# A Way to (S)hell – Einstieg in die Programmierung mit Bash

**Chemnitzer Linux-Tage 2025**

B1 Systems GmbH

22. März 2025

If you have any questions, comments or want to report errors in the training material please post them to doku@b1-systems.de.

Please note that all soft and hardware names, trademarks and product names of the respective firms used in this manual remain property of their holders even if not marked accordingly.

# Inhaltsverzeichnis



# 1. Shell lernen

Die Shell `bash` kann am besten durch Benutzung gelernt werden, indem so viele Dinge wie möglich im Terminal gemacht werden.
Folgende Alternativen zu GUI-Programmen lassen sich in der Shell nutzen:

**E-Mail** `neomutt` / `mutt`

**Browser** `lynx`

**Dateimanager** `cd`, `ls` / `ranger`

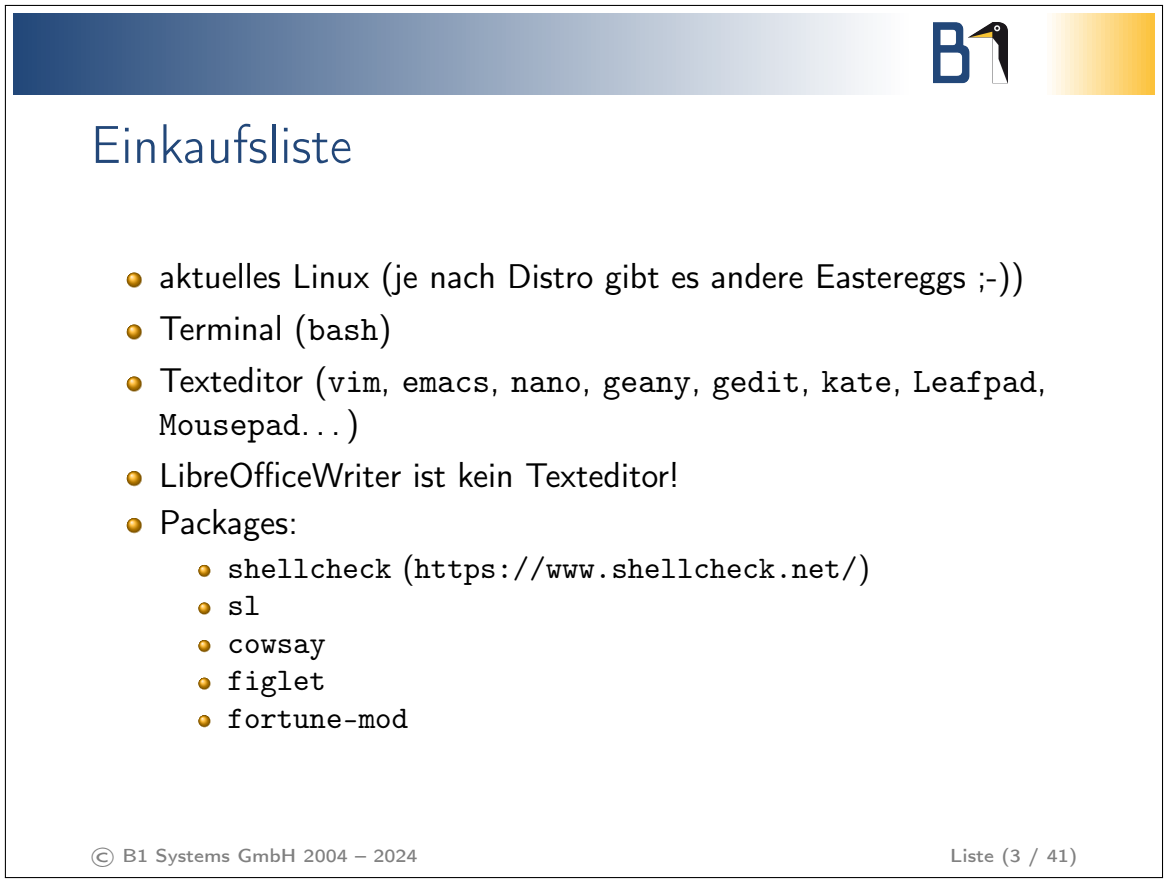
**Bildbetrachter** `feh`, `imv`

**Aufgabenliste** `taskwarrior`

**Kalender** `calcurse`, `cal`

Diese Liste ist als Anregung gedacht und hat keinen Anspruch auf Vollständigkeit, bitte um weitere Tools ergänzen!

# 2. Liste

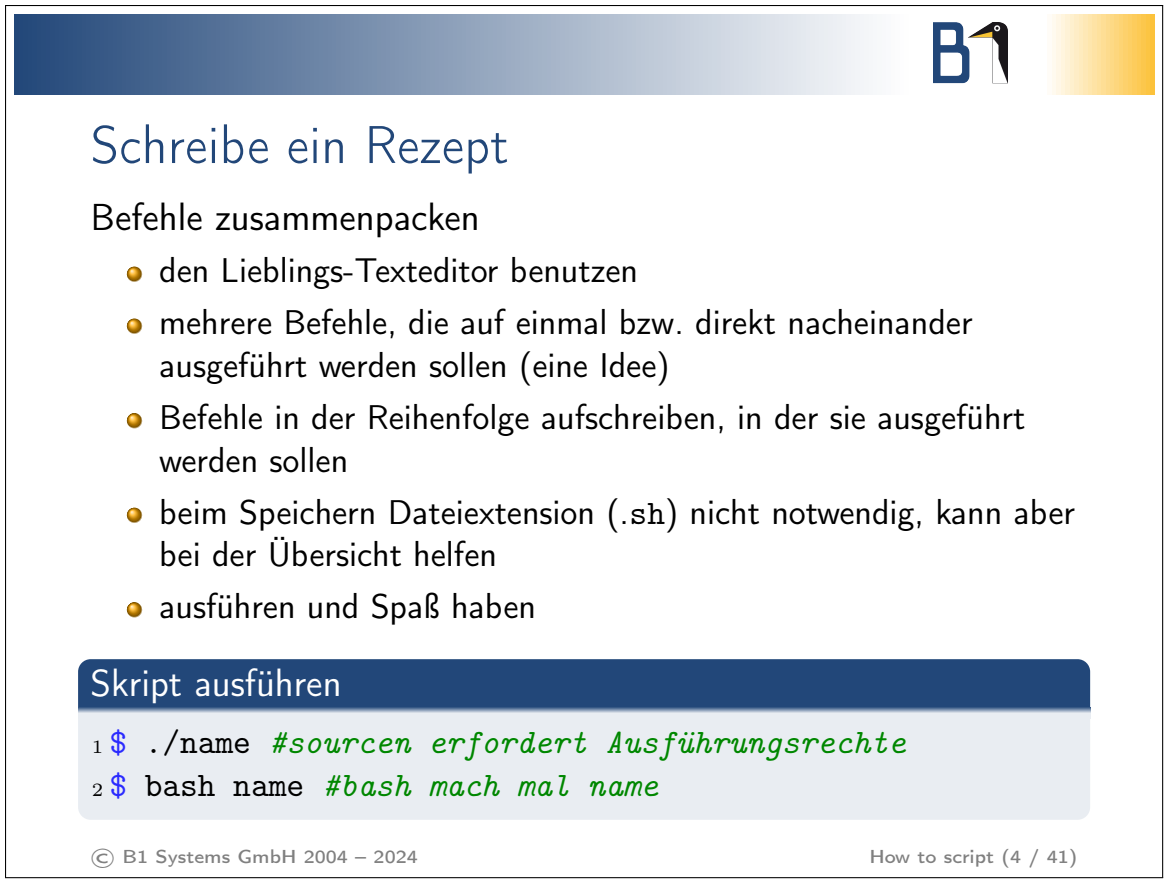


# 3. Was braucht es zum Programmieren?

Auf jeden Fall Kreativität . . . , wichtig ist auch, nicht gleich mit dem ersten Skript die Welt verändern zu wollen. Je kleiner das erste Skript ist, desto eher gibt es auch ein Erfolgserlebnis.

## 3.1. How to script

Skripte schreiben kann mit dem Schreiben von Rezepten verglichen werden.

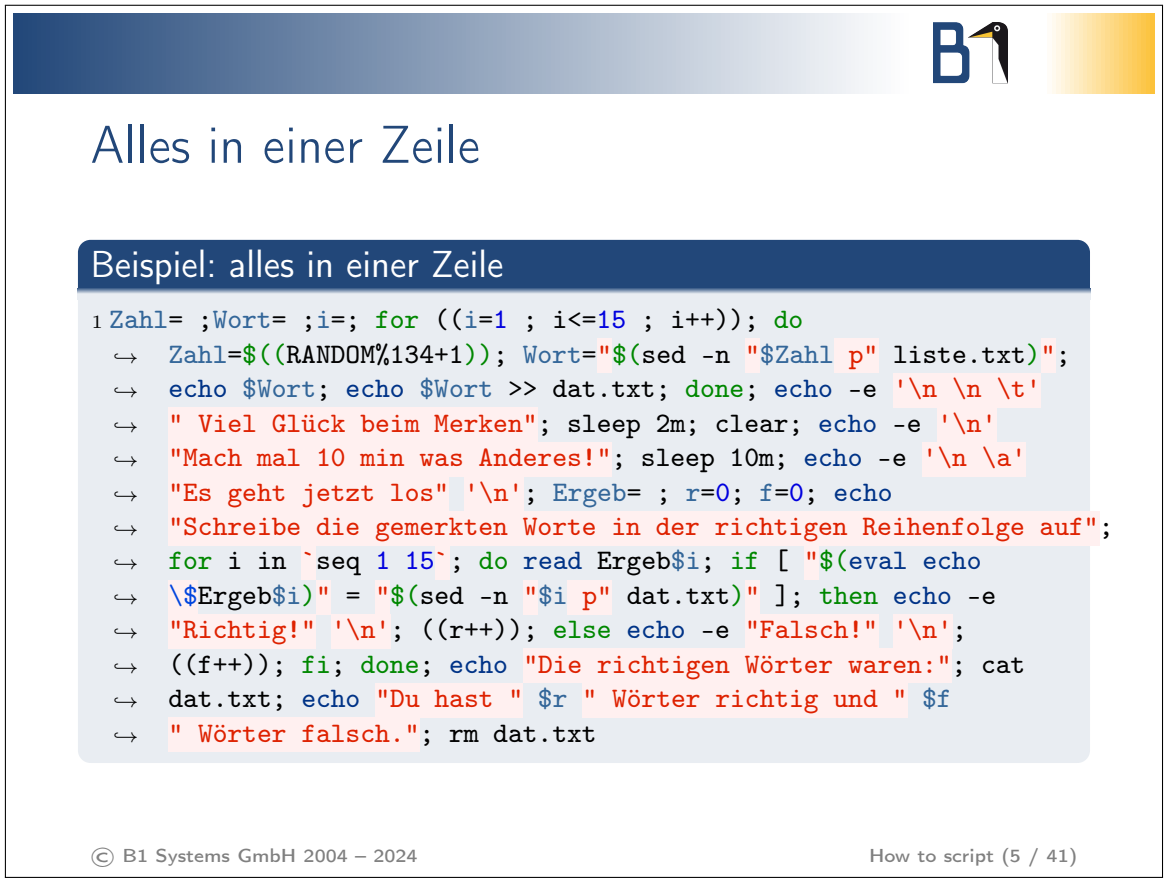



Egal um welche Programmiersprache es sich handelt, es gibt ein paar Basics, die gute Skripte ausmachen.

## 3.2. Zeilen im Skript

### 3.2.1. Zeilenumbrüche

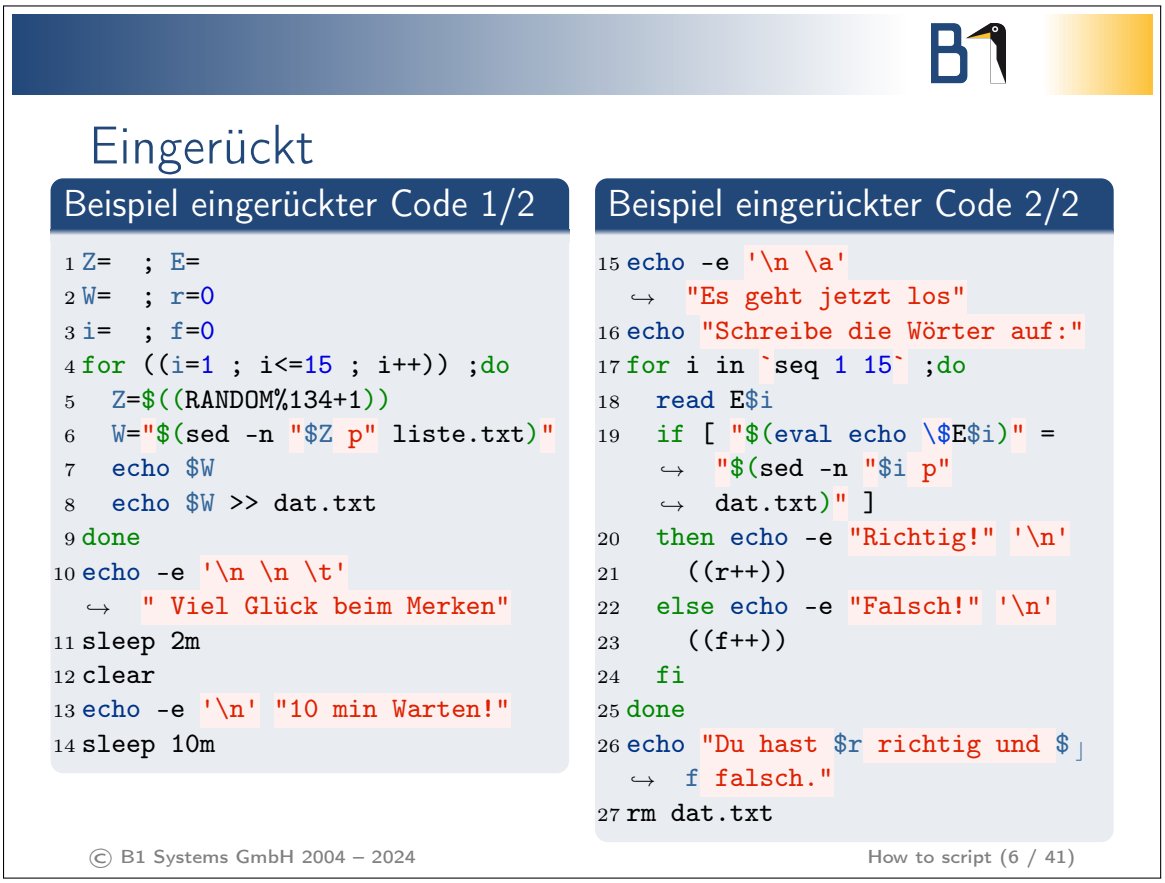
Bis auf sehr wenige Programmiersprachen sind Zeilenumbrüche nicht unbedingt notwendig. Aber je mehr in einer Zeile als Anweisung geschrieben steht, desto schwerer fällt es in der Regel, das Skript/Programm zu lesen und nachzuvollziehen, was es tut. Daher mache Zeilenumbrüche immer spätestens dann, wenn das Zeilenende erreicht ist, und du ein Symbol für das Zeilenende (oft ;) eingeben musst.



Alles in einer Zeile

Beispiel: alles in einer Zeile

```
1 Zahl= ;Wort= ;i=; for ((i=1 ; i<=15 ; i++)); do
  ↪ Zahl=$((RANDOM%134+1)); Wort="$(sed -n "$Zahl p" liste.txt)";
  ↪ echo $Wort; echo $Wort >> dat.txt; done; echo -e '\n \n \t'
  ↪ " Viel Glück beim Merken"; sleep 2m; clear; echo -e '\n'
  ↪ "Mach mal 10 min was Anderes!"; sleep 10m; echo -e '\n \a'
  ↪ "Es geht jetzt los" '\n'; Ergeb= ; r=0; f=0; echo
  ↪ "Schreibe die gemerkten Worte in der richtigen Reihenfolge auf";
  ↪ for i in `seq 1 15`; do read Ergeb$i; if [ "$(eval echo
  ↪ \$Ergeb$i)" = "$(sed -n "$i p" dat.txt)" ]; then echo -e
  ↪ "Richtig!" '\n'; ((r++)); else echo -e "Falsch!" '\n';
  ↪ ((f++)); fi; done; echo "Die richtigen Wörter waren:"; cat
  ↪ dat.txt; echo "Du hast " $r " Wörter richtig und " $f
  ↪ " Wörter falsch."; rm dat.txt
```



Eingerückt

Beispiel eingerückter Code 1/2

```
1 Z=  ; E=
2 W=  ; r=0
3 i=  ; f=0
4 for ((i=1 ; i<=15 ; i++)) ;do
5   Z=$((RANDOM%134+1))
6   W="$(sed -n "$Z p" liste.txt)"
7   echo $W
8   echo $W >> dat.txt
9 done
10 echo -e '\n \n \t'
   ↪  " Viel Glück beim Merken"
11 sleep 2m
12 clear
13 echo -e '\n' "10 min Warten!"
14 sleep 10m
```

Beispiel eingerückter Code 2/2

```
15 echo -e '\n \a'
   ↪  "Es geht jetzt los"
16 echo "Schreibe die Wörter auf:"
17 for i in `seq 1 15` ;do
18   read E$i
19   if [ "$(eval echo \$E$i)" =
     ↪  "$(sed -n "$i p"
     ↪  dat.txt)" ]
20   then echo -e "Richtig!" '\n'
21     ((r++))
22   else echo -e "Falsch!" '\n'
23     ((f++))
24   fi
25 done
26 echo "Du hast $r richtig und $
   ↪  f falsch."
27 rm dat.txt
```




### 3.2.2. Variablennamen/Funktionsnamen

Es ist für dein zukünftiges Ich wichtig, Variablennamen nach Möglichkeit so auszuwählen, dass sofort klar wird, was sie enthalten oder wofür sie verwendet werden. Man sagt auch, es sollten „sprechende“ Namen verwendet werden. Der Name **`Meine_Variable1`** ist in keinster Weise hilfreich, auch die Benennung nach Sternen, Städtenamen, Flüssen oder Tieren mag von Kreativität zeugen, hilft aber nicht, das Skript auch zu einem späteren Zeitpunkt noch zu verstehen.

## 3.3. Sprechende Namen im Skript

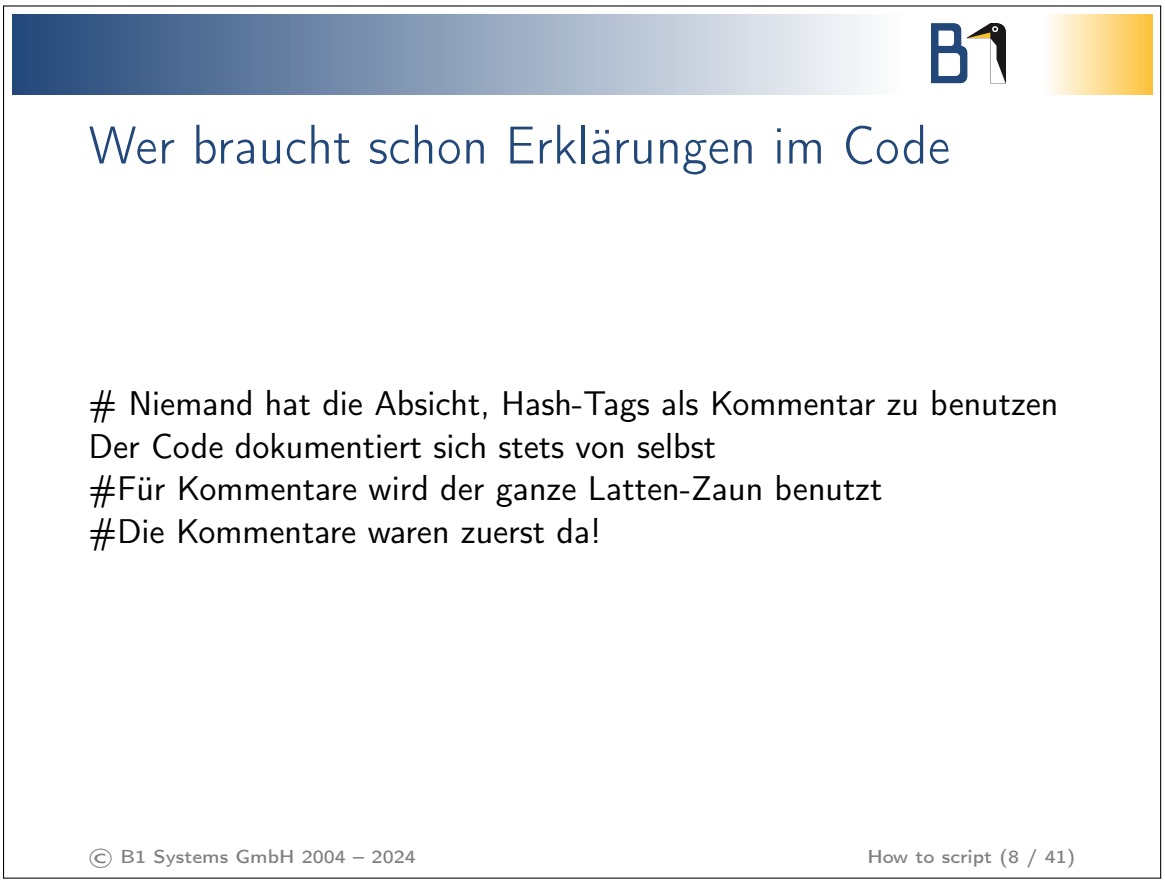
# Sprechende Namen im Skript

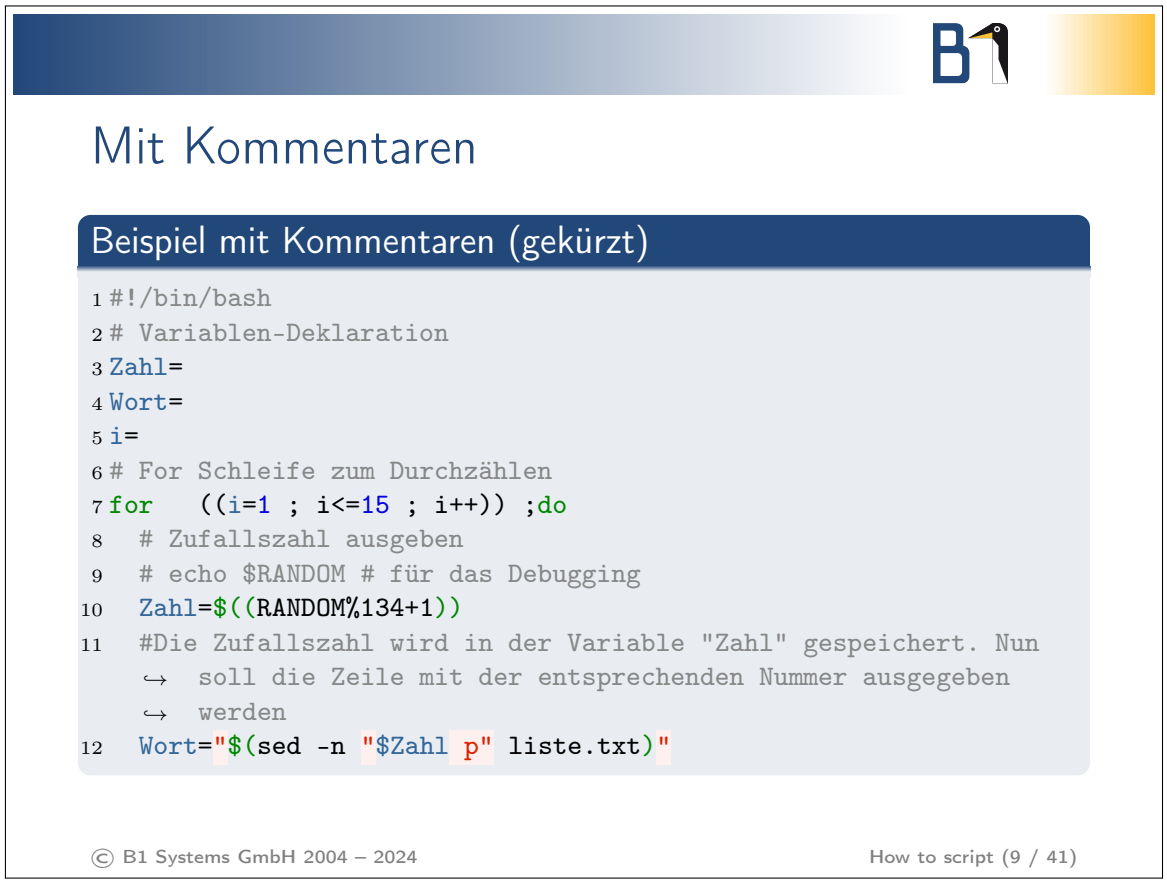
### Beispiel sprechende Namen im Code (gekürzt)

```
1 #!/bin/bash
2 Zahl=
3 Wort=
4 i=
5 for ((i=1 ; i<=15 ; i++))
6 do
7   Zahl=$((RANDOM%134+1))
8   Wort="$(sed -n "$Zahl p" liste.txt)"
9   echo $Wort
10  echo $Wort >> dat.txt
11 done
12 echo -e '\n \n \t' " Viel Glück beim Merken"
13 sleep 2m
14 clear
15 echo -e '\n' "Mach mal 10 min was Anderes!"
```

## 3.4. Kommentare

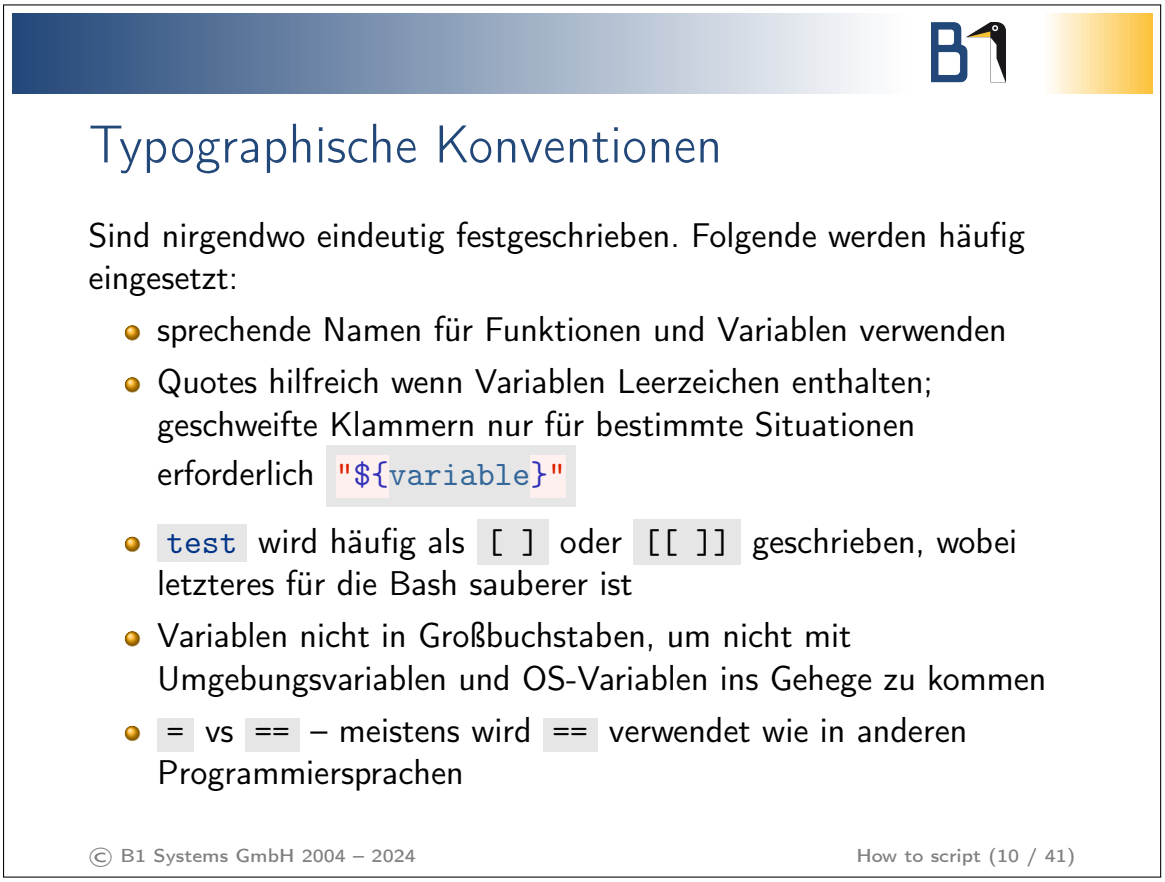
Kommentare können den Code erklären und sind vor allem dann zu verwenden, wenn man für die Lösung nachschlagen müsste. Gerade in der Anfangszeit kann nicht zu viel kommentiert werden. Kommentiere daher auch durchaus deine Gedanken, warum du etwas so oder so löst, das erleichtert deinem zukünftigen Ich, den von dir geschriebenen Code nachzuvollziehen.





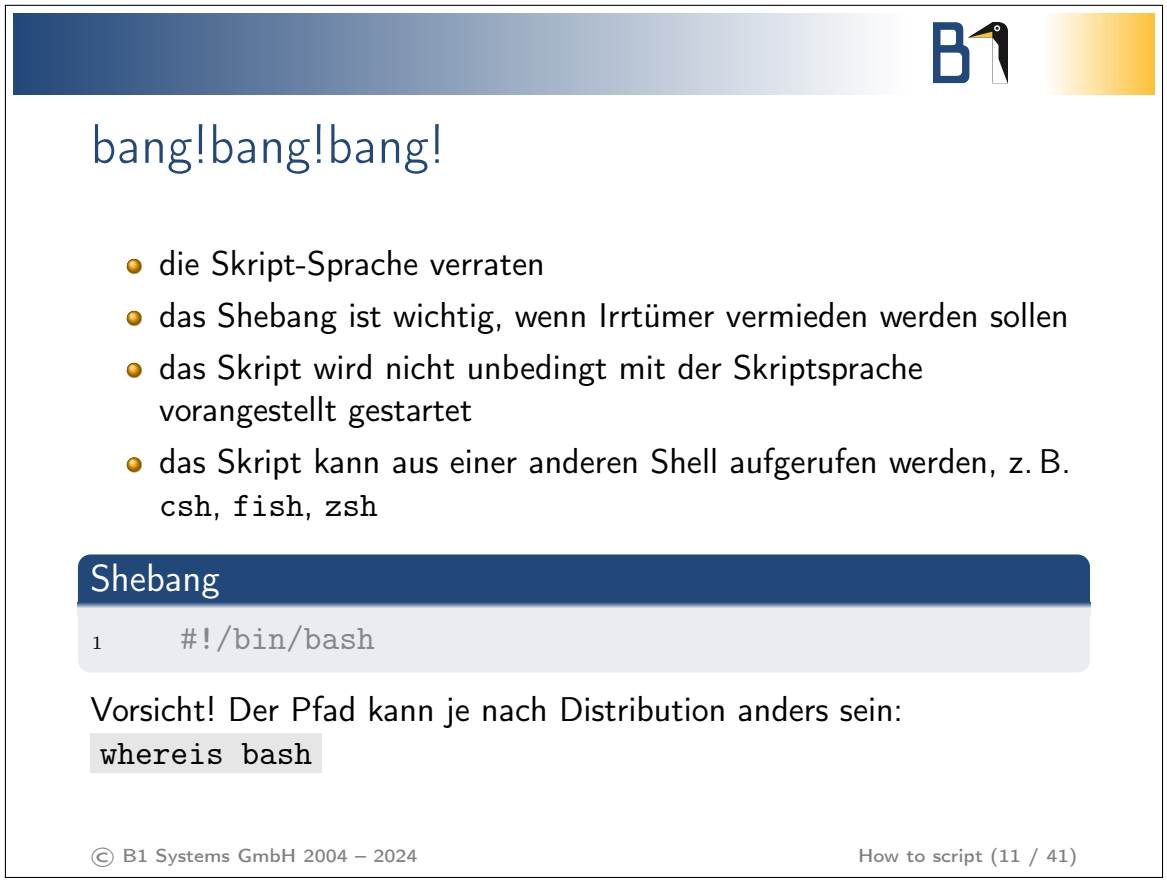

Alle machen beim Programmieren Fehler, das gehört dazu. Fehler sind keine Feinde, sondern Freunde, die dir helfen, anders über Dinge nachzudenken und die Perspektive zu wechseln. Aus ihnen kannst du lernen, besseren Code und bessere Skripte sowie Programme zu schreiben. Nutze daher deine Fehler als Chance, dich zu verbessern.

## 3.5. Konventionen



## 3.6. Shebang

Da du nicht wissen kannst, ob dein zukünftiges Ich dein Skript noch mit der Bash aufruft oder wie das ist, wenn dein Skript von jemand anderem genutzt wird, ist es sinnvoll, als erstes ein „Shebang“ zu schreiben.



bang!bang!bang!

- die Skript-Sprache verraten
- das Shebang ist wichtig, wenn Irrtümer vermieden werden sollen
- das Skript wird nicht unbedingt mit der Skriptsprache vorangestellt gestartet
- das Skript kann aus einer anderen Shell aufgerufen werden, z. B. `csh`, `fish`, `zsh`

**Shebang**

```
1    #!/bin/bash
```

Vorsicht! Der Pfad kann je nach Distribution anders sein:
`whereis bash`



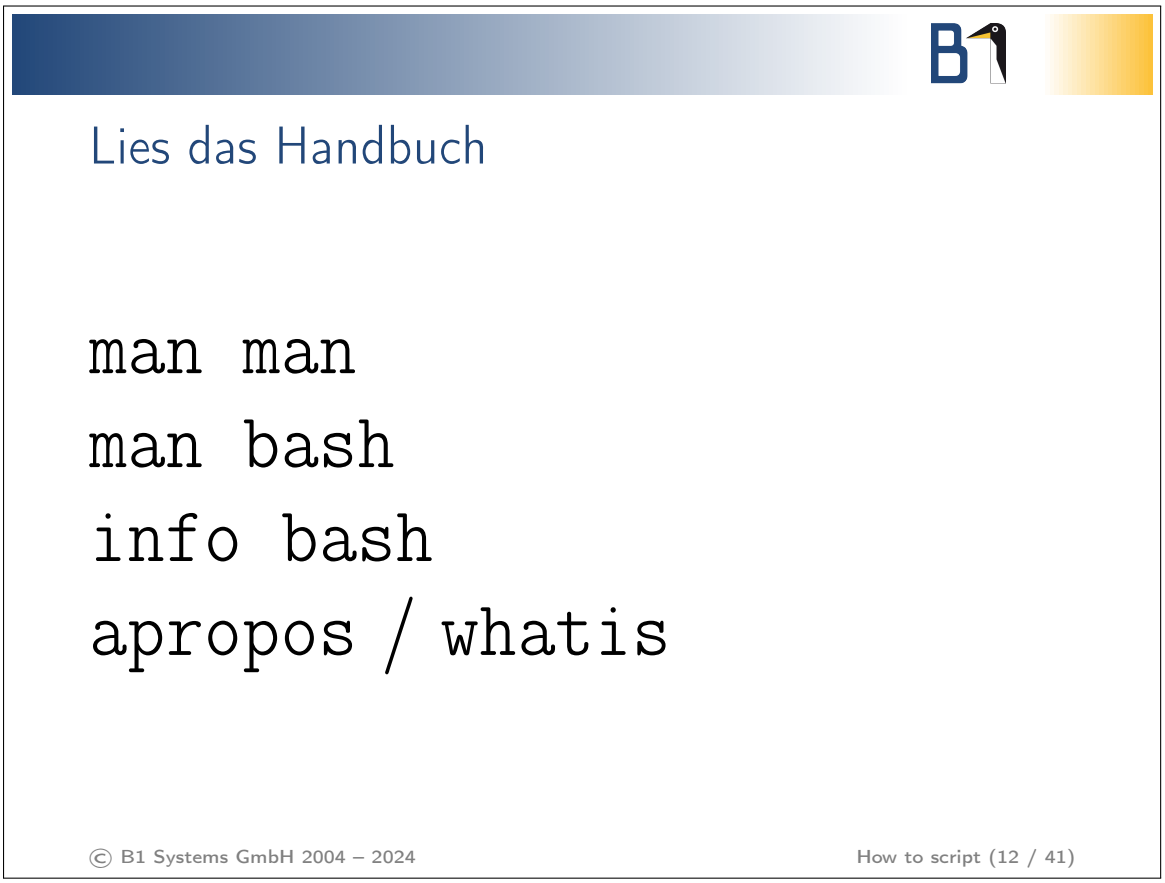

## 3.7. Debugging

Wie geht Debugging?

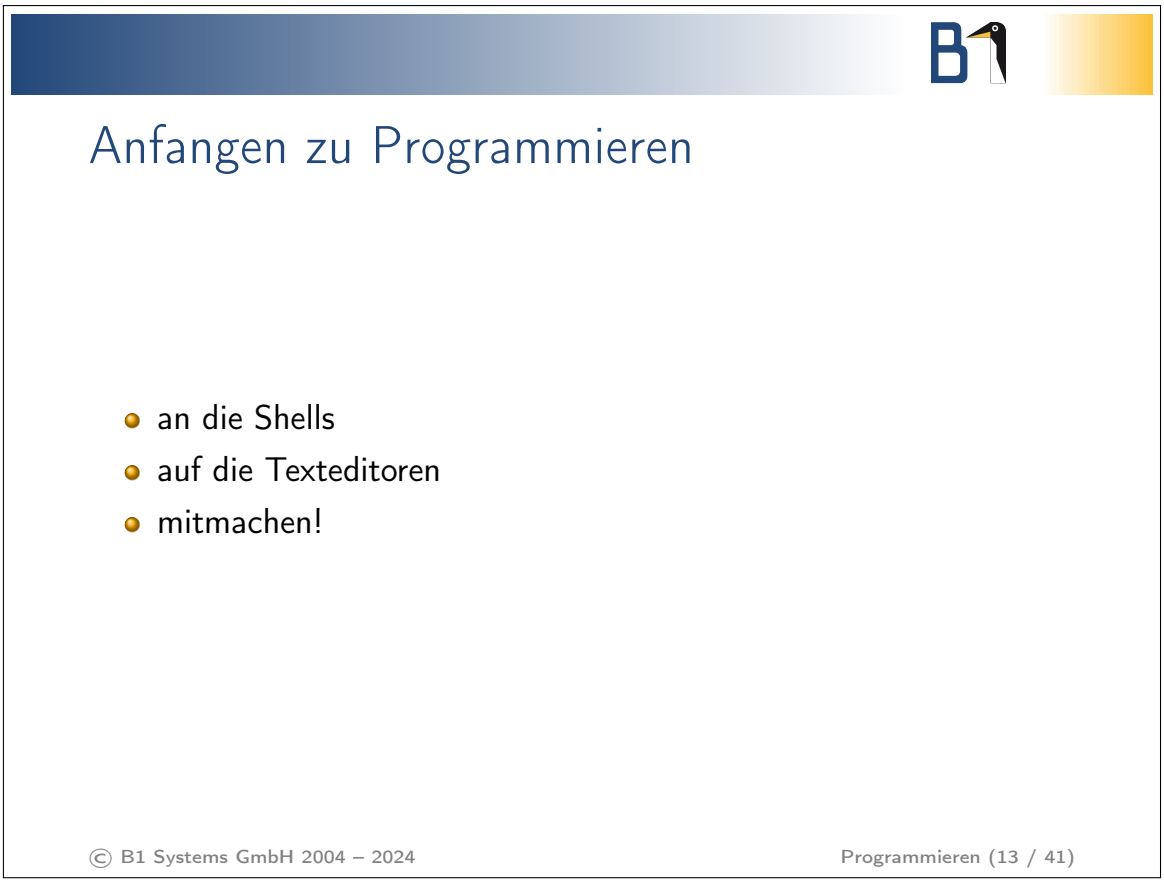
- beenden im Fehlerfall
  - im Skript `set -o errexit`
  - im Terminal `set -e`
- prüft ungesetzte Variablen (funktioniert nicht mit Umgebungsvariabeln)
  - im Skript `set -o nounset`
  - im Terminal `set -u`
- schreibt mit, was gerade ausgeführt wird, dabei werden Variablen aufgelöst

  - im Skript `set -o xtrace`
  - im Terminal `set -x`
- fancy Debugger muss nachinstalliert werden `shellcheck`

### 3.8. Hilfe

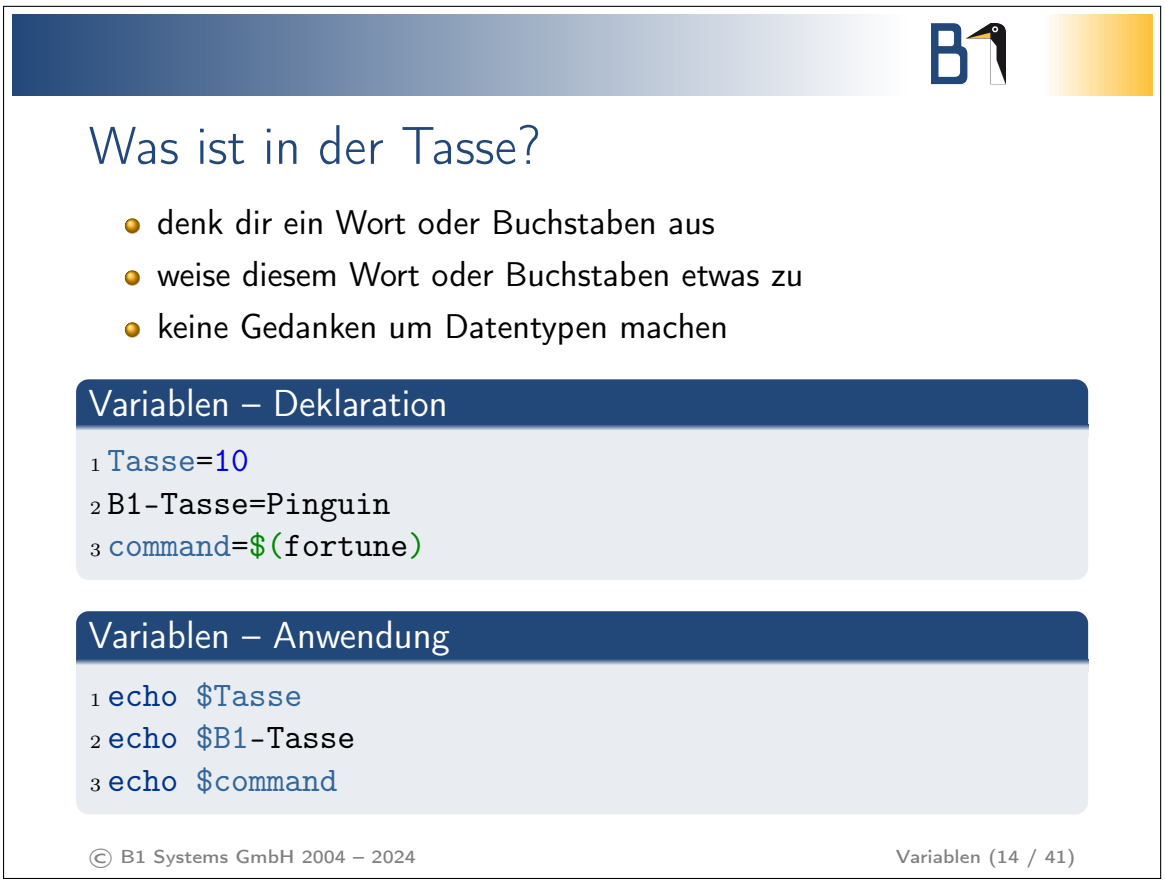


# 4. Programmieren



# 5. Variablen

In Variablen kannst du Dinge schreiben. Das können die Ergebnisse von Befehlen sein oder Werte (z. B. Zahlen) oder auch Text. Du entscheidest, was du dort hinein schreibst. Eine Variable kann eine beliebige Anzahl an Zeichen sein. Am Besten überlegst du dir einen sprechenden Namen für deine Variable.



## Was ist in der Tasse?

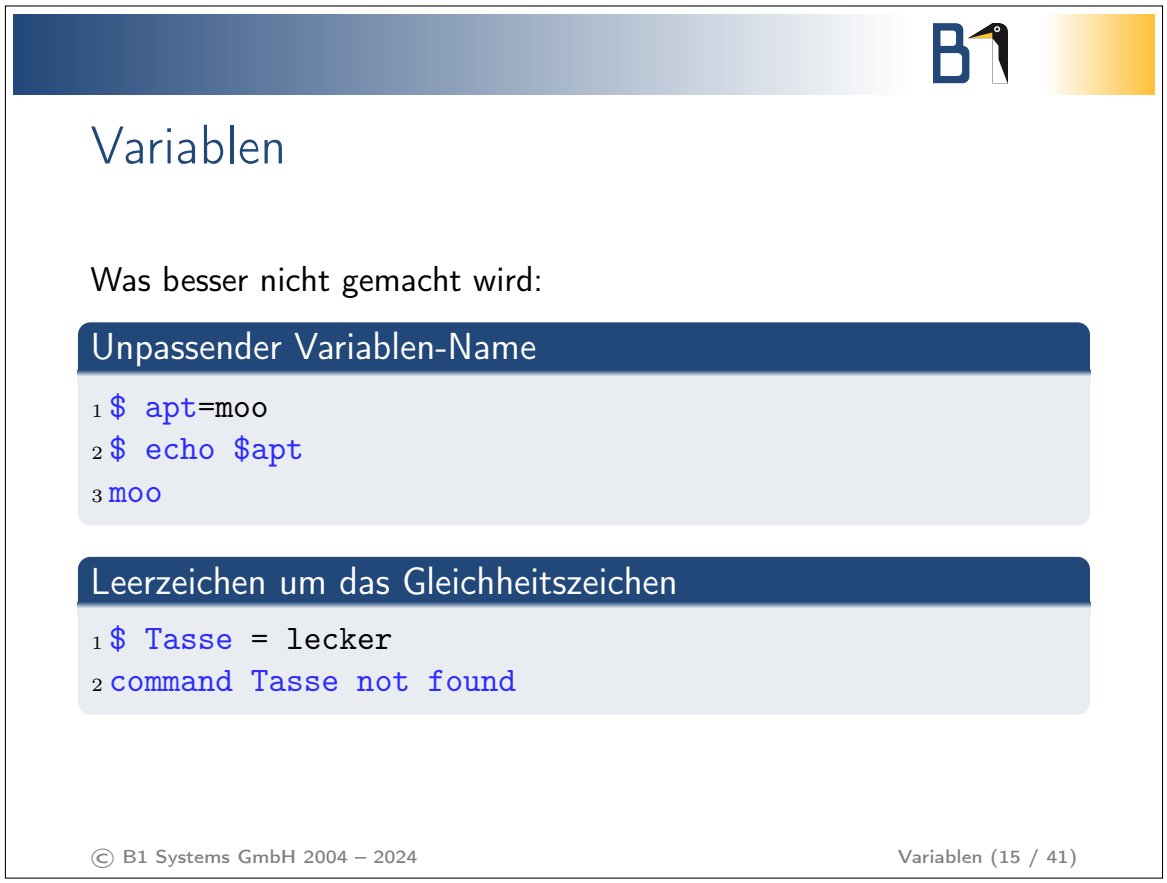
- denk dir ein Wort oder Buchstaben aus
- weise diesem Wort oder Buchstaben etwas zu
- keine Gedanken um Datentypen machen

### Variablen – Deklaration

```
1 Tasse=10
2 B1-Tasse=Pinguin
3 command=$(fortune)
```

### Variablen – Anwendung

```
1 echo $Tasse
2 echo $B1-Tasse
3 echo $command
```



## Variablen

Was besser nicht gemacht wird:

### Unpassender Variablen-Name

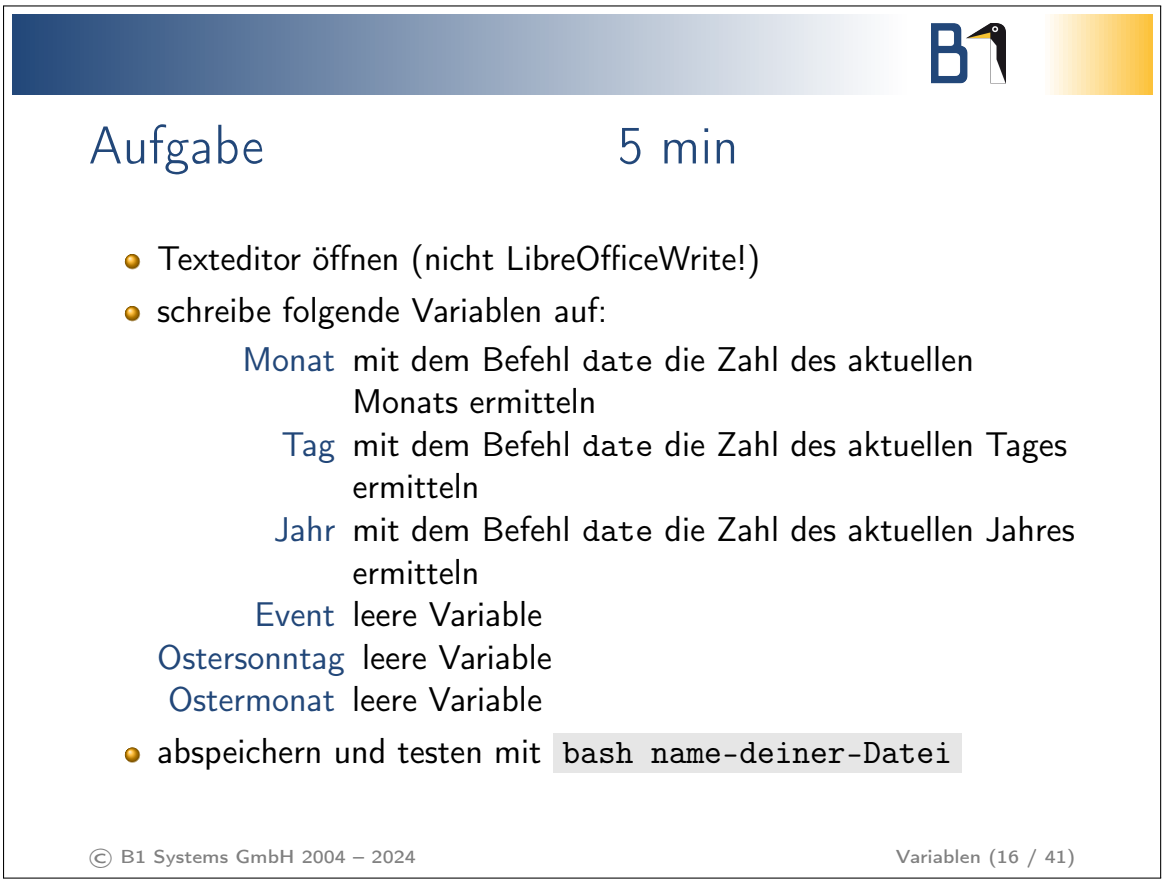
```
1 $ apt=moo
2 $ echo $apt
3 moo
```

### Leerzeichen um das Gleichheitszeichen

```
1 $ Tasse = lecker
2 command Tasse not found
```




Wichtig: es darf *kein* Leerzeichen um das = stehen, weil die Bash ein = mit Leerzeichen als Vergleich wertet!
Oft werden in Bash-Skripten oder auch anderen Programmiersprachen relativ weit oben im Skript die wichtigsten Variablen deklariert (schon mal leer belegt). Diese Variablen sind globale Variablen, was bedeutet, dass sie auch innerhalb von Funktionen genutzt und dort mit Werten belegt werden können, die auch in anderen Funktionen genutzt werden können und überall im Skript verwendbar sind. Variablen können jederzeit überall im Skript einfach zugewiesen werden.
Variablen, die nur innerhalb von Funktionen genutzt werden, sind außerhalb der Funktionen nicht mit dem Wert nutzbar, der ihnen innerhalb der Funktion gegeben wurde.



## Aufgabe 5 min

- Texteditor öffnen (nicht LibreOfficeWrite!)
- schreibe folgende Variablen auf:

| | |
|---|---|
| Monat | mit dem Befehl `date` die Zahl des aktuellen Monats ermitteln |
| Tag | mit dem Befehl `date` die Zahl des aktuellen Tages ermitteln |
| Jahr | mit dem Befehl `date` die Zahl des aktuellen Jahres ermitteln |
| Event | leere Variable |
| Ostersonntag | leere Variable |
| Ostermonat | leere Variable |

- abspeichern und testen mit `bash name-deiner-Datei`

# 6. Kommunikations-Skills

## Kommunikation mit Usern

Wenn Text im Terminal ausgegeben werden soll, wird `echo` als Befehl verwendet:

**Beispiel Ausgabe**

```
1 echo "Was macht ein Pirat am Computer?"
2 echo "Enter drücken"
```

Wenn du den Nutzer des Programms etwas eingeben lassen willst, kannst du `read` verwenden:

**Beispiel Eingabe**

```
1 echo "Was macht ein Pirat am Computer?"
2 read Antwort
3 echo "Enter drücken"
4 echo "deine Antwort war $Antwort"
```

 Kommunikations-Skills (17 / 41)

Es gibt bei bei der Ausgabe von Text mit `echo` bei den Anführungszeichen bedeutende Unterschiede. Wenn der Text nach dem `echo` von `" "` eingeschlossen wird, werden Variablen, die innerhalb des Textes stehen, aufgelöst:

**Beispiel doppelte Anführungsstriche**

```
1 $ ausgabeText="Hallo Welt"
2 $ echo "Grußformel $ausgabeText"
3 Grußformel Hallo Welt
```

Wenn der Text jedoch in einfachen `' '` steht, wird er so ausgegeben, wie er geschrieben wurde:

**Beispiel einfache Anführungsstriche**

```
1 $ echo 'Grußformel $ausgabeText'
2 Grußformel $ausgabeText
```

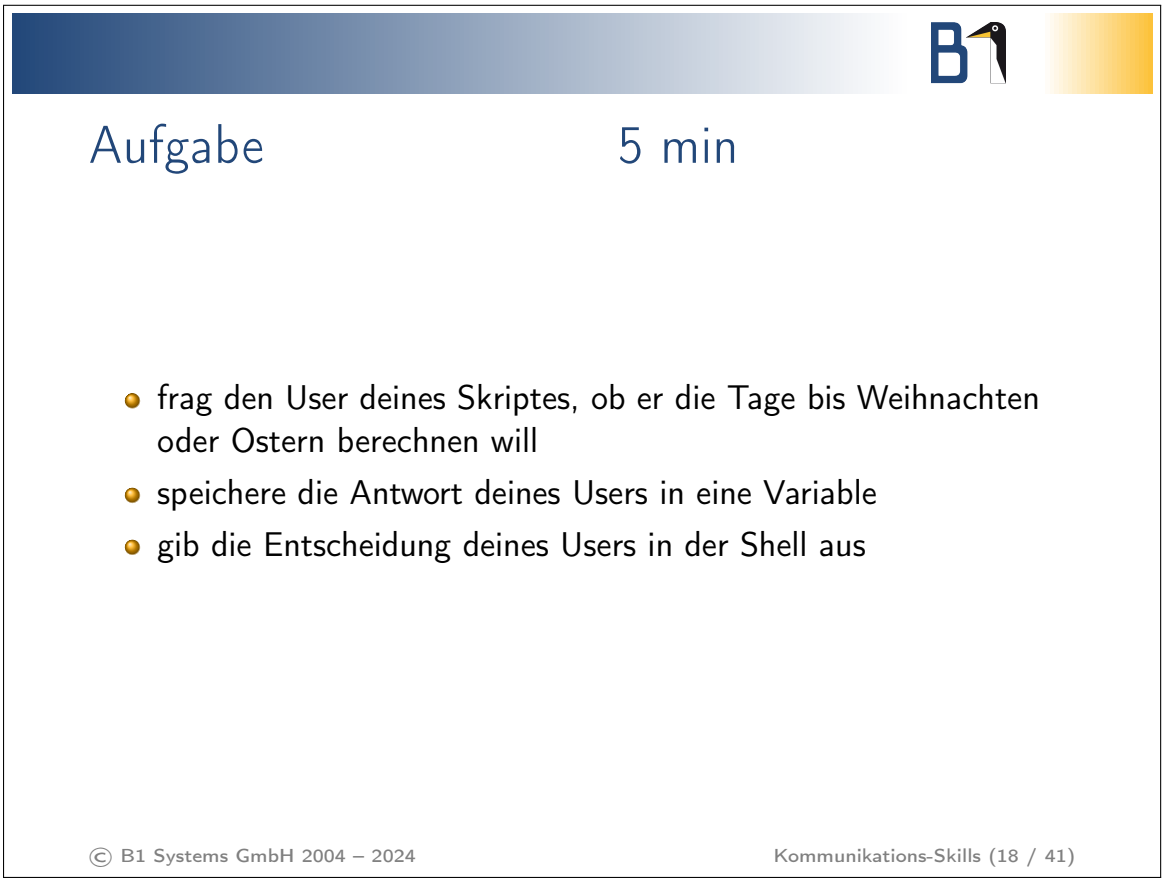
Natürlich kann auch der Nutzer des Skriptes nach bestimmten Dingen gefragt werden und Sachen eingeben, die dann wiederum den weiteren Verlauf des Skriptes bestimmen können. Dafür wird der Befehl

```
1 read
```

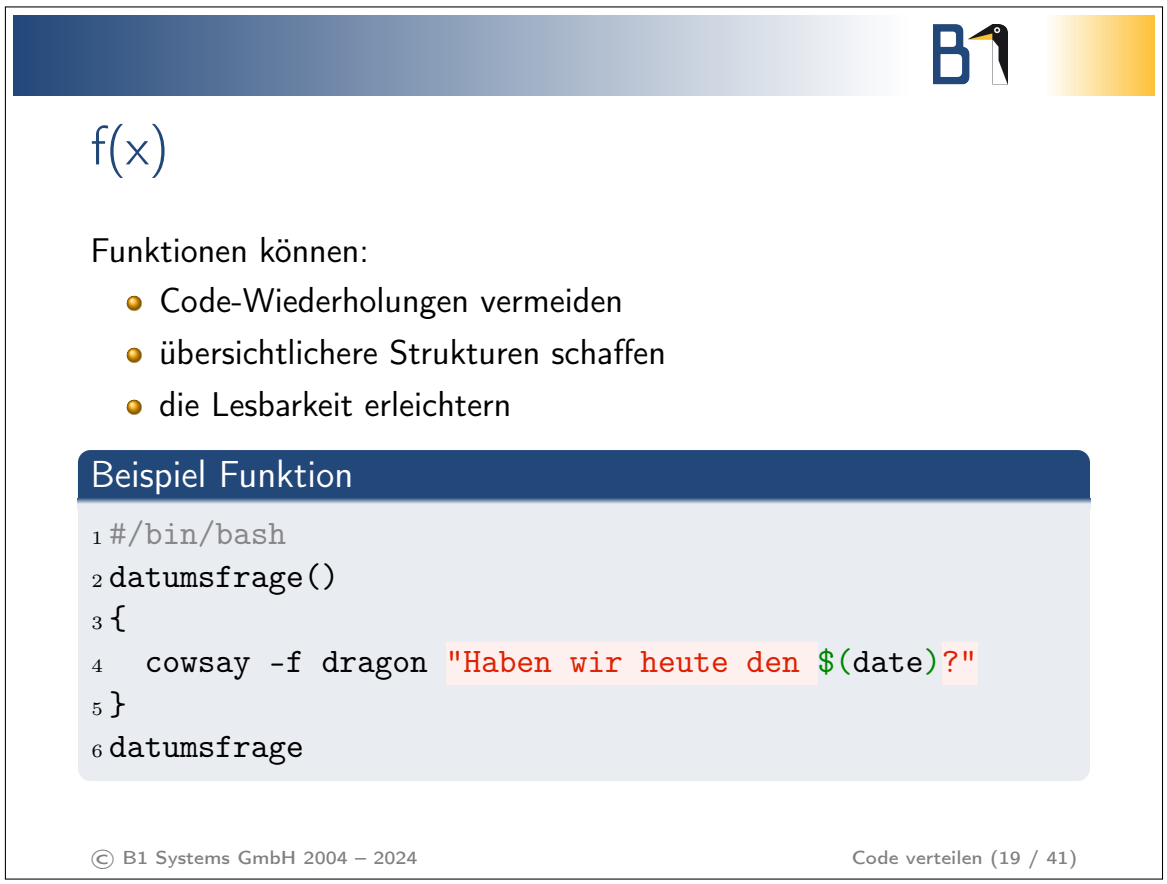
genutzt. Die Manpage zum `read`-Befehl ist nicht so leicht zu finden:

```
1 man bash
```

unter SHELL BUILTIN COMMANDS und dort unter read.
Mit dem Befehl kann mehr angestellt werden, als nur eine Nutzer-Eingabe abzufangen, aber hier beschränken wir uns auf die Nutzer-Eingabe.



# 7. Code verteilen



f(x)

Funktionen können:

- Code-Wiederholungen vermeiden
- übersichtlichere Strukturen schaffen
- die Lesbarkeit erleichtern

**Beispiel Funktion**

```
1 #/bin/bash
2 datumsfrage()
3 {
4   cowsay -f dragon "Haben wir heute den $(date)?"
5 }
6 datumsfrage
```


Code verteilen (19 / 41)

Funktionen sind Programm-Blöcke, die zusammengefasst und über einen Funktionsnamen aufgerufen werden. Dadurch kann das Skript strukturiert werden und es können Entscheidungen getroffen werden, wann welche Funktionen des Skriptes genutzt werden. Auch hier ist es gut, für die Lesbarkeit des Skriptes möglichst sprechende Funktionsnamen zu vergeben.
Es gibt zwei Methoden, Funktionen zu erstellen:

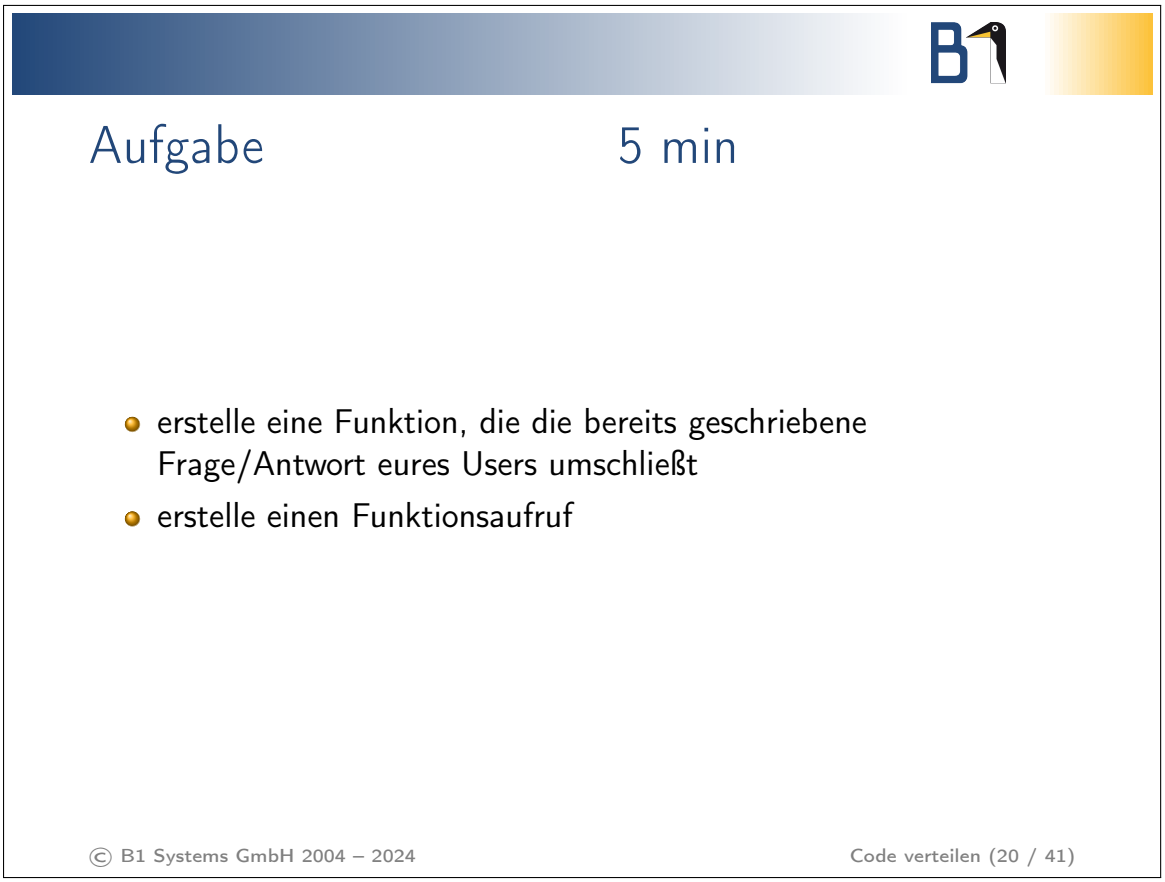
**Beispiel Funktion mit function Deklaration**

```
1 function meine_erste_Funktion()
2 {
3   echo "Hallo aus der Funktion: meine_erste_Funktion"
4 }
5 meine_erste_Funktion
```

**Beispiel Funktion ohne function Deklaration**

```
1 meine_zweite_Funktion()
2 {
3   echo "Hallo aus der Funktion: meine_zweite_Funktion"
4 }
5 meine_zweite_Funktion
```

Mit Funktionen kann noch mehr als nur das Strukturieren von Skripten erreicht werden, aber das ist ein Thema für Fortgeschrittene.



# 8. Logik

Die Logik ist in der Computerei sehr wichtig und meiner Meinung nach verbessert sich die Logik, je mehr Skripte geschrieben werden. Es geht dabei immer darum, ob etwas wahr (in der Bash 0) oder falsch (in der Bash ein anderer Wert als 0) ist. Es wird also etwas getestet. Dafür gibt es den Befehl: `test`. Allerdings wird mensch kaum auf Skripte in freier Wildbahn treffen, die diesen Befehl tatsächlich verwenden. Meistens wird die Abkürzung `[ ]` oder `[[ ]]` verwendet. Innerhalb der Klammern steht der zu testende Ausdruck.

## Wie funktioniert die Logik?

Warum testen?

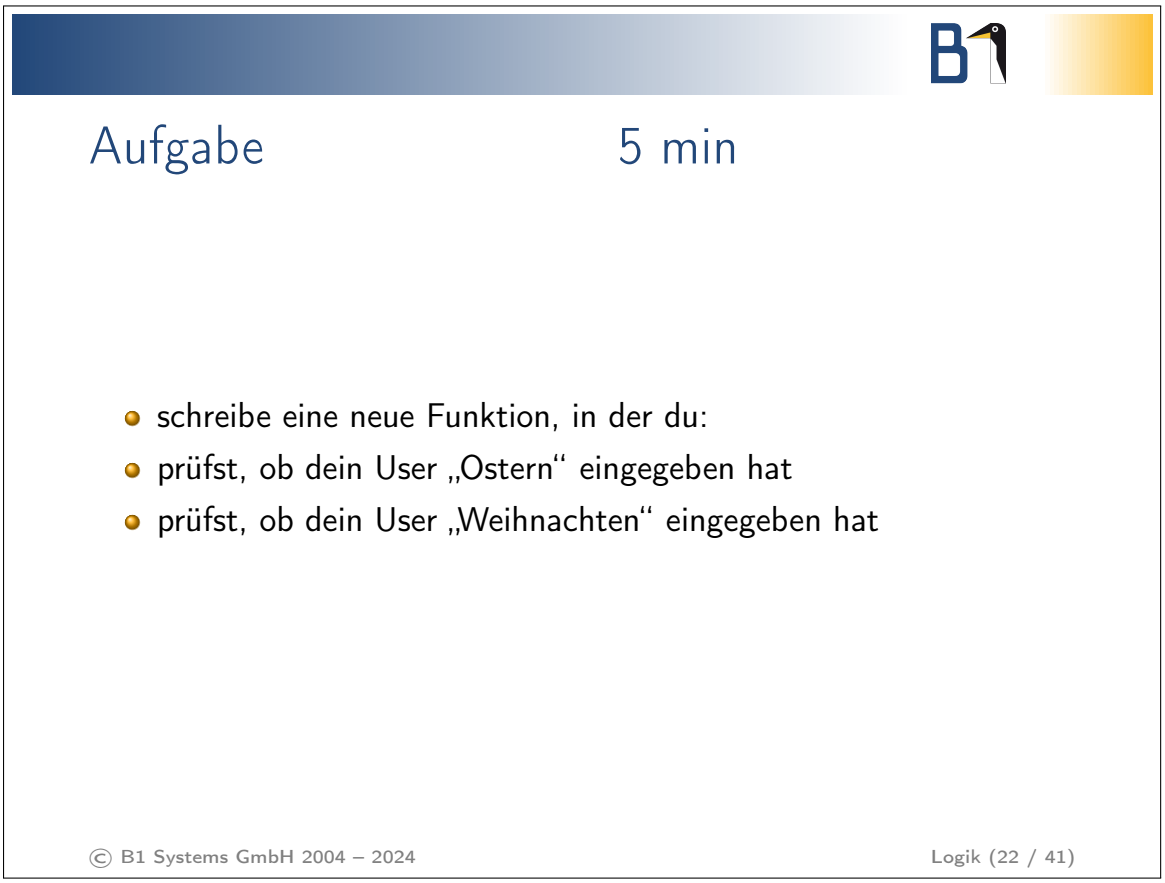
- ich will wissen, ob etwas richtig ist
- ich will abhängig vom Wert einer Variablen etwas tun
- ich will etwas nicht tun, wenn ...

**Beispiel Test**

```
1 read -p "Was packst du in die Tasse? " Tasse
2 test $Tasse = "Kaffee"
3 echo $?
4 [ $Tasse = "Kaffee" ] # andere Schreibweise
5 echo $?
6 [[ $Tasse = "Kaffee" ]] # andere Schreibweise
7 echo $?
```

© B1 Systems GmbH 2004 – 2024 Logik (21 / 41)

`$?` gibt den Status/Rückgabewert des letzten Befehls zurück. Dieser ist 0, wenn alles funktioniert hat, was in der Bash auch „wahr“ bedeutet.



Natürlich können Tests auch miteinander verknüpft werden. Dafür gibt es verschiedene LOGISCHE Operatoren:

## Wie funktioniert die Logik 1/3

UND prüfen, ob alle Tests wahr sind

### Beispiel UND

```
1 B1_Tasse="Tee"
2 Tasse="Kaffee"
3 test $B1_Tasse = "Tee" && [ $Tasse = "Kaffee" ]
4 echo $?
```

# Wie funktioniert die Logik 2/3

ODER prüfen, ob ein Test von mehreren Werten wahr ist

## Beispiel ODER

```
1 B1_Tasse="Tee"
2 Tasse="Kaffee"
3 test $B1_Tasse = "1x" || [ $Tasse = "Kaffee" ]
4 echo $?
```

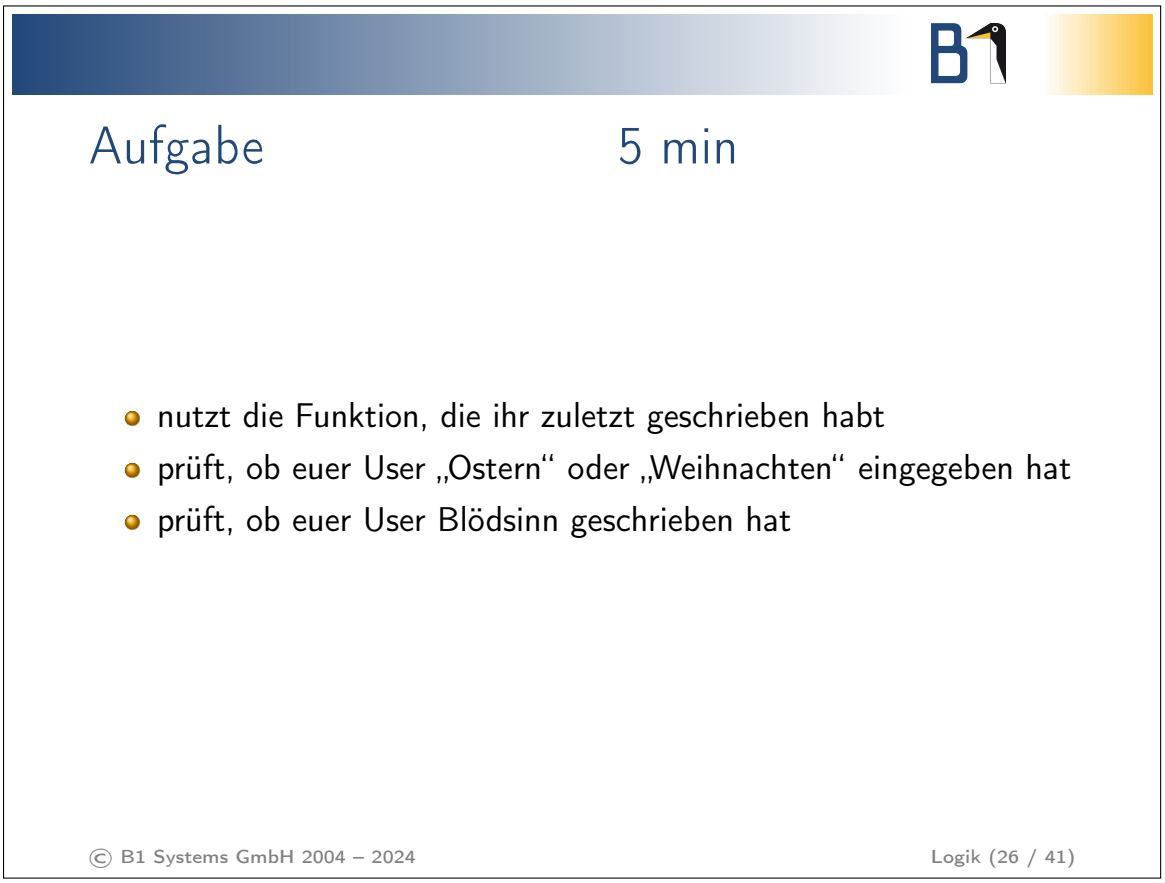
# Wie funktioniert die Logik 3/3

nicht prüfen, ob ein Test falsch ist

## Beispiel NICHT

```
1 Tasse="Tee"
2 test $Tasse != "Kaffee"
3 echo $?
```

Die Logik kann beliebig verknüpft werden – also viel Spaß!



# 9. Wenn ...

Diese Logik wird genutzt für Abzweigungen im Programm bzw. Skript. Wir kennen das aus Wenn-Dann-Sätzen in unserer Sprache.
Wenn eine Bedingung zutrifft machen wir etwas.
Wenn die Sonne scheint, dann setzen wir die Sonnenbrille auf.
Manchmal müssen wir auch erklären, was wir machen, wenn eine Bedingung nicht zutrifft.
Wenn die Sonne scheint, dann setzen wir die Sonnenbrille auf, sonst spannen wir den Regenschirm auf.
Manchmal haben wir auch mehrere Optionen, die wir prüfen. Wenn die Sonne scheint, dann setzen wir die Sonnenbrille auf, falls es schneit, holen wir den Schlitten aus dem Keller, sonst spannen wir den Regenschirm auf.
Bedingung, was wenn die Bedingung wahr ist, 2. Bedingung, was wenn Bedingung 2 wahr ist, falls keine Bedingung zutrifft, gilt Folgendes.

## Wenn Du nicht sofort tust, was ich sage, dann

...

`if`

Wenn, prüfe ob das wahr ist, dann...

### Beispiel Verzweigung

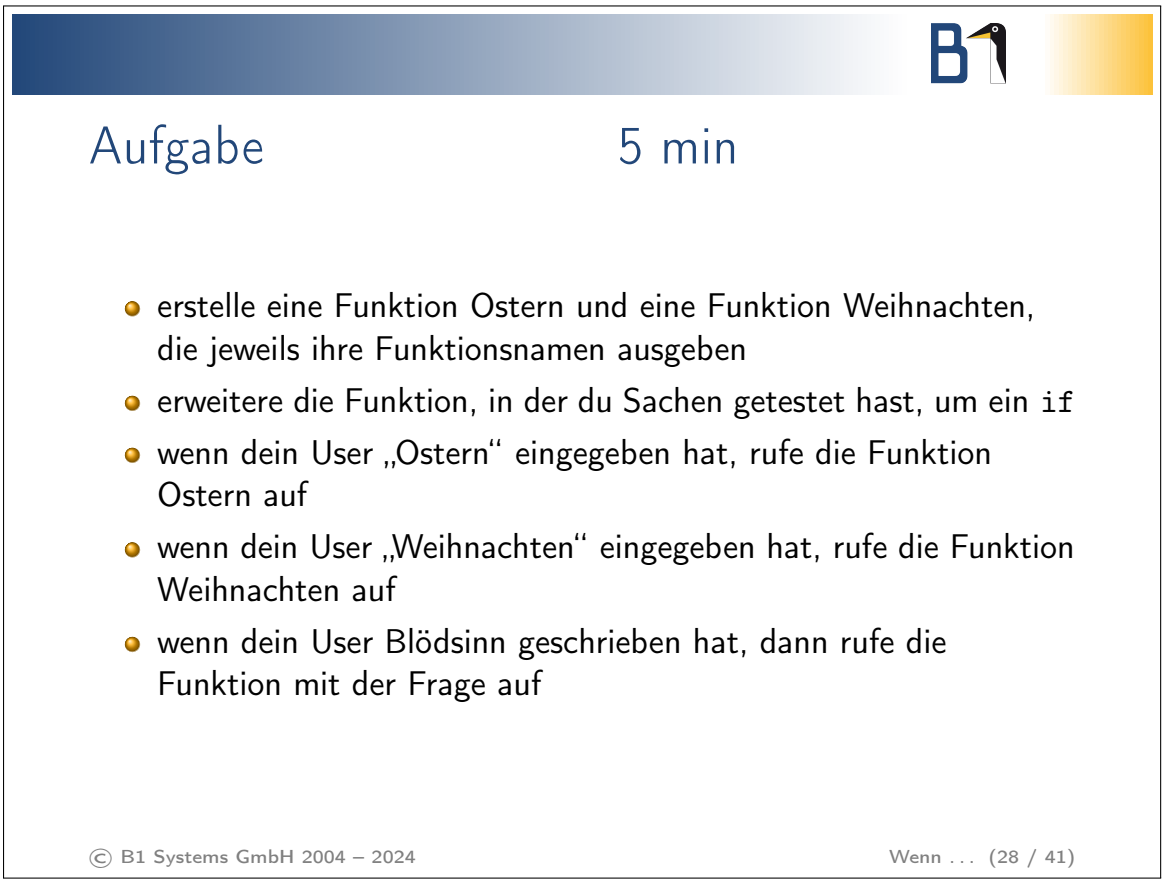
```
1 if test "$Tasse" = "Kaffee"; then echo "lecker"; fi
2 if [[ $Tasse == "Tee" ]] ; then echo "lecker" ; else
  ↪  cowsay "Ostfriese";fi
3 if [[ -z $Tasse ]] ; then echo "Tasse leer"; elif test
  ↪  $Tasse = "Kaffee" ; then cowsay "lecker $Tasse " ;
  ↪  else cowsay "So'n Tee";fi
```

`else` muss nicht unbedingt verwendet werden

`elif` Bedingung mehrfach Verkettungen

`-z` die Länge der ZEICHENKETTE ist Null

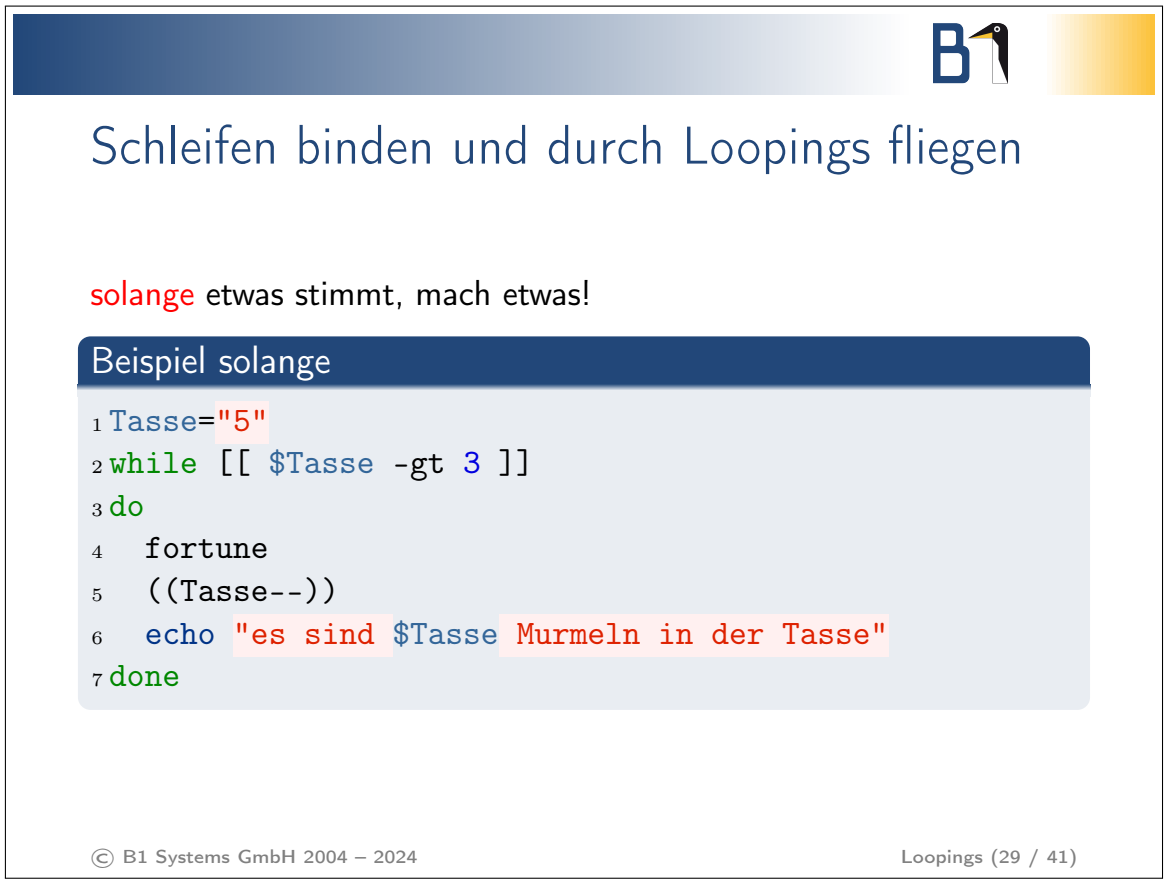
`=` vs `==` egal, solange Leerzeichen verwendet werden



# 10. Loopings

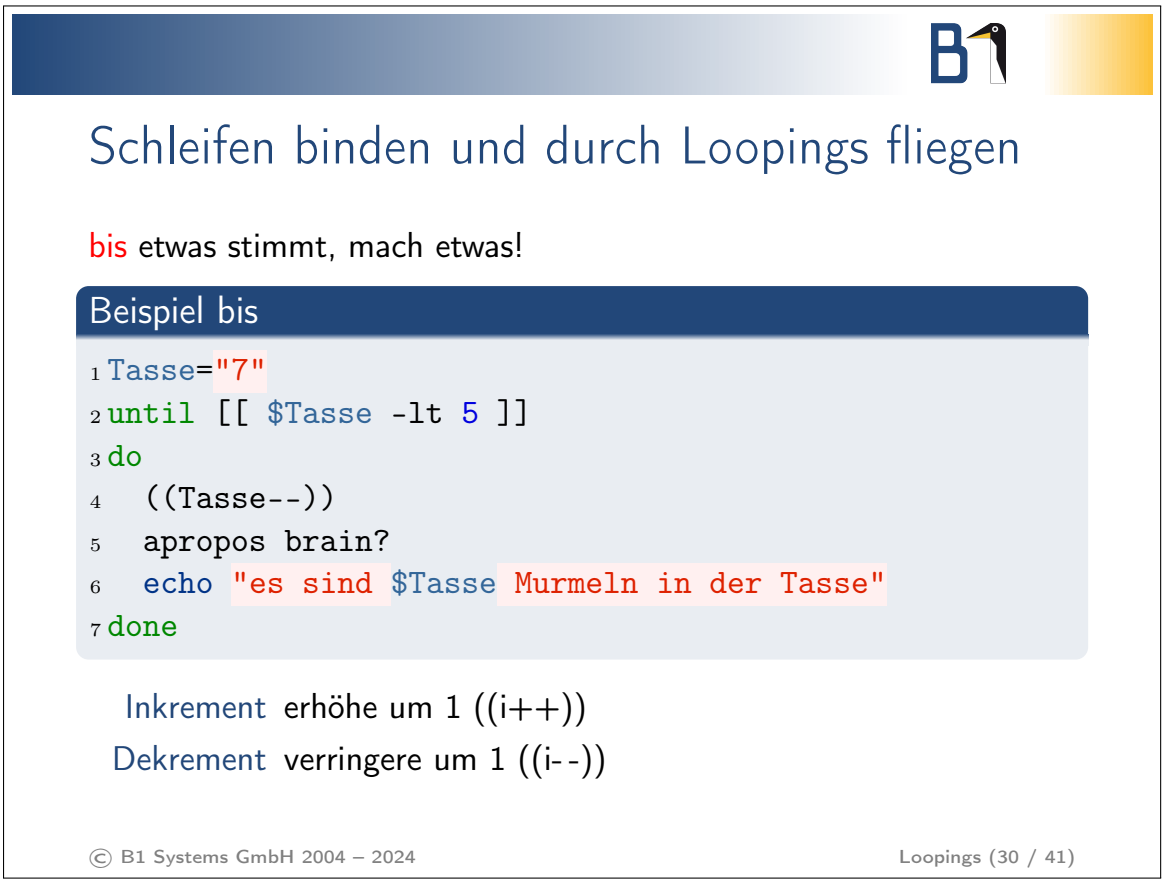
Es kann durchaus sinnvoll sein, etwas solange zu wiederholen, bis ein bestimmtes Ergebnis eingetreten ist. Dieses Konstrukt wird *Schleife* genannt. Diese sind nach folgendem Muster aufgebaut:
Schleifen-Kopf, hier steht die zu erfüllende Bedingung
Schleifen-Körper, hier stehen die Anweisungen, die wiederholt werden sollen, bis die Bedingung erfüllt ist.
Aus einer Schleife kann immer auch ausgebrochen werden mit dem Befehl `break`

Es gibt drei Formen von Schleifen:



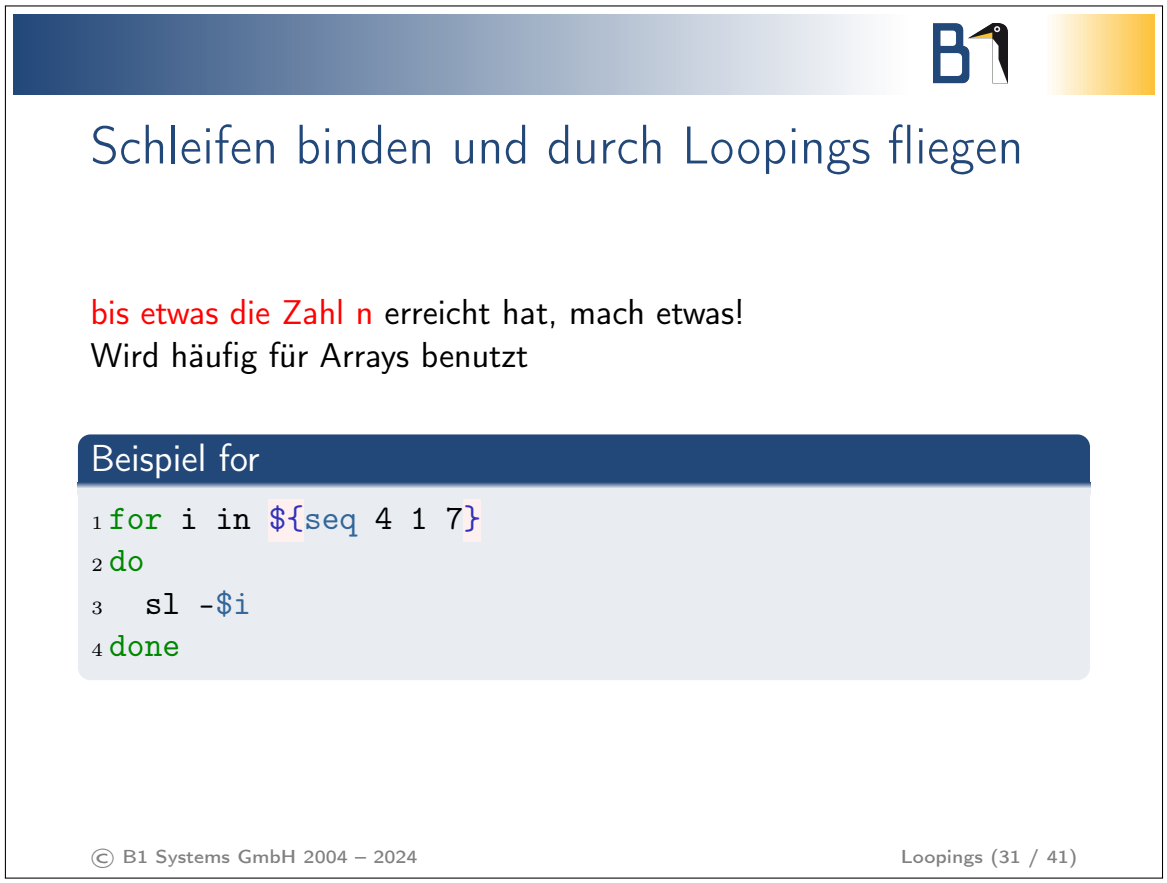

### 10.0.1. `while`

Bei einer While-Schleife wird solange etwas getan, *wie die Bedingung erfüllt ist*. Das bedeutet, dass zuerst die Bedingung geprüft wird. Ist diese wahr, werden die Anweisungen im Schleifen-Körper ausgeführt, wenn nicht, wird der Schleifen-Körper komplett übersprungen.



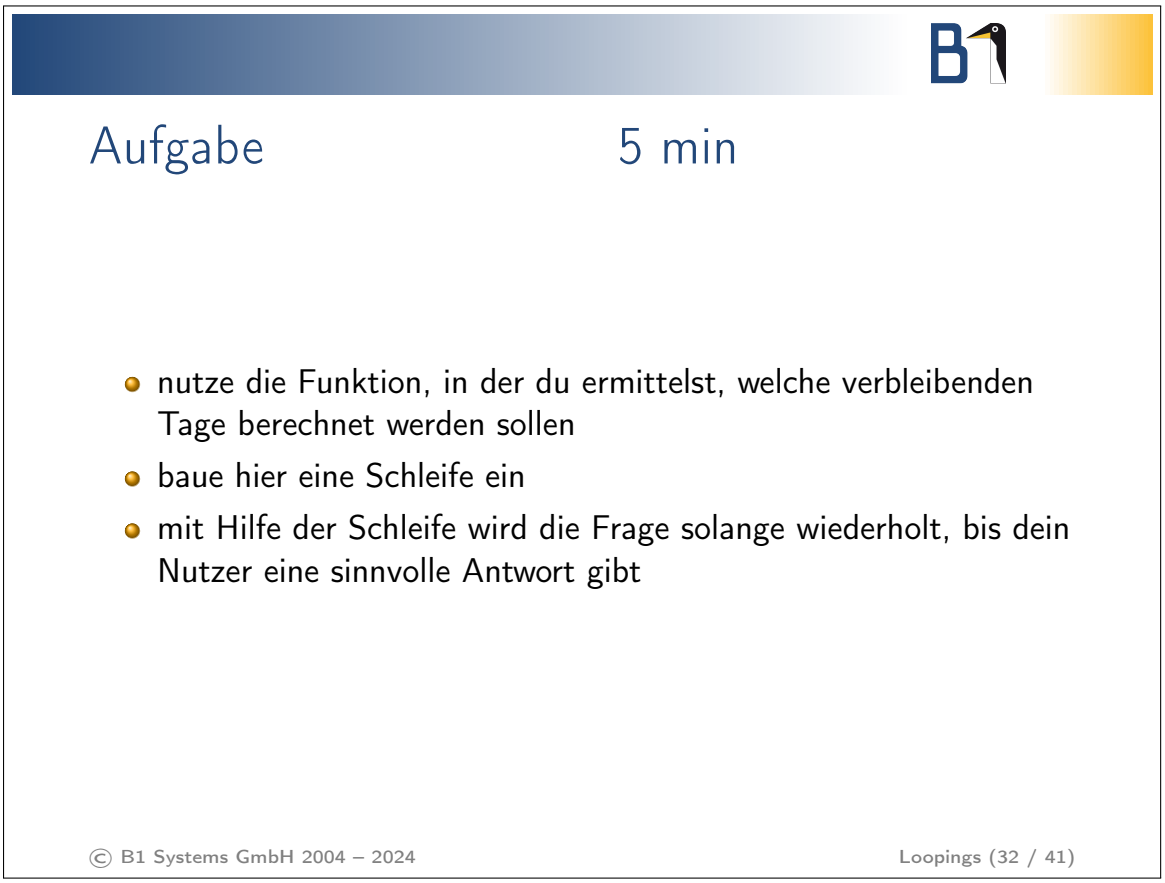

### 10.0.2. `until`

Bei einer Until-Schleife wird solange etwas getan, wie die Bedingung *nicht* erfüllt ist. Das bedeutet, dass zuerst die Bedingung geprüft wird. Ist diese *nicht* wahr, werden die Anweisungen im Schleifen-Körper ausgeführt, wenn sie wahr ist, wird der Schleifen-Körper komplett übersprungen.

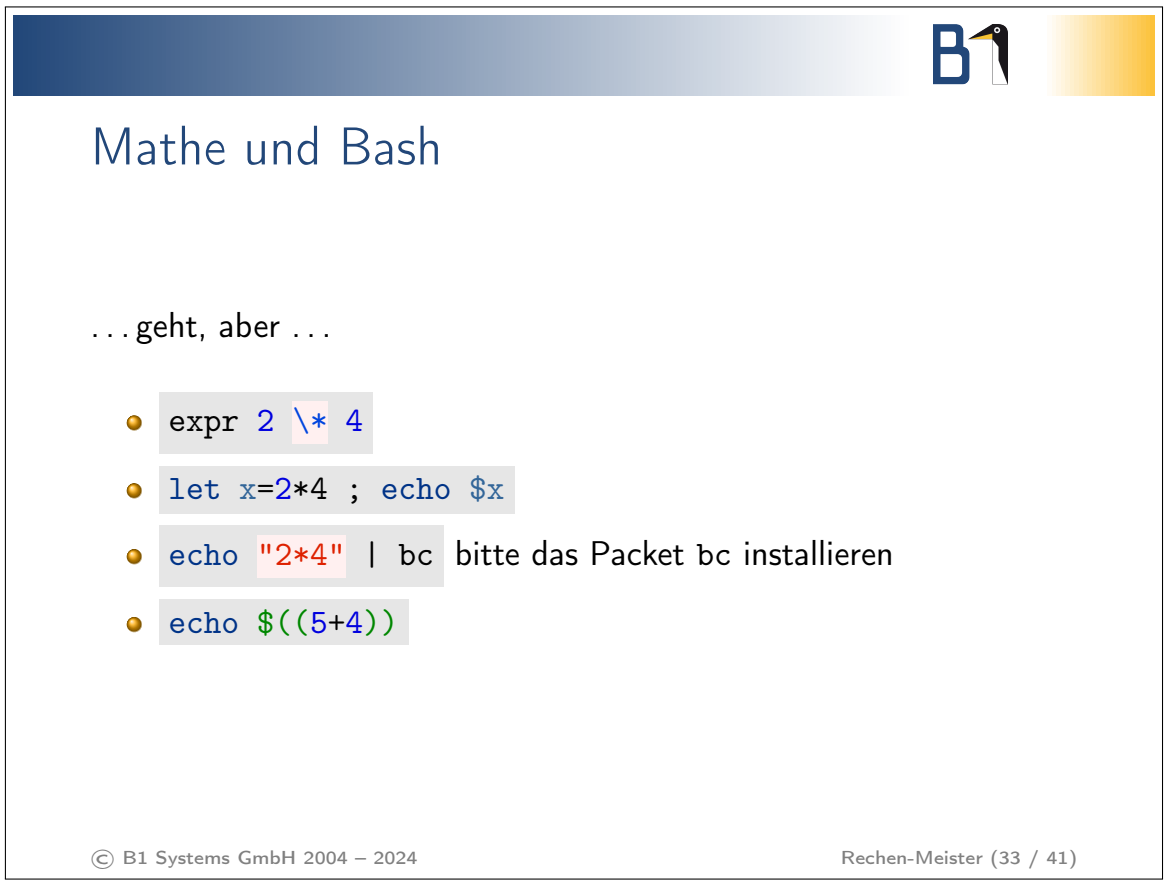


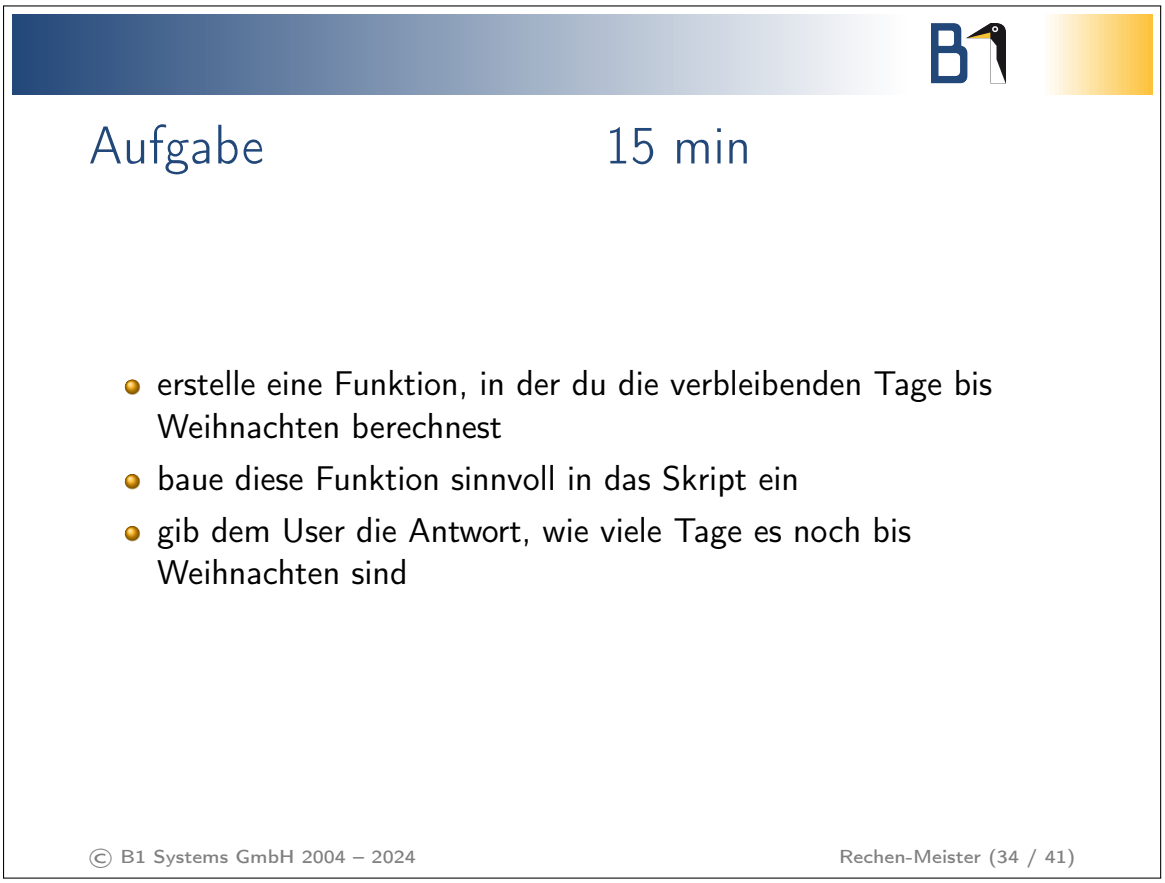

### 10.0.3. `for`

Eine for-Schleife wird verwendet, um durch ein Array oder eine Form von Liste zu gehen und bei jedem Element etwas zu machen. Wir werden hier aber nicht ausführlich darauf eingehen.



# 11. Rechen-Meister

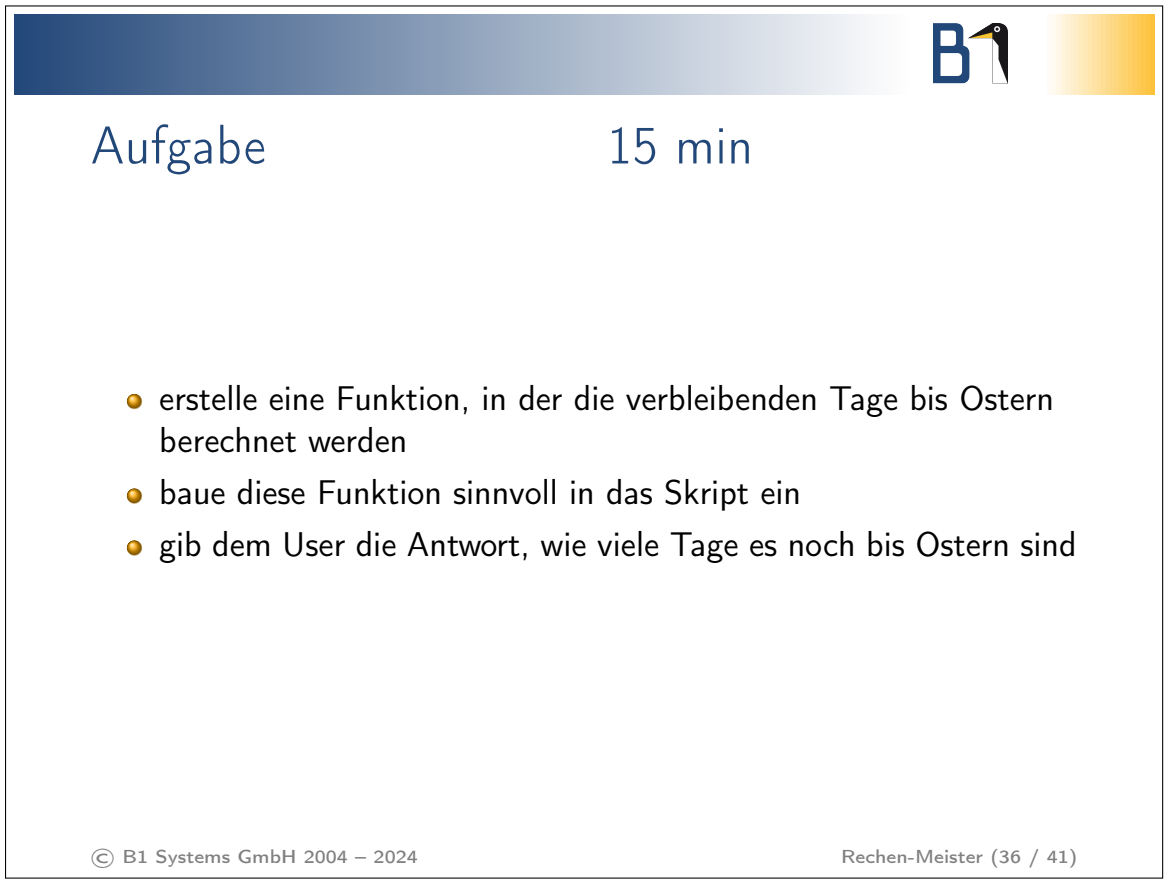




# Osterberechnung

## Osterformel nach Harold Spencer Jones

```
1  a=$(expr $Jahr % 19)
2  b=$(expr $Jahr / 100)
3  c=$(expr $Jahr % 100)
4  d=$(expr $b / 4)
5  e=$(expr $b % 4)
6  f=$(expr $(expr $b + 8) / 25)
7  g=$(expr $(expr $b - $f + 1) / 3)
8  h=$(expr $(expr 19 \* $a + $b - $d - $g + 15) % 30 )
9  i=$(expr $c / 4)
10 k=$(expr $c % 4)
11 l=$(expr $(expr 32 + 2 \* $e + 2 \* $i - $h - $k) % 7 )
12 m=$(expr $(expr $a + 11 \* $h + 22 \* $l) / 451)
13 Ostermonat=$(expr $(expr $h + $l - 7 \* $m + 114) / 31)
14 Ostersonntag=$(expr $(expr $h + $l - 7 \* $m + 114 ) % 31)
15 Ostersonntag=$(expr $Ostersonntag + 1)
16 echo "Ostersonntag ist am $Ostersonntag'.'$Ostermonat"
```



# Tipps und Tricks

**script** zeichnet auf, was in der Shell passiert (inklusive der Typos)

**scriptreplay** spielt wieder ab, was du aufgezeichnet hast, mit Timing-Datei sogar in Echtzeit

**Passwörter und Passphrases** haben in Skripten nichts verloren

**root-Rechte in Skripten erlangen** lass es sein

Bitte vergiss nicht, Fehler einzubauen, das hat den größten Lerneffekt!

# Teil I.
# Anhang

## A. Pipes

### Auf der Leitung stehen

Was ist eine Pipe?

- die Ausgabe eines Befehls an einen anderen weitergeben
- | wird durch folgende Tasten erstellt:

  AltGr

  +

  <>|

**Pipe-Beispiel**

```
fortune | cowsay | lolcat
```

das geht beliebig oft

## B. Im Fall von...

### Im Fall von, tue ...

Verschachtelungen

Wenn `if` für die Verschachtelungen unübersichtlich wird:

**case-select Beispiel**

```
case $Tasse in
  Kaffee)
    cowsay -f tux "Lecker Kaffee"
  ;;
  Tee)
    cowsay -f turtle "Lecker Tee"
  ;;
  *)
    cowsay -f ghostbusters "uaaahhh Gespenster"
  ;;
esac
```

lässt sich nur im Skript benutzen, nicht in der Kommandozeile

# C. Array

### Boxen

Wenn Variablen nicht ausreichen

Was sind Arrays?
eine Box für mehrere Werte

**Beispiel Arrays**

```
1 Box=(1 pinguin 3 4)
2 echo $Box
3 #mit index
4 echo ${Box[2]}
5 #gesamtes Array
6 echo ${Box[*]}
7 echo ${Box[@]}
```

hint: * und @ unterscheiden sich.
@ hat Newline zwischen den Index-Werten (wird beim Looping fliegen relevant).

# D. Lösungen der Übungsaufgaben

## D.1. Variabeln

**Variablen Deklaration**

```
1 #!/bin/bash
2
3 #Dieses Skript berechnet die verbleibenden Tage bis
  ↪  Weihnachten oder Ostern
4
5 ## Variabeln Deklaration
6 Monat=$(date +%m)
7 Tag=$(date +%e)
8 Jahr=$(date +%Y)
9 Event=""
10 Ostersonntag=""
11 Ostermonat=""
```

## D.2. Kommunikation mit Usern

**Frage nach gewünschtem Event**

```
read -p
  ↪  "Zu welchem Event sollen die verbleibenden Tage berechnet"
  ↪  "werden? (W)eihnachten\(O)stern " Event
echo "Du hast dich für $Event entschieden"
```

## D.3. Funktionen

**Frage nach gewünschtem Event**

```
EventFrage()
{
  read -p
    ↪  "Zu welchem Event sollen die verbleibenden Tage berechnet"
    ↪  "werden? (W)eihnachten\(O)stern " Event
  echo "Du hast dich für $Event entschieden"
}
EventFrage
```

## D.4. Logik

**Basis Logik**

```
CheckEvent()
{
# prüfe
[[ $Event = "Weihnachten" ]]
echo "Event ist Weihnachten: $?""
[[ $Event = "Ostern" ]]
echo "Event ist Ostern: $?"
}
EventFrage
EventCheck
```

## D.5. Logik Verknüpfungen

Erweitere Logik

```
CheckEvent()
{
# prüfe
[[ $Event = "Weihnachten" ]] || [[ $Event = "Ostern" ]]
echo "Es wurde Ostern oder Weihnachten eingegeben: $?"
[[ $Event != "Weihnachten" ]] || [[ $Event != "Ostern" ]]
echo "Es wurde was anderes als Ostern oder Weihnachten"
  ↪  "eingegeben: $?"
}
EventFrage
EventCheck
```

## D.6. Wenn, dann, sonst

if

```
Weihnachten()
{
  echo "Weihnachten"
}
Ostern()
{
  echo "Ostern"
}
CheckEvent()
{
  if [[ $Event = "Weihnachten" ]] || [[ $Event = "W" ]]
  then
    echo "berechne Tage bis Weihnachten"
    Weihnachten
  elif [[ $Event = "Ostern" ]] || [[ $Event = "O" ]]
  then
    echo "berechne Tage bis Ostern"
    Ostern
  else
    echo "Eingabe konnte nicht Verifiziert werden"
    EventFrage
  fi
}
```

## D.7. Schleifen

**Looping**

```
CheckEvent()
{
  while [[ -z $Event ]]
  do
    EventFrage

    # prüfe
    if [[ $Event = "Weihnachten" ]] || [[ $Event = "W" ]]
    then
      echo "berechne Tage bis Weihnachten"
      Event="W"
      break

    elif [[ $Event = "Ostern" ]] || [[ $Event = "O" ]]
    then
      echo "berechne Tage bis Ostern"
      Event="O"
    else
      echo "Eingabe konnte nicht Verifiziert werden"
      Event=""
    fi
  done
  if [[ $Event = "W" ]]
  then
    Weihnachten
  else
    Ostern
  fi
}
```

## D.8. Weihnachtsberechnung

**Weihnachtsberechnung**

```
Weihnachten()
{
 if [[ $Monat == 12 && $Tag -ge 24 &&  $Tag -lt 27 ]]
 then
   figlet "Happy Xmas"
   figlet -t "Es ist Weihnachtszeit"
 elif [[ $Monat == 12 && $Tag -ge 26 ]]
```

```
8  then
9      ((jahr++))
10     for i in $(seq $Monat 1 11)
11     do
12       Monatstage=$(expr $Monatstage + $(date -d
          ↪  "$jahr-$Monat-1 +1 month -1 day" +%d ))
13     done
14     Tage_bis_Weihnachten=$(expr $Monatstage + 24 )
15   elif [[ $Monat -lt 12 ]]
16   then
17     for i in $(seq $Monat 1 11)
18     do
19     Monatstage=$(expr $Monatstage + $(date -d
        ↪  "$jahr-$Monat-1 +1 month -1 day" +%d ))
20     done
21     Tage_bis_Weihnachten=$(expr $Monatstage + 24 )
22   else
23     Tage_bis_Weihnachten=$(expr $Monatstage + $(expr 24 -
        ↪  $(date +%d)))
24   fi
25   figlet "Es sind noch " $Tage_bis_Weihnachten
      ↪  " bis Weihnachten"
26 }
```

## D.9. Osterberechnung

Osterberechnung

```
1 function Osterberechnung()
2 {
3 # Modulo (Berechnung mit Rest)
4   a=$(expr $Jahr % 19)
5   echo $a
6 # geteilt/ Division
7   b=$(expr $Jahr / 100)
8   echo $b
9 # Modulo (Berechnung mit Rest)
10  c=$(expr $Jahr % 100)
11  echo $c
12 # geteilt/Division
13  d=$(expr $b / 4)
14  echo $d
15 # Modulo (Berechnung mit Rest)
16  e=$(expr $b % 4)
```

```
17   echo $e
18 # variable b + 8, das Ergebnis geteilt durch 25
19   f=$(expr $(expr $b + 8) / 25)
20   echo $f
21 # Variable b minus Variable f + 1, das Ergebnis durch 3
22   g=$(expr $(expr $b - $f + 1) / 3)
23   echo $g
24 # 19 mal Variable a + Variable b - variable d - variable g
   ↪  + 15, das Ergebnis Modulo 30
25   h=$(expr $(expr 19 \* $a + $b - $d - $g + 15) % 30 )
26   echo $h
27 # Variable c durch 4
28   i=$(expr $c / 4)
29   echo $i
30 # Variable c Modulo 4
31   k=$(expr $c % 4)
32   echo $k
33 # ( 32 + 2 mal Variable e + 2 mal Variable i  - Variable h
   ↪  - Variable k ) Modulo 7
34   l=$(expr $(expr 32 + 2 \* $e + 2 \* $i - $h - $k) % 7 )
35   echo $l
36 # ( Variable a + 11 mal Variable h + 22 mal Variable l)
   ↪  durch 451
37   m=$(expr $(expr $a + 11 \* $h + 22 \* $l) / 451)
38   echo $m
39 # ( Variable h + Variable l - 7 mal Variable m + 144) durch
   ↪  31
40   Ostermonat=$(expr $(expr $h + $l - 7 \* $m + 114) / 31)
41   echo $Ostermonat
42 # ( Variable h + Variable l - 7 mal Variable m + 144)
   ↪  Modulo 31
43   Ostersonntag=$(expr $(expr $h + $l - 7 \* $m + 114 ) %
     ↪  31)
44   Ostersonntag=$(expr $Ostersonntag + 1)
45
46 }
47
48 function Ostern()
49 {
50   Osterberechnung
51   Start=$(expr $Ostersonntag - 2)
52   echo $Start "Start"
53   now=$(date --date="today" +%j)
54
55     if [[ $Monat == $Ostermonat ]]
56     then
```

```
57
58          if [[ $Tag = $Start || $Tag -gt $Start && $Tag -lt
            ↪  $(expr $Start + 4) ]]
59          then
60              cowsay -f bunny "Frohe Ostern"
61          fi
62
63          if [[ $Tag -lt $Start ]]
64          then
65              TagesAnzahl=$(expr $Start - $Tag)
66              echo -e $gruen "Es sind noch " $TagesAnzahl
                ↪  " bis Ostern "$nc
67          elif [ $Tag -gt $(expr $Start +5) ]
68          then
69              echo -e $rot
                ↪  "für diese Jahr leider schon vorbei :-( "
                ↪  $nc
70          fi
71      elif [[ $Monat -lt $Ostermonat ]]
72      then
73          easter=$(date --date="$Jahr-0$Ostermonat-$Start"
            ↪  +%j)
74
75          Tage=$(expr $easter - $now)
76          figlet "Es sind noch " $Tage " bis zu Ostern"
77      else [[ $Monat -gt $Ostermonat ]]
78          easter=$(date --date="$jahr-0$Ostermonat-$Start"
            ↪  +%j)
79
80          Tage=$(expr $now + $easter)
81          figlet "Es sind noch " $Tage " bis Ostern"
82      fi
83  }
```

# E. cheatsheet

Hier ist eine Variante des vollständigen Skriptes

```
1 #/bin/bash
2
3 ### Variabeln Deklaration
4 Monat=$(date +%m)
5 Tage=$(date +%e)
6 Jahr=$(date +%Y)
7
8 Event=""
9 Ostersonntag=""
10 Ostermonat=""
11
12 # Farbgestaltung mit Shellescapes für echo (mit -e benutzbar)
13 gruen="\e[32m"
14 blau="\e[34m"
15 rot="\033[1;31m"
16 nc="\033[0m"
17
18 EventFrage()
19 {
20     read -p "Zu welchem Event sollen die verbleibenden Tage berechnet werden?"
       ↪ "(W)eihnachten\(O)stern "
       ↪ Event
21 }
22
23 CheckEvent()
24 {
25 while [[ -z $Event ]]
26 do
27     EventFrage
28
29     # prüfe
30     if [[ $Event = "Weihnachten" ]] || [[ $Event = "W" ]]
31     then
32         echo "berechne Tage bis Weihnachten"
33         Event="W"
34         break
35
36     elif [[ $Event = "Ostern" ]] || [[ $Event = "O" ]]
37     then
38         echo "berechne Tage bis Ostern"
39         Event="O"
40     else
```

```
41          echo "Eingabe konnte nicht Verifiziert werden"
42          Event=""
43      fi
44 done
45 if [[ $Event = "W" ]]
46 then
47   Weihnachten
48 else
49   Ostern
50 fi
51 }
52
53 function Osterberechnung()
54 {
55
56 # Modulo (Berechnung mit Rest)
57   a=$(expr $Jahr % 19)
58   echo $a
59 # geteilt/ Division
60   b=$(expr $Jahr / 100)
61   echo $b
62 # Modulo (Berechnung mit Rest)
63   c=$(expr $Jahr % 100)
64   echo $c
65 # geteilt/Division
66   d=$(expr $b / 4)
67   echo $d
68 # Modulo (Berechnung mit Rest)
69   e=$(expr $b % 4)
70   echo $e
71 # variable b + 8, das Ergebnis geteilt durch 25
72   f=$(expr $(expr $b + 8) / 25)
73   echo $f
74 # Variable b minus Variable f + 1, das Ergebnis durch 3
75   g=$(expr $(expr $b - $f + 1) / 3)
76   echo $g
77 # 19 mal Variable a + Variable b - variable d - variable g +
   ↪  15, das Ergebnis Modulo 30
78   h=$(expr $(expr 19 \* $a + $b - $d - $g + 15) % 30 )
79   echo $h
80 # Variable c durch 4
81   i=$(expr $c / 4)
82   echo $i
83 # Variable c Modulo 4
84   k=$(expr $c % 4)
85   echo $k
86 # ( 32 + 2 mal Variable e + 2 mal Variable i  - Variable h -
   ↪  Variable k ) Modulo 7
```

```
87   l=$(expr $(expr 32 + 2 \* $e + 2 \* $i - $h - $k) % 7 )
88   echo $l
89 # ( Variable a + 11 mal Variable h + 22 mal Variable l) durch
     ↪  451
90   m=$(expr $(expr $a + 11 \* $h + 22 \* $l) / 451)
91   echo $m
92 # ( Variable h + Variable l - 7 mal Variable m + 144) durch 31
93   Ostermonat=$(expr $(expr $h + $l - 7 \* $m + 114) / 31)
94   echo $Ostermonat
95 # ( Variable h + Variable l - 7 mal Variable m + 144) Modulo 31
96   Ostersonntag=$(expr $(expr $h + $l - 7 \* $m + 114 ) % 31)
97   Ostersonntag=$(expr $Ostersonntag + 1)
98
99 }
100
101 function Ostern()
102 {
103   Osterberechnung
104   Start=$(expr $Ostersonntag - 2)
105   echo $Start "Start"
106   now=$(date --date="today" +%j)
107
108     if [[ $Monat == $Ostermonat ]]
109     then
110
111         if [[ $Tage = $Start || $Tage -gt $Start && $Tage -lt
              ↪  $(expr $Start + 4) ]]
112         then
113             cowsay -f bunny "Frohe Ostern"
114         fi
115
116         if [[ $Tage -lt $Start ]]
117         then
118             TagesAnzahl=$(expr $Start - $Tage)
119             echo -e $gruen "Es sind noch " $TagesAnzahl
                  ↪  " bis Ostern "$nc
120         elif [ $Tage -gt $(expr $Start +5) ]
121         then
122             echo -e $rot
                  ↪  "für diese Jahr leider schon vorbei :-( " $nc
123         fi
124     elif [[ $Monat -lt $Ostermonat ]]
125     then
126         easter=$(date --date="$Jahr-0$Ostermonat-$Start" +%j)
127
128         Tage=$(expr $easter - $now)
129         figlet "Es sind noch " $Tage " bis zu Ostern"
130     else [[ $Monat -gt $Ostermonat ]]
```

```
131         easter=$(date --date="$Jahr-0$Ostermonat-$Start" +%j)
132
133         Tage=$(expr $now + $easter)
134         figlet "Es sind noch " $Tage " bis Ostern"
135     fi
136 }
137
138 Weihnachten()
139 {
140     if [[ $Monat == 12 && $Tage -ge 24 &&  $Tage -lt 27 ]]
141     then
142       figlet "Happy Xmas"
143       figlet -t "Es ist Weihnachtszeit"
144     elif [[ $Monat == 12 && $Tage -ge 26 ]]
145     then
146         ((jahr++))
147         for i in $(seq $Monat 1 11)
148         do
149             Monatstage=$(expr $Monatstage + $(date -d
                ↪  "$jahr-$Monat-1 +1 month -1 day" +%d ))
150         done
151         Tage_bis_Weihnachten=$(expr $Monatstage + 24 )
152     elif [[ $Monat -lt 12 ]]
153     then
154          for i in $(seq $Monat 1 11)
155          do
156              Monatstage=$(expr $Monatstage + $(date -d
                 ↪  "$Jahr-$Monat-1 +1 month -1 day" +%d ))
157          done
158         Tage_bis_Weihnachten=$(expr $Monatstage + 24 )
159   else
160       Tage_bis_Weihnachten=$(expr $Monatstage + $(expr 24 -
          ↪  $(date +%d)))
161   fi
162       figlet "Es sind noch " $Tage_bis_Weihnachten
          ↪  " bis Weihnachten"
163 }
164
165 CheckEvent
```