert ist das betreffende Objekt.

Syntax: `Object.defineProperty (Objektname, Eigenschaftsname, Descriptor)`

Eigenschaften, die mit defineProperty() erstellt worden sind, sind per default immutable (unchangeable).

Der **Property Descriptor** ist ein Objekt, das eine einzelne Eigenschaft um zusätzliche Attribute (Beschreibungen) erweitert.
Der Descriptor ist entweder ein **Date Descriptor** oder ein **Accessor Descriptor**:

1) **Data Descriptors** ein **Data Descriptor** enthält:

`configurable`	Boolean-Flag (true/false) Bei `true` lässt sich der Descriptor ändern und die Eigenschaft löschen. Bei defineProperty ist der default `false`
`enumerable`	Boolean-Flag (true/false) Bei `true` erscheint die Eigenschaft bei einer Enumeration (Seite 332)
`value`	Beliebiger Datentyp. Default ist `undefined`
`writable`	Boolean-Flag (true/false) Bei `true` lässt sich der der Value über den Zuweisungsoperator ändern Bei defineProperty ist der default `false`

2) **Accessor Descriptors** zusätzlich zu configurable und enumerable ein **getter-setter Pair of Functions**

`configurable`	Boolean-Flag (true/false) Bei `true` lässt sich der Descriptor ändern und die Eigenschaft löschen. Bei defineProperty ist der default `false`
`enumerable`	Boolean-Flag (true/false) Bei `true` erscheint die Eigenschaft bei einer Enumeration (Seite 332)
`get`	Die Funktion, die als getter der Eigenschaft dient. Der Rückgabewert der Funktion ist der Wert der Eigenschaft (default ist undefined)
`set`	Die Funktion, die als setter der Eigenschaft dient Die Funktion hat genau einen Parameter. Das Argument wird zum Wert der Eigenschaft.

Beispiel 1: den Wert einer Eigenschaft auf read-only setzen (Wert lässt sich mich Zuweisungsoperator nicht ändern)

```
var maus = {}

Object.defineProperty( maus, 'tierart',                 // Data Descriptor
    {
       value: 'Nager',
       enumerable: false,
       writable: false,                                 // per Zuweisungsoperator nicht änderbar
       configurable: true
    }
 );

var info=Object.getOwnPropertyDescriptor( maus, 'tierart' );

console.log (info)
console.log (maus.tierart)
maus.tierart='Fisch'                                    // Cannot assign to read only property 'tierart' of object
```

Ergibt als Output:

```
Object {value: "Nager", writable: false, enumerable: false, configurable: true}
Nager
Uncaught TypeError: Cannot assign to read only property 'tierart' of object
```

Beispiel 2:

Eigenschaft `gewicht` ist in kg. *Setter* `pfund` wandelt in kg um, und schreibt den Wert in die Eigenschaft `gewicht`.
Getter `pfund` gibt als Rückgabewert das Gewicht in Pfund zurück.
Eigenschaft `groesse` ist in cm. *Setter* `inch` wandelt in cm um, und schreibt den Wert in die Eigenschaft `groesse`
Getter `inch` gibt als Rückgabewert die Größe in Zoll zurück.

```
class Person {
  constructor(vorname,  groesse, gewicht) {
    this.vorname = vorname;
    this.groesse = groesse;
    this.gewicht = gewicht;
  }

 get inch() {return this.groesse/2.54}
 set inch(wert) {this.groesse=wert*2.54}

 get pfund() {return this.gewicht*2.20462262185 }
 set pfund(wert) {this.gewicht=wert/2.20462262185}
}

var person = new Person('Carol', 180, 74);

console.log(person.inch + ' zoll');                     //70.86614173228347 zoll
console.log(person.groesse + ' cm');                    //180 cm

 console.log(person.pfund + ' pfund');                  //163.1420740169 pfund
 console.log(person.gewicht + ' kg');                   //74 kg

person.pfund = 155.55;                                  // Eigenschaft pfund setzen
console.log('setze pfund auf 155.55');                  // gewicht in kg wird dank setter automatisch gesetzt

 console.log(person.pfund + ' pfund');                  //55.55 pfund
 console.log(person.gewicht + ' kg');                   //70.55629315346083 kg

person.gewicht = 72.54;                                 // Eigenschaft gewicht setzen
console.log('setze gewicht auf 72.54');                 // pfund wird dank getter automatisch errechnet

 console.log(person.pfund + ' pfund');                  //159.923324988999 pfund
 console.log(person.gewicht + ' kg');                   //  72.54 kg
```

this

Das Keyword `this` ist ein **Object Binding** (Referenz/Bindung auf ein bestimmtes Objekt).

Fünf Binding-Arten (1 = höchste Priorität):

1)	New **Binding**	`this` im Konstruktur bezieht sich auf das neu erzeugte Objekt	Seite 328
2)	**Hard Binding**	entweder mit einer Wrapper-Funktion oder mit bind()	Seite 327
3)	**Explicit Binding**	der Funktion wird mit `call()` und `apply()` das gewünschte Objekt übergeben	Seite 325
4)	**Implicit Binding**	die Funktion ist eine Methode eines Objektes (*call with a context*)	Seite 323
5)	**Default Binding**	funktioniert nur ohne "use strict": `this` sucht auch im äußeren Scope	Seite 321

Default Binding ohne "use strict"

Wenn das Objekt nicht explizit (**Explicit Binding**) oder implizit (**Implicit Binding**) angegeben worden ist, sucht `this` das passende Objekt zuerst im **Function Scope**. Wird es nicht fündig, dann in den äußeren Scopes. Dies wird **Default Binding** genannt.

Das Suchen nach einem passenden Objekt findet nur statt, wenn `"use strict"` nicht angegeben worden ist.

Beispiel1: `this` nimmt das Objekt im **Function Scope**

```
// "use strict"

function ichBin ()
 {
    rasse='Pudel'
    console.log ('Ich bin ein ' + this.rasse)          // bei "use strict" gibt es ein:
                                                       // Error: rasse is not defined
 }

rasse='Dackel'

ichBin()
```

Ergibt als Output:

```
Ich bin ein Pudel
```

Beispiel2: `this` findet das Objekt nicht im Function Scope, und nimmt das Objekt im Global Scope

```
// "use strict"

function ichBin ()
 {
    console.log ('Ich bin ein ' + this.rasse)          // bei "use strict" gibt es ein:
                                                       // Error: rasse is not defined
 }

rasse='Dackel'

ichBin()
```

Ergibt als Output:

```
Ich bin ein Dackel
```

Implicitly lost

Wenn einer Variable eine einzelne Methode zugewiesen wird (also ohne Objekt), entfällt die implizite Bindung. Ohne `"use strict"` sucht `this` dann auch in den äußeren Scopes.

Beispiel:

```
// "use strict"

function sprich()
 {
    console.log( this.wert )
 }

var meinObjekt =
 {
    wert: 666,
    meineMethode: sprich
 }

var meineFunktion = meinObjekt.meineMethode              // nur die Methode wird zugewiesen

var wert = "Global Scope";

meineFunktion()
```

Ergibt als Output:

```
Global Scope
```

Mit **`"use strict"`** ist der Output:

```
Cannot read property 'wert' of undefined
```

Implicit Binding

Bei einem implizitem Binding, wird einer Objekt-Eigenschaft eine Funktion zugewiesen.

Ist die Funktion extern, erhält die Eigenschaft die Referenz auf die externe Funktion.

Beispiel1: der Objekt-Eigenschaft `sprich` wird die externe Funktion `bell` zugewiesen

```
function bell()
 {
    console.log( this.name +': wuff' );        // Die Funktion bell wird für beide Objekte verwendet
 }


var hund =
 {
    name: 'Bello',
    sprich: bell                               // die Funktion sprich verweist auf die Funktion bell
 }


var huendin =
 {
    name: 'Alice',
    sprich: bell
 }

hund.sprich()                                  // Bello: wuff

huendin.sprich()                               // Alice: wuff
```

Beispiel2: die Objekt-Eigenschaft `sprich` ist eine interne Funktion

```
var hund =
 {
    name: 'Bello',
    rasse: 'Dackel',

    sprich: function ()
       {
          return 'ich bin ' +
          this.name +
          ', ein ' +
          hund.rasse
       }
 }

console.log ( hund.sprich() )
```

Ergibt als Output:

```
ich bin Bello, ein Dackel
```

Object Property Reference Chain

Ist das Objekt eine Eigenschaft eines äußeren Objektes, sucht `this` nur nach Keynamen im inneren Objekt.

Beispiel 1a: `this` nimmt die Variable `wert` vom Objekt `zweites`

```
function meineFunktion()
 {
    console.log( this.wert );
 }


var zweites =
 {
    wert: 99,                                  // Variable wert vom Objekt zweites wird verwendet
    Funktion: meineFunktion
 }


var erstes =
 {
    wert: 66,
    zweites: zweites
 }

erstes.zweites.Funktion();                  //99
```

Beispiel 1b: Objekt `zweites` besitzt keine Eigenschaft namens `wert`

```
function meineFunktion()  {
    console.log( this.wert );
 }

var zweites = {
    Funktion: meineFunktion                //  Objekt zweites besitzt keine Eigenschaft namens wert
 }

var erstes = {
    wert: 66,
    zweites: zweites
 }


erstes.zweites.Funktion();                      // undefined
```

Ergibt als Output:

```
undefined
```

Explicit Binding

Jede Funktion hat 3 built-in Methoden, um ein bestimmtes Objekt für das Binding zu übergeben:

`call()`	ein Objekt, und eine beliebige Anzahl von Argumenten (Argumented List)	Seite 411
`apply()`	ein Objekt, und ein Array (das Array enthält die Funktionsargumente)	Seite 412
`bind()`	ein Objekt wird fix mit der Funktion verbunden (auch call() und apply() sicher)	Seite 414

Beispiel 1: die Methode `call` (siehe Seite 411) übergibt die angegebene Objekt-Referenz an die Funktion `ichBin`

```
function ichBin ()
 {
    console.log ('ich bin: ' + this.name)     // this bezieht sich auf carol
 }

var carol = {name: "Carol"}

ichBin.call(carol)                    //die Methode call übergibt das Objekt carol an die Funktion
```

Ergibt als Output:

```
 ich bin: Carol
```

Beispiel2: das Objekt wird mit `call` an `alter()` und dann von dort an `erhoehen()` übergeben

```
function erhoehen (aenderungswert)
 {
    return this.alter + aenderungswert;          //liest die Eigenschaft alter
                                                 //von der von call übergebenen Objekt-Referenz
 }

function alter(wert)
 {
    var aenderungswert=wert;

    console.log ("Alter von "+this.name +": " + this.alter)
    console.log ("erhöhen um: " + wert)

    this.alter = erhoehen.call( this,  aenderungswert );       //call übergibt Ojekt-Referenz

    console.log ("Neues Alter: " + this.alter)
 }

var carol = {alter: 22, name: 'Carol'}  //

var mike = {alter: 29, name: 'Mike'}

alter.call(carol,2)

alter.call(mike,-3)                      //die Methode call übergibt die Objekt-Referenz an alter()
```

Ergibt als Output:

```
    Alter von Carol: 22
    erhöhen um: 2
    Neues Alter: 24

    Alter von Mike: 29
    erhöhen um: -3
    Neues Alter: 26
```

null als Übergabe erzwingt Default-Binding

Wird bei call statt einem Objekt der Datentyp null übergeben, greift das Default-Binding.

Bei "use strict" wird in diesem Fall ein Error erzeugt.

```
// "use strict"

function ichBin ()
 {
    console.log ('ich bin: ' + this.name)
 }

var carol = {name: "Carol"}

name="Alice"                        // Default Binding Wert

ichBin.call(null)                   // ich bin Alic
```

Ergibt als Output:

```
 Ich bin ein Dackel
```

Bei "use strict" ergibt null als Übergabewert einen Error:

```
Uncaught TypeError: Cannot read property 'name' of null
```

Explizit vor Implizit

Das **Explicit Binding** hat eine höhere Priorität als das **Implicit Binding**.

Beispiel: das explizite call() Objekt hund2 zieht vor dem implizitem Objekt hund1

```
function ichBin()
    {
       console.log('Ich bin ein ' + this.rasse );
    }

var hund1 =
    {
       rasse: 'Dackel',
       sprich: ichBin
    }

var hund2 =
    {
       rasse: 'Schäfer',
       sprich: ichBin
    }

hund1.sprich()                          // Dackel

hund2.sprich()                          // Schäfer

hund1.sprich.call( hund2 )              // Schäfer          Objekt hund2 vor Objekt hund1
```

Ergibt als Output:

```
 Ich bin ein Dackel
 Ich bin ein Schäfer
 Ich bin ein Schäfer
```

Hard Binding

Zwei Möglichkeiten:

1) eine Wrapper-Funktion ruft die eigentliche Funktion auf (immer mit einem bestimmten Objekt)
2) die Methode `bind()`

Beispiel: `meinWrapper` ist die Wrapper-Funktion. Diese ruft die eigentliche Funktion `meineFunktion` auf.

```
function meineFunktion(uebergabe)
    {
       console.log( this.wert, uebergabe );
       return this.wert + uebergabe;
    }

var meinObjekt =    {  wert: 2 }

var meinWrapper = function()                                    // die Wrapper-Funktion
   {
    return meineFunktion.apply( meinObjekt, arguments );        // die eigentliche Funktion
   }

var ergebnis = meinWrapper( 3 );
console.log( ergebnis );

var meinObjekt2 = {wert: 6 }

var ergebnis = meinWrapper.call(meinObjekt2, 3 );       // meinObjekt2 wird vom Wrapper ignoriert
console.log( ergebnis );
```

bind()

Jede Funktion besitzt die Methode `bind()`.

Der Übergabewert ist das zu bindende Objekt. Der Rückgabewert ist die fix verbundende Funktion.

`bind()` wird bei der Zuweisung einer Variable (neue Funktion) verwendet. Die neue Funktion ist an das Objekt gebunden. Dieses festgelegte Objekt kann auch mit `call()` oder `apply()` nicht überschrieben werden.

Als Konstruktor kann eine **Hard Binded Methode** trotzdem ein neues Objekt, mit neuen Werten anlegen (siehe Seite 330).

Beispiel: das Objekt

```
const bello = { rasse: 'Dackel', fell: 'braun' };                  // Objekt bello

const bella = { rasse: 'Pudel', fell: 'weiss' };                   // Objekt bella

const wuff = { rasse: 'Schäfer', fell: 'schwarz' };                // Objekt wuff

function ichBin (alter,geschlecht)
{
 console.log (`Ich bin ein ${this.rasse}. Fell: ${this.fell}. Alter: ${alter}, ${geschlecht}` );
 }

var ichBinGebunden=ichBin.bind(wuff);

ichBinGebunden.call(bella,3,'weiblich');                // Objekt bella wird ignoriert


ichBinGebunden.call(bello,5,'männlich');                // Objekt bello wird ignoriert

ichBinGebunden.call(wuff,7,'männlich');
```

Ergibt als Output:

```
Ich bin ein Schäfer. Fell: schwarz. Alter: 3, Geschlecht: weiblich
Ich bin ein Schäfer. Fell: schwarz. Alter: 5, Geschlecht: männlich
Ich bin ein Schäfer. Fell: schwarz. Alter: 7, Geschlecht: männlich
```

Konstruktor

Eine Funktion kann auch als Konstruktor (Bauplan) für Objekte dienen. Siehe OOP Seite Fehler! Textmarke nicht definiert..
Der erste Buchstabe des Funktionsnamen ist bei Konstruktor-Funktionen üblicherweise großgeschrieben.
Für this Seite 321.

Beispiel1: eine Konstruktur-Funktion namens Hund

```
function Hund()
 {
    this.rasse = 'Dackel';              // In einem Konstruktor bezieht sich this auf
                                        // das neu erstellte Objekt.
 }

var meinHund = new Hund ()              // ein neues Objekt (neue Instanz von Hund) namens meinHund erzeugen

console.log(meinHund.rasse)
```

Ergibt als Output:

```
Dackel
```

Beispiel2: Konstruktur-Funktion mit einem Übergabe-Parameter (Variable namens hundename)

```
"use strict"

function Hund (hundename)
 {
    this.rasse = 'Dackel';              // In einem Konstruktor bezieht sich this auf das neu erstellte Objekt.

    this.hundename=hundename            // hundename ist der Übergabeparameter.
                                        // Der Eigenschaft des neues Objektes wird der Übergabewert zugewiesen
 }

var meinHund1 = new Hund()              // Ein neues Objekt (Instanz) namens meinHund1 erzeugen

console.log(meinHund1.rasse)            // Dackel


var meinHund2 = new Hund ('Bello')              // Ein neues Objekt (Instanz) namens meinHund2 erzeugen

console.log(meinHund2.rasse)                    // Dackel

console.log(meinHund2.hundename)                // Bello
```

Ergibt als Output:

```
Mein Name ist Bello und ich bin ein Dackel.
```

Konstruktor-Methode

Der Konstruktor kann auch eine Funktion eines bestehenden Objektes sein.

Objekteigenschaften, die nicht im Konstruktor sind, werden bei der Instanzierung nicht vom alten Objekt übernommen.

Beispiel:

```
function Hund (hundename)                                   // Konstruktur Hund
   {
      this.rasse = 'Dackel';

      this.hundename=hundename
   }


var meinHund  =                                             // Objekt meinHund
   {
      meineFunktion:Hund,                                   // Konstruktor-Methode meineFunktion
      hundename:'Alice',
      alter: 3
   }

var neuerHund = new meinHund.meineFunktion ('Bello')        // Ein neues Objekt (Instanz) erzeugen

console.log(neuerHund.rasse)                                // rasse ist im Konstruktor festgelegt

console.log(neuerHund.hundename)                            // hundename ist der Übergabewert

console.log(neuerHund.alter)                                // alter ist undefined (nicht im Konstruktor)

console.log(neuerHund instanceof meinHund.meineFunktion)    // true
```

Ergibt als Output:

```
Dackel
Bello
undefined
true
```

Hard Binded Konstruktor

Ein **Hard Binded Konstruktor** kann trotzdem ein neues Objekt, mit neuen Werten erzeugt.

Für Hard Binded siehe Seite 327.

Beispiel: hund2 ist hard binded mit hund1
hund3 hat trotzdem sein eigenes Alter

```
function meineFunktion (uebergabe)                    //Konstruktor meineFunktion
   {
      this.alter = uebergabe;
   }

var hund1 = {alter:6};                                //  Objekt hund1 Alter von 6 zuweisen

console.log (hund1.alter)                             //6



var hund2 = meineFunktion.bind( hund1 );              //Funktion hund2 hat ein Hard Binding mit Objekt hund1

console.log (typeof hund2)                            //function
console.log (hund2.alter)                             //undefined


hund2( 22  );                                 //Obwohl Objekt hund2, wird dank Hard Binding das Objekt hund1 geändert

console.log (hund1.alter)                     // 22


var hund3 = new hund2( 66 );
console.log( hund1.alter );                   // 22
console.log( hund3.alter );                   // 66
```

Ergibt als Output:

```
6
function
undefined
22
22
66
```

with

Das Keyword `with` bestimmt für den jeweiligen Programmblock das default-Objekt. Falls das gewünschte Objekt im Programmblock nicht gefunden wird, wird das von `with` bestimmte Objekt probiert.

Der Übergabewert ist das gewünschte Objekt.

Syntax:

```
with (Objektname) { Anweisungen }
```

Sind die Anweisungen einzeilig, können die geschweiften Klammern entfallen.

Bei `"use strict"` ist die Verwendung nicht erlaubt.

Beispiel:

```
with (Math)
  {
     console.log(PI);                  // PI ist nicht vorhanden, darum wird Math verwendet
     console.log(sin(PI));
     console.log(Math.PI);
  }
```

Ergibt als Output:

```
3.141592653589793
1.2246467991473532e-16
3.141592653589793
```

Objekt Enumeration

Für die Enumeration aller Eigenschaften (ohne Eigenschaften vom Datentyp `symbol`) eines Objektes gibt es 2 Möglichkeiten: die `for in` Schleife und `Object.keys`

Berücksichtigt werden die Eigenschaften, die beim **Property Descriptor** die Beschreibung `enumerable` auf `true` haben.

Für **Property Descriptor** Seite 319.

for in Schleife

Keys, die vom Datentyp `symbol` sind, werden von der `for in` Schleife nicht berücksichtigt.

Beispiel:

```
const MEINSYMBOL = Symbol();

const objekt = { a: 1, b: 2, c: 3, [MEINSYMBOL]: 4 };

for(let eigenschaft in objekt)
{
 let wert=objekt[eigenschaft]

 console.log(`${eigenschaft}: ${wert}`);
}
```

Ergibt als Output (der Wert 4 fehlt, da der Key `MEINSYMBOL` vom Datentyp `symbol` ist):

```
1
2
3
```

Mit der Methode `hasOwnProperty()` werden vererbte Eigenschaften erkannt:

```
const MEINSYMBOL = Symbol();

const objekt = { a: 1, b: 2, c: 3, [MEINSYMBOL]: 4 };

for(let eigenschaft in objekt)
{
 let wert= objekt [eigenschaft]

 if(!objekt.hasOwnProperty(eigenschaft)) continue;

 console.log(`${eigenschaft}: ${wert}`);
}
```

Object.keys

`Object.keys` enthält alle **enumerable string properties** eines Objektes. Für **Property Descriptor** Seite 319.

Beispiel:

```
const MEINSYMBOL = Symbol();

const meinObj = { a: 1, b: 2, c: 3, [MEINSYMBOL]: 4 };

Object.keys(meinObj).forEach(eigenschaft => console.log(`${eigenschaft}: ${meinObj[eigenschaft]}`))
```

Ergibt als Output (der Wert 4 fehlt, da der Key `MEINSYMBOL` vom Datentyp `symbol` ist):

```
1
2
3
```

In-build Objekte

Inbuild (vordefinierte) Objekte:

- String Seite 333
- Boolean Seite 334
- Number Seite 334
- Date Seite 335
- Map
- Set

String-Objekt

Der Konstruktur `String` erzeugt ein String-Objekt. Die Methode `valueOf()` gibt den Inhalt als `string` zurück.

Beispiel:

```
var foo = new String('wuff');              // Erzeuge mit dem Konstruktur String ein Objekt namens foo

console.log (foo)

console.log (typeof foo)                   // foo hat den Datentyp object

console.log (foo instanceof String)        // true, da foo eine Instanz vom Objekt String ist

console.log (foo.valueOf())                // der Inhalt des Objekts ist wuff und vom Datentyp string
```

Ergibt als Output:

```
String {0: "w", 1: "u", 2: "f", 3: "f", length: 4, [[PrimitiveValue]]: "wuff"}
object
true
wuff
```

Nur mit der Methode `valueOf()` kann der Inhalt vom Objekt abgefragt werden:

```
var foo = new String('');                          // foo hat den Datentyp object

if (foo) {                                         // foo ist true, da vom Datentyp object
      console.log ('foo ist true')
   }

if (!foo.valueOf()) {                              // false, da Leerstring
      console.log ('valueOf ist: ' + foo.valueOf())
   }
```

Ergibt als Output:

```
foo ist true
valueOf ist:
```

Number-Objekt

Der Konstruktur `Number` erzeugt ein Number-Objekt. Die Methode `valueOf()` gibt den Inhalt als `number` zurück.
Beispiel:

```
var foo = new Number(123.45);              // Erzeuge mit dem Konstruktur Number ein Objekt namens foo

console.log (foo)

console.log (typeof foo)                   // foo hat den Datentyp object

console.log (foo instanceof Number)        // true, da foo eine Instanz vom Objekt Number ist

console.log (foo.valueOf())                // der Inhalt des Objekts ist  wuff und vom Datentyp number
```

Ergibt als Output:

```
Number {[[PrimitiveValue]]: 123.45}
object
true
123.45
```

Nur mit der Methode `valueOf()` kann der Inhalt vom Objekt abgefragt werden:

```
var foo = new Number(0);                                  // foo hat den Datentyp object

if (foo) {                                                //   foo ist true, da vom Datentyp object
      console.log ('foo ist true')
   }

if (!foo.valueOf()) {                                     // false, da 0
      console.log ('valueOf ist: ' + foo.valueOf())
   }
```

Ergibt als Output:

```
foo ist true
valueOf ist: 0
```

Boolean-Objekt

Der Konstruktur `Boolean` erzeugt ein Boolean-Objekt. Die Methode `valueOf()` gibt den Inhalt als `boolean` zurück.
Beispiel:

```
var foo = new Boolean(true);               // Erzeuge mit dem Konstruktur Boolean ein Objekt namens foo

console.log (foo)

console.log (typeof foo)                   // foo hat den Datentyp object

console.log (foo instanceof Boolean)       // true, da foo eine Instanz vom Objekt Boolean ist

console.log (foo.valueOf())                // der Inhalt des Objekts ist  true und vom Datentyp Boolean
```

Ergibt als Output:

```
Boolean {[[PrimitiveValue]]: true}
object
true
true
```

Nur mit der Methode valueOf() kann der Inhalt vom Objekt abgefragt werden:

```
var foo = new Boolean(false);                    //foo hat den Datentyp object

if (foo) {                                       //   foo ist true, da vom Datentyp object
      console.log ('foo ist true')
   }

if (!foo.valueOf()) {                            //false, da Wert false
      console.log ('valueOf ist: ' + foo.valueOf())
   }
```

Ergibt als Output:

```
foo ist true
valueOf ist: false
```

Date-Objekt

Mögliche Varianten der Übergabe-Parameter (siehe auch mozilla.org):

```
new Date();                    // default ist gegenwärtige Zeit (local time)

new DateUTC();                 // default ist gegenwärtige UTC

new Date(Timestamp);           // Millisekunden (Integer) seit 1970-01-01 00:00:00:000 UTC

new Date(String);              // String-Syntax siehe ISO8601
                               // Entspricht Date.parse() Seite 336
```

Zwischen 2 und 7 Parameter (Monat beginnt bei 0):

```
new Date(Jahr, Monat[, Tag[, Stunde[, Minute[, Sekunde[, Millisekunden] ] ] ] ]);
```

Das **Jahr** ist vierstellig. Falls zweistellig, wird ein 19 vorne hinzugefügt (aus 88 wird 1988).

```
var zeit = new Date(88,11,27);
console.log (zeit)                 // Tue Dec 27 1988 00:00:00 GMT+0100
```

Der **Monat** ist ein Integer zwischen **0** und 11. Bei einem Überlauf wird das Datum automatisch angepasst (aus zB 12 wird 0, und das Jahr wird um ein 1 erhöht):

```
var zeit = new Date(2032,15,11);        // da 15 größer 11, werden 4 Monate dazugezählt: April
console.log (zeit)                      // Mon Apr 11 2033 00:00:00 GMT+0200
```

Der **Tag** ist ein Integer zwischen 1 und 31. Bei einem Überlauf wird das Datum automatisch angepasst (aus zB 32 wird 1, und Monat/Jahr wird erhöht):

```
var zeit = new Date(88,11,631);         // da 631 größer 31, werden 600 Tage dazugezählt:
console.log (zeit)                      // Thu Aug 23 1990 00:00:00 GMT+0200
```

Auch **Stunde** (0-23), **Minute** (0-59) und **Sekunde** (0-59) sind überlauffähig:

```
var zeit = new Date(88,11,31,24,12,12);     // da 24 größer 23, wird Jahr/Monat/Tag angepasst:
console.log (zeit)                          // Sun Jan 01 1989 00:12:12 GMT+0100
```

Im Gegensatz zu Object und Array, kennt Date bei der Erzeugung von Objekten keine Literal-Syntax.
Ohne dem Keyword new wird nicht der Konstruktor, sondern die Funktion aufgerufen (die einen String zurückgibt).

Methoden

Date.parse()

Die Methode `Date.parse()` wandelt einen `string` in einen timestamp (Datentyp `number`) um.

Mögliche Stringvarianten siehe Siehe RFC2822, und ISO8601

Beispiel:

```
var zeit =  Date.parse("Sun Jan 01 1989 00:12:12 GMT+0100")
console.log (typeof zeit)                                    // number
console.log (zeit)
```

Ergibt als Output:

```
number
599613132000
```

Date.now()

Die Methode `Date.now()` gibt die aktuelle Lokalzeit als Timestamp zurück (`number`).

Beispiel:

```
var zeit =  Date.now()
console.log (typeof zeit)
console.log (zeit)                          // 599613132000

console.log (Date(zeit))
```

Ergibt als Output:

```
number
599613132000
Sun Jan 01 1989 00:12:12 GMT+0100
```

Get-Methoden

Beispiel: Donnerstag, 27. Mai 2032 19h:41m:21s 921ms (Lokalzeit: +2 UTC)

```
var now = new Date()                        // now ist eine Instanz vom Objekt Date

console.log(typeof now)                     // object
console.log(now)                            // Thu May 27 2032 19:41:49 GMT+0200

console.log(now.valueOf())                  // 1969292509921            (Timestamp)
console.log(now.getTime())                  // 1969292509921            (Timestamp)

console.log(now.toString())                 // Thu May 27 2032 19:41:49 GMT+0200
console.log(now.toDateString())             // Thu May 27 2032
console.log(now.toISOString())              // 2032-05-27T17:41:49.921Z       siehe wikipedia.org
console.log(now.toLocaleString ())          // 27.5.2032, 19:41:49
console.log(now.toLocaleTimeString())       // 19:41:49
console.log(now.toJSON())                   // 2032-05-27T17:41:49.921Z

console.log(now.getTimezoneOffset()         // -120     Zonenzonendifferenz in Minuten, um auf UTC zu
kommen

console.log(now.getDate())                  // 27       Tag des Monats
console.log(now.getFullYear())              // 2032     Jahr vierstellig
console.log(now.getYear())                  // 132      Jahr zwei- bis dreistellig     Deprecated

console.log(now.getMonth())                 // 4        Monat (0-11)
console.log(now.getDay())                   // 4        (0=Sonntag, 1=Montag, 6=Samstag)

console.log(now.getHours())                 // 19       Stunden (0-23)
console.log(now.getMinutes())               // 41       Minuten (0-59)
console.log(now.getSeconds())               // 49       Sekunden (0-59)
console.log(now.getMilliseconds())          // 921      Millisekunden (0-999)
```

```
// UTC:

console.log(now.toUTCString())              // Thu, 27 May 2032 17:41:49 GMT

console.log(now.getUTCDate())               // 27        Tag des Monats
console.log(now.getUTCFullYear())           // 2032      Jahr vierstellig

console.log(now.getUTCMonth())              // 4         Monats (0-11)
console.log(now.getUTCDay())                // 4         (0=Sonntag, 1=Montag, 6=Samstag)

console.log(now.getUTCHours())              // 17        Stunden (0-23)
console.log(now.getUTCMinutes())            // 41        Minuten (0-59)
console.log(now.getUTCSeconds())            // 49        Sekunden (0-59)
console.log(now.getUTCMilliseconds())       // 921       Millisekunden (0-999)
```

Set-Methoden

Datum und Zeit setzen:

```
var now = new Date()                        // now ist eine Instanz vom Objekt Date

now.setDate(27)                             // 27        Tag des Monats
now.setFullYear(2032)                       // 2032      Jahr vierstellig
now.setMonth(4)                             // 4         Monat (0-11)

now.setHours(19)                            // 19        Stunden (0-23)
now.setMinutes(41)                          // 41        Minuten (0-59)
now.setSeconds(49)                          // 49        Sekunden (0-59)
now.setMilliseconds(921)                    // 921       Millisekunden (0-999)

now.setTime(1969292509921)                  // 1969292509921            (Timestamp)

// UTC:

now.setUTCDate(27)                          // 27        Tag des Monats
now.setUTCFullYear(2032)                    // 2032      Jahr vierstellig
now.setUTCMonth(4)                          // 4         Monat (0-11)

now.setUTCHours(19)                         // 19        Stunden (0-23)
now.setUTCMinutes(41)                       // 41        Minuten (0-59)
now.setUTCSeconds(49)                       // 49        Sekunden (0-59)
now.setUTCMilliseconds(921)                 // 921       Millisekunden (0-999)

now.setUTCTime(1969292509921)               // 1969292509921            (Timestamp)
```

In-build-Eigenschaften

Jedes Objekt besitzt diese vordefinierten Properties (Objekt-Eigenschaften siehe Seite 316):

1	`constructor`	die Funktion, die das gegenwärtige Objekt erzeugt hat
2	`hasOwnProperty(`*`Eigenschaftsname`*`)`	
3	`isPrototypeOf(`*`Objektname`*`)`	
4	`propertyIsEnumerable(`*`Eigenschaftsname`*`)`	
5	`toLocaleString()`	
6	`toString()`	
7	`valueOf()`	

Eigenschaft constructor

Enthält den Konstruktor, der das (gegenwärtige) Objekt erzeugt hat. Das erzeugte Objekt ist eine Instanz vom Konstruktor.

Beispiel: die Funktion `Hund` ist der Konstruktur für das Objekt `Hund2`

```
function Hund() {
    this.rasse = 'Dackel';
 }

var Hund2 = new Hund();              // erzeuge das Objekt Hund2 anhand vom Konstruktur Hund
                                     // Hund2 ist eine Instanz von Hund

console.log (Hund2.constructor)
```

Ergibt als Output:

```
Hund() {
this.rasse = 'Dackel';
}
```

Eigenschaft hasOwnProperty

Enthält den Konstruktor, der das (gegenwärtige) Objekt erzeugt hat.

Beispiel: die Funktion `Hund` ist der Konstruktur für das Objekt `Hund2`

```
function Hund() {
    this.rasse = 'Dackel';
 }

var Hund2 = new Hund();              // erzeuge das Objekt Hund2 anhand vom Konstruktur Hund
                                     // Hund2 ist eine Instanz von Hund

console.log (Hund2.constructor)
```

Ergibt als Output:

```
Hund() {
this.rasse = 'Dackel';
}
```

Klasse

Ein Objekt (Seite 310) ist eine **Dynamic Collection of Properties** (eine Sammlung von Eigenschaften, die sich jederzeit ändern lassen).

Eine Objekt-Eigenschaft besteht aus dem Eigenschaftsnamen (Key Name), dem Wert (Value) und Deskriptions (writeable, enumerable, configurable, get, set)

Der Wert einer Eigenschaft (Property) kann von jedem Datentyp (Seite 298) sein.

Der Zugriff auf eine Eigenschaft (Property) erfolgt über einen **Unique Key**. Der Key ist ein Identifier (Seite 284) vom Datenstyp `string` (Seite 306) oder `symbol` (Seite 305).

Die Zugriff-Notation (Seite 316) ist entweder:

```
Objektname.Eigenschaftsname
```

oder

```
Objektname[Eigenschaftsname]
```

Begriffe

Klasse

Klassen sind Baupläne für **Objekte**. Lässt sich von einer Klasse, weitere Klassen (**Subklasse**n) ableiten, wird sie **Superklasse** genannt. Ein Objekt, das nach einem Bauplan erzeugt worden ist, wird **Instanz** genannt.

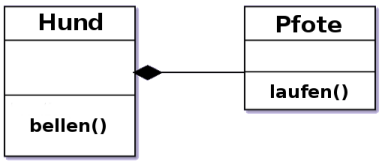
Bei der **Vererbung** (Inheritance) wird ein bestehendes Objekt kopiert (und damit ein neues Objekt erzeugt). Wird beim neuen Objekt das Verhalten einer Methode geändert (Methoden-Overriding), spricht mal von **Polymorphismus** (Polymorphism). Beim Polymorphismus haben die Methoden der vererbten Objekten zwar den gleichen Namen, aber ein unterschiedliches Verhalten.
JavaScript kennt keine Klassen. Das Keyword `class` gibt es ab ES6 und ist syntaktischer Zucker (**Syntactical Sugar**) für die Prototyp-basierte Vererbung (**prototype-based inheritance**).

Association

Unter Association versteht man das Zusammenfassen von mehreren Klassen zu einer einzigen. Man unterscheidet zwischen **Aggregation** und **Composition**.

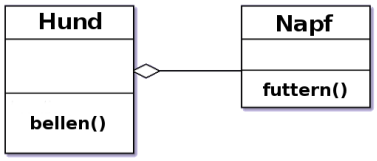
Composition

Bei einer **Composition** besitzt eine Klasse eine andere.
In diesem Beispiel besitzt die Klasse Hund (Owner) die Klasse Pfote. Die Klasse Pfote macht als einzelne Klasse keinen Sinn (**Death Relationship**).



Aggregation

Bei der Aggregation sind beide Klassen unabhängig voneinander. Die Klasse `Napf` kann auch für die Klasse `Katze` verwendet werden.



class

Eine Klasse ist ein Bauplan für ein Objekt. Zum Beispiel die Klasse Hund, die Hunde generisch (im Allgemeinen) abbildet. Ein Objekt, das auf einer Klasse (Bauplan) beruht, wird **Instanz** genannt.

Das Keyword `class` gibt es ab ES6 und ist syntaktischer Zucker (**Syntactical Sugar**) für die Prototyp-basierte Vererbung (**prototype-based inheritance**).

Die Erzeugung der Klasse Hund:

```
class Hund
    {

    }
```

new

Ein Objekt, das auf einer Klasse (Bauplan) beruht, wird **Instanz** genannt. Objekte werden mit dem Keyword `new` erzeugt.

Die Objekte (Instanzen) `Bello` und `Klaeff` (von der Klasse `Hund`):

```
class Hund   {
 }

const Bello = new Hund();
const Klaeff = new Hund();

console.log(typeof Hund)                    // function
console.log(typeof Bello)                   // object
```

Ergibt als Output:

```
function
object
object
```

instanceof

Mit dem Operator `instanceof` kann überprüft werden, ob ein Objekt von einer bestimmten Klasse abstammt:

Syntax: *Objektname* `instanceof` *Klassenname*

Beispiel: ist das Objekt `Bello` eine Instanz der Klasse `Hund`?

```
class Hund   {
 }

const Bello = new Hund();

console.log (Bello instanceof Hund)
```

Ergibt als Output:

```
true
```

Prototyp-basierte Vererbung

Der Prototyp ist ein Objekt und nicht nur ein Bauplan (im Gegensatz zu einer Klasse, siehe Seite Fehler! Textmarke nicht definiert.).

JavaScript kennt keine echten Klassen. Das Keyword `class` gibt es ab ES6 und ist jediglich **syntaktischer Zucker** (**Syntactical Sugar**) für die Prototyp-basierte Vererbung (prototype-based inheritance).

Für `Object.defineProperty` und **Property Descriptors** siehe Seite 319.

Beispiel 1a: ohne dem Keyword `class`:

```
function Person (vorname,  groesse, gewicht)
 {
    this.vorname = vorname;
    this.groesse = groesse;
    this.gewicht = gewicht;

    Object.defineProperty (this,'inch',
          {
             get: function() {return this.groesse/2.54},
             set: function(wert) {this.groesse=wert*2.54},
             enumerable: true,
             configurable: true
          }
       );

    Object.defineProperty (this,'pfund',
          {
             get: function() {return this.gewicht*2.20462262185 },
             set: function(wert) {this.gewicht=wert/2.20462262185},
             enumerable: true,
             configurable: true
          }
       );
 }
```

Beispiel 1b: mit dem Keyword `class`:

```
class Person
 {
    constructor(vorname,  groesse, gewicht) {
          this.vorname = vorname;
          this.groesse = groesse;
          this.gewicht = gewicht;
       }

    get inch() {return this.groesse/2.54}
    set inch(wert) {this.groesse=wert*2.54}

    get pfund() {return this.gewicht*2.20462262185 }
    set pfund(wert) {this.gewicht=wert/2.20462262185}
 }
```

Array

Ein Array kann mehr als einen Wert (gleichzeitig) enthalten und ist somit kein Skalar. Arrays sind eine besondere Form von Objekten. Sie besitzen eine natürliche Ordnung (**natural order**), die bei 0 beginnt und fortlaufend ist (0 kommt vor 1).

Feldgröße

Die Größe (Length) des Arrays ist nicht fix (jederzeit können Elemente hinzugefügt oder entfernt werden).

Feldwerte

Die Feldwerte eines Arrays können unterschiedliche Datentypen besitzen. Mögliche Datentypen:

```
string, number, boolean, array, objekt, function
```

Arrays sind somit **inhomogen** (nonhomogeneous).

Als Feldwert ist auch eine Funktion möglich. Siehe Beispiel5 auf Seite 344.

Feldindex

Der Feldindex ist nummerisch (**Enumerated Array**) und beginnt bei 0. **Assoziative Arrays** (auch Hash und Dictionary genannt) sind nicht möglich.

Array-Erzeugung

Ein Array kann auf **drei Arten** erzeugt werden:

Array-Literal

`[10]` — Array-Literal mit dem Feldwert 10 (Seite 343)

`[1,2,3]` — Array-Literal mit den Felderwerten `1, 2 und 3`

Wird **Homogenes Array** genannt, da alle Elemente den gleichen Datentyp haben.

`[1,'two',null]` — 1. Feldwert ist eine Zahl, der 2. Feldwert ist String und der 3. boolean.

Wird **Inhomogenes Array** genannt, da alle Elemente den gleichen Datentyp haben.

Konstruktor `Array()`

`new Array(10)` — erzeugt ein Array mit **10 Feldelementen** (Seite 344)

```
var foo = new Array(10);
console.log(foo.length);              //  10 Elemente
```

Wird keine Zahl übergeben, sondern ein oder mehrere Strings, werden aus den Strings die Feldelemente.

```
var tiere = new Array ('10')
console.log(tiere.length);            //  1 Element
```

Drei Felder:

```
var tiere = new Array ('10','11','12')
console.log(tiere.length);        //  3 Elemente
```

Methode `Array.of`

`Array.of(10,4)` — erzeugt ein Array mit **zwei Feldelementen** (Seite 344)

Aus jedem Übergabewert wird ein Feldelement:

```
var foo=Array.of(10,4,5);
console.log(foo.length);          //  3 Elemente
console.log(foo[0]);              //  10
console.log(foo[2]);              //  5
```

Array Literal

Das Array-Literal besteht nur aus dem Wert (des Feldes). Eingeschlossen in eckigen Klammern:

```
[ Feldwert ]
```

Ist mehr als ein Literal vorhanden, müssen diese durch ein Komma getrennt werden.

Beispiel1: ein Array, das nur aus Strings besteht

```
var tiere =["Mäuse", "Katzen", "Hunde"];
```

Um ein 4. Tier hinzuzufügen (Indexzähler beginnt bei 0):

```
tiere[3]="Ratten";
```

Beispiel2: ein leeres Array deklarieren

```
var tiere=[];
```

Um das 30. Tier hinzuzufügen (Indexzähler beginnt bei 0):

```
tiere[29]="Mäuse";
```

Die Objekt-Eigenschaft `length` gibt 30 Einträge zurück (höchster Indexwert + 1):

```
console.log(tiere.length);          // 30
```

Beispiel3: nur Kommas

```
var foo=[,,]
console.log(foo.length);          //  3 Elemente
console.log(foo[0]);              //  10
console.log(foo[1]);              //  5
```

Ergibt als Output:

```
3
undefined
undefined
```

Beispiel4: ein Array, dessen Feldwerte ebenfalls ein Array sind

```
var foo = [
              [1, 3, 5],
              [2+2, 2*6, 6],
              ['a', 4, true]
           ];

console.log(foo[0])               // 1. Element
console.log(foo[1])               // 2. Element
console.log(foo[2])               // 3. Element
```

Ergibt als Output:

```
[1, 3, 5]
[4, 12, 6]
["a", 4, true]
```

Beispiel5: ein Array, das aus Objekten besteht

```
var foo = [
            { name: "Katze", gewicht: 2 },
            { name: "Hund", gewicht: 6 }
          ];

console.log(foo[0])
console.log(foo[1])
```

Ergibt als Output:

```
[1, 3, 5]
[4, 12, 6]
["a", 4, true]
```

Beispiel6: eine Funktion als Feldwert

```
var foo = [];
foo[0] = function (schreib) {console.log(schreib); };

foo[0]('wuff');
```

Ergibt als Output:

```
wuff
```

Konstruktur Array()

Mit dem Keyword `new` und dem Objekt-Konstruktor `Array()` das Objekt `tiere` erzeugen (*tiere* ist eine Instanz vom Konstruktur *Object*):

```
var tiere = new Array ("Mäuse", "Katzen", "Hunde");
```

Ein Konstruktur ist ein Funktion-Objekt, um (neue) Objekte erzeugen zu können.

Methode Array.of()

Die Methode erzeugt aus den übergebenen Werten ein Array. Index beginnt bei 0 (fortlaufend).
Beispiel:

```
var foo=Array.of(4,8,10,15);
console.log(foo.length);          // 4
console.log(foo);                 // [4, 8, 10, 15]
```

Erzeugt als Output:

```
4
[4, 8, 10, 15]
```

Array Eigenschaften (Array Properties)

Drei Array-Eigenschaften:

prototype

Ermöglicht das Hinzufügen von Eigenschaften.

constructor

Gibt den Konstruktor zurück (der das Array erzeugt hat).

Beispiel:

```
var foo=[11,12,13,14]
console.log(foo.constructor);
```

Ergibt als Output:

```
Array() { [native code] }
```

length

Das Auslesen gibt die Anzahl (höchster Indexwert) der gegenwärtigen Index-Elemente zurück.

Die letzte Indexposition ist Länge-1 (da der Index bei 0 beginnt).

Das Zuweisen bestimmt die Anzahl/Größe des Arrays.

Beispiel1: Array-Größe auslesen

```
var foo = [11,12,13,14]

console.log(foo.length);                    // 4 Elemente
console.log(foo);

foo.length=2                                // Größe auf 2 Elemente setzen

console.log(foo);
```

Ergibt als Output:

```
4
[11, 12, 13, 14]
[11, 12]
```

Beispiel2:

```
var foo = [11,12]

foo.length=4                                // Größe auf 4 Elemente setzen
console.log(foo.length);                    // 4 Elemente
console.log(foo);

for (var feld in foo) console.log(feld + ' = ' + foo[feld])
```

Ergibt als Output:

```
4
[11, 12]
0 = 11
1 = 12
```

Array Methoden

	Methode	Rückgabewert	Callback Funktion	
Suchen:				
überprüfe Feldwert auf Vorhandensein	`includes()`	Boolean	nein	Seite 351
suche bestimmten Feldwert im Array(-Range)	`indexOf()`	Indexposition	nein	Seite 351
suche mit Funktion	`find()`	Feldwert	ja	Seite 350
suche mit Funktion	`findIndex()`	Indexposition	ja	Seite 350
überprüfe mit Funktion: alle true?	`every()`	Boolean	ja	Seite 348
überprüfe mit Funktion: mindestens einer true?	`some()`	Boolean	ja	Seite 358
mit Funktion suchen & Array erzeugen	`filter()`	Array	ja	Seite 349
Werte verändern:				
Array(-Range) mit einem bestimmten Wert füllen	`fill()`		nein	Seite 349
einen Range mit anderem Range überschreiben	`copyWithin()`		nein	Seite 347
Reihenfolge der Felder invertieren	`reverse()`		nein	Seite 356
Range löschen und durch neue Elemente ersetzen	`splice()`	Array (gelöschte Felder)	nein	Seite 360
Objekt oder String erzeugen				
aus Arraybereich neues Array erzeugen	`slice()`	Array	nein	Seite 357
neues Array-Iterator-Objekt (Key-Value pairs)	`entries()`	Objekt	nein	Seite 348
mit Funktion suchen & Array erzeugen	`filter()`	Array	ja	Seite 349
mit allen Feldwerten einen String erzeugen	`join()`	String	nein	Seite 352
mit allen Feldwerten einen String erzeugen	`toString()`	String	nein	Seite 361
Arrays und Elemente verschmelzen	`concat()`	Array	nein	Seite 347
Array-Iterator-Paare erzeugen (Feld/Indexwert)	`entries()`	Array-Iterator-Ojekt	nein	Seite 348
eine Funktion mit jedem Feldwert aufrufen	`map()`	Array	ja	Seite 353
Range löschen und durch neue Elemente ersetzen	`splice()`	Array (gelöschte Felder)	nein	Seite 360
Feldelement löschen/hinzufügen				
erstes Feldelement löschen	`shift()`	Feldwert	nein	Seite 356
letztes Feldelement löschen	`pop()`	Feldwert	nein	Seite 353
Feldelemente hinzufügen (am Array-Ende)	`push()`	Feldwert	nein	Seite 354
Feldelemente hinzufügen (am Array-Anfang)	`unshift()`	Number (Arraylänge)	nein	Seite 362
Range löschen und durch neue Elemente ersetzen	`splice()`	Array (gelöschte Felder)	nein	Seite 360
komplette Travesierung				
eine Funktion mit jedem Feldwert aufrufen	`forEach()`		ja	Seite 351
eine Funktion mit jedem Feldwert aufrufen	`map()`	Array	ja	Seite 353
Zwischenwert wird immer mitgeschliffen	`reduce()`	Variable	ja	Seite 354
Zwischenwert wird immer mitgeschliffen	`reduceRight()`	Variable	ja	Seite 355

concat()

mehrere Arrays und Elemente verschmelzen

Rückgabewert: neues Array

Übergabeparameter sind beliebig viele Elemente und Array. Siehe mozilla.org

Beispiel1: Verbindet zwei (oder mehrere) Arrays und erzeugt daraus ein neues Array:

```
var neuesArray = altesArray1.concat(altesArray2 [, altesObjektn])
```

Beispiel:

```
var tiere = new Array ("Mäuse", "Katzen");
var futter = new Array ("Käse", "Milch");
var alle =  tiere.concat (futter);
console.log(alle);              // ["Mäuse", "Katzen", "Käse", "Milch"]
```

Beispiel2: Fügt Feldelemente (neue Werte) zu einem bestehendem Array hinzu:

```
var neuesArray = altesArray.concat(neuerWert1 [, neuerWertn])
```

Beispiel:

```
var tiere = new Array ("Mäuse", "Katzen");

var alle = tiere.concat("Hunde","Vögel");

console.log(alle);              // ["Mäuse", "Katzen", "Hunde", "Vögel"]
```

copyWithin

bestimmen Range mit bestimmten Range überschreiben

Kopiert Feldelemente und überschreibt mit ihnen eine andere Stelle im Array (ab ES6). Siehe mozilla.org

```
copyWithin(target, start[, end])
```

target	Indexposition, aber der die Feldelemente überschrieben werden. Wenn negativ, Zählung von hinten.
start	Indexposition, ab der die Feldelemente kopiert werden. Wenn negativ, Zählung von hinten.
end	Indexposition, bis zu der die Feldelemente (exklusive dieser Position) kopiert werden sollen Ohne Angabe von end, wird soviel wie möglich kopiert und überschrieben.

Die Methode verändert das angegebene Array.
Die Methode verändert nicht die Größe des Arrays (nicht passende Elemente werden weggeworfen).
Indexzähler beginnt bei 0.

Beispiel1:

Kopiere ab dem 2. Zeichen (da die Zählung bei 0 beginnt, der Buchstabe c) bis exklusive 3. Zeichen, und überschreibe ab dem 5. Zeichen (f):

```
var foo = ['a','b','c','d','e','f','g','h'].copyWithin(5, 2, 3)
console.log(foo);     // ["a", "b", "c", "d", "e", "c", "g", "h"]
```

Beispiel2: Kopiere h auf a

```
var foo = ['a','b','c','d','e','f','g','h'].copyWithin(0, 7,8)
console.log(foo);     // ["a", "b", "c", "d", "e", "c", "g", "h"]
```

every()

Überprüfe, ob alle Elemente bei der callback-Funktion `true` zurückgeben. Beim ersten `false` wird abgebrochen

Rückgabewert: Boolean

Der Übergabeparameter ist die Funktion, mit der jeder Feldwert überprüft werden soll (siehe mozilla.org).

```
Array.every(Suchfunktion)
```

Gibt die Funktion eine Expression zurück, die `false` ergibt, wird die Überprüfung abgebrochen und die Methode gibt `false` zurück. Siehe auch Funktion `filter()` auf Seite 349 und Funktion `some()` auf Seite 358.

`every()` traversiert das Array und übergibt der Funktion jeweils den Feldwert, den Indexwert und das Array.

Beispiel:

```
var zahlen = new Array (12, 8, 7, 10, 9);

function geradeZahl(zahl, index, array) {

 var gerade = Boolean(!(zahl%2))

 console.log('index: '+index+'   wert: '+zahl+'    gerade: '+gerade)

  return gerade
}

console.log(zahlen.every(geradeZahl))
```

Ergibt als Output:

```
index: 0   wert: 12    gerade: true
index: 1   wert: 8    gerade: true
index: 2   wert: 7    gerade: false
false
```

entries()

Rückgabewert: neues Array Iterator Objekt

Die Methode erzeugt ein neues **Array-Iterator-Objekt**. In diesem gibt es für jeden Indexeintrag ein Key-Value Paar (siehe mozilla.org).

Beispiel:

```
var buchstaben = ['a', 'b', 'c'];
var paare = buchstaben.entries();

console.log(paare.next().value); // [0, 'a']
console.log(paare.next().value); // [1, 'b']
console.log(paare.next().value); // [2, 'c']
```

Ergibt als Output:

```
[0, "a"]
[1, "b"]
[2, "c"]
```

filter()

mit Funktion suchen

Rückgabewert: neues Array (mit nur bestimmten Einträgen vom alten Array)

Der Übergabeparameter ist die Funktion, mit der jeder Feldwert überprüft werden soll (siehe mozilla.org). Der Rückgabewert ist ein neues Array, das nur die Feldelemente besitzt, bei der die Funktion true zurückgab.

```
Array.filter(Suchfunktion)
```

filter() traversiert das Array und übergibt der Funktion jeweils den Feldwert, den Indexwert und das Array.

Beispiel: filtere die geraden Zahlen (das neue Array enthält keine ungeraden Zahlen mehr)

```
var zahlen = [2,3,7,12,14,17,43,66];

function geradeZahl (wert,index)
 {
    console.log("index: "+index+" wert: "+wert);
    return !(wert%2);
 }
var zahlen2=zahlen.filter(geradeZahl);

console.log(zahlen2+' (length: '+zahlen2.length+')');
```

Ergibt als Output:

```
[2, 12, 14, 66] (length: 4)
```

fill()

befüllt Range (oder ganzes Array) mit bestimmten Wert

Die Methode befüllt ein Array mit einem bestimmten Wert (ab ES6). Siehe mozilla.org
Drei Übergabeparameter (`start` und `end` sind optional):

```
Array.fill(Feldwert [, start, end] )
```

Zielarray	das Array, das mit Werten befüllt/überschrieben werden soll
start	optional: befülle von Indexposition `start` (inklusive dieser Position)
end	optional: befülle bis Indexposition `end` (exklusive dieser Position)

Beispiel1: befülle das ganze Array mit dem Wert `5`

```
var zahlen = [2,3,7,12,14];

zahlen.fill(5);
console.log(zahlen);
```

Ergibt als Output:

```
[5, 5, 5, 5, 5]
```

Beispiel2: befülle das Array (von Position `2`, bis exklusive Position `4`) mit dem Wert `5`

```
var zahlen = [2,3,7,12,14];

zahlen.fill(5,2,4);
console.log(zahlen);
```

Ergibt als Output:

```
[2, 3, 5, 5, 14]
```

find()

suche mit Funktion

Rückgabewert: Feldwert

Der Übergabeparameter ist die Suchfunktion, mit der im Array gesucht werden soll (siehe mozilla.org):

```
Array.find(Suchfunktion)
```

find() traversiert das Array und übergibt der Funktion den Feldwert.
Gibt die Suchfunktion eine Expression zurück, die `true` ergibt, wird der Suchvorgang abgebrochen und der Methode der gesuchte Feldwert übermittelt. Die Methode `findIndex()` gibt den Indexwert zurück (siehe Seite 350).

Dieses Beispiel sucht die erste Ganzzahl im Array `zahlen`:

```
var zahlen = [41, 33, 13, 40, 47, 50, 60];

function ganzzahlCheck(zahl) {

   if (zahl%2==0) {return true}

}

console.log(zahlen.find(ganzzahlCheck)) // 40 (Indexwert 3)
```

findIndex()

suche mit Funktion

Rückgabewert: Feldposition

Der Übergabeparameter ist die Suchfunktion, mit der im Array gesucht werden soll (siehe mozilla.org):

```
Array.findIndex(Suchfunktion)
```

findIndex() traversiert das Array und übergibt der Funktion den Feldwert, den Indexwert und das Array.
Gibt die Suchfunktion eine Expression zurück, die `true` ergibt, wird der Suchvorgang abgebrochen und der Methode der Indexwert übermittelt. Die Methode find() gibt den Feldwert zurück (siehe Seite 350).

Dieses Beispiel sucht die erste Ganzzahl im Array `zahlen`:

```
var zahlen = [41, 33, 13, 40, 47, 50, 60];

function ganzzahlCheck(wert,indexwert,array) {
   console.log(indexwert+': '+wert);
   return wert%2==0;
}

console.log(zahlen.findIndex(ganzzahlCheck))  // Indexwert 3 (40)
```

Ergibt als Output:

```
3
```

forEach()

Array-Travesierung

Der Übergabewert ist die Funktion, mit der jeder Feldwert aufgerufen werden soll (siehe mozilla.org).

Die Methode traversiert das gesamte Array und übergibt der Funktion den Feldwert, den Indexwert und das Array.

```
Array.findIndex(Funktion)
```

Beispiel:

```
var zahlen = [41, 33, 13, 40];

function machwas(wert,indexwert,feld) {
   console.log(indexwert+': '+wert+ ' L: '+ feld.length);
}

zahlen.forEach(machwas);
```

Ergibt als Output:

```
0: 41 L: 4
1: 33 L: 4
2: 13 L: 4
3: 40 L: 4
```

includes()

überprüfe Feldwert auf Vorhandensein

Der Übergabewert ist der Wert, nach dem im Array gesucht werden soll. Gibt es diesen Wert als Feldwert, dann gibt die Methode `true` zurück. Siehe mozilla.org.

Beispiel: Suche die Zahlen `40` und `50`:

```
var zahlen = [41, 33, 13, 40];
console.log (zahlen.includes(40) )
console.log (zahlen.includes(50) )
```

Ergibt als Output:

```
true
false
```

indexOf()

suche bestimmten Feldwert im Array(-Range)

Rückgabewert: Indexposition

Übergabewert ist der gesuchte Feldwert und optional die Startposition. Rückgabwert ist die Indexposition vom ersten Fund (-1, wenn Feldwert nicht gefunden). Siehe mozilla.org

```
Array.findOf(Feldwert [,Startposition])
```

Beispiel:

```
var zahlen = [41, 33, 13, 40, 47, 50, 60];

console.log (zahlen.indexOf(13,5));    // 6
```

Ergibt als Output:

```
6
```

join()

erzeugt aus allen Feldwerten einen String

Rückgabewert: String

Die Methode join() erzeugt aus allen Feldwerten einen String. Der optionale Parameter ist der Seperator (string).

```
Array.findOf([Seperatorstring])
```

Beispiel:

```
var zahlen = [41, 33, 13, 40, 47];
console.log (zahlen.join(' * '));
```

Ergibt als Output:

```
41 * 33 * 13 * 40 * 47
```

keys()

Rückgabewert: Array Iterator, der die Keys (Indexwerte) enthält

Beispiel:

```
var arr = ["a", "b", "c"];
var iterator = arr.keys();

console.log(iterator.next()); // { value: 0, done: false }
console.log(iterator.next()); // { value: 1, done: false }
console.log(iterator.next()); // { value: 2, done: false }
console.log(iterator.next()); // { value: undefined, done: true }
```

lastIndexOf()

suche rückwärts (rechts nach links) bestimmten Feldwert im Array(-Range)

Rückgabewert: Indexposition

Übergabewert ist der gesuchte Feldwert (und optional die Startposition). Rückgabwert ist die Indexposition vom ersten Fund (-1, wenn Feldwert nicht gefunden). Siehe mozilla.org

Der Wert wird mit **String Equality** gesucht (nicht nur der Wert, sondern auch der Datentyp muss übereinstimmen.

```
Array.findOf(Feldwert [,Startposition])
```

Beispiel:

```
var zahlen = [41, 33, 13, 40, 13, 47, 50, 60];
console.log (zahlen.lastIndexOf(13));     // 4
```

Ergibt als Output:

```
4
```

map()

Array-Travesierung mit Funktion

Rückgabewert: Array

Der Übergabewert ist die Callback-Funktion, mit der jeder Feldwert aufgerufen werden soll (siehe mozilla.org). Default für den optionalen Parameter `this` ist `window`.

```
Array.map(Funktion [,this])
```

Die Methode traversiert das gesamte Array und übergibt der Callback-Funktion den Feldwert, den Indexwert und das Array. Aus den Rückgabewerten der Funktion wird ein neues Array erstellt.

Beispiel1:

```
var zahlen = [1, 4, 9];
var ergebnis = zahlen.map(Math.sqrt);

console.log(ergebnis);
```

Ergibt als Output:

```
[1, 2, 3]
```

Beispiel2:

```
var zahlen = [1, 4, 9];
var ergebnis = zahlen.map(machwas);

function machwas (wert,indexwert,feld) {
   console.log(indexwert+': '+wert+ ' L: '+ feld.length);
   return Math.sqrt(wert);
}

console.log(ergebnis);
```

Ergibt als Output:

```
[1, 2, 3]
```

pop()

löscht letztes Array-Element

Rückgabewert: gelöschter Feldwert

Die Methode pop() entfernt das letzte Element in einem Array. Der Rückgabewert ist dieser letzte Feldwert.

Beispiel:

```
var zahlen = [1, 4];

console.log(zahlen.pop());
console.log(zahlen.pop());
console.log(zahlen.pop());
```

Ergibt als Output:

```
1
4
undefined
```

push()

Feldelemente hinzufügen

Rückgabewert: die neue Array-Größe

Der Übergabewert ist der Feldwert, der dem Array hinzugefügt wird. Der Rückgabewert ist die neue Array-Größe.

Beispiel:

```
var zahlen = [1, 4];

console.log(zahlen.push(9, 11));   // 4
console.log(zahlen);
```

Ergibt als Output:

```
4
[1, 4, 9, 11]
```

reduce()

Array-Travesierung mit Funktion: bei jedem Feldelement erhält man eine Variable zurück (Function Return). Diese Variable wird beim nächsten Feldelement der Funktion wieder übergeben.

Rückgabewert: Variable (beliebiger Datentyp)

Der Übergabewert ist die Callback-Funktion, mit der jeder Feldwert aufgerufen werden soll (siehe mozilla.org).

```
Array.reduce(Funktion [,Startvariable])
```

Die Methode traversiert das gesamte Array und übergibt der Callback-Funktion eine Variable (die zuvor als Function-Return oder beim ersten Aufruf als optionalen Parameter bekommt).

Nicht vorhandene Feldelemente werden übersprungen. Laufrichtung: von rechts nach links (vorwärts)

Beispiel1: ohne Startwert

```
var zahlen = [1,4,5];

function addieren(zwischensumme, zahl) {
    return zwischensumme + zahl;
}

var a = zahlen.reduce(addieren);
console.log (a)
```

Ergibt als Output:

```
10
```

Beispiel2: mit Startwert 10

```
var zahlen = [1,4,5];

function addieren(zwischensumme, zahl, indexposition) {
    return zwischensumme + zahl;
}

var a = zahlen.reduce(addieren,10);
console.log (a)
```

Ergibt als Output:

```
20
```

Beispiel3: mit Startwert 20

```
var zahlen = [1,4,5];

function subtrahieren (zwischenwert, feldwert, indexposition)
{

 console.log(`Zwischenwert: ${zwischenwert} / Feldwert: ${feldwert} / Index: ${indexposition}`);

 return zwischenwert - feldwert;

}

var a = zahlen.reduce(subtrahieren,20);
console.log ('Endwert: ' + a)
```

Ergibt als Output:

```
Zwischenwert: 20 / Feldwert: 1 / Index: 0
Zwischenwert: 19 / Feldwert: 4 / Index: 1
Zwischenwert: 15 / Feldwert: 5 / Index: 2
Endwert: 10
```

reduceRight()

Array-Travesierung mit Funktion: bei jedem Feldelement erhält man eine Variable zurück (Function Return). Diese Variable wird beim nächsten Feldelement der Funktion wieder übergeben.

Rückgabewert: Variable (beliebiger Datentyp)

Der Übergabewert ist die Callback-Funktion, mit der jeder Feldwert aufgerufen werden soll (siehe mozilla.org).

```
Array.reduceRight(Funktion [,Startvariable])
```

Die Methode traversiert das gesamte Array und übergibt der Callback-Funktion eine Variable (die zuvor als Function-Return oder beim ersten Aufruf als optionalen Parameter bekommt).

Nicht vorhandene Feldelemente werden übersprungen. Laufrichtung: von rechts nach links (rückwärts)

Beispiel1: ohne Startwert

```
var zahlen = [1,4,5];

function addieren(zwischensumme, zahl) {
    return zwischensumme + zahl;
}

var a = zahlen.reduceRight(addieren);
console.log (a)
```

Ergibt als Output:

```
10
```

Beispiel2: mit Startwert 10

```
var zahlen = [1,4,5];

function addieren(zwischensumme, zahl, indexposition) {
    return zwischensumme + zahl;
}

var a = zahlen.reduceRight(addieren,10);
console.log (a)
```

Ergibt als Output:

```
20
```

Beispiel3: mit Startwert 20

```
var zahlen = [1,4,5];

function subtrahieren (zwischenwert, feldwert, indexposition)
{

 console.log(`Zwischenwert: ${zwischenwert} / Feldwert: ${feldwert} / Index: ${indexposition}`);

 return zwischenwert - feldwert;

}

var a = zahlen.reduceRight(subtrahieren,20);
console.log ('Endwert: ' + a)
```

Ergibt als Output:

```
Zwischenwert: 20 / Feldwert: 5 / Index: 2
Zwischenwert: 15 / Feldwert: 4 / Index: 1
Zwischenwert: 11 / Feldwert: 1 / Index: 0
Endwert: 10
```

reverse()

Invertiert die Reihenfolge

Kein Rückgabewert. Die Methode verändert das angegebene Array.

Beispiel:

```
var foo = ['a', 'b', 'c'];
foo.reverse();

console.log(foo)
```

Ergibt als Output:

```
Array [ "c", "b", "a" ]
```

shift()

Entfernt das erste Feldelement. Kein Übergabewert.

Rückgabewert: Feldwert des ersten Elementes

Die Methode verändert das angegebene Array.

Beispiel:

```
var foo = ['a', 'b', 'c'];
var geloescht=foo.shift();

console.log('gelöschter Wert: ' + geloescht)
console.log(foo)
```

Ergibt als Output:

```
gelöschter Wert: a
Array [ "b", "c" ]
```

slice()

aus einem bestimmten Arraybereich ein neues Array erzeugen. Parameter sind Start- und Endwert

Rückgabewert: Array

Aus einem Arraybereich wird ein neues Array erzeugt. Parameter für Start- und Endposition sind optional.

Das Array wird nicht verändert.

Siehe auch mozilla.org.

Syntax:

```
Array.slice([Startposition,Endposition])
```

Der Startwert der Indexbereich kann auch negativ sein.

Der Endwert des Indexbereich ist **exklusive** diesem Feldelement (vom 1. bis zum 2. Element gibt nur das 1. Element zurück).

Beispiel1: keine Parameter: das gesamte Array wird kopiert.

```
var foo = ['a', 'b', 'c', 'd', 'e'];
var foo2= foo.slice();
console.log(foo2);
```

Ergibt als Output:

```
Array [ "a", "b", "c", "d", "e" ]
```

Beispiel2: nur der Start der Indexposition

```
var foo = ['a', 'b', 'c', 'd', 'e'];
var foo2= foo.slice(2);   // ab dem 3. Indexwert (Zählung beginnt bei 0)
console.log(foo2);
```

Ergibt als Output:

```
Array [ "c", "d", "e" ]
```

Beispiel3: negativer Startwert (Bereich ab dem -n. Element von hinten)

```
var foo = ['a', 'b', 'c', 'd', 'e'];
var foo2= foo.slice(-2);     // ab dem vorletzten Element bis zum letzten Element
console.log(foo2);
```

Ergibt als Output:

```
Array [ "d", "e" ]
```

Beispiel4: von-bis Bereich

```
var foo = ['a', 'b', 'c', 'd', 'e'];
var foo2= foo.slice(1,4);
console.log(foo2);
```

Ergibt als Output:

```
Array [ "d", "e" ]
```

some()

Überprüfe, ob mindestens 1 Element bei der callback-Funktion `true` zurückgibt. Beim ersten `true` wird abgebrochen.
Rückgabewert: Boolean

Der Übergabeparameter ist die Funktion, mit der jeder Feldwert überprüft werden soll (siehe mozilla.org).

```
Array.some(Suchfunktion)
```

Gibt die Funktion eine Expression zurück, die `true` ergibt, wird die Überprüfung abgebrochen und die Methode gibt `false` zurück. Siehe auch Funktion `filter()` auf Seite 349 und Funktion `every()` auf Seite 348.
`some()` traversiert das Array und übergibt der Funktion jeweils den Feldwert, den Indexwert und das Array.

```
var zahlen = new Array (3, 5, 12, 8, 7);

function geradeZahl(zahl, index, array) {

 var gerade = Boolean(!(zahl%2))

 console.log('index: '+index+'   wert: '+zahl+'    gerade: '+gerade)

  return gerade
}

console.log(zahlen.some(geradeZahl))
```

Ergibt als Output:

```
wert: 3    gerade: false
wert: 5    gerade: false
wert: 12    gerade: true
true
```

sort()

Wird keine eigene (Sortier-)Funktion angegeben, werden die Feldelemente als Strings sortiert .
Kein Rückgabewert. Das Array wird verändert.

Der optionale Übergabeparameter ist die Funktion, mit der das Array sortiert werden soll (siehe mozilla.org).
Ohne eigener Sortierfunktion wird jedes Zeichen nach dem Unicode Code Point sortiert. Für **Unicode** siehe Seite 234 und für **Code Point** siehe Seite 223.

```
Array.some(Sortierfunktion)
```

Beispiel 1: Buchstaben

```
var feld = new Array ('z','ac','ab','ö','o','b','B','A','3','a','ß','Ö');

feld.sort();
console.log(feld)
```

Ergibt als Output:

```
["3", "A", "B", "a", "ab", "ac", "b", "o", "z", "Ö", "ß", "ö"]
```

Zuerst die Zahlen, dann Großbuchstaben, Kleinbuchstaben, große Umlaute, ß und kleine Umlaute

Beispiel 2: Zahlen

```
var feld = new Array (5, 12, 222, 22, 10, 9, 2, 100);

feld.sort();
console.log(feld)
```

Ergibt als Output (Zahlen als Strings sortiert):

```
[10, 100, 12, 2, 22, 222, 5, 9]
```

Comparison Function (eigene Vergleichsfunktion)

Die Methode sort() kann auch mit einer selbstgeschriebenen Vergleichsfunktion sortieren.

Die Vergleichsfunktion muss genau 2 Parameter besitzen.
Der Rückgabewert ist nach diesen Regeln definiert:

-1	negative Zahl, wenn Argument1 vor Argument2 kommen soll
0	Null, wenn beide Argumente gleichwertig sind
1	positive Zahl, wenn Argument1 nach Argument2 kommen soll

Beispiel 1a: Zahlenstrings ohne zusätzlicher Sortierfunktion

```
var feld = new Array ('20','12','2','111');
feld.sort();
console.log(feld)
```

Ergibt als Output (Zahlen als Strings sortiert):

```
["111", "12", "2", "20"]
```

Beispiel 1b: Zahlenstrings mit zusätzlicher Sortierfunktion (aufwärts sortieren)

```
function sortieren (a,b)  {

    if ( parseFloat(a) == parseFloat(b) ) return 0          // 0, wenn gleichwertig
       else if ( parseFloat(a) < parseFloat(b) ) return -1  // -1 wenn a vor b kommen soll
       else if ( parseFloat(a) > parseFloat(b) ) return 1   // 1, wenn b vor a kommen soll
 }

 var feld = new Array ('20','12','2','111');

feld.sort(sortieren);

console.log(feld)
```

Ergibt als Output (Zahlen als Strings sortiert):

```
["0", "0", "2", "12", "12", "20", "20", "111"]
```

Beispiel 2: zuerst die geraden Zahlen, dann die ungeraden Zahlen

```
function sortieren (a,b)
 {
    let gerade_a = !(a%2)
    let gerade_b = !(b%2)

    if ( gerade_a == gerade_b ) {  return 0 }                    // 0, wenn gleichwertig
       else if ( gerade_a && !gerade_b ) { return -1 }           // -1, wenn a vor b kommen soll
       else if ( !gerade_a && gerade_b ) { return 1 }            // 1, wenn b vor a kommen soll
 }

var feld = new Array (12, 7, -11, 0, 9, 0, 33, 8, 12, -33, 60, 10, 5, 7, 100, 11, 10);

feld.sort(sortieren);

console.log(feld)
```

Ergibt als Output:

```
[12, 12, 100, 0, 10, 0, 60, 8, 10, 7, -33, 33, 9, 5, 7, -11, 11]
```

splice()

Splice 1) löscht einen Bereich (ab *Startposition* lösche *Loeschanzahl* Elemente)
und 2) fügt an dieser Stelle neue Elemente hinzu (optional)
Der Rückgabewert ist ein Array mit den gelöschten Elementen. Das Array wird verändert.
Siehe mozilla.org

```
Array.splice([Startposition [,Loeschanzahl, NeuesFeld1, NeuesFeldN] )
```

Wird kein einziges Argument übergeben, wird das Array nicht verändert.

Wird nur die Startposition angegeben (ohne Loeschanzahl), werden alle Elemente ab dieser Position gelöscht.
Ist die Startposition negativ, wird von hinten die Startposition bestimmt (-2 löscht die letzten zwei Elemente).

Beispiel1: lösche alle Elemente ab der 2. Position (Zählung beginnt bei 0)

```
var foo = new Array('a', 'b', 'c', 'd', 'e');

foo.splice(2)

console.log(foo)
```

Ergibt als Output:

```
["a", "b"]
```

Beispiel2: lösche alle Elemente

```
var foo = new Array('a', 'b', 'c', 'd', 'e');

foo.splice(0)

console.log(foo)
```

Ergibt als Output:

```
[]
```

Beispiel3: lösche die letzten 2 Elemente

```
var foo = new Array('a', 'b', 'c', 'd', 'e');

foo.splice(-2)

console.log(foo)
```

Ergibt als Output:

```
["a", "b", "c"]
```

Beispiel4: lösche das 3. Element (Zählung beginnt bei 0)

```
var foo = new Array('a', 'b', 'c', 'd', 'e');

foo.splice(2,1)

console.log(foo)
```

Ergibt als Output:

```
["a", "b", "d", "e"]
```

Beispiel5: lösche das 2. Element und füge 2 Elemente an dieser Stelle ein ('x', 'y')

```
var foo = new Array('a', 'b', 'c', 'd', 'e');

foo.splice(1,1,['x','y')

console.log(foo)
```

Ergibt als Output:

```
["a", "x", "y", "c", "d", "e"]
```

toLocaleString()

Erzeugt wie toString() aus dem Array einen String (nach local definiertem Schema). Siehe mozilla.org

toString()

Die Datentypen `object`, `number`, `string`, `boolean` besitzen die Methode `toString()`. Siehe Seite 307.

Siehe mozilla.org

Beispiel:

```
var foo=Array.of(10,4);

var fooString=foo.toString();

console.log(fooString);

console.log('type: ' + typeof fooString);
```

Ergibt als Output:

```
10,4
type: string
```

unshift()

Fügt vorne neue Elemente hinzu (am Anfang des Arrays)

Der Rückgabewert ist die neue Länge des Arrays. Die Funktion verändert das Array.

Siehe mozilla.org

```
Array.unshift (Element1 [,ElementN] )
```

Beispiel:

```
var foo = new Array('a', 'b', 'c');

foo.unshift('x','y')

console.log(foo)
```

Ergibt als Output:

```
10,4
type: string
```

valueOf()

Die Datentypen `object`, `number`, `string`, `boolean` besitzen die Methode `valueOf()`. Siehe Seite 307.

Beispiel:

```
var foo=Array.of(10,4);

var fooObjekt=foo.valueOf();

console.log(fooObjekt);

console.log('type: ' + typeof fooObjekt);
```

Ergibt als Output:

```
10,4
type: object
```

Operatoren

Operatoren mit nur einer Expression (und somit nur einen Operanden) werden Unäre Operatoren genannt. Der Ternäre Operator hat 3 Operatoren. Für Expression siehe Seite 285.

Arithmetische Operatoren

Siehe auch Zahlen-Literal (Seite 282) und Datentyp `number` (Seite 299).

Operator				Beispiel	Ergebnis	
+	Addition			3 + 2	5	Seite 365 (String-Konkatenation Seite 380)
-	Subtraktion			3 - 2	1	Seite
*	Multiplikation			3 * 2	6	Seite 364
/	Division	Inequality		3 / 2	1.5	Seite 364
%	Modulo	Remainder		3 % 2	1	
**	Exponentiation			3 ** 2	9	
+	Plus Vorzeichen	Unary plus	unärer Operator	+1	+1	
–	Minus Vorzeichen	Unary negation	unärer Operator	-1	-1	
++	Inkrement		unärer Operator	++1	2	Pre-Inkrement (1. erhöhen, 2. auswerten/zuweisen)
				1++	2	Post-Inkrement 1. (zuweisen, 2. erhöhen)
--	Dekrement		unärer Operator	--1	0	Pre-Dekrement (1. senken, 2. zuweisen)
				1--	0	Post-Dekremtent (1. zuweisen, 2. senken)

In- und Dekrement

Bei einem Post-Inkrement wird zuerst der Wert ausgewertet und dann erhöht, bei einem Pre-Inkrement wird zuerst erhöht. In diesem Beispiel wird zuerst der Wert `i` ausgewertet und `c` zugewiesen:

```
var i=0, c=0
 i = 5;
 c = i++;
console.log(c);
```

Ergibt als Output:

```
5
```

Beispiel2: In diesem Beispiel wird zuerst der Wert i um 1 erhöht:

```
var i=0, c=0
 i = 5;
 c = ++i;
console.log(c);
```

Ergibt als Output:

```
6
```

Beispiel3:

```
var x=2, y=0;
y = x++ + x++;
console.log('y: '+y);
console.log('x: '+x);
```

Ergibt als Output:

```
5
4
```

Erklärung:

```
x++   +  x++
2     +  x++          x hat den Wert 3
2     +  3            x hat den Wert 4, y hat den Wert 5
```

Multiply *

Wenn einer der Operanden nicht vom Datentyp `number` ist, findet eine automatische Typkonvertierung mit der Funktion Number() statt (siehe Seite 302).

Regeln für den Rückgabewert:

Wenn einer der beiden Operanden `NaN` ist, dann ist `NaN` der Rückgabewert. `NaN` siehe Seite 301

`Infinity * 0`	ergibt `NaN`.	`Infinity` siehe Seite 301
`Infinity` * negativer Zahl	ergibt `-Infinity`	
`Infinity` * positiver Zahl	ergibt `Infinity`	
`Infinity` * `Infinity`	ergibt `Infinity`	

Beispiel:

```
var x = Infinity * -3

console.log (x)

console.log (typeof x)
```

Ergibt als Output:

```
-Infinity
```

Divide /

Wenn einer der Operanden nicht vom Datentyp `number` ist, findet eine automatische Typkonvertierung mit der Funktion `Number()` statt (siehe Seite 302).

Operand1 / Operand2

Regeln für den Rückgabewert:

Wenn einer der beiden Operanden `NaN` ist, dann ist `NaN` der Rückgabewert. `NaN` siehe Seite 301

0 / 0 ergibt `NaN`

positive Zahl / 0	ergibt `Infinity`	
negative Zahl / 0	ergibt `-Infinity`	
`Infinity` / `Infinity`	ergibt `NaN`	`Infinity` siehe Seite 301
`Infinity` / positive Zahl	ergibt `Infinity`	
`Infinity` / negative Zahl	ergibt `-Infinity`	

Beispiel:

```
var x = -3 / 0

console.log (x)

console.log (typeof x)
```

Ergibt als Output:

```
-Infinity
```

Add +

Wenn einer der Operanden nicht vom Datentyp `number` ist, findet eine automatische Typkonvertierung mit der Funktion Number() statt (siehe Seite 302). Ausnahme: Ist einer der Operanden ein String, finden eine Konkatenation statt (Seite 380).

Regeln für den Rückgabewert:

Wenn einer der Operanden `NaN` ist, dann ist `NaN` der Rückgabewert.

`Infinity + Infinity` ergibt Infinity
`-Infinity + Infinity` ergibt NaN
`-Infinity + -Infinity` ergibt -Infinity

Subtract -

Wenn einer der Operanden nicht vom Datentyp `number` ist, findet eine automatische Typkonvertierung mit der Funktion Number() statt (siehe Seite 302).

Regeln für den Rückgabewert:

Wenn einer der Operanden `NaN` ist, dann ist `NaN` der Rückgabewert.

`Infinity - Infinity` ergibt NaN
`-Infinity - Infinity` ergibt -Infinity
`-Infinity - -Infinity` ergibt NaN

Modulus

Wenn einer der Operanden nicht vom Datentyp `number` ist, findet eine automatische Typkonvertierung mit der Funktion Number() statt (siehe Seite 302).
Operand1 wird durch Operand2 dividiert. Der Rückgabewert ist der Rest der Division.

Regeln für den Rückgabewert:

Wenn einer der beiden Operanden `NaN` ist, dann ist `NaN` der Rückgabewert. `NaN` siehe Seite 301
Wenn einer der beiden Operanden `Infinity` ist, dann ist `NaN` der Rückgabewert.
`0 % 0` ergibt `NaN`

Beispiel:

```
var x = 6 % 4                          //  6 durch 4, Rest ist 2

console.log (x)

console.log (typeof x)
```

Ergibt als Output:

```
2
```

Vergleichsoperatoren

Man unterscheidet zwischen Überprüfung auf Gleichheit (Equality) und Überprüfung auf Relation.

Equality

Bei Überprüfung auf Gleichheit unterscheidet man zwischen strikter und abstrakter Gleichheit.

Strict Equality

Nicht nur der Wert, sondern auch der Datentyp muss identisch sein. Siehe Datentypen Seite 298 .

Beispiel:

```
3 === "3"      ergibt false
3 === 3        ergibt true
```

Die Werte `NaN` (Datentyp `number`) können nur mit der Funktion `isNaN()`, siehe Seite 301, oder mit der Methode `Object.is()` auf strikte Gleichheit überprüft werden. Nur `Object.is()` erkennt zB den Unterschiede zwischen `+0` und `-0`.

Abstract Equality

Auch **Loose Equality** genannt. Der Wert muss identisch sein, der primitive Datentyp darf unterschiedlich sein.

Wenn ein Operand **`number`** ist, und der andere ein **`string`** (mit Ziffern), wird (für den Vergleich) der Datentyp `string` in `number` umgewandelt. Dies geschieht automatisch mit der Funktion `Number()` (siehe Seite 302).

Beispiele:

```
3 == "3"               ergibt true
3 == 3                 ergibt true
'carol' == 'Carol'     ergibt false      String-Vergleich ist case sensitive.
'carol' == 'carol'     ergibt true
```

Ist einer der Operanden ein `object`, wird versucht über die Eigenschaft `value.Of` einen Wert für den Vergleich zu bekommen. Ist `value.Of` nicht vorhanden, wird versucht mit `toString()` einen Wert (für den Vergleich) zu erzeugen (Seite 307).

Beispiel: Boolean-Objekt wird automatisch in `number` umgewandelt (`true` ist `1`, `false` ist `0`)

```
var foo1 = new Boolean (true)
var foo2 = new Boolean (false)

console.log(foo1.valueOf());                          // true
console.log(foo1.toString());                         // true

if (foo1 == 1) {  console.log('foo1 ist >= 1'); }
if (foo2 == 0) {  console.log('foo2 ist <= 0'); }
```

Ergibt als Output:

```
foo1 ist == 1
foo2 ist == 0
```

NaN

Ist ein Operand `NaN` (Datentyp `number`), ist ein Vergleich mit Gleichheitsoperator nicht möglich. Überprüfung auf Wert `NaN` siehe Seite 301.

Beispiele:

```
foo=0/0;
console.log( NaN===NaN )            // false  Übereinstimmung nicht erkannt
console.log( foo===NaN )            // false  Übereinstimmung nicht erkannt
console.log( isNaN(foo) )                     // true  Übereinstimmung erkannt
console.log( Object.is(foo,NaN) )             // true  Übereinstimmung erkannt
```

Ergibt als Output:

```
false
false
true
true
```

Kombinierte Gleichheitsoperatoren

Operator			Beispiel	Ergebnis
==	Gleichheit	Equality	3 == 3	true
			3 == 4	false
!=	Ungleichheit	Inequality	3 != 3	false
			3 != 4	true
===	Strikte Gleichheit	Strict equality	3 === "3"	false
			3 === 3	true
!==	Strikte Ungleichheit	Strict inequality	3 !== "3"	true
			3 !== 4	false

Gleichheitsoperatoren und Object.is() im Vergleich:

Wert 1	Wert 2	== Operator	=== Operator	Object.is()
undefined	undefined	true	true	true
null	null	true	true	true
true	true	true	true	true
false	false	true	true	true
"foo"	"foo"	true	true	true
{ foo: "bar" }	x	true	true	true
0	0	true	true	true
+0	-0	true	true	false
0	false	true	false	false
""	false	true	false	false
""	0	true	false	false
"0"	0	true	false	false
"9"	9	true	false	false
[1,2]	"1,2"	true	false	false
new String("foo")	"foo"	true	false	false
null	undefined	true	false	false
null	false	false	false	false
undefined	false	false	false	false
{ foo: "bar" }	{ foo: "bar" }	false	false	false
new String("foo")	new String("foo")	false	false	false
0	null	false	false	false
0	NaN	false	false	false
"foo"	NaN	false	false	false
NaN	NaN	false	false	true

Beispiel:

```
if (+0 == -0)  console.log( 'yep 1' )                  // true     Unterschied nicht erkannt

if (Object.is(+0,-0))  console.log( 'yep 2' )          // false    Unterschied erkannt
```

Ergibt als Output:

```
yep 1
```

Relationaloperatoren

Ist einer der Operanden ein `object`, wird versucht über die Eigenschaft `value.Of` einen Wert für den Vergleich zu bekommen. Ist `value.Of` nicht vorhanden, wird versucht mit `toString()` einen Wert (für den Vergleich) zu erzeugen (Seite 307).

Wenn ein Operand **number** ist, und der andere ein **string** (mit Ziffern), wird (für den Vergleich) der Datentyp `string` in `number` umgewandelt. Dies geschieht automatisch mit der Funktion `Number()` (siehe Seite 302).

Wenn ein Operand **number** ist, und der andere ein **boolean**, wird (für den Vergleich) der Datentyp `boolean` in `number` umgewandelt. `true` ergibt `1` und `false` ergibt `0`.

Sind beide Operanden string, wird jedes Zeichen einzeln nach ihrem ASCII-Wert verglichen.

Ist ein Operand `NaN` (Datentyp `number`), ist kein Vergleich mit Relationaloperatoren möglich. Überprüfung auf Wert `NaN` siehe Seite 301.

Operator			Beispiel	Ergebnis	
>	größer als	Greater than	`4 > 3`	`true`	
			`3 > 4`	`false`	
			`'a' < 'b'`	`true`	ASCII-Wert (Seite231) von a, ist kleiner als ASCII-Wert von b
>=	größer als oder gleich	Greater than or equal to	`7 >= 7`	`true`	
			`6 >= 7`	`false`	
<	kleiner als	Less than	`7 < 9`	`true`	
			`9 < 7`	`false`	
			`'a' > 'A'`	`true`	ASCII-Wert (Seite231) von a, ist kleiner als ASCII-Wert von b
<=	kleiner oder gleich	Less than or equal to	`3 <= 3`	`true`	
			`4 <= 3`	`false`	

Beispiel1: Boolean-Objekt wird automatisch in `number` umgewandelt (`true` ist `1`, `false` ist `0`)

```
var foo1 = new Boolean (true)
var foo2 = new Boolean (false)

console.log(foo1.valueOf());                    // true
console.log(foo1.toString());                   // true

if (foo1 >= 1) {  console.log('foo1 ist >= 1'); }
if (foo2 <= 0) {  console.log('foo2 ist <= 0'); }
```

Ergibt als Output:

```
foo1 ist >= 1
foo2 ist <= 0
```

Beispiel2: String mit String Vergleiche

```
if ('AB' > 'AA') {   console.log('AB ist > AA'); }        // B kommt nach A

if ('Aa' > 'AA') {   console.log('Aa ist > AA'); }        // kleines a kommt nach großem A

if ('1' < 'A') {  console.log('1 ist < A'); }       // A kommt nach 1

if ('ö' > 'Ö') {  console.log('ö ist > Ö'); }       // kleines ö kommt nach großem Ö

if ('ö' > 'z') {  console.log('ö ist > z'); }       // kleines ö kommt nach kleinem z
```

Ergibt als Output:

```
AB ist > AA
Aa ist > AA
1 ist < A
ö ist > Ö
ö ist > z
```

Beispiel3: `string` (`NaN`) mit `number` Vergleich

```
if ('a' >= 0 ) { console.log('a >= 0') }              // aus 'a' (string)  wird NaN
 else if ('a' < 0 ) { console.log('a < 0') }

var x=Number('a')                    // aus 'a' (string)  wird NaN

console.log('x ist: ' + x)
```

Ein `string`, der sich nicht nach `number` konvertieren lässt (weil er zB keine Zahlen enthält), erhält den Wert `NaN` Überprüfung auf Wert `NaN` siehe Seite 301. Ergibt als Output:

```
x ist: NaN
```

Beispiel4: `string` (`Zahl`) mit `number` Vergleich

```
if ('a' >=0 ) { console.log('a >= 0') }              // aus 'a' (string)  wird NaN
 else if ('a' < 0 ) { console.log('a < 0') }

var x=Number('a')                    // aus 'a' (string)  wird NaN

console.log('x ist: ' + x)
```

Ein `string`, der sich nicht nach `number` konvertieren lässt (weil er zB keine Zahlen enthält), erhält den Wert `NaN`
Ergibt als Output:

```
x ist: NaN
```

Ternäre Operator (Ternary Operator)

Der ternäre Operator benötigt 3 Operanden und hat 2 Operatoren: **?** und **:**
Syntax:

(*Conditionexpression*) **?** *Expression1* **:** *Expression2*

Wenn die Expression der Bedingung (Condition) true ist, gibt der Operator den Wert von *Expression1* zurück (ansonsten den Wert von *Expression2*). Siehe Expression Seite 285.
Da der Rückgabewert von einer Bedingung abhängt, wird dieser Operator auch **Conditional Operator** genannt.

Beispiel1:

```
x = (1) ? 6 : 3-3
console.log(x)
```

Ergibt als Output `6` (da 1 true ist, wird der 1. Ausdruck zurückgegeben).

Beispiel2:

```
x = (2-2) ? 6 : 3-3
console.log(x)
```

Ergibt als Output `0` (da 2-2 `false` ist, wird der 2. Ausdruck zurückgegeben).

Logische Operatoren (Logical Operators)

Logische Operatoren werten Expressions (siehe Seite 285) als Boollean (Seite 304) aus.

Diese Werte werden als `false` gewertet:

```
undefined
null
false
0
NaN
''                              (Leerstring)
```

Logische Operatoren werden oft in bedingter Flusskontrolle (siehe Seite 382) verwendet.

Operator					Beispiel	Ergebnis
&&	logisches UND	AND		true, wenn beide Operanden true sind	1 && 0+3	true
				false, wenn mindestens ein Operand false ist	1 && 4-4	false
||	logisches ODER	OR		true, wenn mindestens ein Operand true ist	7 || 0	true
				false, wenn beide Operanden false sind	2-2 || NaN	false
!	logisches NICHT	NOT	unärer Operator	false, wenn der Operand true ist	!0	true
				true, wenn der Operand false ist	!1	false
					!(1-1)	true

Inklusive OR

Das logische OR wird auch **Inklusive OR** genannt (die Auswertung ist auch true, wenn beide Operanden true sind).

JavaScript kennt ein **Bitwise XOR**, aber kein **Logical XOR**. Bei einem XOR darf nur ein Operand true ergeben.

truthy, falsy

Wenn ein Wert (bzw Expression) zu `true` konvertiert werden kann, wird er **truthy** (siehe mozilla.org) genannt. Wenn ein Wert zu `false` konvertiert werden kann, wird er **falsy** genannt (siehe mozilla.org).

!!

Zwei Rufzeichen (zweimal logische NOT) ergeben den Boolean-Wert von einem Ausdruck:

```
var x = !!0
console.log (x)                         // false
console.log (typeof x)                  // boolean
```

Ergibt:

```
false
boolean
```

||

Das logische ODER ist eine **Short-Circuit Evaluation**, siehe Seite 372.

ist einer der Operanden `true`, dann ist der Rückgabwert auch `true` (ansonsten `false`):

Operand1	Operand2	Rückgabewert
true	true	true
true	false	true
false	true	true
false	false	false

Wenn einer der Operanden nicht vom Datentyp boolean ist, gelten diese Regeln für den Rückgabewert:

- Ist der erste Operand ein `object`, dann ist der erste Operand (das Objekt) der Rückgabewert.
- Wenn der erste Operand `false` ergibt, dann ist der zweite Operand der Rückgabewert.
- Wenn beide Operanden `object` sind, dass ist der erste Operand (1. Objekt) der Rückgabewert.
- Wenn beide Operanden `null` sind, dann ist `null` der Rückgabewert.

Beispiel1:

```
var x = false || true                //  x ist true
```

Beispiel2:

```
var Hund = {rasse: 'Dackel'};
var x = false || Hund                // x ist Objekt Hund
```

&&

Das logische AND ist eine **Short-Circuit Evaluation**, siehe Seite 372.

Ist einer der Operanden `false`, dann ist der Rückgabwert auch `false` (ansonsten `true`):

Operand1	Operand2	Rückgabewert
true	true	true
true	false	false
false	true	false
false	false	false

Wenn einer der Operanden nicht vom Datentyp `boolean` sind, gelten diese Regeln für den Rückgabewert:

- Sind beide Operanden `object`, dann ist das 2. Objekt der Rückgabewert
- Ist nur der erste Operand ein `object`, dann ist der 2. Operand der Rückgabewert.
- Ist nur der zweite Operand ein `object`, dann ist wird das Objekt nur dann zurückgegeben, wenn der 1. Operand `true` ergibt
- Ist einer der Operanden `null`, dann ist `null` der Rückgabewert
- Ist einer der Operanden `NaN`, dann ist `NaN` der Rückgabewert
- Ist einer der Operanden `undefined`, dann ist `undefined` der Rückgabewert

Beispiel1:

```
var x = false || true                //  x ist true
```

Beispiel2: NaN

```
var x = NaN && 9
console.log (x)
console.log (typeof x)
```

Ergibt als Output:

```
NaN
number
```

Short-Circuit Evaluation (Kurzschlußauswertung)

Bei einem logischen AND müssen beide Operanden die Bedingung erfüllen. Erfüllt die erste Expression (bzw Operand) die Bedingung nicht, wird die zweite Expression nicht überprüft (bzw ausgeführt).

In diese Beispiel wird `x` nicht erhöht. Da 0 `false` ergibt, wird `x++` nicht ausgeführt (bzw überprüft):

```
var x=0;
if (0 && x++) { }
console.log(x)
```

Ergibt als Output:

```
0
```

Die Überprüfung der Expressions wird bei der ersten nicht erfüllten Bedingung abgebrochen.
Dies wird Short-Circuit Evaluation (Kurzschlußauswertung) genannt. Logische Operatoren haben somit eine Assoziativität von links nach rechts.

In diesem Beispiel wird x erhöht. Die zweite Expression wird überprüft und ausgeführt:

```
var x=0;
if (0 || x++) { }
console.log(x)
```

Ergibt als Output:

```
1
```

Bitwise Operators

Alle Zahlen (auch Ganzzahlen) sind intern als **double-precision floating-point number** abgelegt (64bit IEEE 754).
Für **bitwise operators** wird aus **double** ein 32bit **integer** (big endian). Siehe **endian** Seite 235.

Die Konstante `Number.MIN_SAFE_INTEGER` (`-9007199254740991`) enthält den Integer-Minimalwert und `Number.MAX_SAFE_INTEGER` (`9007199254740991`) den Maximalwert.

Operator					Beispiel	Ergebnis
&	bitweises UND	AND		1, wenn beide Bits 1 sind	`0b1010 & 0b1100`	`0b1000`
|	bitweises ODER	OR		1, wenn eines der Bits 1 ist	`0b1010 | 0b1100`	`0b1110`
^	bitweises XOR	XOR		1, wenn beide Bits unterschiedlich	`0b1010 ^ 0b1100`	`0b0110`
~	bitweises NOT	NOT	unary Operator	negiert das Bit (0 wird 1, 1 wird 0)	`~0b1010`	`0b0101`
<<	left shift				`0b1010 << 1`	`0b10100`
					`0b1010 << 2`	`0b101000`
>>	right shift			Vorzeichen bleibt erhalten		
>>>	zero fill right shift					

Comma Operator

Der Komma-Operator hat die niedrigste Präzedenz (siehe Seite 378). Der Operator wertet 2 Ausdrücke aus und gibt den Wert vom 2. Ausdruck zurück. Sind mehrere Komma-Operatoren vorhanden, wird der letzte Operand als Rückgabewert genommen.

Beispiel 1:

```
var x
x = 1, 2
console.log(x)
```

Der Zuweisungsoperator hat die Präzedenz von 3 und wird deshalb vor dem Komma-Operator ausgeführt.
Darum ist der Output:

```
1
```

Beispiel 2:

```
var x
x = (1, 2);
console.log(x)
```

Die runden Klammern haben die Präzedenz von 19. Darum wird zuerst der Komma-Operator ausgeführt.
Der Operator gibt den 2. Wert zurück:

```
2
```

Beispiel 3:

```
var x=10, y
y = (1, 2, ++x - 5);
console.log(y)
```

Die höchste Präzedenz hat das Inkrement (Prefix siehe Seite 363): zuerst wird 10 um 1 erhöht. Von 11 wird 5 abgezogen.
Der Komma-Operator gibt den Wert 6 zurück:

```
6
```

Beispiel 4:

```
var x=10, y
y = (1,2, ++x - x++);
console.log(y)
console.log(x)
```

Mit der Expression ++x erhält x den Wert 11.
Der Ausdruck x++ gibt 11 zurück (die Erhöhung um 1 auf 12 erfolgt später, da Postfix).
y ist 0, da 11 minus 11 Null ergibt. x ist 12. Ergibt als Output:

```
0
12
```

Beispiel 5:

```
var x
x = (1, 2, 3, 4, 5);
console.log(x)
```

Die runden Klammern haben die Präzedenz von 19. Darum wird zuerst der Komma-Operator ausgeführt.
Der Operator gibt immer den 2. Wert zurück. Der endgültige Wert des Ausdrucks ist 5.

```
5
```

void Operator

Der `void` Operator wertet den Operanden (Expression) aus und gibt (immer) `undefined` zurück.

Beispiel:

```
var x=0;
console.log(void x++ )
console.log(x)
```

Ergibt als Output:

```
undefined
1
```

typeof Operator

Der `typeof` Operator gibt den Datentyp des Operanden zurück. Der Rückgabewert ist ein string.

Mögliche Rückgabewerte:

```
undefined
null
boolean
number
string
function
object
```

Aus historischen Gründen gibt `typeof` bei `null` ein `object` zurück (obwohl `null` ein eigener Datentyp ist).

Beispiele:

```
console.log(typeof undefined)         // undefined

console.log(typeof null)              // object

console.log(typeof true)        // boolean
console.log(typeof 1)           // number
console.log(typeof "")          // string

console.log(typeof function() {} )    // function

console.log(typeof {} )               // object
```

instanceof Operator

Mit dem Operator `instanceof` kann überprüft werden, ob ein Objekt von einer bestimmten Klasse abstammt:

Syntax: *Objektname* `instanceof` *Klassenname*

Beispiel: ist das Objekt Hund eine Instanz der Klasse Hund?

```
class Hund  {
 }

const Bello = new Hund();

console.log (Bello instanceof Hund)
```

Ergibt als Output:

```
true
```

Zuweisungsoperator (Assignment Operator)

Der Assignment Operator `=` weist einer Variable einen Wert zu.

Da die Assoziativität rechts-nach-links ist, muss die Variable links vom Operator sein.

Der Variable `foo` wird das Literal `9` zugewiesen (siehe **Literale** Seite 282):

```
foo = 9
```

Variablen kopieren (Variable-zu-Variable Zuweisung)

primitive datatype

Bei primitiven Datentypen sind beide Variablen unabhängig voneinander und können somit eigene (unabhängige) Werte besitzen (**Primitive Values** werden im **Stack** gespeichert). Beispiel:

```
var foo1 =55;

var foo2 = 66;              // foo1 hat weiterhin den Wert 55

var foo1 = 44;              // foo2 hat weiterhin den Wert 66

console.log('foo1 ist: ' + foo1 + '    und foo2 ist: ' + foo2)    //  foo1 ist: 44  und  foo2 ist: 66
```

complex datatyp

Wird einer Variable eine Variable vom Datentyp `object` zugewiesen, erhält die neue Variable nur einen Verweis auf die alte Variable (**Reference Values** werden im **Heap** gespeichert):

jede Änderung in einem Objekt, bewirkt gleichzeitig auch eine Änderung im anderen Objekt

Dies geschieht auch, wenn das Objekt an eine Funktion übergeben wird: Seite 409

Objects are passed and copied by reference.

Beispiel:

```
var hund1 = new Object();

var hund2 = hund1;                          // hund2 die Variable hund1 zuweisen

hund2.name = "Bello";                       // hund2 den Namen Bello zuweisen
console.log('hund1: ' + hund1.name);        // beide Hunde heissen jetzt Bello
console.log('hund2: '+ hund2.name);

hund1.name = "Max";                         // hund1 den Namen Max zuweisen
console.log('hund1: ' + hund1.name);        // beide Hunde heissen jetzt Max
console.log('hund2: '+ hund2.name)
```

Compound Operator

Ein Compound Operator ist eine **Kombination** von einem **arithmetischen Operator** und dem **Zuweisungsoperator**.

Beispiel1:

```
var x=3
var y=9

y-=x                    // y = y - x
                        // y = 9 - 3

console.log (y)
```

Ergibt als Output:

```
6
```

Beispiel2:

```
var x=3
var y=9

y/=x                    // y = y / x
                        // y = 9 / 3

console.log (y)
```

Ergibt als Output:

```
3
```

Beispiel3:

Kombinierter Operator	Beispiel	ist ident mit
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y
<<=	x <<= y	x = x % y
>>=	x >>= y	x = x >> y
>>>=	x >>>= y	x = x >>> y
&=	x &= y	x = x & y
|=	x |= y	x = x | y
^=	x ^= y	x = x ^ y

Destrukturierende Zuweisung (Destructuring Assignment)

Bei der destrukturierenden Zuweisung werden die Werte eines Objektes (oder eines Arrays) mit nur einer Zuweisung in mehrere Variablen zerlegt.

Array-Destrukturierung

In diesem Beispiel geschieht die Zerlegung (in Einzelzuweisungen) über den Indexwert:

```
var foo = [11,22,33]

var a=foo[0]
var b=foo[1]
var c=foo[2]
var d=foo[3]

console.log(a,b,c,d);
```

Ergibt als Output:

```
11 22 33 undefined
```

Bei der **destrukturierenden Zuweisung** geschieht die Zerlegung in einem Schritt (mit nur einer Zuweisung):

```
var foo = [11,22,33]

var [a,b,c,d]=foo;

console.log(a,b,c,d);
```

Ergibt als Output (die 4. Indexposition ist unbelegt):

```
11 22 33 undefined
```

Diese Art der destrukturierenden Zuweisung funktioniert nicht nur bei einem Array, sondern bei jedem **iterable Object**.
Siehe auch Array-Destrukturierung bei einem Funktionsargument Seite 406.

Variable Swap

Mit dem **Destructuring Assignment** kann recht einfach der Wert zweier Variablen getauscht werden:

```
var x = 5, y = 10;
[x,y] = [y,x];
console.log(x)
console.log(y)
```

Ergibt als Output:

```
[33, 44]
```

spread Operator

Mit dem **spread Operator** `...` werden die restlichen Feldwerte in ein neues Array kopiert.
In diesem Beispiel der 3. und 4. Feldwert:

```
foo = [11,22,33,44]
var [x,y, ...foobar]=foo
console.log(foobar)
```

Ergibt als Output:

```
[33, 44]
```

Objekt-Destrukturierung

Bei der Destrukturierung eines Objektes müssen die identischen Identifier (Seite 284) verwendet werden.

Siehe auch Objekt-Strukturierung bei einem Funktionsargument Seite 406.

Beispiel:

```
var foo = { x: 11, y: 22, z : 33 }

var { y, x, z, a } = foo;

console.log(x)
console.log(y)
console.log(z)
console.log(a)
```

Ergibt als Output:

```
11
22
33
undefined
```

Ohne Deklaration (Seite 287) muss die linke Seite zusätzlich in runde Klammern gesetzt werden (ansonsten wird es als Programmblock erkannt):

```
( { y, x, z, a } ) = foo;
```

Assoziativität (Associativity)

Die Assoziativität bestimmt die horizontale Verarbeitungsreihenfolge bei Operatoren mit gleicher Präzedenz (Seite 378).

Beispiel1: Der Division-Operator hat eine Assoziativität von Links nach Rechts:

```
8 / 4                                   von links nach rechts:8 dividiert durch 4
```

Beispiel2: Der Zuweisungsoperator ist rechtsassoziativ:

```
var x=2, y=3;
x = y = 1;
console.log(x);
console.log(y);
```

Zuerst erhält y den Wert 1. Danach erhält x den Wert von y.

Ergibt als Output:

```
1
1
```

Präzedenz (Precedence)

Die Operator-Präzedens bestimmt die Priorität der Verarbeitungsreihenfolge. Die Präzedens 1 hat die geringste Priorität.

Im Beispiel 1 ergibt die Expression `9/3-2` den Wert 1:

```
var foo = 9 / 3 - 2
console.log (foo)
```

Zuerst wird die Division durchgeführt (Präzedenz 13), danach die Subtraktion (Präzedenz 12).

Ergibt als Output:

```
1
```

19	Grouping		(…)
18	Member Access	left-to-right	… . …
	Computed Member Access	left-to-right	… […]
	new (mit argument list)		new … (…)
17	Function Call	left-to-right	… (…)
	new (ohne argument list)	right-to-left	new …
16	Postfix Increment (Seite 363)		… ++
	Postfix Decrement (Seite 363)		… --
15	Logical NOT (Seite 370)	right-to-left	! …
	Bitwise NOT (Seite 372)	right-to-left	~ …
	Unary Plus	right-to-left	+ …
	Unary Negation	right-to-left	- …
	Prefix Increment (Seite 363)	right-to-left	++ …
	Prefix Decrement (Seite 363)	right-to-left	-- …
	typeof (Seite 374)	right-to-left	typeof …
	void (Seite 374)	right-to-left	void …
	delete	right-to-left	delete …
14	Exponentiation (Seite 363)	right-to-left	… ** …
	Multiplication (Seite 363)	left-to-right	… * …
	Division (Seite 363)	left-to-right	… / …
	Remainder (Seite 363)	left-to-right	… % …
13	Addition (Seite 363)	left-to-right	… + …
	Subtraction (Seite 363)	left-to-right	… - …
12	Bitwise Left Shift (Seite 372)	left-to-right	… << …
	Bitwise Right Shift (Seite 372)	left-to-right	… >> …
	Bitwise Unsigned Right Shift (Seite 372)	left-to-right	… >>> …
11	Less Than (Seite 367)	left-to-right	… < …
	Less Than Or Equal (Seite 367)	left-to-right	… <= …
	Greater Than (Seite 367)	left-to-right	… > …
	Greater Than Or Equal (Seite 367)	left-to-right	… >= …
	in	left-to-right	… in …
	instanceof (Seite 375)	left-to-right	… instanceof …
10	Equality (Seite 366)	left-to-right	… == …
	Inequality (Seite 366)	left-to-right	… != …
	Strict Equality (Seite 366)	left-to-right	… === …
	Strict Inequality (Seite 366)	left-to-right	… !== …
9	Bitwise AND (Seite 372)	left-to-right	… & …
8	Bitwise XOR (Seite 372)	left-to-right	… ^ …
7	Bitwise OR (Seite 372)	left-to-right	… | …
6	Logical AND (Seite 370)	left-to-right	… && …
5	Logical OR (Seite 370)	left-to-right	… || …
4	Conditional (Ternärer Operator) Seite 369	right-to-left	… ? … : …
3	Assignment (Zuweisung) Seite 379	right-to-left	… = …
			… += …
			… -= …
			… **= …
			… *= …
			… /= …
			… %= …
			… <<= …
			… >>= …
			… >>>= …
			… &= …
			… ^= …
			… |= …
2	yield	right-to-left	yield …
	yield*	right-to-left	yield* …
1	Spread (Seite)		… …
0	Comma / Sequence (Seite 373)	left-to-right	… , …

String-Konkatenation

Siehe auch String-Literal (Seite 282) und Datentyp `string` (Seite 306).

Wenn einer der beiden Operanden ein String ist, werden die Operanden nicht als Zahl bewertet, sondern als String. Der Rückgabewert ist die Kombination der beiden Strings.

Operator			Beispiel	Ergebnis	
+	String-Verkettung		`'a2' + '4' + 3`	a243	siehe auch Addition Seite 363

Beispiel1: + als String-Konkatenation:

```
var x = "5" + "3" + "a"

console.log (x)                         // 53a
console.log (typeof x)                  // string
```

Ergibt als Output:

```
53a
string
```

Beispiel2: + als Addition-Operator

```
var x = 5 + 3

console.log (x)
console.log (typeof x)
```

Ergibt als Output:

```
8
number
```

Beispiel3: + als Addition-Operator und als String-Konkatenation

```
var x = (5 + 3) + "a"

console.log (x)                         // 8a
console.log (typeof x)                  // string
```

Ergibt als Output:

```
8a
string
```

spread Operator

Der Spread Operator besteht aus 3 Punkten und zerlegt ein Array in ein Argumented List.

```
function details ( [name, ...derRest] )
 {
    console.log (`Name: ${name} `)                          //1. Parameter ist name
    for (var i=0;i<derRest.length;i++)
    {
       console.log(`i ${i}: ` + derRest[i]);                //der 2. bis 4. Parameter ist in der Rest
    }
 }
var hund = ['Bello',2,'Dackel','braunes Fell']

details (hund);
```

Ergibt als Output:

```
Name: Bello
i 0: 2
i 1: Dackel
i 2: braunes Fell
```

Control Flow (Flusskontrolle)

Man unterscheidet zwischen **bedingter Flusskontrolle** (Seite 382), **Schleifen** (Seite 386) und Unterprogrammen. Mit den Sprunganweisungen `break` und `continue` wird der Fluss in Schleifen verändert, mit `return` wird eine Funktion verlassen.
Mit `threw` wird eine **Exception** ausgelöst, die mit einem **Exception Handler** abgefangen wird.

Variablen sind in den Programmblöcken nur dann lokal, wenn sie im Programmblock mit `let` oder `const` deklariert worden sind (siehe Seite 287).

Conditional Statements (Bedingte Flusskontrolle)

Für bedingte Abläufe gibt es den **ternären Operator** (Seite 369), **switch** (Seite 384) und **if/else-Anweisungen**.

if

Bei der if-Flusskontrolle wird eine Expression (siehe Seite 285) ausgewertet.
Syntax:

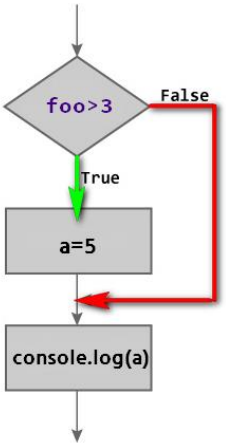
```
if (Expression) Anweisungen
```

Ist das Ergebnis der Expression `true` (siehe Boolean Seite 304), wird der Programmblock der If-Bedingung ausgeführt. Ist das Ergebnis `false`, wird dieser `if`-Programmblock übersprungen. Ist der `if`-Programmblock mehrzeilig, muss er zwingend in geschweiften Klammern stehen.

Für Logische Operatoren Seite 370, für Vergleichsoperatoren Seite 366, für die Kurzschluss-Auswertung Seite 372.

In diesem Beispiel wird die Expression `foo>3` geprüft:

```
var foo = 4;
var a=1;
if (foo>3) // ist true
   {
      a=5;
   }
console.log(a)   // 5
```



Ergibt als Output:

```
5
```

In diesem Beispiel wird die Expression `foo<3` geprüft:

```
var foo = 4;
var a=1;
if (foo<3) {  a=5;  }

console.log(a)   // 1
```

Ergibt als Output:

```
1
```

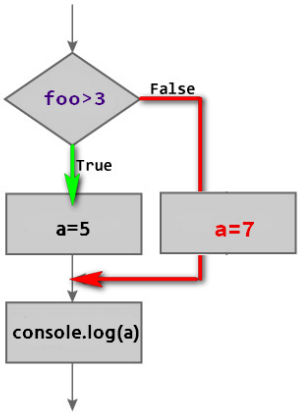
if else

Bei der if-else-Flusskontrolle wird eine Expression (siehe Seite 285) ausgewertet.

```
if (Expression) Anweisungen
   else Anweisungen
```

Ist das Ergebnis der Expression `true` (siehe Boolean Seite 304), wird der Programmblock der If-Bedingung ausgeführt.
Ist das Ergebnis `false`, wird dieser else-Programmblock ausgeführt.

In diesem Beispiel wird die Expression `foo<3` geprüft:




```
var foo = 4;
var a=1;

if (foo<3) {  a=5;  }
   else  {  a=7;  }

console.log(a)   // 7
```

Ergibt als Output:

```
7
```

if else if

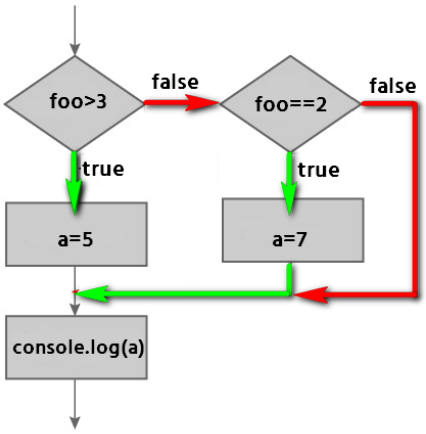
Bei der `if else if` Flusskontrolle gibt es 2 Bedingungen. Zunächst wird nur die erste Expression ausgewertet. Ist das Ergebnis `true`, wird der dazugehörige Programmblock ausgeführt.
Ist das Ergebnis `false`, wird die zweite Expression ausgewertet (falls `true`, wird der dazugehörige Programmblock ausgeführt).

Beliebig viele `else if` Abfragen sind möglich (alle werden sequentiell überprüft).
Optional ist am Ende aller `else if` Abfragen der `else` Block möglich.

Syntax:

```
if (Expression) Anweisungen
   else if  (Expression) Anweisungen
   [ else Anweisungen ]
```

Beispiel 1:

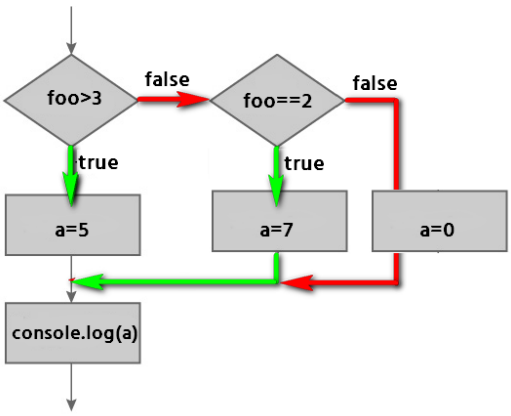



```
var foo = 2;
var a=1;

if (foo>3) {  a=5;  }
   else if (foo==2) {  a=7;  }

console.log(a)
```

Ergibt als Output:

```
7
```

Beispiel 2: mit einem `else` Block




```
var foo=1, a=1;
if (foo>3) {  a=5;  }
   else if (foo==2) {  a=7;  }
   else { a=0; }

console.log(a)
```

Ergibt als Output:

```
0
```

switch

Bei `switch` wird das Ergebnis einer Expression (Seite 285) mit `case`-Expressions verglichen. Beliebig viele case-Abfragen sind möglich (sie werden sequentiell durchlaufen).

Wenn das Expression-Ergebnis mit einer `case`-Expression übereinstimmt, werden alle folgenden Anweisungen, bis zum ersten vorkommenden `break` durchgeführt (falls kein break mehr vorkommt, bis zum switch-Blockende).

Die optionale `default` Block wird ausgeführt, wenn kein passendes case gefunden worden ist (entspricht einem `else` bei `if`).

Für Logische Operatoren Seite 370, für Vergleichsoperatoren Seite 366, für die Kurzschluss-Auswertung Seite 372.

Syntax:

```
switch (Expression)
{
    case (Literal): Anweisungen
    [break]

    [default: Anweisungen]
}
```

Beispiel 1:

```
var a="", b=""
switch (5+4)
 {
    case 2: b="a2"; break
    case 9: b="a9"; break
    case 12: b="a12"; break
    case 15: b="a15";
 }
 console.log(b)
```

Ergibt als Output:

```
a9
```

Beispiel 2: **fall-through**

Nach dem übereinstimmenden `case`, werden alle Anweisungen bis zum ersten `break` ausgeführt (auch wenn `case` oder `default`-Anweisungen dazwischenliegen):

```
var a="", b=""
switch (5+4)
 {
    case 2: b="a2"; break
    case 9: b="a9";
    case 12: b="a12"; break;          // fall-through bis inklusive case12
    case 15: b="a15"; break
 }
 console.log(b)
```

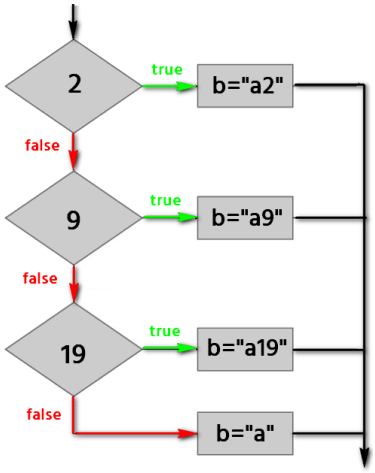
Ergibt als Output:

```
a12
```

Beispiel 3: default

Gibt es keine `case`-Übereinstimmung, werden alle Anweisungen zwischen `default` und dem ersten (folgenden) `break` durchgeführt (falls es kein `break` mehr gibt, bis zum `switch`-Blockende).
Die Position der `default`-Anweisung spielt keine Rolle.

```
var a="";
var b=""
switch (5+2)
 {
    case 2: b="a2"; break
    default: b="a"; break
    case 9: b="a9"; break
    case 19: b="a19"; break
 }
 console.log(b)
```



Ergibt als Output:

```
a
```

Beispiel 4: default mit **fall-through**

Alle Anweisungen zwischen `default` und dem ersten (folgenden) `break` werden durchgeführt (auch wenn case-Anweisungen dazwischen liegen):

```
var a="";
var b="";
switch (5+2)
 {
    case 2: b="a2"; break
    default: b="a";
    case 9: b="a9"; break ;                    // fall-through bis inklusive case9
    case 19: b="a19"; break
 }
 console.log(b)
```

Ergibt als Output:

```
a9
```

Loops (Schleifen)

Man unterscheidet zwischen der `for`-Schleife (Seite 388), der `while`-Schleife und der `do-while`-Schleife (Seite 387).

while Schleife

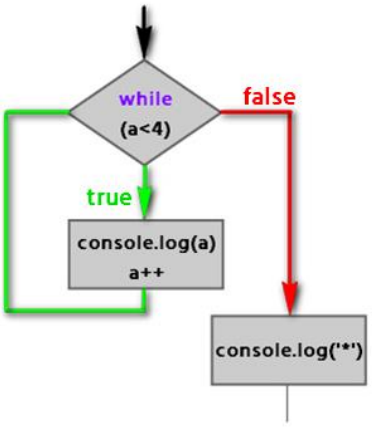
Am Schleifenkopf wird mit `while` eine Expression (siehe Seite 285) ausgewertet. Solange das Ergebnis der Expression `true` ist (siehe Boolean Seite 304), wird der Programmblock der Schleife ausgeführt.

Bei mehrzeiligen Anweisungen muss der Schleifen-Programmblock in geschweiften Klammern stehen.

Da die Exit-Bedingung am Anfang (Kopf) steht, wird die Schleife **Pre-test Loop** genannt .

Syntax:

```
while (Expression)
{
   Anweisungen
}
```



Beispiel:

```
var a=1;
while (a<4)
   {
      console.log(a);
      a++
   }
console.log('*');
```

Ergibt als Output:

```
1
2
3
*
```

break

Mit dem Keyord `break` (im Schleifen-Programmblock) wird die Schleife an dieser Position verlassen. Beispiel:

```
var a=1;
while (a<4)
  {
     console.log(a);
     if (a==2) break        // verlasse die Schleife, wenn a gleich 2
     a++
  }
console.log('*');
```

Ergibt als Output:

```
1
2
*
```

continue

Mit dem Keyword `continue` (im Schleifen-Programmblock) wird zum Schleifenkopf gesprungen.

Beispiel:

```
var a=0;
while (a<3)
{
  a++
  if (a==2) continue        // springe zum Schleifenkopf, wenn a gleich 2
  console.log(a);
}
console.log('*');
```

Ergibt als Output (der Wert 2 wird nicht geschrieben):

```
1
3
*
```

do while Schleife

Am Schleifenende wird mit `while` eine Expression (siehe Seite 285) ausgewertet. Solange das Ergebnis der Expression `true` ist (siehe Boolean Seite 304), wird der Programmblock der Schleife ausgeführt.

Bei mehrzeiligen Anweisungen muss der Schleifen-Programmblock in geschweiften Klammern stehen.

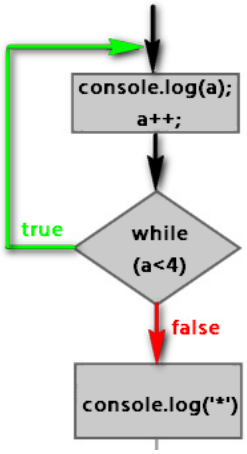
Der Schleifenkopf besteht aus dem Keyword `do`. Da die Exit-Bedingung am Schluss steht, wird die Schleife **Post-test Loop** genannt (somit wird der Programmblock mindestes einmal durchlaufen).

Syntax:

```
do {
      Anweisungen
}  while (Expression)
```

Beispiel:

```
do
{
  console.log(a);
  a++;
}
while (a<4)
console.log('*');
```



Ergibt als Output:

```
1
2
3
*
```

break

Mit dem Keyord `break` (im Schleifen-Programmblock) wird die Schleife an dieser Position verlassen.

Beispiel:

```
do
{
  console.log(a);
  a++;
  if (a==2) break
}
while (a<4)
console.log('*');
```

Ergibt als Output:

```
1
*
```

continue

Mit dem Keyword `continue` (im Schleifen-Programmblock) wird zum Schleifenende (`while`) gesprungen.

Beispiel:

```
var a=0;
do
{
  a++;
  if (a==2) continue
  console.log(a);
}
while (a<3)
```

Ergibt als Output:

```
1
3
```

for Schleife

Am Schleifenkopf wird im `for`-Kopf eine Expression (siehe Seite 285) ausgewertet. Solange das Ergebnis der Expression `true` ist (siehe Boolean Seite 304), wird der Programmblock der Schleife ausgeführt.

Der `for`-Kopf besteht aus 3 Teilen (getrennt mit 2 Semikolons):

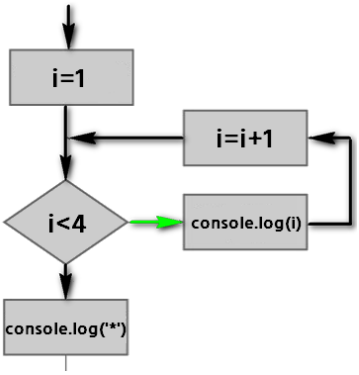
1) **Initialisierung** von Variablen (darunter auch der Schleifenzähler)
2) **Expression** (in der die Schleifenzählervariable vorkommt)
3) **Zuweisungen** mit den Zuweisungsoperatoren (darunter der Schleifenzählerupdate)

Der erste Teil wird nur einmal zu Beginn ausgeführt, Expression-Auswertung und Zuweisungen werden bei jedem Schleifendurchlauf ausgeführt. Für Logische Operatoren Seite 370, für Vergleichsoperatoren Seite 366.
Syntax des Kopfes:

```
for (Variableinitialisierungen ; Expression ; Schleifenzählerupdate)
 {
  Anweisungen
 }
```

Beispiel:

```
var i;
for (i=1;i<4;i=i+1)
 {
    console.log(i)
 }
console.log('*')
```



Ergibt als Output:

```
1
2
3
*
```

Im ersten Teil des Kopfes können beliebig viele Variablen initialisiert (oder nur deklariert) werden. Im dritten Teil können beliebig viele Zuweisungen stattfinden:

```
for (let foobar, i=0, j=5    ; j<8 ;     i--, j++)
console.log(i+' '+j);
console.log('*');
```

Ist der Programmblock der Schleife einzeilig, muss der Block nicht in geschweiften Klammern stehen:
Die 2. Zeile wird dreimal durchlaufen, die 3. Zeile gehört nicht mehr zur Schleife und wird nur einmal ausgeführt.

Ergibt als Output:

```
0 5
-1 6
-2 7
8 *
```

break

Mit dem Keyord `break` (im Schleifen-Programmblock) wird die Schleife an dieser Position verlassen.

Beispiel:

```
var i;
for (i=1;i<4;i=i+1)
 {
    if (i==2) break
    console.log(i)
 }
console.log('*')
```

Ergibt als Output:

```
1
*
```

continue

Mit dem Keyword `continue` (im Schleifen-Programmblock) wird zum Schleifenkopf gesprungen.

Beispiel:

```
var i;
for (i=1;i<4;i=i+1)
 {
    if (i==2) break
    console.log(i)
 }
console.log('*')
```

Ergibt als Output:

```
1
*
```

Label

Siehe Seite 392. Label stehen unmittelbar vor dem Schleifenkopf (`for`, `while`, `do`) und markieren (benennen) die Schleife. Das Label besteht aus dem Identifier (siehe Seite 284) und einem Doppelpunkt.

Labels werden in Kombination mit den Keywords `break` und `continue` verwendet. Durch die Angabe des Labels kann eine bestimmte Schleife angesprochen werden und nicht nur die gerade ausgeführte.

for in

Mit der `for in` Schleife werden sequentiell alle Property-Keys eines Objektes ausgelesen. Keys, die vom Datentyp `symbol` sind, werden von der `for in` Schleife nicht berücksichtigt. Siehe **Objekt** Seite 310 und **Objekt Enumeration** Seite 332.

Die Reihenfolge ist unbestimmt.

Syntax:

```
for (Objekteigenschaft in Objekt)
  {
      Anweisungen
  }
```

Beispiel:

```
var maus = {fell: "braun", id: 32, "magKaese": true}

for (var eigenschaft in maus)
  {
   console.log(eigenschaft + ': ' + maus[eigenschaft])
  }
```

Ergibt als Output:

```
fell: braun
id: 32
magKaese: true
```

for of

Mit der `for of` Schleife wird sequentiell ein **iterable Object** oder ein **Array** ausgelesen.

Im Unterschied zu `for in` gibt es keinen Indexwert (nur den eigentlichen Wert).

Syntax:

```
for (Indexvariable of Objektvariable)
  {
   Anweisungen
  }
```

Beispiel:

```
var tiere =["Mäuse", "Katzen", "Hunde"];
for (var feld of tiere)
  {
   console.log('Wert: ' + feld )
  }
```

Ergibt als Output:

```
Wert: Mäuse
Wert: Katzen
Wert: Hunde
```

Label

Label stehen unmittelbar vor dem Schleifenkopf (`for`, `while`, `do`) und markieren (benennen) die Schleife. Das Label besteht aus dem Identifier (siehe Seite 284) und einem Doppelpunkt.

Syntax:

Labelname:

Labels werden in Kombination mit den Keywords `break` und `continue` verwendet. Durch die Angabe des Labels kann eine bestimmte Schleife angesprochen werden und nicht nur die gerade ausgeführte.

Beispiel: ohne Label wird die erste Schleife (i) zweimal durchlaufen

```
for (var i=1;i<=2;i++)
 {
    for (var j=1;j<=2;j++)
       {
          console.log('i: '+i+' j: '+j);
          break                                    // beende j (die zweite Schleife)
       }
 }
```

Ergibt als Output:

```
i: 1 j: 1
i: 2 j: 1
```

Mit dem Label `ersteSchleife` bezieht sich das break auf die 1. Schleife. i wird nur einmal durchlaufen:

```
ersteSchleife:
for (var i=1;i<=2;i++)
 {
    for (var j=1;j<=2;j++)
       {
          console.log('i: '+i+' j: '+j);
          break ersteSchleife
       }
 }
```

Ergibt als Output:

```
i: 1 j: 1
```

Funktion

Eine Funktion ist ein Unterprogramm. Eine Funktion kann beliebig viele Übergabewerte (Parameter) besitzen. Sie kann 1 Variable zurückgeben.

Datentyp

Die Funktion ist vom Datentyp `function`, zählt aber auch zu den Objekten (siehe **Complex Datatypes** Seite 298 und **Primitive Data Types** Seite 298).

Beispiel: mit dem Operator `typeof` den Datentyp abfragen

```
function foo () {}                  // Funktion namens foo erzeugen
console.log (typeof foo)            // Datentyp  function
```

First-class Object

Funktionen sind First Class Objects:

1) eine Funktion kann ein ein Argument einer (anderen) Funktion sein
2) ein Funktion kann der Return-Wert einer (anderen) Funktion sein
3) eine Expression sein (Function Expression Seite 399)
4) eine IIFE sein (Function Expression, die sich beim Auswerten auch aufruft, Seite 401)
5) jede Funktion ist eine Instanz vom Objekt **`Object`** (und besitzt somit Eigenschaften und Methoden)
6) sie können einer Variable zugewiesen werden (sowohl Funktionsvariable, als auch Funktionsdeklaration)
7) ein Feldwert eines Arrays sein
8) eine Eigenschaft eines Objektes sein (dann werden sie Methoden genannt)

Bei der Deklaration (Seite 398) einer Funktion wird auch ein Objekt erzeugt (Instanz vom Objekt `Object`):

```
function KatzeFuettern ()
    {
        console.log("Maus und Milch");
    }

console.log (typeof KatzeFuettern);                          // function

console.log (KatzeFuettern instanceof Object);               // true (ist Instanz vom Objekt Object)
console.log (KatzeFuettern instanceof Function);             // true

console.log (KatzeFuettern);
```

Ergibt als Output:

```
function
true
true
KatzeFuettern() { console.log("Maus und Milch"); }
```

Eine Funktion hat Properties (Eigenschaften eines Objektes):

```
function KatzeFuettern ()
 {
     console.log("Maus und Milch");
 }

KatzeFuettern.Futter = "Mäusemilch";                         // Eigenschaft mit dem Namen Futter
console.log(KatzeFuettern.Futter);
```

Ergibt als Output:

```
Mäusemilch
```

Einer Variable eine Funktion zuweisen (siehe Funktion-Referenz 394):

```
function KatzeFuettern () {
        console.log("Maus und Milch");
    }

KatzeFuettern.Futter = "Mäusemilch";

PussyFuettern = KatzeFuettern;

console.log (PussyFuettern.Futter);
console.log (typeof PussyFuettern);                          // PussyFuettern ist vom Typ function

console.log (PussyFuettern instanceof KatzeFuettern);        // false
console.log (PussyFuettern instanceof Object);               // PussyFuettern ist Instanz von Object

console.log (KatzeFuettern instanceof Object);               // KattzeFuettern ist Instanz von Object
```

Ergibt als Output:

```
Mäusemilch
function
false
true
true
```

Referenz

Wie bei allen Objekten (siehe Seite 310), ist der Funktionsname (siehe Identifier 284) nur eine Referenz auf den Heap. Wird einer Variable ein Objekt zugewiesen (Zuweisung, also keine neue Instanz), enthält die neue Variable ebenfalls nur eine Referenz (auf die gleiche Stelle im Heap).

Objects are passed and copied by reference.

Beispiel1: einer Variable eine Funktion zuweisen und eine Variable-Eigenschaft (Property siehe Seite 316) ändern

```
function addiere (x,y) {
   return x + y
}

var addition = addiere                  // der Variable addition wird die Funktion addiere zugewiesen
                                        // bei der Zuweisung wird die Funktion nicht aufgerufen

addiere.foo="666"                       // Funktion addition wird mitgeändert, da
                                        // beide Funktionen auf die gleiche Stelle im Heap zeigen:

console.log (addition.foo)              // 666

addition.foo="777"                      // Funktion addiere wird mitgeändert:

console.log (addiere.foo)               // 777

// -------------------------
addiere = null                          // Variable addiere wird auf null gesetzt
console.log (addition.foo)              // 777 (Variable addition ist weiterhin vorhanden)
```

Erhält als Output:

```
5
```

Beispiel2: einer Objekt-Eigenschaft eine Funktion zuweisen

```
function addiere (x,y) {                // Deklaration der Funktion addiere (mit den Parameter x und y)

    return x + y
}

var rechne = {}                         // Objekt rechne erstellen (Literal-Erstellung)

rechne.addition = addiere;              // das Objekt rechne erhält eine Methode namens addition
                                        // die Methode addition verweist auf die Funktion addiere

var summe = rechne.addition (3,2)       // die Methode addition (des Objektes rechne) addiert 3 und 2

console.log (summe)                     // 5
```

Erhält als Output:

```
5
```

Beispiel3: einer Variable eine Funktionsvariable zuweisen

```
var verdoppeln = function (x) {  return x * 2  }

var z = verdoppeln                      // es wird keine Kopie wie bei Primitive Values erzeugt,
                                        // sondern nur die Referenz (Zeiger) wird kopiert


var y = z (3)

console.log (typeof z)                                  // function

console.log (z instanceof verdoppeln)                   // false

console.log (z instanceof Object)                       // true

console.log (y)                                         // 6
```

Erhält als Output:

```
function
false
true
6
```

Beispiel 4a: Zwei Objekte verwenden als Methode die gleiche externe Funktion.
`this` zeigt per **Implicit Binding** auf das (richtige) gerade verwendete Objekt.

```
function Funktion () {
    console.log( this.rasse )
 }

var rasse = "Dackel";

var hund1 =
    {
       rasse: "Schäfer",
       meineFunktion: Funktion
    }

var hund2 =
    {
       rasse: "Pudel"
    }

hund1.meineFunktion();                              // Schäfer

hund2.meineFunktion = hund1.meineFunktion           // hund2 bekommt per Zuweisung die Funktion

hund1.meineFunktion()                               // Schäfer

hund2.meineFunktion()        // Pudel (this bezieht sich auf Objekt hund2)
                             // trotz gleicher Funktion (bzw gleicher Funktion-Referenz)
```

Ergibt als Output:

```
Schäfer
Schäfer
Pudel
```

Beispiel 4b: Objekt hund2 erhält per Zuweisung eine Methode von Objekt hund1
`this` zeigt per **Implicit Binding** auf das (richtige) gerade verwendete Objekt.

```
var rasse = "Dackel";

var hund1 =
 {
    rasse: "Schäfer",
    meineFunktion: function ()
          {
             console.log( this.  rasse )
          }
 }

var hund2 = {  rasse: "Pudel"   }

hund1.meineFunktion();                              // Schäfer

hund2.meineFunktion = hund1.meineFunktion           // hund2 bekommt per Zuweisung die Funktion

hund1.meineFunktion()                               // Schäfer


hund2.meineFunktion()        // Pudel (this bezieht sich auf Objekt hund2)
                             // trotz gleicher Funktion (bzw gleicher Funktion-Referenz)
```

Ergibt als Output:

```
Schäfer
Schäfer
Pudel
```

Erzeugung

Bei der Erzeugung (Deklaration) von Funktionen unterscheidet man zwischen

1 Funktionsdeklaration als **Function Statement** (Seite 398)

Wird die Funktion erst nach ihrer ersten Verwendung deklariert, wird die Deklaration vom Compiler intern nach oben verschoben (**Function Hoisting** siehe Seite 399).

```
function meineFunktion (x) { return x * 2; }
```

2 Funktionsdeklaration mit einer **Function Expression** (Seite 399)

Bei der **Function Expression** wird die Funktionsdeklaration einer Variable zugewiesen.
Bei dieser Art der Funktionserzeugung findet kein **Function Hoisting** (Seite 399) statt.

```
function meineFunktion (x) { return x * 2; }
```

3 **Konstruktor Function**

```
var meineFunktion = new Function ( "x", " return x * 2;" )
```

4 **Immediately Invoked Function Expressions**

Die Function Expression erhält zusätzlich vorne und hinten runden Klammern und wird vom Compiler bereits bei der Zuweisung einmal ausgeführt. Die Expression gibt nur den Rückgabewert der Funktion zurück. Die Funktion wird danach gelöscht. (Seite 401)

Beispiel 1a: Deklaration der Funktion `verdoppeln`

```
function verdoppeln (x) {
    return x * 2
 }

var y=verdoppeln (3)

console.log (y)                                  //6
console.log (typeof verdoppeln)                  //function
console.log (typeof y)                           //number
```

Beispiel 1b: die Deklaration als **Function Expression** (siehe Seite 399.)

```
var verdoppeln = function (x) {
    return x * 2
 }
var y=verdoppeln (3)
console.log (y)                                  //6
```

Beispiel 1c: Die **Function Expression** in **Arrow Notation** (Seite 400)

```
var verdoppeln =  (x) =>  x * 2

var y=verdoppeln (3)

console.log (y)                                  //6
```

Beispiel 1d: Erzeugung mit dem Konstruktor **Function**

```
var verdoppeln = new Function ("x"," return x * 2;" )

var y=verdoppeln (3)

console.log (y)                                  //6
```

Function Statement

Die Deklaration einer Funktion besteht aus 2 Teilen. Dem **Kopf** (Funktionsname, Parameter) und dem **Body** (Programmblock):

```
function Funktionsname ( [Parameter1, Parameter2, ...] )
{
   Anweisungen
   [ return Expression ]
}
```

Parameter und das Keyword `return` sind optional (das Keyword `return` kann beliebig oft angegeben werden).

return

Das optionale Keyword `return` beendet die Ausführung vom Unterprogramm und gibt den Wert der Expression zurück. Die Anzahl der `return`-Keywords in einer Funktion ist unbegrenzt.
Ohne `return` gibt die Funktion den Wert `undefined` zurück (also Konstruktor ist der default-Wert `this`).

Beispiel:

```
function foo (a,b)
    {
       return a*b                          // Funktion wird an dieser Stelle beendet

       console.log('drin')                 // diese Zeile wird nicht ausgeführt
    }
 var x = foo (2,3)
 console.log (x)
```

Ergibt als Output:

```
6
```

Overloading

Es gibt kein **Overloading** (unterschiedliche, aber gleich lautende Funktionen). Bei Overloading erkennt der Compiler anhand der Argumentanzahl, welche Funktion gemeint ist.

Beispiel: die zweite, gleichlautende Funktion überschreibt die erste

```
function foo (a,b)   {
       return a*a
 }


function foo (a)  {
       return a
 }

 var x = foo (2)
 console.log (x)
```

Ergibt als Output:

```
2
```

Function Declaration Hoisting

Funktionsdeklarationen sollten vor ihrer ersten Verwendung stehen. Sollte das nicht der Fall sein, wird von JavaScript die Deklaration automatisch nach oben verschoben (dies wird **Function Hoisting** genannt).

Verschoben wird zum Anfang des gegenwärtigen Programmblocks.

Aus diesem Beispiel:

```
{
      {
         foo()                                  // Aufruf der Funktion foo vor der Funktionsdeklaration
         function foo() {}
      }
}
```

wird intern:

```
{
      {
         function foo() {}                      //  Funktionsdeklaration am Anfang des jeweiligen Blockes
         foo()
      }
}
```

Die Expression einer Funktionsdeklaration (Seite 399), die einer Variable zugeordnet wird, wird intern nicht verschoben (kein Function Hoisting):

```
foo ();                                 // Error: foo is not a function

let foo = function() { }
```

Erzeugt als Output:

```
can't access lexical declaration foo before initialization
```

Function Expression

Die Funktion-Deklaration wird einer Variable zugewiesen (siehe auch **Referenz** auf Seite 394).

Beispiel:

```
var addiere = function foo (x,y)
 {
    return x+y
 }
var s = addiere (3,5)
console.log (s)
```

Ergibt als Output:

```
 8
```

foo ist nur intern sichtbar, nicht ausserhalb:

```
var addiere = function foo (x,y)
 {
    console.log('foo intern: ' + typeof foo)          // Funktion foo ist intern bekannt

    console.log('addiere intern: ' +typeof addiere)   // Funktion addiere ist intern bekannt
 }

console.log('foo extern:' + typeof foo)               // Funktion foo ist extern unbekannt
```

Ergibt als Output:

```
foo intern: function
addiere intern: undefined
foo extern:undefined
```

Anonyme Function Expression

In diesem Beispiel hat die Deklaration keinen Funktionsnamen (**Anonyme Funktion**):

```
var addiere = function (x,y)                    // anonyme Funktion
 {
    console.log(this.name)
    return x+y
 }

console.log ('3+5 = ' + addiere(3,5))
```

Ergibt als Output:

```
3+5 = 8
```

Kein Function Declaration Hoisting

Die Expression einer Funktionsdeklaration (Seite 399), die einer Variable zugeordnet wird, wird intern nicht verschoben (kein Function Hoisting, siehe Seite 399):

```
foo ();                                         // Error: foo is not a function
                                                // Funktionsaufruf vor ihrer Deklaration

let foo = function()
 {
    console.log('f');
 }
```

Erzeugt als Output :

```
can't access lexical declaration `foo' before initialization
```

Arrow Notation

Ab ES6 gibt es bei **anonymen Funktionen** als Shorthand die **Arrow Notation**.

Beispiel 1a:

Funktionsdeklaration mit dem Keyword `function`

```
var addiere = function (x,y)                    // anonyme Funktion
   {
      return x+y
   }

console.log ('3+5 = ' + addiere(3,5))
```

Ergibt als Output:

```
3 + 5 = 8
```

Beispiel 1b:

Das Keyword `function` fällt weg, nach den Parametern folgt der Arrow **=>**

```
var addiere = (x,y) =>                          // anonyme Function-Expression mit Arrow-Notation
   {
      return x+y
   }

console.log ('3+5 = ' + addiere(3,5))

console.log (typeof addiere)                    // function
```

Ergibt als Output:

```
3 + 5 = 8
function
```

Beispiel 1c:

Besteht der Programmblock nur aus einem `return` und einer Expression, sind `return` und geschweifte Klammern optional:

```
var addiere = (x,y) => x+y

console.log ('3+5 = ' + addiere(3,5))

console.log (typeof addiere)                    // function
```

Ergibt als Output:

```
3 + 5 = 8
function
```

Konstruktor Function

Eine Funktion mit dem Konstruktor `Function` erzeugen:

```
var meineFunktion = new Function ( "x", " return x * x;" )
```

erzeugt die gleiche Funktion wie:

```
function meineFunktion (x) { return x * 2; }
```

und mit einer Function Expression (das Keyword `function` kann als Operator angesehen werden):

```
var meineFunktion = function (x) { return x * 2; };
```

Immediately Invoked Function Expression (IIFE)

Eine IIFE ist eine Function Expression, die sich beim Auswerten auch aufruft (im Gegensatz zu einer normalen Function Expression, siehe Seite 399).

Vorteile:

1) Variablen innerhalb von der IIFE befinden sich im Function Scope
2) Funktion wird ohne expliziten Aufruf einmalig ausgeführt

Struktur:

```
(function()
 {
    // Anweisungen
 }
)
();
```

Damit der Compiler die IIFE als eigenständige Expression ansieht (und nicht mit vorherigen Source Code in Verbindung bringt), wird vor der Expression meist ein Operator gesetzt. Zum Beispiel `void`

Beispiel1: IIFE mit dem `void` Operator (siehe Seite 374):

```
void
 (
    function meineFunktion ()
    {
       console.log( "wuff")
    }
 )
 ();

console.log ( typeof meineFunktion )          // undefined (meineFunktion ist unbekannt)
```

Ergibt als Output:

```
wuff
undefined
```

Beispiel2: IIFE mit dem = Operator (der Zuweisungsoperator weist den Wert von `return` zu)

```
var foo =
 (
    function meineFunktion ()
    {
       return "wuff"
    }
 )
 ();

console.log ( foo )                          // foo
console.log ( typeof meineFunktion )         // undefined (meineFunktion ist unbekannt)
```

Ergibt als Output:

```
wuff
string
```

Beispiel3: Im IIEFE kann beliebiger Code gekapselt werden. Der Programmblock läuft im Function Scope der Expression.

```
(function() {

    "use strict";

    var foo = 1

    function machWas() {
        console.log ('*')
    }

    function machNochWas() {
        console.log ('+')
    }
}());

console.log ( typeof machWas )            // undefined
console.log ( typeof foo )                // undefined
console.log ( typeof machNochWas )        // undefined
```

Ergibt als Output:

```
undefined
undefined
undefined
```

Parameter

Argument versus Parameter

Der Parameter ist der Identifier (Seite 284) bei der Deklaration, das Argument der Wert bei einem Aufruf.

Parameter werden manchmal auch **Formal Arguments** genannt.

Parameter Scope

Parameter-Variablen sind nur innerhalb der Funktion sichtbar (gültig). Siehe **Function Scope** Seite 407.

Wenn ein Objekt an eine Funktion übergeben wird, wird die Referenz übergeben, siehe **Variable Copy** auf Seite 375. Der externe Identifier vom Objekt ist auch in der Funktion vorhanden, siehe Seite 409.

Parameter Anzahl

Die Funktion-Deklaration kann beliebig viele Parameter besitzen. Ebenso können beliebig viele Argumente an eine Funktion übergeben werden (JavaScript kennt kein **Overloading**, siehe Seite 398).

Die lokale built-in Variable `arguments` ist ein Array ähnliches Objekt (siehe Seite 404), dass alle Übergabewerte des aktuellen Aufrufes enthält (Feldelement 0 ist das erste Argument, Feldelement 1 ist das zweite Argument, ...).

Property length

Die Eigenschaft `length` enthält die Anzahl der Parameter, mit der die Funktion deklariert worden ist.

```
var addiere = function (x,y)                        // anonyme Funktion
 {
    return x+y
 }

console.log ('3+5 = ' + addiere(3,5))

console.log ('Anzahl Parameter = ' + addiere.length)        //
```

Ergibt als Output:

```
2
```

"use strict"

Die Verwendung von "use strict" (Seite 291) bewirkt zwei Einschränkung bei der Verwendung von Funktionen:

1. Identifier (Funktionsname und Parameter) darf nicht `eval` oder `arguments` lauten
2. Identifier der Argumente müssen eindeutig sein (Paramater-Name darf nur einmal vorkommen)

Beispiel1: Funktionsname lautet `eval`

```
function eval ()
 {
    // bei "use strict": Uncaught SyntaxError: Unexpected eval or arguments in strict mode
 }
```

Beispiel2: Parametername `foo` wird doppelt verwendet

```
function foobar (foo, foo)
 {
    // bei "use strict": Duplicate parameter name not allowed in this context
 }
```

arguments

Die lokale built-in Variable `arguments` ist ein Array ähnliches Objekt, dass alle Übergabewerte des aktuellen Aufrufes enthält (Feldelement 0 ist das erste Argument, Feldelement 1 ist das zweite Argument, ...).

Die einzelnen Feldwerte können jeden beliebigen Datentyp beinhalten.

Beispiel1: zwei Argumente werden der Funktion übergeben

```
function foo ()
 {
    return '1: ' + arguments[0] + '    2: ' + arguments[1]
 }

console.log ( foo ('a','b') )
```

Ergibt als Output:

```
 1: a    2: b
```

Beispiel2: mit einer `for of` Schleife (Seite) das Array `arguments` auslesen 351

```
function foo ()
 {

    for (var feld of arguments)
       {
          console.log('Wert: ' + feld )
       }

 }

 foo ('a','b','c')
```

Ergibt als Output:

```
Wert: a
Wert: b
Wert: c
```

Funktion als Argument

Das Argument kann auch eine Funktion sein.

Beispiel 1:

```
let addieren = function (a,b) { return a+b }

let calc = function (zahl1, zahl2, callback)
 {
    console.log (typeof callback)                // function

    return callback (zahl1, zahl2)
 }

var z = calc (2,5,addieren)

console.log (z)
```

Ergibt als Output:

```
function
7
```

Beispiel 2: Function Expression als Argument

```
let calc = function (zahl1, zahl2, callback)
 {
    console.log (typeof callback)

    return callback (zahl1, zahl2)
 }

var z = calc (2,5, function (a,b) {return a + b} )

console.log (z)
```

Ergibt als Output:

```
function
7
```

Destrukturierung

Destrukturierung von Objekten und Arrays, um die Einzelwerte (als Argumente) an die Funktion übergeben zu können:

- siehe Objekt-Destrukturierung Seite 406 und Seite 378
- siehe Array-Destrukturierung Seite 406 und Seite 377
- siehe Funktion-Methode `call()` Seite 411
- siehe Funktion-Methode `apply()` Seite 412

Objekt Destrukturierung

Siehe **Destructuring Assignment** Seite 378. Die Identifier (Seite 284) für die Parameter müssen identisch mit den Objekt-Eigenschaftsnamen sein.

Beispiel:

```
function details ( {name,alter,rasse} )
{
 console.log (`Name: ${name}, Alter: ${alter}, Rasse: ${rasse} `)
}
var hund = new Object();

hund.name='Bello'
hund.rasse='Dackel'
hund.alter=2

details (hund);
```

Ergibt als Output:

```
Name: Bello, Alter: 2, Rasse: Dackel
```

Array Destrukturierung

Siehe Array Destrukturierung Seite 377.

Beispiel1:

```
function details ( [name,alter,rasse] )
 {
    console.log (`Name: ${name}, Alter: ${alter}, Rasse: ${rasse} `)
 }
var hund = ['Bello',2,'Dackel']

details (hund);
```

Ergibt als Output:

```
Name: Bello, Alter: 2, Rasse: Dackel
```

Beispiel2: mit dem spread Operator `...`

```
function details ( [name, ...derRest] )
 {
    console.log (`Name: ${name} `)                          //1. Parameter ist name
    for (var i=0;i<derRest.length;i++)
    {
       console.log(`i ${i}: ` + derRest[i]);                //der 2. bis 4. Parameter ist in der Rest
    }
 }
var hund = ['Bello',2,'Dackel','braunes Fell']

details (hund);
```

Ergibt als Output:

```
Name: Bello
i 0: 2
i 1: Dackel
i 2: braunes Fell
```

Variable Scope

Scope (Visibility) beschreibt die Sichtbarkeit (Gültigkeit) einer existierenden Variable in einem bestimmten Bereich. Eine Variable beginnt ab ihrer Deklaration zu existieren.

Siehe auch **Scope** Seite 422, **Deklaration** Seite 287 und `let` Seite 293.

Für `this`, **Call-Stack** und **Call-Site** (Object Binding) siehe Seite 421.

externe Variable-Deklaration

Variablen, die ausserhalb deklariert worden sind, können auch innerhalb der Funktion sichtbar sein.

Beispiel 1a: `let` Variable-Deklaration, die Funktionsdeklaration befindet sich in einem embedded-Block (Nested Block)

```
{
   let bar=1;                                          //let Variable-Deklaration im Block Scope
      {
         foo()
         function foo()
            {
               console.log ('1: ' + typeof bar);       // number (sichtbar, da im Block Scope)
            }
      }
   console.log ('2: ' + typeof bar);                   // number (sichtbar, da im Block Scope)
}
console.log ('3: ' + typeof bar);                      // undefined (nicht sichbtar, da Out of Scope)
```

Beispiel 1b: `let` Variable-Deklaration, die Funktionsdeklaration befindet sich in einem anderen Block

```
{
   let bar=1;                                          //let Variable-Deklaration
}

{
      {
         foo()
         function foo()
            {
               console.log ('1: ' + typeof bar);       // undefined (nicht sichbtar, da Out of Scope)
            }
      }
   console.log ('2: ' + typeof bar);                   // undefined (nicht sichbtar)
}
console.log ('3: ' + typeof bar);                      // undefined (nicht sichbtar)
```

Beispiel 2A: `var` Variable-Deklaration, die Funktionsdeklaration befindet sich in einem anderen Block

```
{
   var bar=1;                                          //var Variable-Deklaration im Global Scope
}

{
      {
         foo()
         function foo()
            {
               console.log ('1: ' + typeof bar);       // number (sichtbar, da im gleichen Scope)
            }
      }
   console.log ('2: ' + typeof bar);                   // number (sichtbar, da im gleichen Scope)
}
console.log ('3: ' + typeof bar);                      // number (sichtbar, da im gleichen Scope)
```

Beispiel2:

```
function foo ()
 {
    console.log (typeof bar);          // number (sichtbar, da im gleichen Function Scope)

    bar++                              // der neue Wert 3 bleibt auch extern enthalten
 }

var bar = 2                            // bar ist im Global Scope
foo();
console.log( bar );                    // bar ist auch ausserhalb der Funktion weiterhin 3
```

Ergibt als Output:

```
 3
```

interne Variable-Deklaration

Variablen, die innerhalb der Funktion deklariert werden (`var`,`let`,`const`), sind nur innerhalb der Funktion sichtbar.

Beispiel1: intern deklarierte Variable `y`

```
function foo ()   {
    var y = 2                          // die lokale Varible y ist nur innerhalb der Funktion gültig
}
foo();
console.log(typeof y );
```

Erzeugt als Output:

```
undefined
```

Beispiel2: Parameter-Variablen sind nur innerhalb der Funktion gültig. Bei Objekt-Parameter siehe Beispiel 3.

```
function foo (x)  {
    x++
    console.log(`Innerhalb: ${x}`);                 // x ist in der Funktion 3
 }


var x= 10;
foo(2);
console.log( `Ausserhalb: ${x}` );                  // x ist ausserhalb der Funktion 10
```

Ergibt als Output:

```
Innerhalb: 3
Ausserhalb: 10
```

Beispiel3: Parameter ist ein Objekt. Auch wenn die Variable lokal ist, hat sie die gleiche Referenz wie die externe Variable

```
function NameSetzen(meinObjekt)
 {
    meinObjekt.meinName = 'Bello';             // auch  hund.meinName wird mitgeändert

    hund.meinName = 'Max';                     // die externe Variable wird verändert
 }

var hund = new Object();
NameSetzen(hund);

console.log('hund.meinName: ' + hund.meinName);     // hund.meinName: Max
```

Objekt als Funktion-Argument

Wenn ein Objekt an eine Funktion übergeben wird, wird die Referenz übergeben, siehe **Variable Copy** auf Seite 375.

Der externe Identifier vom Objekt ist auch in der Funktion vorhanden.

Objects are passed and copied by reference.

Beispiel1:

```
function NameSetzen(meinObjekt)
 {
    meinObjekt.name='Bello'      //  meinObjekt ist zwar eine lokale Variable,
                                 //  die Referenz zeigt aber an die gleiche Stelle im Heap.
                                 // Eine Änderung bei meinObjekt wirkt sich auch auf das externe Objekt hund aus
 }

var hund = new Object();

NameSetzen(hund);

console.log(hund.name);
```

Ergibt als Output:

```
Bello
```

Beispiel2: der externe Identifier hund ist auch in der Funktion vorhanden

```
function NameSetzen(meinObjekt)
 {
    meinObjekt.meinName = 'Bello';

    hund.meinName = 'Max';                    //die externe Variable wird verändert
 }

var hund = new Object();
NameSetzen(hund);

console.log('hund.meinName: ' + hund.meinName);     // hund.meinName: Max
```

Ergibt als Output:

```
hund.meinName: Max
```

Funktion als Objekt-Eigenschaft

Eine Objekt-Eigenschaft kann auch eine Funktion sein. In diesem Fall wird eine Funktion auch Methode genannt.

Beispiel1:

```
const rechne = {
   addiere: function (a,b) { return a+b; }
  }
console.log(rechne.addiere(4,3) )
```

Ergibt als Output:

```
7
```

Beispiel2: ab ES6 gibt es für Beispiel 1 einen Shortcut (Keyword `function` entfällt)

```
const rechne = {
  addiere (a,b) { return a+b; }
 }
console.log(rechne.addiere(4,3) )
```

Ergibt als Output:

```
7
```

Aufruf (Invoke)

Beim **Call (Invoke)** sind die Parameter (Seite 403) optional und durch Komma getrennt.

In JavaScript ruft `foo (1,2,3)` die gleiche Funktion wie `foo()` und `foo(1)` auf.

Es gibt kein Overloading (unterschiedliche, aber gleich lautende Funktionen). Bei Overloading erkennt der Compiler anhand der Argumentanzahl, welche Funktion gemeint ist.

Siehe auch **Immediately Invoked Function Expressions** auf Seite 401.

Jeder Funktionsaufruf läuft in einem neuen Execution Context (und hat seinen eigenen Scope).

Aufruf-Objektmethoden

Jede Funktion besitzt 3 Objekt-Methoden:

1) Mit der Methode **`call()`** wird ein Objekt übergeben, worauf sich die Funktion mit `this` beziehen soll (siehe Seite 411). Nach dem Objekt folgen die einzelnen Funktionsargumente.
   Für `this`, **Call-Stack** und **Call-Site** siehe Seite 421.
   Siehe mozilla.org

2) mit der Methode **`apply()`** wird ein Objekt übergeben, worauf sich die Funktion mit `this` beziehen soll (siehe Seite 412). Nach dem Objekt folgt ein Array (aus dem automatisch die einzelnen Funktionsargumente gebildet werden). Für `this`, **Call-Stack** und **Call-Site** siehe Seite 421.
   Siehe mozilla.org

3) mit der Methode **`bind()`** wird die Funktion an ein bestimmtes Objekt gebunden. Dieses festgelegte Objekt kann auch mit `call()` oder `apply()` nicht überschrieben werden. Für `this`, **Call-Stack** und **Call-Site** siehe Seite 421.
   Siehe mozilla.org

call()

Dieser Methode wird ein Objekt übergeben.

Die aufgerufene Funktion kann mit dem Keyword `this` auf die Objekt-Eigenschaften zugreifen.

Folgen weitere Parameter (optional), werden diese den Funktionsargumenten zugeordnet. Siehe auch mozilla.org

Syntax: **call** (***Objektname*** [, *Parameter2*, *ParameterN*])

Beispiel1: der Funktion `ichBin` wird einmal das Objekt `bello` und einmal das Objekt `bella` übergeben

```
const bello = { rasse: 'Dackel', fell: 'braun' };
const bella = { rasse: 'Pudel', fell: 'weiss' };

function ichBin ()
 {
    console.log ( `Ich bin ein ${this.rasse}. Mein Fell ist  ${this.fell}.` );
 }

ichBin.call(bello);              // das Keyword  this bezieht sich auf das Objekt  bello

ichBin.call(bella);              // das Keyword this bezieht sich auf das Objekt bello
```

Ergibt als Output:

```
Ich bin ein Dackel. Mein Fell ist braun.
Ich bin ein Pudel. Mein Fell ist weiss.
```

Beispiel2: zusätzlich zum Objekt wird ein Argument übergeben (das Alter des Hundes)

```
const bello = { rasse: 'Dackel', fell: 'braun' };
const bella = { rasse: 'Pudel', fell: 'weiss' };

function ichBin (alter)
   {
   console.log ( `Ich bin ein ${this.rasse}. Mein Fell ist  ${this.fell}. Mein Alter: ${alter}` );
   }

ichBin.call(bello,12);                          // das Keyword  this bezieht sich auf das Objekt  bello
                                                // 12 ist der erste Funktion-Parameter

ichBin.call(bella,13);                          // das Keyword  this bezieht sich auf das Objekt  bella
                                                // 13 ist der erste Funktion-Parameter
```

Ergibt als Output:

```
Ich bin ein Dackel. Mein Fell ist braun. Mein Alter: 12
Ich bin ein Pudel. Mein Fell ist weiss. Mein Alter: 13
```

apply()

Dieser Methode wird ein Objekt übergeben.

Die aufgerufene Funktion kann mit dem Keyword `this` auf die Objekt-Eigenschaften zugreifen.

Der 2. Parameter ist optional und ein Array. Siehe mozilla.org.

Aus dem Array werden automatisch die einzelnen Funktionsargumente (1. Feldwert = 1. Funktionsparameter).

Syntax: **apply** (***Objektname*** [, *Arrayname*])

Beispiel1: der Funktion `ichBin` wird einmal das Objekt `bello` und einmal das Objekt `bella` übergeben

```
const bello = { rasse: 'Dackel', fell: 'braun' };
const bella = { rasse: 'Pudel', fell: 'weiss' };

function ichBin ()
   {
      console.log ( `Ich bin ein ${this.rasse}. Mein Fell ist  ${this.fell}.` );
   }

ichBin.apply(bello);                          // das Keyword this bezieht sich auf das Objekt bello

ichBin.apply(bella);                          // das Keyword this bezieht sich auf das Objekt bella
```

Ergibt als Output:

```
Ich bin ein Dackel. Mein Fell ist braun.
Ich bin ein Pudel. Mein Fell ist weiss.
```

Beispiel2: zusätzlich zum Objekt wird ein Array (mit 2 Elementen) übergeben (Alter und Geschlecht)

```
const bello = { rasse: 'Dackel', fell: 'braun' };
const bella = { rasse: 'Pudel', fell: 'weiss' };

function ichBin (alter,geschlecht)
 {
 console.log (`Ich bin ein ${this.rasse}. Fell: ${this.fell}. Alter: ${alter}, ${geschlecht}` );
 }


ichBin.apply(bello,[5,'männlich']);

ichBin.apply(bella,[3,'weiblich']);
```

Ergibt als Output:

```
Ich bin ein Dackel. Fell: braun. Alter: 5, Geschlecht: männlich
Ich bin ein Pudel. Fell: weiss. Alter: 3, Geschlecht: weiblich
```

call und apply im Vergleich

Beispiel: beim Objekt `bello` wird `call` verwendet, bei Objekt `bella` wird `apply` verwendet

```
const bello = { rasse: 'Dackel', fell: 'braun' };
const bella = { rasse: 'Pudel', fell: 'weiss' };

function ichBin (alter,geschlecht)
 {
  console.log (`Ich bin ein ${this.rasse}. Fell: ${this.fell}. Alter: ${alter}, ${geschlecht}` );
 }


ichBin.call(bello,5,'männlich');                // das Keyword this bezieht sich auf das Objekt bello
                                                // Parameter werden als Argument List übergeben


ichBin.apply(bella,[3,'weiblich']);             // das Keyword this bezieht sich auf das Objekt bella
                                                // Parameter werden als Argument List übergeben
```

Ergibt als Output:

```
Ich bin ein Dackel. Fell: braun. Alter: 5, Geschlecht: männlich
Ich bin ein Pudel. Fell: weiss. Alter: 3, Geschlecht: weiblich
```

bind()

Jede Funktion besitzt die Methode `bind()`. Übergabewert ist ein Objekt.
Sie wird bei der Zuweisung einer Variable (neue Funktion) verwendet. Die neue Funktion ist an das Objekt gebunden.
Dieses festgelegte Objekt kann auch mit `call()` oder `apply()` nicht überschrieben werden.

Beispiel1:

```
const bello = { rasse: 'Dackel', fell: 'braun' };
const bella = { rasse: 'Pudel', fell: 'weiss' };
const wuff = { rasse: 'Schäfer', fell: 'schwarz' };

function ichBin (alter,geschlecht)
{
 console.log (`Ich bin ein ${this.rasse}. Fell: ${this.fell}. Alter: ${alter}, ${geschlecht}` );
 }

var ichBinGebunden=ichBin.bind(wuff);

ichBinGebunden.call(bella,3,'weiblich');              // Objekt bella wird ignoriert
ichBinGebunden.call(bello,5,'männlich');              // Objekt bello wird ignoriert
ichBinGebunden.call(wuff,7,'männlich');
```

Ergibt als Output:

```
Ich bin ein Schäfer. Fell: schwarz. Alter: 3, Geschlecht: weiblich
Ich bin ein Schäfer. Fell: schwarz. Alter: 5, Geschlecht: männlich
Ich bin ein Schäfer. Fell: schwarz. Alter: 7, Geschlecht: männlich
```

Werden neben dem Objekt auch Argumente übergeben, werden diese zu fixen Defaultwerten.

Beispiel2: beim Binding werden die ersten 2 Parameter fix vergeben: 1 und 2

```
function meineFunktion (a,b,c)
 {
    console.log("a:" + a + ", b:" + b + ", c:" + c);
 }


// De-Militarized Zone
var ø = Object.create( null );                          // leeres DMZ Object (Seite 415)


var bindedFunktion = meineFunktion.bind( ø, 1,2 );

bindedFunktion ( 3 ) ;                                  // 1 2 3

bindedFunktion ( 4 ) ;                                  // 1 2 4
```

Ergibt als Output:

```
a:1, b:2, c:3
a:1, b:2, c:4
```

De-Militarized Zone (DMZ)

Wird bei den Methoden `call()` und `apply()` kein Objekt übergeben (das ist ohne Error-Meldung nur ohne `"use strict"` möglich), greift das **Default-Binding**.

Um Seiteneffekten vorzubeugen, wird in diesem Fall der Methode `bind()` ein leeres DMZ Objekt übergeben (dieses DMZ Objekt wird nicht mehr verwendet).

`null` als Übergabe erzwingt Default-Binding

Wird bei call statt einem Objekt der Datentyp null übergeben, greift das Default-Binding.
Bei `"use strict"` wird in diesem Fall ein Error erzeugt.

```
// "use strict"

function ichBin ()
 {
    console.log ('ich bin: ' + this.name)
 }

var carol = {name: "Carol"}

name="Alice"                        // Default Binding Wert

ichBin.call(null)                   // ich bin Alic
```

Ergibt als Output:

```
 Ich bin ein Dackel
```

Bei `"use strict"` ergibt `null` als Übergabewert einen Error:

```
Uncaught TypeError: Cannot read property 'name' of null
```

`bind` mit einem DMZ Objekt

```
function meineFunktion (a,b,c)
 {
    console.log("a:" + a + ", b:" + b + ", c:" + c);
 }


// De-Militarized Zone
var ø = Object.create( null );                          // leeres DMZ Object


var bindedFunktion = meineFunktion.bind( ø, 1,2 );

bindedFunktion ( 3 ) ; // 1 2 3

bindedFunktion ( 4 ) ; // 1 2 4
```

Ergibt als Output:

```
 a:1, b:2, c:3
 a:1, b:2, c:4
```

Function Scope (Script, Block, Closure)

Ob eine extern deklarierte Variable sichtbar ist, wird durch die Art (`var/let`) und durch die Position der Deklaration (Lexical Scope) bestimmt (siehe Seite 407 und Seite 422) .

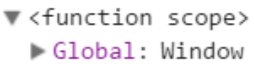
Ob und in welcher Gruppenbezeichnung eine extern deklarierte Variable im **Function Scope** abgelegt wird:

1) eine **externe var-Variable** im **Global Execution Context** wird nicht im Function Scope abgelegt
2) eine **let/const-Variable** im **Global Execution Context** werden in der Gruppe Script abgelegt
3) eine **let/const-Variable** im **Block Execution Context** wird in der Gruppe Block abgelegt
4) eine **let/const-Variable** im **Function Execution Context** wird in der Gruppe Closure abgelegt

Beispiel 1a: `var` Variablen im **Global Execution Context** werden nicht im Function Scope abgelegt

```
var foo=0

function Funktion ()
 {
    foo+
 }

console.dir (Funktion)
```

Ergibt in der Konsole vom Chrome:

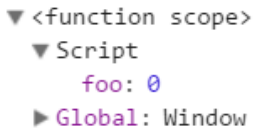


Script

Beispiel 1b: `let/const` Variablen im **Global Execution Context** werden in der Gruppe Script abgelegt

```
let foo=0

function Funktion ()
 {
    foo++
 }

console.dir (Funktion)
```

Ergibt in der Konsole vom Chrome:

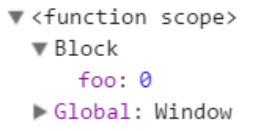


Block

Beispiel2: `let/const` Variablen im **Block Execution Context** werden in der Gruppe *Block* abgelegt

```
{
 let foo=0

 function Funktion ()
    {
       foo++
    }

 console.dir (Funktion)
}
```

Ergibt in der Konsole vom Chrome:



Closure

Beispiel3: Funktionsvariablen einer anderen (äußeren) Funktion, werden im **Function Scope** unter *Closure* abgelegt

```
{
   let foo=0

   function aussen()
      {
         let foo2=0                          // Block-Variable von der Funktion aussen

         function Funktion ()
            {
               foo++                         // Block-Variable von der Function Funktion
               foo2++                        // Closure-Variable von der Function Funktion
            }

         console.dir (Funktion)
      }
}

aussen ()

console.dir (aussen)
console.dir (Funktion)                       // undefined (von aussen nicht sichtbar)
```

Ergibt in der Konsole vom Chrome:

```
  name: "Funktion"
▶ prototype: Object
▶ __proto__: function ()
▼ <function scope>
  ▼ Closure
      foo2: 0
  ▼ Block
      foo: 0
  ▶ Global: Window
```

```
  name: "aussen"
▶ prototype: Object
▶ __proto__: function ()
▼ <function scope>
  ▼ Block
      foo: 0
  ▶ Global: Window
```

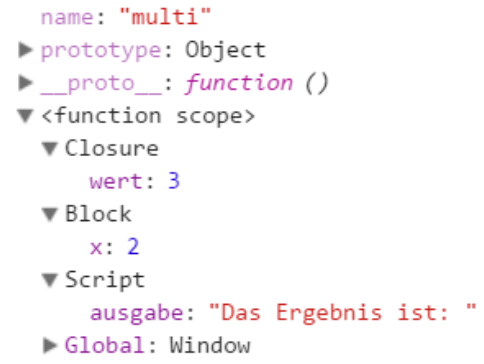
High Order Function

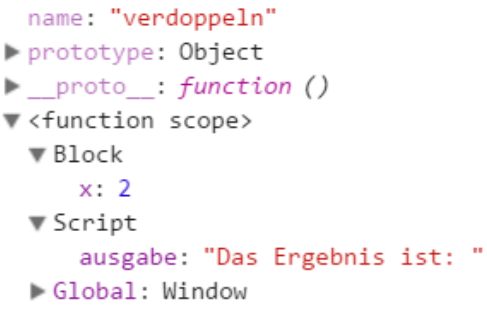
Befindet sich die Funktionsdeklaration in der Funktionsdeklaration einer anderen Funktion, ist die innere Funktion im **Function Execution Context** der höheren Funktion.

Beispiel:

```
let ausgabe = "Das Ergebnis ist: ";                          // Script Scope

   {
      let x = 2                                              // Block Scope

       var verdoppeln = function ()
         {
            var wert = 3                                     // wert ist im Block Scope von verdoppeln

            function multi () { return wert * x }            // wert ist im Closure Scope von multi

            var ergebnis=multi (x)                           // ergebnis ist im Block Scope von verdoppeln

            console.dir (multi)

            return ergebnis

         }
   }

var y = verdoppeln()

console.log (y)                                              //6

console.dir (verdoppeln)
```

Ergibt in der Konsole von Google Chrome:





Callback-Function

Eine Funktion, die per Übergabeparameter eine andere Funktion aufruft, nennt sich **High Order Function**.

Als Beispiel für eine **High Order Function**:

```
Array.some(Suchfunktion)
```

Der Übergabeparameter ist die Funktion, mit der jeder Feldwert überprüft werden soll (siehe mozilla.org).

Gibt die Callback-Funktion eine Expression zurück, die `true` ergibt, wird die Überprüfung abgebrochen und die Methode gibt `false` zurück.
`some()` traversiert das Array und übergibt der Funktion jeweils den Feldwert, den Indexwert und das Array.

Beispiel 1: gibt es mindestens eine gerade Zahl im Array?

```
var zahlen = new Array (3, 5, 12, 8, 7);

function geradeZahl(zahl, index, array)                    // geradeZahl ist die Callback-Funktion
 {
  var gerade = Boolean(!(zahl%2))

  console.log('index: '+index+'   wert: '+zahl+'    gerade: '+gerade)

   return gerade
 }

console.log(zahlen.some(geradeZahl))                       // some ist die High Order Function
```

Ergibt als Output:

```
wert: 3    gerade: false
wert: 5    gerade: false
wert: 12    gerade: true
true
```

Beispiel 2a: add ist die High-Order-Function. callback ist im 1. Aufruf KonsoleLog, im 2. Aufuf KonsoleInfo

```
function add (a,b, callback)                               // add ist die High-Order-Function
 {
      var result = a+b
      callback (result)
      console.log ('Funktionsname: ' + callback.name + ' / a: ' + a+' / b: '+ b)
 }

function KonsoleLog(result)
 {
   console.log ('result-log: ' + result)                   //  KonsoleLog / a: 2 / b: 3
 }

function KonsoleInfo(result)
 {
   console.info ('result-info: ' + result)
 }

add (2,3,KonsoleLog)                  // add ist die High-Order-Function, KonsoleLog wird im 3. Argument aufgerufen

add (7,1,KonsoleInfo)                 // add ist die High-Order-Function, KonsoleInfo wird im 3. Argument aufgerufen
```

Ergibt als Output:

```
result-log: 5
KonsoleLog / a: 2 / b: 3
result-info: 8
Funktionsname: KonsoleInfo / a: 7 / b: 1
```

Beispiel 2c: das Argument ist eine Funktionsdeklaration

```
function add (a,b, callback)                                    // add ist die High-Order-Function
 {
       var result = a+b
       callback (result)
       console.log ('Funktionsname: ' + callback.name + ' / a: ' + a+' / b: '+ b)
 }


add (2,3,
       function foo (result)
          {
             console.info ('result-info: ' + result);
             console.log ('result-log: ' + result)
          }
    )
```

Ergibt als Output:

```
result-info: 5

result-log: 5

Funktionsname: foo / a: 2 / b: 3
```

Call-Stack & Call-Site

Eine Funktion kann eine andere Funktion aufrufen, und diese zweite Funktion kann ein dritte aufrufen. Die Reihenfolge der Funktionsaufrufe (**Function Chain**) werden im **Call-Stack** abgelegt (Last-In, First-Out). Jeder Funktionsaufruf besitzt seinen eigenen **Execution Context** (Seite 416).

Wird die dritte Funktion beendet, läuft das Programm im **Execution Context** der zweiten Funktion weiter. Der **Call-Stack** ist leer, wenn alle aufgerufenen Funktionen beendet worden sind.

`this` (Seite 321) wird vom **Call-Site** bestimmt, der im Zeitpunkt des Funktionsaufrufes gilt.

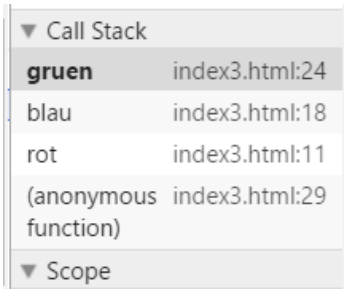
Beispiel 1: Die Funktion `rot` ruft die Funktion `blau` auf. Und Funktion `blau` ruft Funktion `gruen` auf:

```
function rot() {                                    // call-stack ist: rot
                                                    // call-site ist im global scope

                                                    // bei "use strict" wird this nicht mit
                                                    // Global Scope verbunden!

    console.log( "rot" );
    blau();                                         // <-- call-site für blau
 }


function blau() {                                   // call-stack ist: rot -> blau
                                                    // call-site ist in rot
    console.log( "blau" );
    gruen();                                        // <-- call-site für gruen
 }


function gruen() {                                  // call-stack is: rot -> blau -> gruen
                                                    // call-site ist in blau
    debugger;
    console.log( "gruen" );
 }


rot();                                              // <-- call-site für rot
```

Ergibt als Output:

```
rot
blau
gruen
```

Der Call-Stack im Google-Chrome:



Scope (Gültigkeitsbereich)

Scope (Bereich) beschreibt die Sichtbarkeit (Visibility) einer existierenden Variable (bzw Funktion) in einem bestimmten Bereich. Eine Variable oder Funktion beginnt ab ihrer Deklaration (Seite 287) zu existieren.
Ob eine existierende Variable gerade sichtbar ist, hängt vom Ausführungskontext (**Execution Context**) ab.
Es gibt drei Gültigkeitsbereiche:

Global Scope
Variablen und Funktionen die im **Global Execution Context** (äußerster Kontext) deklariert werden, sind an jeder Stelle im Programm verfügbar

Function-level Scope
Funktion
Jede Funktion hat ihre eigenen Kontext. Variablen und Funktionen, die in einer deklariert werden, sind lokal und somit von außen nicht sichtbar (siehe Seite 393)

Eine Ausnahme gibt es bei Argumenten vom Datentyp `object` (siehe Seite 409)

Jeder Funktionsaufruf läuft in einem neuen Execution Context (und hat seinen eigenen Scope).

Block-level Scope
lokale Variable. Lebt nur innerhalb von einem { } Block (inklusive **Nested Blocks**)
Für **`let`** Seite 293, **`const`** Seite 293, **`with`** Seite 428, `catch`-Block
Auch **Funktionsdeklarationen** sind im **Block Scope** (und somit nur innerhalb des Blockes sichtbar und aufrufbar).

Lexical Scope

Eine Variable ist in dem Scope (Block) gültig, in dem sie deklariert worden ist (und in allen dazugehörigen Unterblöcken). Dieses Verhalten wird Lexical Scope genannt.

Wenn ein laufendes Programm von einem Scope in den anderen wechselt, ändern sich die verfügbaren Variablen und Funktionen. Dies wird **Execution Context** (Ausführungskontext) genannt.

Der **Global Execution Context** ist der äußerste Kontext. Variablen und Funktionen, die im äußersten Kontext deklariert werden, sind global und überall im Programm verfügbar (**Global Scope**, siehe Seite 424).

Bei einem Browser ist das *window object* der äußerste Kontext. Alle globalen Variablen und Funktionen sind Eigenschaften und Methoden des *window object*.

Jede Funktion hat seinen eigenen **Function Execution Context** (**Function Scope**, siehe Seite 424)

Scope Chain

Durch das Verschachteln von Funktionsdeklarationen entsteht eine **Scope Chain** (Verkettung von Gültigkeitsbereichen):

```
var Farbe = "blue";                          //Farbe          im    global context
function aendereFarbe()                      //aendereFarbe   im    global context
   {
   var zweiteFarbe = "red";                  //zweiteFarbe    im    function context von aendereFarbe
   function tauscheFarben()                  //tauscheFarben  im    function context von aendereFarbe
      {
            var tempColor = zweiteFarbe;     //zweiteFarbe    im    function context von aendereFarbe
            zweiteFarbe = Farbe;             //Farbe          im    global context
            Farbe = tempColor;               //tempColor      im    function context von tauscheFarben
      }
   tauscheFarben();
   }
aendereFarbe()
console.log ( Farbe )                              // red
```

Unterschied zu PHP:

1) beim zweiten Aufruf von `aendereFarbe()` erscheint `Fatal error: Cannot redeclare tauscheFarben()`

2) weder `Farbe`, noch `zweiteFarbe` ist **In Scope**

Function Scope

Ob eine extern deklarierte Variable sichtbar ist, wird durch die Art (`var/let`) und durch die Position der Deklaration (Lexical Scope) bestimmt (siehe Seite 407 und Seite 422) .

Für **Function Scope** (Script, Block, Closure) siehe Seite 416.

Ob und unter welcher Gruppenbezeichnung eine extern deklarierte Variable im **Function Scope** abgelegt wird:

1) eine **externe var-Variable** im **Global Execution Context** wird nicht im Function Scope abgelegt (ist aber sichtbar)
2) eine **let/const-Variable** im **Global Execution Context** wird in der Gruppe Script abgelegt
3) eine **let/const-Variable** im **Block Execution Context** wird in der Gruppe Block abgelegt
4) eine **let/const-Variable** im **Function Execution Context** wird in der Gruppe Closure abgelegt

Für **Funktion** siehe Seite 393. Für **Variable-Scope** siehe Seite 407.

In Scope

Eine Variable »*In Scope*« ist sichtbar und damit auch verfügbar.

Beispiel:

```
{
 let a=1;
 console.log(typeof a)    // a ist In Scope
}

console.log(typeof a)    // a ist Out Of Scope
```

Ergibt als Output:

```
number
undefined
```

Out of Scope

Eine Variable ist »*Out of Scope*«, wenn sie nicht mehr sichtbar ist.

Beispiel:

```
{
   let foo=1;
}                                  // foo ist out of scope

console.log(typeof foo);           // ausserhalb nicht sichtbar
```

Ergibt als Output:

```
undefined
```

Global Scope

Variablen und Funktionen, die im äußersten Programmblock deklariert werden, sind an jeder Stelle im Programm sichtbar. Sie sind im Global Scope.

Für **Deklaration** siehe Seite 287. Für **Declaration Hoisting** siehe Seite 288.

Beispiel1: die Variable `foo` ist im Global Scope (siehe Keyword `var` Seite 287):

```
function foo ()   {
    console.log ( x );                  //  x ist auch im Scope der Funktion (auch in der Funktion sichtbar)
 }

var x = 1                               //  Deklaration mit var im Global Scope
foo();
```

Ergibt als Output:

```
1
```

Beispiel2: die Funktion `foo` ist nicht im Global Scope. Sie ist außerhalb ihres Blockes nicht sichtbar (siehe Seite 407)

```
{
    function foo () {
       console.log ( x );               //  foo ist in einem Closure (siehe Seite 407)
    }
}

var x = 1
foo();                                  //  foo ist nicht sichtbar: Compiler Error
```

Ergibt als Output:

```
Uncaught ReferenceError: foo is not defined
```

Beispiel2: die Variable `foo` wurde nicht im äußersten Programmblock deklariert, ist aber dank `var` im Global Scope

```
{
    {
         var foo=1
    }
}

console.log (foo)
```

Ergibt als Output:

```
1
```

Function Scope

Für **Funktion** siehe Seite 393. Für **Variable-Scope** siehe Seite 407.

Für **Function Scope** (Script, Block, Closure) siehe Seite 416.

Block Scope

Ab ECMA-Script 6 (Harmony) werden mit dem Keywords `let` (siehe Seite 293) und `const` (Seite 293) Variablen im **Block Scope** deklariert.

Auch **Funktionsdeklarationen** sind im Block Scope (und somit nur innerhalb des Blockes sichtbar und aufrufbar).

let/var Deklarierung

Die gleiche Variablebezeichnung kann zwar im gleichen { } Programmblock nicht noch einmal deklariert werden, aber in einem Unterblock (**Nested Block**).

Nested Blocks erlaubt **Variable Masking** (Seite 427):

```
var foo=2;                          // erste Deklaration (im Global Scope)
 {
   let foo=3;                       // zweite Deklaration (im Block Scope)
                                    // das lokal foo überlagert das globale foo
   console.log('drin ' + foo);
 }
console.log('aussen ' + foo);       // das globale foo hat weiterhin den Wert 2
```

Eine Variable kann in ihrem Block nicht vor ihrer `let`- oder `const`-Deklarierung verwendet werden (im Gegensatz zu var , siehe **Declaration Hoisting** auf Seite 288):

```
x = 1
let x
```

Ergibt als Output:

```
Uncaught ReferenceError: x is not defined
```

Im Gegensatz zu `var`, darf mit `let` (und `const`) die gleiche Variable nicht noch einmal deklariertwerden (zumindest nicht im gleichen Block).

Beispiel1a: foo wird zweimal im gleichen Programmblock, in der gleichen Ebene deklariert:

```
{
   let foo=1;
   let foo=4;
}
```

Ergibt als Output eine Fehlermeldung:

```
Identifier 'foo' has already been declared
```

Beispiel1b: foo wird zweimal im gleichen Programmblock, aber in unterschiedlichen Ebenen deklariert (Nested Block)

```
{
   let foo=1
      {
         let foo=4              // neue foo stellt das alte foo in den Schatten
                                // siehe Variable Masking (Seite 427):
   }
   console.log(foo)    // 1
}
```

Ergibt als Output:

```
1
```

Visibility (Sichtbarkeit)

Die Sichtbarkeit liegt innerhalb des { } Blockes (inklusive Nested Blocks), in der die Variable deklariert worden ist.

Beispiel1a:

```
console.log(typeof foo);              // foo existiert nicht
 {
    const foo=1;                      // foo wird mit const deklariert und ist »in scope«
    console.log(typeof foo);
 }
console.log(typeof foo);              // foo existiert nicht mehr
```

Ergibt als Output:

```
undefined
number
undefined
```

Beispiel1b: zum Vergleich das Beispiel1a mit var:

```
var foo=2;                            // foo ist »in scope« (und zwar im Global Scope)
{

 console.log('drin1 ' + foo);         // foo ist (auch im Block) weiterhin »in gloabl scope«

 var foo=3;                           // foo wird von 2 auf 3 gesetzt

 console.log('drin2 ' + foo);

}

console.log('aussen ' + foo);      // foo ist weiterhin 3
```

Ergibt als Output:

```
drin1 2
drin2 3
aussen 3
```

Beispiel2: foo ist auch in **Nested Block** sichtbar

```
console.log(typeof foo);              // foo existiert nicht
 {
    const foo=1;                      // foo wird mit const deklariert und ist »in scope«
    {
       {
          console.log(typeof foo);    // auch im nested block sichtbar
       }
    }
 }
console.log(typeof foo);              // foo existiert nicht mehr
```

Ergibt als Output:

```
undefined
number
undefined
```

Variable Masking (Variable Shadowing)

Variable Masking tritt auf, wenn eine gleichlautende Variable auch in einem anderen Scope existiert (und dort von dieser überdeckt und in den Schatten gestellt wird).

Beispiel1: an outer scope variable is shadowed by a closer (inner) variable (eine innere verdeckt die äußere Variable)

```
var foo=2;                              // foo im Global Scope
 {
   let foo=3;                           // ein neues foo wird im Block Scope erzeugt
   console.log('drin ' + foo);          // das neue foo verdeckt das erste foo (das sich im Global Scope befindet)
   // foo goes out of scope
 }
console.log('aussen ' + foo);           // das alte foo (im Global Scope) ist weiterhin 2
```

Ergibt als Output:

```
drin 3
aussen 2
```

Beispiel2:

Nested Scopes: foo wird zweimal maskiert (in 2 Scope-Ebenen überdeckt)

Die Verkettung bzw Verschachtelung von Scopes wird **Scope Chain** genannt.

```
var x = 1;
console.log('x: ' + x); // 1

 {
      let x = 2;
      console.log('1. nested scope: ' + x);  // 2

      {
            let x = 3;
            console.log('2. nested scope: ' + x);  // 3
      }
 }

console.log('x: ' + x); // 1
```

Ergibt als Output:

```
x: 1
1. nested scope: 2
2. nested scope: 3
x: 1
```

with

Das Keyword `with` bestimmt für den jeweiligen Programmblock das default-Objekt. Falls das gewünschte Objekt im Programmblock nicht gefunden wird, wird das von `with` bestimmte Objekt probiert.

Der Übergabewert ist das gewünschte Objekt.

Syntax:

```
with (Objektname) { Anweisungen }
```

Sind die Anweisungen einzeilig, können die geschweiften Klammern entfallen.

Bei `"use strict"` ist die Verwendung nicht erlaubt.

Beispiel:

```
with (Math)
 {
    console.log(PI);                  // PI ist nicht vorhanden, darum wird Math verwendet
    console.log(sin(PI));

    console.log(Math.PI);
 }
```

Ergibt als Output:

```
3.141592653589793
1.2246467991473532e-16
3.141592653589793
```

Keywords

Aktuelle und reservierte Keywords	Reservierte Keywords bis ECMAScript 1.3
await (reserved for future) break case class catch const continue debugger default delete do else enum (reserved for future) export extends false (literal value) finally for function if implements (reserved for future) import in instanceof interface (reserved for future) let new null (literal value for future) package (reserved for future) private (reserved for future) protectd (reserved for future) public (reserved for future) return super static (reserved for future) switch this throw true (literal value) try typeof var void while with yield	abstract boolean byte char double final float goto int long native short synchronized transient volatile

Obtrusive and Unobtrusive JavaScript

Obtrusive JavaScript beinhaltet **Inline JavaScript**, bei **Unobtrusive JavaScript** sind HTML und JavaScript getrennt.
Bei **Unobtrusive JavaScript** werden HTML, CSS und JavaScript in drei Ebenen (MVC) aufgeteilt:

HTML	mit HTML-Tags werden Inhalt, Struktur und Semantik bestimmt	Model
CSS	mit CSS wird die Präsentation (Optik) der Seite bestimmt	View
JavaScript	die Skriptsprache sorgt für Interaktion	Controller

Obtrusive JavaScript

Bestimmte HTML-Attribute sind Event-Attribute. Sie starten das JavaScript im Attributwert, wenn das Event erfüllt ist (siehe Event **Handler** Seite 156).
Als Beispiel das Event-Attribut `onclick` und `onsubmit`:

```
<a href="#" onclick="alert('Ade, schnöde Welt (triggered by onclick event)'); ">klick mich</a>

<form action="seite.php" method="post"  onsubmit="return validateForm();">
```

Inline-JavaScript sollte nur dann verwendet werden, wenn der Author keinen Zugriff auf das eingebettete oder externe JavaScript besitzt. Siehe auch Inline Style auf Seite 210.
HTML, CSS und JavaScript sollten getrennt sein und im Idealfall in seperaten Files liegen. Dies erleichtert die Trennung von Inhalt (HTML), Optik (CSS) und Interaktion (JavaScript).

Unobtrusive JavaScript

Unter **Unobtrusive JavaScript** versteht man die Trennung von HTML, CSS und JavaScript. Im Idealfall sind diese 3 Ebenen in jeweils eigenen Files.

Damit ein externes JavaScript auf HTML-Elemente zugreifen kann, muss für jedes gewünschte HTML-Element das `id` Attribut gesetzt sein (der Attributwert muss eindeutig sein).

3-Schritt Methode

```
<h1 id="klickMich">klick mich</h1>
Event-Ergebnis: <b id="aenderMich"></b>

<script>
   /*
   event handler: Die Funktion aktion wird ausgeführt, wenn beim ELement b (mit der ID klickMich),
   das Event onclick ausgelöst wird. */
   var aktion = function(){
       document.getElementById('aenderMich').innerHTML = 'Inhalt vom Element b geändert'
   }


   /*
   event definition: beim html-Element (mit der id klickMich) das Event oncklick definieren.
   Wenn das Event onclick eintritt, soll die Funktion aktion aufgerufen werden.
   */
   var eventDefinition = function(n){
       document.getElementById('klickMich').onclick = aktion;
   }

   // nachdem der User-Agent die Seite komplett geladen hat, die Funktion eventDefinition aufrufen
   window.onload = eventDefinition;

</script>
```

2-Schritt Methode

```
<h1 id="klickMich">klick mich</h1>
Event-Ergebnis: <b id="aenderMich"></b>

<script>
   /*
   event handler: Die Funktion aktion wird ausgeführt, wenn beim ELement b (mit der ID klickMich),
   das Event onclick ausgelöst wird. */
   var aktion = function(){
       document.getElementById('aenderMich').innerHTML = 'Inhalt vom Element b geändert'
   }


   /*
   event definition: beim html-Element (mit der id klickMich) das Event oncklick definieren.
   Wenn das Event onclick eintritt, soll die Funktion aktion aufgerufen werden.
   */
   window.onload  = function(){
       document.getElementById('klickMich').onclick = aktion;
      }
</script>
```

Progressive Enhancement

Unter Progressive Enhancement versteht man, dass neue Technologien eine Webseite nur erweitern (die Technologie darf nicht vorrausgesetzt werden).

JavaScript in HTML

Es gibt vier Möglichkeiten, JavaScript mit HTML zu verbinden:

1) Inline als Attributwert
2) Inline als Pseudo-URL (als Link oder als Adresse im Browser)
3) Embedded mit `<script>`
4) Embedded mit `<script>` als externes File

Inline JavaScript

Bestimmte HTML-Attribute sind Event-Attribute. Sie starten das JavaScript im Attributwert, wenn das Event erfüllt ist (siehe HTML-Attribut **Event Handler** Seite 156).

Als Beispiel das Event-Attribut `onclick` und `onsubmit`:

```
<a href="#" onclick="alert('Ade, schnöde Welt (triggered by onclick event)'); ">klick mich</a>
<form action="seite.php" method="post"  onsubmit="return validateForm();">
```

Inline-JavaScript sollte nur dann verwendet werden, wenn der Author keinen Zugriff auf das eingebettete oder externe JavaScript besitzt. Siehe auch Inline Style auf Seite 210.

HTML, CSS und JavaScript sollten getrennt sein und im Idealfall in seperaten Files liegen. Dies erleichtert die Trennung von Inhalt (HTML), Optik (CSS) und Interaktion (JavaScript). Siehe **Unobtrusive JavaScript** auf Seite 430.

Pseudo-URL

Als Link oder als Adresse im Browser.

Beispiel:

```
<!DOCTYPE html>
<html>
<head> <title>Hello World</title> </head>
<body>
    <a href="javascript:alert('Ade, schnöde Welt');">klick mich</a>
</body>
</html>
```

Embedded JavaScript mit <script>

Mit dem Element `script` wird JavaScript-Sourcecode in ein HTML-File eingebettet. Der Sourcecode kann sich auch in einem externen File befinden.

Optionale Attribute:

`async`	mit HTML5 eingeführtes Attribut. Attributwert: `async` der User-Agent parst weiter die Seite, und führt gleichzeitig das externe JavaScript aus
`charset`	Zeichensatz vom externen JavaScript-File. Auflistung aller RFC2978 Character-Sets siehe IANA.
`src`	URL vom externen File
`defer`	Attributwert: `defer` das externe JavaScript-File wird erst dann ausgeführt, wenn die Seite vollständig geparst worden ist.

`type`	bestimmt die Skriptsprache (seit HTML5 ist das Attribut optional).
	Zum Beispiel:
	`text/javascript` — seit HTML5 default
	`text/ecmascript`
	`application/javascript`

`language`	deprecated

externes File

Ist weder `async`, noch `defer` gesetzt, stoppt der User-Agent das Parsen der Seite und führt das externe JavaScript aus (nach der Ausführung, setzt der User-Agent das Parsen fort).

Für jedes externe File wird ein eigener http-Request benötigt.

XHMTL

Für XHTML muss der Inhalt mit CDATA markiert werden (um von einem XML-Parser ignoriert zu werden):

```
<script type="text/javascript">
<![CDATA[
       alert('Hello World');
]]>
</script>
```

<noscript>

Der Text im `noscript` Element, wird dann ausgeführt (bzw angezeigt), wenn JavaScript im User-Agent deaktiviert oder nicht implementiert ist.

Ab HTML5 darf `noscript` auch im `head` stehen:

```
<head>
 <meta charset="utf-8">
 <title>Hello World</title>
 <noscript>
    Dein Browser unterstützt JavaScript nicht.
    <meta http-equiv="refresh" content="5; URL=http://www.example.com/">
 </noscript>
</head>
```

Position im HTML-File

Der User-Agent lädt und parst eine HTML-Seite von oben nach unten. Je weiter oben sich das script Element befindet, umso früher wird auch das eingebettete JavaScript ausgeführt.

Wird die JavaScript Ausführung zu früh gestartet, besteht die Gefahr, dass noch nicht alle Elemente geladen sind und das JavaScript auf ein noch nicht vorhandenes Element zugreifen möchte.

Um die zu frühe Ausführung zu verhindern, wird entweder das `script` Element an das Ende der HTML-Seite gesetzt oder die Ausführung wird mit einer Bedingung eingeschränkt.

Zwei Events bieten sich als Start-Bedingung an:

`onload`	das Event tritt ein, wenn alle Dateien der Seite geladen sind
`DOMContentLoaded`	das Event tritt ein, wenn alle HTML-Elemente (Tags) als DOM vorhanden sind

Beispiel: Bei Klick auf den Link die Funktion `aufruf` starten und den Text vom Element `a` ändern.

Das `script` Element wird erst nach dem `a` Element positioniert:

```
<a href="#" id="klickMich">klick mich</a>
<script>
 function aufruf (){

    document.getElementById('geklickt').innerHTML ='geklickt';                //Text von a ändern
 }
 let elem = document.getElementById("klickMich");

 elem.addEventListener("click", aufruf, false);
</script>
```

PHP

Kapitel 11

PHP

Literal, Identifier, Variable, Token, Expression

Literale sind unveränderliche Werte, die für sich selbst stehen. **Variablen** stehen für änderbare Werte. **Konstanten** stehen für unveränderliche Werte. **Identifiers** sind Variablebezeichnungen. **Token** sind kleinstmögliche Einzelteile in der Syntax einer Programmiersprache.

Literal

Ein Literal ist ein fixer (konstanter) Wert, der für sich selbst steht (also keine Variable) und somit immutable (unveränderlich) ist.

String-Literal

Ein String-Literal steht in Anführungszeichen (doppelte oder einfache) und beinhaltet Zeichen:

- Ziffern, Buchstaben
- Control Characters (siehe **ASCII C0** Seite 225 und **ASCII C1** Seite 227)

Beispiel für ein String-Literal: `"bin ein literal"`

Multi-Line

Ein **Line Continuation Character (siehe Backslash** in JavaScript, Seite 286) ist nicht notwendig:

```
print ("bin
 ein
 text");
```

Ergibt als Output:

```
bin ein text
```

Escape Sequence (Fluchtsequenz)

Damit bestimmte Zeichen in einem String-Literal nicht interpretiert werden, müssen sie escaped werden. Dies geschieht, indem der **Escape-Character** Backslash (\) vorangestellt wird. Der Escape-Character wird auf deutsch **Fluchtsymbol** genannt. Ein escaped Backslash (\\) ist ein **Literal Backslash** (der Backslash steht für sich selbst).
Siehe auch **ASCII C0** auf Seite 226, und **ASCII C1 auf** Seite 227

Als Beispiel ein String der in doppelten Anführungzeichen steht:

```
"Er ist cool."
```

Kommt ein Anführungszeichen im String vor, muss dieses escaped werden:

```
"Er ist \"cool\"."
```

Ergibt als Output:

```
Er ist cool.
```

Control Character	Beschreibung	ASCII-Wert	
\\	literal backslash		
\a	alert (bell)	07	BEL
\b	backspace	08	BS
\f	formfeed	12	FF
\n	newline (line feed)	10	LF
\r	carriage return	13	CR
\h	horizontal tab	09	HT
\v	vertical tab	11	VT
\x*nn*	hexadecimal ascii value nn		
\$	dollar sign		
\{	left brace	123	
\}	right brace	125	
\[	left bracket	91	
\]	right bracket	93	

Anführungszeichen

Zwei Arten von Anführungszeichen sind möglich:

" doppeltes Anführungszeichen (Double Quotes) `print ("x ist 3. ");`

' einfaches Anführungszeichen (Single Quotes) `print ('x ist 3.');`

Escape-Sequenzen und Variablen werden bei doppelten Anführungszeichen durch ihren Wert ersetzt:

```
$foo = 6;

echo 'foo ist $foo \n<br>';
echo "foo ist \n $foo ";
```

Ergibt als Output:

```
foo ist $foo \n
foo ist 6
```

Number Literal

Ein **Number Literal** ist eine Zahl. Siehe Datentyp `integer` und `double` (Datentypen Seite).

Beispiel für ein Number-Literal: `123.456`

```
print gettype (3);          // integer

print gettype (3.3);        // double
```

Boolean Literal

Ein **Boolean Literal** hat nur 2 mögliche Werte: `true` und `false`. Siehe auch Datentyp `boolean` auf Seite .

Beispiel:

```
$foo=true;
print "$foo ".gettype($foo);              // 1 boolean

$foo=false;
print "\n$foo ".gettype ($foo);           //  boolean
```

Exponent Literal

Der Buchstabe e ist der Exponent-Literal. Er verschiebt das Komma in einer Zahl nach rechts oder nach links.

Die Zahl nach e bestimmt die Anzahl der Kommastellen.

Beispiel1:

```
$y = 12.34e-1 ;             // Komma um eine Stelle nach links

print $y;
```

Ergibt als Output:

```
1.234
```

Beispiel2:

```
$y = 12.34e+1 ;             // Komma um eine Stelle nach rechts

print $y;
```

Ergibt als Output:

```
123.4
```

Variable

Variablen beinhalten Werte ("*Variablen stehen für einen Wert*"). Der Variablename wird **Identifier** (Seite 438) genannt.

Die Art des Wertes hängt vom Datentyp (Seite 448) ab.

Ein Datentyp, der (gleichzeitig) nur einen Wert enthalten kann, wird Skalar (Seite 448) genannt.

Gewöhnliche Variablen beginnen in PHP mit einem Dollarzeichen.

Kein führendes Dollarzeichen haben Namen von:

1) Konstanten
2) Klassen
3) Methoden bzw Funktionen

Variablen (inklusive Konstanten) und Built-in Funktionen sind case-sensitive:

`$foo` und `$Foo` sind 2 unterschiedliche Variablen.

Variable Inspecting (Debug)

Zwei Funktionen zeigen den Inhalt von Variablen an:

`print_r()`		Seite
`var_dump()`	zeigt neben dem Inhalt, auch den Datentyp und die Größe an	Seite

Identifiers

Identifier Names sind Bezeichnungen von Variablen und Funktionen.

1. Namen dürfen **beginnen** mit:

   Buchstaben (a-z, A-Z und ASCII 127 bis ASCII 255)

   Unterstrich (siehe Seite 229)
2. Namen dürfen Ziffern beinhalten.
3. Namen düfen **kein reserviertes Wort** sein (zb `$this`).
4. Namen dürfen `kein` Whitespace beinhalten.

Konstanten haben kein führendes Dollarzeichen und werden meist **großgeschrieben** (alle in der Variable vorkommenden Buchstaben).
Klassen haben kein führendes Dollarzeichen und beginnen meist mit einem Großbuchstaben.
Funktionen haben kein führendes Dollarzeichen.

Case Sensitive

Variablen (inklusive Konstanten) und Built-in Funktionen sind case-sensitive: `$foo` und `$Foo` sind 2 unterschiedliche Variablen.

Case InSensitive

Eigene Funktionsnamen sind Case Insensitive.

```
function abc () {echo "aye";}
ABC();
```

Ergibt als Output:

```
aye
```

Camel case

Bei **Camel Case** werden die einzelnen Wörter (einer Variablebezeichnung) ohne Trennzeichen zusammengeschrieben. Der erste Buchstabe eines Wortes wird großgeschrieben (dadurch ergeben sich optisch Kamelhöcker).
Beispiele:

```
$currentCounterFlag=0; $whitePlayerStatus=2;
```

Snake case

Bei **Snake Case** werden die einzelnen Wörter (einer Variablebezeichnung) mit dem Unterstrich getrennt (der Unterstrich erinnert optisch an eine Schlange).
Beispiel:

```
$current_counter_flag=0; $white_player_status=2;
```

Dynamisch erzeugter Identifier

Der Identifier einer Variable kann auch dynamisch mit einem String erzeugt werden.

Beispiel:

```
$temp="foo";
$$temp=12.34;

echo $foo;
```

Erzeugt als Output:

```
12.34
```

Konstante

Konstanten sind Variablen, die ihren Wert nicht ändern können (Fixwerte).
Sie besitzen kein führendes Dollarzeichen und können bis auf Klassen jeden Datentyp besitzen (ab PHP7 auch `array`).
Eine Konstante wird meist durchgehend großgeschrieben.

define()

Mit der Funktion `define()` wird eine Konstante angelegt.

Syntax:

```
define ( Identifier , Wert [, Case-Insensitive  ] )
```

Der erste Parameter ist ein String, der den Identifier enthält.
Der zweite Parameter ist der Wert. Bis auf Objekte ist jeder Datentyp möglich (ab PHP7 auch array).
Der optionale dritte Parameter ist ein Boolean. Bei `false` (default) ist der Identifier case-sensivite.

Wenn die Konstante erzeugt werden kann, ist der Rückgabewert `true`.

Beispiel1a: `SPRICH` ist eine Konstante mit dem String 'wuff' als Wert (case-sensitive)

```
define('SPRICH','wuff',0);

echo SPRICH;
```

Ergibt als Output:

```
wuff
```

Beispiel1b: `Sprich` ist eine Konstante mit dem String 'wuff' als Wert (case-insensitive)

```
define('Sprich','wuff',1);

echo Sprich;
```

Ergibt als Output:

```
wuff
```

Beispiel2: `Sprich` ist eine Konstante mit den Array-Werten 5, 6 und 7 (case-insensitive)

```
define('SPRICH', array(5,6,7), 1);

print_r (Sprich);                        // Groß-Kleinschreibung wird ignoriert
```

Ergibt als Output:

```
Array ( [0] => 5    [1] => 6    [2] => 7 )
```

constant()

Die Funktion `constant()` gibt den Wert einer Konstante zurück.

Der Übergabewert ist der Identifier (String) der Konstante. Ist die Konstante unbekannt, ist `null` der Rückgabewert.

Syntax:

```
constant ( Identifier )
```

Beispiel:

```
define ('foo', 666 , 1);

echo constant('foo');
echo "<br>" . foo;
```

Ergibt als Output:

```
666
666
```

Built-in Konstante

PHP-Version

PHP_VERSION	string	PHP Version. Notation:	major.minor.release[extra]
PHP_MAJOR_VERSION	integer	PHP Major-Version	9 in v9.12.45
PHP_MINOR_VERSION	integer	PHP Minor-Version	12 in v9.12.45
PHP_RELEASE_VERSION	integer	PHP Release-Version	45 in v9.12.45
PHP_VERSION_ID	integer	PHP Version	91245 für v9.12.45
PHP_EXTRA_VERSION	string	PHP extra-Version	-extra in v9.12.45-extra

Mathematik

M_PI		3.14159265358979323846	Pi
M_E		2.7182818284590452354	e
M_LOG2E		1.4426950408889634074	log_2 e
M_LOG10E		0.43429448190325182765	log_10 e
M_LN2		0.69314718055994530942	log_e 2
M_LN10		2.30258509299404568402	log_e 10
M_PI_2		1.57079632679489661923	pi/2
M_PI_4		0.78539816339744830962	pi/4
M_1_PI		0.31830988618379067154	1/pi
M_2_PI		0.63661977236758134308	2/pi
M_SQRTPI		1.77245385090551602729	sqrt(pi)
M_2_SQRTPI		1.12837916709551257390	2/sqrt(pi)
M_SQRT2		1.41421356237309504880	sqrt(2)
M_SQRT3		1.73205080756887729352	sqrt(3)
M_SQRT1_2		0.70710678118654752440	1/sqrt(2)
PHP_ROUND_HALF_UP		Round halves up	
PHP_ROUND_HALF_DOWN		Round halves down	
PHP_ROUND_HALF_EVEN		Round halves to even numbers	
PHP_ROUND_HALF_ODD		Round halves to odd numbers	
NAN	float	NAN	Not A Number
INF	float	INF	The infinite

Error, Warning, Notice

Alle Konstanten sind vom Datentyp `integer`. Siehe php.net

Die Zahlen werden für ein Bitmask verwendet (um zu bestimmen, welche Fehler angezeigt werden sollen).

`E_ERROR`	1	Fatal Run-Time Error (Skript wird beendet)
`E_WARNING`	2	Run-Time Warning
`E_PARSE`	4	Compiler Parse Error
`E_NOTICE`	8	Run-Time Hinweis
`E_CORE_ERROR`	16	
`E_CORE_WARNING`	32	
`E_COMPILE_ERROR`	64	Compiler-Error der Zend Scripting Engine
`E_COMPILE_WARNING`	128	Compiler-Warnung der Zend Scripting Engine
`E_USER_ERROR`	256	wird durch die Funktion `trigger_error()` erzeugt
`E_USER_WARNING`	512	wird durch die Funktion `trigger_error()` erzeugt
`E_USER_NOTICE`	1024	wird durch die Funktion `trigger_error()` erzeugt
`E_STRICT`	2048	Kompatibilitätshinweise
`E_RECOVERABLE_ERROR`	4096	wenn es keinen user defined handle gibt, wird ein `E_ERROR` daraus
`E_DEPRECATED`	8192	Hinweis auf deprecated Code
`E_USER_DEPRECATED`	16384	wird durch die Funktion `trigger_error()` erzeugt
`E_ALL`	32767	

Integer

`PHP_INT_MAX`	integer	Integer-Maximalwert (siehe Integer Seite 449)
`PHP_INT_MIN`	integer	Integer-Minimumwert (siehe Integer Seite 449)
`PHP_INT_SIZE`	integer	Integer-Größe in Bytes

Sonstiges

PHP_OS	string	Operating System
PHP_ZTS	integer	Zend Thread Safety (ist 1, wenn eingeschaltet) --enable-maintainer-zts
PHP_DEBUG	integer	eine PEAR Library (0, wenn nicht vorhanden)
PHP_MAXPATHLEN	integer	maximale Stringlänge von Pfad und Filename
PHP_SAPI	string	die Art des PHP-Interface (php.net) Bei Command-Line-Interface ist der Wert `cli` Siehe auch `php_sapi_name()`
PHP_EOL	string	End-Of-Line Character
DEFAULT_INCLUDE_PATH	string	
PEAR_INSTALL_DIR	string	
PEAR_EXTENSION_DIR	string	
PHP_EXTENSION_DIR	string	
PHP_PREFIX	string	The value "--prefix" was set to at configure.
PHP_BINDIR	string	Specifies where the binaries were installed into.
PHP_BINARY	string	Specifies the PHP binary path during script execution.
PHP_MANDIR	string	Specifies where the manpages were installed into.
PHP_LIBDIR	string	
PHP_DATADIR	string	
PHP_SYSCONFDIR	string	
PHP_LOCALSTATEDIR	string	
PHP_CONFIG_FILE_PATH	string	
PHP_CONFIG_FILE_SCAN_DIR	string	
PHP_SHLIB_SUFFIX	string	The build-platform's shared library suffix, such as "so" (most Unixes) or "dll"
PHP_FD_SETSIZE	string	The maximum number of file descriptors for select system calls

Token

Token sind kleinstmögliche Einzelteile in der Syntax einer Programmiersprache:

7) Keyword
   Beispiel: `while`
8) Identifier
   Beispiel: `binKlassenname`
9) String-Literal
   Beispiel: `"bin text"`
10) Number-Literal:
   Beispiel: `123.456`
11) Seperator Character `.,;(){}[]`
   Beispiel: `{`
12) Operator
   Beispiel: `+`

Expression (Ausdruck)

Eine Expression erzeugt einen Wert. Kann eine einzelne Variable oder ein einzelnes Literal sein. Oder eine Kombination aus Literalen, Operatoren, Variablen und Rückgabewerten von Funktionen sein.

Für Expressions bei **bedingter Flusskontrolle** siehe Seite . Für Datentyp `boolean` siehte Seite .

Beispiele:

```
9
"bin ein text"
true
3+4
5*$x
meineFunktion()-7+$x
```

Statement

Ein Statement ist eine Anweisung, die eine Aktion durchführt.

Als Beispiel eine Zuweisung: `$foo = 9 +1`
`9+1` ist eine Expression (als Summe ergibt die Expression den Wert `11`, als Boolean den Wert `true`). Mit dem Zuweisungsoperator (Gleichheitszeichen) wird der Variable der Wert 11 zugewiesen.

Semikolon

Bei mehr als einem Statement pro Textzeile, müssen Statements durch Semikolons getrennt werden.

Beispiel: `$foo1 = 9+1; $foo2= 7+1;`

Das Semikolon am Ende einer Zeile ist zwingend notwending (im Gegensatz zu JavaScript).
Vor dem schließendem PHP-Tag ist das Semikolon optional.
Beispiele:

```
<?php
$foo = 9 +1;
if ($foo) {$foo2=1; $foo3=2;}
echo "ende"
?>
```

Block Statements (Compound Statements)

Mehrere Statements werden mit geschweiften Klammern zu einem Block (**Compound Statements**) zusammengefasst.

Sie werden bei der Flusskontrolle verwendet (siehe Seite).

Beispiel:

```
<?php
 $foo = 9 +1;
 if ($foo)
    {
       $foo2=1;
       $foo3=2;                                    // ; (Semikolon) auch bei Block-Ende verpflichtend
    }
 echo "ende"
?>
```

Whitespace

Leerraum spielt bei PHP fast keine Rolle (Whitespace-Insensitive). Siehe auch Whitespace Seite 237.

Ausnahmen:

- Kein Leerraum zwischen **<?** und **php** (PHP-Tag)
- Keine Trennung von Keywords und Identifier. Zum Beispiel: `$haus nummer`
- Kein Leerraum bei **Heredoc-** und **Nowdoc**-Closing Identifier.

Beispiel 1a:

```
function
 sprich()
    {
       echo "wuff";
    }

sprich();
```

Ergibt als Output:

```
 wuff
```

Beispiel 1b: bis auf Whitespace mit Beispiel 1a ident

```
function sprich() {  echo "wuff";  }

sprich();
```

Ergibt als Output:

```
 wuff
```

Source Code Remark

Mit // und # wird ein einzeiliger Kommentar im Source Code hinzugefügt, mit /* und */ ein mehrzeiliger.

#	Remark für eine einzelne Zeile
//	Remark für eine einzelne Zeile
/* */	Remark, der über mehrere Zeilen gehen kann

single line comment

Ein // (2 forward slashes) und ein # ermöglicht einen einzeiligen Remark (Comment) im SourceCode.

Shell-style Comment #

C++ Comment //

Beispiel:

```
$foo++;    // erhöhe die Variable um 1
$foo--;    # senke die Variable um -1
```

multi-line comment

Ein /* und */ ermöglicht eine mehrzeilige Quelltext-Bemerkung.

Keine Verschachtelung möglich.

C Comment /* */

Beispiel:

```
/* bin ein
mehrzeiliger Remark */
```

Commenting Out

Das Umwandeln von Statements zu Remarks, wird **Commenting Out** genannt:

Beispiel:

```
// $foo=$foo+1;
$foo++;
```

PHP Tag

Text ausserhalb des PHP-Tags wird vom Compiler als HTML betrachtet und nicht interpretiert.

Der PHP-Tag beginnt mit **<?php** und endet mit **?>**

Es sind beliebig viele PHP-Tags in einem File möglich.

Beispiel:

```
<html>
   <?php echo "Fehlt es mir etwa an Verrückten?"; ?>
</html>
```

Ergibt als Output:

```
Fehlt es mir etwa an Verrückten?
```

Kurzform (Short Delimiter)

Die Kurzform kann in der `php.ini` explizit ein- und ausgeschaltet werden (default ist false).

Der Tag beginnt mit **<?** und endet mit **?>**

Beispiel:

```
<html>
  <? echo "Fehlt es mir etwa an Verrückten?"; ?>
</html>
```

Ergibt als Output:

```
Fehlt es mir etwa an Verrückten?
```

echo Tag

Mit dem echo Tag wird eine Expression geprinted.

Syntax:

```
<?= Expression ?>
```

Beispiel:

```
<html>

<?php $foo="Fehlt es mir etwa an Verrückten," ?>

<hr>

<?=$foo?>

</html>
```

Ergibt als Output:

```
Fehlt es mir etwa an Verrückten?
```

Deprecated Tags

Deprecated ist der ASP-Tag **<%** und der Script-Tag:

```
<% echo " Fehlt es mir etwa an Verrückten?"; %>

<script language="php">
 echo " Fehlt es mir etwa an Verrückten?";
</script>
```

Datentypen (Data Types)

PHP ist ein eine **Loosely (weakly) Typed Language.** Der Datentyp ist kontextabhängig und wird bei Bedarf automatisch geändert (**Dynamically Typed** Language).

Für die Typkonvertierung (**Type Casting** Seite 462) gibt es 5 Funktionen und 6 Operatoren.

Ein Datentyp, der (gleichzeitig) nur einen Wert enthalten kann, wird **Skalar** (Seite 448) genannt.
Datentypen, die mehrere Werte gleichzeitig enthalten können, werden **Compound Data Types** genannt (Seite 456).

Es gibt **6 Primitive Values** (Skalare):

1. int
2. double
3. string
4. boolean

Zwei **Special Types** Skalare:

5. null
6. resource

2 Compound Data Types (auch **Collection Types** genannt):

1) array
2) object

Skalar (Primitive Value)

Skalare sind Datentypen, die (gleichzeitig) nur einen Wert beinhalten können (auch **Single Value** genannt).
Die Wertzuweisung erfolgt bei einem Skalar ***by Value*** (siehe Seite).
PHP kennt 5 Arten von Skalaren:

Integer	`int`	Signed Integer	Ganzzahl
Float	`double`	Signed Floating Point Number	Fließkommazahl
		Neben Fließkommazahlen, gibt es auch diese 2 Werte:	
		`NAN` Not A Number	
		`INF` Infinity	
Binary	`string`	Binary Data	Zeichenkette bzw binäre Daten
Boolean	`boolean`	true false	
Null	`NULL`	NULL	
Resource	`resource`	resource	

Integer (Ganzzahl)

PHP kennt nur **Signed Integers** (Ganzzahlen mit Vorzeichen). Überprüfung auf Integer mit der Funktion `is_int()`.

Konvertierung mit dem Operator `(int)` und den Funktionen `intval()` und `settype()`

Größe

Bei 32bit ist der Wertebereich:	± 2.147.483.648	(±2 Milliarden)
Bei 64bit ist der Wertebereich:	± 9.223.372.036.854.775.808	(±9 Trillionen)

Die Konstante `PHP_INT_SIZE` ist die Integer-Größe in Bytes.

Die Konstante `PHP_INT_MAX` ist der Maximalwert, die Kontante `PHP_INT_MIN` der kleinstmögliche Wert.

Beispiel:

```
print PHP_INT_SIZE;
print "<br>Max: ". PHP_INT_MAX;
print "<br>Min: ". PHP_INT_MIN;
```

Ergibt als Output:

```
4
Max: 2147483647
Min: -2147483648
```

Zahlenbasis

Integer-Werte können dezimal (Zahlenbasis 10), hexadezimal (Zahlenbasis 16), oktal (Zahlenbasis 8) oder binär sein:

```
echo 255;                    // dezimal
echo "\n" . 0xff;            // hexadezimal beginng mit 0x
echo "\n" . 0377;            // octal beginnt mit 0
echo "\n" . 0b11111111;      // binär beginnt mit 0b
```

Ergibt als Output:

```
255
255
255
255
```

Alle Werte werden beim Output (`print`, `echo`) in die Zahlenbasis 10 konvertiert.

is_int()

Die Funktion `is_int()` gibt `true` zurück, wenn der Übergabewert vom Datentyp `integer` ist.

Alias: `is_integer()`

Beispiel:

```
$foo=6;

echo is_int($foo);
echo "<br>" . gettype($foo);
```

Ergibt als Output:

```
1
integer
```

intval()

Die Funktion `intval()` konvertiert den Übergabewert in einen `int` (Integer).

Der optionale 2. Parameter ist die Zahlenbasis.

Der zu konvertierende Datentyp ist meist `string` und `boolean`.

Syntax:

```
intval ( Expression [, Basis ] )
```

Ein leeres Array ergibt 0, ein nicht leeres Array ergibt 1. **`NULL`** ergibt 0.

Ein Objekt ergibt `1` und ein Notice, dass eine Konvertierung nicht möglich ist:

```
Notice: Object of class could not be converted to int
```

Beispiel:

```
$foo = "123a45";                    // string

echo intval ($foo,10);              // alles nach dem 1. nicht nummerischen Zeichen wird weggeworfen

$foo = false;                       // boolean
echo intval ($foo,10);              // aus false wird 0

$foo = true;                        // boolean
echo intval ($foo,10);              // aus true wird 1

$foo = 2.9;                         // float
 echo intval ($foo,10);             // aus 2.9 wird 2
```

Weitere Beispiele siehe `intval()` auf Seite 462.

(int)

Der Operator `(int)` konvertiert den Operanden in einen Integer.

Beispiel:

```
$foo = "123a45";                // string

echo (int) $foo;                // alles nach dem 1. nicht nummerischen Zeichen wird weggeworfen

$foo = false;                   // boolean
echo (int) $foo;                // aus false wird 0

$foo = true;                        // boolean
echo (int) $foo;                    // aus true wird 1

$foo = 2.9;                         // float
 echo (int) $foo;                   // aus 2.9 wird 2
```

Weitere Beispiele siehe `(int)` auf Seite 471.

Floating Point Number (Fließkommazahl)

Der Datentyp `double` wird in PHP auch `float` genannt.

Die Größe ist entweder 32Bit oder 64Bit (siehe auch IEEE 754).

Für die Konvertierung gibt es die Funktionen `floatval()`, `settype()` und den Operator `(float)`.

NAN (Not A Number)

Der Wert `NaN` steht für **Not A Number** und ist vom Datentyp `double`. Überprüfung auf Wert `NAN` siehe Seite .

Beispiel:

```
$foo=0/0;                                   // Warning: Division by zero
print($foo);
print(gettype ($foo) );
```

Ergibt als Output:

```
NAN
double
```

INF (Infinity)

Der Wert `INF` steht für unendlich und ist vom Datentyp `double`.

Beispiel 1:

```
$foo1=pow(10,1000);
$foo2=log(0);
print($foo1 . ' = ' .  gettype( $foo1) . "<br>");
print($foo2 . ' = ' . gettype ( $foo2));
```

Ergibt als Output:

```
INF = double
-INF = double
```

Beispiel 2: Der Wert `INF` minus `INF` ergibt `NaN`.

```
$foo = INF - INF;

print($foo . ' = ' .  gettype( $foo) . "<br>");
```

Ergibt als Output:

```
NAN = double
```

Beispiel 3: `INF` ergibt als Boolean den Wert `true`

```
$foo=INF;
if ($foo) print($foo . ' ist true. ' . gettype($foo))
```

Ergibt als Output:

```
foo ist true. double
```

(float)

Der Operator `(float)` konvertiert den Operanden in eine Fließkommazahl.

Beispiel 1: String to float

```
$foo="123.45a67";                    // alles nach dem 1. nicht nummerischen Zeichen (und . ) wird weggeworfen

echo (float) $foo;
```

Ergibt als Output:

```
123.45
```

Beispiel 2: Boolean to float

```
$foo=false;                          // aus false wird 0
echo (float) $foo;

$foo=true;                           // aus true wird 1
echo (float) $foo;
```

Ergibt als Output:

```
01
```

floatval()

Die Funktion `floatval()` konvertiert den Übergabewert in einen `double` (aka `float`).

Der zu konvertierende Datentyp ist meist `string` und `boolean`.

Syntax:

```
floatval ( Expression )
```

Ein leeres Array ergibt 0, ein nicht leeres Array ergibt 1.

`NULL` ergibt 0.

Ein Objekt ergibt 1 und ein Notice, dass eine Konvertierung nicht möglich ist:

```
Notice: Object of class could not be converted to float
```

Beispiel 1: String to float

```
$foo="123.45a67";                    // alles nach dem 1. nicht nummerischen Zeichen (und . ) wird weggeworfen

echo floatval ($foo);
```

Ergibt als Output:

```
123.45
```

Beispiel 2: Boolean to float

```
$foo=false;                          // aus false wird 0
echo floatval ($foo);

$foo=true;                           // aus true wird 1
echo floatval ($foo);
```

Ergibt als Output:

```
01
```

Output & Compare von »Floating Point Number«

IEEE 754 ergibt Unschärfen, die vor allem bei Überprüfung auf Gleichheit zu Problemen führt.
Zum Beispiel ergibt 0.1 + 0.2 beim Vergleich nicht 0.3 (**nur beim Output ist die Summe 0.3**).

Beispiel 1a: `0.1 + 0.2` ergibt nicht `0.3`

```
if (0.1 + 0.2 != 0.3)
   {
      print ('0.1 + 0.2  ist ungleich  0.3< br>');
   }
print (0.1 + 0.2);              //    0.30000000000000004      wird als  0.3  angezeigt
```

Ergibt als Output:

```
0.1 + 0.2  ist ungleich  0.3
0.3
```

Beispiel 1b: für den Vergleich die Fließkommazahlen in Integer umwandeln

```
if (intval(0.1 * 10) + intval(0.2 * 10) == 3)
   {
      print ('0.1 + 0.2  ist  0.3 <br>');
   }
```

Ergibt als Output:

```
0.1 + 0.2  ist  0.3
0.2
```

Beispiel 1c: mit `bccomp()` vergleichen (wenn identisch, ist der Rückgabewert `false`)

```
if ( ! bccomp(0.1 + 0.2, 0.3) )
{
   print ('0.1 + 0.2  ist  0.3 <br>');
}
```

Ergibt als Output:

```
0.1 + 0.2  ist  0.3
```

boolean

Der Datentyp `boolean` hat nur 2 mögliche Werte:

```
true
false
```

Für die Datentypkonvertierung gibt es die Funktionen `boolval()`, `settype()` und den Operator `(bool)`

Diese Werte ergeben `false`:

`NULL`	Datentyp `NULL`
`0`	Datentyp `int`
`-0`	Datentyp `int`
`""`	Datentyp `string`
`''`	Datentyp `string`
`false`	

Diese Werte ergeben `true`:

`INF`	Datentyp `double`
`NAN`	Datentyp `double`
`"false"`	
`"0"`	
`"NAN"`	
`8.3`	
`true`	

Beispiel:

```
$foo=INF;
if ($foo) print($foo . ' ist true. ' . gettype($foo))
```

Ergibt als Output:

```
foo ist true. number
```

truthy, falsy

Wenn ein Wert (bzw Expression) zu `true` konvertiert werden kann, wird er **truthy** (siehe mozilla.org) genannt. Wenn ein Wert zu `false` konvertiert werden kann, wird er **falsy** genannt (siehe mozilla.org).

Siehe auch Logische Operatoren (Seite) und die Methode `boolval()` Seite 465 .

boolval()

Die Funktion `boolval()` konvertiert den Übergabewert in einen `boolean`.

Der zu konvertierende Datentyp ist meist `string`, `float` und `int`.

Syntax:

```
boolval ( Epression )
```

Eine Zahl ungleich 0 ergibt `true` (0 ergibt `false`). Auch die Werte `NAN` und `INF` ergeben `true`

Ein leeres Array ergibt `false`, ein nicht leeres Array ergibt `true`.

Ein Objekt ergibt immer true.

`null` ergibt false

Beispiel 1: String to boolean

```
$foo1="123.45a67";                                  //   "123.45a67"        true
echo "foo1: " . boolval ($foo1);

$foo2="0";                                          //   "0"                false
echo "<br>foo2: " . boolval ($foo2);

$foo3="false";
echo "<br>foo3: " . boolval ($foo3);                //   "false"            true

$foo4="";
echo "<br>foo4: " . boolval ($foo4);                //   ""                 false
```

Ergibt als Output:

```
foo1: 1
foo2:
foo3: 1
foo4:
```

Beispiel 2: alle Zahlen ungleich 0 sind `true`. Auch die Werte `NAN` und `INF` ergeben `true`

```
$foo1=123.45;                                       // 123.45 ist true
echo "foo1: " . boolval ($foo1);

$foo2=0;                                            // 0 ist false
echo "<br>foo2: " . boolval ($foo2);

$foo3=NAN;
echo "<br>foo3: " . boolval ($foo3);                // NAN ist true

$foo4=INF;
echo "<br>foo4: " . boolval ($foo4);                // INF ist true
```

Ergibt als Output:

```
foo1: 1
foo2:
foo3: 1
foo4: 1
```

Weitere Beispiele siehe Seite 465.

Composite Data Types

Zu d

Wertzuweisung (Value Assignment)

Es gibt 2 Arten der Wertzuweisung:

1) by Value — Der Wert wird kopiert und der Variable zugewiesen
Default-Vorgang bei einem **Skalar**.

2) by Reference — nur der Zeiger (auf die Daten) wird der Variable zugewiesen
Default-Vorgang bei einem **Objekt**.

by Value

Der Wert wird kopiert und der Variable zugewiesen. Default bei Skalaren.

Zum Beispiel:

```
$foo1="abc";

$foo2=$foo1;

echo $foo2;
```

Ergibt als Output:

```
abc
```

Wird der Wert von `$foo1` geändert, bleibt der Wert von `$foo2` unberührt:

```
$foo1="abc";

$foo2=$foo1;

$foo1=NULL;                    // der Wert von $foo1 wird gelöscht

echo $foo2;                    // $foo2 ist weiterhin abc
```

Ergibt als Output:

```
abc
```

by Reference

Nicht der Wert, sondern der Zeiger auf den Wert im Heap wird zugewiesen. Default bei Objekten.

explizite Referenzierung

Das kann explizit bei der Zuweisung durch `=@` bestimmt werden. Wird eine der Variablen geändert, werden somit automatisch alle anderen Variablen(werte) geändert.

Erst wenn der letzte Zeiger auf den Heap gelöscht wird (bzw die letzte Variable, die ihn verwendet), wird auch der Wert im Heap gelöscht.

Syntax: *$variablename2* = &*$variablename1*

Beispiel1:

```
$foo1="abc";

$foo2= &$foo1;                 // nicht der Wert, sondern die Referenz auf $foo1 wird zugewiesen

$foo1="xyz";                   // der Wert (im Heap) wird auf xyz geändert
echo $foo2;                    // xyz
```

Ergibt als Output:

```
xyz
```

Beispiel2:

```
$foo1="abc";

$foo2= &$foo1;              // nicht der Wert, sondern die Referenz auf $foo1  wird zugewiesen

unset ($foo1);              // die Variable $foo1 wird gelöscht
                            // da die Referenz auf den Heap noch in der Variable $foo2 verwendet wird,
                            // werden nicht die eigentlichen Daten gelöscht

echo $foo2;                 // der Wert abc wird angezeigt
```

Ergibt als Output:

```
abc
```

Objekte

Objekte werden per Referenzierung zugewiesen (vor PHP 5.0 wurde das Objekt kopiert). Wird einer Variable eine Objektvariable zugewiesen, erhält eine Referenz (und nicht eine Kopie des bestehenden Objektes) .

Beispiel:

```
class hund
 {
    function __construct ($name,   $rasse)
       {
          $this -> name = $name;
          $this -> rasse = $rasse;
       }
 }


$bello=new hund("Bello","Dackel");

$klaeff = $bello;                       // $klaeff erhält die Referenz von $bello

$klaeff->rasse="Pudel";                 // rasse von $klaeff wird geändert
                                        // und automatisch auch die rasse von $bello

var_dump($bello);
```

Ergibt als Output:

```
object(hund)#1 (2) {
  ["name"]=>
  string(5) "Bello"
  ["rasse"]=>
  string(5) "Pudel"
}
```

Variable Inspecting (Debug)

Zwei Funktionen zeigen den Inhalt von Arrays und Objekten an:

`print_r()`	Rückgabewert ist der Inhalt der Variable	Seite 459
`var_dump()`	zeigt neben dem Inhalt, auch die Datentypen und Größen	Seite 460

print_r()

Die Funktion `print_r()` gibt den Inhalt einer Variable zurück. Im Gegensatz zu `print()`, kann es auch Arrays/Objekte darstellen und den Output (optional) in eine Variable schreiben.

Syntax:

```
print_r ( $variable [, $inhalt ] )
```

Wenn der optionale Parameter *$inhalt* angegeben wird, wird der Inhalt nicht geprintet, sondern in die Variable *$inhalt* kopiert.

Beispiel 1: Array

```
$foo = (array) [100,200,300] ;

print_r ($foo, false);
```

Ergibt als Output:

```
Array
(
    [0] => 100
    [1] => 200
    [2] => 300
)
```

Beispiel 2: Objekt

```
class hund
 {

    function __construct ($name, $rasse)
       {
          $this->name = $name;
          $this->rasse = $rasse;
       }

 }

$bello = new hund ("Bello", "Dackel");

print_r ($bello, false);
```

Ergibt als Output:

```
hund Object
(
    [name] => Bello
    [rasse] => Dackel
)
```

var_dump()

Die Funktion `var_dump()` printet die Datentypen, Werte und Stringlänge einer Variable. Anzahl der Parameter ist beliebig. Mit der Magic Methode `__debuginfo()` lässt sich die Darstellung auch selber bestimmen.

Syntax:

```
var_dump ( $variable [, $variable2, … ] )
```

Beispiel 1: Array

```
$foo = (array) [100,200,300] ;

var_dump ($foo);
```

Ergibt als Output:

```
array(3) {
  [0]=>
  int(100)
  [1]=>
  int(200)
  [2]=>
  int(300)
}
```

Beispiel 2: Objekt

```
class hund
 {
    function __construct ($name, $rasse)
       {
          $this->name = $name;
          $this->rasse = $rasse;
       }
 }

$bello = new hund ("Bello", "Dackel");

var_dump ($bello);
```

Ergibt als Output:

```
object(hund)#1 (2) {
  ["name"]=>
  string(5) "Bello"
  ["rasse"]=>
  string(6) "Dackel"
}
```

Beispiel 3: Boolean

```
$foo = false ;
var_dump ($foo);
```

Ergibt als Output:

```
bool(false)
```

Beispiel 4: NULL

```
$foo = NULL ;
var_dump ($foo);
```

Ergibt als Output:

```
NULL
```

Beispiel 5a: Objekt

```
class hund
 {
    private $hundeinfo="bin die info";

    function __construct ($name,   $rasse) {
          $this -> name = $name;
          $this -> rasse = $rasse;
       }

    function ichBin () {
          return "Ich bin ". $this->name. ", ein ". $this->rasse;
       }
 }

$bello=new hund("Bello","Dackel");

var_dump ($bello);
```

Ergibt als Output:

```
object(hund)#1 (3)
{
  ["hundeinfo":"hund":private]=>
  string(12) "bin die info"
  ["name"]=>
  string(5) "Bello"
  ["rasse"]=>
  string(6) "Dackel"
}
```

Beispiel 5b: Objekt mit __debuginfo()

```
class hund
 {
    private $hundeinfo="bin die info";

    function __construct ($name,   $rasse) {
          $this -> name = $name;
          $this -> rasse = $rasse;
       }

    function __debuginfo ()  {
             return ["Hundename" => $this->name,  "Rassenbezeichnung" => $this->rasse] ;
       }
    function ichBin () {
          return "Ich bin ". $this->name. ", ein ". $this->rasse;
       }
 }

$bello=new hund("Bello","Dackel");

var_dump ($bello);
```

Ergibt als Output:

```
object(hund)#1 (2)
{
  ["Hundename"]=>
  string(5) "Bello"
  ["Rassenbezeichnung"]=>
  string(6) "Dackel"
}
```

Type Casting

Es gibt 6 Operatoren und 5 Funktionen für das Konvertieren von Datentpyen.

Fünf Funktionen:

`intval()`	to integer	Seite 462	
`floatval()`	to float	Seite 463	alternative Schreibweise: `doubleval()`
`boolval()`	to boolean	Seite 465	
`strval()`	to string	Seite 467	
`settype()`		Seite 468	

Sechs Operatoren:

`(int)`	Seite 471	alternative Schreibweise: `(integer)`
`(float)`	Seite 472	alternative Schreibweise: `(double)`
`(boolean)`	Seite 473	
`(string)`	Seite 475	alternative Schreibweise: `(binary)`
`(array)`	Seite	
`(object)`	Seite	

Funktionen

Es gibt 5 Funktionen für das Konvertieren von Datentpyen (**Data Types** Seite 448).

intval()

Die Funktion `intval()` konvertiert den Übergabewert in einen `int` (Integer).

Der optionale 2. Parameter ist die Zahlenbasis.

Der zu konvertierende Datentyp ist meist `string` und `boolean`.

Syntax:

```
intval ( Expression [, Basis ] )
```

Ein leeres Array ergibt 0, ein nicht leeres Array ergibt 1.

`NULL` ergibt 0.

Ein Objekt ergibt `1` und ein Notice, dass eine Konvertierung nicht möglich ist:

```
Notice: Object of class could not be converted to int
```

Beispiel 1: String to Integer

```
$foo = "123a45";                    // alles nach dem 1. nicht nummerischen Zeichen wird weggeworfen

echo intval ($foo,10);
```

Ergibt als Output:

```
123
```

Beispiel 2: Boolean to Integer

```
$foo = false;                       // aus false wird 0
echo intval ($foo,10);

$foo = true;                        // aus true wird 1
echo intval ($foo,10);
```

Ergibt als Output:

```
01
```

Beispiel 3: leeres **Array** ergibt 0

```
$foo = array();
echo intval ($foo,10);
```

Ergibt als Output:

```
0
```

Beispiel 4: nicht leeres **Array** ergibt 1

```
$foo = array(2,4,6,7);
echo intval ($foo,10);
```

Ergibt als Output:

```
1
```

Beispiel 5: NULL ergibt 0

```
$foo = NULL;
echo intval ($foo,10);
```

Ergibt als Output:

```
0
```

Beispiel 6: ein Objekt ergibt immer 1 und ein Notice

```
class hunde {
       public $rasse = "Dackel";
    }

$bello = new hunde();

echo intval ($bello,10);
```

Ergibt als Output:

```
Notice: Object of class hunde could not be converted to int

1
```

floatval()

Die Funktion `floatval()` konvertiert den Übergabewert in einen `double` (aka `float`).

Der zu konvertierende Datentyp ist meist `string` und `boolean`.

Syntax:

```
floatval ( Expression )
```

Ein leeres Array ergibt 0, ein nicht leeres Array ergibt 1.

`NULL` ergibt 0.

Ein Objekt ergibt 1 und ein Notice, dass eine Konvertierung nicht möglich ist:

```
Notice: Object of class could not be converted to float
```

Beispiel 1: String to float

```
$foo="123.45a67";                    // alles nach dem 1. nicht nummerischen Zeichen (und . ) wird weggeworfen

echo floatval ($foo);
```

Ergibt als Output:

```
123.45
```

Beispiel 2: Boolean to float

```
$foo=false;                          // aus false wird 0
echo floatval ($foo);

$foo=true;                           // aus true wird 1
echo floatval ($foo);
```

Ergibt als Output:

```
01
```

Beispiel 3: leeres **Array** ergibt 0

```
$foo=array();
echo floatval ($foo);
```

Ergibt als Output:

```
0
```

Beispiel 4: nicht leeres **Array** ergibt 1

```
$foo=array(2,4,6,7);
echo floatval ($foo);
```

Ergibt als Output:

```
1
```

Beispiel 5: ein Objekt ergibt 1 und eine Notice

```
class hunde  {
       public $rasse = "Dackel";
    }

$bello = new hunde();

echo floatval ($bello);
```

Ergibt als Output:

```
Notice: Object of class hunde could not be converted to float

1
```

Beispiel 6: NULL ergibt 0

```
$foo=floatval(NULL);
echo $foo."<br>Datentyp: ".gettype($foo);
```

Ergibt als Output:

```
0
Datentyp: double
```

boolval()

Die Funktion `boolval()` konvertiert den Übergabewert in einen `boolean`.

Der zu konvertierende Datentyp ist meist `string`, `float` und `int`.

Syntax:

```
boolval ( Epression )
```

Eine Zahl ungleich 0 ergibt `true` (0 ergibt `false`). Auch die Werte `NAN` und `INF` ergeben `true`

Ein leeres Array ergibt `false`, ein nicht leeres Array ergibt `true`.

Ein Objekt ergibt immer true.

`null` ergibt false

Beispiel 1: String to boolean

```
$foo1="123.45a67";                                  //   "123.45a67"        true
echo "foo1: " . boolval ($foo1);

$foo2="0";                                          //   "0"                false
echo "<br>foo2: " . boolval ($foo2);

$foo3="false";
echo "<br>foo3: " . boolval ($foo3);                //   "false"            true

$foo4="";
echo "<br>foo4: " . boolval ($foo4);                //   ""                 false
```

Ergibt als Output:

```
foo1: 1
foo2:
foo3: 1
foo4:
```

Beispiel 2: leeres **Array** ergibt false

```
$foo=array();                                       // leeres Array ist      false
echo "foo: " . boolval ($foo);
```

Ergibt als Output:

```
foo:
```

Beispiel 3: nicht leeres **Array** ergibt true

```
$foo=array(2,4,6,7);                                // nicht leeres Array     true
echo "foo: " . boolval ($foo);
```

Ergibt als Output:

```
1
```

Beispiel 4: ein Objekt ergibt immer true (und kein zusätzliches Notice wie bei `intval` und `floatval`)

```
class hunde {
       public $rasse = "Dackel";
    }

$bello = new hunde();

echo "bello: " . boolval ($bello);
```

Ergibt als Output:

```
1
```

Beispiel 5: alle Zahlen ungleich 0 sind `true`. Auch die Werte `NAN` und `INF` ergeben `true`

```
$foo1=123.45;                                   // 123.45 ist true
echo "foo1: " . boolval ($foo1);

$foo2=0;                                        // 0 ist false
echo "<br>foo2: " . boolval ($foo2);

$foo3=NAN;
echo "<br>foo3: " . boolval ($foo3);            // NAN ist true

$foo4=INF;
echo "<br>foo4: " . boolval ($foo4);            // INF ist true
```

Ergibt als Output:

```
foo1: 1
foo2:
foo3: 1
foo4: 1
```

Beispiel 6: `null` ergibt `false`

```
$foo=array();                                   // leeres Array ist false
echo "foo: " . boolval ($foo);
```

Ergibt als Output:

```
foo:
```

strval()

Die Funktion `strval()` konvertiert den Übergabewert in einen `string`.

Der zu konvertierende Datentyp ist meist `int` und `double`.

Syntax:

```
strval ( Expression )
```

Objekte müssen für eine sinnvolle Konvertierung die **Magic Methode** `__toString()` besitzen.

Ohne `__toString()` erscheint der Fehler: `Catchable fatal error: Object could not be converted to string`

Beispiel 1: float to String

```
$foo=123.456;

echo strval ($foo);
```

Ergibt als Output:

```
123.456
```

Beispiel 2: Boolean to String

```
$foo1=false;                                    //aus false wird ein  Leerstring
echo "foo1: ". strval ($foo1);


$foo2=true;                                     //aus true wird 1
echo "<br>foo2: ". strval ($foo2);
```

Ergibt als Output:

```
foo1:

foo2: 1
```

Beispiel 3: für **Magic Methods** Seite 499 und für `__toString` Seite 501.

```
class hund
 {
    function __construct ($rasse,$name)
       {
           $this->rasse = $rasse;
           $this->name = $name;
       }

    function __toString()
       {
          return "Ich bin " . $this->name . ", ein ".$this->rasse;
       }

 }

$bello = new hund('Dackel','Bello');

echo strval($bello);
```

Erzeugt als Output:

```
Ich bin Bello, ein Dackel
```

settype()

Die Funktion `settype()` konvertiert eine Variable in einen bestimmten Datentyp.

Syntax:

```
settype ( $Identifier , "Datentyp" )
```

Datentyp ist ein String, der diese Werte haben kann:

"bool"	alternativ:	"boolean"
"int"	alternativ:	"integer"
"float"	alternativ:	"double"
"string"		
"array"		
"object"		
"null"		

Die Variable kann von jedem Datentyp sein.

Der Rückgabewert ist `true`, wenn die Konvertierung erfolgreich war (ansonsten `false`).

Beispiel 1: Boolean to String.

```
$foo1=false;
settype ($foo1,"string");
echo "foo1: $foo1";                          // foo1:


$foo2=true;
settype ($foo2,"string");
echo "<br>foo2: $foo2";                      // foo2: 1
```

Ergibt als Output:

```
foo1:
foo2: 1
```

Beispiel 2: Boolean to Integer.

```
$foo1=false;
settype ($foo1,"int");
echo "foo1: $foo1";                          // foo1: 0


$foo2=true;
settype ($foo2,"int");
echo "<br>foo2: $foo2";                      // foo2: 1
```

Ergibt als Output:

```
foo1: 0
foo2: 1
```

Beispiel 3: Double to Boolean.

```
$foo1=0.9;
settype ($foo1,"boolean");
echo "foo1: $foo1";             // foo1:

$foo2=INF;
settype ($foo2,"boolean");
echo "<br>foo2: $foo2";         // foo2: 1

$foo3=NAN;
settype ($foo3,"boolean");
echo "<br>foo3: $foo3";         // foo3: 1

$foo4=INF;
settype ($foo4,"boolean");
echo "<br>foo4: $foo4";         // foo4: 1
```

Ergibt als Output:

```
foo1: 1
foo2: 1
foo3: 1
foo4: 1
```

Beispiel 4: Integer to Boolean.

```
$foo1=0;
settype ($foo1,"boolean");
echo "foo1: $foo1";             // foo1:


$foo2=4;
settype ($foo2,"boolean");
echo "<br>foo2: $foo2";         // foo2: 1
```

Ergibt als Output:

```
foo1:
foo2: 1
```

Beispiel 5: Double to Integer.

```
$foo1=1.9;
settype ($foo1,"int");
echo "foo1: $foo1";               // foo1: 1


$foo2=2.3;
settype ($foo2,"int");
echo "<br>foo2: $foo2";           // foo2: 2
```

Ergibt als Output:

```
foo1:
foo2: 1
```

Beispiel 6: NULL to String, Integer and Boolean.

```
$foo1=NULL;
settype ($foo1,"string");
echo "foo1: $foo1";             // foo1: empty string


$foo2=NULL;
settype ($foo2,"int");
echo "<br>foo2: $foo2";         // foo2: 0


$foo3=NULL;
settype ($foo3,"boolean");
echo "<br>foo3: $foo3";         // foo3: false
```

Ergibt als Output:

```
foo1:
foo2: 1
foo3:
```

Beispiel 7: Object to String. Für **Magic Methods** Seite 499 und für `__toString` Seite 501.

```
class hund
 {
    function __construct ($rasse,$name)
       {
           $this->rasse = $rasse;
           $this->name = $name;
       }

    function __toString()
       {
          return "Ich bin " . $this->name . ", ein ".$this->rasse;
       }

 }

$bello = new hund('Dackel','Bello');

setype ($bello,"string");

echo $bello;
```

Erzeugt als Output:

```
Ich bin Bello, ein Dackel
```

Operatoren

Es gibt 6 Operatoren für das Konvertieren von Datentpyen (**Data Types** Seite 448).

(int)

Der Operator `(int)` konvertiert den Operanden in einen Integer.

Beispiel 1: String to Integer

```
$foo = (int) "123a45";                    // alles nach dem 1. nicht nummerischen Zeichen wird weggeworfen

echo $foo;
```

Ergibt als Output:

```
123
```

Beispiel 2: Boolean to Integer

```
$foo= (int) false;                        // aus false wird 0
echo $foo;

$foo = (int) true;                        // aus true wird 1
echo $foo;
```

Ergibt als Output:

```
01
```

Beispiel 3: leeres **Array** ergibt 0

```
$foo = (int) array();
echo $foo;
```

Ergibt als Output:

```
0
```

Beispiel 4: nicht leeres **Array** ergibt 1

```
$foo = (int) array (2,4,6,7);
echo $foo;
```

Ergibt als Output:

```
1
```

Beispiel 5: NULL ergibt 0

```
$foo = (int) NULL;
echo $foo;
```

Ergibt als Output:

```
0
```

Beispiel 6: ein Objekt ergibt immer 1 und ein Notice: Object could not be converted to int

```
class hunde {
       public $rasse = "Dackel";
    }

$bello = new hunde();

echo (int) $bello;
```

Ergibt als Output:

```
Notice: Object of class hunde could not be converted to int

1
```

(float)

Der Operator (float) konvertiert den Operanden in eine Fließkommazahl.

Beispiel 1: String to float

```
$foo="123.45a67";                    // alles nach dem 1. nicht nummerischen Zeichen (und . ) wird weggeworfen

echo (float) $foo;
```

Ergibt als Output:

```
 123.45
```

Beispiel 2: Boolean to float

```
$foo=false;                          // aus false wird 0
echo (float) $foo;

$foo=true;                           // aus true wird 1
echo (float) $foo;
```

Ergibt als Output:

```
 01
```

Beispiel 3: leeres **Array** ergibt 0

```
$foo=array();
echo (float) $foo;
```

Ergibt als Output:

```
0
```

Beispiel 4: nicht leeres **Array** ergibt 1

```
$foo=array(2,4,6,7);
echo (float) $foo;
```

Ergibt als Output:

```
1
```

Beispiel 5: ein Objekt ergibt immer 1 und eine Notice

```
class hunde {
       public $rasse = "Dackel";
    }

$bello = new hunde();

echo (float) $bello;
```

Ergibt als Output:

```
Notice: Object of class hunde could not be converted to float

1
```

Beispiel 6: NULL ergibt 0

```
$foo= (float) NULL;
echo $foo."<br>Datentyp: ".gettype($foo);
```

Ergibt als Output:

```
0
Datentyp: double
```

(boolean)

Der Operator (boolean) konvertiert den Operanden in eine Fließkommazahl.

Beispiel 1: String to boolean

```
$foo1="123.45a67";                                  //   "123.45a67"        true
echo "foo1: " . (boolean) $foo1;

$foo2="0";                                          //   "0"                false
echo "<br>foo2: " . (boolean) $foo2;

$foo3="false";
echo "<br>foo3: " . (boolean) $foo3;                //   "false"            true

$foo4="";
echo "<br>foo4: " . (boolean) $foo4;                //   ""                 false
```

Ergibt als Output:

```
foo1: 1
foo2:
foo3: 1
foo4:
```

Beispiel 2: leeres Array ergibt false

```
$foo=array();                                       // leeres Array ist      false
echo "foo: " . (boolean) $foo;
```

Ergibt als Output:

```
foo:
```

Beispiel 3: nicht leeres **Array** ergibt `true`

```
$foo=array(2,4,6,7);                                    // nicht leeres Array     true
echo "foo: " . (boolean) $foo;
```

Ergibt als Output:

```
1
```

Beispiel 4: ein Objekt ergibt immer `true` (und kein zusätzliches Notice wie bei `intval` und `floatval`)

```
class hunde {
       public $rasse = "Dackel";
    }

$bello = new hunde();

echo "bello: " . (boolean) $bello;
```

Ergibt als Output:

```
1
```

Beispiel 5: alle Zahlen ungleich 0 sind `true`. Auch die Werte `NAN` und `INF` ergeben `true`

```
$foo1=123.45;                                           // 123.45 ist true
echo "foo1: " . boolval ($foo1);

$foo2=0;                                                // 0 ist false
echo "<br>foo2: " . boolval ($foo2);

$foo3=NAN;
echo "<br>foo3: " . boolval ($foo3);                    // NAN ist true

$foo4=INF;
echo "<br>foo4: " . boolval ($foo4);                    // INF ist true
```

Ergibt als Output:

```
foo1: 1
foo2:
foo3: 1
foo4: 1
```

Beispiel 6: `null` ergibt `false`

```
$foo=array();                                           // leeres Array ist false
echo "foo: " . boolval ($foo);
```

Ergibt als Output:

```
foo:
```

(string)

Der Operator `(string)` konvertiert den Operanden in einen String. Alternative Schreibweise: `(binary)`

Beispiel 1: float to String

```
$foo=123.456;

echo (string) $foo;
```

Ergibt als Output:

```
 123.456
```

Beispiel 2: Boolean to String

```
$foo1=false;                                    //aus false wird ein  Leerstring
echo "foo1: ". (string) $foo1;


$foo2=true;                                     //aus true wird 1
echo "<br>foo2: ". (string) $foo2;
```

Ergibt als Output:

```
foo1:

foo2: 1
```

Beispiel 3: für **Magic Methods** Seite 499 und für `__toString` Seite 501.

```
class hund
 {
    function __construct ($rasse,$name)
       {
           $this->rasse = $rasse;
           $this->name = $name;
       }

    function __toString()
       {
          return "Ich bin " . $this->name . ", ein ".$this->rasse;
       }

 }

$bello = new hund('Dackel','Bello');

echo (string) $bello;
```

Erzeugt als Output:

```
Ich bin Bello, ein Dackel
```

(array)

Der Operator `(array)` konvertiert den Operanden in ein Array.

Beispiel 1: Array erzeugen

```
$foo = (array) [2,4,6,7];

print_r ($foo);
```

Ergibt als Output:

```
Array ( [0] => 2    [1] => 4    [2] => 6    [3] => 7 )
```

Beispiel 2: assoziatives Array erzeugen

```
$foo = (array) ["Name" => "Bello", "Rasse" => "Dackel"];

print_r ($foo);
```

Ergibt als Output:

```
Array ( [Name] => Bello     [Rasse] => Dackel )
```

Beispiel 3: Objekt zu Array konvertieren

```
class hund
 {
    function __construct($name,$rasse)
       {
          $this->name = $name;
          $this->rasse= $rasse;
       }
 }

$bello = new hund("Bello","Dackel");

$bello = (array) $bello;

print_r ($bello);
```

Ergibt als Output:

```
Array ( [name] => Bello  [rasse] => Dackel )
```

(object)

Der Operator `(object)` konvertiert den Operanden in ein Objekt.

Beispiel 1: Array in Objekt konvertieren

```
$foo = (array) [100,200,300,400];

$foo = (object) $foo;

var_dump($foo);
```

Ergibt als Output:

```
object(stdClass)#1 (4) { [0]=> int(100) [1]=> int(200) [2]=> int(300) [3]=> int(400) }
```

Beispiel 2: assoziatives Array in Objekt konvertieren

```
$foo = (array) ["Name" => "Bello", "Rasse" => "Dackel"];

$foo = (object) $foo;

var_dump($foo);
```

Ergibt als Output:

```
Array ( [Name] => Bello     [Rasse] => Dackel )
```

Operatoren

Type Casting

Es gibt 6 Operatoren für das Konvertieren von Datentpyen:

Präzedenz (Precedence)

Die Operator-Präzedens bestimmt die Priorität der Verarbeitungsreihenfolge. Die Präzedens `1` hat die geringste Priorität.

Im Beispiel 1 ergibt die Expression `9/3-2` den Wert 1:

```
$foo = 9 / 3 - 2
print ($foo)
```

Zuerst wird die Division durchgeführt (Präzedenz 14), danach die Subtraktion (Präzedenz 13).

Ergibt als Output:

```
1
```

21	Grouping		(…)
20	New Object	left-to-right	new …
	Klonen	left-to-right	clone …
19	Array Subscript		[…]
18	(int) Cast	left-to-right	(int) …
	(double) Cast	left-to-right	(double) …
	(string) Cast	left-to-right	(string) …
	(boolean) Cast	left-to-right	(boolean) …
	(object) Cast	left-to-right	(object) …
	(array) Cast	left-to-right	(array) …
17	instanceof	left-to-right	… instanceof …
16	Postfix Increment	left-to-right	… ++
	Postfix Decrement	left-to-right	… --
15	Logical NOT	right-to-left	! …
	Bitwise NOT	right-to-left	~ …
	Unary Plus	right-to-left	+ …
	Unary Negation	right-to-left	- …
	Prefix Increment	right-to-left	++ …
	Prefix Decrement	right-to-left	-- …
14	Exponentiation	right-to-left	… ** …
	Multiplication	left-to-right	… * …
	Division	left-to-right	… / …
	Remainder	left-to-right	… % …
13	Addition	left-to-right	… + …
	Subtraction	left-to-right	… - …
12	Bitwise Left Shift	left-to-right	… << …
	Bitwise Right Shift	left-to-right	… >> …
	Bitwise Unsigned Right Shift	left-to-right	… >>> …
11	Less Than	left-to-right	… < …
	Less Than Or Equal	left-to-right	… <= …
	Greater Than	left-to-right	… > …
	Greater Than Or Equal	left-to-right	… >= …
10	Equality	left-to-right	… == …
	Inequality	left-to-right	… != …
	Strict Equality	left-to-right	… === …
	Strict Inequality	left-to-right	… !== …
9	Bitwise AND	left-to-right	… & …
8	Bitwise XOR	left-to-right	… ^ …
7	Bitwise OR	left-to-right	… | …
6	Logical AND	left-to-right	… && …
5	Logical OR	left-to-right	… || …
4	Conditional (Ternärer Operator)	right-to-left	… ? … : …
3	Assignment (Zuweisung)	right-to-left	… = …
			… += …
			… -= …
			… **= …
			… *= …
			… /= …
			… %= …
			… <<= …
			… >>= …
			… >>>= …
			… &= …
			… ^= …
			… |= …
0	Comma / Sequence	left-to-right	… , …

Scope (Gültigkeitsbereich)

Scope (Bereich) beschreibt die Sichtbarkeit (Visibility) einer existierenden Variable (bzw Funktion) in einem bestimmten Bereich.

PHP kennt kein **Block-Scope** (wie zB in JavaScript, Seite 425).

PHP kennt keine **Scope Chain** (Verkettung von Gültigkeitsbereichen). Dieses Verhalten gibt es zum Beispiel in JavaScript (Seite 422), durch das Verschachteln von Funktionsdeklarationen.

PHP kennt:

1) **Global Scope** Seite 480

   Nur **Konstanten** sind automatisch im **Global Scope**. Damit eine äußere Variable auch in der Funktion sichtbar ist, muss die Variable (innerhalb der Funktion) mit dem Keyword `global` auf Global Scope gesetzt werden.

2) **Static Scope** Seite 482

   Variablen, die innerhalb einer Funktion, mit dem Keyword `static` auf **Static Scope** gesetzt werden, behalten ihren Wert bei Funktionsende. Beim nächsten Funktionsaufruf wird der Wert nicht zurückgesetzt.

3) **Local Scope**

   Variablen innerhalb einer Funktion sind im **Local Scope**. Sie sind ausserhalb der Funktion nicht sichtbar und somit **Out Of Scope**. Bei Beendigung der Funktion wird die Variable (und ihr Wert) entfernt.

   Soll die Funktion-Variable auch beim nächsten Aufruf vorhanden sein, muss sie mit dem Keyword `static` auf **Static Scope** gesetzt werden.

Global-Scope

Im Gegensat zu JavaScript, sind äußere Variablen in der Funktion nicht sichtbar (Global-Scope Seite 424).

Damit eine äußere Variable auch in der Funktion sichtbar ist, muss die Variable (innerhalb der Funktion) mit dem Keyword `global` auf Global Scope gesetzt werden.

Nur **Konstanten** sind automatisch im **Global Scope**.

Beispiel 1a: äußere Variable in einer Funktion

```
$foo="wuff";

function sprich()
 {
    echo $foo;                    // $foo ist nicht sichtbar: Out of Scope
 }

sprich();
```

Ergibt als Output:

```
Notice: Undefined variable: foo
```

Beispiel 1b: **globale Variable** (in einer **Funktion**) mit dem Keyword `global` sichtbar machen

```
$foo="wuff";

function sprich()
 {
    global $foo;          // äußere Variable $foo auf Global Scope setzen

    echo $foo;
 }

sprich();
```

Ergibt als Output:

```
wuff
```

Beispiel 2: in **JavaScript** ist eine **äußere Variable** auch in der **Funktion** sichtbar

```
var foo='wuff'

function sprich() {
 console.log(foo)              // foo ist in JavaScript auch in der Funktion sichtbar
}

sprich()
```

Ergibt als Output:

```
wuff
```

Konstante

Eine **Konstante** ist im **Global Scope** und somit auch in Funktionen und Klassen sichtbar.

Beispiel:

```
define('SPRICH','wuff',0);          //Konstante SPRICH
$foo="Dackel";

function meldung()
    {
       echo SPRICH;                 //Konstante ist In Scope

       echo $foo;                   //Variable ist Out of Scope
    }

meldung();
```

Ergibt als Output:

```
wuff
Notice: Undefined variable: foo
```

$GLOBALS

Im vordefinierten Array `$GLOBALS` befinden sich alle globalen Variablen.

Diese Built-in-Variable ist in einer Funktion **In Scope** und ermöglicht somit einen Zugriff auf Variablen ausserhalb der Funktion.

Beispiel:

```
$foo = 666;

function meineFunktion ()
 {
    echo $GLOBALS['foo'];                // Built-in Array
 }

meineFunktion();
```

Ergibt als Output:

```
666
```

Static Scope

I

Control Flow (Flusskontrolle)

Man unterscheidet zwischen **bedingter Flusskontrolle** (Seite 483), **Schleifen** (Seite 487) und Unterprogrammen. Mit den Sprunganweisungen `break` und `continue` wird der Fluss in Schleifen verändert, mit `return` wird eine Funktion verlassen.
Mit `exit` (aka `die`) wird das PHP-Skript beendet. Mit `goto` wird an ein Label gesprungen.

Conditional Statements (Bedingte Flusskontrolle)

Für bedingte Abläufe gibt es den **ternären Operator** (Seite), **switch** (Seite) und **if/else-Anweisungen**.

if

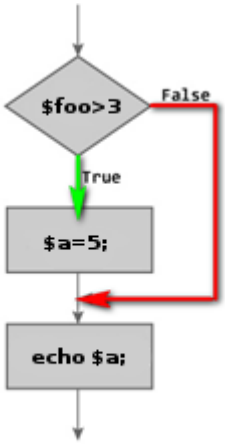
Bei der if-Flusskontrolle wird eine Expression (siehe Seite 444) ausgewertet.
Syntax:

```
if (Expression) Anweisungen
```

Ist das Ergebnis der Expression `true` (siehe Boolean Seite 454), wird der Programmblock der If-Bedingung ausgeführt. Ist das Ergebnis `false`, wird dieser `if`-Programmblock übersprungen. Ist der `if`-Programmblock mehrzeilig, muss er zwingend in geschweiften Klammern stehen.

Für Logische Operatoren Seite , für Vergleichsoperatoren Seite , für die Kurzschluss-Auswertung Seite_.

In diesem Beispiel wird die Expression `$foo>3` geprüft:




```
$foo = 4;
$a=1;
if ($foo>3)      // true
   {
      $a=5;
   }

echo $a;
```

Ergibt als Output:

```
5
```

In diesem Beispiel wird die Expression `foo<3` geprüft:

```
$foo = 4;
$a=1;
if ($foo<3){  $a=5;   }

echo $a;                        // 1
```

Ergibt als Output:

```
1
```

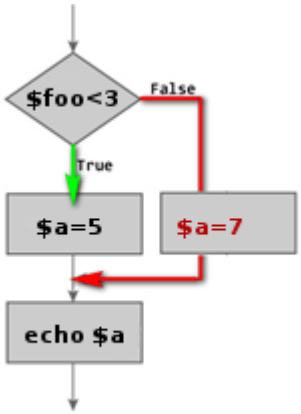
if else

Bei der if-else-Flusskontrolle wird eine Expression (siehe Seite 444) ausgewertet.

```
if (Expression) Anweisungen
   else Anweisungen
```

Ist das Ergebnis der Expression `true` (siehe Boolean Seite 454), wird der Programmblock der If-Bedingung ausgeführt.
Ist das Ergebnis `false`, wird dieser else-Programmblock ausgeführt.

In diesem Beispiel wird die Expression `$foo<3` geprüft:




```
$foo = 4;
$a=1;

if ($foo<3){  $a=5; }
   else  {  $a=7;   }

echo $a;                  // 7
```

Ergibt als Output:

```
7
```

if elseif

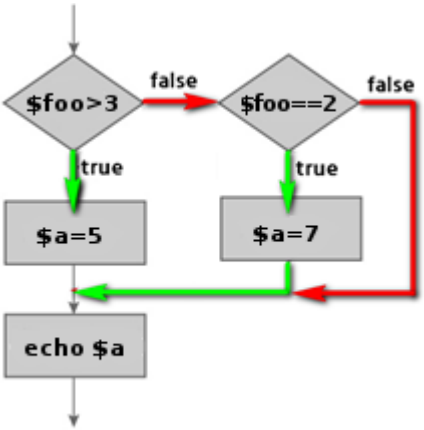
Bei der `if elseif` Flusskontrolle gibt es 2 Bedingungen. Zunächst wird nur die erste Expression ausgewertet. Ist das Ergebnis `true`, wird der dazugehörige Programmblock ausgeführt.
Ist das Ergebnis `false`, wird die zweite Expression ausgewertet (falls `true`, wird der dazugehörige Programmblock ausgeführt).

Beliebig viele `elseif` Abfragen sind möglich (alle werden sequentiell überprüft).
Optional ist am Ende aller `elseif` Abfragen der `else` Block möglich.

Syntax:

```
if (Expression) Anweisungen
   elseif  (Expression) Anweisungen
   [ else Anweisungen ]
```

Beispiel 1:

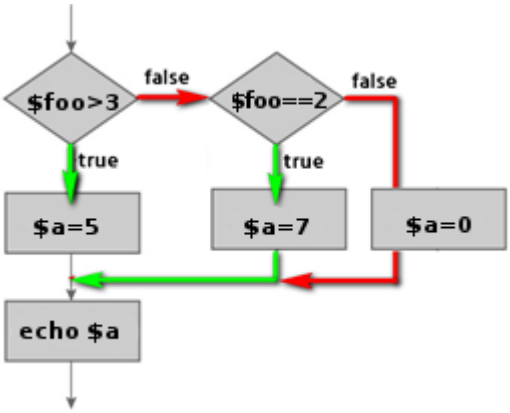



```
$foo = 2;
$a=1;

if ($foo>3){  $a=5; }
   elseif ($foo==2) {  $a=7;   }

echo $a;
```

Ergibt als Output:

```
7
```

Beispiel 2: mit einem `else` Block




```
$foo=1; $a=1;
if ($foo>3){  $a=5; }
   elseif ($foo==2) {  $a=7;   }
   else { $a=0; }

echo $a;
```

Ergibt als Output:

```
0
```

switch

Bei `switch` wird das Ergebnis einer Expression (Seite 444) mit Expressions verglichen. Beliebig viele case-Abfragen sind möglich (sie werden sequentiell durchlaufen).

Wenn das Expression-Ergebnis mit einer `case`-Expression übereinstimmt, werden alle folgenden Anweisungen, bis zum ersten vorkommenden `break` durchgeführt (falls kein break mehr vorkommt, bis zum switch-Blockende).

Die optionale `default` Block wird ausgeführt, wenn keine passende case-Expression gefunden worden ist (entspricht einem `else` bei `if`).

JavaScript erlaubt bei den case-Abfragen nur Literale (Seite 384).

Syntax:

```
switch (Expression)
{
    case (Expression): Anweisungen
    [break]

    [default: Anweisungen]
}
```

Beispiel 1:

```
$a=""; $b="";
switch (5+4)
 {
    case 2: $b="a2"; break;
    case 9: $b="a9"; break;
    case 12: $b="a12"; break;
    case 15: $b="a15";
 }
echo $b;
```

Ergibt als Output:

```
a9
```

Beispiel 2: **fall-through**

Nach dem übereinstimmenden `case`, werden alle Anweisungen bis zum ersten `break` ausgeführt (auch wenn `case` oder `default`-Anweisungen dazwischenliegen):

```
$a=""; $b="";
switch (5+4)
 {
    case 2: $b="a2"; break;
    case 9: $b="a9";
    case 12: $b="a12"; break;          // fall-through bis inklusive case12
    case 15: $b="a15"; break;
 }
echo $b;
```

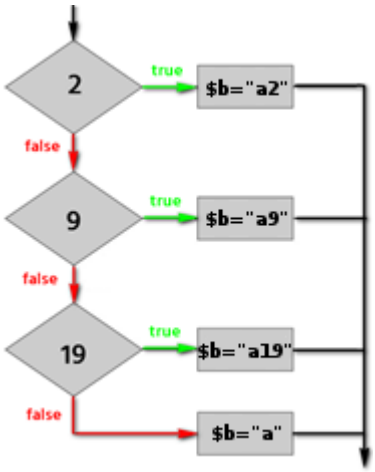
Ergibt als Output:

```
a12
```

Beispiel 3: default

Gibt es keine `case`-Übereinstimmung, werden alle Anweisungen zwischen `default` und dem ersten (folgenden) `break` durchgeführt (falls es kein `break` mehr gibt, bis zum `switch`-Blockende).
Die Position der `default`-Anweisung spielt keine Rolle.

```
$a="";
$b="";
switch (5+2)
 {
    case 2: $b="a2"; break;
    default: $b="a"; break;
    case 9: $b="a9"; break;
    case 19: $b="a19"; break;
 }
echo $b;
```



Ergibt als Output:

```
a
```

Beispiel 4: **fall-through**

Alle Anweisungen zwischen `default` und dem ersten (folgenden) `break` werden durchgeführt (auch wenn case-Anweisungen dazwischen liegen):

```
$a="";
$b="";
switch (5+2)
 {
    case 2: $b="a2"; break;
    default: $b="a";
    case 9: $b="a9"; break;
    case 19: $b="a19"; break;
 }
 echo $b;
```

Ergibt als Output:

```
a9
```

Loops (Schleifen)

Man unterscheidet zwischen der `for`-Schleife (Seite 490), der `while`-Schleife und der `do-while`-Schleife (Seite 489).

while Schleife

Am Schleifenkopf wird mit `while` eine Expression (siehe Seite 444) ausgewertet. Solange das Ergebnis der Expression `true` ist (siehe Boolean Seite 454), wird der Programmblock der Schleife ausgeführt.

Da die Exit-Bedingung am Anfang (Kopf) steht, wird die Schleife **Pre-test Loop** genannt.

Bei mehrzeiligen Anweisungen steht der Schleifen-Programmblock in

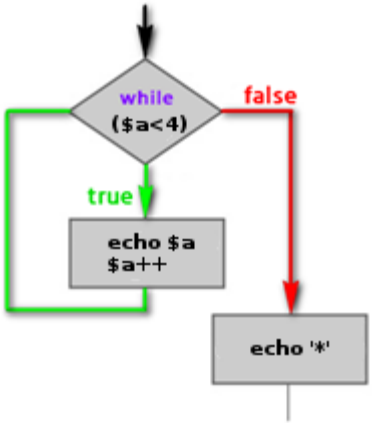
1) geschweiften Klammern oder
2) in `: endwhile;`

Syntax-Variante 1: **geschweifte Klammern**

```
while (Expression)
   {
      Anweisungen
   }
```

Beispiel:

```
$a=1;
while ($a<4)
   {
      echo $a;
      $a++;
   }
echo '*';
```



Ergibt als Output:

```
123*
```

Syntax-Variante 2: `:` `endwhile`

```
while (Expression)
   :
      Anweisungen
   endwhile;
```

Beispiel:

```
$a=1;
while ($a<4)
   :
      echo $a;
      $a++;
   endwhile;

echo '*';
```

Ergibt als Output:

```
123*
```

break

Mit dem Keyord break (im Schleifen-Programmblock) wird die Schleife an dieser Position verlassen. Beispiel:

```
$a=1;
while ($a<4)
  {
     echo $a;
     if ($a==2) break;         // verlasse die Schleife, wenn $a gleich 2
     $a++;
  }
echo '*';
```

Ergibt als Output:

```
12*
```

continue

Mit dem Keyword continue (im Schleifen-Programmblock) wird zum Schleifenkopf gesprungen.
Beispiel:

```
$a=0;
while ($a<3)
{
  $a++;
  if ($a==2) continue;      // springe zum Schleifenkopf, wenn a gleich 2
  echo $a;
}
echo '*';
```

Ergibt als Output (der Wert 2 wird nicht geschrieben):

```
13*
```

do while Schleife

Am Schleifenende wird mit `while` eine Expression (siehe Seite 444) ausgewertet. Solange das Ergebnis der Expression `true` ist (siehe Boolean Seite 454), wird der Programmblock der Schleife ausgeführt.

Bei mehrzeiligen Anweisungen muss der Schleifen-Programmblock in geschweiften Klammern stehen.

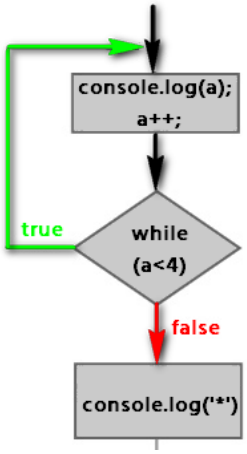
Der Schleifenkopf besteht aus dem Keyword `do`. Da die Exit-Bedingung am Schluss steht, wird die Schleife **Post-test Loop** genannt (somit wird der Programmblock mindestes einmal durchlaufen).

Syntax:

```
do {
      Anweisungen
}  while (Expression);
```

Beispiel:

```
$a="";

do
{
  echo $a;
  $a++;
}
while ($a<4);
echo '*';
```



Ergibt als Output:

```
123*
```

break

Mit dem Keyord `break` (im Schleifen-Programmblock) wird die Schleife an dieser Position verlassen.

Beispiel:

```
$a="";

do
{
  echo $a;
  $a++;
  if ($a==2) break;
}
while ($a<4);
echo '*';
```

Ergibt als Output:

```
1*
```

continue

Mit dem Keyword `continue` (im Schleifen-Programmblock) wird zum Schleifenende (`while`) gesprungen.

Beispiel:

```
$a=0;
do
{
  $a++;
  if ($a==2) continue;
  echo $a;
}
while ($a<3);
```

Ergibt als Output:

```
1
3
```

for Schleife

Am Schleifenkopf wird im `for`-Kopf eine Expression (siehe Seite 444) ausgewertet. Solange das Ergebnis der Expression `true` ist (siehe Boolean Seite 454), wird der Programmblock der Schleife ausgeführt.

Der `for`-Kopf besteht aus 3 Teilen (getrennt mit 2 Semikolons):

1) **Initialisierung** von Variablen (darunter auch der Schleifenzähler)
2) **Expression** (in der die Schleifenzählervariable vorkommt)
3) **Zuweisungen** mit den Zuweisungsoperatoren (darunter der Schleifenzählerupdate)

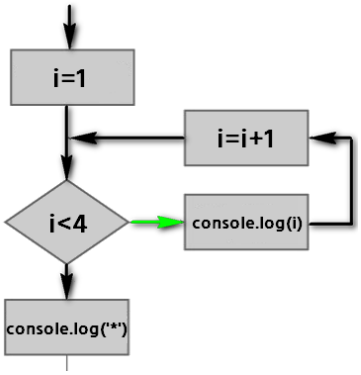
Der erste Teil wird nur einmal zu Beginn ausgeführt, Expression-Auswertung und Zuweisungen werden bei jedem Schleifendurchlauf ausgeführt. Für Logische Operatoren Seite, für Vergleichsoperatoren Seite.

Syntax-Variante 1: Programmblock mit geschweiften Klammern

```
for (Variableinitialisierungen ; Expression ; Schleifenzählerupdate)
 {

 }
```

Beispiel:

```
for ($i=1; $i<4; $i=$i+1)
 {
    echo $i;
 }
echo '*';
```



Ergibt als Output:

```
1
2
3
*
```

Im ersten Teil des Kopfes können beliebig viele Variablen initialisiert werden. Im dritten Teil können beliebig viele Zuweisungen stattfinden:

```
for ($i=0, $j=5   ; $j<8   ;     $i--, $j++)
 echo "$i $j";
echo '*';
```

Ist der Programmblock der Schleife einzeilig, muss der Block nicht in geschweiften Klammern stehen:
Die 2. Zeile wird dreimal durchlaufen, die 3. Zeile gehört nicht mehr zur Schleife und wird nur einmal ausgeführt.

Ergibt als Output:

```
0 5
-1 6
-2 7*
```

Syntax-Variante 2: Programmblock in : endfor

Statt den geschweiften Klammern (Schleifenblock) kann auch ein Doppelpunkt und das Keyword endfor verwendet werden.

```
for (Variableinitialisierungen ; Expression ; Schleifenzählerupdate)
    :

    endfor;
```

Beispiel:

```
for ($i=1; $i<4; $i=$i+1)
 :
    echo $i;
 endfor;
echo '*';
```

Ergibt als Output:

```
1
2
3
*
```

break

Mit dem Keyord break (im Schleifen-Programmblock) wird die Schleife an dieser Position verlassen.

Beispiel:

```
for ($i=1; $i<4; $i=$i+1)
 {
    if ($i==2) break;
    echo $i;
 }
echo '*';
```

Ergibt als Output:

```
1
*
```

continue

Mit dem Keyword continue (im Schleifen-Programmblock) wird zum Schleifenkopf gesprungen.

Beispiel:

```
for ($i=1; $i<4; $i=$i+1)
 {
    if ($i==2) break;
    echo $i;
 }
echo '*';
```

Ergibt als Output:

```
1
*
```

for each

Mit der `for each` Schleife werden sequentiell alle Keys und Werte eines Arrays ausgelesen.

Syntax-Variante 1: nur die Werte

```
foreach ($array as $wert)
 {

 }
```

Beispiel:

```
$foo = (array) [100,200,300];

foreach ($foo as $wert)
 {
  echo "<br> $wert";
 }
```

Ergibt als Output:

```
100
200
300
```

Syntax-Variante 2: Schlüssel und Werte

```
foreach ($array as $schluessel => $wert)
 {

 }
```

Beispiel1:

```
$foo = (array) [100,200,300];

foreach ($foo as $schluessel => $wert)
    {
     echo "<br> $wert";
    }
```

Ergibt als Output:

```
0 = 100
1 = 200
2 = 300
```

Beispiel2: assoziatives Array

```
$foo = (array) ['Name' => 'Bello', 'Rasse' => 'Dackel'];

foreach ($foo as $schluessel => $wert)
    {
     echo "<br> $wert";
    }
```

Ergibt als Output:

```
Name => Bello
Rasse => Dackel
```

Label & goto

Ein Label ist eine Sprungmarke und besteht aus dem Identifier (siehe Seite 438) und einem Doppelpunkt.

Label-Syntax:

```
Labelname:
```

Label-Identifier müssen eindeutig (unique) sein. Gesprungen wird mit dem Keyword goto.
Goto-Syntax:

```
goto Labelname;
```

Beispiel: ohne Label wird die erste Schleife (i) zweimal durchlaufen

```
for ($i=1;$i<=2;$i++)
 {
    for ($j=1;$j<=2;$j++)
       {
          echo "<br> i: $i  j: $j";
          if ($j==1) {break;}                        // beende $j (die zweite Schleife)
       }
 }
```

Ergibt als Output:

```
i: 1 j: 1
i: 2 j: 1
```

Die goto Anweisung springt zum Label ende. $i wird nur einmal durchlaufen:

```
for ($i=1;$i<=2;$i++)
 {
    for ($j=1;$j<=2;$j++)
       {
          echo "<br> i: $i  j: $j";
          if ($j==1) {goto ende;}                // beende $j (die zweite Schleife)
       }
 }

ende:
echo "<br> ende";
```

Ergibt als Output:

```
i: 1 j: 1
```

exit (die)

Mit dem Keyword `exit` wird das PHP-Skript beendet. Alias ist `die`

Beispiel

```
for ($j=1;$j<=4;$j++)
  {
      echo "<br> j: $j";
      if ($j==2) {exit;}
  }

ende:
echo "<br> ende";
```

Ergibt als Output:

```
j: 1
j: 2
```

Klassen & Objekte

Ein Objekt ist ein Datentyp, der mehr als einen Wert (gleichzeitig) enthalten kann. Ein Objekt ist somit kein Skalar.

Klassen sind Baupläne für Objekte. Lässt sich von einer Klasse weitere Klassen (**Subklasse**n) ableiten, wird sie **Superklasse** genannt. Diese Superklasse wird auch als **Parent Class** und **Base Class** bezeichnet.

Ein Objekt, das nach einem Bauplan erzeugt worden ist, wird **Instanz** (Instance, Occurence) genannt.

Multiple Vererbung (Multiple Inheritance) ist in PHP nicht möglich.

Klasse

Eine Klasse ist ein Bauplan für ein Objekt. Dieser Bauplan beschreibt Variablen und Funktionen eines Objektes.
Die erzeugten Objekte werden auch **Instanzen** (der jeweiligen Klasse) genant.

Die Struktur einer Klasse besteht aus dem Keyword `class`, dem *Klassennamen* und dem Body (in geschweiften Klammern):

```
class   Klassenname
   {
   }
```

Eigenschaft

Eine Klasse wird durch Eigenschaften **(Properties)** bestimmt (auch **Members** genannt).

Die Deklarationen der Eigenschaften werden im Body der Klasse angegeben. Eine Eigenschaft kann eine Variable oder eine Funktion (Methode) sein.

Access Specifiers (Scope)

Für den Gültigkeitsbereich (Scope) gibt es 3 **Access Specifier**:

1)	`private`	Zugriff nur innerhalb der Klasse
2)	`protected`	Zugriff nur innerhalb der Klasse und den abgeleiteten Kindklassen (Derived Classes)
3)	`public`	Zugriff ist auch von aussen möglich

Data Member

Data Member sind Eigenschaften, die Werte enthalten. Die Syntax einer Variable-Deklaration:

```
AccessSpecifier   Identifier   ;
```

Optional kann bei der Deklaration auch ein Literal (String oder Number) angegeben werden:

```
AccessSpecifier   Identifier   =   Literalexpression ;
```

Der Wert kann nur ein Literal bzw eine Literal-Expression sein (Variablen sind nicht möglich).

Beispiel:

```
class hunde
   {
      public $rasse = "Dackel";
   }

$bello = new hunde();                          // bello ist eine Instanz der Klasse hunde

echo $bello->rasse;                            // mit -> erfolgt der Zugriff auf eine Eigenschaftsvariable
```

Ergibt als Output:

```
Dackel
```

Bei einem Objekt können neue Data Members (Properties) auch nachträglich hinzugefügt werden.
Beispiel:

```
class hunde
   {
      public $rasse = "Dackel";
   }

$bello = new hunde();

$bello->rufname = "Bello";                     // neue Eigenschaft rufname

echo $bello->rufname;                          // Bello
```

Ergibt als Output:

```
Bello
```

Methode

Funktionen einer Klasse werden Methoden genannt.

Identifier

Der Name einer Methode:

1) kann nicht `stdClass` sein
2) sollte nicht mit 2 Unterstrichen beginnen
   PHP besitzt built-in Methoden, deren Identifier mit 2 Unterstrichen beginnen (zB `__sleep`, `__wakeup`, `__construct`). Sie werden **Magic Methods** genannt.

Syntax

Die Syntax einer Methoden-Deklaration (ohne Funktionsparameter):

```
class   Klassenname
   {
          function Funktionsname  ()
             {
             }
   }
```

Beispiel: Methode `ichBin` hat 1 Übergabeparameter

```
class hunde
    {
       public $rasse = "Dackel";                  // Data Member rasse

       function ichBin ($hundename)               // Methode ichBin
          {
             return "Ich bin $hundename";
          }
    }

$bello = new hunde();

echo $bello -> ichBin('Bello');                   // der Methode ichBin wird ein String übergeben
```

Ergibt als Output:

```
Ich bin Bello
```

$this

`$this` ist eine built-in Variable, die nur innerhalb einer Klassendeklaration verwendet werden kann.

Sie steht für das gegenwärtig verwendete Objekt und ermöglicht den Zugriff auf eine Objekt-Variable.

Beispiel:

```
class hunde {
       public $rasse = "Dackel";                                  // Data Member rasse

       public function ichBin ($hundename) {
          return "Ich bin $hundename, ein ".$this->rasse;         // this bezieht sich auf Bello
       }

    }

$bello = new hunde();


echo $bello -> ichBin('Bello');
```

Ergibt als Output:

```
Ich bin Bello
```

self

`$this` ist eine built-in Variable, die nur innerhalb einer Klassendeklaration verwendet werden kann.

Sie steht für das gegenwärtig verwendete Objekt

Magic Methods

Vordefinierte Built-in Methoden werden **Magic Methods** genannt.

`__construct()`	Seite 500	Konstruktur-Methode. Sie wird aufgerufen, wenn mit der Klasse ein neues Objekt erzeugt wird
`__toString()`	Seite 501	String-Konvertierung Die Methode wird aufgerufen, wenn das Objekt (bzw der Inhalt des Objektes) zu einem String konvertiert werden soll.

__call()

__callStatic()

__clone()

__debugInfo()

__destruct()

__get()

__invoke()

__isset()

__set()

__set_state()

__sleep()

__unset()

__wakeup()

__construct()

PHP hat einen vordefinierten Identifier für eine Konstruktor-Methode: `__construct()`

Sie wird aufgerufen, wenn mit der Klasse ein neues Objekt erzeugt wird.

Früher war eine Methode, deren Name mit dem Klassenname identisch war, die Konstruktor-Methode.

Dieses Verhalten ist deprecated.

Beispiel 1a: die Methode wird aufgerufen, wenn das Objekt `bello` erzeugt wird

```
class hunde
    {
       function __construct()
          {
             print "Konstruktor-Methode wurde aufgerufen.";
          }
    }

$bello = new hunde();                          // eine neue Instanz von der Klasse hunde erzeugen
```

Ergibt als Output:

```
Konstruktor-Methode wurde aufgerufen.
```

Beispiel 2a: der Methode wird ein Parameter übergeben (und der Parameterwert einer Objekt-Eigenschaft zugeordnet)

```
class hunde
    {
        function __construct($rasse)
            {
               $this->rasse = $rasse;              // der Eigenschaft rasse den Wert 'Dackel' zuweisen
            }
    }

$bello = new hunde('Dackel');                      // dem Konstruktor der Literal 'Dackel' übergeben

echo "Ich bin ein ". $bello->rasse;
```

Ergibt als Output:

```
Ich bin ein Dackel
```

__toString()

Die Methode wird aufgerufen, wenn das Objekt (bzw der Inhalt des Objektes) zu einem String konvertiert werden soll.

Beispiel:

```
class hund
 {
    function __construct ($rasse,$name)
       {
           $this->rasse = $rasse;
           $this->name = $name;
       }

    function __toString()
       {
          return "Ich bin " . $this->name . ", ein ".$this->rasse;
       }

 }

$bello = new hund('Dackel','Bello');

echo strval($bello);
```

Erzeugt als Output:

```
Ich bin Bello, ein Dackel
```

__debuginfo()

Die Methode wird bei var_dump() aufgerufen.

Beispiel:

```
class hund
 {
    private $hundeinfo="bin die info";

    function __construct ($name,   $rasse) {
          $this -> name = $name;
          $this -> rasse = $rasse;
       }

    function __debuginfo ()  {
             return ["Hundename" => $this->name,  "Rassenbezeichnung" => $this->rasse] ;
       }
    function ichBin () {
          return "Ich bin ". $this->name. ", ein ". $this->rasse;
       }
 }

$bello=new hund("Bello","Dackel");

var_dump ($bello);
```

Ergibt als Output:

```
object(hund)#1 (2)
{
  ["Hundename"]=>
  string(5) "Bello"
  ["Rassenbezeichnung"]=>
  string(6) "Dackel"
}
```

Objekt

Ein Objekt ist eine Instanz von einer Klasse.

Mit dem Keyword new wird ein neues Objekt erstellt. Syntax für die Generierung eines Objektes:

```
$Objektname = new Klassenname
```

Identifier von Klassen und Eigenschaften werden ohne $ angegeben.

Beispiel:

```
class hunde
    {
       public $rasse = "Dackel";
    }

$bello = new hunde();              // das Objekt bello ist eine Instanz der Klasse hunde
                                   // Internal Identifier ist in diesem Beispiel 1

echo gettype ($bello);             // object

$klaeff = new hunde();             // das Objekt klaeff ist eine Instanz der Klasse hunde
                                   // Internal Identifier ist in diesem Beispiel 2


var_dump($klaeff);
```

Ergibt als Output:

```
object
object(hunde)#2 (0) { }
```

Property

Eigenschaften werden auch **Members** genannt. Sie beinhalten die Daten und die Unterprogramme (Methoden) eines Objektes. Man unterscheidet somit 2 Arten von Eigenschaften:

1) Daten
2) Methoden (Funktionen)

Zugriff

Der Zugriff auf eine Objekt-Eigenschaft erfolgt über die -> Notation.

Syntax:

```
Objektname -> Eigenschaftsname
```

Beispiel:

```
class hunde {
       public $rasse = "Dackel";
    }

$bello = new hunde();

echo $bello->rasse;                        // rasse ist eine Eigenschaft vom Objekt bello
```

Ergibt als Output:

```
Dackel
```

Methode

Eine Methode ist eine Funktion eines Objektes.

Cookie

Ein Cookie ist ein Textfile, dass sich auf dem Rechner des Client befindet.

setcookie()

Bei Aufruf dieser Funktion erzeugt PHP im http-Header diesen Eintrag:

```
Set-Cookie: <em>value</em>
[; expires=<em>date</em>]
[; domain=<em>domain</em>]
[; path=<em>path</em>]
[; secure]
```

Die Angaben in den eckigen Klammern sind optional.

Security

Unter dem Begriff **Security** fallen auch Bereiche, die nicht direkt mit Software zu tun haben.
Zum Beispiel bei **Mißbrauch (Abuse)** von Diensten oder **Social Engineering**.

Mißbrauch (Abusement)

Bei Abusement wird ein Dienst auf eine Weise verwendet, die entweder vom Anbieter nicht vorgesehen oder untersagt wurde. Für den Mißbrauch muss somit keine Sicherheitslücke ausgenutzt werden.

Abuse of Storage

Speicherplatz wird mit überflüssigem, unpassendem oder verbotenem Inhalt belegt. Oft in Verbindung mit Traffic-Abusement.

Sockenpuppen (Sock Puppets)

Sockenpuppen sind getarnte Mehrfach-Accounts eines einzelnen Users. Von außen ist nicht zu erkennen, dass sich hinter diesen Accounts die gleiche Person verbirgt.
Mit Sockenpuppen können Abstimmungen, Bewertungen und Diskussionen manipuliert werden. Siehe Wikipedia.

Meatpuppets

Im Gegensatz zu Sockenpuppen, sind Meatpuppets keine Fake-Accounts (also Accounts von echten Benutzern). Meatpuppets handeln im geheimen Auftrag anderer. Siehe Wikipedia.

Astroturfing

Bei Astroturfing handelt es sich um politische oder kommerzielle Public Relation mit Socken/Meatpuppets. Die Puppen erzeugen ungewöhnlich viel an Leserbriefen, Blogs, Forenposts und emails. Siehe Wikipedia.

Fake-Account

Eine Accout, der vorsätzlich mit falschen Daten angelegt worden ist.

Troll

Ein Social-Media-Account, der vorsätzlich zur Störung, Provokation oder Manipulation angelegt worden ist.
Siehe Wikipedia.

Trollarmee

Mehrere Trolle mit gleichem Auftrag (oder zumindest gleicher Intention) werden Trollarmee (siehe Wikipedia) genannt. Das Vorgehen ist destruktiver als beim Astroturfing.

SQL

Kapitel 12

SQL & postgreSQL

Relation

Die Relation ist der Teil einer Datenbank-Tabelle, die nur aus den eigentlichen Werten (Body) besteht. Im Head der Datenbank-Tabelle befinden sich die Meta-Informationen (Attribute, Tabellenname, Datentypen).
Auch das Ergebnis einer Select-Anweisung, ist wieder eine Relation.

Body

Eine Tabelle, die nur aus den eigentlichen Werten (Body) besteht, wird Relation genannt.
Der Tabellenname ist der Relationsname.

Relationsschema

Ein Relationsschema beschreibt die Struktur der Tabelle. Der Tabellenname (**Relationsname**), die Attributsdefinitionen (Name, Datentyp, Größe) und die Indexdefinitionen bilden das **Relationsschema**.

Attribut (Head)

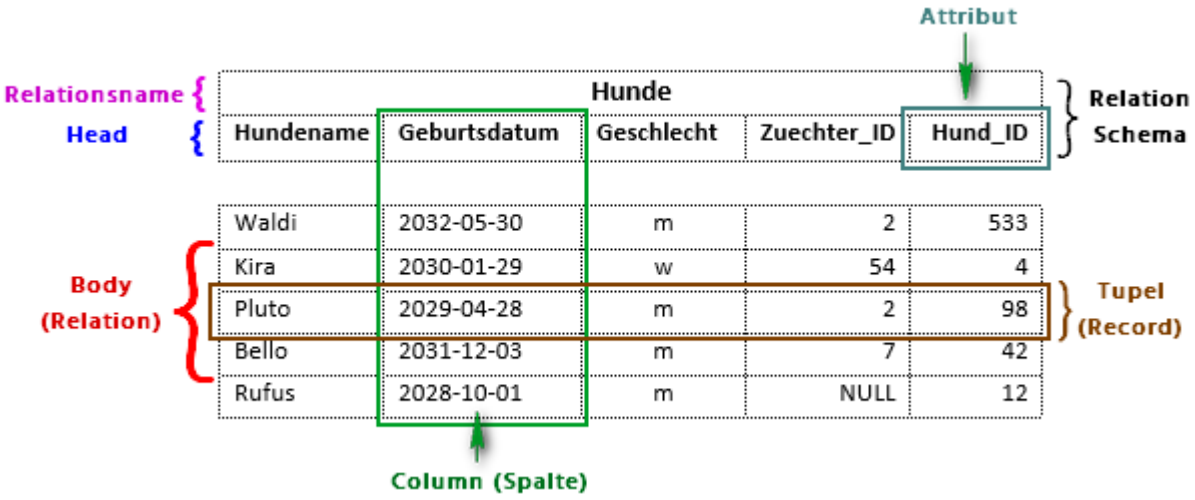
Die Attributbezeichnungen sind die Spaltennamen in der zweiten Zeile der Abbildung. Dieser Tabellenkopf (Head) dient nur zur Veranschaulichung und ist nicht Bestandteil der echten Datentabelle. Jedes Attribut besitzt seinen eigenen Datentyp.

Attributwert

Der eigentliche Dateninhalt einer Tabellenzelle wird Attributwert genannt. Alle Spalten (Zellen) einer Zeile sind ein Datensatz (Record) und werden **Tupel** genannt.

Tupel, Record Set

Eine einzelne Zeile (**Row**) in der Datentabelle wird **Tupel** und Datensatz (Record) genannt. Mehrere Zeilen sind ein **Record Set**. Alle Zeilen sind der Body (der Head ist das Relationen-Schema mit den Attributbezeichnungen)



Domain

Die Domain ist der mögliche (erlaubte) Wertebereich einer Spalte. Ein Eintragen eines nicht erlaubten Wertes wird von der Datenbank verhindert.

Heap

Der Heap ist die eigentliche, unsortierte Datentabelle. Eine sortierte Datentabelle ist ein **Clustered Index**.

Clustered Index (SQL Server)

Bei einem Clustered Index (MS SQL Server), sind bereits die eigentlichen Daten sortiert. Für gewöhnlich ist eine Datenbank-Tabelle nicht sortiert und wird deshalb Heap (Haufen) genannt.
Die Bedeutung bei einem Buch:
In einem Lexikon sind die Lexikon-Stichwörter bereits alphabetisch sortiert (ein Verzeichnis ist somit nicht notwendig).

psql

psql ist ein Shell-Programm zur Verwaltung von Datenbanken eines postgreSQL-Servers.

Siehe postgresql.org oder ein Shell-Aufruf mit dem Parameter help: **psl -help**

Shell Parameter

```
-h, --host=HOSTNAME
-p, --port=PORT          Port des Datenbankservers (Standard: »5432«)
-U, --username=NAME      Datenbank-Benutzername

-d, --dbname=DBNAME      Datenbank, zu der verbunden werden soll

-f, --file=DATEINAME     Anweisungen aus Datei ausführen und danach beenden
-l, --list               verfügbare Datenbanken auflisten und beenden

-v, --set=, --variable=NAME=WERT
                         psql-Variable NAME auf WERT setzen
                         (z.B. -v ON_ERROR_STOP=1)

-V, --version            Versionsinformationen anzeigen, dann beenden
```

Meta-Commands (Backslash Commands)

psql führt sql-Anweisungen aus, und besitzt auch eigene Anweisungen, die immer mit \ beginnen:

```
\q                    quit
\c Datenbankname      eine bestimmte Datenbank verwenden (connect)
\l                    alle Datenbanken anzeigen (list)
\d                    alle Tabellen anzeigen (Wildcard ist *).
\d Tabellenname       Attribute der angegebenen Tabelle anzeigen
\h                    help
```

Login

Mit dem User postgres:

```
C:\Program Files\PostgreSQL\bin>psql -U postgres
Passwort für Benutzer postgres:
psql (9.6rc1)
Geben Sie »help« für Hilfe ein.
postgres=#
```

Exit

Mit der Anweisung \c wird das Programm beendet:

```
postgres=# \q

C:\Program Files\PostgreSQL\bin>
```

Server-Version

Die Funktion version() gibt die postgresql-Version zurück:

```
postgres=# select version();
                           version
-------------------------------------------------------------
 PostgreSQL 19.6rc1, compiled by Visual C++ build 9800, 64-bit
(1 Zeile)
postgres=#
```

Oder:

```
C:\Program Files\PostgreSQL\bin>psql --version
psql (PostgreSQL) 9.6rc1
```

Datenbank

connect

Mit der Anweisung `\c` verbindet man sich zu einer bestimmten Datenbank (des postgresql-Servers):

```
postgres=# \c wilma
Sie sind jetzt verbunden mit der Datenbank »wilma« als Benutzer »postgres«.
wilma=#
```

create

Mit `create database` eine Datenbank anlegen:

```
postgres=# create database foo;
CREATE DATABASE
postgres=#
```

drop

mit drop database eine Datenbank löschen:

```
postgres=# drop database foo;
DROP DATABASE
postgres=#
```

list

Mit `\l` alle Datenbanken auflisten:

```
postgres=# \l
                                    Liste der Datenbanken
     Name      | Eigentümer | Kodierung |    Sortierfolge     |     Zeichentyp      |
---------------+------------+-----------+---------------------+---------------------+----
meinedatenbank | postgres   | UTF8      | German_Germany.1252 | German_Germany.1252 |
postgres       | postgres   | UTF8      | German_Germany.1252 | German_Germany.1252 |
(2 Zeilen)
```

wildcard

Wildcard ist der Stern.

Beispiel: alle Datenbanken, die mit `meineda` beginnen:

```
postgres=# \l meine*
                                    Liste der Datenbanken
     Name      | Eigentümer | Kodierung |    Sortierfolge     |     Zeichentyp      |
---------------+------------+-----------+---------------------+---------------------+----
meinedatenbank | postgres   | UTF8      | German_Germany.1252 | German_Germany.1252 |
(1 Zeile)
```

SQL Notationsbeschreibung

Erklärung zur Syntax-Notation:

- Anweisungen in **eckigen Klammern** sind optional
- bei Anweisungen in **geschweiften Klammern,** ist eine in den Klammern angegebene Alternative verpflichtend
- bei Anweisungen, die mit einem **|** getrennt sind, ist nur eine einzelne wählbar
- bei **3 Punkte**n darf die vorherige Anweisung öfters (nacheinander) angegeben werden
- Anweisungen in **Großbuchstaben** sind SQL-Keywords
- mit runden Klammern werden Anweisungen gruppiert

Relation (Tabelle)

Die Anweisung `\d` (ohne Parameter) zeigt alle Tabellen der gegenwärtigen Datenbank an.

Tabellen anzeigen

Beispiel: alle Tabellen anzeigen lassen

```
postgres=# \d
          Liste der Relationen
 Schema |  Name  |   Typ    | Eigentümer
--------+--------+---------+------------
 public | hunde  | Tabelle | postgres
 public | rassen | Tabelle | postgres
(2 Zeilen)
```

Wildcard

Der Wilcard ist der Stern.

Beispiel: alle Relationen, die mit `hund` beginnen:

```
postgres=# \d hund*
             Tabelle "public.hunde"
    Spalte     |          Typ          | Attribute
---------------+-----------------------+-----------
 hundename     | character varying(50) |
 geburtsdatum | date                  |
 geschlecht    | sex                   |
 rasse         | integer               |

            Tabelle "public.hunderassen"
 Spalte |      Typ      | Attribute
--------+---------------+-----------
 rasse  | character(50) |
```

Attribute einer Tabelle

Beispiel: die Attribute der Tabelle `hunde` anzeigen

```
postgres=# \d hunde
                Tabelle  "public.hunde"
    Spalte     |          Typ          | Attribute
---------------+-----------------------+-----------
 hundename     | character varying(50) |
 geburtsdatum | date                  |
 geschlecht    | sex                   |
 rasse         | integer               |
```

SQL Syntax

Geschichte

SQL-86	SQL1	
SQL-92	SQL2	
SQL:1999	SQL3	Ab SQL:1999 wird im Standard zwischen **Core Specification** und **Extensions** unterschieden. Die **Core Specification** sollte von jeder RDBMS unterstützt werden.
SQL:2003		XML hinzugefügt
SQL:2006		XML erweitert, objekt-orientierte Features
SQL:2008		zusätzliche objekt-orientierte Features

Verben

Meistens beginnt ein SQL-Statement mit einem Verb-Kommando und endet immer mit einem Semikolon.
Das Semikolon ist somit auch das Trennzeichen (Delimiter) bei mehreren Statements.

Beispiele von Verben:

SELECT	selektiere Daten aus einer Tabelle	Daten-Auswahl
UPDATE	Werte aktualisieren	Daten-Änderung
CREATE	erzeugen von Tabellen, Datenbank, Index, ...	Erstellung
DROP	löschen von Tabellen (Relationen) und Datenbanken	Tabellen&Datenbank-Löschung
DELETE	löschen von Zeilen einer Relation (Daten-Löschung	Datenzeilen-Löschung
INSERT	Zeilen in eine Tabelle (Relation) einfügen	Datenzeilen hinzufügen

Beispiel: zeige die Datenbankversion an

```
postgres=# select version();
                           version
-------------------------------------------------------------
 PostgreSQL 19.6rc1, compiled by Visual C++ build 9800, 64-bit
(1 Zeile)
postgres=#
```

Data Manipulation

SELECT	Daten selektieren und das Ergebnis als Tabelle zurückgeben
INSERT	Datenzeilen zu einer bestehen Tabelle hinzufügen
UPDATE	Datenzeilen ändern
MERGE	Conditionally inserts/updates/deletes new and existing rows
DELETE	Datenzeilen entfernen

Data Definition

CREATE TABLE	eine neue Tabelle erstellen
DROP TABLE	eine Tabelle löschen
ALTER TABLE	Struktur einer (bestehenden) Tabelle ändern
CREATE VIEW	eine neue Sicht (View) erstellen
DROP VIEW	einen View Löschen
CREATE INDEX	einen Index erstellen
DROP INDEX	einen Index löschen
CREATE SCHEMA	ein neues Schema erstellen
DROP SCHEMA	ein Schema löschen
CREATE DOMAIN	eine neue Domain (erlaubter Wertebereich einer Spalte) erstellen
ALTER DOMAIN	eine Domain ändern
DROP DOMAIN	eine Domain löschen

Access Control

GRANT Grants	user access privileges
REVOKE	Removes user access privileges
CREATE ROLE	Adds a new role to the database
GRANT ROLE	Grants role containing user access privileges
DROP ROLE	Removes a role from the database

Transaction Control

COMMIT	beendet die gegenwärtige Transaktion (Transaction)
ROLLBACK	bricht die gegenwärtige Transaktion ab
SET TRANSACTION	Defines data access characteristics of the current transaction
START TRANSACTION	beginnt eine neue Transaktion
SAVEPOINT	bestimmt einen Recovery-Punkt für eine Transaktion

Programmatic SQL

DECLARE	Defines a cursor for a query
EXPLAIN	Describes the data access plan for a query
OPEN	Opens a cursor to retrieve query results
FETCH	Retrieves a row of query results
CLOSE	Closes a cursor
PREPARE	Prepares a SQL statement for dynamic execution
EXECUTE	Executes a SQL statement dynamically
DESCRIBE	Describes a prepared query

Whitespace

Whitespace (Seite 237) kann beliebig gesetzt werden.

Beispiel1a:

```
select hundename,rasse,geschlecht  from hunde;
```

Beispiel1b:

```
select
    hundename,
    rasse,
    geschlecht
from
    hunde;
```

Komma (List Separator)

Das Komma ist der List-Separator. Der Separator ist notwendig, wenn mehr als ein Item angesprochen werden soll.

Beispiel1a: zeige die Spalte hundename an (von der Tabelle hunde)

```
select hundename from hunde;
```

Beispiel1b: zeige die Spalten hundename, rasse, geschlecht an (von der Tabelle hunde)

```
select hundename, rasse, geschlecht  from hunde;
```

Remark (Bemerkung)

Zwei Minuszeichen markieren den Beginn einer Userbemerkung. Das Zeilenende ist auch das Ende der Bemerkung.

Beispiel:

```
--    suche nach dem Film Sophie''s Choice
select * from filme where filmname='Sophie''s Choice';
```

Case Sensitive

Für String-Literale siehe Seite 518.

Keyword

SQL ist bei den Keywords **Case Insensitive**. Keywords werden vom SQL-Interpreter in Großbuchstaben umgewandelt werden.

Identifier

Laut SQL:99 und SQL:2003 sollten Identifier (für Tabellen, Spalten und Datenbanken) **Case Insensitive** sein.
Identifier in Anführungszeichen sollten **Case Sensitive** sein.

Firebird, DB2 und **Oracle** wandeln Identifier, die nicht in Anführungszeichen stehen, in **Großbuchstaben** um.

postgreSQL wandelt in **Kleinbuchstaben** um.

Bei **MySQL** hängt das Verhalten vom OS und von der Konfiguration ab. Siehe Variablen `lower_case_file_system` und `lower_case_table_names`. Spaltennamen sind immer Case Sensitive (egal, ob mit oder ohne Anführungszeichen).

Bei **T-SQL** ist der Identifier nur beim `create` **Case Sensitive** (danach nicht mehr).

Beispiel 1:

```
postgres=# create table Wilma (foo int);
CREATE TABLE
postgres=# drop table wilma;
DROP TABLE
```

Beispiel 2:

```
postgres=# create table Wilma (foo int);
CREATE TABLE
postgres=# drop table wIlma;
DROP TABLE
```

Beispiel 3:

```
postgres=# create table "Wilma" (foo int);
CREATE TABLE
postgres=# drop table wilma;
FEHLER:  Tabelle »wilma« existiert nicht
```

Beispiel 4: Identifier dürfen nur in **doppelten Anführungszeichen** stehen (nicht in einfachen)

```
postgres=# create table 'Wilma' (foo int);
FEHLER:  Syntaxfehler bei »'Wilma'«
ZEILE 1: create table 'Wilma' (foo int);
```

Literale

Literale sind Fixwerte und somit unveränderlich.

Arten von Literalen:

Boolean Seite 517

Der Datentyp Boolean kennt nur die 2 Werte `true` und `false`

Beim Cast von `NULL` kommt NULL zurück.

MySQL, **DB2**, **T-SQL** kennen diesen Literal-Datentyp nicht.

Null Seite 521

der Datentyp `NULL` hat nur einen möglichen Wert: `NULL`

`NULL` kann nur mit den Operatoren `IS NULL` und `IS NOT NULL` verglichen werden.

String Seite 518

Zeichenkette (**Alphanumeric Literal**)

Zahl Seite 522

SQL kennt nicht die Literale `INFINITY` und `NAN`

e Exponent Seite 521

Der Buchstabe **e** ist der Exponent Literal.

Er verschiebt das Komma in einer Zahl nach rechts oder nach links.

Die Zahl nach e bestimmt die Anzahl der Kommastellen.

Beispiele:

`12.34e-1`	Komma um eine Stelle nach links:	`1.234`
`12.34e+1`	Komma um eine Stelle nach rechts:	`123.4`

X Hexadec Seite 522

der Buchstabe **X** ist das Hexadezimal-Literal

nach dem X folgt in einfachen Anführungszeichen ein hexadezimaler Wert

Beispiel:

`X'ff'`	postgreSQL:	`01000001`
`X'41'`	MySQL:	`A`

DATE Seite 523

die Zeichenkette `DATE` ist das Date-Literal.

Nach dem `DATE` folgt in einfachen Anführungszeichen eine Datumsangabe

Beispiel:

`DATE'8 December 2043'`	postgreSQL:	`2043-12-08`
`DATE'8 December 2043'`	MySQL:	`Incorrect DATE value`

TIME Seite 523

Die Zeichenkette `TIME` ist das Time-Literal.

Nach dem `TIME` folgt in einfachen Anführungszeichen eine Zeitangabe.

Beispiel: `TIME'22:10'` ergibt: `22:10:00`

Beispiel:

`TIME'2044-02-04 12:11'`	postgreSQL:	`12:11:00`
`TIME'2044-02-04 12:11'`	MySQL:	`Incorrect TIME value`

TIMESTAMP Seite 523

Die Zeichenkette `TIMESTAMP` ist das Timestamp-Literal.

Nach dem `TIMESTAMP` folgt in einfachen Anführungszeichen eine Datumsangabe (optional mit Zeitangabe).

Beispiel:

`TIMESTAMP'2043-12-11'`	postgreSQL:	`2043-12-11 00:00:00`
	MySQL:	`Incorrect DATETIME value`

Boolean Literal

Der Datentyp Boolean hat nur 2 mögliche Werte: `true` und `false`

MySQL, **DB2**, **T-SQL** kennen diesen Literal-Datentyp nicht.

Beispiel 1: Boolean-Literal `false` selektieren. Ergibt **f**

```
select false;
```

Ergibt als Output:

```
bool
------
 f
```

Beispiel 2: Boolean-Literal `true` selektieren. Ergibt **t**

```
select true;
```

Ergibt als Output:

```
bool
------
 t
```

Beispiel 3: Integer-Literal `666` als Boolean-Cast ergibt **t**

```
select cast (666 as boolean);
```

Ergibt als Output:

```
bool
------
 t
```

Beispiel 4: Literal `NULL` als Boolean-Cast gibt NULL zurück

```
select cast (NULL as boolean);
```

Ergibt als Output:

```
bool
------
```

Beispiel 5: Literal `NULL` mit dem Operator `IS NULL` ergibt **t**

```
select null is  null;
```

Ergibt als Output:

```
bool
------
 t
```

String Literal

String-Literale müssen in **einfachen Anführungszeichen** stehen.

```
select 'foo' from werte where g1='a';
```

Ergibt in **postgreSQL** (Beispiel funktioniert auch in **DB2, T-SQL** und **MySQL**):

```
 blub | ?column?
------+----------
    4 | foo
```

Doppelte Anführungszeichen erlaubt nur MySQL:

```
select "foo" from werte where g1="a";
```

Ergibt:

```
+-----+
| foo |
+-----+
| foo |
+-----+
```

Alias-Identifier

Alias-Identifier für Spalten müssen in **doppelten Anführungszeichen** stehen.

Groß-Kleinschreibung bei **Identifier** siehe Seite 515.

```
select w1 as "blub" from werte;
```

Ergibt in **postgreSQL** (Beispiel funktioniert auch in **DB2, T-SQL** und **MySQL**):

```
blub
------
    3
    4
```

Einfache Anführungszeichen erlauben **T-SQL** und **MySQL**:

```
select w1 as 'blub' from werte;
```

Ergibt in MySQL:

```
+------+
| blub |
+------+
|    3 |
|    4 |
+------+
```

Backticks erlaubt nur **MySQL**:

```
select w1 as `blub` from werte;
```

Ergibt in MySQL:

```
+------+
| blub |
+------+
|    3 |
|    4 |
+------+
```

Bei Tabellen- und Datenbank-Identifiert erlaubt MySQL nur Backticks (also keine Anführungszeichen).

Identifier

Identifier für Spalten, Tabellen und Datenbanken müssen in **doppelten Anführungszeichen** stehen.

Laut SQL:99 und SQL:2003 sollten Identifier (für Tabellen, Spalten und Datenbanken) **Case Insensitive** sein.
Indentifier in Anführungszeichen sollten **Case Sensitive** sein.

Firebird, DB2 und **Oracle** wandeln Identifier, die nicht in Anführungszeichen stehen, in **Großbuchstaben** um.

postgreSQL und **MySQL** wandeln in **Kleinbuchstaben** um.

MySQL erlaubt bei Identifier nur Backticks.

Beispiel 1: beim `create` die Tabelle `werte` ohne Anführungszeichen

```
create table werte (w1 int);
```

Wird in postgreSQL zu: `werte`
In DB2, Oracle, Firebird: `WERTE`

Beispiel 1a: **postgreSQL** (beim `create` in Kleinbuchstaben umgewandelt)

```
select w1 from "werte";
```

Ergibt in **postgreSQL**:

```
w1
----
  3
```

Beispiel 1b: **DB2** (beim `create` in Großbuchstaben umgewandelt)

```
select w1 from "WERTE";
```

Beispiel 1c: **T-SQL** ist nur beim `create` **Case Sensitive (nur doppelte**

```
select w1 from "WeRtE";
```

Beispiel 1d: **MySQL** ist immer **Case InSensitive (**nur Backticks erlaubt)

```
select w1 from `WeRtE`;
```

Beispiele

Beispiel 1: der String-Literal `wilma`

```
select 'wilma';
```

Ergibt als Output (in **DB2** nicht erlaubt, da Tabellenreferenz zwingend notwendig):

```
?column?
----------
 wilma
```

Beispiel 2: zwei String-Literale mit || konkatenieren (für || bei **MySQL** siehe pipes_as_concat)

```
select 'carol ' || 'king' from werte;
```

Ergibt als Output (**T-SQL** nicht möglich. Per default bei **MySQL** nicht möglich):

```
?column?
----------
 carol king
```

Nur **T-SQL** kennt + als Konkatenation-Operator:

```
select 'carol ' + 'king' from werte;
```

Beispiel 3a: der String-Literal `Bello` in einfachen Anführungszeichen

```
postgres=# select * from "Hunde" where rufname='Bello';
 rufname
------
(0 Zeilen)
```

Beispiel 3b: doppelte Anführungszeichen erzeugen einen Error:

```
postgres=# select * from "Hunde" where rufname="Bello";
FEHLER:  Spalte »Bello« existiert nicht
ZEILE 1: select * from "Hunde" where rufname="Bello";
                                            ^
```

Escape Character

Wenn im String-Literal ein einfaches Anführungszeichen vorkommt, muss dieses escaped (gekennzeichnet) werden. Und zwar mit einem weiteren, gleich darauf folgendem Anführungszeichen.

Beispiel: suche in der Tabelle `filme` nach dem Film `Sophie's Choice`

```
select * from filme where filmname='Sophie''s Choice';
```

NULL Literal

Der Datentyp `NULL` hat nur einen möglichen Wert: `NULL`

`NULL` kann nur mit den Operatoren `IS NULL` und `IS NOT NULL` verglichen werden.

Beispiel 1: Literal `NULL` als Boolean-Cast gibt NULL zurück (im Output nicht sichtbar)

```
select cast (NULL as boolean);
```

Ergibt als Output:

```
bool
------
```

Beispiel 2: Literal `NULL` mit dem Operator **`IS NULL`** ergibt **t**

```
select null is  null;
```

Ergibt als Output:

```
bool
------
 t
```

Beispiel 3: Literal `NULL` mit dem Operator **`IS NOT NULL`** ergibt **f**

```
select null is not null;
```

Ergibt als Output:

```
bool
------
 f
```

Exponent-Literal

Der Buchstabe **e** ist der Exponent Literal. Er verschiebt das Komma in einer Zahl nach rechts oder nach links.

Die Zahl nach e bestimmt die Anzahl der Kommastellen.

Beispiel1:

```
select  12.34e-1 ;             -- Komma um eine Stelle nach links
```

Ergibt als Output:

```
1.234
```

Beispiel2:

```
select 12.34e+1     ;        -- Komma um eine Stelle nach rechts
```

Ergibt als Output:

```
123.4
```

Zahlen-Literal

Alle Ganz- und Kommazahlen sind Zahlen-Literale.

Beispiel1: der Integer-Literal `666`

```
select  666 ;
```

Ergibt als Output:

```
666
```

Beispiel2: der Kommazahl-Literal `-12.34`

```
select  -12.34 ;
```

Ergibt als Output:

```
-12.34
```

Hexadezimal Literal

Der Buchstabe **X** ist das Hexadezimal-Literal.

Nach dem X folgt in einfachen Anführungszeichen ein hexadezimaler Wert

Beispiel1: `X'ff'` ergibt bei postgreSQL eine binäre Zahl: `01000001`

`X'41'` ergibt bei MySQL die ASCII-Entsprechung: `A`

Beispiel2: `X'77696C6D61'` postgreSQL: `01110111011010010110110001101101011000 01`

`X'77696C6D61'` MySQL: `wilma`

Beispiel3: der Kommazahl-Literal `X'A0'`

```
select  X'AA' ;
```

Ergibt als Output:

```
10100000
```

DATE Literal

Die Zeichenkette `DATE` ist das Date-Literal.

Nach dem `DATE` folgt in einfachen Anführungszeichen eine Datumsangabe.

Beispiel:	`DATE'8 December 2043'`	postgreSQL:	`2043-12-08`
	`DATE'8 December 2043'`	MySQL:	`Incorrect DATE value`
Beispiel:	`DATE'2044/12/01'`	ergibt:	`2044-12-01`
Beispiel:	`DATE'2044.12.01'`	ergibt:	`2044-12-01`
Beispiel:	`DATE'2044 12 01'`	postgreSQL:	`2044-12-01`
	`DATE'2044 12 01'`	MySQL:	`Incorrect DATE value`
Beispiel:	`DATE'2044.12.01 09:01:22'`	postgreSQL:	`2044-12-01`
	`DATE'2044.12.01 09:01:22'`	MySQL:	`Incorrect DATE value`

TIME Literal

Die Zeichenkette `TIME` ist das Time-Literal.

Nach dem `TIME` folgt in einfachen Anführungszeichen eine Zeitangabe.

Beispiel:	`TIME'22:10'`	ergibt:	`22:10:00`
Beispiel:	`TIME'2044-02-04 12:11'`	postgreSQL:	`12:11:00`
	`TIME'2044-02-04 12:11'`	MySQL:	`Incorrect TIME value`

TIMESTAMP Literal

Die Zeichenkette `TIMESTAMP` ist das DateTime-Literal.

Nach dem `TIMESTAMP` folgt in einfachen Anführungszeichen eine Datumsangabe (optional mit Zeitangabe).

Beispiel:	`TIMESTAMP'2043-12-11'`	postgreSQL:	`2043-12-11 00:00:00`
		MySQL:	`Incorrect DATETIME value`
Beispiel:	`TIMESTAMP'2043-12-11 22:32'`	postgreSQL:	`2043-12-11 22:32:00`
	`TIMESTAMP'2043-12-11 22:32'`	MySQL:	`Incorrect DATETIME value`
Beispiel:	`TIMESTAMP'2043-12-11 12:22'`	ergibt :	`2043-12-11 12:22:00`

Expression (Ausdruck)

Eine Expression erzeugt einen Wert. Kann eine einzelne Variable oder ein einzelner Literal sein (**Single Expression**). Oder eine Kombination aus Literalen, Operatoren, Variablen und Rückgabewerten von Funktionen (**Compound Expression**).

Literale siehe Seite 516.

Single Expression

Beispiel: ein einzelner Zahlen-Literal

```
select  666 ;            -- Komma um eine Stelle nach links
```

Ergibt als Output:

```
666
```

Compound Expression

Beispiel: zwei Zahlen-Literale mit einem Multiplikator (Multiplikationsoperator).

```
select  3*2 ;
```

Ergibt als Output:

```
6
```

Compound Expression mit Identifier

Beispiel: Spalten-Identifier `meineSpalte` mit dem Vergleichsoperator auf den Wert 2 vergleichen

```
select  * from meineTabelle where meineSpalte=2 ;
```

Funktion

Beispiel: die Funktion `version()` gibt als Rückgabewert die Server-Version zurück

```
select  version() ;
```

Ergibt als Output:

```
PostgreSQL 32.1rc1, compiled by Visual C++ build 21800, 128-bit
```

Datentpyen

Zahlen (Numbers)

ISO-SQL kennt: 3 Integertypen (Seite 527)

SMALLINT	Seite 528	
INT	Seite 529	`INTEGER` ist ein Alias für `INT`
BIGINT	Seite 530	

2 exakte Fixed-Point Typen (Seite 531)

NUMERIC	Seite 531	
DECIMAL	Seite 531	`DEC` ist ein Alias für `DECIMAL`

3 (annähernd genaue) Floating-Point-Typen (Seite 533)

FLOAT	Seite 536
REAL	Seite 534
DOUBLE PRECISION	Seite 535

Die Byte-Größe wird von ANSI/ISO nicht bestimmt (verwendete Anzahl von Bytes hängt von der Implementierung ab). ANSI/ISO kennt die MySQL-Zusätze `signed` und `unsigned` nicht (bei `unsigned` beginnt der Wertebereich bei >=0).

postgreSQL kennt:

MONEY	Seite 538
SERIAL	Seite 537

Numerics (Zahlen)

3 ISO-Typen für Integer (Seite 527):

	Byte-Größe**	DB	Wertebereich	
SMALLINT	2 Byte		-32768 (-2^{15})	+32767 ($2^{15}-1$)
INT	4 Byte		-2147483648 (-2^{31})	+2147483647 ($2^{31}-1$)
BIGINT	8 Byte		-2^{63}	$2^{63}-1$
TINYINT	1 Byte	T-SQL:	0	255
		MySQL***:	0	255
			-128	127
MEDIUMINT	3 Byte	MySQL***:	0	16777215
			8388608	8388607
SERIAL	8 Byte	MySQL:	Syntax Zucker für: BIGINT UNSIGNED NOT NULL AUTO_INCREMENT UNIQUE	
	4 Byte	postgreSQL:	Syntax Zucker für eine SEQUENCE (auto-increment Integer)	
SMALLSERIAL	2 Byte	postgreSQL:	Syntax Zucker für eine SEQUENCE (auto-increment Integer)	
BIGSERIAL	8 Byte	postgreSQL:	Syntax Zucker für eine SEQUENCE (auto-increment Integer)	

2 ISO-Typen für genaue Kommazahlen (Seite 531)

	Precision	Scale	DB	precision = Ziffern insgesamt, davon scale Kommastellen
decimal	p	s		Datenbank muss zwingend den angegebenen Datentyp verwenden.
			T-SQL:	Maximum von p ist 18, Default-p ist 10
			MySQL:	Maximum von p ist 38, Default-p ist 18
			postgreSQL:	Maximum von p ist 1000, Maximum von s ist 100
numeric	p	s		wie decimal, DB darf aber auch einen genaueren Datentyp verwenden.

Typen für Money (Seite 538):

	Byte-Größe**	DB	Wertebereich	
money	8 Byte*	T-SQL:	-922337203685477.5808	922337203685477.5807
		postgreSQL:	-92233720368547758.08	92233720368547758.07
smallmoney	4 Byte	T-SQL:	-214,748.3648	214,748.3647

3 ISO-Typen für *annähernd* genau Kommazahlen nach IEEE_754 (Seite 533):

	Precision ist dynamisch (abhängig von der jeweiligen Zahl und der Byte-Größe). MySQL kennt zusätzlich p und s				
real			4 Byte	1E-37	1E+37
double precision			8 Byte	1E-307	1E+308
	Bit-Anzahl Mantisse			MySQL kennt zusätzlich s	
float	b	1-24	4 Byte	1E-37	1E+37
		25-53	8 Byte	1E-307	1E+308
	Bit	Scale			
	b	s	MySQL:	zusätzlich (optional) Scale: immer auf s Kommastellen runden & formatieren	

* 1 Byte Ersparnis zu decimal(19,4): -999,999,999,999,999.9999 bis 999,999,999,999,999.9999

** ANSI/ISO bestimmt keine Byte-Größen

*** MySQL kennt als Erweiterung signed (mit Vorzeichen) und unsgined (kein Vorzeichnen). Bei unsigned beginnt der Wertebereich bei 0

Integer (Ganzzahl)

ISO-SQL kennt: 3 Integertypen (SMALLINT, INT, BIGINT). INTEGER ist ein Alias für INT.

Die Byte-Größe wird von ANSI/ISO nicht bestimmt (verwendete Anzahl von Bytes hängt von der Implementierung ab).

ANSI/ISO kennt die MySQL-Zusätze signed und unsigned nicht (bei unsigned beginnt der Wertebereich bei >=0).

3 ISO-Typen für Integer (Ganzzahlen):

	Byte-Größe**	DB	Wertebereich	
SMALLINT	2 Byte		-32768 (-2^{15})	+32767 ($2^{15}-1$)
INT	4 Byte		-2147483648 (-2^{31})	+2147483647 ($2^{31}-1$)
BIGINT	8 Byte		-2^{63}	$2^{63}-1$
TINYINT	1 Byte	T-SQL:	0	255
		MySQL:	0	255
			-128	127
MEDIUMINT	3 Byte	MySQL***:	0	16777215
			8388608	8388607
SERIAL	8 Byte	MySQL:	Syntax Zucker für: BIGINT UNSIGNED NOT NULL AUTO_INCREMENT UNIQUE	
	4 Byte	postgreSQL:	Syntax Zucker für eine SEQUENCE (auto-increment Integer)	
SMALLSERIAL	2 Byte	postgreSQL:	Syntax Zucker für eine SEQUENCE (auto-increment Integer)	
BIGSERIAL	8 Byte	postgreSQL:	Syntax Zucker für eine SEQUENCE (auto-increment Integer)	

Kommazahl eintragen

Vor dem Eintrag einer Kommazahl in eine Spalte mit exaktem Datentyp, wird die Kommazahl von der Datenbank automatisch kaufmännisch gerundet (falls notwendig):

Beispiel: int

```
wilma=# create table meinetabelle (spalte int);
CREATE TABLE
wilma=# insert into meinetabelle (spalte) values (12.04);
INSERT 0 1
wilma=# select * from meinetabelle;
 spalte
--------
     12
(1 Zeile)
```

Nicht passende Zahl eintragen

Der Versuch, eine nicht passende Zahl (außerhalb des Wertebereiches) einzutragen, löst einen Error aus.

Beispiel: bei smallint der Wert 99999999

```
wilma=# create table meinetabelle (spalte smallint );
CREATE TABLE
wilma=# insert into meinetabelle (spalte) values (99999999);
FEHLER:  smallint ist außerhalb des gültigen Bereichs
```

Datentyp SMALLINT

Der Datentyp `SMALLINT` ist für Ganzzahlen (keine Kommastellen). Die Größe und der Wertebereich hängt von der Implementierung ab (**darf aber nicht größer als INTEGER sein**):
Nur MySQL kennt als Erweiterung `signed` (mit Vorzeichen) und `unsgined` (kein Vorzeichnen). Bei `unsigned` beginnt der Wertebereich bei 0.

Byte-Größe:

postgreSQL:	SMALLINT	2 Byte	-32768	32767	
T-SQL:	SMALLINT	2 Byte	-32768	32767	
MySQL:	SMALLINT	2 Byte	-32768	32767	(signed)
			0	65535	(unsigned)

Der Versuch eine zu große Zahl einzutragen, erzeugt einen "**außerhalb des gültigen Bereichs**"-Fehler.
Beispiel:

```
create table meinetabelle
 (
    spalte smallint
 );
insert into meinetabelle (spalte) values (123456789);
```

Erzeugt als Output:

```
 FEHLER: smallint ist außerhalb des gültigen Bereichs
```

Eine Kommazahl wird gerundet eingetragen.
Beispiel:

```
create table meinetabelle
 (
    spalte smallint
 );
insert into meinetabelle (spalte) values (123.5);
```

Erzeugt als Output:

```
wilma=# select * from meineTabelle;
 spalte
--------
    124
(1 Zeile)
```

Datentyp INT

Der Datentyp `INT` ist für Ganzzahlen (keine Kommastellen). Die Größe und der Wertebereich hängt von der Implementierung ab (**darf aber nicht größer als BIGINT sein**).
Nur MySQL kennt als Erweiterung `signed` (mit Vorzeichen) und `unsigned` (kein Vorzeichnen). Bei `unsigned` beginnt der Wertebereich bei 0.

`INTEGER` ist ein Alias für `INT`.

Byte-Größe:

postgreSQL:	INT	4 Byte	-2147483648	2147483647	
T-SQL :	INT	4 Byte	-2147483648	2147483647	
MySQL:	INT	4 Byte	-2147483648	2147483647	(signed)
			0	4294967295	(unsigned)

Der Versuch eine zu große Zahl einzutragen, erzeugt einen "**außerhalb des gültigen Bereichs**"-Fehler.

Beispiel:

```
create table meinetabelle
 (
    spalte int
 );
insert into meinetabelle (spalte) values (1234567890123456);
```

Erzeugt als Output:

```
 FEHLER: int ist außerhalb des gültigen Bereichs
```

Eine Kommazahl wird gerundet eingetragen.

Beispiel:

```
create table meinetabelle
 (
    spalte int
 );
insert into meinetabelle (spalte) values (123.5);
```

Erzeugt als Output:

```
wilma=# select * from meineTabelle;
 spalte
--------
    124
(1 Zeile)
```

Datentyp BIGINT

Der Datentyp `BIGINT` ist für Ganzzahlen (keine Kommastellen). Die Größe und der Wertebereich hängt von der Implementierung ab.

Nur MySQL kennt als Erweiterung `signed` (mit Vorzeichen) und `unsgined` (kein Vorzeichnen). Bei `unsigned` beginnt der Wertebereich bei 0.

Byte-Größe:

postgreSQL:	BIGINT	4 Byte	-2147483648	9223372036854775807	
T-SQL:	BIGINT	4 Byte	-2147483648	9223372036854775807	
MySQL:	BIGINT	4 Byte	-2147483648	9223372036854775807	(signed)
			-9223372036854775808	18446744073709551615	(unsigned)

Der Versuch eine zu große Zahl einzutragen, erzeugt einen "**außerhalb des gültigen Bereichs**"-Fehler.
Beispiel:

```
create table meinetabelle
 (
    spalte int
 );
insert into meinetabelle (spalte) values (1234567890123456);
```

Erzeugt als Output:

```
 FEHLER: int ist außerhalb des gültigen Bereichs
```

Eine Kommazahl wird gerundet eingetragen.
Beispiel:

```
create table meinetabelle
 (
    spalte int
 );
insert into meinetabelle (spalte) values (123.5);
```

Erzeugt als Output:

```
wilma=# select * from meineTabelle;
 spalte
--------
    124
```

Fixed-Point exact Number

Datentyp NUMERIC (DECIMAL)

Der Datentyp **NUMERIC** ist für **genaue Kommazahlen.** Die Genauigkeit muss exakt der vom User angegebenen Genauigkeit entprechen (im Unterschied zum Datentyp `decimal`).
Bei **DECIMAL** darf die Datenbank auch einen genaueren Datentyp verwenden (der einzige Unterschied zu numeric).

Bei der Typdefinition kann **Precision** (Ziffern insgesamt) und **Scale** (Kommastellen) angegeben werden.

Default für Scale ist 0. Default für Precision hängt von der Implementierung ab.

DEC ist ein Alias für DECIMAL.

2 ISO-Typen für genaue Kommazahlen (Fixed-Point)

	Precision	Scale	DB	**precision** = Ziffern insgesamt, davon **scale** Kommastellen
DECIMAL	p	s		Datenbank muss zwingend den angegebenen Datentyp verwenden.
			T-SQL:	Maximum von p ist 18, Default-p ist 10
			MySQL:	Maximum von p ist 38, Default-p ist 18
			postgreSQL:	Maximum von p ist 1000, Maximum von s ist 100
NUMERIC	p	s		wie decimal, DB darf aber auch einen genaueren Datentyp verwenden.

Jede Kombination stellt einen **eigenen Datentyp** dar. `numeric (6,0)` ist ein anderer Datentyp als `numeric (6,1)`

Voraussetzung für Precision: `p > 0`
Voraussetzung für Scale: `s >= 0`
`s < p`

Anzahl Vorkommaziffern: `p - s`

p bestimmt die Gesamt-Anzahl der Ziffern (Vorzeichen und das Kommazeichen sind keine Ziffern).

Syntax-Möglichkeiten

Die p und s-Angaben sind optional (die Scale-Angabe kann nur in Kombination mit Precision verwendet werden):

```
NUMERIC
NUMERIC ( p )
NUMERIC ( p , s )
```

Beispiel: Wertebereich von `-9.99` bis `+9.99` benötigt: `DECIMAL (3,2)`
Bei der Anzeige werden fehlende Nachkommastellen mit 0 aufgefüllt.

```
wilma=# create table meinetabelle (spalte numeric (3,2) );
CREATE TABLE
wilma=# insert into meinetabelle (spalte) values (-9.4);
INSERT 0 1
wilma=# insert into meinetabelle (spalte) values (0);
INSERT 0 1
wilma=# select * from meinetabelle;
 spalte
--------
  -9.40
   0.00
```

DECIMAL versus NUMERIC

Bei DECIMAL darf die Datenbank auch einen genaueren Datentyp verwenden. Das ist der einzige Unterschied zu NUMERIC.

Kommazahl eintragen

Vor dem Eintrag einer Kommazahl in eine Spalte mit exaktem Datentyp, wird die Kommazahl von der Datenbank automatisch kaufmännisch gerundet (falls notwendig):

Beispiel: NUMERIC (2,0)

```
wilma=# create table meineTabelle (meineSpalte NUMERIC (2,0) );
CREATE TABLE
wilma=# insert into meineTabelle (meineSpalte) values (3.9) ;
INSERT 0 1
wilma=# select * from meineTabelle;
 meinespalte
-------------
           4
```

Überlauf (Nicht passende Zahl)

Der Versuch, eine nicht passende Zahl (außerhalb des Wertebereiches) einzutragen, löst einen Error aus.

Beispiel: bei `NUMERIC(2,1)` der Wert `-99.8`

```
wilma=# create table meinetabelle (spalte NUMERIC (2,1) );
CREATE TABLE
wilma=# insert into meinetabelle (spalte) values (-9.9);
INSERT 0 1
wilma=# insert into meinetabelle (spalte) values (-99.8);
FEHLER:  Feldüberlauf bei Typ »NUMERIC«
DETAIL:  Ein Feld mit Präzision 2, Skala 1 muss beim Runden einen Betrag von weniger als 10^1
         ergeben.
```

Speicherplatz in Bytes

SQL Server:	Genauigkeit	Speicherplatz in Bytes
	1 - 9	5
	10-19	9
	20-28	13
	29-38	17
postgreSQL:	two bytes for each group of four decimal digits, plus three to eight bytes overhead.	

Floating-Point Numbers (IEEE 754)

ANSI/ISO-SQL gibt keine Bytegröße vor (die verwendete Byte-Größe ist somit von der Implementierung abhängig).

IEEE_754 kann nicht jede Kommazahl exakt abbilden. Die Genauigkeit hängt von der jeweiligen Zahl und der Byte-Größe ab. Diese Datentypen werden auch **Approximate Numeric Types** genannt.

3 ISO-Typen für Kommazahlen nach IEEE_754 (*annähernd* genau):					
	Precision ist dynamisch (abhängig von der jeweiligen Zahl und der Byte-Größe). MySQL kennt zusätzlich p und s				
REAL			4 Byte	1E-37	1E+37
DOUBLE PRECISION			8 Byte	1E-307	1E+308
	Bit-Anzahl Mantisse			MySQL kennt zusätzlich s (Scale)	
FLOAT	b	1-24	4 Byte	1E-37	1E+37
		25-53	8 Byte	1E-307	1E+308
	Bit	Scale			
	b	s	MySQL:	zusätzlich (optional) Scale: immer auf s Kommastellen runden & formatieren	

Beim Datentyp FLOAT kann die Anzahl der **Bits für die Mantisse** (siehe IEEE_754) angegeben werden.

Ist b kleiner als 25, verwenden SQL Server und postgreSQL immer 24 Bits für die Mantisse.

Ist b größer als 24, verwenden SQL Server und postgreSQL immer 53 Bits für die Mantisse.

Ungenauigkeit

IEEE_754 kann nicht jede Kommazahl exakt abbilden.

Die Genauigkeit hängt von der jeweiligen Zahl und der Byte-Größe ab.

Beispiel: `0.1 + 0.2` ergibt bei MySQL `0.30000000000000004`

```
drop table if exists meineTabelle;
create table meineTabelle (meineSpalte double precision);
insert into  meineTabelle (meineSpalte) values (0.1);
insert into  meineTabelle (meineSpalte) values (0.2);
select sum(meineSpalte) as summe from meineTabelle;
```

postgresql Ausgabe:

```
postgres=# select sum(meineSpalte) from meineTabelle;
 sum
-----
 0.3
```

mysql Ausgabe:

```
mysql> select sum(meinespalte) from meinetabelle;
+---------------------+
| sum(meinespalte)    |
+---------------------+
| 0.30000000000000004 |
+---------------------+
1 row in set (0.00 sec)
```

Datentyp REAL

ANSI/ISO-SQL gibt keine Bytegröße vor (die verwendete Byte-Größe ist somit von der Implementierung abhängig).
IEEE_754 kann nicht jede Kommazahl exakt abbilden. Die Genauigkeit hängt von der jeweiligen Zahl und der Byte-Größe ab.
postgreSQL kennt als Alias **float4**

REAL für Kommazahlen nach IEEE_754 (*annähernd* genau):				
	Precision ist dynamisch (abhängig von der jeweiligen Zahl und der Byte-Größe). MySQL kennt zusätzlich p und s			
REAL		4 Byte	1E-37	1E+37

Beispiel:

```
create table meineTabelle (meineSpalte1 REAL , meineSpalte2  DOUBLE PRECISION);

insert into  meineTabelle  values (1234567890.1234, 1234567890.1234);
```

Ergibt als Output:

```
wilma=# select * from meineTabelle;
 meinespalte1 |  meinespalte2
--------------+-----------------
 1.23457e+009 | 1234567890.1234
```

MySQL

MySQL kennt zusätzlich noch 2 Parameter: p Anzahl der Vorkommastellen
s Anzahl der Nachkommastellen

Syntax: REAL (p , s)

Die Parameter sind optional und können nur als Paar angegeben werden.

Der Versuch, eine zu große Zahl einzutragen, erzeugt einen **"Out of range value"**-Fehler.

Beispiel:

```
create table meineTabelle ( meineSpalte REAL (2,2));

insert into  meineTabelle values (123.4);
```

Erzeugt als Output: ERROR 1264 (22003): Out of range value for column 'meineSpalte'

Fehlende Nachkommastellen werden mit 0 aufgefüllt.

Beispiel:

```
create table meineTabelle
   (
      meineSpalte REAL (6,3)
   );

insert into  meineTabelle values (123.4);
```

Erzeugt als Output:

```
mysql> select * from meinetabelle;
+--------------+
| meineSpalte  |
+--------------+
|      123.400 |
+--------------+
```

Datentyp DOUBLE PRECISION

ANSI/ISO-SQL gibt keine Bytegröße vor (die verwendete Byte-Größe ist somit von der Implementierung abhängig). IEEE_754 kann nicht jede Kommazahl exakt abbilden. Die Genauigkeit hängt von der jeweiligen Zahl und der Byte-Größe ab.

postgreSQL kennt als Alias **float8**

DOUBLE PRECISION für Kommazahlen nach IEEE_754 (*annähernd* genau):				
	Precision ist dynamisch (abhängig von der jeweiligen Zahl und der Byte-Größe). MySQL kennt zusätzlich p und s			
DOUBLE PRECISION		8 Byte	1E-307	1E+308

Beispiel:

```
create table meineTabelle (meineSpalte1 real , meineSpalte2 double precision);

insert into  meineTabelle  values (1234567890.1234, 1234567890.1234);
```

Ergibt als Output:

```
wilma=# select * from meineTabelle;
 meinespalte1 |  meinespalte2
--------------+----------------
 1.23457e+009 | 1234567890.1234
```

MySQL

MySQL kennt zusätzlich noch 2 Parameter: p Anzahl der Vorkommastellen
s Anzahl der Nachkommastellen

Syntax: DOUBLE PRECISION (p,s)

Die Parameter sind optional und können nur als Paar angegeben werden.

Der Versuch, eine zu große Zahl einzutragen, erzeugt einen **"Out of range value"**-Fehler.

Beispiel:

```
create table meineTabelle (   meineSpalte DOUBLE PRECISION (2,2) );

insert into  meineTabelle values (123.4);
```

Erzeugt als Output: ERROR 1264 (22003): Out of range value for column 'meineSpalte'

Fehlende Nachkommastellen werden mit 0 aufgefüllt.

Beispiel:

```
create table meineTabelle
   (
      meineSpalte double precision (6,3)
   );

insert into  meineTabelle values (123.4);
```

Erzeugt als Output:

```
mysql> select * from meinetabelle;
+--------------+
| meineSpalte  |
+--------------+
|      123.400 |
+--------------+
```

Datentyp FLOAT

ANSI/ISO-SQL gibt keine Bytegröße vor (die verwendete Byte-Größe ist somit von der Implementierung abhängig).
IEEE_754 kann nicht jede Kommazahl exakt abbilden. Die Genauigkeit hängt von der jeweiligen Zahl und der Byte-Größe ab.

Beim Datentyp FLOAT kann die **Anzahl** der **Bits für die Mantisse** (siehe IEEE_754) angegeben werden.
Ist b kleiner als 25, verwenden T-SQL/MySQL/postgreSQL immer 24 Bits für die Mantisse.
Ist b größer als 24, verwenden T-SQL/MySQL/postgreSQL immer 53 Bits für die Mantisse.

Syntax: `FLOAT (b)`

Default für b ist bei T-SQL Server 53, bei postgreSQL und MySQL 24.

3 ISO-Typen für Kommazahlen nach IEEE_754 (*annähernd* genau):

	Bit-Anzahl Mantisse				
FLOAT	b	1-24	4 Byte	1E-37	1E+37
		25-53	8 Byte	1E-307	1E+308
	Bit	Scale			
	b	s	MySQL:	MySQL kennt zusätzlich Scale (Ergebnis wird auf s Kommastellen gerundet)	

Beispiel: bei `FLOAT (2)` nimmt postgreSQL den Datentyp `REAL`

```
create table meineTabelle (meineSpalte float (2) );
insert into  meineTabelle (meineSpalte) values (12.12345);
insert into  meineTabelle (meineSpalte) values (1234.12345);
```

Ergibt als Output:

```
wilma=# select * from meineTabelle;
 meinespalte
-------------
     12.1235
     1234.12
```

MySQL

MySQL kennt zusätzlich den Parameter Scale (nur in Kombination mit b). Er bestimmt die Anzahl der Kommastellen.
Scale kann die Genauigkeit negativ beeinflussen:

```
create table meineTabelle
 (
    meineSpalte float (51),
    meineSpalte2 float (51,2)
 );

insert into  meineTabelle values (123467890.12345, 123467890.12345);
```

Ergibt als Output:

```
mysql> select * from meinetabelle;
+-----------------+--------------+
| meineSpalte1    | meineSpalte2 |
+-----------------+--------------+
| 123467890.12345 | 123467888.00 |
+-----------------+--------------+
```

Bei der Anzeige werden fehlende Nachkommastellen mit 0 aufgefüllt.

SERIAL

Beim Datentyp `SERIAL` erzeugt postgreSQL eine **Sequenz**.

	Byte-Größe	DB	Wertebereich
SERIAL	8 Byte	MySQL:	Syntax Zucker für: BIGINT UNSIGNED NOT NULL AUTO_INCREMENT UNIQUE
	4 Byte	postgreSQL:	Syntax Zucker für eine SEQUENCE (auto-increment Integer)
SMALLSERIAL	2 Byte	postgreSQL:	Syntax Zucker für eine SEQUENCE (auto-increment Integer)
BIGSERIAL	8 Byte	postgreSQL:	Syntax Zucker für eine SEQUENCE (auto-increment Integer)

Beispiel:

```
CREATE TABLE meineTabelle (
      meineSpalte SERIAL
    );
```

Ist gleichzusetzen mit:

```
CREATE SEQUENCE meineSequenz;

CREATE TABLE meineTabelle (
      meineSpalte integer NOT NULL UNIQUE PRIMARY KEY DEFAULT nextval('meineSequenz')
    );

ALTER SEQUENCE meineSequenz OWNED BY meineTabelle.meineSpalte;
```

MONEY

Der Datentyp MONEY ist eine **Fixed-Point-Number**, die bei der Ausgabe optisch formatiert wird (Anzahl Kommastellen, Währungszeichen und Tausendertrennzeichen). Die Art der Formatierung hängt bei postgreSQL von lc_monetary ab.

postgreSQL-Default für die Anzahl der Kommazahlen: 2
SQL-Server-Default für die Anzahl der Kommazahlen: 4

Typen für Money (Geld):

	Byte-Größe	DB	Wertebereich	
MONEY	8 Byte*	T-SQL:	-922,337,203,685,477.5808	922,337,203,685,477.5807
		postgreSQL:	-92233720368547758.08	92233720368547758.07
SMALMONEY	4 Byte	T-SQL::	-214,748.3648	214,748.3647

* 1 Byte Ersparnis zu decimal(19,4): -999,999,999,999,999.9999 bis 999,999,999,999,999.9999

Vor dem Eintrag wird der Wert auf 2 Kommastellen gerundet.

Beispiel:

```
create table meineTabelle (meineSpalte MONEY );
insert into  meineTabelle (meineSpalte) values (1234567890.1234);
```

Ergibt als Output:

```
postgres=# select  sum(meineSpalte)  from meineTabelle;
        sum
--------------------
 1.234.567.890,12 €
```

Fehlende Nachkommastellen werden mit Nullen aufgefüllt.

Beispiel:

```
create table meineTabelle (meineSpalte MONEY );
insert into  meineTabelle (meineSpalte) values (1234);
```

Ergibt als Output:

```
wilma=# select * from meineTabelle;
 meinespalte
-------------
   1.234,00 €
```

Der Versuch, einen zu großen Wert einzutragen, erzeugt einen Fehler.

Beispiel:

```
create table meineTabelle (meineSpalte MONEY );
insert into  meineTabelle (meineSpalte) values (99999999999999999999);
```

Ergibt als Output:

```
FEHLER:  bigint ist außerhalb des gültigen Bereichs
```

SELECT Blöcke

Das einfachste `select`-Query besteht nur aus einem einzigen Kernblock (ohne Nebenblöcken). Komplexere Queries bestehen aus mehreren Kernblöcken, Nebenblöcken und einem Endblock.

Kernblock

SQL-Notation Seite 509.

Kernblock-Syntax einer `select`-Query:

```
SELECT    [ ALL | DISTINCT ]     Spaltenauswahlliste
   FROM Tabellenliste
      [ WHERE Bedingung ]
      [ GROUP BY Spaltenliste    [ HAVING Bedingung ]    ]
```

Spaltenauswahlliste:

Die Spaltenauswahlliste besteht aus mindestens einem Spaltenausdruck (Notation 509):

```
Spaltenausdruck [ [ AS ] Aliasname ] [ , ... ]
```

Der **Spaltenausdruck** muss nicht zwinged eine Spaltenbezeichnung sein, sondern kann auch ein **Literal** (Seite 516) oder eine **Expression** (Seite 524) sein.

Beispiel 1: `666` ist ein **Integer-Literal**, `"teuflische zahl"` ein **String-Literal**

```
select 666 as "teuflische zahl";
```

Ergibt als Output:

```
teuflische zahl
-----------------
              666
```

Beispiel 2: String-Literal mit einer Zahl konkatenieren (`4*3` ist eine Expression, die `12` ergibt)

```
select 'Ergebnis von 4*3 ist ' || 4*3;
```

Ergibt als Output:

```
        ?column?
-------------------------
 Ergebnis von 4*3 ist 12
(1 Zeile)
```

Beispiel 3: Spalten `foo1` und `foo2` von der Tabelle foobar

```
select foo1, foo2 from foobar;
```

Beispiel 4: Spalte `foo1` als Alias `'Höhe'` und `foo2` als Alias `'Breite'` von der Tabelle foobar

```
select foo1, foo2 from foobar;
```

Tabellenliste

Die Tabellenliste besteht aus mindestens einer Tabellenreferenz.

```
Tabellenreferenz [, …]
```

Die Tabellenreferenz ist entweder:

1) Tabellenname
2) Joinausdruck
3) wiederum eine Tabellenreferenz
4) ein **Kernblock**

Die Syntax der Tabellenreferenz (Notation 509):

```
Tabellenname [ [ AS ] Aliasname ]
   | ( Kernblock ) [ [ AS ] Aliasname ]
   | ( Tabellenreferenz ) [ [ AS ] Aliasname ]
   | Joinausdruck [ [ AS ] Aliasname ]
```

Joinausdruck

Bei einem **Join-Ausdruck** wird ein Join mit einem Join-Operator erzeugt.

Bei einem Join (Verbund) wird aus zwei Tabellen ein **Kartesisches Produkt** (Cartesian Product, Seite 562).

Arten von Join-Operatoren (Operatoren Seite 541):

1. `CROSS JOIN` Seite 562
2. `INNER JOIN` Seite 565
3. `OUTER JOIN` Seite 567
4. `NATURAL JOIN` Seite 570

Neben-Block

In einem Nebenblock befindet sich immer ein Kernblock. Nebenblöcke können öfters angegeben werden.

Syntax eines Nebenblockes (Notation siehe Seite 509):

```
[    { UNION [ ALL ] | EXCEPT |  INTERSECT }   Kernblock ]   [ ... ]
```

End-Block

Der Endblock besteht aus der `order by` Anweisung.

Syntax vom Endblock:

```
[ ORDER BY Ordnungsliste ]
```

Operatoren

String Konkatenation

Verkettungsoperator für Zeichenketten (Strings) sind zwei **Vertical Lines** (ASCII 127). Der Operator wird auch **Double Pipe** und **Vbar (Vertical Bars)** genannt.

||	DB2, postgreSQL, Informix, Oracle Bei MySQL muss pipes_as_concat aktiviert sein.
+	T-SQL

Beispiel1: die String-Literale `'Hallo'` und `' Carol'` verknüpfen

```
select 'Hallo'  ||  ' Carol';
```

Ergibt als Output:

```
Hallo Carol
```

Vergleichsoperatoren

Bei Vergleichsoperatoren (Comparison Operators) unterscheidet man zwischen

1) Vergleich von **zwei Skalaren/Literalen**

```
5 < 10
```

2) Vergleich von **einem Skalar/Literal** mit **Werten aus einer Menge**

```
3 IN (2, 3, 4)
```

Vergleiche Skalar mit Skalar

Operator			postgresql	T-SQL	DB2	MySQL	Oracle	Firebird
<>	545	ungleich	ja	ja	ja	ja	ja	ja
!=	545	ungleich	ja	ja	ja	ja	ja	ja
^=	545	ungleich	nein	nein	ja	nein	ja	ja
!<	544	nicht kleiner als	nein	ja	ja	nein	nein	ja
!>	544	nicht größer als	nein	ja	ja	nein	nein	ja
=	543	gleich						
>	544	größer						
<	544	kleiner						
>=	543	größer gleich						
<=	543	kleiner gleich						

Vergleiche Skalar mit Array/Liste/Relation

Operator		Menge	
IN	545	kommaseparierte Liste einspaltige Relation	`3 IN (2, 3, 4)`
ANY		Array einspaltige Relation	`1988 > ANY (array[1985,1999])`
ALL		Array einspaltige Relation	`1988 > ALL (array[1987,1999])`

Gleichheit (Equality)

Der **Equal Operator** gibt `true` zurück, wenn die zwei einzelnen Operanden den gleichen Wert besitzen.

Equal Sign (Gleichheitszeichen) ist: **=**

Syntax: *Operand1* **=** *Operand2*

Tabelle "werte"

g1	g2	w1	w2
a	w	4	2
b	w	3	4
b	z	0	9
b	z	5	3
c	x	7	0
c	x	8	7
c	y	1	3
c	y	7	0

Beispiel 1:

```
select w2 from werte where w2=3;
```

Ergibt als Output:

```
 w2
----
  3
  3
```

Beispiel 2: die Kombination mit dem **kleiner Operator** **<** ergibt eine *kleiner-gleich*-Abfrage

```
select w2 from werte where w2<=2;
```

Ergibt als Output:

```
 g1 | g2 | w1 | w2
----+----+----+----
 a  | w  |  4 |  2
 c  | x  |    |  0
 c  | y  |    |  0
```

Beispiel 3: Kombination mit dem **größer Operator** **>** ergibt eine *größer-gleich*-Abfrage

Tabelle "werte"

g1	g2	w1	w2
a	w	4	2
b	w	3	4
b	z	0	9
b	z	5	3
c	x	7	0
c	x	8	7
c	y	1	3
c	y	7	0

```
select w2 from werte where w2 >= 7;
```

Ergibt als Output:

```
 w2
----
  9
  7
```

Beispiel 4: die Kombination mit dem **Negationsoperator** **!** ergibt eine *Ist -ungleich*-Abfrage

```
select * from werte where g1!='b' and g1!='c';
```

Ergibt als Output:

```
 g1 | g2 | w1 | w2
----+----+----+----
 a  | w  |  4 |  2
```

kleiner (less)

Der **Less Operator** gibt `true` zurück, wenn der linke Operand kleiner als der rechte Operand ist.

Die Kombination mit dem Gleichheitsoperator ergibt eine *kleiner-gleich*-Abfrage.

Syntax: *Operand1* **<** *Operand2*

Tabelle "werte"

g1	g2	w1	w2
a	w	4	2
b	w	3	4
b	z	0	9
b	z	5	3
c	x	7	0
c	x	8	7
c	y	1	3
c	y	7	0

Beispiel 1:

```
select w2 from werte where w2 < 3;
```

Ergibt als Output:

```
 w2
----
  2
  0
  0
```

Negationsoperator

Die Variante mit dem **Negationsoperator** ! kennt nur **DB2** und **T-SQL**.

```
select w2 from werte where w2 !< 7;       -- entspricht w2>=7
```

Ergibt als Output:

```
 w2
----
  9
  7
```

größer (greater)

Der **Greater Operator** gibt `true` zurück, wenn der linke Operand größer als der rechte Operand ist.

Die Kombination mit dem Gleichheitsoperator ergibt eine *größer-gleich*-Abfrage.

Syntax: ***Operand1*** **>** ***Operand2***

Tabelle "werte"

g1	g2	w1	w2
a	w	4	2
b	w	3	4
b	z	0	9
b	z	5	3
c	x	7	0
c	x	8	7
c	y	1	3
c	y	7	0

Beispiel 1:

```
select w2 from werte where w2>6;
```

Ergibt als Output:

```
9
7
```

Negationsoperator

Die Variante mit dem Negationsoperator ! kennt nur **DB2** und **T-SQL**.

```
select w2 from werte where w2 !> 2;       -- entspricht w2<=2
```

Ergibt als Output:

```
2
0
0
```

Ungleichheit (non-equality)

Der **Ungleich-Operator** gibt `true` zurück, wenn der linke Operand, nicht den gleichen Wert wie der rechte Operand besitzt. Die Variante mit dem Negationsoperator wird von allen modernen Datenbanken unterstützt.

ANSI/ISO-SQL: *Operand1* **<>** *Operand2*
Syntax-Variante: *Operand1* **!=** *Operand2*
DB2/Oracle-Syntax: *Operand1* **^=** *Operand2*

			postgresql	T-SQL	DB2	MySQL	Oracle	Firebird
<>		ungleich	ja	ja	ja	ja	ja	ja
!=		ungleich	ja	ja	ja	ja	ja	ja
^=		ungleich	nein	nein	ja	nein	ja	ja

Beispiel 1: die Variante mit dem Negationsoperator !

```
select g1 from werte where g1!='b' and g1!='c';
```

Ergibt als Output:

```
 g1
----
 a
```

Beispiel 2: die ANSI/ISO-Variante <>

```
select g1 from werte where g1<>'b' and g1<>'c';
```

Ergibt als Output:

```
 g1
----
 a
```

Tabelle "werte"

g1	g2	w1	w2
a	w	4	2
b	w	3	4
b	z	0	9
b	z	5	3
c	x	7	0
c	x	8	7
c	y	1	3
c	y	7	0

IN

Gibt `true` zurück, wenn der linke Operand in der rechten Listen vorkommt.

Syntax: ***Operand*** **IN** (***Menge***)

Es gibt 2 Arten von Mengen:

1) Kommaseparierte **Literale** (Seite 516)
   ```
   3 IN (2, 3, 4)
   ```
2) durch ein Subquery erzeugte **Relation**, die **genau aus einer 1 Spalte** beseht.
   ```
   3 IN (select wert from zahlen)
   ```

Beispiel 1a: Liste aus Literalen: kommt 3 in den Zahlen 2, 3 und 4 vor

```
select 3 IN (2,3,4);
```

Ergibt als **postgreSQL**-Output (Query erzeugt in **DB2** und **T-SQL** einen Error):

```
t
```

Ergibt als **MySQL**-Output:

```
1
```

Beispiel 1B: Liste aus Literalen: kommt 3 in den Zahlen 2,3 und 4 vor

```
select 3 IN (2, 4);
```

Ergibt als **postgreSQL**-Output(Query erzeugt in **DB2** und **T-SQL** einen Error):

```
f
```

Ergibt als **MySQL**-Output:

```
0
```

Beispiel 2a: Liste aus Literalen: Comics auf dem Jahr 1955 und 1969

```
select * from comics  where jahr IN (1955, 1969);
```

Ergibt als Output:

```
Superman            | 1969
Donald              | 1955
Donald              | 1969
Garfield            | 1955
```

Tabelle "comics"

comic	jahr
Spiderman	1985
Batman	1993
Batman	1985
Lupo	1978
Superman	**1969**
Lupo	1985
Donald	**1955**
Donald	**1969**
Spiderman	1978
Garfield	**1955**

Beispiel 2b: Liste aus Subquery: Comics, mit Jahreszahlen von Donald

```
select comic, jahr from comics
   where jahr IN (select jahr from comics where comic='Donald');
```

Ergibt als Output:

```
Superman            | 1969
Donald              | 1955
Donald              | 1969
Garfield            | 1955
```

ANY

Gibt `true` zurück, wenn der Vergleich mit **mindestens einem Wert aus der Menge** `true` ergibt.

Syntax: ***Operand*** ***Vergleichsoperator*** **ANY** (***Menge***)

Es gibt 2 Arten von Mengen:

1) **Array**:

```
3 >= ANY  (arry[1,2,3])
```

2) durch eine Subquery erzeugte **Relation**, die aus **genau einer 1 Spalte** beseht.

```
3 >= ANY (select wert from zahlen)
```

Beispiel 1: kommt 3 im Array vor

```
select 3 = ANY (array[2,3,4]);
```

Ergibt als **postgreSQL**-Output (Query erzeugt in **DB2, MySQL** und **T-SQL** einen Error):

```
t
```

Beispiel 1b: gibt es eine Array-Zahl, die größer als 7 ist

```
select 7 > ANY (array[6,9,11]);
```

Ergibt als **postgreSQL**-Output (Query erzeugt in **DB2** und **T-SQL** einen Error):

```
t
```

Beispiel 2: Comics, die jünger sind, als eines der Lupo-Comics

```
select comic, jahr from comics
where jahr > ANY
   (select jahr from comics where comic='Lupo');
```

Ergibt als Output:

```
          comic          | jahr
-------------------------+------
 Spiderman               | 1985
 Batman                  | 1993
 Batman                  | 1985
 Lupo                    | 1978
 Lupo                    | 1985
 Spiderman               | 1978
```

Tabelle "comics"

```
          comic          | jahr
-------------------------+------
 Spiderman               | 1985
 Batman                  | 1993
 Batman                  | 1985
 Lupo                    | 1978
 Superman                | 1969
 Lupo                    | 1985
 Donald                  | 1955
 Donald                  | 1969
 Spiderman               | 1978
 Garfield                | 1955
```

ALL

Gibt `true` zurück, wenn der Vergleich mit **allen Werten aus der Menge** `true` ergibt.

Syntax: ***Operand*** ***Vergleichsoperator*** **ALL** (***Menge***)

Es gibt 2 Arten von Mengen:

1) **Array**:

```
2 >= ALL  (arry[1,2,3])
```

2) durch eine Subquery erzeugte **Relation**, die aus **genau einer 1 Spalte** beseht.

```
3 >= ALL (select wert from zahlen)
```

Beispiel 1: hat jedes Arrayfeld den Wert 3

```
select 3 = ALL (array[2,3,4]);
```

Ergibt als **postgreSQL**-Output (Query erzeugt in **DB2, MySQL** und **T-SQL** einen Error):

```
f
```

Beispiel 2: Comics, die jünger sind, als alle Lupo-Comics

```
select comic, jahr from comics
   where jahr > ALL
      (select jahr from comics where comic='Lupo');
```

Ergibt als Output:

```
          comic          | jahr
-------------------------+------
 Batman                  | 1993
```

Tabelle "comics"

```
          comic          | jahr
-------------------------+------
 Spiderman               | 1985
 Batman                  | 1993
 Batman                  | 1985
 Lupo                    | 1978
 Superman                | 1969
 Lupo                    | 1985
 Donald                  | 1955
 Donald                  | 1969
 Spiderman               | 1978
 Garfield                | 1955
```

Präzedenz (Precedence)

Die Operator-Präzedens bestimmt die Priorität der Verarbeitungsreihenfolge. Die Präzedens 1 hat die geringste Priorität. Im Beispiel ergibt die Expression 9/3-2 den Wert 1:

```
select 9/3-2;
```

Zuerst wird die Division durchgeführt (Präzedenz 9), danach die Subtraktion (Präzedenz 8).
Ergibt als Output:

```
 ?column?
----------
        1
```

14	.	Namenseparator (Tabelle.Spalte)	left-to-right	… . …
13	::	Datentyp-Konvertierung	left-to-right	… :: …
12	[ ]	Array-Klammern	left-to-right	[…]
11	+	unäres Plus (+ Vorzeichen)	right-to-left	+ …
	-	unäres Minus (- Vorzeichen)	right-to-left	- …
10	^	Potenz	left-to-right	… ^ …
9	*	Multiplikation	left-to-right	… * …
	/	Division	left-to-right	… / …
	%	Rest einer Division	left-to-right	… % …
8	+		left-to-right	… + …
	-		left-to-right	… - …
7	sonstige, hier nicht erwähnte Operatoren			
6	IN			
	BETWEEN			
	LIKE			
	ILIKE			
	SIMILAR			
5	<	kleiner		… < …
	<=	kleiner gleich		… <= …
	>	größer		… > …
	>=	größer gleich		… >= …
	<>	ungleich		… <> …
	!=	ungleich		… != …
	=	gleich		… = …
4	IS	ist		
	ISNULL	ist NULL		
	NOTNULL	ist nicht NULL		
3	NOT	logisches Nicht	right-to-left	… NOT …
2	AND	logisches Und	left-to-right	… AND …
1	OR	logisches Oder	left-to-right	… OR …

Ergebnislimitierung

		postgreSQL	T-SQL	DB2	MySQL
TOP ***N***	zeige die ersten ***N*** Zeilen	nein	ja	nein	nein
FETCH FIRST ROWS ***N*** **ONLY**	zeige die ersten ***N*** Zeilen *	ja	nein	ja	nein
OFFSET ***N*** **ROWS**	die ersten ***N*** Zeilen überspringen	ja	ja	nein	nein
LIMIT ***N***	die ersten ***N*** Zeilen anzeigen	ja	nein	ja	ja
LIMIT ***N*** **OFFSET** ***X***	die ersten ***X*** Zeilen überspringen danach die ersten ***N*** Zeilen anzeigen	ja	nein	ja	ja

* ISO-SQL:2008 `FETCH FIRST`

top

Das Keyword `top` wird nur von T-SQL unterstützt und steht **vor der Spaltenauswahlliste** (Seite 539).

Der Parameter bestimmt die Zeilenanzahl.

Syntax:

```
TOP Zeilenanzahl   Spaltenauswahliste
```

Zeilenanzahl muss >= 0 sein.

Die Angabe der Spaltenauswahlliste ist verpflichtend.

Beispiel: die 2 ersten Zeilen anzeigen

```
select  TOP 2  comic, id  FROM comics ORDER BY id DESC;
```

Ergibt als **T-SQL** Output:

```
        comic          | jahr
-----------------------+------
 Garfield              | 10
 Spiderman             | 9
```

Tabelle "comics"

```
        comic          | jahr
-----------------------+------
 Spiderman             | 1985
 Batman                | 1993
 Batman                | 1985
 Lupo                  | 1978
 Superman              | 1969
 Lupo                  | 1985
 Donald                | 1955
 Donald                | 1969
 Spiderman             | 1978
 Garfield              | 1955
```

set rowcount

Die Anzahl der Ergebniszeilen kann mit `set rowcount` eingeschränkt werden.

Beispiel: die 2 ersten Zeilen anzeigen

```
set rowcount 2;
select  comic, id  FROM comics ORDER BY id DESC;
```

Ergibt als Output:

```
        comic          | jahr
-----------------------+------
 Garfield              | 10
 Spiderman             | 9
```

fetch first rows only

Die Keyword-Phrase wird postgreSQL und DB2 unterstützt und steht am Ende eines Statements.

Der Parameter bestimmt die Zeilenanzahl.

Syntax:

```
FETCH FIRST ROWS Zeilenanzahl ONLY
```

Zeilenanzahl muss >= 0 sein.

Beispiel: die 2 ersten Zeilen anzeigen

```
select  jahr  FROM comics  group by  jahr  ORDER BY jahr DESC  fetch first 2 rows only  ;
```

Ergibt als **DB2** Output:

```
        comic          | jahr
-----------------------+------
 Garfield              | 10
 Spiderman             | 9
```

offset rows

Das Keyword-Paar `OFFSET ROWS` wird von **postgreSQL** und **T-SQL** unterstützt (nicht von **MySQL** und **DB2**).

Der Parameter bestimmt die Anzahl der **ersten Zeilen, die nicht angezeigt werden**.

Syntax:

```
OFFSET  Zeilenanzahl ROWS
```

Zeilenanzahl muss >= 0 sein.
Die Angabe des Keyword `ROWS` ist bei **postgreSQL** optional.

Beispiel: zeige die ersten 7 Zeilen nicht an

```
select  comic, id from comics order by id OFFSET 7 ROWS;
```

Ergibt als Output:

```
        comic          | id
-----------------------+----
 Donald                |  8
 Spiderman             |  9
 Garfield              | 10
```

limit

Das Keyword `LIMIT` wird von **postgreSQL, DB2** und **MySQL** unterstützt (nicht von **T-SQL**).

Der Parameter bestimmt die Anzahl der **ersten Zeilen, die angezeigt werden sollen**.

Syntax:

```
LIMIT  Zeilenanzahl
```

Zeilenanzahl muss >= 0 sein.

Beispiel: zeige die ersten 2 Zeilen an

```
select  comic, id from comics order by id DESC LIMIT 2 ;
```

Ergibt als Output:

```
        comic          | jahr
-----------------------+------
 Garfield              | 10
 Spiderman             | 9
```

Bei **DB2** muss `DB2_COMPATIBILITY_VECTOR=MYS` gesetzt sein:

```
db2stop
db2set DB2_COMPATIBILITY_VECTOR=MYS
db2start
```

limit offset

Das Keyword-Paar `LIMIT OFFSET` wird von **postgreSQL, DB2** und **MySQL** unterstützt (nicht von **T-SQL**).

Der 2. Parameter bestimmt die Anzahl der **ersten Zeilen, die übersprungen werden sollen**.

Der 1. Parameter bestimmt die Anzahl der **ersten Zeilen, die nach dem Sprung angezeigt werden sollen**.

Syntax:

```
LIMIT  Zeilenanzahl  OFFSET  Offsetzeilenanzahl
```

Zeilenanzahl und ***Offsetzeilenanzahl*** müssen >= 0 sein.

Beispiel: überspringen die ersten 8 Zeilen und zeige danach die ersten 6 Zeilen an

```
select  comic, id from comics order by id DESC LIMIT 6 OFFSET 8;
```

Ergibt als Output:

```
        comic          | id
-----------------------+----
 Batman                |  2
 Spiderman             |  1
```

Tabelle "comics"

```
        comic          | jahr
-----------------------+------
 Spiderman             | 1985
 Batman                | 1993
 Batman                | 1985
 Lupo                  | 1978
 Superman              | 1969
 Lupo                  | 1985
 Donald                | 1955
 Donald                | 1969
 Spiderman             | 1978
 Garfield              | 1955
```

MySQL, DB2 Syntax-Variante

Syntax:

```
LIMIT  Offsetzeilenanzahl , Zeilenanzahl
```

Zeilenanzahl und ***Offsetzeilenanzahl*** müssen >= 0 sein.

Beispiel: überspringe die ersten 8 Zeilen und zeige danach die ersten 6 Zeilen an

```
select  comic, id from comics order by id DESC LIMIT 8, 6 ;
```

Ergibt als Output:

```
         comic          | id
------------------------+----
 Batman                 |  2
 Spiderman              |  1
```

Tabelle "comics"

comic	jahr
Spiderman	1985
Batman	1993
Batman	1985
Lupo	1978
Superman	**1969**
Lupo	1985
Donald	**1955**
Donald	**1969**
Spiderman	1978
Garfield	**1955**

Bei **DB2** muss `DB2_COMPATIBILITY_VECTOR=MYS` gesetzt sein:

```
db2stop
db2set DB2_COMPATIBILITY_VECTOR=MYS
db2start
```

Case When Then

CASE WHEN THEN ist eine generische Expression, mit der mehrere Expressions der Reihe nach überprüft werden. Der CASE-Ausdruck kann an jeder Stelle verwendet werden, an der auch eine normale Expression stehen kann.

CASE-Anweisungen lassen sich verschachteln (**Nested CASE** siehe Seite 557).

Die ELSE Angabe ist optional. Tritt zur Laufzeit eine Situation ein, in der keine WHEN-Expression übereinstimmt, ist der Rückgabewert NULL. In der **Stored Procedure** Variante wird das fehlende ELSE als kritischer Fehler gewertet.

In der **Stored Procedure** Variante wird mit END CASE beendet. Siehe Seite 592.

Die when-Bedingung kann auch eine **Logic Expression** sein. ZB EXISTS Seite 557 und IN Seite 557.

Syntax-Variante 1

Die jeweilige **WHEN**-Expression wird auf **true** überprüft. Nach dem ersten Treffer werden die folgenden (restlichen) WHEN-Überprüfungen übersprungen.

Syntax:

```
CASE
   WHEN Expression
      THEN
         Anweisungen
    [...]

   [ ELSE
      Anweisungen ]
END
```

Syntax-Variante 2

Bei der switch Variante wird eine **CASE**-Expression mit (fast) beliebig vielen WHEN-Expressions **auf Gleichheit überprüft.**

Syntax:

```
CASE Expression

   WHEN Expression
      THEN
         Anweisungen
    [...]

   [ ELSE
      Anweisungen ]

END CASE;
```

Beispiele

Beispiel 1: Projektion: den Wert 4 durch 'vier' und 9 durch 'neun' ersetzen

```
select w2, CASE
             WHEN w2=7 THEN 'sieben'
             WHEN w2=9 THEN 'neun'
           END
 from werte where w2>3;
```

Ergibt als Output:

```
 w2 |  case
----+--------
  4 |
  9 | neun
  7 | sieben
```

Tabelle "werte"

```
 g1 | g2 | w1 | w2
----+----+----+----
 a  | w  |  4 |  2
 b  | w  |  3 |  4
 b  | z  |  0 |  9
 b  | z  |  5 |  3
 c  | x  |    |  0
 c  | x  |  8 |  7
 c  | y  |  1 |  3
 c  | y  |    |  0
```

Beispiel 2a: Projektion: einen Spaltenwert unterschiedlich berechnen

```
select w2, CASE
             WHEN w2>7 THEN w2*100
             WHEN w2>5 THEN w2*1000
           END
 from werte where w2>3;
```

Ergibt als Output:

```
 w2 | case
----+------
  4 |
  9 |  900
  7 | 7000
```

Beispiel 2b: Projektion: diese Syntax-Variante funktioniert nur bei **T-SQL** (nicht bei **MySQL, DB2, postgreSQL**)

```
select w2, foo =  CASE
       WHEN w2>7 THEN w2*100
       WHEN w2>5 THEN w2*1000
    END
 from werte where w2>3;
```

Ergibt als **T-SQL** Output:

```
 w2 | foo
----+------
  4 | NULL
  9 | 900
  7 | 7000
```

Beispiel 3: Update-Wert hängt von CASE ab

```
update werte set g1 =  CASE g1
                          WHEN 'a' THEN '1'
                          WHEN 'b' THEN '2'
                          ELSE '3'
                       END;

select g1 from werte group by g1;
```

Ergibt als Output:

```
 g1
----
 2
 3
 1
```

Beispiel 4: where-Klausel hängt von CASE ab

```
select g1,w1 from werte where w2 =
    CASE g1
       WHEN 'a' THEN 2
       WHEN 'b' THEN 4
      ELSE 4
    END;
```

Ergibt als Output:

```
 g1 | w1
----+----
 b  |  3
 a  |  4
```

Tabelle "werte"

g1	g2	w1	w2
a	w	4	2
b	w	3	4
b	z	0	9
b	z	5	3
c	x		0
c	x	8	7
c	y	1	3
c	y		0

Beispiel 5: Verschachtelte CASE-Abfragen (Nested CASE)

Tabelle "werte"

g1	g2	w1	w2
a	w	4	2
b	w	3	4
b	z	0	9
b	z	5	3
c	x		0
c	x	8	7
c	y	1	3
c	y		0

```
select g1,w1 from werte where w2 =
    CASE g1
       WHEN 'a' THEN 2
       WHEN 'b'
          THEN
             CASE
                WHEN g2='z' THEN 9
                WHEN w1=4 THEN 5
              END
    END ;
```

Ergibt als Output:

```
 g1 | w1
----+----
 a  |  4
 b  |  0
```

Beispiel 6: WHEN-Expression mit EXISTS

Tabelle "links"

wert1	id1
a	1
f	6
a	2

Tabelle "rechts"

wert3	id3
a	1
a	26
y	28

```
select wert1,id1,
   CASE
   WHEN EXISTS (select 1 from rechts where wert3=wert1)
         THEN 'in rechts vorhanden'
    ELSE 'in rechts nicht vorhanden'
   END
from links;
```

Ergibt als Output:

```
 wert1 | id1 |           case
-------+-----+--------------------------
 a     |   1 | in rechts vorhanden
 f     |   6 | in rechts nicht vorhanden
 a     |   2 | in rechts vorhanden
```

Beispiel 6: WHEN-Expression mit IN

```
select wert1,id1,
   CASE
   WHEN wert1 IN ('b','c','f')
      THEN 'ist b, c oder f'
    ELSE 'weder b, c noch f'
   END
from links;
```

Ergibt als Output:

```
 wert1 | id1 |       case
-------+-----+------------------
 a     |   1 | weder b, c noch f
 f     |   6 | ist b, c oder f
 a     |   2 | weder b, c noch f
```

IS NULL

Der Operator `IS NULL` überprüft, ob der Operand den Wert `NULL` besitzt.

`NULL` ist vom Datentyp `NULL` und der einzige möglich Wert von diesem Datentyp.

Verbund (Join)

Join Operatoren

Bei jeder Art von Join (Verbund) wird ein **Kreuzprodukt** von 2 Tabellen gebildet. Bei einem **Self-Join** verbindet sich eine Tabelle mit sich selbst.

Welche Zeilen der beiden Tabellen dafür verwendet werden, hängt ab von:

1) **Art des Join (bzw Join-Operator)**

CROSS JOIN	562	Ein **Bulk Cross Join** Join (ein unbedingter Cross Join wird **Bulk Cross Join** genannt). **DB2** und **MySQL** erlauben die Verwendung von `on`, **postgreSQL** und **T-SQL** nicht.
INNER JOIN	565	Mit `ON` bedingter Cross Join. Nur bei **MySQL** ist die Angabe von `ON` beim `INNER JOIN` nicht zwingend.
OUTER JOIN	567	Ein Inner Join mit zusätzlich allen rausgefilterten Zeilen. Die Angabe von `ON` beim `OUTER JOIN` zwingend notwendig. Bei **postgreSQL** muss beim `FULL OUTER JOIN` die Join-Bedingung zwingend ein **Equi-Join** sein. T-SQL kannte die Outer-Join Operatoren `*=` und `*=`
NATURAL JOIN	570	Eine Join-Bedingung, die Gleichheit von Spaltennamen und Spaltenwert verlangt (und somit immer ein Equi-Join ist). Nur postgreSQL hat volle Natural Join Unterstützung (MySQL kennt FULL OUTER nicht). **T-SQL** kennt `NATURAL JOIN` nicht, DB2 nur in Verbindung mit ON (und führt sich als normalen Join aus).

2) **on** oder **using** als explizite Join-Bedingung
DB2 und **T-SQL** kennen `USING` nicht.

Da SQL-89 noch keine Join-Operatoren kennt, muss der Inner Join mit der `where`-Klausel gebildet werden.

Theta-Join

Bei einem **Theta-Join** (θ-Join) wird ein Spaltenwert der 1. Tabelle, mit einem Spaltenwert der 2. Tabelle verglichen.
Ist der Vergleichsoperator nur das Gleichheitszeichen, ist der **Theta-Join** ein **Equi-Join** (ansonsten ein **Range-Join**).

```
select * from links LEFT OUTER JOIN rechts   ON    wert1<=wert2;
```

Equi-Join

Der **Equi-Join** ist ein Theta-Join, bei dem ein Spaltenwert der 1. Tabelle, auf Gleichheit mit einem Spaltenwert der 2. Tabelle verglichen wird. Ein **Natural Join** ist immer ein **Equi-Join**. Ein **Bulk Cross Join** ist nie ein **Equi-Join** (da bedingungslos).

```
select * from links FULL OUTER JOIN rechts   ON    links.id1=9 and rechts.id2=9;
```

Non-Theta-Join

Der Join-Filter muss nicht zwingend auf beide Tabellen zielen oder 2 Spaltenwerte vergleichen.
Zum Beispiel ein Filter nur auf die erste Tabelle:

```
select * from links FULL OUTER JOIN rechts   ON    id1=9;
```

Oracle, T-SQL, DB2, MySQL, postgreSQL

MySQL kennt `FULL OUTER JOIN` nicht.

T-SQL kennt `NATURAL JOIN` nicht.

Bei **postgreSQL** muss beim `FULL OUTER JOIN` die Join-Bedingung zwingend ein **Equi-Join** * sein.

DB2 akzeptiert die `NATURAL JOIN` Syntax nur in Verbindung mit **`ON`** (so wie immer) und führt sie als gewöhnliche Joins aus.

		postgreSQL	T-SQL	DB2	MySQL	Oracle
CROSS JOIN	ohne ON	ja	ja	ja	ja	ja
	mit ON	nein	nein	ja	ja	nein
INNER JOIN	ohne ON	nein	nein	nein	ja	nein
	mit ON	ja	ja	ja	ja	ja
LEFT/RIGHT OUTER JOIN	ohne ON	nein	nein	nein	nein	nein
	mit ON	ja	ja	ja	ja	ja
FULL OUTER JOIN	ohne ON	nein	nein	nein	nein	nein
	mit ON	ja*	ja	ja	nein	ja
NATURAL INNER JOIN	ohne ON	ja	nein	nein	ja	
	mit ON	nein	nein	ja**	nein	
NATURAL FULL OUTER JOIN	ohne ON	ja	nein	nein	nein	
	mit ON	nein	nein	ja**	nein	
NATURAL LEFT/RIGHT OUTER JOIN	ohne ON	ja	nein	nein	ja	
	mit ON	nein	nein	ja**	nein	

Cross Join (Kartesisches Produkt)

Bei einem Kartesischem Produkt (**Kreuzprodukt**) wird die 1. Zeile der ersten Tabelle, mit jeder Zeile der zweiten Tabelle verbunden. Danach wird die 2. Zeile der ersten Tabelle, mit jeder Zeile der zweiten Tabelle verbunden. Und so fort.

Somit wird **jeder einzelne** Datensatz der ersten Tabelle, immer mit **allen Datensätzen** der zweiten Tabelle zusammengeführt. Das Kreuzprodukt wird auch **Kreuzverknüpfung** genannt.

Ein **Cross Join** ohne Verknüpfungsbedingung wird **Bulk Cross Join** genannt.
Cross Joins mit Verknüpfungsbedingung sind **Inner Joins** und **Outer Joins**.

		postgreSQL	T-SQL	DB2	MySQL	Oracle
CROSS JOIN	ohne ON	ja	ja	ja	ja	ja
	mit ON	nein	nein	ja	ja	nein

DB2 und **MySQL** erlauben beim CROSS JOIN die Verwendung von ON.
postgreSQL, Oracle und **T-SQL** werten das als Syntax-Fehler.

Bulk Cross Join

Bulk Cross Join:

```
select * from links CROSS JOIN rechts;
```

SQL89-Variante:

```
select * from links, rechts;
```

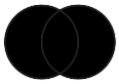


Tabelle "links"	Tabelle "rechts"
wert1 | id1 -------+----- a | 1 f | 6 a | 2	wert3 | id3 -------+----- a | 1 a | 26 y | 28

Ergibt als Output:

```
 wert1 | id1 | wert3 | id3
-------+-----+-------+-----
 a     |   1 | a     |   1
 a     |   1 | a     |  26
 a     |   1 | y     |  28
 f     |   6 | a     |   1
 f     |   6 | a     |  26
 f     |   6 | y     |  28
 a     |   2 | a     |   1
 a     |   2 | a     |  26
 a     |   2 | y     |  28
```

Non-Theta-Join

Non-Theta-Join: der Filter vergleicht nicht die Spaltenwerte direkt miteinander

```
select * from links CROSS JOIN rechts  WHERE wert1='a' and wert3='a';
```

SQL89-Variante:

```
select * from links, rechts WHERE wert1='a' and wert3='a';
```

Ergibt als Output:

```
 wert1 | id1 | wert3 | id3
-------+-----+-------+-----
 a     |   1 | a     |   1
 a     |   1 | a     |  26
 a     |   2 | a     |   1
 a     |   2 | a     |  26
```

Theta Join

Theta-Join: der Filter vergleicht einen Spaltenwert der 1. Tabelle, direkt mit einem Spaltenwert der 2. Tabelle. Der **Range-Join** vergleicht den Werte-Bereich von id1 und id3: id1 <= id3

```
select * from links CROSS JOIN rechts
  WHERE wert1='a' and id1 <= id3;
```

SQL89-Variante:

```
select * from links, rechts where wert1='a' and id1 <= id3;
```

Tabelle "links"	Tabelle "rechts"
wert1 | id1 -------+----- a | 1 f | 6 a | 2	wert3 | id3 -------+----- a | 1 a | 26 y | 28

Ergibt als Output:

```
 wert1 | id1 | wert3 | id3
-------+-----+-------+-----
 a     |   1 | a     |   1
 a     |   1 | a     |  26
 a     |   2 | a     |  26
 a     |   1 | y     |  28
 a     |   2 | y     |  28
```

Equi-Join

Equi-Join: der Filter vergleicht Spaltenwert der 1. Tabelle, mit dem Spaltenwert der 2. Tabelle auf Gleichheit

```
select * from links CROSS JOIN rechts   WHERE id1 = id3;
```

SQL89-Variante:

```
select * from links, rechts where id1 = id3;
```

Ergibt als Output:

```
 wert1 | id1 | wert3 | id3
-------+-----+-------+-----
 a     |   1 | a     |   1
```

Multiple Join

Die Ausführungsrichtung bei einem Join-Operator ist von links nach rechts.
Mit dem Ergebnis von einem Operator wird der nächste Operator ausgeführt.

Da beim Cross Join die `on/using` Bedingung nicht erlaubt ist, wird von allen drei Tabellen ein Cross Join erzeugt und nachträglich mit der `where`-Klausel gefiltert.

Equi-Join: der Filter überprüft auf Gleichheit in allen drei Tabellen

```
select * from links CROSS JOIN mitte CROSS JOIN rechts
   WHERE wert1=wert3 and id2= id3;
```

SQL89-Variante:

```
select * from links, rechts, mitte
   WHERE wert1=wert3 and id2=id3;
```

Tabelle "links"	Tabelle "mitte"	Tabelle "rechts"
wert1 | id1 -------+----- a | 1 f | 6 a | 2	wert2 | id2 -------+----- a | 1 a | 13 x | 15	wert3 | id3 -------+----- a | 1 a | 26 y | 28

Zur Veranschaulichung ein Zwischenergebnis von den Tabellen "links" und "mitte":

```
select * from links CROSS JOIN mitte     WHERE wert1 = wert2;
```

```
 wert1 | id1 | wert2 | id2
-------+-----+-------+-----
 a     |   1 | a     |   1
 a     |   1 | a     |  13
 a     |   2 | a     |   1
 a     |   2 | a     |  13
```

Ergibt als Output:

```
 wert1 | id1 | wert2 | id2 | wert3 | id3
-------+-----+-------+-----+-------+-----
 a     |   1 | a     |   1 | a     |   1
 a     |   2 | a     |   1 | a     |   1
```

Inner Join

Der **Inner Join** ist ein Kreuzprodukt (Seite 562), bei dem vorher die **ON** Expression ausgewertet wurde.

Erst nach der ON-Filterung der beiden Tabellen, wird mit den zwei Filterergebnissen ein **Cross Join** durchgeführt.

Syntax mit dem `INNER JOIN` Operator:

```
Tabellenname1   INNER JOIN   Tabellenname2   { ON | USING }   Expression
```

Der Filter kann durch die **where**-Klausel erweitert werden.

Bei **postgreSQL, Oracle, T-SQL** und **DB2** ist die Angabe von `ON` beim `INNER JOIN` zwingend notwendig (im Gegensatz zu **MySQL**).

		postgreSQL	T-SQL	DB2	MySQL	Oracle
INNER JOIN	ohne ON	nein	nein	nein	ja	nein
	mit ON	ja	ja	ja	ja	ja

Non-Theta Join

Non-Theta-Join: die ON-Bedingung überprüft die Spalten `wert1` und `wert3`

```
select * from links  INNER JOIN rechts on wert1='a' and wert3='a';
```

SQL89-Variante:

```
select * from links, rechts where wert1='a' and wert3='a';
```

Tabelle "links"

```
 wert1 | id1
-------+-----
 a     |   1
 f     |   6
 a     |   2
```

Tabelle "rechts"

```
 wert3 | id3
-------+-----
 a     |   1
 a     |  26
 y     |  28
```

Ergibt als Output:

```
 wert1 | id1 | wert3 | id3
-------+-----+-------+-----
 a     |   1 | a     |   1
 a     |   1 | a     |  26
 a     |   2 | a     |   1
 a     |   2 | a     |  26
```

Theta Join

Non-Theta-Join: die **Range-Bedingung** vergleicht den Werte-Bereich von id1 und id3: id1 <= id3

```
select * from links INNER JOIN rechts on wert1='a' and id1<=id3;
```

SQL89-Variante:

```
select * from links, rechts
   where links.wert1='a' and id1 <= id3;
```

Ergibt als Output:

```
 wert1 | id1 | wert3 | id3
-------+-----+-------+-----
 a     |   1 | a     |   1
 a     |   1 | a     |  26
 a     |   2 | a     |  26
 a     |   1 | y     |  28
 a     |   2 | y     |  28
```

Equi-Join

Equi-Join: die Bedingung vergleicht eine Spalte der 1. Tabelle, mit einer Spalte der 2. Tabelle auf Gleichheit

```
select * from links INNER JOIN rechts ON id1 = id3;
```

SQL89-Variante: `select * from links, rechts where id1 = id3;`

Tabelle "links"	Tabelle "rechts"
wert1 | id1 -------+----- a | 1 f | 6 a | 2	wert3 | id3 -------+----- a | 1 a | 26 y | 28

Ergibt als Output:

```
 wert1 | id1 | wert3 | id3
-------+-----+-------+-----
 a     |   1 | a     |   1
```

Multiple Join

Die Ausführungsrichtung bei einem Join-Operator ist von links nach rechts.
Mit dem Ergebnis von einem Operator wird der nächste Operator ausgeführt.

Beispiel:

```
select * from links
  INNER JOIN mitte ON wert1=wert2
  INNER JOIN rechts ON id3=1;
```

Tabelle "links"	Tabelle "mitte"	Tabelle "rechts"
wert1 | id1 -------+----- a | 1 f | 6 a | 2	wert2 | id2 -------+----- a | 1 a | 13 x | 15	wert3 | id3 -------+----- a | 1 a | 26 y | 28

Zur Veranschaulichung ein Zwischenergebnis von den Tabellen "links" und "mitte":

```
select * from links CROSS JOIN mitte     WHERE wert1 = wert2;
```

```
 wert1 | id1 | wert2 | id2
-------+-----+-------+-----
 a     |   1 | a     |   1
 a     |   1 | a     |  13
 a     |   2 | a     |   1
 a     |   2 | a     |  13
```

Ergibt als Output:

```
 wert1 | id1 | wert2 | id2 | wert3 | id3
-------+-----+-------+-----+-------+-----
 a     |   1 | a     |   1 | a     |   1
 a     |   1 | a     |  13 | a     |   1
 a     |   2 | a     |   1 | a     |   1
 a     |   2 | a     |  13 | a     |   1
```

Outer Join

Der **Outer Join** ist ein **Inner Join** (Seite 565), bei dem zusätzlich noch die rausgefilterten Zeilen von der linken Tabelle (**Left Join**), oder von der rechten Tabelle (**Right Join**) oder von beiden Tabellen (**FULL JOIN**) dazukommen.

Syntax mit dem `OUTER JOIN` Operator:

```
Tabelle1   { LEFT | RIGHT | FULL } [ OUTER ] JOIN   Tabelle2   { ON | USING }   Expression
```

Die Angabe von `ON` ist beim `INNER JOIN` zwingend notwendig.

T-SQL kannte (ab **SQL Server 7** deprecated) als Left-Outer-Operator `*=` und als Right-Outer-Operator `=*`

Alt:
```
SELECT * FROM links, rechts
    WHERE links.id1 *= rechts.id3;
```

Neu:
```
SELECT * FROM links
    LEFT OUTER JOIN rechts  ON links.id1 = rechts.id3;
```

		postgreSQL	T-SQL	DB2	MySQL	Oracle
`LEFT/RIGHT OUTER JOIN`	mit `ON`	ja	ja	ja	ja	ja
`FULL OUTER JOIN`	mit `ON`	ja*	ja	ja	nein	ja
*	Bei **postgreSQL** muss beim `FULL OUTER JOIN` die Join-Bedingung zwingend ein **Equi-Join** sein.					

Left Outer Join

Der **Left Outer Join** ist ein **Inner Join** (Seite 565), bei dem zusätzlich noch alle rausgefilterten Zeilen von der linken Tabelle dazukommen.

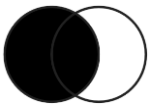
Beispiel:
```
select * from links LEFT OUTER JOIN rechts ON wert1=wert3;
```

Tabelle "links"

```
 wert1 | id1
-------+-----
 a     |   1
 f     |   6
 a     |   2
```

Tabelle "rechts"

```
 wert3 | id3
-------+-----
 a     |   1
 a     |  26
 y     |  28
```



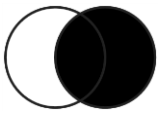
Ergibt als Output:

```
 wert1 | id1 | wert3 | id3
-------+-----+-------+-----
 a     |   1 | a     |   1
 a     |   1 | a     |  26
 a     |   2 | a     |   1
 a     |   2 | a     |  26
 f     |   6 |       |
```

Right Outer Join

Der **RIGHT Outer Join** ist ein **Inner Join** (Seite 565), bei dem zusätzlich noch alle rausgefilterten Zeilen von der rechten Tabelle dazukommen.

Beispiel: `select * from links RIGHT OUTER JOIN rechts ON id1=id3;`



Ergibt als Output:

```
 wert1 | id1 | wert3 | id3
-------+-----+-------+-----
 a     |   1 | a     |   1
       |     | a     |  26
       |     | y     |  28
```

Full Outer Join

Der **FULL Outer Join** ist ein **Inner Join** (Seite 565), bei dem zusätzlich noch alle rausgefilterten Zeilen von der linken Tabelle und alle rausgefilterten Zeilen von der rechten Tabelle dazukommen.

PostgreSQL: Nur beim `FULL OUTER JOIN` muss die Join-Bedingung zwingend ein Equi-Join sein.

Fehler-Meldung: `wird nur für Merge- oder Hash-Verbundfähige Verbundbedingungen unterstützt`

Wobei postgreSQL `on 1=1` erlaubt:

```
select * from links FULL OUTER JOIN rechts ON 1=1;
```

MySQL: keine `FULL OUTER` Unterstützung

Beispiel: Non-Equi-Join (wird von postgreSQL nicht erlaubt)

```
select * from links FULL OUTER JOIN rechts ON id1=1 and id3=1;
```

Tabelle "links"

```
 wert1 | id1
-------+-----
 a     |   1
 f     |   6
 a     |   2
```

Tabelle "rechts"

```
 wert3 | id3
-------+-----
 a     |   1
 a     |  26
 y     |  28
```

DB2 Output:

```
 wert1 | id1 | wert2 | id2
-------+-----+-------+-----
 a     |   1 | a     |   1
       |     | y     |  28
       |     | a     |  26
 f     |   6 |       |
 a     |   2 |       |
```

Beispiel : Equi-Join

```
select * from links FULL OUTER JOIN rechts ON id1=id3;
```

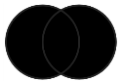


Tabelle "links"	Tabelle "rechts"
wert1 | id1 -------+----- a | 1 f | 6 a | 2	wert3 | id3 -------+----- a | 1 a | 26 y | 28

Output:

```
 wert1 | id1 | wert3 | id3
-------+-----+-------+-----
 a     |   1 | a     |   1
 a     |   2 |       |
 f     |   6 |       |
       |     | a     |  26
       |     | y     |  28
```

ON versus WHERE

Beim `OUTER JOIN` kann es eine Rolle spielen, ob ein Element beim **ON** oder in der **where**-Klausel stehen soll.

Beispiel:

Tabelle "personen"	Tabelle "adressen"
prs_id | stadt_id | vorname | nachname --------+----------+---------+------------- 1 | | Alice | King 2 | 2 | Carol | King 3 | 2 | Anna | Butterblume 4 | 2 | Mike | King	stadt_id | stadt | land ----------+----------+------------- 1 | Madrid | Spanien 2 | München | Deutschland 3 | New York | USA

Der Filter **nachname='King'** in der where-Klausel liefert ein Ergebnis ohne **Anna Butterblume**

```
SELECT vorname, nachname, stadt from personen
   LEFT JOIN adressen ON adressen.stadt_id = personen.stadt_id
   WHERE nachname='King';
```

Ergibt als Output:

```
 vorname | nachname |  stadt
---------+----------+---------
 Mike    | King     | München
 Carol   | King     | München
 Alice   | King     |
```

Der Filter **nachname='King'** beim ON liefert ein Ergebnis mit **Anna Butterblume:**

```
SELECT vorname, nachname, stadt from personen
   LEFT JOIN adressen ON adressen.stadt_id = personen.stadt_id and nachname='King';
```

```
 vorname |  nachname   |  stadt
---------+-------------+---------
 Alice   | King        |
 Carol   | King        | München
 Anna    | Butterblume |
 Mike    | King        | München
```

Natural Join

Bei einem `NATURAL JOIN` werden nur die Spalten verglichen, die in beiden Tabellen gleichlautend sind.
Das Ergebnis sind alle Zeilen, in denen die Werte (der gleichlautenden Spalten) identisch sind.
Spalten, dessen Spaltenbezeichnung in der anderen Tabelle nicht vorkommen, werden für den Wert-Vergleich ignoriert.

Ein **Natural Join** ist immer ein **Equi-Join** (Gleichheit von Spaltennamen und Spaltenwert).

Syntax: **`NATURAL [ { LEFT | RIGHT | INNER | FULL ] JOIN`**

Default ist INNER

		postgreSQL	T-SQL	DB2	MySQL
`NATURAL CROSS JOIN`	ohne `ON`	nein	nein	ja	nein
	mit `ON`	nein	nein	nein	nein
`NATURAL INNER JOIN`	ohne `ON`	ja	nein	nein	ja
	mit `ON`	nein	nein	ja	nein
`NATURAL FULL OUTER JOIN`	ohne `ON`	ja	nein	nein	nein
	mit `ON`	nein	nein	ja	nein
`NATURAL LEFT/RIGHT OUTER JOIN`	ohne `ON`	ja	nein	nein	ja
	mit `ON`	nein	nein	ja	nein

Nur **postgreSQL** unterstützt Natural Join komplett (**MySQL** kennt `FULL OUTER` überhaup nicht).

T-SQL kennt `NATURAL JOIN` überhaupt nicht.

DB2 kennt die Natural-Syntax nur in Kombination mit `ON` und führt das Statement als normalen Join aus (nur die `ON` Bedingung wird berücksichtigt).

Natural Cross Join

Die (sinnlose) `NATURAL` Syntax-Variante von einem **Cross Join** kennt nur **DB2**:

```
select * from links natural cross join rechts ;
```

Ergibt als Output einen **Bulk Cross Join** (vollständiger **Cross Join**).

Bei einem **Natural Cross Join** erlaubt **DB2** keine ON Einschränkung (bei anderen **Natural Joins** erlaubt sie `ON`).

Natural Inner Join

Ein `INNER JOIN` liefert nur Zeilen, bei denen die gesuchten Werte in beiden Tabellen vorkommen.
Bei einem `NATURAL JOIN` müssen die Spaltenbezeichnungen identisch sein. Spalten, dessen Spaltenbezeichnung in der anderen Tabelle nicht vorkommen, werden für den Wert-Vergleich ignoriert.

`NULL` wird ignoriert und erscheint nicht im Ergebnis.
Eine Einschränkung mit `on` ist nur möglich bei **DB2** (nicht bei **postgreSQL** und **MySQL**).

Beispiel: `select * from links NATURAL INNER JOIN rechts;`

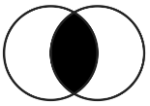


Tabelle "links"	Tabelle "rechts"

```
 wert | id
------+----
 a    |  1
      | 17
 b    |  3
 d    |  4
 d    |  5
 f    |  6
 y    | 18
 a    |  1
```

```
 wert | id
------+----
 a    |  1
 e    | 12
 a    | 13
 x    | 15
 d    |  4
 d    | 16
      | 17
 y    | 18
 d    |  4
 d    |  4
```

postgreSQL Ergebnis: zweimal a1, dreimal d4, einmal y18

```
 wert | id
------+----
 a    |  1
 a    |  1
 d    |  4
 d    |  4
 d    |  4
 y    | 18
```

Natural Left Join

Ein `NATURAL LEFT JOIN` liefert:

- einen `LEFT OUTER JOIN`
- als Einschränkung muss die Spaltenbezeichnung identisch sein. Spalten, dessen Spaltenbezeichnung in der anderen Tabelle nicht vorkommen, werden für den Wert-Vergleich ignoriert.

Beispiel: `select * from links NATURAL LEFT JOIN rechts;`

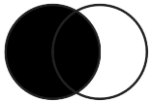


Tabelle "links"	Tabelle "rechts"

```
 wert | id
------+----
 a    |  1
      | 17
 b    |  3
 d    |  4
 d    |  5
 f    |  6
 y    | 18
 a    |  1
```

```
 wert | id
------+----
 a    |  1
 e    | 12
 a    | 13
 x    | 15
 d    |  4
 d    | 16
      | 17
 y    | 18
 d    |  4
 d    |  4
```

postgreSQL Ergebnis:

```
 wert | id
------+----
 a    |  1
 a    |  1
 b    |  3
 d    |  4
 d    |  4
 d    |  4
 d    |  5
 f    |  6
 y    | 18
      | 17
```

MySQL Ergebnis:

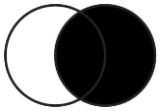
```
+------+------+
| wert | id   |
+------+------+
| a    |    1 |
| a    |    1 |
| d    |    4 |
| y    |   18 |
| d    |    4 |
| d    |    4 |
| NULL |   17 |
| b    |    3 |
| d    |    5 |
| f    |    6 |
+------+------+
```

Natural Right Join

Ein `NATURAL RIGHT JOIN` liefert:

- `RIGHT OUTER JOIN`
- als Einschränkung muss die Spaltenbezeichnung identisch sein. Spalten, dessen Spaltenbezeichnung in der anderen Tabelle nicht vorkommen, werden für den Wert-Vergleich ignoriert.

Beispiel: `select * from links NATURAL RIGHT JOIN rechts;`



wert	id
a	1
	17
b	3
d	4
d	5
f	6
y	18
a	1

wert	id
a	1
e	12
a	13
x	15
d	4
d	16
	17
y	18
d	4
d	4

postgreSQL Ergebnis:

wert	id
a	1
a	1
a	13
d	4
d	4
d	4
d	16
e	12
x	15
y	18
	17

MySQL Ergebnis:

wert	id
a	1
d	4
d	4
d	4
y	18
a	1
e	12
a	13
x	15
d	16
NULL	17

Natural Full Join

Ein `NATURAL FULL JOIN` ist ein `NATURAL INNER JOIN`, bei dem zusätzlich einmal die restlichen Zeilen der linken Tabelle und einmal die restlichen Zeilen der rechten Tabelle hinzugefügt werden.

MySQL kennt `NATURAL FULL JOIN` nicht ().

Beispiel:

```
select * from links NATURAL FULL JOIN rechts;
```

Tabelle "links"	Tabelle "rechts"

```
 wert | id
------+----
 a    |  1
      | 17
 b    |  3
 d    |  4
 d    |  5
 f    |  6
 y    | 18
 a    |  1
```

```
 wert | id
------+----
 a    |  1
 e    | 12
 a    | 13
 x    | 15
 d    |  4
 d    | 16
      | 17
 y    | 18
 d    |  4
 d    |  4
```

Ergibt als Output:

```
 wert | id
------+----
 a    |  1
 a    |  1
 a    | 13
 b    |  3
 d    |  4
 d    |  4
 d    |  4
 d    |  5
 d    | 16
 e    | 12
 f    |  6
 x    | 15
 y    | 18
      | 17
      | 17
```

Aggregate Functions

Siehe postgresql.org

			Alias	Datentyp	
AVG()	581	Durchschnittswert		Zahl	
MAX()	580	Maximalwert von Text oder Zahlen			
MIN()	579	Minimalwert von Text oder Zahlen			
SUM()	578	Summiert Spaltenwerte		Zahl	
COUNT(*)	576	Zeilenanzahl aller Zeilen (inklusive NULL Zeilen)			
COUN()	576	Zeilenanzahl, in der die angegebene Spalte nicht NULL ist			
STRING_AGG()	582	konkateniert Text-Spalten		Text	nur postgreSQL
BOOL_AND		true, wenn alle Werte true sind	every()	Boolean	nur postgreSQL
BOOL_OR		true, wenn mindestens ein Wert true ist		Boolean	nur postgreSQL

where

Wird die where-Klausel verwendet, dann ist die Nennung von Spalten in der Projektion eingeschränkt.

In der Projektion dürfen dann nur stehen:

1) Spalten mit einer Aggregatfunktion
2) Spalten, die in einem group by vorkommen.

Tabelle "werte"

g1	g2	w1	w2
a	w	4	2
b	w	3	4
b	z	0	9
b	z	5	3
c	x		0
c	x	8	7
c	y	1	3
c	y		0

Nur **MySQL** kennt diese Einschränkung nicht:

```
mysql> select w2,count(*) from werte;
+------+----------+
| w2   | count(*) |
+------+----------+
|    0 |        8 |
+------+----------+
```

Für **T-SQL, DB2** und **postgreSQL** muss w2 bei einem goup by dabei sein:

```
select w2, max(w2) from werte group by w2;
```

Die where-Einschränkung wird vor der Aggregatfunktion durchgeführt.

Beispiel 1a: count() ohne where ergibt 6 Zeilen

```
select count(w1) from werte;
```

Ergibt: 6

Beispiel 1b: count() mit where ergibt 4 Zeilen

```
select count(w1) from werte where w1>2;
```

Ergibt: 4

group by

Wird `group by` verwendet, dann bezieht sich die Aggregat-Funktion auf die jeweilige Gruppe.

Beispiel 1a: ohne `group by`

```
select sum(w1) from werte;
```

Ergibt:

```
 sum
-----
  21
```

Beispiel 1b: mit `group by`

```
select sum(w1) from werte group by g1;
```

Ergibt:

```
 sum
-----
   9
   8
   4
```

count()

Anzahl der Zeilen, bei denen die Expression nicht `NULL` ergibt.

Syntax: `count( Expression )`

Rückgabewert: `bigint`

`count` lässt sich mit `ALL` und `DISTINCT` kombinieren

Wird die **where**-Klausel verwendet, dann wird die Verwendung von Spalten in der Projektion eingeschränkt:

1) nur Spalten mit einer Aggregatfunktion
2) oder Spalten, die in einem `GROUP BY` vorkommen.

Die `where`-Einschränkung wird vor der Aggregatfunktion durchgeführt.

Die `having`-Einschränkung von `group by` wird vor der Aggregatfunktion durchgeführt.

count(*)

Besteht die Expression nur aus einem Stern, dann liefert `count(*)` alle Zeilen der Tabelle (inklusive `NULL`).

* lässt sich nicht mit `DISTINCT` kombinieren

Beispiel 1: ohne Table-Referenz

```
select count(3*2);
```

Ergibt: `1`

Möglich in **postgreSQL** und **MySQL** (Error bei **DB2**).

Beispiel 2: `count(*)` ohne Table-Referenz

```
select count(*);
```

Ergibt: `1`

Möglich in **postgreSQL** und **MySQL** (Error bei **DB2**).

Beispiel 3: alle Zeilen (inklusive `NULL`)

```
select count(*) from werte;
```

Ergibt: `8`

Tabelle "werte"

g1	g2	w1	w2
a	w	4	2
b	w	3	4
b	z	0	9
b	z	5	3
c	x		0
c	x	8	7
c	y	1	3
c	y		0

Tabelle "werte"

g1	g2	w1	w2
a	w	4	2
b	w	3	4
b	z	0	9
b	z	5	3
c	x		0
c	x	8	7
c	y	1	3
c	y		0

Beispiel 4: alle Zeilen, in denen w1 **nicht NULL**

```
select count(w1) from werte;
```

Ergibt: 6

Beispiel 5: alle Zeilen, in denen w2 **nicht NULL**

```
select count(w2) from werte;
```

Ergibt: 8

Beispiel 6: jeden Zahlenwert in der Spalte w2 nur einmal berücksichtigen (Zahl 2 und 3 sind doppel).

```
select count(DISTINCT w2) from werte;
```

Ergibt: 6

Beispiel 7: alle Zeilen, in denen w1 nicht **NULL**

```
select count(w1) from werte;
```

Ergibt: 6

Beispiel 8: w2 gruppieren, anzeigen und zählen

```
select w2, count(w2) from werte group by w2;
```

Ergibt:

```
3 |     2
4 |     1
7 |     1
9 |     1
2 |     1
0 |     2
```

Beispiel 9: w2 gruppieren, w1 anzeigen

```
select w1, w2, count(w2) from werte group by w2;
```

Nicht erlaubt in **postgreSQL** und **DB2**, da w1 nicht im group by vorkommt

Ergibt in MySQL:

```
| w1   | count(w2) |
+------+-----------+
| NULL |         2 |
|    4 |         1 |
|    1 |         2 |
|    3 |         1 |
|    8 |         1 |
|    0 |         1 |
```

Beispiel 8: w2 Spaltenwerte, die öfters als einmal vorkommen, zählen und anzeigen: `count()` in `having`-Bedingung

```
select w2, count(w2) from werte group by w2 having count(w2)>1;
```

Ergibt:

```
 w2 | count
----+-------
  3 |     2
  0 |     2
```

sum()

Summiert die Werte einer nummerischen Spalte.

Syntax: `sum( Expression )`

Rückgabewert: `bigint` bei `smallint` oder `int` Argument
`numeric` bei `bigint` Argument
ansonsten ist Übergabetyp gleich Rückgabetyp

Wird die `where`-Klausel verwendet, dann wird die Verwendung von Spalten in der Projektion eingeschränkt:

1) nur Spalten mit einer Aggregatfunktion
2) oder Spalten, die in einem `GROUP BY` vorkommen.

Die `where`-Einschränkung und die `having`-Einschränkung von `group by` werden vor der Aggregatfunktion durchgeführt.

Tabelle "werte"

g1	g2	w1	w2
a	w	4	2
b	w	3	4
b	z	0	9
b	z	5	3
c	x	7	0
c	x	8	7
c	y	1	3
c	y	7	0

Beispiel 1: ohne Table-Referenz

```
select sum(8+2);
```

Ergibt:

```
10
```

Möglich in **postgreSQL, T-SQL** und **MySQL** (Error bei **DB2**).

Beispiel 2: Summe aller Werte der Spalte `w2`

```
select sum(w2) from werte;
```

Ergibt:

```
28
```

Beispiel 3: mit `DISTINCT` jeden Wert nur einmal

```
select sum(DISTINCT w2) from werte;
```

Ergibt:

```
25
```

Beispiel 4: Summe der Spalte `w1` durch 10

```
select sum(w2/10) from werte;
```

Ergibt:

```
2.800000000000000000
```

Beispiel 5: Summe aller Werte der Spalte `w1`

```
select sum(w1) from werte;
```

Ergibt:

```
35
```

Beispiel 6: mit `where` auf `g1='b'` einschränken

```
select sum(w1) from werte  where g1='b';
```

Ergibt:

```
8
```

Beispiel 7: Summe jeder g1 Gruppe

```
select sum(w1) from werte  group by w1;
```

Ergibt:

```
 sum
-----
  23
   8
   4
```

min()

Der kleineste Wert einer Spalte.

Syntax: `min( Expression )`

Rückgabetyp ist gleich Übergabetyp (Numeric, String, Array, Date)

Wird die **where**-Klausel verwendet, dann wird die Verwendung von Spalten in der Projektion eingeschränkt:

1) nur Spalten mit einer Aggregatfunktion
2) oder Spalten, die in einem `GROUP BY` vorkommen.

Die `where`-Einschränkung und die `having`-Einschränkung von `group by` werden vor der Aggregatfunktion durchgeführt.

Tabelle "werte"

g1	g2	w1	w2
a	w	4	2
b	**w**	**3**	**4**
b	**z**	**0**	**9**
b	**z**	**5**	**3**
c	x	7	0
c	x	8	7
c	y	1	3
c	y	7	0

Beispiel 1: kleinster Wert der Spalte w1

```
select min(w1) from werte;
```

Ergibt:

```
 min
-----
   0
```

Beispiel 2: der kleinste w1-Wert, der über 5 liegt

```
select min(w1) from werte where w1>5;
```

Ergibt:

```
 min
-----
   7
```

Beispiel 3: der kleinste w1-Wert jeder Gruppe

```
select g1, min(w1) from werte group by g1;
```

Ergibt:

```
 g1 | min
----+-----
 c  |   1
 b  |   0
 a  |   4
```

Beispiel 4a: den kleinsten Wert von w2 mit `min(w2)` ermitteln und mit einem subquery die Anzahl ermitteln

```
select w2,count(w2) from werte where w2 = (select min(w2) from werte);
```

Nicht erlaubt in **postgreSQL, T-SQL** und **DB2** (w2 wird nicht gruppiert).

Beispiel 4b: den kleinsten Wert von w2 mit `min(w2)` ermitteln und mit einem subquery die Anzahl ermitteln

```
select w2,count(w2) from werte  where w2 = (select min(w2) from werte) group by w2;
```

Ergibt:

```
 w2 | count
----+-------
  0 |     2
```

max()

Der größte Wert einer Spalte.

Syntax: `max( Expression )`

Rückgabetyp ist gleich Übergabetyp (Numeric, String, Array, Date)

Wird die **where**-Klausel verwendet, dann wird die Verwendung von Spalten in der Projektion eingeschränkt:

1) nur Spalten mit einer Aggregatfunktion
2) oder Spalten, die in einem `GROUP BY` vorkommen.

Die `where`-Einschränkung und die `having`-Einschränkung von `group by` werden vor der Aggregatfunktion durchgeführt.

Tabelle "werte"

g1	g2	w1	w2
a	w	4	2
b	**w**	**3**	**4**
b	**z**	**0**	**9**
b	**z**	**5**	**3**
c	x	7	0
c	x	8	7
c	y	1	3
c	y	7	0

Beispiel 1: größter Wert der Spalte w1

```
select max(w1) from werte;
```

Ergibt:

```
 min
-----
   8
```

Beispiel 2: der größte w1-Wert, der unter 7 liegt

```
select max(w1) from werte where w1<7;
```

Ergibt:

```
 min
-----
   5
```

Beispiel 3: der größte w1-Wert jeder Gruppe

```
select g1, max(w1) from werte group by g1;
```

Ergibt:

```
 g1 | min
----+-----
 c  |   8
 b  |   4
 a  |   4
```

Beispiel 4a: den größten Wert von w2 mit `max(w2)` ermitteln und mit einem subquery die Anzahl ermitteln

```
select w2,count(w2) from werte where w2 = (select max(w2) from werte);
```

Nicht erlaubt in **postgreSQL, T-SQL** und **DB2** (w2 wird nicht gruppiert).

Beispiel 4b: den kleinsten Wert von w2 mit `min(w2)` ermitteln und mit einem subquery die Anzahl ermitteln

```
select w2,count(w2) from werte  where w2 = (select max(w2) from werte) group by w2;
```

Ergibt:

```
 w2 | count
----+-------
  9 |     1
```

avg()

Der Mittelwert einer nummerischen Spalte.

Syntax: `max( Expression )`

Rückgabetyp ist gleich Übergabetyp (Numeric)

Wird die **where**-Klausel verwendet, dann wird die Verwendung von Spalten in der Projektion eingeschränkt:

1) nur Spalten mit einer Aggregatfunktion
2) oder Spalten, die in einem `GROUP BY` vorkommen.

Die `where`-Einschränkung und die `having`-Einschränkung von `group by` werden vor der Aggregatfunktion durchgeführt.

Tabelle "werte"

g1	g2	w1	w2
a	w	4	2
b	**w**	**3**	**4**
b	**z**	**0**	**9**
b	**z**	**5**	**3**
c	x	7	0
c	x	8	7
c	y	1	3
c	y	7	0

Beispiel 1: Mittelwert der Spalte w1

```
select avg(w1) from werte;
```

Ergibt: `4.3750000000000000`

Beispiel 2: w1-Mittelwert, bei allen Werten unter 7

```
select avg(w1) from werte where w1<7;
```

Ergibt: `2.6000000000000000`

Beispiel 3: der w1-Mittelwert jeder Gruppe

```
select g1, avg(w1) from werte group by g1;
```

Ergibt:

```
 g1 |        avg
----+-------------------
 c  | 5.7500000000000000
 b  | 2.6666666666666667
 a  | 4.0000000000000000
```

Beispiel 4: den Mittelwert von w1 mit `avg(w1)` ermitteln und alle w1 anzeigen, die darüber lieben

```
select w1, (select avg(w1) from werte)
   from werte where w1 > (select avg(w1) from werte) ;
```

Ergibt:

```
 w1 |        avg
----+-------------------
  7 | 4.3750000000000000
  8 | 4.3750000000000000
  7 | 4.3750000000000000
  5 | 4.3750000000000000
```

string_agg()

Konkateniert die Zeichenketten einer Textspalte.

Syntax: `max( Expression , Separator)`

Rückgabetyp ist gleich Übergabetyp (Text).
Der Separator ist ein String.

Wird die **where**-Klausel verwendet, dann wird die Verwendung von Spalten in der Projektion eingeschränkt:

1) nur Spalten mit einer Aggregatfunktion
2) oder Spalten, die in einem `GROUP BY` vorkommen.

Die `where`-Einschränkung und die `having`-Einschränkung von `group by` werden vor der Aggregatfunktion durchgeführt.

Tabelle "werte"

g1	g2	w1	w2
a	w	4	2
b	**w**	**3**	**4**
b	**z**	**0**	**9**
b	**z**	**5**	**3**
c	x	7	0
c	x	8	7
c	y	1	3
c	y	7	0

Beispiel 1: jeden Feldwert von w1 konkatenieren

```
select string_agg(g1,'*') from werte;
```

Ergibt: `c*b*a*b*c*c*c*b`

Beispiel 2: jeden Feldwert vom gruppierten w1 konkatenieren

```
select string_agg(g1,'*') from werte group by g1;
```

Ergibt:

```
string_agg
------------
 c*c*c*c
 b*b*b
 a
```

Subquery

Ein Subyquery (Unterabfrage) ist ein Query (SQL-Statement) innerhalb eines anderen Statements.

Diese Unterabfrage kann ein `SELECT`, `DELETE`, `UPDATE` oder ein `INSERT` sein.

Positionen

Subqueries können an 3 Stellen stehen:

- als **Column** in der **Spaltenauswahlliste**
- als **Relation (Derived Table)** in der **Tabellenliste**
- als Bestandteil einer **Condition** (**Bedingung**)

Als Beispiel der SQL-Kernblock (Seite 539):

```
SELECT    [ ALL | DISTINCT ]     Spaltenauswahlliste          ❶
   FROM Tabellenliste                                        ❷

      [ WHERE Bedingung ]                                    ❸

      [ GROUP BY Spaltenliste

           [ HAVING Bedingung ]                              ❸

      ]
```

Spaltenauswahlliste

Beispiel: Alle Zeilen, in denen w1 größer als der Durchschnittswert ist.

```
select w1, (select avg(w1) from werte) from werte
 where  w1 > (select avg(w1)  from werte);
```

Ergibt als Output:

```
 7 | 4.3750000000000000
 8 | 4.3750000000000000
 7 | 4.3750000000000000
 5 | 4.3750000000000000
```

Tabelle "werte"

g1	g2	w1	w2
a	w	4	2
b	w	3	4
b	z	0	9
b	z	5	3
c	x	7	0
c	x	8	7
c	y	1	3
c	y	7	0

Tabellenliste (Derived Table)

Erzeugt der Subquery eine **Relation**, wird diese **Derived Table** genannt.

Für die erzeugte Relation muss immer ein *Alias* zugewiesen werden. Ausnahme ist **DB2**.

```
select w1 from (select * from werte where w1> 5 ) AS werte2;
```

Ergibt als Output:

```
 7
 8
 7
```

Bedingung (Condition)

Beispiel: Alle Zeilen, in denen w1 größer als der Durchschnittswert ist.

```
select w1 from werte    where w1 > (select avg(w1) as durch from werte);
```

Ergibt als Output:

```
 w1
----
  7
  8
  7
  5
```

Nested Subquery

Ein Subquery kann sich auch in einem anderen Subquery befinden.

Beispiel: nested Subquery ermittelt als Condition den w1-Durchschnittswert
der äußere Query erzeugt eine **Derived Table** (Seite 583)

```
select w1
 from (select * from werte
           where w1> ( select avg(w1) from werte) ) as werte2;
```

Ergibt als Output:

```
 w1
----
  7
  8
  7
  5
```

Tabelle "werte"

g1	g2	w1	w2
a	w	4	2
b	w	3	4
b	z	0	9
b	z	5	3
c	x	7	0
c	x	8	7
c	y	1	3
c	y	7	0

Korrelation

Noncorrelated Subquery

Eine nicht-korrelierte Unterabfrage ist in sich abgeschlossen (gekapselt).

Correlated Subquery

Eine korrelierte Unterabfrage beinhaltet Informationen vom äußeren Query (*outer references*.

Transaktion

Eine **Transaktion** besteht aus mehreren SQL-Statements, die eine logische Einheit bilden. Die Transaktion gilt erst dann als durchgeführt, wenn jedes einzelne Statement erfolgreich durchgeführt werden konnte.
Wenn ein Statement fehlschlägt, werden die bisher erfolgten Statements rückgängig gemacht (und der Zustand vor der Transaktion erreicht ist).
Ein Transaktion beginnt mit dem Keyword `BEGIN`, wird mit `ROLLBACK` abgebrochen oder mit `COMMIT` abgeschlossen.
Für das Setzen des Isolation-Levels siehe Seite 587.

ACID

Jede Transaktion lässt sich mit den **ACID**-Eigenschaften beschreiben (siehe wikipedia.org):

Atomicity	**Atomarität**. Die Statements einer Transaktion sind atomar und bilden eine logische Einheit. Durch ihre Unteilbarkeit werden entweder alle oder keine Statements durchgeführt.
Consistency	**Konsistenz**. Die Daten sind während und nach der Transaktion konsistent.
Isolation	**Isolation**. Die Transaktion wird isoliert (unabhängig) von anderen Transaktionen durchgeführt. Der Grad der Isolation wird **Isolation Level** (Seite 586) genannt.
Durability	**Dauerhaftigkeit**. Die Datenänderung einer erfolgreichen Transaktion ist von Dauer.

Isolationsphänomene

SQL-92 erlaubt die Aufweichung der ACID-Eigenschaften und beschreibt 3 Phänomene (Konfliktsituationen), die bei gleichzeitig laufenden Transaktionen entstehen:

1. **Dirty Reads**
2. **Nonrepeatable Reads**
3. **Phantom Reads**

1) Dirty Reads

Ein Query außerhalb einer laufenden Transaktion liest Daten, die noch nicht erfolgreich geschrieben und beendet worden sind (noch kein `commit`). Der Lesezugriff ist schmutzig, da die Daten inkonsistent sind und von einem Rollback zurückgenommen werden können (reading uncommitted modifications).

Beispiel:

Transaktion A		Transaktion B
`begin`		
		`begin`
	Für Transaktion-B ist x 10:	`select x`
`update x=100`		
	Dirty Read, da Transaktion-B den Wert 10 liest:	`select x`
`rollback`		

Dirty Read ist bei postgreSQL nicht möglich. Auch nicht im `read uncommitted` Level.

2) Nonrepeatable Reads

Ein Query innerhalb einer Transaktion wird wiederholt und erhält andere Daten als beim ersten Mal (da inzwischen die Änderungen einer anderen Transaktion erfolgreich beendet worden ist). Beispiel:

Transaktion A		Transaktion B
		`begin`
	Für Transaktion-B ist x 100:	`select x`
`begin`		
`update x=200`		
`commit`		
	Für Transaktion-B ist x 200:	`select x`

3) Phantom Reads

Ein Query innerhalb einer Transaktion wird wiederholt und erhält (oder betrifft) mehr Datenzeilen als beim ersten Mal (da inzwischen die Änderungen einer Transaktion erfolgreich beendet worden sind). Die neuen Datenzeilen werden Phantome genannt. Beispiel:

Transaktion A		Transaktion B
		`begin`
	Transaktion-B sieht 100 Zeilen	`count(*)`
`begin`		
`insert`		
`commit`		
	Transaktion-B sieht 101 Zeilen	`count(*)`

Lost Update

Das Update einer Transaktion wird vom Update einer anderen Transaktion aufgehoben. Beispiel:

Transaktion A		Transaktion B
		`begin`
`begin`		
`update x=200`		
	Transaktion-B hebt das Update von Transaktion-A auf:	`update x=100`
`commit`		
		`commit`

Isolation Level

SQL-92 beschreibt 4 Isolationsstufen:

	Dirty Read	Non-Repeatable Read	Phantom Read
READ UNCOMMITTED	erlaubt*	erlaubt	erlaubt
READ COMMITTED		erlaubt	erlaubt
REPEATABLE READ			erlaubt
SERIALIZABLE			

* **Dirty Read** ist bei postgreSQL nicht möglich. Auch nicht im `read uncommitted` Level.

Default Isolation Level:

	postgreSQL	DB2	MySQL	T-SQL	Oracle
Default:	READ COMMITTED	READ COMMITTED*	REPEATABLE READ	READ COMMITTED	READ COMMITTED

* CURSOR STABILITY genannt

Transaktionsverwaltung

Eine Transaktion beginnt mit einer **BOT**-Anweisung (**Begin of Transaction**):

```
BEGIN
```

Die Anweisung ROLLBACK bricht die Transaktion ab und die getätigten Statements werden rückgängig gemacht:

```
ROLLBACK
```

Die Anweisung COMMIT schließt die Transaktion ab.

```
COMMIT
```

Set Transaction Isolation Level

Der **Isolation Level** kann nur innerhalb der laufenden Transaktion (und vor dem ersten `select, insert, delete, update, fetch, copy`) gesetzt werden:

```
postgres=# begin;
BEGIN
postgres=# set transaction isolation level read uncommitted;
SET
```

Beim Versuch den **Isolation Level** außerhalb einer Transaktion zu setzen (postgreSQL):

```
WARNUNG:  SET TRANSACTION kann nur in Transaktionsblöcken verwendet werden
```

Beim Versuch den **Isolation Level** nach dem ersten Query zu setzen (postgreSQL):

```
FEHLER:  SET TRANSACTION ISOLATION LEVEL muss vor allen Anfragen aufgerufen werden
```

Locks

In der Tabelle `pg_locks` sind alle gegenwärtig gesperrten Tabellen ersichtlich.

```
select t.relname,l.locktype,page,virtualtransaction,pid,mode,granted
from pg_locks l, pg_stat_all_tables t
where l.relation=t.relid
order by relation asc;
```

Stored Procedures (PL/pgSQL)

DO

Mit dem Keyword **DO** kann ein **Code Block** (Seite 588) ausgeführt werden.

Syntax mit einfachen Anführungszeichen:

```
DO [ LANGUAGE Sprache ]  ' Codeblock ' ;
```

Syntax mit dem $$ Literal:

```
DO [ LANGUAGE Sprache ]  $$ Codeblock $$ ;
```

Default für den optionalen Parameter **LANGUAGE** ist `plpgsql` .

Beispiel1: Blockcode in einfachen Anführungszeichen

```
DO 'BEGIN update zahlen set zahl=999; END ';
```

Beispiel2: Blockcode im $$ Literal

```
DO 'BEGIN update zahlen set zahl=999; END ';
```

Beispiel3: Blockcode mit Whitespace

```
DO
  '
    BEGIN
      update zahlen set zahl=333;
    END;
  ';
```

Code Block

Ein Code-Block besteht aus 1 bis 3 Teilen:

1)	Label	definiert die Bezeichnung (optional)
2)	Variable-Deklaration	beginnt mit dem Keyword **DECLARE** (optional)
3)	Body	beginnt mit dem Keyword **BEGIN** und endet mit **END**

Die Syntax von einem Code Block:

```
[ <<Label>> ]
    [ DECLARE   Deklarationen ]
    BEGIN
       Anweisungen
    END
[ Label ] ;
```

Ein **Code Block** kann Bestandteil von `DO` (Seite 588) oder einer Funktion sein.

Beispiel: *Hello, World* mit dem Keyword RAISE

```
DO
    $$
        BEGIN
            raise notice 'ade, schnöde welt';
        END;
    $$;
```

Ergibt als Output:

```
HINWEIS:  ade, schnöde welt
```

Variable-Deklaration

Mit dem Keyword DECLARE wird (einmalig) der Anfang der Variablendeklarationen gekennzeichnet.

Jede verwendete Variable im Body muss zwingend vorher deklariert werden.
Ohne Initialisierung (Wertzuweisung) hat die Variable den Wert NULL.

Syntax:

```
DECLARE
   Variablename
      [CONSTANT]
      Datentyp
      [NOT NULL]
      [
         { DEFAULT | := }   Expression
      ]
   ;
```

Beispiel: Variable foo1 als Integer mit dem Wert 100, Variable foo2 als String mit dem Wert blub

```
DO
 $$
 DECLARE
    foo1 decimal := 100;
    foo2 varchar(20) := 'blub';

  BEGIN
              raise notice 'foo1 ist %, und foo2 ist %', foo1, foo2;
  END;
$$;
```

Ergibt als Output:

```
HINWEIS:  foo1 ist 100, und foo2 ist blub
```

Assignment Operator

Mit dem Zuweisungsoperator **:=** wird einer Variable eine Expression zugewiesen.

Syntax: *Operand1* **:=** *Operand2* ;

Beispiel:

```
foo1 decimal := 1;
```

Der Vergleichsoperator ist =

Datentyp aus Tabellen-Schema

Der Datentyp bei der Variable-Deklaration kann auch aus der Tabellen-Definition abgeleitet werden.

Syntax:

```
DECLARE Variablename meineTabelle.
```

Alias

Mit **ALIAS FOR** kann eine alternative Variablebezeichnung bestimmt werden.

Syntax: *Variablebezeichnung1* **ALIAS FOR** *Variablebezeichnung2* **;**

Beispiel: foo3 ist ein Alias von foo2

```
DO
 $$
   DECLARE
     foo2 varchar(20) := 'blub';
     foo3 ALIAS for foo2;

   BEGIN
          raise notice 'foo2 ist % und foo3 ist %',  foo2, foo3;
   END;
$$;
```

Ergibt als Output:

```
HINWEIS:  foo2 ist blub und foo3 ist blub
```

Conditional Statements (Bedingte Flusskontrolle)

if then

Bei der IF-Flusskontrolle wird eine Expression ausgewertet. Ist das Ergebnis der Expression `true`, wird der Programmblock (zwischen `THEN` und `END IF`) der `IF`-Bedingung ausgeführt.
Ist das Ergebnis `false`, wird dieser `THEN`-Programmblock übersprungen.
Syntax:

```
IF Expression
    THEN
       Anweisungen
END IF;
```

Beispiel:

```
DO
$$
 DECLARE
   x int := 10;
  BEGIN
       IF x>5
          THEN
             raise notice 'x ist % und damit groesser als 5.',  x;
       END IF;
  END;
$$;
```

Ergibt als Output:

```
HINWEIS:  foo2 ist blub und foo3 ist blub
```

if then else

Bei der IF-ELSE-Flusskontrolle wird eine Expression ausgewertet. Ist das Ergebnis `true`, wird der `THEN`-Programmblock ausgeführt, ansonsten der `ELSE`-Programmblock
Syntax:

```
IF Expression
    THEN
       Anweisungen
    ELSE
       Anweisungen

 END IF;
```

Beispiel:

```
DO $$
 DECLARE
   x int := 10;
  BEGIN

    IF x=3
       THEN
          raise notice 'x ist 3';
       ELSE
          raise notice 'x ist nicht 3';
    END IF;

 END;
$$;
```

Ergibt als Output:

```
HINWEIS:  x ist nicht 3
```

CASE WHEN

Bei der CASE WHEN Flusskontrolle können beliebig viele Expressions kaskadiert überprüft werden.
CASE WHEN als Logic Expression siehe Seite .

Syntax-Variante 1

Die jeweilige **WHEN**-Expression wird auf **true** überprüft. Nach dem ersten Treffer werden die folgenden (restlichen) WHEN-Überprüfungen übersprungen.

Die ELSE Angabe ist bei der Syntaxüberprüfung optional. Tritt zur Laufzeit eine Situation ein, in der keine WHEN-Expression übereinstimmt, wird das fehlende ELSE als kritischer Fehler gewertet.

Syntax:

```
CASE
   WHEN Expression
      THEN
         Anweisungen
    [...]

   ELSE
      Anweisungen
END CASE;
```

Beispiel 1a:

```
DO´$$
 DECLARE   x int := -2;
  BEGIN
       CASE
          WHEN x=0
             THEN
                raise notice 'x ist 0';
          WHEN x<0
            THEN
                raise notice 'x ist kleiner 0';
           ELSE
              raise notice 'x ist groesser 0';
       END CASE;
  END;
 $$
```

Ergibt als Output:

```
x ist kleiner 0
```

Beispiel 1b: ELSE fehlt

```
DO
$$
 DECLARE  x int := 0;
  BEGIN
       CASE
          WHEN x>0
             THEN
                raise notice 'x ist groesser 0';

          WHEN x<0
            THEN
                raise notice 'x ist kleiner 0';
       END CASE;
  END;
$$;
```

Ergibt als Output:

```
FEHLER:  Fall nicht gefunden
TIP:  Der CASE-Anweisung fehlt ein ELSE-Teil.
```

Syntax-Variante 2

Bei der switch Variante wird eine CASE-Expression mit (fast) beliebig vielen WHEN-Expressions **auf Gleichheit überprüft.**

Syntax:

```
CASE Expression

   WHEN Expression
      THEN
         Anweisungen
    [...]

   [ ELSE
      Anweisungen ]

END CASE;
```

Beispiel 1:

```
DO
$$
 DECLARE    x int := 0;
  BEGIN
       CASE x
          WHEN 2-2
             THEN
                raise notice 'x ist 0';
          WHEN 1
              THEN
                  raise notice 'x ist -1';
            ELSE
               raise notice 'x ist weder 0, noch -1';
       END CASE;
 END;
$$;
```

Ergibt als Output:

```
HINWEIS:  x ist 0
```

Beispiel 2: ELSE fehlt

```
DO
$$
  BEGIN
      CASE 1-1
         WHEN -1
            THEN
               raise notice 'x ist -1';

         WHEN 1
            THEN
               raise notice 'x ist 1';
       END CASE;
  END;
$$;
```

Ergibt als Output:

```
FEHLER:  Fall nicht gefunden
TIP:  Der CASE-Anweisung fehlt ein ELSE-Teil.
```

Beispiel:

```
DO
$$
 DECLARE
   x int := 2;
   y int := 3;

  BEGIN

    CASE x

       WHEN 1                                                    /* Literal */
          THEN
          raise notice 'x ist 1';

       WHEN 1+1                                                  /* Literale mit Operator */
          THEN
          raise notice 'x ist 2';

       WHEN y                                                    /* Skalar-Variable */
          THEN
          raise notice 'x ist 3';

       ELSE
          raise notice 'x > 3';

    END CASE;

  END;
 $$;
```

Ergibt als Output:

```
HINWEIS:  x ist 2
```

Remarks

postgreSQL kennt einen **Single Line Comment** und einen **Block Comment** (über mehrere Zeilen).

Single Line Comment

Der **Single Line Comment** beginnt mit 2 Minuszeichen. Alles danach (bis zum Zeilenende) wird vom Interpreter ignoriert.

Beispiel:

```
DECLARE    foo2 varchar(20) := 'blub'; -- Variable foo2 deklarieren und initialisieren
```

Block Comment

Der **Block Comment** beginnt mit **/*** und endet mit ***/**. Alles dazwischen wird vom Interpreter ignoriert.

Beispiel:

```
DECLARE

 /*
   Variable foo2
   deklarieren und initialisieren
 */

foo2 varchar(20) := 'blub';
```

Beispieltabellen

Beispieltabellen links, mitte, rechts

```
DROP TABLE IF EXISTS links;
CREATE TABLE links (wert1 char(1),   id1 int   );

DROP TABLE IF EXISTS mitte;
CREATE TABLE mitte (wert2 char(1),   id2 int    );


DROP TABLE IF EXISTS rechts;
CREATE TABLE rechts ( wert3 char(1),    id3 int  );


INSERT INTO  links (wert1,id1) VALUES ('a',1);
INSERT INTO  links (wert1,id1) VALUES ('f',6);
INSERT INTO  links (wert1,id1) VALUES ('a',2);

INSERT INTO  mitte (wert2,id2) VALUES ('a',1);
INSERT INTO  mitte (wert2,id2) VALUES ('a',13);
INSERT INTO  mitte (wert2,id2) VALUES ('x',15);

INSERT INTO rechts (wert3,id3)  VALUES ('a',1);
INSERT INTO rechts (wert3,id3)  VALUES ('a',26);
INSERT INTO rechts (wert3,id3)  VALUES ('y',28);
```

Beispieltabellen links, rechts

```
DROP TABLE links;
CREATE TABLE links   (
    wert char(1),
        id int
    );


DROP TABLE rechts;
CREATE TABLE rechts  (
    wert char(1),
        id int
    );

INSERT INTO  links (wert,id)
VALUES ('a',1),('a',17),('b',3),('d',4),('d',5),('f',6),('y',18),('a',1);


INSERT INTO rechts (wert,id)
VALUES ('a',1),('e',12),('a',13),('x',15),('d',4),('d',16),(null,17),('y',18),('d',4),('d',4) ;
```

Beispieltabelle werte

```
drop table if exists werte;
create table werte
 (
    g1 char(1),
    g2 char(1),
    w1 decimal,
    w2 decimal
 );

insert into werte values ('c','x',7,0), ('b','w',3,4),
('a','w',4,2),('b','z',0,9),('c','x',8,7);

insert into werte values ('c','y',7,0), ('c','y',1,3), ('b','z',5,3);
```

Beispieltabelle zahlen

```
drop table if exists zahlen;
create table zahlen
(
 zahl decimal,
 id int
);

insert into zahlen values (10,1), (20,2), (30,3);
```

Beispieltabelle comics

```
DROP TABLE IF EXISTS comics;
CREATE TABLE comics (
    heftnr int,
    comic char(20),
    jahr int,
    id int
    );


INSERT INTO  comics (heftnr,comic,jahr,id)
 VALUES
 (124,'Spiderman',1985,1),
 (3,'Batman',1993,2),
 (22,'Batman',1985,3),
 (55,'Lupo',1978,4),
 (63,'Superman',1969,5),
 (65,'Lupo',1985,6),
 (46,'Donald',1955,7),
 (32,'Donald',1969,8),
 (3,'Spiderman',1978,9),
 (3,'Garfield',1955,10);
```

Datenbank-Modelle

Kapitel 13

Datenbank

Datenbank-Modelle

Eine Datenbank (Database) ist eine strukturierte Sammlung von Daten.
Ein Datenbankmodell beschreibt die Datenstrukturen und ihrer Beziehungen untereinander

Hierarchische Datenbank-Modell

Das hierarchische Datenbankmodell (HDM) ist das älteste Modell und entstand im **I**nformation **M**anagement **S**ystem (IMS) von IBM. IMS wurd zwischen 1966 und 1968 von IBM, Rockwell und Caterpillar für das Apollo-Mondprogramm entwickelt. Seit 1969 wird es von IBM weiterentwickelt.

Die Datenstruktur ist ein hierarchisch aufgebauter Baum. Da die Anzahl der Ebenen (Baumtiefe) unterschiedlich sein kann, wird diese Baumart **non-balanced Tree** genannt.

Die Reihenfolge der Knoten bestimmt die Beziehung. Darum die Bezeichnung **hierarchisch**.

Man unterscheidet zwischen 2 Ebenen:

1) **Typebene** (Definition)
   Die Definition der **Objekte** und ihren Beziehungen untereinander.

2) **Ausprägungsebene**
   Die eigentlichen Daten. Die Objekte werden hier **Entitytypen** genannt.

Typebene (Datenbankschema)

Die oberste Ebene (Level 1) besteht nur aus einer **Wurzel** (**Root**). Ein Objekt, dass Vater (Elter) und Kinder hat, wird **Knoten** (**Node**) genannt. Ein Objekt ohne Kinder ist ein **Blatt** (und somit immer die unterste Ebene des jeweiligen Astes).

Der Baum besteht aus:

- der Wurzel (Knoten ohne Vater)
- den Knoten
- den Kanten. Sie bilden die hierarchischen Beziehungen ab (Vater-Kind)
- den Blättern (Knoten ohne Kinder)

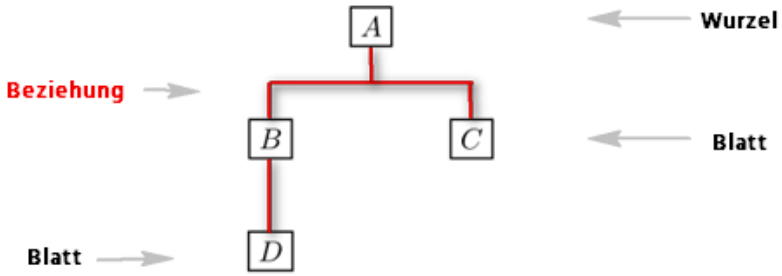
Jeder Knoten kann mehrere Kinder (**childs**) haben.
Jeder Knoten hat immer genau einen Elternknoten (**parent** node). Die jeweils untersten Ebenen werden Blätter (**Leaves**) genannt. Mit dieser Struktur sind **nur 1:n Beziehungen** möglichen.

Beispiel:

A ist die **Wurzel** (nur Kinder).

B ist ein **Knoten** (hat Elter und Kinder).

D und **C** sind **Blätter** (Endknoten).



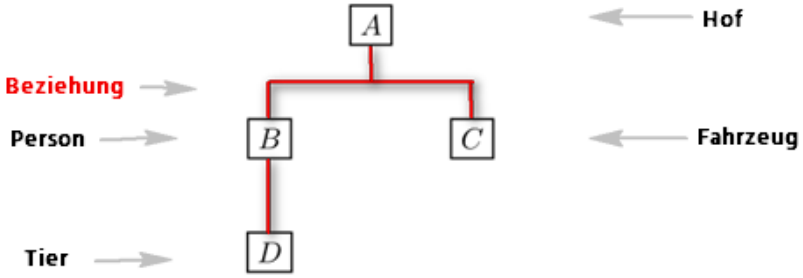
Die Objekte mit Bezeichnungen:

A ist der Bauernhof.

B ist eine Person.

C ist ein Fahrzeug.

D ist ein Tier.



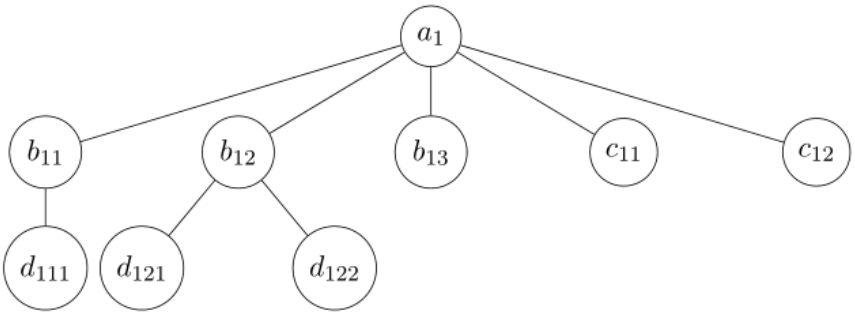
Ausprägungsebene (Daten)

Wenn die Objekte mit konkreten Werten (Daten) gefüllt werden, spricht man von der **Ausprägungsebene** (auch Instanzenebene und Satzebene) genannt. Auf dieser Ebene werden die Objekte Entities genant.

Nicht immer müssen alle Äste voll belegt sein. Im Beispiel hat b11 ein Kind, b12 hat 2 Kinder und b13 hat keine Kinder.

Die Darstellung des Bauernhofes in der Ausprägungsebene (Instanzenebene bzw Satzebene):

a1	Bauerhof Gruber
b11	Bäuering Maria
b12	Bauer Franz
b13	Tochter Alice
c11	Traktor
c12	Mähdrescher
d111	Henne Sofie
d121	Schwein Karl
d122	Kuh Angelika



Das Auslesen erfolgt sequentiell in Präorder-Reihenfolge und startet somit in der Wurzel:

a 1 → b 11 → d 111 → b 12 → d 121 → d 122 → b 13 → c 11 → c 12

Vorteil

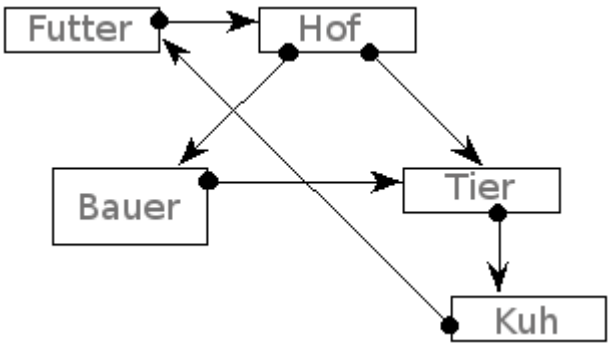
Das Modell ist leicht verständlich (übersichtlich).

Nachteile

- keine n:m Beziehungen möglich.
- reduntante Daten. Wenn Datensätze in mehreren Beziehungen stehen können, müssen sie mehrfach angelegt (und später auch mehrfach aktualisiert und gelöscht werden).
  In IMS wurde dieses Problem durch den Einsatz von virtuellen Datensätzen behoben.
- Strukturänderungen sind sehr schwierig

Vernetztes Datenbank-Modell

Das vernetzte Datenbankmodell entstand.



Vorteile

- n:m Beziehungen sind möglich (im Gegensatz zum Hierarchischen Datenbank-Modell)

Nachteile

- je komplexer die Beziehungen, umso mehr leidet die Übersichtlichkeit
- Strukturänderungen sind schwiering

Datenbank-Index

Buchindex

Damit ein Mensch den Index eines Buches optimal verwenden kann, muss dieser Index (das Stichwortverzeichnis) sortiert sein. Zum Beispiel nach alphabetisch sortierten Stichwörtern: zu jedem Stichwort gibt es die zugehörigen Seitenzahlen.

Stichwortverzeichnis

Agenda 2010 35, 61, 76
angemessenes Wirtschaftswachstum 48
Arbeitslosenquote 32, 38, 45, 49f., 84, 91
Arbeitslosigkeit 45ff., 55ff., 75f., 78f., 84ff., 91
Athens before the referendum 14
Außenhandel 20, 27, 32
Außenhandelsdaten 27, 32
außenwirtschaftliche Ungleichgewichte 55
austerity 16, 35f., 40, 67, 69ff.
Berliner Republik 58
Binnennachfrage 27, 58ff.
China 71, 74
Claus Köhler 31, 37, 45, 48, 51, 77, 86, 89
competitiveness 70
Freiheit von Forschung und Lehre 76
Geldpolitik 51f., 55ff.
German Council of Economic Experts 31, 37, 45, 82, 89
Germany 31, 35ff., 67, 70, 77, 80, 89
Gesamtnachfrage 59f.
Greece 14ff., 35f., 40, 68ff., 73f., 80
Handelsbilanzüberschuss 29
Haushaltsdefizite 45
Hörfunkrat 6ff., 10
industrial and agricultural development 74
industry 70f., 73, 81
Inflationsziel 18, 51, 56, 61, 83
Institut der Deutschen Wirtschaft (IW) 75
Irland 82ff., 88
Italien 20, 22

Stichwörter, die mit dem Buchstaben A beginnen, stehen am Verzeichnisanfang, M in der Mitte und mit Z am Verzeichnisende. Somit ist es nicht notwendig, das gesamte Stichwortverzeichnis zu lesen, um das gesuchte Wort im Index zu finden.

Table Scan

Ohne Stichwortverzeichnis muß das gesamt Buch gelesen werden, um das gewünschte Wort zu finden. Bei einer Datenbank-Tabelle nennt man diesen Suchvorgang **table scan**.

Index Seek

Das Suchen im Verzeichnis wird **index seek** genannt.

Wird bei einem Buchindex ein neues Stichwort hinzugefügt, wird das Stichwort an die richtige Stelle eingefügt und die restlichen Einträge nach hinten verschoben.

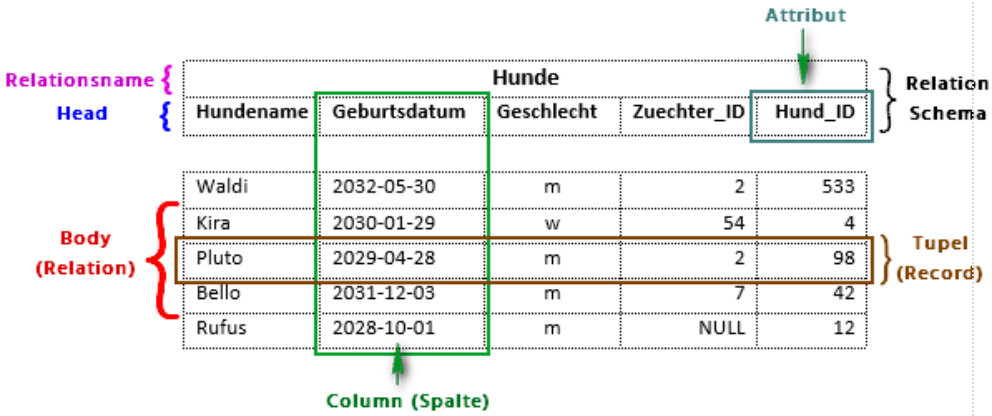
Sowohl ein Buch, als auch eine Tabelle, kann mehrere Verzeichnisse besitzen: Namen, Abbildungen, Jahreszahlen und Orte.

Datenbank-Index (Tabellenindex)

Die Daten in einer Datenbank sind als Tabellen organisiert. Diese Datenbank-Tabellen werden **Relationen** genannt.

Der **Relationsame** ist der Tabellenname.

Die Zeilen werden **Datensätze** (**Tupel** und **Records**) genannt. Mehrere Zeilen sind ein **Record Set**. Die Spaltenüberschriften in der zweiten Zeile (**Head**) der Tabelle sind **Attributbezeichnungen**. Sie sind nur Bestandteil des **Relationsschema**, und nicht in der eigentlichen Datentabelle (**Body**) enthalten. Wird der Body ohne Sortierung verwaltet, wird er **Heap** genannt. Der Inhalt der einzelnen Daten-Zellen sind die **Attributwerte**.



Um den Zugriff auf bestimmte Datensätze zu beschleunigen, ist eine zusätzliche Datenstruktur notwendig: der **Index**.

Dieser Index ist eine eigenständige Datenstruktur und benötigt seinen eigenen Speicherplatz.

Der Index enthält jene Daten, nach denen in der eigentlichen Datentabelle gesucht wird. Somit muss bei jeder Änderung der Datentabelle, auch der Index aktualisiert werden.

Wird bei einem Buchindex ein neues Stichwort hinzugefügt, wird das Stichwort an die richtige Stelle eingefügt und die restlichen Einträge nach hinten verschoben.

Bei einem Datenbank-Index wäre so eine Einfügung und Verschiebung aufwendig - und zum Glück auch nicht notwendig. Die physische Reihenfolge der Einträge (Stichwörter) spielt keine Rolle. Ein **neuer Listeneintrag** (**Knoten**) wird am Ende des Index hinzugefügt. Damit der Index trotzdem sortiert ist, werden bei jedem Listeneintrag zwei weitere Informationen hinzugefügt. Jeder Knoten kennt seine zwei Nachbarn in der gewünschten Sortierung. Diese Struktur wird **doppelt verkettete Liste** genannt.

Tabelle

Relation (Body)

Eine Tabelle, die nur aus den eigentlichen Werten (Body) besteht, wird Relation genannt.
Der Tabellenname ist der Relationsname.

Relationsschema

Ein Relationsschema beschreibt die Struktur der Tabelle. Der Tabellenname (**Relationsname**), die Attributsdefinitionen (Name, Datentyp, Größe) und die Indexdefinitionen bilden das **Relationsschema**.

Attribut (Head)

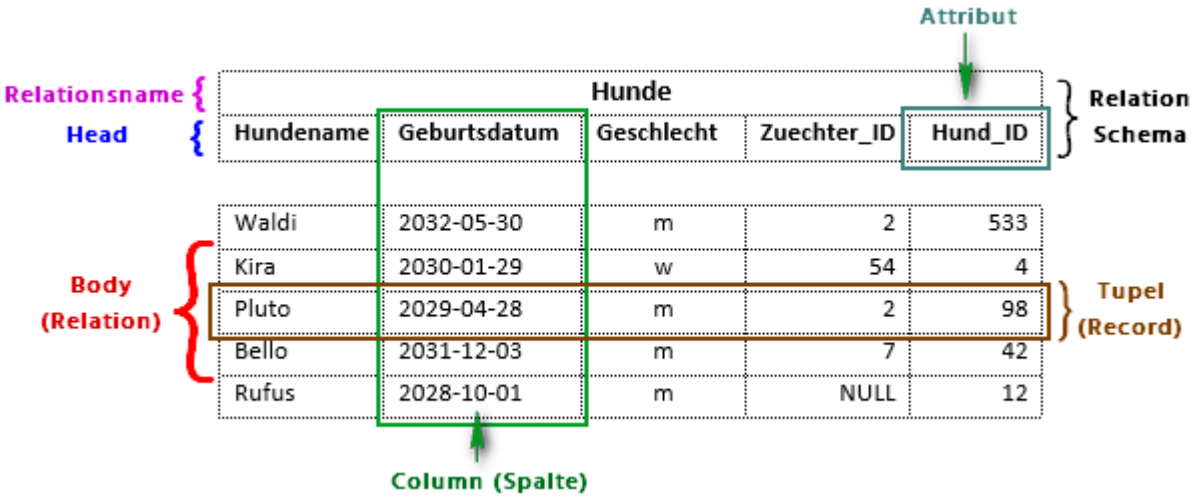
Die Attributbezeichnungen sind die Spaltennamen in der zweiten Zeile der Abbildung. Dieser Tabellenkopf (Head) dient nur zur Veranschaulichung und ist nicht Bestandteil der echten Datentabelle. Jedes Attribut besitzt seinen eigenen Datentyp.

Attributwert

Der eigentliche Dateninhalt einer Tabellenzelle wird Attributwert genannt. Alle Spalten (Zellen) einer Zeile sind ein Datensatz (Record) und werden **Tupel** genannt.

Tupel, Record Set

Eine einzelne Zeile (**Row**) in der Datentabelle wird **Tupel** und Datensatz (Record) genannt. Mehrer Zeilen sind ein **Record Set**. Alle Zeilen sind der Body (der Head ist das Relationen-Schema mit den Attributbezeichnungen)



Domain

Die Domain ist der mögliche (erlaubte) Wertebereich einer Spalte. Ein Eintragen eines nicht erlaubten Wertes wird von der Datenbank verhindert.

Heap

Der Heap ist die eigentliche, unsortierte Datentabelle. Eine sortierte Datentabelle ist ein **Clustered Index**.

Clustered Index

Bei einem Clustered Index, sind bereits die eigentlichen Daten sortiert. Für gewöhnlich ist eine Datenbank-Tabelle nicht sortiert und wird deshalb Heap (Haufen) genannt.

Die Bedeutung bei einem Buch:
In einem Lexikon sind die Lexikon-Stichwörter bereits alphabetisch sortiert (ein Verzeichnis ist somit nicht notwendig).

B+Tree

Doppelt verkettete Listen

In einer logischen Indexstruktur kennt jeder Knoten (Listeneintrag) seine zwei Nachbarn der gewünschten Sortierung. Der Index ist somit logisch und nicht physisch sortiert.

Bei einer physischen Sortierung wird ein neuer Eintrag an die jeweilige Stelle des Index geschrieben (die nachfolgenden Einträge müssen verschoben werden).

Bei einer logischen Sortierung wird der neue Eintrag am Indexende hinzugefügt. Bei den betroffenen alten Einträgen muss der Verweis auf den Nachbarn geändert werden. Ein Verschieben von großen Datenmengen ist nicht notwendig.

Blattknoten (Leaf Node) im B-Tree

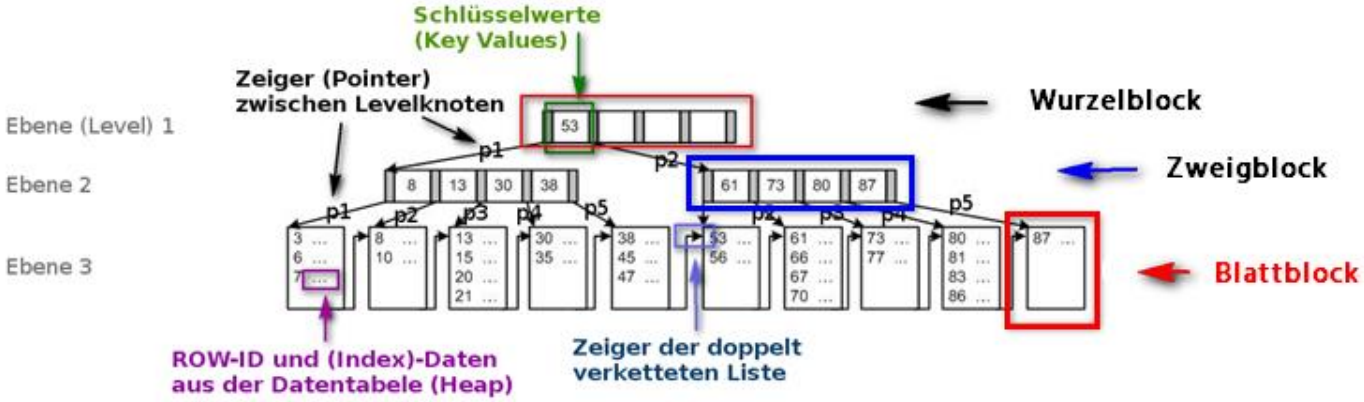
Die doppelt verketteten Listeneinträge werden Blattknoten (leaf node) genannt. Blattknoten sind die unterste Ebene eines Index. Mehrere Listeneinträge werden zu Blöcken zusammengefasst.

Die oberen Ebenen sind ein B-Tree (balanced tree). Der B-Tree ist ein vollständig balanzierter binärer Baum.

Somit besteht ein Index (B+Tree) aus zwei unterschiedlichen Strukturen:

- Die oberen Index-Einträge sind als B-Tree (**Balanced Tree**) abgelegt.
  B-Tree **steht nicht für ~~binary tree~~**. Im Beispiel ist die oberste Zeile der Wurzelknoten. Die zweite Zeile besteht aus Zweigknoten (Branch Nodes), die auch Innere Knoten genannt werden.
- Die **Blattknoten** sind imme die unterste Ebene und werden als **doppelt verkettete Liste** angelegt. Die Leaves Nodes enthalten die Zeiger auf die eigentlichen Datentabellen. Also der Verweis auf Seiten- und Zeilennummer des Heaps (mittels ROW-ID).
  Im Beispiel kann jeder Blatt-Block bis zu 4 Einträge enthalten. Die ideale Blockbelegung sind 2 Einträge (also die Hälfte der Maximalbelegung).
  Wenn ein Zweigknoten zum Beispiel bis zu 3 Einträge (p-1=4) fassen kann, dann muss dieser Knoten 5 Zeiger (p=5) auf die nächste Ebene besitzen:

  p 5 tree-pointers
  p-1 4 key-values



Ein Index verändert nie die eigentliche Datentabelle. Aber jede Änderung der Datentabelle bewirkt eine Änderung des Index (also eine Index-Aktualisierung).

B-Tree versus B+Tree

Ein B+Tree ist ein B-Tree (Balanced Tree), der um eine zweite Datenstruktur erweitert worden ist.
Die unterste Ebene (Blattknoten) wird als doppelt verkettete Liste verwaltet und enthält die Index-Tabellendaten. Die restlichen Ebenen sind der B-Tree.
Die oberen Ebenen enthalten die key values und die pointer auf den nächsten Knoten. Diese Ebenen stellen den B-Tree dar.

Baumknoten im B+Tree

Jeder Suchvorgang beginnt im Wurzelknoten, folgt den Zweigknoten und endet im Blattknoten.
Vom Blattknoten erfolgt mittels der ROW-ID der Sprung in den Heap, der eigentlichen Datentabelle.

Wurzelknoten (Root Node)

Der Wurzelknoten ist immer die oberste Ebene. Und somit die Startebene bei einem Suchvorgang.
Jeder Blattknoten hat immer die gleiche Entfernung zur Wurzel.

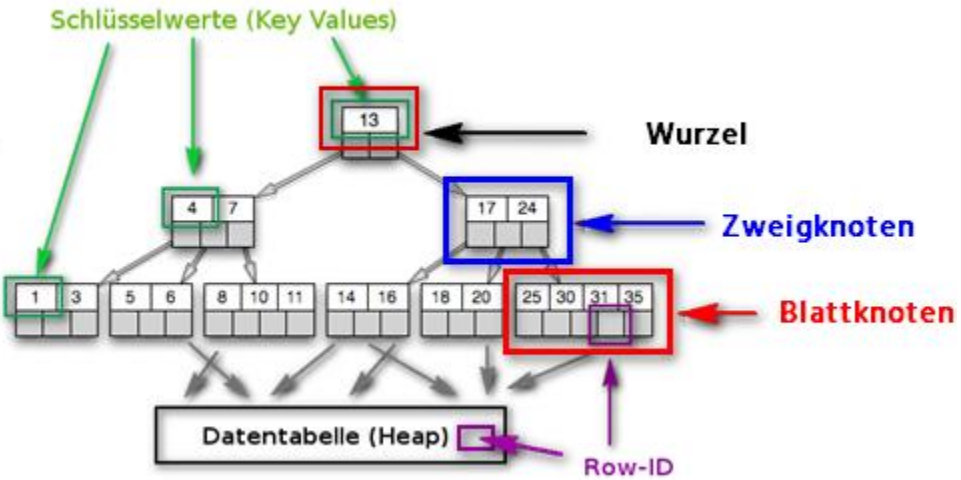
Zweigknoten (branch node)

Nach dem Wurzelknoten folgen die Zweigknoten (Branch Nodes), auch Innere Knoten genannt). Wie die Wurzel (Root), enthalten auch sie die Key-Values und Zeiger auf die nächste Ebene (Level). Die Schlüsselwerte sind die Indexdaten, nach denen gesucht wird.
Wenn ein Zweigknoten zum Beispiel bis zu 3 Einträge (p-1=4) fassen kann, dann muss dieser Knoten 5 Zeiger (p=5) auf die nächste Ebene besitzen:

p 5 tree-pointers
p-1 4 key-values

Blattknoten (Leaf Node)

Nach dem Wurzelknoten und den Zweigknoten (branch nodes) kommen die Blattknoten (Leaf Nodes). Sie enthalten die Zeiger (ROW-ID) auf die eigentliche Datentabelle (Heap).
Falls die gewünschten Daten nicht im Blattknoten sind, werden sie über die ROW-ID in der Datentabelle geholt. Dieser Vorgang wird Table Access by Index Row-ID genannt.



Blattknoten Scans

Der Blattknoten-Scan ist entweder ein schneller **Index Unique Scan** oder ein **Index Range Scan**:

Index Unique Scan

Der gesuchte Index-Wert ist maximal einmal im Blattknoten vorhanden. Wenn es ein Treffer gibt, muß im Blattknoten nicht nach weiteren Einträgen gesucht werden. Falls beim Treffer weitere gewünschten Daten nicht im Blatt vorhanden sind, müssen diese mit einem **Table Access by Index Row-ID** aus der eigentlichen Datentabelle (Heap) geholt werden.

Index Range Scan

Der gesuchte Index-Wert kann öfters im Blattknoten vorhanden sein. Der Range Scan muß die benachbarten Einträge solange verfolgen, bis alle Einträge mit diesem Wert gefunden worden sind .

Wie beim **Index Unique Scan:** Falls bei den Treffern weitere gewünschten Daten nicht im Blatt vorhanden sind, müssen diese mit einem **Table Access by Index Row-ID** aus der eigentlichen Datentabelle (Heap) geholt werden.

Datentabelle (Heap)

Der letzte Schritt im Suchvorgang ist der Sprung vom Blattknoten in den Heap. Der Zugriff erfolgt über die ROW-ID.

Die *erste Macht* des Index

Ein Balanced Tree hat immer eine einheitliche Tiefe: Der Weg zwischen dem Wurzelknoten und den Blattknoten ist immer und überall gleich lang.

Clustered-Index

MicroSoft-SQL-Server und MySQL (mit InnoDB) bezeichnen Tabellen, die nur aus einer Indexstruktur bestehen, als **Clustered Index** . Bei Oracle werden solche Tabellen **Index-Organized Table** (IOT) genannt.

Somit ist die eigentliche Datentabelle bereits sortiert und benötigt keinen Index. Bei Büchern entspricht das dem Lexikon.

Index-Organized Table (IOT)

Die eigentliche Datentabelle ist bereits sortiert und benötigt somit keinen Index. Bei Büchern entspricht das dem Lexikon. MicroSoft-SQL-Server und MySQL (mit InnoDB) bezeichnen Tabellen, die nur aus einer Indexstruktur bestehen, als **Clustered Index**.

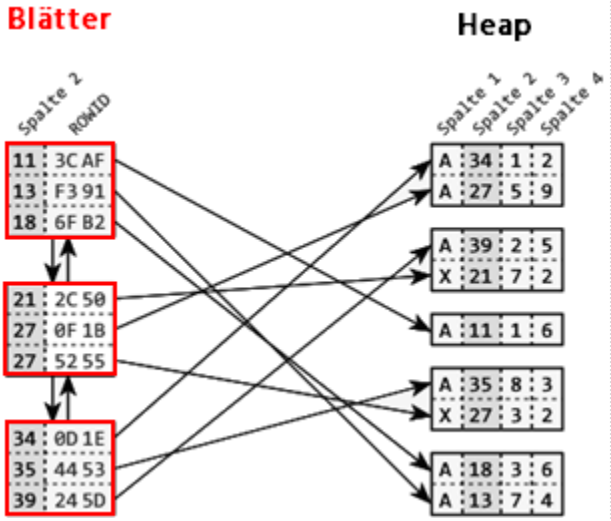
Suchschritte

Ein Indexzugriff besteht mindestens 2 Schritten. Die dritte und letzte Operation ist der Zugriff auf die eigentliche Datentabelle. Dieser Schritt ist nur notwendig, wenn die gewünschten Daten sich nicht im Index befinden.

1. Durchwandern des B-Tree (Balanced Tree)
2. Verfolgen der Blattknoten
3. Zugriff auf den Heap (optional)

Nur beim ersten Schritt gibt es durch die Baumstruktur eine Obergrenze für die Anzahl der Lesezugriffe: die Anzahl der Baumebenen

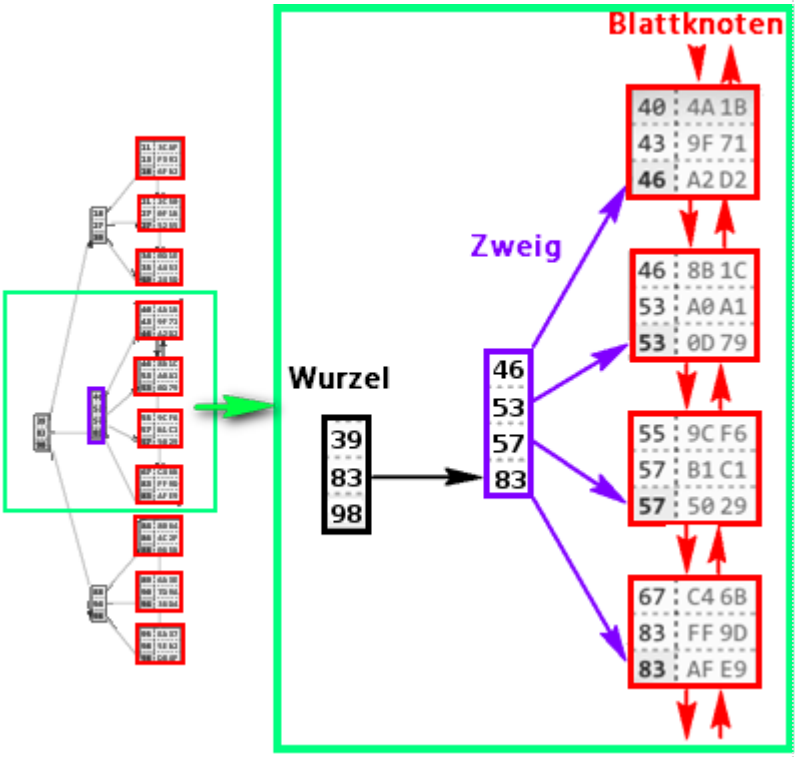
Suchbeispiel



Die linken Blöcke sind die Blattknoten. Die rechten Blöcke sind die eigentliche Datentabelle. Diese unsortierte Datentabelle wird **Heap** genannt.
Der Blattknoten enthält die indizierten Daten des Heap (Spalte 2) und die ROWID der Datentabelle.

Die **Blattknoten** (Leaf Nodes) enthalten in diesem Beispiel 30 Werte (10 pages zu je 3 Einträgen).
Diese leaf nodes sind als doppelt verkette Liste abgelegt.
Der **Wurzelknoten** (Root Node) und die **Zweigknoten** (Branch Nodes) sind der **B-Tree** (balanced tree).

Jede Indexsuche beginnt beim Wurzelknoten, folgt den weiteren Zweigknoten des Baumes und endet bei den Blattknoten.



In diesem Beispiel entspricht jeder Sucheintrag im Zweigknoten dem größten Wert eines Blattknoten-Block.

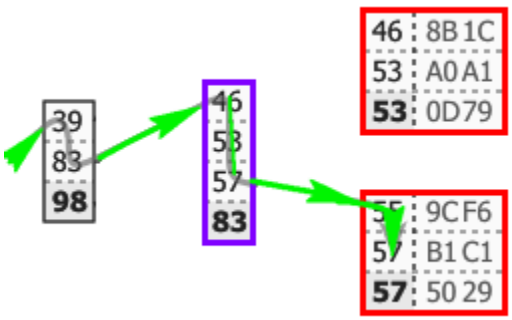
Da der größte Wert im Beispiel-Blattknoten 46 ist (siehe Abbildung rechts), ist der erste Eintrag im Zweigknoten ebenfalls 46.

Die Einträge des Wurzelknotens werden der Reihe nach mit dem Suchbegriff 57 verglichen.

Bis ein Eintrag gefunden wird, der größer-gleich dem Suchbegriff ist.

In der Abbildung trifft das auf den Eintrag 83 zu.

Von der 2. Ebene geht es weiter in den Blattknoten:
Und vom Blattknoten mit den Zeigern ROWID `B1C1` und ROWID `5020` in die Datentabelle (Heap):



Baumgröße im B+Tree

Maximale Indexeinträge

Gegeben sind Zweigknoten, die bis zu vier Einträge fassen können.
Bei einer **Baumtiefe** (tree height) von 3 ergibt das 4^3=64 maximale Einträge:

$$\text{Zweigknoteneinträge}^{\text{Baumtiefe}} = \text{maximale Indexeinträge}$$

$$4^3 = 64$$

$$4^{10} = 1048576$$

Bei einer Baumtiefe von 10 sind das 1048576 maximale Einträge (4^{10}).
Je größer die Basis, umso flacher bleibt der Baum. Der Logarithmus kehrt dieses Verhältnis um: die Baumtiefe entspricht also $\log_4$(Indexeinträge)

Baumtiefe	Max-Indexeinträge
3	64
4	256
5	1.024
6	4.096
7	16.384
8	65.536
9	262.144
10	1.048.576

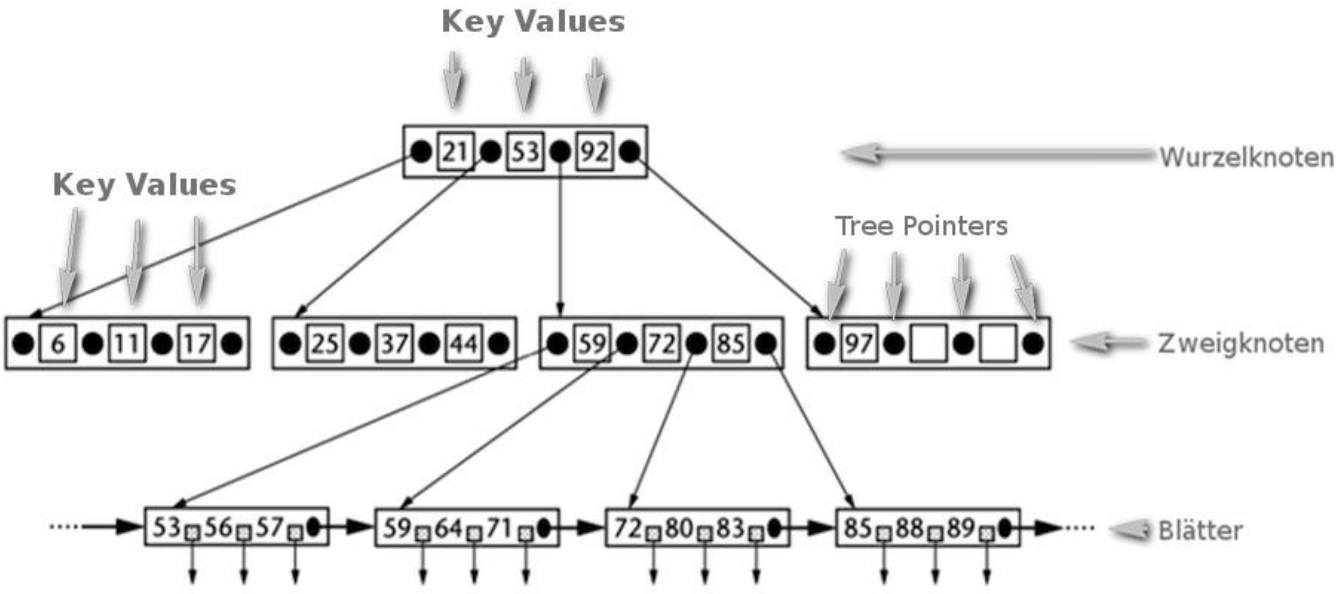
Tree-Height & Tree-Order

Zweigknoten bestehen aus Zeigern (Pointer) und Schlüsselwerten (Key Values). Die Schlüsselwerte sind die Indexdaten, nach denen gesucht wird. Die Zeiger verweisen auf die nächste Baumebene (Level).

Beispiel:

Wenn ein Zweigknoten bis zu 3 Einträge (P-1=3) fassen kann, dann muss dieser Knoten 4 Zeiger besitzen (P=4). Im Beispiel beträgt die Anzahl der Level 3 (Tree-Height).

	Anzahl	
Tree-Pointer	4	Ebenen-Zeiger The variable P represents the order of the B+tree, the fan-out of pointers from one node to the next lower node in the tree.
Key-Values	3	Schlüsselwerte
Level	3	Baumtiefe The height of a B+tree is the level of the leaf nodes, given that the root node is defined as Level 1.



B+Baum mit Compound-Primary-Index

Ein Schlüssel, der aus mehr als einer Spalte besteht, wird zusammengesetzter Schlüssel genannt. Auf englisch **Compound Key** und **Composite Key** genannt.

Einen Index auf die Spalten employee_id und subsidiary_id anlegen:

```
create unique index employee_pk on employees (employee_id, subsidiary_id)
```

Der passende select dazu:

```
select first_name, last_name
  from employees where employee_id = 123 and subsidiary_id = 20
```

Die Datenbank führt nur einen **INDEX UNIQUE SCAN** durch: da es maximal nur einen Wert geben kann, müssen die benachbarten Blattknoten nicht gelesen werden.

Wird nur die Spalte subsidiary_id selektiert, kann der Index nicht greifen.

```
select first_name, last_name from employees where subsidiary_id = 20
```

Da die Spalte subsidiary_id alleine nicht eindeutig ist, muss die Datenbank die Blattknoten verfolgen, um alle Einträge zu finden. Es findet also ein **INDEX RANGE SCAN** statt.

Für dieses Problem gibt es 2 Lösungen:

1) beim create wird die Reihenfolge verändert (subsidiary_id vor employee_id):

```
CREATE UNIQUE INDEX employee_pk ON employees (subsidiary_id ,employee_id)
```

2) oder es wird ein zweiter Index angelegt. Und zwar nur mit der Spalte subsidiary_id :

```
CREATE UNIQUE INDEX employee_pk ON employees (subsidiary_id )
```

Lösung 1 ist zu bevorzugen, da sie Speicher und Ressourcen spart (bei einer Tabellenänderung muß nur ein Index, und nicht zwei aktualisiert werden).

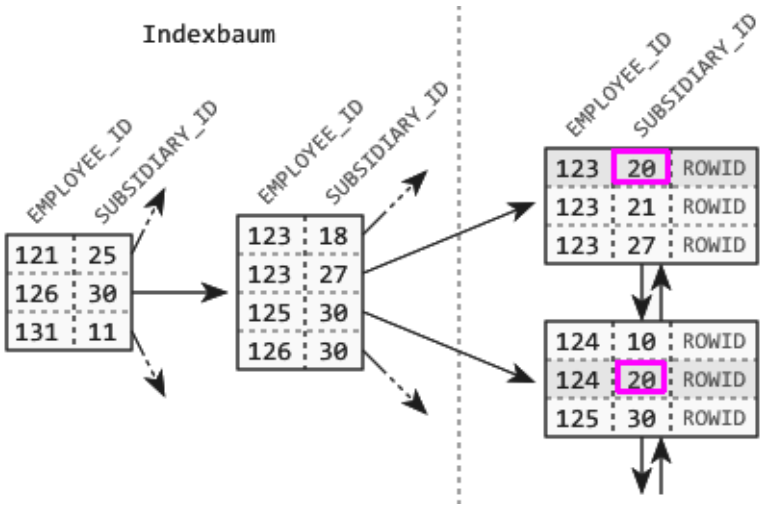
Auch ein zusammengesetzter Index (Compound Index) ist ein B+Tree:

Die erste Spalte wird als vorrangiges Sortierkriterium verwendet.

Die zweite Spalte bestimmt die Reihenfolge nur dann, wenn zwei Einträge denselben Wert in der ersten Spalte haben.

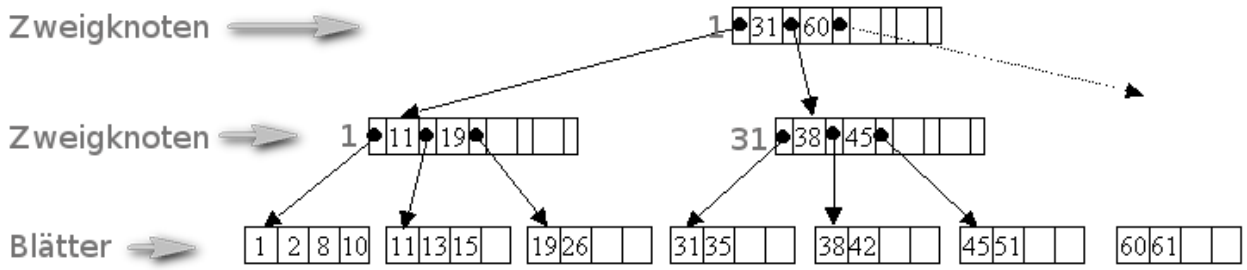
Ein zusammengesetzter Index (Composite Index) kann benutzt werden, wenn mit den führenden Spalten gesucht wird.

Das heißt, ein Index mit drei Spalten kann für Suchen mit der ersten Spalte alleine, mit den ersten beiden oder mit allen drei Spalten gemeinsam verwendet werden.



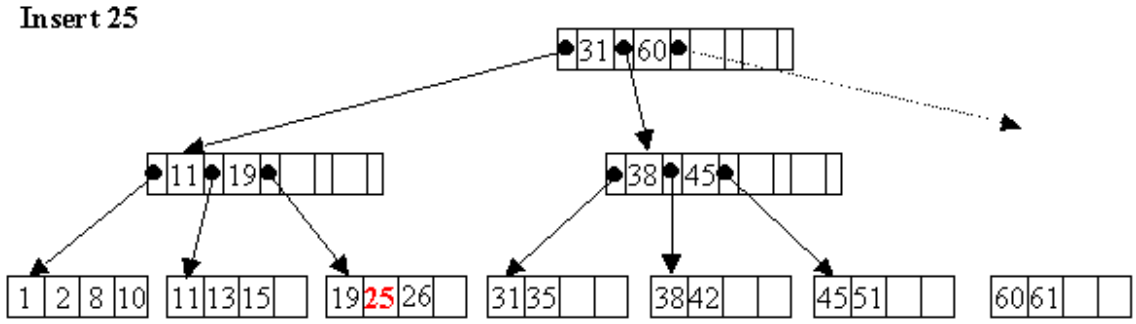
Baumteilung im B+Tree

In der Abbildung sind die oberen 2 Ebenen die Zweige und die unterste Ebene die Blätter des Baumes. Jeder Knoten kann bis zu 4 Einträge fassen. Nur das 1. Blatt ist bereits komplett (ideal ist etwa die Hälfte, also 2 Einträge):



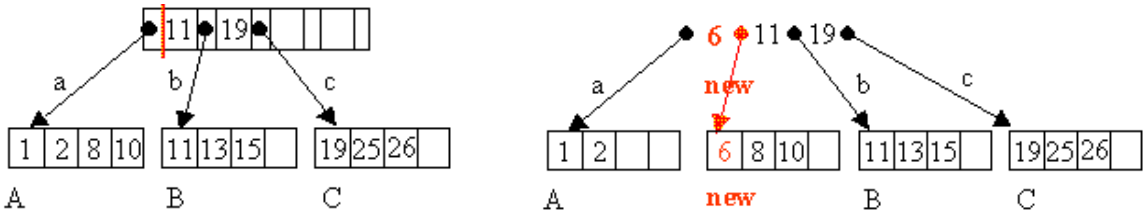
Blatt-Liste erweitern

Für das Hinzufügen vom Wert 25 muß nur das 3. Blatt verändert werden. Die Zahl 25 wird zur **verdoppelt verkettete Liste** hinzugefügt. Danach werden die Zeiger von 19 und 26 angepasst:

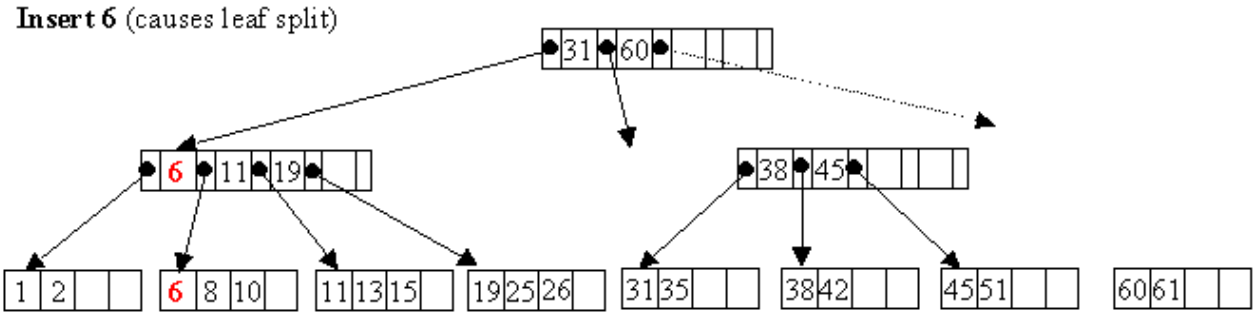


Blattteilung (Leaf Split)

Für das Hinzufügen vom Wert 6 muß das 1. Blatt geteilt werden. Das bedeutet, dass auch ein neues Blatt (Leaf) erzeugt wird. Dieses neue Blatt wird mit etwa der Hälfte der Daten vom bestehenden Blatt befüllt (Zahlen 8 und 10):

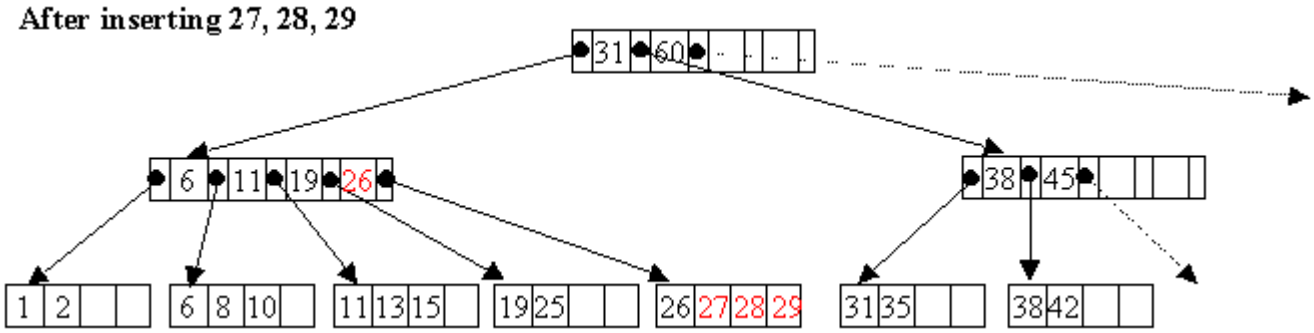


Ergibt:

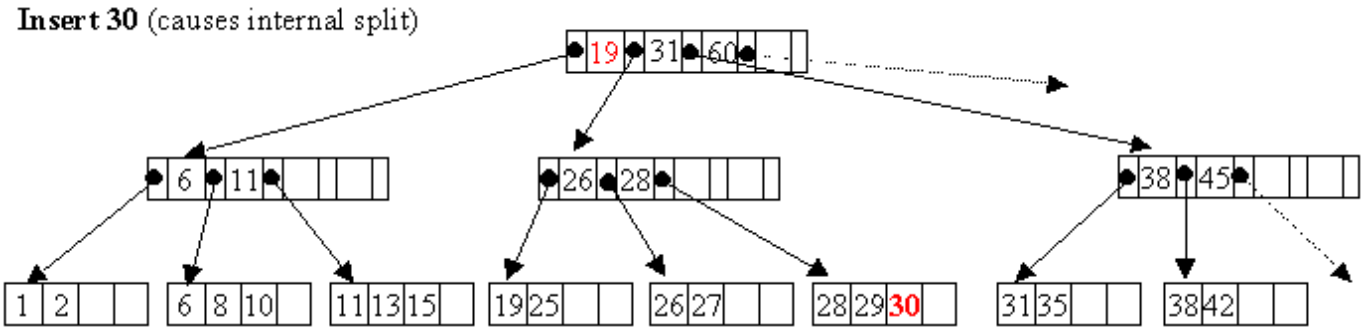


Knotenteilung (Node Split)

Für das Hinzufügen von 27, 28, und 29 muß das 4. Blatt geteilt werden. Für das neues Blatt wird der alte Listenwert 26 mitgenommen.



Für das Hinzufügen von 30 muß der oberste Zweigknoten geteilt werden:



Page Split beim MS-SQL-Server

Der Microsoft-SQL-Server speichert sowohl die Daten, als auch den Index, in Dateien ab. Diese Dateien sind für gewöhnlich in 8 KB großen Speicherseiten unterteilt.

Bei einer gewöhnlichen Tabelle (kein *Clustered Index*) sind die Daten unsortiert in den Speicherseiten zu finden. Dies wird Heap (Haufen) genannt.

Der Server versucht möglichst viele Einträge in jeder Page unterzubringen. Ist der neue Listeneintrag größer als der freie Platz in einer Page, muß die Page aufgeteilt werden. Es entstehen zwei halbvolle Seiten. In diesem Fall muß auch in der Ebene darüber ein neuer Eintrag erzeugt werden. Auch hier kann es zu einem Page Split kommen.

Nachteile: Ein Page Split kostet Zeit. Der Index wird fragmentierter.

Bei Microsoft-SQL-Server kann der Index mit REORGANIZE neu organisiert und mit REBUILD neu erstellt werden.

Bei REBUILD kann optional ein Füllfaktor angegeben werden. Der Füllfaktor bestimmt, bis zu welchem Prozentsatz die Speicherseiten des Index gefüllt werden. Der freigelassene Platz wird genutzt, um die Fragmentierung zu verlangsamen.

```
ALTER INDEX indexname ON tabellenname REORGANIZE

ALTER INDEX indexname ON tabellenname REBUILD WITH (FILLFACTOR = fuellfaktor)
```

Ein Füllfaktor von 80% gibt an, dass beim Erstellen des Index die Speicherseiten nur zu 80% gefüllt werden (20% bleiben für zusätzliche Indexeinträge frei).

Beim Erstellen des Index:

```
CREATE INDEX indexname ON tabellenname (feldliste)
WITH (FILLFACTOR = fuellfaktor)
```

Der Füllfaktor bezieht sich nur auf die Blattebene des Index. Sollen auch die anderen pages diesen Füllfaktor erhalten:

```
CREATE INDEX indexname ON tabellenname (feldliste)
WITH (FILLFACTOR = fuellfaktor, PAD_INDEX = ON)
```

Index-Scan Operationen

Die Zugriffsoperationen bei der Oracle:

Index Unqiue Scan

Ein INDEX UNQIUE SCAN durchwandert nur den Index-Baum. Die Oracle-Datenbank verwendet diese Operation, wenn ein Constraint sicherstellt, dass maximal ein Eintrag dem Abfragekriterium entspricht.

Index Range Scan

Ein INDEX RANGE SCAN durchwandert den Index-Baum und folgt anschließend der Blattknoten-Kette, um alle Treffer zu finden. Diese Operation verwendet Oracle, wenn mehrere Treffer möglich sind.

Table Access By Index Rowid

Lädt eine Zeile aus der Datentabelle anhand einer ROWID aus der Daten-Tabelle. Diese Operation wird für jeden Treffer der vorangegangen Indexoperation durchgeführt.

Für den INDEX RANGE SCAN muss eventuell ein sehr großer Teil des Index gelesen werden. Wenn dann noch ein Tabellenzugriff für jede Daten-Zeile durchgeführt wird, kann eine Abfrage trotz Index sehr langsam sein.

```
-------------------------------------------------------------
|Id |Operation                    | Name         | Rows | Cost |
-------------------------------------------------------------
| 0 |SELECT STATEMENT             |              |    1 |    2 |
| 1 | TABLE ACCESS BY INDEX ROWID| EMPLOYEES    |    1 |    2 |
|*2 |  INDEX UNIQUE SCAN          | EMPLOYEES_PK |    1 |    1 |
-------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("EMPLOYEE_ID"=999)
```

Table Access Full (Full-Table-Scan)

Als Beispiel eine Suche in der Spalte subsidiary_id:

```
SELECT first_name, last_name   FROM employees  WHERE subsidiary_id = 20
```

Oracle-Ausführungsplan

```
-----------------------------------------------------
| Id | Operation         | Name      | Rows | Cost |
-----------------------------------------------------
|  0 | SELECT STATEMENT  |           |  106 |  478 |
|* 1 |  TABLE ACCESS FULL| EMPLOYEES |  106 |  478 |
-----------------------------------------------------
```

Der Ausführungsplan zeigt, dass kein Index benutzt wird. Stattdessen wird eine **Table Access Full**-Operation durchgeführt. Die Datenbank liest die ganze Datentabelle und vergleicht jede Zeile mit der where-Klausel. Dieser Vorgang wird auch **Full-Table-Scan** genannt.

Die Ausführungszeit steigt proportional mit der Tabellengröße.

Der **Full-Table-Scan** kann die beste Zugriffsmethode sein, wenn:

- Die Abfrage sowieso einen großen Teil der Datentabelle zurückgibt.
- Der Indexzugriff einen zu hohen Mehraufwand darstellt, und darum gleich auf die Datentabelle zugreift. Ohne den Umweg von einzelnen Raw-ID-Zugriffen, wird die Datentabelle gleich in mehreren Blöcken gelesen (**Multi Block Read**).

MySQL-Ausführungsplan

```
+----+-----------+------+------+-------+-------------+
| id | table     | type | key  | rows  | Extra       |
+----+-----------+------+------+-------+-------------+
|  1 | employees | ALL  | NULL | 11000 | Using where |
+----+-----------+------+------+-------+-------------+
```

Der MySQL Zugriffstyp ALL zeigt einen Full-Table-Scan an: es werden alle Zeilen gelesen und gegen die where-Klausel geprüft.

PostgreSQL-Ausführungsplan

```
                    QUERY PLAN
-------------------------------------------
 Seq Scan on employees
   (cost=0.00..1398.50 rows=1070 width=14)
   Filter: (subsidiary_id = 2::numeric)
```

Die PostgreSQL-Operation Seq Scan zeigt einen Full-Table-Scan an.

Microsoft SQL-Server-Ausführungsplan

```
|--Table Scan(OBJECT:employees,
              WHERE:subsidiary_id=30)
```

DB2-Server-Ausführungsplan

```
Explain Plan
-------------------------------------------------------------
ID | Operation          |                      Rows | Cost
 1 | RETURN             |                           |  689
 2 |   TBSCAN EMPLOYEES | 1195 of 10000 ( 11.95%) |  689

Predicate Information
 2 - SARG (Q1.SUBSIDIARY_ID = +00002.)
```

Die Operation TBSCAN zeigt bei DB2 an, dass die ganze Tabelle gelesen wird (entspricht Oracle TABLE ACCESS FULL).

Constraint *Primary Key*

Ein constraint ist eine Bedingung (Einschränkung) für einen möglichen Wert einer Variable. Also der erlaubte Wertebereich (Domain). Der constraint kann zum Beispiel beim Definieren einer Tabellenspalte verwendet werden. In diesem Fall ist nur dann das Eintragen eines Wertes möglich, wenn der Skalarwert innerhalb des erlaubten Wertebereichs (Domain) liegt. Beispiele von constraints:

NOT NULL	Der Skalar kann/darf nicht NULL sein
PRIMARY KEY	Der Skalar muss einzigartig (eindeutig) sein und darf nicht NULL sein
FOREIGN KEY	Der Skalar muss auf *Referentielle Integrität* geprüft werden
UNIQUE	Der Skalar muss innerhalb des Attributes einzigartig (eindeutiger Wert) sein

Beispiel:

```
CREATE TABLE person(
    id      NUMERIC,
    name    VARCHAR (90),
    dob     DATE,
    born_in VARCHAR (90),
    CONSTRAINT person_prime PRIMARY KEY(id),
    UNIQUE (name, dob, born_in)
);
```

Primary Key (Primärschlüssel)

Dieser Skalar muß eindeutig sein und darf nicht den Wert NULL haben. Ein Skalar ist eine Variable, die maximal einen Wert beinhalten kann.

Eine Suche nach dem Primärschlüssel darf somit maximal einen Eintrag liefern. Die Datenbank muss also nur den Indexbaum durchwandern und nicht der Blattknoten-Kette folgen.

Beispiel beim Anlegen einer Tabelle:

```
CREATE TABLE employees (
  employee_id   INT, #  employee_id   NUMBER     NOT NUL,
  subsidiary_id   NUMERIC ,
  first_name    VARCHAR (100) NOT NULL,
  CONSTRAINT employees_pk PRIMARY KEY (employee_id)
)
```

Mit der Definition des Primärschlüssels wird automatisch ein Primary-Index auf `EMPLOYEE_ID` angelegt. Weder die Spaltendefinition UNIQUE, noch ein CREATE UNIQUE INDEX ist nowendig.

Primärschlüssel ohne Unique Index

Ein Primärschlüssel erfordert nicht zwingend einen Unique Index – man kann dafür auch einen Non-Unique Index verwenden. Das führt dazu, dass die Oracle einen INDEX RANGE SCAN verwendet.

Einer der Gründe einen Non-Unique Index für einen Primärschlüssel zu verwenden, sind verzögerte Constraints (deferrable constraints). Im Gegensatz zu normalen Constraints werden verzögerte erst geprüft, wenn die Transaktion mittels commit abgeschlossen wird. Verzögerte Constraints werden benötigt, um Tabellen mit zirkulären Abhängigkeiten zu befüllen.

Beispiel

Die Beispiel-Tabelle employees:

```
CREATE TABLE employees (
    employee_id   NUMERIC,
    subsidiary_id   NUMERIC ,
    first_name    VARCHAR (100) ,
    last_name     VARCHAR (100) ,
    date_of_birth DATE          ,
    phone_number  VARCHAR (100),
    CONSTRAINT employees_pk  PRIMARY KEY (employee_id)
)
```

Mit der Definition des Primärschlüssels wird automatisch ein Index auf die Spalte employee_id angelegt.
Somit ist für den Primärschlüssel keine explizite create index-Anweisung notwendig. Auch die Angabe von unique ist optional und nicht zwingend.

Der Primärschlüssel garantiert, dass die Abfrage maximal einen Eintrag liefert.
Eine Zeile mit bereits existierender employeed_id einzutragen, wird von der Datenbank verhindert:

```
doppelter Schlüsselwert verletzt Unique-Constraint „employees_pk“
Duplicate entry '22' for key 'PRIMARY'
```

Diese SQL-Abfrage sucht einen Angestellten mit dem Primärschlüssel:

```
SELECT first_name, last_name   FROM employees  WHERE employee_id = 123
```

Oracle Ausführungsplan

```
-------------------------------------------------------------------
|Id |Operation                    | Name          | Rows | Cost |
-------------------------------------------------------------------
| 0 |SELECT STATEMENT             |               |    1 |    2 |
| 1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES     |    1 |    2 |
|*2 |   INDEX UNIQUE SCAN         | EMPLOYEES_PK  |    1 |    1 |
-------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("EMPLOYEE_ID"=123)
```

Der Ausführungsplan der Oracle zeigt einen INDEX UNIQUE SCAN an:

Die Oracle durchwandert die Ebenen vom B-Tree (balanced tree).
Die Blattknoten (doppelt verkettete Liste) kann er bereits beim ersten Treffer verlassen und mit der RowID in die Datentabelle employees springen. Dieses Springen in den Heap wird TABLE ACCESS BY INDEX ROWID genannt.

MySQL-Ausführungsplan:

```
----+-----------+-------+---------+---------+------+-------+
| id | table     | type  | key     | key_len | rows | Extra |
+----+-----------+-------+---------+---------+------+-------+
|  1 | employees | const | PRIMARY | 5       |    1 |       |
+----+-----------+-------+---------+---------+------+-------+
```

Der Zugriffstyp const entspricht dem INDEX UNIQUE SCAN der Oracle-Datenbank.

DB2-Ausführungsplan

Der DB2-Server wirft beim last_explained-View zurück:

```
Explain Plan
-----------------------------------------------------------
ID | Operation             |                 Rows | Cost
 1 | RETURN                |                      |   13
 2 |  FETCH EMPLOYEES      |    1 of 1 (100.00%) |   13
 3 |   IXSCAN EMPLOYEES_PK | 1 of 1000 (   .10%) |    6

Predicate Information
 3 - START (Q1.EMPLOYEE_ID = +00123.)
      STOP (Q1.EMPLOYEE_ID = +00123.)
```

Die Operation IXSCAN entspricht einem

- Oracle INDEX RANGE SCAN oder einem
- Oracle INDEX UNIQUE SCAN

Somit ist in diesem explain nicht eindeutig ersichtlich, ob es ein Range-Scan oder ein Unique-Scan ist. Die FETCH-Operation stellt den Tabellenzugriff dar. Dieser Zugriff in den Heap wird bei Oracle TABLE ACCESS BY INDEX ROWID genannt.

PostgreSQL-Ausführungsplan

```
                          QUERY PLAN
-----------------------------------------
 Index Scan using employees_pk on employees
   (cost=0.00..8.27 rows=1 width=14)
   Index Cond: (employee_id = 123::numeric)
```

Die PostgreSQL Operation Index Scan ist eine Kombination der Oracle-Operationen INDEX [UNIQUE/RANGE] SCAN und TABLE ACCESS BY INDEX ROWID. An einem PostgreSQL Ausführungsplan kann man nicht erkennen, ob der Indexzugriff mehr als eine Zeile liefern könnte.

Microsoft SQL-Server

```
|--Nested Loops(Inner Join)
   |--Index Seek(OBJECT:employees_pk,
   |               SEEK:employees.employee_id=@1
   |            ORDERED FORWARD)
   |--RID Lookup(OBJECT:employees,
                   SEEK:Bmk1000=Bmk1000
                LOOKUP ORDERED FORWARD)
```

Die **SQL Server** Operationen INDEX SEEK und RID Lookup entsprechen den Oracle-Operationen INDEX RANGE SCAN und TABLE ACCESS BY ROWID. Im Gegensatz zu Oracle zeigt der SQL-Server-Ausführungsplan das Zusammenführen der Index- und Tabellendaten mit dem Nested Loops Join explizit an.

Compound Primary Key

Ein zusammengesetzter Schlüssel (Composite Key) besteht aus mehr als einer Spalte.

Die Tabelle employees:

```
CREATE TABLE employees (
    employee_id   NUMERIC  ,
    subsidiary_id   NUMERIC ,
    first_name    VARCHAR (100),
    last_name     VARCHAR (100),
    date_of_birth DATE,
    phone_number  VARCHAR (100)
)
```

Das Anlegen eines Unique-Index auf die zwei Spalten employee_id und subsidiary_id:

```
CREATE UNIQUE INDEX employee_pk
  ON employees (employee_id, subsidiary_id)
```

Beispiel 1: der Index wird verwendet

Die Reihenfolge der Spalten in der SQL-Anweisung muß ident mit der Reihenfolge der Index-Spalten sein.

```
SELECT first_name, last_name FROM employees WHERE employee_id =123 AND subsidiary_id = 20
```

Beispiel 2: der Index wird nicht verwendet

Ein select nur auf die Spalte subsidiary_id:

```
SELECT first_name, last_name FROM employees WHERE subsidiary_id = 20
```

Der Index wird nicht verwendet. Für dieses select gibt es 2 Lösungen.

Lösung 1 (andere Spaltenreihenfolge)

Beim create wird die Reihenfolge verändert (subsidiary_id vor employee_id):

```
CREATE UNIQUE INDEX employee_pk ON employees (subsidiary_id ,employee_id)
```

Lösung 2 (zweiter Index)

Es wird ein zweiter Index angelegt. Und zwar nur mit der Spalte subsidiary_id :

```
CREATE UNIQUE INDEX employee_pk ON employees (subsidiary_id )
```

Ein zusammengesetzter Index (Compound Index) kann benutzt werden, wenn mit den führenden Spalten gesucht wird.

Das bedeutet: Ein Index mit drei Spalten kann für das Suchen

- mit der ersten Spalte alleine,
- mit den ersten beiden Spalten,
- oder mit allen drei Spalten gemeinsam verwendet werden.

Beispiel 1 im Detail

Die Select-Anweisung auf employee_id und subsidiary_id:

```
SELECT first_name, last_name
  FROM employees
 WHERE employee_id    = 123
   AND subsidiary_id = 30
```

Oracle-Ausführungsplan

```
-------------------------------------------------------------------
|Id |Operation                    | Name         | Rows | Cost |
-------------------------------------------------------------------
| 0 |SELECT STATEMENT             |              |    1 |    2 |
| 1 | TABLE ACCESS BY INDEX ROWID| EMPLOYEES    |    1 |    2 |
|*2 |  INDEX UNIQUE SCAN          | EMPLOYEES_PK |    1 |    1 |
-------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("EMPLOYEE_ID"=123 AND "SUBSIDIARY_ID"=30)
```

Wenn der Zugriff mit dem vollständigen Primärschlüssel erfolgt, kann die Datenbank einen INDEX UNIQUE SCAN durchführen – egal wie viele Spalten der Index hat.

MySQL-Ausführungsplan

```
+----+-----------+-------+---------+---------+------+-------+
| id | table     | type  | key     | key_len | rows | Extra |
+----+-----------+-------+---------+---------+------+-------+
|  1 | employees | const | PRIMARY | 10      |    1 |       |
+----+-----------+-------+---------+---------+------+-------+
```

Wie zuvor (Kapitel »Primary Key«) verwendet MySQL den Zugriffstypen const, da nach einer Konstanten gesucht wird. Die verwendete Schlüssellänge (key_len) ist jedoch auf 10 gestiegen, da nun zwei Spalten des Index benutzt werden.

PostgreSQL-Ausführungsplan

```
                    QUERY PLAN
-------------------------------------------
 Index Scan using employees_pk on employees
   (cost=0.00..8.27 rows=1 width=14)
   Index Cond: ((employee_id    = 123::numeric)
            AND (subsidiary_id = 30::numeric))
```

DB2-Ausführungsplan

```
Explain Plan
-------------------------------------------------------
ID | Operation             |                  Rows | Cost
 1 | RETURN                |                       |   13
 2 |  FETCH EMPLOYEES      |     1 of 1 (100.00%) |   13
 3 |   IXSCAN EMPLOYEES_PK | 1 of 10000 (   .01%) |    6

Predicate Information
 3 - START (Q1.EMPLOYEE_ID = +00123.)
     START (Q1.SUBSIDIARY_ID = +00030.)
      STOP (Q1.EMPLOYEE_ID = +00123.)
      STOP (Q1.SUBSIDIARY_ID = +00030.)
```

Microsoft SQL-Server-Ausführungsplan

```
|--Nested Loops(Inner Join)
   |--Index Seek(OBJECT:employees_pk,
   |               SEEK:employee_id=@1 AND subsidiary_id=@2
   |            ORDERED FORWARD)
   |--RID Lookup(OBJECT:employees,
                 SEEK:Bmk1000=Bmk1000
               LOOKUP ORDERED FORWARD)
```

Beispiel 2 im Detail

Die Select-Anweisung nur auf subsidiary_id:

```
SELECT first_name, last_name  FROM employees WHERE subsidiary_id = 20
```

Der Index greift nicht, da die Spalte subsidiary_id die 2. Spalte im Index ist. Wäre sie die 1. Spalte, würde der Index ziehen.

Oracle-Ausführungsplan

```
-------------------------------------------------
| Id | Operation         | Name      | Rows | Cost |
-------------------------------------------------
|  0 | SELECT STATEMENT  |           |  106 |  478 |
|* 1 |  TABLE ACCESS FULL| EMPLOYEES |  106 |  478 |
-------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("SUBSIDIARY_ID"=20)
```

Der Ausführungsplan zeigt, dass der Index nicht benutzt wird. Stattdessen wird eine TABLE ACCESS FULL-Operation durchgeführt.

TABLE ACCESS FULL-Operation

Dabei liest die Datenbank die ganze Tabelle und vergleicht jede Zeile mit der where-Klausel. Die Ausführungszeit steigt proportional mit der Tabellengröße. Wenn die Tabelle auf das Zehnfache wächst, dauert der TABLE ACCESS FULL auch zehn Mal so lange.

MySQL-Ausführungsplan

```
+----+-----------+------+------+-------+-------------+
| id | table     | type | key  | rows  | Extra       |
+----+-----------+------+------+-------+-------------+
|  1 | employees | ALL  | NULL | 11000 | Using where |
+----+-----------+------+------+-------+-------------+
```

Der MySQL Zugriffstyp ALL zeigt einen Full-Table-Scan an: es werden alle Zeilen gelesen und gegen die **where**-Klausel geprüft.

PostgreSQL-Ausführungsplan

```
                      QUERY PLAN
-----------------------------------------
 Seq Scan on employees
   (cost=0.00..1398.50 rows=1070 width=14)
   Filter: (subsidiary_id = 2::numeric)
```

Die PostgreSQL-Operation Seq Scan zeigt einen Full-Table-Scan an.

Microsoft SQL-Server-Ausführungsplan

```
|--Table Scan(OBJECT:employees,
              WHERE:subsidiary_id=30)
```

DB2-Ausführungsplan

```
Explain Plan
-------------------------------------------------------
ID | Operation         |                    Rows | Cost
 1 | RETURN            |                         |  689
 2 |  TBSCAN EMPLOYEES | 1195 of 10000 ( 11.95%) |  689

Predicate Information
 2 - SARG (Q1.SUBSIDIARY_ID = +00002.)
```

Die Abfrage wurde mit SUBSIDIARY_ID = 2 ausgeführt, um das gewünschte Ergebnis zu erhalten.

Die Operation TBSCAN zeigt bei DB2 an, dass die ganze Tabelle gelesen wird (entspricht Oracle TABLE ACCESS FULL).

Lösung 1 im Detail

Beim create wird die Reihenfolge verändert: subsidiary_id vor employee_id:

```
CREATE UNIQUE INDEX employee_pk ON employees (subsidiary_id ,employee_id)
```

Oracle-Ausführungsplan

Der Ausführungsplan zeigt, dass der umgedrehte Index benutzt wird:

```
-------------------------------------------------------------
|Id |Operation                    | Name        | Rows | Cost |
-------------------------------------------------------------
| 0 |SELECT STATEMENT             |             |  106 |   75 |
| 1 | TABLE ACCESS BY INDEX ROWID| EMPLOYEES   |  106 |   75 |
|*2 |  INDEX RANGE SCAN           | EMPLOYEE_PK |  106 |    2 |
-------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("SUBSIDIARY_ID"=20)
```

Da die Spalte `SUBSIDIARY_ID` alleine nicht eindeutig ist, muss die Datenbank die Blattknoten verfolgen, um alle Einträge zu finden. Es findet also ein `INDEX RANGE SCAN` statt.

MySQL-Ausführungsplan

```
+----+-----------+------+---------+---------+------+-------+
| id | table     | type | key     | key_len | rows | Extra |
+----+-----------+------+---------+---------+------+-------+
|  1 | employees | ref  | PRIMARY | 5       |  123 |       |
+----+-----------+------+---------+---------+------+-------+
```

Der MySQL-Zugriffstyp ref enspricht einem INDEX RANGE SCAN der Oracle-Datenbank.

PostgreSQL-Ausführungsplan

```
                    QUERY PLAN
-----------------------------------------------
 Bitmap Heap Scan on employees
 (cost=24.63..1529.17 rows=1080 width=13)
   Recheck Cond: (subsidiary_id = 2::numeric)
   -> Bitmap Index Scan on employees_pk
      (cost=0.00..24.36 rows=1080 width=0)
      Index Cond: (subsidiary_id = 2::numeric)
```

PostgreSQL führt hier zwei Operationen durch: einen **Bitmap Index Scan** gefolgt von einem **Bitmap Heap Scan**. Diese Operationen entsprechen ungefähr dem INDEX RANGE SCAN und dem TABLE ACCESS BY INDEX ROWID.

Der Unterschied ist, dass das der Bitmap Index Scan zuerst alle passenden Zeilen aus dem Index holt, diese dann entsprechend ihrer physischen Speicherposition in der Tabelle sortiert und dann erst den Zugriff auf die Tabelle durchführt (Bitmap Heap Scan).

DB2-Ausführungsplan

```
Explain Plan
-----------------------------------------------------------------
ID | Operation               |                     Rows | Cost
 1 | RETURN                  |                          |  128
 2 |  FETCH EMPLOYEES        |  1195 of 1195 (100.00%) |  128
 3 |   RIDSCN                |  1195 of 1195 (100.00%) |   43
 4 |    SORT (UNQIUE)        |  1195 of 1195 (100.00%) |   43
 5 |     IXSCAN EMPLOYEES_PK | 1195 of 10000 ( 11.95%) |   43

Predicate Information
 2 - SARG (Q1.SUBSIDIARY_ID = +00002.)
 5 - START (Q1.SUBSIDIARY_ID = +00002.)
      STOP (Q1.SUBSIDIARY_ID = +00002.)
```

die IXSCAN-Operation für den Indexzugriff und die FETCH-Operation für den Tabellenzugriff.

Dazwischen tauchen jedoch die unerwarteten Operationen SORT und RIDSCAN auf:
die SORT-Operation sortiert die Indexeinträge entsprechend der physischen Speicheraddresse der zugehörigen Zeile in der Heap-Tabelle – sinngemäß entsprechend der ROWID, die bei DB2 RID genannt wird.
Die RIDSCN-Operation lädt dann alle betroffenen Datenseiten aus der Heap-Tabelle, wobei aufeinanderfolgende Blöcke in eine Leseoperation zusammengefasst werden.

Optimzier (Ausführungsplaner)

Der Optimizer wird auch **Query Planner** genannt. Er erstellt den Ausführungsplan für eine SQL-Anweisung. Es gibt 2 Varianten:

Regelbasierter Optimizer (RBO)

Der Rule-Based Optimizer (RBO) wählt nur nach fixen Vorgaben die optimale Variante.

Kostenbasierter Optimizer (CBO)

Der Cost Based Optimizer (CBO) erstellt mehrere Pläne, bewertet sie und entscheidet sich situationsbedingt für die optimale Variante.

Ein kostenbasierter Optimizer nutzt Statistiken auf Tabellen-, Index- und Spalten-Ebene. Die meisten Statistiken werden auf Spaltenebene geführt: die Anzahl der unterschiedlichen Werte, der kleinste und größte Wert, die Zahl der NULL-Einträge und das Histogramm, das die Werteverteilung anzeigt.

Auf Tabellenebene sind es Tabellengröße (in Zeilen und Blöcken) und die durchschnittliche Zeilenlänge.

Die wichtigsten Index-Statistiken sind die Tiefe des Baumes, die Zahl der Blattknoten, die Zahl der unterschiedlichen Einträge und der Clustering-Faktor.

Der Clustering-Faktor eines Indexes ist ein indirektes Maß für die Wahrscheinlichkeit, mit der zwei aufeinanderfolgende Indexeinträge auf denselben Tabellenblock verweisen. Der Optimizer berücksichtigt diese Wahrscheinlichkeit bei der Berechnung des Cost-Wertes einer `TABLE ACCESS BY INDEX ROWID`-Operation.

Als Beispiel ein select in der Oracle::

```
SELECT first_name, last_name, subsidiary_id, phone_number
  FROM employees  WHERE last_name  = 'WINAND' AND subsidiary_id = 30
```

Ergibt diesen Ausführungsplan:

```
-------------------------------------------------------------------
|Id |Operation                    | Name         | Rows | Cost |
-------------------------------------------------------------------
| 0 |SELECT STATEMENT             |              |    1 |   30 |
|*1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES    |    1 |   30 |
|*2 |   INDEX RANGE SCAN          | EMPLOYEES_PK |   40 |    2 |
-------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
  1 - filter("LAST_NAME"='WINAND')
  2 - access("SUBSIDIARY_ID"=30)
```

Daraus läßt sich ablesen:

INDEX RANGE SCAN

Es wird ein Index verwendet. Und zwar ein Index mit dem Namen EMPLOYEES_PK.
Die Angabe **RANGE SCAN** bedetet, dass die Blattknoten durchwandert werden: der gesuchte Indexwert ist also nicht unique.

TABLE ACCESS BY INDEX

Bei einem TABLE ACCESS BY INDEX sind keine oder nicht alle gewünschten Daten in den Blättern vorhanden. Diese müssen also von der eigentlichen Datentabelle (Heap) geholt werden.

Funktionsbasierender Index (Oracle)

Ein Index, dessen Definition Funktionen oder Ausdrücke enthält, wird **Funktionsbasierender Index (FBI) genannt.** Dabei werden nicht die original Tabellendaten in den Index kopiert, sondern das Ergebnis der Funktion.

Microsoft SQL-Server und MySQL 5.7 können generierte Spalten (**computed columns**) indizieren. Als Alternative kann man bei älteren Versionen der Tabelle eine neue Spalte hinzufügen, die das Ergebnis des Funktionsausdruckes enthält. Diese Spalte muss man bei Änderungen aktuell halten. Zum Beispiel mit einem Datenbank-Trigger. Diese neue Spalte kann normal indiziert und in der where-Klausel verwendet werden.

Computed Columns in SQL-Server:

```
ALTER TABLE employees ADD last_name_up AS UPPER(last_name);
CREATE INDEX emp_up_name ON employees (last_name_up);
```

Computed Columns in MySQL:

```
ALTER TABLE employees
  ADD COLUMN last_name_up VARCHAR(255) AS (UPPER(last_name));
CREATE INDEX emp_up_name ON employees (last_name_up);
```

Die Oracle Datenbank unterstützt funktions-basierende Indizes seit Release 8i. Virtuelle Spalten (virtual column) wurden mit Version 11g eingeführt.

System-Funktion

Als Beispiel eine select in der Oracle. Die Suche soll unabhängig von der Groß-Kleinschreibung sein:

```
SELECT first_name, last_name, phone_number
  FROM employees
 WHERE UPPER(last_name) = UPPER('winand')
```

Mit der Funktion UPPER ist sowohl der Suchbegriff, als auch die Suchspalte (last_name) großgeschrieben. Somit spielt die Groß-Kleinschreibung keine Rolle mehr. Auf der Spalte last_name gibt es einen Index:

```
CREATE INDEX emp_up_name
    ON employees (last_name)
```

Der **Oracle**-Ausführungsplan dazu:

```
-----------------------------------------------------
| Id | Operation         | Name      | Rows | Cost |
-----------------------------------------------------
|  0 | SELECT STATEMENT  |           |   10 |  477 |
|* 1 |  TABLE ACCESS FULL| EMPLOYEES |   10 |  477 |
-----------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter(UPPER("LAST_NAME")='WINAND')
```

Die Oracle zeigt eine **TABLE ACCESS FULL** Operation an, der Index wird also nicht verwendet. Der Optimizer kann zwar den rechten Ausdruck `'WINAND'` interpretieren, aber nicht den linken (siehe Angaben in **Predicate Information**).

Die Lösung ist ein funktionsbasierender Index:

```
CREATE INDEX emp_up_name
    ON employees (UPPER(last_name))
```

Nach dem Erstellen des FBI zeigt der **Oracle**-Ausführungsplan:

```
-------------------------------------------------------------
|Id |Operation                    | Name         | Rows | Cost |
-------------------------------------------------------------
| 0 |SELECT STATEMENT             |              |  100 |   41 |
| 1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES    |  100 |   41 |
|*2 |   INDEX RANGE SCAN          | EMP_UP_NAME  |   40 |    1 |
-------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
  2 - access(UPPER("LAST_NAME")='WINAND')
```

Es wird ein Index Range Scan angezeigt. Das bedeutet, dass ein Index verwendet wird (namens EMP_UP_NAME).
Der Oracle-Optimizier liegt bei der Schätzung der Zeilen falsch: rows 40 und rows 100:

Statistik-Problem des Optimizers

Die Statistiken über die Anzahl der verschiedenen Werte jeder Spalte werden von der Oracle Datenbank auf Tabellenebene geführt. Dadurch können diese Werte für mehrere Indizes genutzt werden (falls eine Spalte mehrfach indiziert ist).
Die FBI-Statistik wird mittels virtueller Spalten ebenfalls auf Tabellenebene geführt. Die Oracle Datenbank ermittelt zwar die Index-Statistiken beim Anlegen des Indexes ab Version 10g automatisch, aktualisiert die Tabellen-Statistiken aber nicht.

Nach der Index-Aktualisierung:

```
-------------------------------------------------------------
|Id |Operation                    | Name         | Rows | Cost |
-------------------------------------------------------------
| 0 |SELECT STATEMENT             |              |    1 |    3 |
| 1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES    |    1 |    3 |
|*2 |   INDEX RANGE SCAN          | EMP_UP_NAME  |    1 |    1 |
-------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
  2 - access(UPPER("LAST_NAME")='WINAND')
```

Statistiken auf Ausdrücke und Spaltenkombinationen wurden mit Oracle release 11g eingeführt.

Exkurs

Eine andere Möglichkeit die Groß- und Kleinschreibung zu ignorieren ist, die Sortierung einer Spalte zu ändern. SQL-Server und MySQL verwenden in der Standard-Konfiguration Collations, die nicht zwischen Groß- und Kleinbuchstaben unterschieden.

Benutzerdefinierte Funktion

Um für einen Index brauchbar zu sein, muß die Funktion **deterministisch** sein: sie muß bei gleichen Eingabewerten immer das gleiche Ergebnis zurückgeben.

Eine **nicht deterministisch** Funktion ist eine Altersberechnung:

```
CREATE FUNCTION get_age(date_of_birth DATE)
RETURN NUMBER
AS
BEGIN
  RETURN
    TRUNC(MONTHS_BETWEEN(SYSDATE, date_of_birth)/12);
END
```

Die Funktion GET_AGE benutzt das aktuelle Datum (SYSDATE), um das Alter zu berechnen

Der Datenbank muß man explizit mitteilen, dass es sich um eine deterministisch Funktion handelt. Bei der Oracle lautet das Keyword **DETERMINISTIC**, bei der PostgreSQL **IMMUTABLE**.

Computed Columns (Microsoft)

Eine berechnete Spalte wird aus einem Ausdruck berechnet, der andere Spalten in derselben Tabelle verwenden kann. Der Ausdruck kann

- der Name einer nicht berechneten Spalte,
- eine Konstante,
- eine Funktion
- oder eine beliebige durch einen oder mehrere Operatoren verbundene Kombination der genannten Möglichkeiten sein.

Der Ausdruck kann keine Unterabfrage sein.

Wenn nichts anderes angegeben wird, handelt es sich bei berechneten Spalten um virtuelle Spalten, die nicht physisch in der Tabelle gespeichert sind. Die darin enthaltenen Werte werden jedes Mal neu berechnet, wenn in einer Abfrage darauf verwiesen wird.

Mit dem Schlüsselwort PERSISTED in den Anweisungen CREATE TABLE und ALTER TABLE, werden berechnete Spalten physisch in der Tabelle gespeichert. Die enthaltenen Werte werden aktualisiert, wenn sich Spalten ändern, die Teil der Berechnung sind.

Berechnete Spalten können in

- Auswahllisten
- WHERE-Klauseln
- ORDER BY-Klauseln
- an Stellen, an denen reguläre Ausdrücke verwendet werden können

Eine berechnete Spalte kann nicht das Ziel einer INSERT- oder UPDATE-Anweisung sein.

Man kann einen Index für eine berechnete Spalte erstellen.

Parametisierte Abfragen (Bind-Variablen)

Patzhalter ersetzen echte Werte in der SQL-Anweisung. Als Platzhalter wird meistens das Fragezeichen verwendet:

?	Fragezeichen (SQL-Standard)
:*name*	Doppelpunkt (proprietär), gefolgt vom Namen des Platzhalters
@*name*	Klammeraffe (proprietär) gefolgt vom Namen des Platzhalters

Im SQL-Standard ist nur das Fragezeichen als Platzhalter vorgesehen.

Die tatsächlichen Werte bekommt die Datenbank über einen (weiteren) API-Aufruf.
Drei übliche Bezeichnungen für die Platzhalter:

Bind-Parameter

dynamische Parameter

Bind-Variablen

Die Verwendung von Bind-Variablen kann 2 Vorteilen bringen:

1. Erschweren einer SQL-Intrusion
2. Performancegewinn der SQL-Abfrage

Für gewöhnlich sieht der Optimizer nur die Platzhalter und keine echten Werte. Somit gibt es kein Histogramm und der Optimizer gelangt immer zur selben Zeilenschätzung und zum selben Cost-Wert. Histogramme spielen ihren Vorteil bei ungleicher Verteilung aus. Bei einer gleichmäßigen Verteilung genügt es meist, die Zahl der unterschiedlichen Werte durch die Anzahl der Zeilen insgesamt zu dividieren.

Datenbanken mit einem Ausführungsplan-Cache (zB SQL-Server, Oracle) können Ausführungspläne wiederverwenden, wenn eine SQL-Anweisung mehrfach ausgeführt wird. Dadurch kann der Aufwand, einen Ausführungsplan zu erstellen, vermieden werden. Dieser Mechanismus funktioniert aber nur, wenn genau dieselbe SQL-Anweisung wiederholt ausgeführt wird. Das ist bei Bind-Variablen nicht möglich.

SQL-Server und Oracle bieten auch eine automatisierte Parametiesirung an. Nicht der User, sondern die Datenbank ersetz die Werte durch Platzhalter.
Diese Funktion heißt bei **Oracle CURSOR_SHARING** und bei SQL-Server **forced parameterization.**

PHP

MySQL-Zugriff ohne Bind-Parameter:

```
$mysqli->query("select first_name, last_name"
             . "  from employees"
             . " where subsidiary_id = " . $subsidiary_id);
```

Mit Bind-Parameter:

```
if ($stmt = $mysqli->prepare("select first_name, last_name"
                           . "  from employees"
                           . " where subsidiary_id = ?"))
 {
    $stmt->bind_param("i", $subsidiary_id);
    $stmt->execute();
 } else {
       /* SQL error */
    }
```

Siehe php.net und php.net

Perl

Ohne Bind-Parameter:

```
my $subsidiary_id;
my $sth = $dbh->prepare(
                  "select first_name, last_name"
                . "  from employees"
                . " where subsidiary_id = $subsidiary_id"
                );
$sth->execute();
```

Mit Bind-Parameter:

```
my $subsidiary_id;
my $sth = $dbh->prepare(
                  "select first_name, last_name"
                . "  from employees"
                . " where subsidiary_id = ?"
                );
$sth->execute($subsidiary_id);
```

Siehe cpan.org

Ruby

Ohne Bind-Parameter:

```
dbh.execute("select first_name, last_name"
          + "  from employees"
          + " where subsidiary_id = #{subsidiary_id}");
```

Mit Bind-Parameter:

```
dbh.prepare("select first_name, last_name"
          + "  from employees"
          + " where subsidiary_id = ?");
dbh.execute(subsidiary_id);
```

Siehe kitebird.com

JAVA

Ohne Bind-Parameter:

```
int subsidiary_id;
Statement command = connection.createStatement(
                    "select first_name, last_name"
                  + "  from employees"
                  + " where subsidiary_id = " + subsidiary_id
                  );
```

Mit Bind-Parameter:

```
int subsidiary_id;
PreparedStatement command = connection.prepareStatement(
                    "select first_name, last_name"
                  + "  from employees"
                  + " where subsidiary_id = ?"
                  );
command.setInt(1, subsidiary_id);
```

Siehe oracle.com

C#

Ohne Bind-Parameter:

```
int subsidiary_id;
SqlCommand cmd = new SqlCommand(
                   "select first_name, last_name"
                 + "  from employees"
                 + " where subsidiary_id = " + subsidiary_id
                 , connection);
```

Mit Bind-Parameter:

```
int subsidiary_id;
SqlCommand cmd =
       new SqlCommand(
                     "select first_name, last_name"
                   + "  from employees"
                   + " where subsidiary_id = @subsidiary_id
                   , connection);
cmd.Parameters.AddWithValue("@subsidiary_id", subsidiary_id);
```

Siehe microsoft.com

Prädikate

Die Oracle Datenbank kann where-Bedingungen (Prädikate) auf drei verschiedene Arten anwenden:

5. **Zugriffsprädikat** access

   Die Zugriffsprädikate bestimmen den Anfangs- und End-Punkt beim Durchlaufen der Blattknote-Liste.

6. **Index-Filterprädikat** filter

   Index-Filterprädikate werden während des Durchsuchens der Blattknoten-Liste verwendet.
   Sie haben aber keinen Einfluss auf die Start- und Stopp-Bedingungen und grenzen den durchsuchten Indexbereich nicht ein.

7. **Tabellen-Filterprädikat** **filter**

   Prädikate, die sich auf Spalten beziehen, die nicht im Index sind, können erst auf Tabellenebene berücksichtigt werden. Dafür muss zuerst ein Tabellenzugriff durchgeführt werden.

Ausführungspläne, die mit DBMS_XPLAN (Oracle) erstellt werden, zeigen die Prädikatstypen in der **Predicate Information:**

```
-----------------------------------------------------
| Id | Operation         | Name       | Rows  | Cost |
-----------------------------------------------------
|  0 | SELECT STATEMENT  |            |     1 | 1445 |
|  1 |  SORT AGGREGATE   |            |     1 |      |
|* 2 |   INDEX RANGE SCAN| SCALE_SLOW |  4485 | 1445 |
-----------------------------------------------------

Predicate Information (identified by operation id):
   2 - access("SECTION"=:A AND "ID2"=:B)
       filter("ID2"=:B)
```

Die Zahl vor den Prädikatstypen bezieht sich auf die Spalte Id.
Der Stern * bedeutet, dass es zu dieser Operation Prädikatsinformationen gibt.

Siehe ID2=:B Sobald ein Prädikat einmal als Filterprädikat aufscheint, ist und bleibt es auch ein Filterprädikat (auch wenn es zusätzlich als Zugriffsprädikat ausgewiesen wird).

MySQL-Ausführungsplan

Spalte	JSON-Name	Beschreibung
id	select_id	The SELECT identifier explain erzeugt einen Ausgabe-Record für jede Tabelle der Query.
select_type		Mögliche Werte: SIMPLE PRIMARY UNION, DEPENDE UNION SUBSELECT DERIVED
table	table_name	The table for the output row Der Name der Tabelle, aus der MySQL die Datensätze einliest.
partitions	partitions	The matching partitions
type	access_type	die Art des Joins. const (ein konstanter Wert kommt in der Query vor) system eq_ref ref_range index ALL
possible_keys	possible_keys	The possible indexes to choose Eine Liste der Indizes, die MySQL für die Suche verwenden kann. Null, wenn es keinen möglichen Index gibt.
key	key	The index actually chosen
		Der Name des Index, für den sich MySQL entschieden hat. Die möglichen Indizies sind in possible_key aufgelistet.
key_len	key_length	The length of the chosen key Die Größe des Schlüssels in Byte.
ref	ref	The columns compared to the index Die Spalten oder Werte, mit denen der Schlüssel verglichen werden soll.
rows	rows	Estimate of rows to be examined Die Anzahl der Datensätze, die MySQL glaubt untersuchen zu müssen, um die Query zu beantworten. Je mehr Tabellenänderungen durchgeführt werden, umso schlechter kann die Abschätzung der MySQL sein. Die ANALYSE TABLE Anweisung kann die Abschätzung verbessern.
filtered	filtered	Percentage of rows filtered by table condition
Extra		Additional information

Stichwortverzeichnis

A-Z

Adjacency ... *Siehe* Adjazenz
Adjazenz ... 30
  Adjazenzmatrix ... 25, 33
ASCII ... 224
  Carriage Return ... 226, 227
  Character Codes ... 228
  Extended ASCII (8bit) ... 230
  File Separator (FS) ... 226, 227
  Line Feed ... 226
Attribut ... 507
  Attributbezeichnung ... 605
  Attributwert ... 507
Ausgangsgrad ... 18, 23
awk
  $0 73
  Action (Aktion) ... 69
  ARGC ... 73
  ARGIND ... 73
  ARGV ... 73
  Arithmetische Funktionen ... 95, 98
  Array ... 78
  asort() ... 96
  asorti() ... 96, 97
  Assignment Operator (Zuweisung) ... 80
  awk Processing Cycle ... 69
  Builtin-Variablen ... 73
  Conditional Operator (Bedingung) ... 80
  CONVFMT ... 73
  delete (Array) ... 79
  ENVIRON (Gawk Environment-Variable) ... 71
  Escape Sequence ... 86, 282, 296, 306, 435
  exit ... 104
  Feld (Zeilenspalte) ... 68
  FILENAME ... 73
  for-Schleife ... 102
  FS 73
  function() ... 99
  getline ... 108
  gsub() ... 92
  IGNORECASE ... 94
  in (Array-Operator) ... 79
  index() ... 92
  Kommandozeilen-Argumente ... 100
  Konkatenation (Verkettung) ... 82
  length() ... 92
  Main Input Loop ... 69
  match() ... 93
  Mehrdimensionales Array ... 78
  mktime() ... 98
  Multiple Delimiter ... 72
  next ... 105
  nextfile ... 105
  NF ... 73
  NR ... 73
  OFS ... 73
  Operatoren ... 80
  Pattern-Action Statement ... 74
  Pipe ... 76
  print ... 87
  printf() ... 88
  Priorität und Assoziativität von Operatoren ... 83
  Record (Datensatz) ... 66
  Regular Expression Operator ... 80
  Regular Expressions ... 85
  Remark ... 75
  return ... 105
  RLENGTH ... 73
  RSTART ... 73
  Shell-Aufruf ... 70
  Skalar ... 77
  split() ... 94
  strftime() ... 89, 98
  String-Funktionen ... 87
  sub() ... 91
  SUBSEP ... 73, 78
  systime() ... 98
  ternäre Operator ... 81
  tolower() ... 92
  toupper() ... 92
  Trennzeichen (Separator) ... 66
  Variablen ... 77
  Vergleich Numeric mit String ... 81
  Vergleich von Nawk, Gawk, AWK ... 111
  while-Schleife ... 103
Azykliksch ... 28
Baum
  Geordneter Wurzelbaum ... 40
  Gewurzelter Baum ... 39
  Spannbaum ... 21, 39
  Suchbaum ... 40
Bayer Tree ... *Siehe* B-Tree
BFS ... *Siehe* Breadth-First Search
Bi-Endian ... 235
Big-Endian ... 235

Binärbaum ... 40
Binary Tree ... 40
Bipartition ... 35
  bipartiter Graphen ... 35
  vollständig bipartiter Graph ... 36
Blatt ... 20
Bogen
  Bogen ... 19
Breadth-First Search ... 37
Breitensuche ... *Siehe* Breath-First-Search
Brücke ... 22
  Brückenkante ... 22
Byte-Order (Byte-Reihenfolge) ... 235
Byte-Order-Mark (BOM) ... 236
C0 Control Characters ... 226
C1 Control Characters ... 227
Camel case ... 284, 438
Caret Notation ... 225
Carriage Return ... 226, 227
Character ... *Siehe* Zeichen
Character Set ... *Siehe* Zeichensatz
charset ... *Siehe* Character Set
Child ... 40
Code Page Table ... *Siehe* Zeichensatztabelle
Code Unit ... 223
Codepage
  Codepage 437 ... 242
  Codepage 850 ... 242
Coley, Arthur ... 39
CSS
  @media ... 216
  absolute Größeneinheiten ... 209
  Box-Modell ... 172
  Cascading (Prioritätskaskade) ... 159
  Counter ... 219
  Deklaration ... 158
  Eltern, Kinder und Nachfahren ... 160
  em ... 210
  Farben (Colors) ... 211
  Fluss und Positionierung ... 180
  Html & CSS ... 210
  Inline Style ... 158, 210
  Kombinatoren ... 169
  Längeneinheit (Unit) ... 209
  Liste ... 217
  Pseudoelemente ... 203
  Pseudoklassen ... 188
  relative Größeneinheiten ... 209
  Selektoren ... 165
  Shorthand Properties ... 205
  Spezifitätsgewichtung von Selektoren ... 159
  Value Processing ... 164
  Vendor Prefix ... 208
  Zeichensatz ... 216
Datenbank
  Access by Index Row-ID ... 608
  Branch Node ... 608
  Clustered-Index ... 609
  Compound Index ... 612
  Datentabelle ... 606
  Heap ... 608
  Index Range Scan ... 625
  Leaf Node ... 608
  Regelbasierter Optimizer (RBO) ... 627
  Tree-Height ... 611
  Wurzelknoten ... 607
Datenbank
  Attributwert ... 605
  Body ... 605
  Database ... 602
  Datenbank ... 602
  Datenbankmodell ... 602
  HDM ... *Siehe* Hierarchische Datenbank-Modell
  Head ... 605
  IMS ... *Siehe* Information Management System
  Index Seek ... 605
  Instanzenebene ... 603
  Kante ... 602
  Knoten ... *Siehe* Node
  Leaves ... *Siehe* Leaf
  Node ... *Siehe* Knoten
  Parent Node ... 602
  Record ... *Siehe* Tupel
  Record Set ... 605
  Root ... *Siehe* Wurzel
  Satzebene ... 603
  Table Scan ... 605
  Tupel ... *Siehe* Datensatz
  Typebene ... 602
  Vernetztes Datenbank-Modell ... 604
  Wurzel ... *Siehe* Root
Datenbank
  Relation ... 606
Datenbank
  Attribut ... 606
Datenbank
  Attributwert ... 606
Datenbank
  Tupel ... 606
Datenbank
  Domain ... 606
Datenbank
  Clustered Index ... 606
Datenbank
  Doppelt verkettete Liste ... 607
Datenbank
  Blattknoten ... 607

Datenbank
Leaf Node 607
Datenbank
Blattknoten 607
Datenbank
B-Tree 607
Datenbank
B+Tree 607
Datenbank
Balanced Tree 607
Datenbank
Zweigknoten ..... *Siehe* Branch Nodes
Datenbank
Branch Node 607
Datenbank
Innerer Knoten 607
Datenbank
Blattknoten 607
Datenbank
Leaf Node 607
Datenbank
Heap 607
Datenbank
B-Tree 608
Datenbank
B+Tree 608
Datenbank
B+Tree 608
Datenbank
B-Tree 608
Datenbank
Baumknoten 608
Datenbank
Wurzelknoten 608
Datenbank
Root Node 608
Datenbank
Wurzelknoten 608
Datenbank
Zweigknoten 608
Datenbank
Zweigknoten 608
Datenbank
Innerer Knoten 608
Datenbank
Blattknoten 608
Datenbank
Blattknoten Scans 609
Datenbank
Index Unique Scan 609
Datenbank
Index Range Scan 609
Datenbank
Heap 609
Datenbank
Balanced Tree 609
Datenbank
Clustered Index 609
Datenbank
Index-Organized Table 609
Datenbank
IOT ..... *Siehe* Index-Organized Table
Datenbank
B-Tree 609
Datenbank
Balanced Tree 609
Datenbank
Heap 609
Datenbank
Blattknoten 610
Datenbank
Heap 610
Datenbank
Leaf Node 610
Datenbank
Wurzelknoten 610
Datenbank
Root Node 610
Datenbank
Zweigknoten 610
Datenbank
Branch Node 610
Datenbank
Baumgröße 611
Datenbank
Baumtiefe 611
Datenbank
Tree Height 611
Datenbank
Baum mit Compound-Primary-Index 612
Datenbank
zusammengesetzter Schlüssel ..... *Siehe* Compound Key
Datenbank
Compound Key 612
Datenbank
Composite Key 612
Datenbank
Baumteilung 613
Datenbank
Blattteilung 613
Datenbank
Leaf Split 613
Datenbank
Knotenteilung 614
Datenbank
Node Split 614
Datenbank
Page Split 615
Datenbank
Heap 615

Datenbank
Index Unqiue Scan 616
Datenbank
Index Range Scan 616
Datenbank
Table Access By Index Rowid 616
Datenbank
Table Access Full 616
Datenbank
Full-Table-Scan 616
Datenbank
Constraint 618
Datenbank
Primary Key ..... *Siehe* Primärschlüssel
Datenbank
Primärschlüssel 618
Datenbank
Compound Primary Key 621
Datenbank
Composite Key 621
Datenbank
Compound Index 621
Datenbank
Table Access Full 623
Datenbank
Optimzier 627
Datenbank
Ausführungsplaner ..... *Siehe* Optimizer
Datenbank
Query Planner ..... *Siehe* Ausführungsplaner
Datenbank
RBO ..... *Siehe* Rule-Based Optimizer
Datenbank
Kostenbasierter Optimizer (CBO) 627
Datenbank
Cost Based Optimizer 627
Datenbank
CBO ..... *Siehe* Cost Based Optimization
Datenbank
Index Range Scan 627
Datenbank
Table Access By Index 627
Datenbank
Funktionsbasierender Index 628
Datenbank
Benutzerdefinierte Funktion 630
Datenbank
Computed Columns 630
Datenbank: 602
Datentabelle 507
DBCS ..... *Siehe* Double-Byte Character Set
DEC Multilingual Character Set 241
Density ..... *Siehe* Dichte, *Siehe* Dichte
Depth-First Search 37
DFS ..... *Siehe* Depth-First Search
Dichte 21
Digraph 19
Directed Graph 26
Disgraph 23
Domain 507, 618
Double-Byte Character Set 232
Ecke 19
Eckpunkt 19
Edge ..... *Siehe* Kante
einfacher Graph 19
Eingangsgrad 18, 23
Endknoten 20
File Separator (FS) 226, 227
gerichteter Graph 19
gewichteter Graph 19
Glyphe 222
Grad 23
Ausgangsgrad 18, 23
Eingangsgrad 18, 23
Graph 18
Adjazenz 30
Adjazenzmatrix 25, 33
Ausgangsgrad 23
Azyklikscher Graph 28
bipartiten Graphen 35
Blatt 20
Bogen 19
Breadth-First Search 37
Brücke 22
Coley, Arthur 39
Depth-First Search 37
Dichte 21
Digraph 19
Disgraph 23
ebener Graph 18
Ecke 19
einfacher Graph 19
Eingangsgrad 23
Endknoten 20
gerichtete Kante 19
Gerichteter Graph 34
gewichteter Graph 19
Gewurzelter Baum 39
Grad 23
isomorph 24
Kantenfolge 28
Kantenzug 28
Knoten 19
Komponente 22
Kreis 28
Label 19
Löschen 18
Masche 28
Mehrfachkante 18

Mengenlehre ... 26
Multigraph ... 19
Nachgänger ... 26
Nullgraph ... 22
parallele Kanten ... 19
planar ... 24
plättbar ... 24
Predecessor ... 19
Quelle ... 23
Regulärer Graph ... 23
Schlinge ... 20, 23
schwach zusammenhänged ... 22
Senke ... 23
Spannbaum ... 21, 39
stark zusammenhänged ... 22
stark zusammenhängend ... 19
Successor ... 19
Suchbaum ... 40
Travesierung ... 37
Ungerichteter Graph ... 26
Ungerichteter Multigraph ... 34
Unvollständiger Graph ... 22, 29
Unzusammenhängender Graph ... 22
Vielfachkeit ... 19
Vielwegbäume ... 40
Vollständiger Graph ... 22, 29
Wald ... 38
Weg ... 28
Zyklischer Graph ... 28
Hierarchische Datenbank-Modell (HDM) ... 602
html
Content-Language ... 125
Doctype ... 125
Eingebetteter Inhalt ... 134
Event Handler ... 156
Favicon ... 154
Formular ... 152
Geänderter/Gelöschter Text ... 134
Generische Container-Tags ... 128
Html5 Tags ... 150
Interaktive Elemente ... 138
Listen, Textgruppen ... 141, 142
Namespace ... 125
Obsolete Attribute ... 145, 149
Obsolete Attribute als Tag-Tabelle ... 147
Obsolete HTML-Tags ... 144
Semantische Tags ... 127
Sprache (Language) ... 125
Tabelle ... 151
Textformatierende semantische Tags ... 129
Umdefinierte HTML-Tags ... 145
Zeichensatz ... 126
Incidence ... *Siehe* Inzidenz
Indegree ... 18, *Siehe* Eingangsgrad
Index
Clustered Index ... 507
Composite Index ... 612
Compound Index ... 612
Information Management System ... 602
Inzidenzmatrix ... Siehe Kanten-Knotenmatrix
ISO-8859 ... 230
ISO-8859-1 (Latin1) ... 231, 240
ISO-8859-15 (Latin9) ... 231, 240
Unterschied zwischen ISO-8859-1 und ISO-8859-15 ... 230
Isolierter Knoten ... 20
Isomorphie
Isomorphe Graphen ... 24
Isomorphie ... 24
JavaScript ... 310
"use strict" ... 291
Array ... 342
Array Methoden ... 346
Arrow Notation ... 400
Assoziativität ... 378
Bitwise Operators ... 372
Block Statements (Compound Statements) ... 285
boolean ... 304
Boolean Objekt ... 304
Boolean() Funktion ... 305
Call-Site ... 323, 421
Call-Stack ... 323, 421
Complex Data Types ... 298
Conditional Statements ... 382, 483
const ... 293
Control Flow (Flusskontrolle) ... 382, 483
Datentypen (Data Types) ... 298
Date-Objekt ... 335
Deklaration ... 287
Destrukturierende Zuweisung ... 377
Dot Notation ... 286, 316
Expression ... 285, 524
falsy ... 304, 370
Function Hoisting ... 399
Hello World ... 297
Identifiers ... 284
if/else ... 382, 483
IIFE ... 401
Infinity ... 301
Initialisierung ... 296
Inline JavaScript ... 432
isNaN() ... 301
Keywords ... 429

Konstruktor ... 328
Label ... 389, 392
let ... 293, 425
line continuation character ... 286
Literal ... 282, 435
Logische Operatoren ... 370
Loops (Schleifen) ... 386
NaN ... 301
null ... 305
number ... 299
Operatoren ... 363
Präzedenz ... 378, 478, 549
Präzedenz (Precedence) ... 378, 478, 549
Primitive Data Types ... 298
Relationaloperatoren ... 368
Scope ... 422
Short-Circuit Evaluation ... 372
Square Bracket Notation ... 286
Statement ... 285
string ... 306
String Funktionen ... 309
switch ... 384
symbol ... 305
Ternäre Operator ... 369
this ... 321
Token ... 285
truthy ... 304, 370
undefined ... 305
Unobtrusive JavaScript ... 430
var ... 287
Variable Masking ... 427
Vergleichsoperatoren ... 363, 366
with ... 331
Kant
Kantenfolge ... 28
Kante ... 19
Brückenkante ... 22
gerichtete Kante ... 19
Kantendichte ... 21
Kanten-Knotenmatrix ... 31
Kantenzug ... 18, 28
parallele Kanten ... 19
Kind ... 40
Knoten ... 40
Blattknoten ... 608
Knoten-Knotenmatrix ... *Siehe* Adjazenzmatrix
Kodierung ... *Siehe* Zeichensatztabelle
Komponente ... 22
Kreis ... 28
Label ... 19, 21
Latin
Latin 9 ... *Siehe* ISO-8859-15
Latin-1 ... *Siehe* ISO-8859-1
Line Feed ... 226
Linux
atime ... 58
awk ... 66
Command Line Interface (CLI) ... 43
Command Line Interpreter (CLI) ... 43
Command Line User Environment (CLOE) ... 43
ctime ... 58
file ... 51
File Classification ... 63
I/O Redirection Operator ... 62
ls 63
Meta Key ... 52
mtime ... 58
noatime ... 58
relatime ... 58
Shell ... 43
Standard Input ... 62
tape archive ... *Siehe* tar
tar ... 59
touch ... 56
Little-Endian ... 235
Majuskel ... 224
markdown
Bild ... 120
Block (Monospace) ... 119
Blockquote ... 116
Escape Character (Fluchtsymbol) ... 123
fett, kursiv, durchgestrichen ... 114
Horizontale Linie ... 116
Liste ... 117
Tabelle ... 122
Überschrift (Headline) ... 115
Uniform Resource Locator (URL) ... 120
Zeilenumbruch ... 113
Masche ... 28
Matrix
Kanten-Knotenmatrix ... 31
Mehrfachkanten ... 18
Middle Endian ... 235
Mixed Endian ... 235
Mojibake ... 223
Multibyte Encoding ... 233, 234
Multigraph ... 19
New Line ... 226
Nikolaus Haus ... 18
No Break Space (NBSP) ... 225, 230
Node ... 40
NUL ... 229
Nullgraph ... 22
Ordnung ... 18
Outdegree ... 18, *Siehe* Ausgangsgrad
PHP

Block Statements (Compound Statements) 445
boolean 454
Composite Data Types 456
Cookie 504, 505
Datentypen (Data Types) 448
die 494
do while 489
exit 494
Expression 444
falsy 454
Floating Point Number (Fließkommazahl) 451
Floating Point Number Comparison 453
for 490
goto 493
Identifier 438
if 483
Infinity 451
Integer (Ganzzahl) 449
Klasse 495
Konstante 439, 481
Label 493
Literal 435
Loop 487
Magic Method 499
NaN 451
Objekt 502
PHP Tag 447
Primitive Data Types 448
Remark 446
Schleife 487
Scope 480
Skalar 448
Statement 444
switch 485
Token 444
truthy 454
Type Casting 462
Variable 437
while 487
Whitespace) 445
planar 24
plättbar 24
postgreSQL
BIGINT 530
BIGSERIAL 527
Boolean Literal 517
Bulk Cross Join 562
Case Sensitive 515
Datentpyen 525
DECIMAL 531
DOUBLE PRECISION 535
Exponent-Literal 521
Fixed-Point exact Number 531
FLOAT 536
Floating-Point Numbers (IEEE 754 533
Hexadezimal Literal 522, 523
Inner Join 565
INT 529
Integer (Ganzzahl) 527
Kartesisches Produkt 562
Komma (List Separator) 514
Kreuzprodukt 562
Literal 516
MEDIUMINT 527
MONEY 538
Notationsbeschreibung 509
NULL Literal 521
NUMERIC 531
Numerics (Zahlen) 526
Outer Join 567
psql 508
REAL 534
Relation 510
Remark (Bemerkung) 514
SELECT Blöcke 539
SERIAL 527, 537
SMALLINT 528
SMALLSERIAL 527
SQL Syntax 511
TINYINT 527
Verben 512
Whitespace 514
Zahlen-Literal 522
Predecessor 19
Quelle 23
Regulärer Graph 23
Relation 507
Scalable Vektor Graphics ..... *Siehe* SVG
Painters-Modell 244
Schleife ..... *Siehe* Schlinge (Multigraph)
Schlinge 23
Self-loop ..... *Siehe* Schlinge (Multigraph)
Senke 23
Snake case 284, 438
Soft Hyphen (SHY) 225, 230
Space (SP) 224
Spanning Tree ..... Siehe Spannbaum
Successor 19
SVG 244
baseProfile 244
Cascading Style Sheets (CSS) 268
circle 254
defs 270

doctype....244
elipse....255
elliptischer Kurvenbogen....262
fill250
g 270
Gruppierungselemente....269
Koordinatensystem....249
kubische Bézierkurve....260
line....255
marker....271
Maßeinheiten....249
Painter's-Modell....244
path....257
polygon....256
polyline....256
quadratische Bézierkurve....259
rectangles....253
Shape Attribute....250
Shape Elemente....250
stroke....252
symbol....269
text....263
Transformation....274
use....273
version....244
viewPort....245
Table
Full-Table-Scan....616
Tiefensuche....*Siehe* Depth-First-Search
Traversal....*Siehe* Travesierung
Travesierung....37
Tupel....507
Undirected Graph....26, 27
Ungerichteter Graph....26, 27
UTF-8....234
UTF-8 Signatur....236
Valenz....*Siehe* Grad
Versalie....224
Vertex....18, 19
Vertices....*Siehe* Vertex
Vielfachkeit....19
Wald....38
Weg
Weg....28
Whitespace....237
Win-1252....241
Zeichen
ASCII (7bit)....224
ASCII C0 Control Characters....226
ASCII C1 Control Characters....227
ASCII Character Codes....228
Bi-Endian....235
Big-Endian....235
Byte-Order....235
Byte-Order-Mark (BOM)....236
Caret Notation....225
Code Point....223, 235
Code Unit....223
Codepage 437....242
Codepage 850....242
Control Characters (Steuerzeichen)....224
DEC Multilingual Character Set....241
Delete (DEL)....224
Double-Byte Character Set....232
Escape Sequence (Fluchsequenz)....225
Extended ASCII (8bit)....230
Fixed-Width Encoding....232
Glyphe....222
ISO-8859-1 (Latin1)....240
ISO-8859-15 (Latin9)....240
Kodierung (Encoding)....223
Little-Endian....235
Majuskel....224
Middle Endian....235
Mixed Endian....235
Mojibake....223
Multibyte Encoding....233, 234
No Break Space (NBSP)....225, 230
NUL....229
Soft Hyphen (SHY)....225, 230
Space (SP)....224
Unterschiede zwischen ISO-8859-1 (Latin1) und ISO-8859-15 (Latin9)....230
UTF-8 Encoding....234
UTF-8 Signatur....236
Variable-width Encoding....232, 233, 234
Versalie....224
Whitespace....237
Win-1252....241
Zeichen....222
Zeichensatz....222
Zeichensatztabelle....222
Zero width no-break space (U+FEFF)....236
Zero width Space (U+200B)....236
Zyklus....*Siehe* Kreis