

! /usr/bin/tclsh

Ausgabe in der Kommandozeile:

```
puts "Hello World";  
puts {Hello World};
```

Ein Semikolon ist nur erforderlich wenn mehr als ein Befehl in einer Zeile steht oder ein Kommentar folgt. Ansonsten ist es optional.

Variablen:

```
set value 2; # Ein Kommentar  
puts "the value is $value";
```

Zugriff auf Variablenwerte mit dem \$-Symbol

- Diese Ausgabe funktioniert nur mit Anführungszeichen "", nicht mit geschweiften Klammern {}

```
set str "Hello World";  
puts "$str"; }  
puts $str;
```

```
set a 5;
```

Set b \$a; # Variablen mit Variablenwerten initialisieren
alternativ:

```
set b [set a 5];
```

set val 1.5;

puts "the value is \$value";

↳ the value is 1.5

puts \$the value is \$value3;

↳ the value is \$value

Arithmetische Operationen

set a "5";

set b 3;

set c [expr "\$a + \$b"];

puts "\$a + \$b = \$c";

Addition: expr \$a + \$b

Subtraktion: expr \$a - \$b

Multiplikation: expr \$a * \$b

Division: expr "12 / 2" → 1

expr 12 / 2 + 3 → 4

expr 12.0 / 2 + 3.0 → 4, 21428

Logische Operationen:

expr \$a && \$b a und b

expr \$a || \$b a oder b

expr ! \$a nicht a

expr \$a & \$b bit - und

expr \$a | \$b bit - oder

expr \$a ^ \$b bit - xor

expr \$a << n n shifts nach links

expr \$a >> n n shifts nach rechts

Bedingte Anweisungen

```
set x 2;  
if { $x == 3 } {  
    puts "x is 3";  
elseif { $x == 2 } {  
    puts "x is 2";  
else {  
    puts "x is not 2 or 3";  
}
```

Die Positionen der
öffnenden geschweiften Klammern
sind wichtig

Vergleichsoperatoren

`==` ist gleich

`!=` ist ungleich

`>` größer als

`<` kleiner als

`>=` größer als oder gleich

`<=` kleiner als oder gleich

Schleifen

```
set x 0;  
while { $x < 6 } {  
    puts "x is $x";  
}
```

x nimmt alle Werte von 0 bis 5 an.

```
for { puts "Stand"; set i 0 } { $i < 2 } {  
    incr i; puts "$i nach increment: $i";  
}
```

Schleifenkörper

```
puts "$i in Schleife: $i";
```

3

→ stand

i in Schleife: 0

i nach increment: 1

i in Schleife: 1

i nach increment: 2

Arrays

```
set ArrayName [Index] value ;  
  
Set values(0) "Hello";  
set values(1) "World";  
puts "$values(0), $values(1)";  
↳ Hello, World
```

Verwendung von Schleifen:

```
set arr(0) "Hi"  
set arr(1) "is"  
set arr(2) "fine";  
  
for {set index 0} {$index < [array size arr]}  
{  
    incr index  
    puts $arr($index);  
}  
  
↳ Hi  
↳ is  
↳ fine
```

Associative Arrays / Dictionaries:

- Alle Arrays sind eigentlich dictionaries, die Schlüssel (keys) können geordnet sein, müssen aber nicht.

```
set $student1 (Name) "Tim";  
set $student1 (id) 2032716;
```

```
puts $student1 (Name);  
puts $student1 (id);
```

foreach - Schleifen:

verwende \$student1 array aus. ↑ Bsp.

```
foreach index [array names $student1] {  
    puts $student1 ($index);  
}
```

⇒ \$index nimmt alle keys von \$student1 an

Einen Eintrag im Array finden:

```
set arr(0) . a .;
```

```
set arr(1) . b .;
```

```
set arr(2) . c .;
```

```
set arr(3) . d .;
```

```
set element-to-find . c .;
```

```
foreach number [array names arr] {
```

```
    if { $arr($number) == $element-to-find } {
```

```
        puts "Found: $arr($number)";
```

3.

3.

=> finde einen eintrag im array indem
alle keys verwendet werden um
die dahinterliegenden einträge zu
vergleichen

student - age

set student(dylan) 25;

set student(tabea) 28;

set student(alex) 22;

set student(mardin) 25;

set student(naomi) 21;

set name-to-find alex;

foreach name [array names student] {

if {\$name == \$name-to-find} {

puts "Name: \$name";

puts "Age: \${student[\$name]}";

}

3

Exec : Shell-Befehle ausführen

exec ls ;

⇒ shell-Command 'ls' ausführen

exec gibt die Ausgabe des aufgerufenen Programms zurück

sed result [exec ls -l];

puts \$result;

→ gibt die Ausgabe 2 mal aus: beim exec Aufruf und beim puts Aufruf

Escape Sequenzen

puts " \n"; # zusätzlicher Zeilenumbruch

tab "\t"

⇒ Sonderzeichen müssen immer mit einem Backslash maskiert werden

String - Operationen

- `String compare string1 string2`
 - ↳ Returns 0 falls gleich
 - ↳ Returns -1 falls Reihenfolge $\text{string1} < \text{string2}$
(Bsp.: "abc" < "abcd")
 - ↳ Returns 1 falls Reihenfolge $\text{string2} < \text{string1}$
(Bsp.: "abcd" > "abc")
- `String index String index`
 - ↳ Gibt das Zeichen an Stelle index zurück
 - puts.[string index "Hallo" 1];
↳ a
- `String length String`
 - ↳ Gibt die Länge der Zeichenkette zurück

- `String range string index1 index2`

↳ Gibt die Zeichen von inkl. index 1 bis inkl. index 2 zurück

puds [String range "Müllermeier" 4 8]

↳ eimer

- `String tolower string`

↳ gibt den String in Kleinbuchstaben zurück

- `String toupper string`

↳ gibt den String in Großbuchstaben zurück

- `String trimright string ? dimcharact?r`

↳ entfernt von rechts Zeichen die in dimcharact?r gelistet sind bis zum ersten nicht gelisteten Zeichen, default : Leerzeichen

String trimright "Test-abc**bab**c" "abc"

↳ " - Test."

- `String trimleft string ? dimcharact?r`

↳ identisch zu trimright

- `String trim`: `String` 3 frisch characters?
kombiniert `String left` und `String right`

`String trim "HelloWorld"`
↳ "String" ↑ Leerzeichen

`String trim :: Error ::`
↳ "Error"

- `String match pattern String`
↳ prüft, ob das pattern auf den string
passt

► Sonder-Ausdrücke im Pattern:

* eine beliebige Zeichenkette, kann auch
leer sein

? ein beliebiges Zeichen

[char] eines der Zeichen von char

\x Das Zeichen x. Diese Funktion wird
verwendet um * [C] in Pattern
zu beschreiben.

Beispiel:

```
set $1 "email@example.com"
set $2 "*@*.com"
if [ ${string} match ${$2} ${$1} ] ; then
    puts "It's a match"
else
    puts "no match"
fi
```

```
set $3 "*@*.de"
if [ ${string} match ${$3} ${$1} ] ; then
    puts "It's a match"
else
    puts "no match"
fi
```

→ It's a match
→ no match

- append original-str append-str
 - ↳ konkatiniert append-str an das Ende von original-str, ergibt keinen neuen string

set \$1 "I love"

append \$1 " tcl "

puts \$\$1

↳ "I love tcl"

Listen

Listen sind geordnete Sammlungen an Zahlen, Strings oder anderen Listen.

Initialisierung:

set listname { item1 item2 item3 ... }

set listname [item1 item2 item3 ...]

set names [split "Arthur~Martin~Thomas" " "]

spaltet String auf Separator

für names

→ Arthur Martin Thomas

set jobs [list "Maurer" "Schreiner" "Tischler"]
erstellt eine Liste

Listen-Länge und Listen-Zugriff

set nums [list 8 7 32 9]

pubs "Länge der Liste: [length \$nums]"

↳ 4

Listen-Länge

pubs "Eintrag an 3. Stelle: [index \$nums 2]"

↳ 32

Zugriff

auf
Listenelement

Index
beginnt
bei 0

! Strings mit Leerzeichen sind automatisch

! auch Listen, solange keine andere Trennung vorliegt !

set x "a b c"

pubs [index \$x 1]

↳ b

Iterieren über listeneinträge

foreach \$x {

 puts \$x

}

⇒ j nimmt im Schleifenkörper die Werte aller Einträge der Liste x an.

Listen konkatinieren:

set c [concat \$a \$b]

Listen oder Variablen / Strings

Einträge hinzufügen / entfernen / ersetzen

set employees [concat "John" "Ryan" "Tina"]

set more-people [insert \$employees 2 "Jacob"]

↑
Stelle an der die
neuen Inhalte kommen

Vorher: 0|1|2|3|..

↑
Neuer Index im Parameter von insert

Nachher: 0|1|M|2|3|..

puts \$more-people

↳ John Ryan Jacob Tina

set others [lreplace \$more-people 2 3 "Marc"]

puts \$others

Ersetze vom Index

↳ John Ryan Marc

2 bis Index 3

lreplace ersetzt die Liste von einem Start - bis zu einem Zielindex mit den da-hinter aufgelisteten Inhalten

lreplace "Hello World" 1 1 "chocolate" "Cake"

↳ Hello chocolate Cake

T
≠
1

lreplace "Hello World" 1 1 "chocolate" Cake

↳ Hello {chocolate Cake }

~

~



set bakery "bread cheesecake croissant"

lappend bakery "Bretzel"

puts \$bakery

↳ bread cheesecake croissant Bretzel

lappend hängt ein Element an das Ende der Liste an. Dabei wird keine neue Liste erzeugt sondern die bestehende Liste verändert.

Listen sortieren in $O(n \log n)$ (Merge-Sort)

set numbers "3 9 1 7 5"

puts [lsort \$numbers]; # alphabetisch aufsteigend

↳ 1 3 5 7 9

optionales flagge

puts [lsort -decreasing \$numbers]

alphabetisch absteigend

↳ 9 7 5 3 1

```
set neg-numbers [list -1.25 -9.28 -6.3]
```

```
puts [lsort -real -decreasing $neg-numbers]
```

↳ -1.25 -6.3 [↑] -9.28

-real wird gesetzt um Vergleiche

zwischen Fließkommazahlen durchzuführen

Teil-Liste extrahieren

```
set prices "12.77 4.23 3.14 9.72"
```

```
puts [lrange $prices 1 2]
```

↳ 4.23 3.14

lrange gibt alle Listen-Einträge von inklusive
dem start- bis inklusive dem Ziel-index zurück

Der Befehl kennt auch das Schlüsselwort end
was representativ für den Index des letzten
Listeneintrags steht. Außerdem kann dazu noch
ein displacement angegeben werden.

```
puts [lrange $prices end end] # letzter Eintrag
```

```
puts [lrange $prices end-2 end] # Letzte 3 Einträge
```

Dateien

Die Zugriffsberechtigungen werden vom Betriebssystem beim Öffnen der Datei überprüft.

Zum Öffnen der Datei kann zwischen verschiedenen Zuständen gewählt werden:

- v öffnet die Datei lesend, Datei muss bereits vorhanden sein
- r+ öffnet die Datei lesend und schreibend, Datei muss bereits vorhanden sein.
- w öffnet die Datei schreibend und erstellt sie, falls sie noch nicht vorhanden ist
- w+ öffnet die Datei lesend und schreibend und erstellt sie, falls sie noch nicht vorhanden ist

- a öffne die Datei schreibend, Datei muss bereits existieren
- a+ öffne die Datei lesend und schreibend, erstelle die Datei falls sie noch nicht existiert.

Bei w und w+ wird der Cursor an den Anfang der Datei gesetzt, bei a und a+ wird der Cursor an das Ende der Datei gesetzt.

Wenn die zu öffnende Datei nicht vorhanden ist, wird der Befehlsaufruf bei der Verwendung von r, r+ oder a einen Fehler.

Beim Öffnen der Datei wird ein Datei - handle zurück gegeben welcher für die lese- und Schreiboperationen benutzt wird.

Wenn die Datei nicht mehr benötigt wird, sollte sie wieder geschlossen werden.

Bsp.: Datei am Stück einlesen

set \$p [open "input.txt" r]

Datei - Handler Aufruf zum Öffnen der Datei

set file-data [read \$p]

puts file-data

=> liest die ganze Datei auf einmal aus
und gibt den Inhalt in der Konsole aus.

close \$p

=> Datei muss nach der Verwendung
wieder geschlossen werden

Bsp.: Datei zeilenweise lesen

```
set fp [open "input.txt" r]
while {[gets $fp data] >= 0} {
    puts $data
```

3

```
close $fp
```

Die Schleife wird solange ausgeführt bis
get a einen Wert < 0 zurück gibt weil das
Ende der Datei erreicht wurde.

Bsp.: in Datei schreiben

```
set fp [open "file.txt" w+]
puts $fp "test"
close $fp
```

=> puts kann neben standard auch in
Dateien schreiben.

Prozeduren / Funktionen

(Name)	(Argumente)	(Anweisungen)
proc name-of-function {	}	{
{		

Beispiel:

```
proc sum {a b} {  
    return [expr $a + $b]  
}
```

puts [sum 10 30] ← Aufruf

Argumente können auch default-Werte haben.

```
proc buy-car {model {color "black"}} {  
    puts "You bought a $model in $color."  
}
```

Diese Variable kann beim Aufruf optional gesetzt werden.

Die Anzahl an Argumenten kann auch beliebig lang sein wenn Listen übergeben werden.

```
set birthday-kids [list "John Ryan Tim"]  
proc best-wishes {names} {  
    set texts {}  
    foreach name $names {  
        set greeting [join [list "Happy Birthday" $name] "\n"]  
        lappend texts $greeting  
    }  
    return texts  
}  
puts [best-wishes $birthday-kids]
```