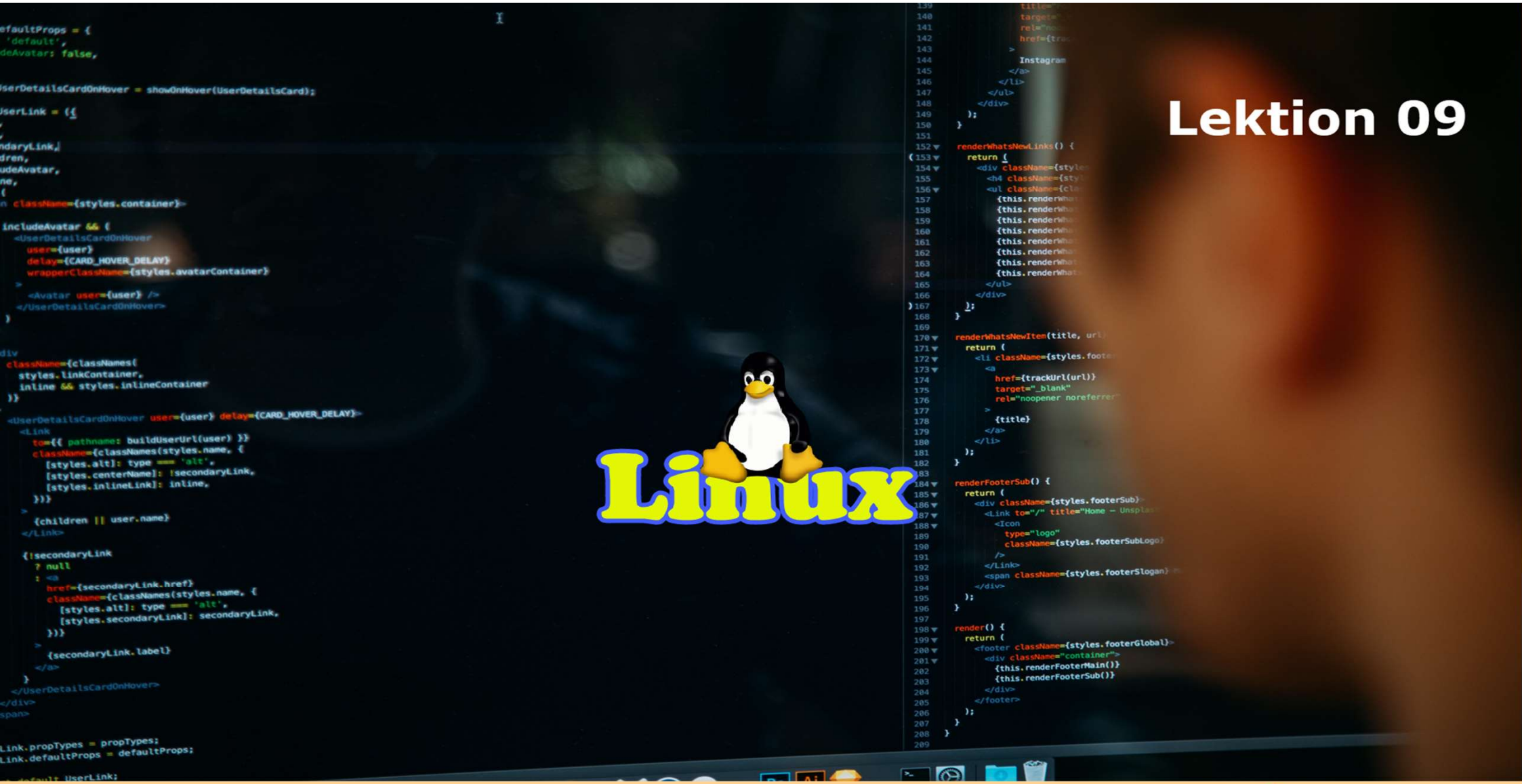

Lektion 09
Linux

# Grundlagen

- Bisher haben wir einige Kommandos kennengelernt und damit eine gute Grundlage für die Arbeit geschaffen
- Viele dieser Kommandos müssen mehrfach täglich für Routinearbeiten genutzt werden
- Und einiges davon lässt sich tatsächlich so gestalten, dass wir uns das Leben leichter machen können
- Zuvor muss man sich allerdings mit der Skriptsprache anfreunden, die uns z.B. von der Bash zur Verfügung gestellt wird

# Die Bash

- Die Bash (Bourne Again Shell) ist eine Arbeitsumgebung, die uns unabhängig von der Distribution zur Verfügung steht
- Mit ihr benutzen wir unsere Kommandos und sind in der Lage versetzt, eigene Skripte zu erstellen
- Die Programmiersprache dahinter ist lange nicht so leistungsfähig wie die bekannten Hochsprachen (zB. C# oder Java), aber sie ist sehr flexibel und genügt, um Alltagsaufgaben zu automatisieren
- In den Skripten geben wir zuerst immer an, für welche Shell das Skript geschrieben ist

# Shebang und Inhalt

- Alle Skripte beginnen mit Shebang. Das ist eine Kombination aus # (sharp aus der Musiknotation) und dem ! (Synonym für Bang) gefolgt vom Pfad zur Bash (/bin/bash), also zum Interpreter, zur Shell. Aus den beiden Zeichen wurde das Wortspiel „Shebang“
- `#!/bin/bash`
- In den folgenden Zeilen steht dann der eigentliche Skriptinhalt, also die abzuarbeitenden Kommandos
- Wie üblich werden diese von oben nach unten ausgeführt

# Erstellungsprozess

- Entsprechend der umask-Regel besitzen neu erstellte Dateien nur Lese- und Schreibrechte für den Besitzer. Das gilt in gleicher Form für ein erstelltes Skript.
- Es soll aber in jedem Fall ausgeführt werden, also muss das Skript im nachhinein noch Ausführrechte erhalten. (LEK) Das geschieht entweder mit
  - `$ chmod u+x <Dateiname>.sh` **oder**
  - `$ chmod 755 <dateiname>.sh (Die Rechte der Gruppe und der Anderen sind hier nebensächlich)`
- Laut Konvention erhält ein Shellskript, wie zu sehen, die Dateiendung .sh
- Danach kann das Shellskript ausgeführt werden, mit
  - `$ ./<Dateiname>.sh`

# Der Vorgang im Beispiel

```
helmut@debian:~$ nano fun.sh
```

```
  GNU nano 7.2                                    fun.sh

#!/bin/bash
echo "Skripte schreiben macht Spaß"
```

```
helmut@debian:~$ nano fun.sh
helmut@debian:~$ chmod u+x fun.sh
helmut@debian:~$ ./fun.sh
Skripte schreiben macht Spaß.
helmut@debian:~$
```

# Kommentare

- Wie üblich sollten die Programmblöcke und Besonderheiten in den Skripten immer gut kommentiert werden
- Kommentare werden in Shell-Skripten mit einem # (Doppelkreuz , Sharp, …) eingeleitet
- Aus Platzgründen verzichte ich in den ersten Skripten auf Kommentare und erläutere die Funktionen auf separaten Folien
- Je komplexer die Skripte werden, umso mehr Kommentare werde ich nutzen, da Sie dann mit den Grundfunktionen vertraut sind

# Vorder- und Hintergrundprozess

- Ohne besondere Angabe laufen die Prozesse im Skript immer im Vordergrund, d.h. sie lassen zur Zeit der Skriptausführung keine anderen Prozesse zu
- Durch Zusatz eines Kaufmannsund (& = Ampersand) können die Skriptprozesse in den Hintergrund verlegt werden, womit es möglich ist, gleichzeitig andere Prozesse auszuführen
- Erstellen wir ein Skript mit dem Namen schlaf.sh und lassen wir ihn im Hintergrund laufen (schlafen)
- Während dieser Zeit lassen wir uns mit ps Informationen über das Skript anzeigen

# Allgemeines zu schlaf.sh

- Der Inhalt bezieht sich immer auf die Eingaben in nano
- Dass ein Skript nach der Erstellung immer ausführbar gemacht werden muss, wird nicht mehr erwähnt
- In schlaf.sh wird das Kommando sleep 120 benutzt, was bedeutet, dass das System 120 Sekunden lang schläft. Da der Prozess im Hintergrund abläuft, sind wir in der Lage in der Zeit andere Kommandos auszuführen, hier um das Skript mit ps zu analysieren

# schlaf.sh

```
helmut@debian:~$ nano schlaf.sh
```

```
  GNU nano 3.2                                          schlaf.sh
```

```
#!/bin/bash
sleep 120
```

```
helmut@debian:~$ ./schlaf.sh &
[1] 3543
helmut@debian:~$ ps -fp 3543
UID          PID  PPID  C STIME TTY          TIME CMD
helmut      3543   855  0 09:27 pts/0    00:00:00 /bin/bash ./schlaf.sh
helmut@debian:~$
```

# Auswertung der schlaf.sh

- Durch den Start im Hintergrund wird die Prozess-ID (PID) angezeigt (bei mir 3543). Diese können wir im Kommando ps durch die Option –p nutzen
- Die wichtigste Information ist die, dass tatsächlich `/bin/bash` als Interpreter benutzt wird
- Durch diese Information lässt sich ableiten, dass im Shebang auch andere Interpreter aufrufbar sind
- Probieren wir es mit Python. Der Interpreter befindet sich in `/usr/bin/python3`
- Den Pfad kann man sich mit `which python3` anzeigen lassen

# hallo.py

```
helmut@debian:~$ nano hallo.py
```


```
helmut@debian:~$ nano hallo.py
helmut@debian:~$ chmod u+x hallo.py
helmut@debian:~$ ./hallo.py
Hallo, hier ist python.
helmut@debian:~$
```

# Auswertung der hallo.py

- Durch die gezeigte Methode lassen sich auch Python-Skripte erstellen.
- Wichtig ist immer die Angabe des Interpreters im Shebang
  - ggf. geben Sie in der Shebang-Zeile /usr/bin/python3 an
- Das soll aber nur ein kleiner Ausflug gewesen sein, denn es geht ja um die Erstellung von Shell-Skripten
- Trotzdem erkennt man, wie variabel das System ist. Variabel ist ein gutes Stichwort.
- Lassen Sie uns sehen, wie man mit Variablen in Shell-Skripten umgeht

# Variablen

- Man kann Variablen in Shell-Skripten benutzen, ohne diese vorher zu deklarieren und einem Typ zuzuordnen
- Variablen sind nichts anderes als Speicherorte, die einen Namen besitzen.
- Sie können sich Variablen als Name-Wert-Paare vorstellen.
  - Verwenden Sie die Syntax `VARIABLEN_NAME="Inhalt"`, um einer Variablen einen Wert zuzuweisen.
  - Verwenden Sie keine Leerzeichen vor oder nach dem Gleichheitszeichen.
  - Bei Variablen wird auch zwischen Groß- und Kleinschreibung unterschieden
  - Variablennamen werden üblicherweise in Großbuchstaben geschrieben und dürfen keine Leerzeichen enthalten. (Großschreibung ist keine Vorschrift, nur ein gutes Unterscheidungsmerkmal)
  - Achten Sie auf sprechende Variablennamen, vor allem in größeren Skripten. Ich habe mir die generelle Großschreibung mit Underscore (_) angewöhnt, da die Umschalttaste gedrückt bleibt

# variable1.sh

```
helmut@debian:~$ nano variable1.sh
```

```
  GNU nano 7.2                                  variable1.sh

#!/bin/bash
MEINE_SHELL="bash"
echo "Ich mag meine $MEINE_SHELL shell."

helmut@debian:~$ chmod u+x variable1.sh
helmut@debian:~$ ./variable1.sh
Ich mag meine bash shell.
helmut@debian:~$
```

# Auswertung der variable1.sh

- Der Variable MEINE_SHELL wird der Wert „bash“ zugewiesen
- Wenn man den Inhalt einer Variable benutzen will (hier anzeigen), so setzt man das $ (Dollarzeichen) direkt vor den Variablenname
- Optional kann man den Variablenname auch in geschweifte Klammern setzen
- Will man dem Inhalt der Variable zusätzliche Daten direkt voranstellen oder folgen lassen, muss der Variablenname in geschweifte Klammern gesetzt werden
- Allgemeine Empfehlung: Schreiben Sie immer geschweifte Klammern (auch wenn es nervt)
- Ich nutze die geschweiften Klammern nur bei ihrer Notwendigkeit

# variable1.sh verbessern

```
helmut@debian:~$ nano variable1.sh
```

```
  GNU nano 7.2                                variable1.sh

#!/bin/bash
MEINE_SHELL="bash"
echo "Ich mag meine ${MEINE_SHELL} shell."
```

```
helmut@debian:~$ chmod u+x variable1.sh
helmut@debian:~$ ./variable1.sh
Ich mag meine bash shell.
helmut@debian:~$
```

# variable2.sh

```
helmut@debian:~$ nano variable2.sh
```

```
  GNU nano 7.2                                variable2.sh

#!/bin/bash
MEIN_ZIEL="Shell-"
echo "1. Ich lerne $MEIN_ZIELSkripte erstellen."
echo "2. Ich lerne ${MEIN_ZIEL}Skripte erstellen."
```

```
helmut@debian:~$ ./variable2.sh
1. Ich lerne  erstellen.
2. Ich lerne Shell-Skripte erstellen.
```

# Verwendung von Kommandos

- Möchte man in einer Variablen ein Kommando benutzen, so setzt man schon bei der Wertzuweisung ein Dollarzeichen vor das Kommando und setzt es selbst in runde Klammern z.B. `PC_NAME=$(hostname)`
- Dafür gibt es auch noch eine veraltete Methode, die das obere Akzentzeichen links neben der Backspace-Taste nutzt `PC_NAME=\`hostname\``. Verwenden Sie diese Art der Zuweisung nicht mehr

# variable3.sh

```
helmut@debian:~$ nano variable3.sh
```

```
  GNU nano 3.2                          variable3.sh

#!/bin/bash
PC_NAME=$(hostname)
PC_IP=$(hostname -I)
echo "Ich sitze am Rechner ${PC_NAME} mit der IP ${PC_IP}."
```

```
helmut@debian:~$ ./variable3.sh
Ich sitze am Rechner debian mit der IP 192.168.2.90 fd00::a00:27ff:fe38:d6cf 200
3:ec:671a:5f00:a00:27ff:fe38:d6cf .
helmut@debian:~$
```

# Bemerkungen zu Kommandos

- Durch die Möglichkeit Kommandos als Variablen zu deklarieren, stehen uns viele Einsatzvarianten zur Verfügung z.B. wie gezeigt, lassen sich schnell Systeminformationen abfragen
- Aber auch interne Abläufe lassen sich so steuern
- Dazu aber später mehr im Workshop (Teil2) ☺

# Gültige Variablennamen

- Nicht jede Buchstaben-, Ziffern- und Zeichenkonstellation ist als Variablenname erlaubt. Variablennamen können trotzdem „sprechend“ sein
- Das ist erlaubt:
  - `ERSTE3BUCHSTABEN=„abc“ (Kombinationen aus Buchstaben und Ziffern)`
  - `ERSTE_DREI_BUCHSTABEN=„abc“ (alles groß, mit Unterstrich getrennt)`
  - `ErsteDreiBuchstaben=„abc“ (die Kamelschreibweise)`
- Das ist nicht erlaubt
  - `3BUCHSTABEN=„abc“  (keine Ziffer als erstes Zeichen)`
  - `Erste-drei-buchstaben=„abc“ (keine Bindestriche)`
  - `erste@Drei@Buchstaben=„abc“  (keine anderen Sonderzeichen)`
- Mit `unset VARIABLE` wird der Inhalt einer Variable gelöscht

# Einige Systemvariablen

- Mit dem Kommando set sieht man im oberen Ausgabebereich alle verfügbaren Systemvariablen. Diese können in den Skripten verwendet werden. Hier ein kleiner Ausschnitt:

| Systemvariable | Beschreibung |
|---|---|
| BASH | Der Ort und der Befehl, mit dem die derzeitige Shell gestartet wird |
| BASH VERSION | Die aktuelle Version der verwendeten Shell |
| COLUMNS | Zeichen in einer Bildschirmzeile |
| EUID | Benutzernummer des Shell-Prozesses |
| HISTFILE | Name der voreingestellten History-Datei; dies ist die Datei, in der die getätigten Kommandos zum Wiederaufruf gespeichert werden |
| HISTSIZE | Die Anzahl der Kommandos, die in der Datei HISTFILE gespeichert werden |
| HOME | Das Heimatverzeichnis des Benutzers |
| HOSTNAME | Der Hostname des Rechners |

# Weitere Systemvariablen

| Systemvariable | Beschreibung |
|---|---|
| IFS | „Internal Field Separators“; die Variable enthält die Zeichen, die die Wörter in einer Kommandozelle trennen, so das Leerzeichen, das Tabulatorzeichen und der Zeilenumbruch **( Return )** |
| LINES | Zeilen auf einer Bildschirmseite |
| LOGNAME | Der Name, mit dem man sich angemeldet hat |
| MAIL | Das Verzeichnis und die Datei, in der E-Mails und sonstige Nachrichten abgelegt werden |
| MAILCHECK | Zeit in Sekunden, nach der in der Datei MAIL automatisch nach neuen Nachrichten geschaut werden soll.<br>Ist 0 eingestellt, wird nach jedem eingegebenen Befehl nachgesehen |
| PATH | **alle** Verzeichnisse werden hier aufgelistet, in denen nach einem Befehlsaufruf nach der jeweiligen Routine gesucht werden soll. Die Verzeichnisse können ergänzt werden, indem man die unterschiedlichen Einträge mit einem Doppelpunkt trennt |
| PS1 | Die Zeichenkette, die am Bildschirmprompt nach dem Einloggen stehen soll |
| PWD | Der Pfadname des aktuellen Verzeichnisses |

# Abfragen(1)

- Skripte sollen die Notwendigkeit ersetzen, dass sich eine Person an einer Tastatur befindet und eine Reihe von Befehlen eingibt.
- Was ist, wenn Sie eine Aufgabe haben, die Sie automatisieren möchten, die aber je nach den Umständen unterschiedliche Aktionen erforderlich machen?
- Derartige Entscheidungen müssen im Skript getroffen werden, wobei wir Bedingungen abfragen und das Skript entsprechend reagieren lassen.

# Abfragen(2)

- Um eine Abfrage zu erstellen, setzen Sie eine Bedingung in eckige Klammern.
  - Die Syntax lautet: [ Bedingung ].
  - Achten Sie darauf, dass nach der öffnenden und vor der schließenden eckigen Klammer ein Leerzeichen steht
- Verschiedene Arten von Bedingungsabfragen sind situationsbedingt möglich.
- Sie können beispielsweise vergleichen,
  - ob Zeichenfolgen gleich sind,
  - ob eine Zahl größer als eine andere ist oder
  - ob eine Datei vorhanden ist.
  - usw...

# Abfragen(3)

- Die Abfrage `[ -e /etc/passwd ]` prüft, ob `/etc/passwd` vorhanden ist.
- Wenn dies der Fall ist, wird true zurückgegeben.
- Wenn die Datei nicht vorhanden ist, wird false zurückgegeben.
- Schauen Sie sich die möglichen Abfrageformen auf den folgenden Folien an

# Zeichenketten-Abfragen (string)

| Abfrage | Reaktion |
|---|---|
| -z string | Wahr (true), wenn die Zeichenkette vorhanden ist |
| -n string | Wahr (true), wenn die Zeichenkette nicht vorhanden ist |
| ! string | Wahr (true), wenn die Zeichenkette nicht vorhanden ist |
| String1 == string2 | Wahr (true), wenn die Zeichenketten identisch (gleich) sind |
| String1 != string2 | Wahr (true), wenn die Zeichenketten nicht identisch (ungleich) sind |

# Datei-Abfragen (file)

| Abfrage | Reaktion |
|---|---|
| -d file | Wahr (true), wenn die Datei ein Verzeichnis ist |
| -e file | Wahr (true), wenn die Datei vorhanden ist (existiert) |
| -f file | Wahr (true), wenn die Datei vorhanden ist und wirklich eine Datei ist |
| -r file | Wahr (true), wenn die Datei vom Benutzer lesbar ist |
| -s file | Wahr (true), wenn die Datei vorhanden und nicht leer ist |
| -w file | Wahr (true), wenn die Datei vom Benutzer schreibbar ist |
| -x file | Wahr (true), wenn die Datei vom Benutzer ausführbar ist |

# Arithmetische Abfragen

| Abfrage | Reaktion |
|---|---|
| arg1 –eq arg2 | Wahr (true), wenn arg1 gleich zu arg2 ist (eq – equal) |
| arg1 –ne arg2 | Wahr (true), wenn arg1 ungleich zu arg2 ist (ne – not equal) |
| arg1 –lt arg2 | Wahr (true), wenn arg1 kleiner als arg2 ist (lt – less than) |
| arg1 –le arg2 | Wahr (true), wenn arg1 kleiner oder gleich arg2 ist (le - less than or equal) |
| arg1 –gt arg2 | Wahr (true), wenn arg1 größer als arg2 ist (gt - greater than) |
| arg1 –ge arg2 | Wahr (true), wenn arg1 größer oder gleich arg2 ist (ge – greater than or equal) |

# Rechen-Operatoren

- Als Operatoren sind u.a. erlaubt:

| Operator | Anwendung |
|---|---|
| /, *, -, + | Grundrechenarten |
| % | Modulo; der Rest einer Division wird als Ganzzahl ausgegeben |
| <, > , <= , >= | Kleiner / größer als / gleich; ist die Behauptung wahr, ansonsten falsch |
| ==, != | Gleich bzw. ungleich; ist die Behauptung wahr, ansonsten falsch |
| &&, \|\| | Logisches UND bzw. logisches ODER; hiermit können Vergleiche hintereinander ausgeführt werden |
| &, \|, ^ | Bitweises anzuwendendes logisches UND, ODER bzw. EXKLUSIV ODER |

# Die if-Anweisung

- Nachdem Sie nun wissen, wie Sie feststellen können, ob eine bestimmte Bedingung erfüllt ist oder nicht, können Sie diese mit der if-Anweisung kombinieren, um Entscheidungen in Ihren Skripten zu treffen.
  - Die if-Anweisung beginnt mit dem Wort "if" und wird dann von einer [Abfrage] gefolgt.
  - Die darauf folgende Zeile enthält das Wort „then".
  - Als nächstes folgt ein Kommando oder eine Reihe von Kommandos, die ausgeführt werden, wenn die getestete Bedingung erfüllt ist.
  - Schließlich endet die if-Anweisung mit fi, also dem rückwärts geschriebenen if ☺.
- Hier ist die allgemeine Syntax.

# Allgemeine if-Syntax und Beispiel

```
if [Abfrage ist wahr (true)]
then
        Kommando1
        Kommando2
        ...
fi
```

```
#!/bin/bash
MEIN_NAME="Helmut"

if [ $MEIN_NAME == "Helmut" ]
then
        echo "Hallo Helmut"
#       echo "Servus $MEIN_NAME"
fi

helmut@debian:~$ ./if-einfach.sh
Hallo Helmut
helmut@debian:~$
```

# Die if-else Anweisung

- Das letzte Beispiel stellt uns nicht gerade vor eine wirkliche Entscheidung, aber es eignet sich gut zur Syntax-Beschreibung
- Mit einem zusätzlichen "else" bekommen wir eine echte Entscheidung entsprechend des Abfrage-Ergebnisses
- Im nächsten Beispiel betrachten wir die allgemeine Syntax von if-else, gefolgt von einem Beispiel
- Doch vorher lernen wir kennen, wie wir Eingaben erstellen und wie diese einer Variable übergeben werden

# Kommandos echo und read

- Die Kommandos `echo` und `read` lassen sich in Skripten gemeinsam einsetzen
- Mit `echo` kann man den erläuternden Text anzeigen lassen und mit `read` wird dann die passende Eingabe vorgenommen
- Damit werden die Skripte flexibel für Eingaben
- Beispiel:
  - `echo –n „Gib deinen Namen ein: “`
  - `read MEIN_NAME`
- -n nach echo bewirkt, dass kein Zeilenumbruch stattfindet
- `$MEIN_NAME` bzw. `${MEIN_NAME}` ist jetzt der Platzhalter für den eingegebenen Namen

# if-else-Syntax und Beispiel

```
if [Abfage ist wahr (true)]
then
        Kommando1
        Kommando2
        ...
else
        Kommando3
        Kommando4
        ...
fi
```

```
#!/bin/bash
echo -n "Gib deinen Namen ein: "
read MEIN_NAME

if [ $MEIN_NAME == "Helmut" ]
then
        echo "Hallo $MEIN_NAME"
else
        echo "Wer bist du? Ach, $MEIN_NAME!"
fi
```

```
helmut@debian:~$ ./if-else.sh
Wer bist du? Ach Susi!
helmut@debian:~$
```

Wie zu sehen ist, wurde der Name „Susi“ eingegeben.

# Mehrfachbedingungen mit elif

- Manchmal genügt eine einfache if-Abfrage nicht
- Für diesen Fall kann man sich der elif-Funktion bedienen.
- Die Bezeichnung "elif" bedeutet so viel wie else if
- Hinter elif wird eine weitere Abfrage formuliert
- Daraus ergibt sich die allgemeine Syntax, die auf der nächsten Folie wieder mit einem Beispiel ergänzt wird

# elif-Syntax und Beispiel

```
if [Abfage ist wahr (true)]
then
        Kommando1
        Kommando2
        ...
elif [Abfrage ist wahr (true)]
then
        Kommando3
        Kommando4
        ...
else
        Kommando5
        Kommando6
        ...
fi
```

```
#!/bin/bash
echo -n "Gib deinen Namen ein: "
read MEIN_NAME

if [ $MEIN_NAME == "Helmut" ]
then
        echo "Hallo $MEIN_NAME"

elif [ $MEIN_NAME == "Heidi" ]
then
        echo "Servus $MEIN_NAME"
else
        echo "Wer bist du? Ach, $MEIN_NAME!"
fi
```

```
helmut@debian:~$ ./elif.sh
Gib deinen Namen ein: Heidi
Servus Heidi
```

# Die for-Schleife

- Wenn Sie eine Aktion für eine Liste von Elementen ausführen möchten, verwenden Sie eine for-Schleife.
- Die erste Zeile einer for-Schleife beginnt mit dem Wort "for", gefolgt von einem Variablennamen, gefolgt vom Wort "in" und einer Liste von Elementen.
- Die nächste Zeile enthält das Wort "do". Platzieren Sie die Anweisungen, die Sie ausführen möchten, in den folgenden Zeilen und
- beenden Sie die for-Schleife mit dem Wort "done" in einer einzelnen Zeile.

# Die for-Schleife

- Zuerst wird der Variable das erste Element in der Liste zugewiesen und der Codeblock ausgeführt.
- Das nächste Element in der Liste wird danach der Variablen zugewiesen und die Befehle werden ausgeführt.
- Dies geschieht für jedes Element in der Liste.
- Hier ist ein einfaches Skript, das zeigt, wie eine for-Schleife funktioniert.

# Die for-Schleife

```
for VARIABLEN_NAME in ARTIKEL_1 ARTIKEL_2 ARTIKEL_N
do
        kommando 1
        kommando 2
        ...
done
```

```
#!/bin/bash
for FORMEN in Kreis Quadrat Rechteck
do
        echo "Formen: $FORMEN"
done
```

```
helmut@debian:~$ ./formen.sh
Formen: Kreis
Formen: Quadrat
Formen: Rechteck
helmut@debian:~$
```

- Welches Ergebnis wird angezeigt, wenn Sie das schließende Anführungszeichen hinter dem Platzhalter $FORMEN entfernen und es dafür mit einem Leerzeichen Abstand, hinter dem Doppelpunkt platzieren?
  echo "Formen: " $FORMEN

# Beispiel: Musiktitel bearbeiten

- Öffnen Sie Ihr Verzeichnis Musik und erstellen Sie mit touch einige Pseudo-Musiktitel. Wichtig ist der Dateipräfix .mp3 z.B. `touch jazz1.mp3` usw.
- Das folgende Skript (im Verzeichnis Musik erstellen) soll alle mp3-Dateien so umbenennen, dass das heutige Datum im Titel erscheint
- Der Dateiname soll datemusic.sh sein

```
helmut@debian:~$ cd Musik/
helmut@debian:~/Musik$ touch jazz1.mp3
helmut@debian:~/Musik$ touch jazz2.mp3
helmut@debian:~/Musik$ touch rock2.mp3
helmut@debian:~/Musik$ touch rock1.mp3
helmut@debian:~/Musik$ touch klassik1.mp3
helmut@debian:~/Musik$ touch klassik2.mp3
helmut@debian:~/Musik$ ls
jazz1.mp3  klassik1.mp3  rock1.mp3
jazz2.mp3  klassik2.mp3  rock2.mp3
```

# Beispiel: Musiktitel bearbeiten

```
helmut@debian:~/Musik$ nano datemusic.sh
helmut@debian:~/Musik$ chmod u+x datemusic.sh
```

```
#!/bin/bash
TITEL=$(ls *mp3)
DATUM=$(date +%F)  # Format JJJJ-MM-TT durch %F

for MUSIC in $TITEL
do
        echo "Umbenennen ${MUSIC} nach ${DATUM}-${MUSIC}"
        mv ${MUSIC} ${DATUM}-${MUSIC}
done
```

```
helmut@debian:~/Musik$ ./datemusic.sh
Umbenennen jazz1.mp3 nach 2021-05-18-jazz1.mp3
Umbenennen jazz2.mp3 nach 2021-05-18-jazz2.mp3
Umbenennen klassik1.mp3 nach 2021-05-18-klassik1.mp3
Umbenennen klassik2.mp3 nach 2021-05-18-klassik2.mp3
Umbenennen rock1.mp3 nach 2021-05-18-rock1.mp3
Umbenennen rock2.mp3 nach 2021-05-18-rock2.mp3
```

```
helmut@debian:~/Musik$ ls *mp3
2021-05-18-jazz1.mp3  2021-05-18-klassik1.mp3  2021-05-18-rock1.mp3
2021-05-18-jazz2.mp3  2021-05-18-klassik2.mp3  2021-05-18-rock2.mp3
```

**Übung:**
**Erstellen Sie auf gleiche Weise Fotos in Ihrem Bilderordner mit dem Dateisuffix .png und geben Sie den Bildern ebenfalls einen Zeitstempel.**
**Dateiname: datepic.sh (im Verzeichnis Bilder)**
**Natürlich geben Sie den entsprechenden Platzhaltern und Variablen Namen, die zu Bildern bzw. Fotos passen.**

Im Skript sind noch einige Schwächen enthalten. Wenn man es nochmal ausführt, wird das Datum auch nochmal hinzugefügt. Überlegen Sie, wie man das vermeiden könnte!

# Die Wochentagsskripte

- Achtung: Ab jetzt werden die Skripte auf deb-s1 erstellt!
- Hier geht es um die Verbesserung der Backup-Strategie.
- Es handelt sich um sehr kleine Skripte, da nur die Variable DATUM mit dem entsprechenden Datumsformat versehen wird, um danach im Dateinamen der Sicherungsdatei eingebunden zu werden.
- Vergessen Sie bitte nicht die einzelnen Dateien ausführbar zu machen.
- Ich habe das differenzielle Sicherungsskript für den Montag erstellt und dieses Skript 3x mit den Namen der restlichen Tage (Di – Do) kopiert.
- Danach habe ich die kopierten Skripte geöffnet und die Inhalte für die jeweiligen Tage angepasst

# Die differentiellen Skripte

```
  GNU nano 3.2                    back-diff-mo.sh

#!/bin/bash
DATUM=$(date +%F)
tar -czf /backup/${DATUM}-home-mo.tgz $(find /home -mtime -3)
```

Differentielle Sicherung vom Montag

```
  GNU nano 3.2                    back-diff-di.sh

#!/bin/bash
DATUM=$(date +%F)
tar -czf /backup/${DATUM}-home-di.tgz $(find /home -mtime -4)
```

Differentielle Sicherung vom Dienstag

```
  GNU nano 3.2                    back-diff-mi.sh

#!/bin/bash
DATUM=$(date +%F)
tar -czf /backup/${DATUM}-home-mi.tgz $(find /home -mtime -5)
```

Differentielle Sicherung vom Mittwoch

```
  GNU nano 3.2                    back-diff-do.sh

#!/bin/bash
DATUM=$(date +%F)
tar -czf /backup/${DATUM}-home-do.tgz $(find /home -mtime -6)
```

Differentielle Sicherung vom Donnerstag

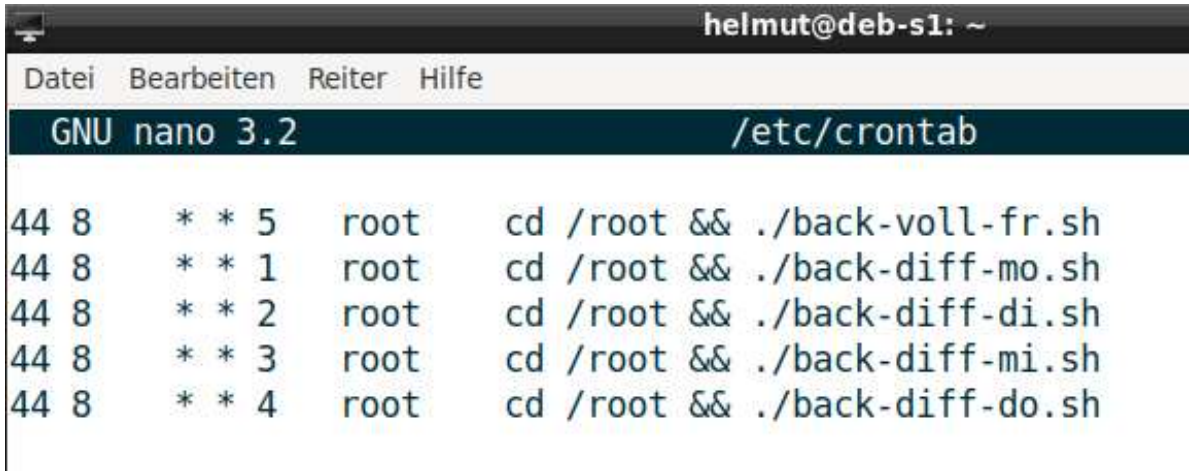
# Die Vollsicherung am Freitag

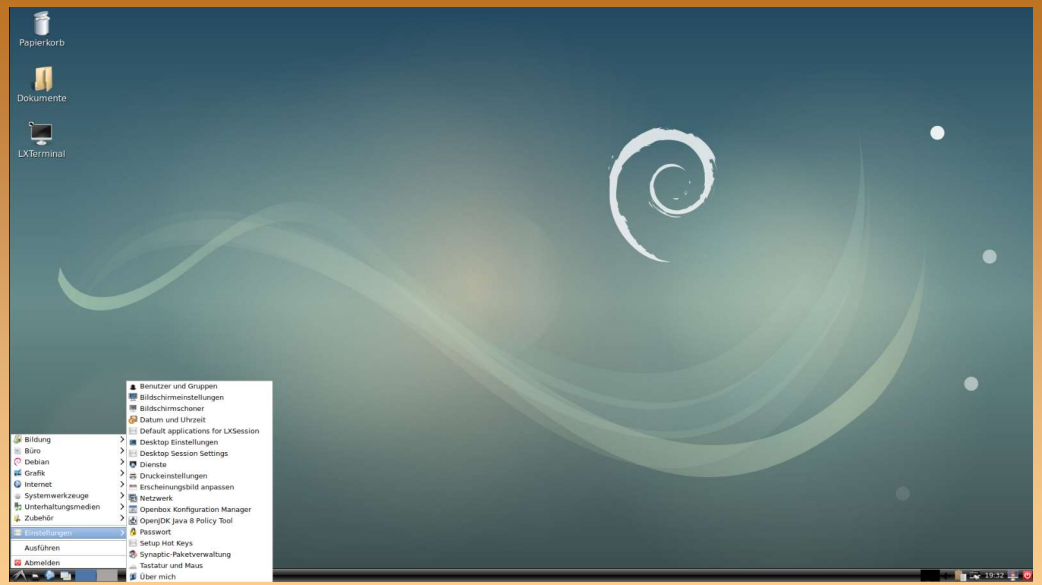
```
  GNU nano 3.2                          back-voll-fr.sh

#!/bin/bash
DATUM=$(date +%F)
tar -czf /backup/${DATUM}-home-voll-fr.tgz /home
```

# Die Crontab-Einträge

- Die Crontab-Einträge erfolgen wie für root gewohnt direkt in der /etc/crontab
- In den Zeilen erfolgt zunächst der Wechsel in den root-Ordner und wenn dies funktioniert hat (logisches UND mit &&), dann wird die jeweilige Sicherungsdatei nach Zeitplan ausgeführt, die sich im root-Ordner befindet.






# Shell Scripting

## Teil 2 (Workshop: Neuen User anlegen)

# Neue Benutzer anlegen

- Anlegen neuer Benutzer für die Abteilungen einkauf und verkauf war bisher ziemlich umständlich
- Das folgende Skript erleichtert dies so weit, dass nur noch der Benutzername und seine Abteilung einzutragen sind
- Den Rest erledigt das Skript alleine
- Das Skript kann nur root erstellen und ausführen, denn nur root darf Benutzer anlegen
- Die Zeilennummerierung bitte nicht eingeben, sie dient nur zur Erläuterung
- Um Passwörter zu generieren, muss das Paket ‚openssl' installiert werden:
- `# apt install openssl`

# User-Skript nu.sh

```
#!/bin/bash
(clear)
echo "Neuen Mitarbeiter anlegen"
for i in {1..25}; do
        echo -n "="
done
echo
echo -n "Name des neuen Mitarbeiters: "
read MA_NAME
echo $MA_NAME
MA_HOME="/home/$MA_NAME"
echo $MA_HOME
echo -n "Arbeit in Abteilung 'einkauf' oder 'verkauf': "
read HAUPTGRUPPE
echo $HAUPTGRUPPE
GRUPPE="kaufleute"
echo $GRUPPE
MA_PASSWD=$(openssl passwd -5 $MA_NAME)
(useradd -m -d $MA_HOME -g $HAUPTGRUPPE -G $GRUPPE -p $MA_PASSWD -s /bin/bash $MA_NAME) && \
echo "Der Mitarbeiter $MA_NAME wurde erstellt."
echo "Beweis:"
(tail -n1 /etc/passwd)
(tail -n1 /etc/shadow)
(tail -n3 /etc/group)
```

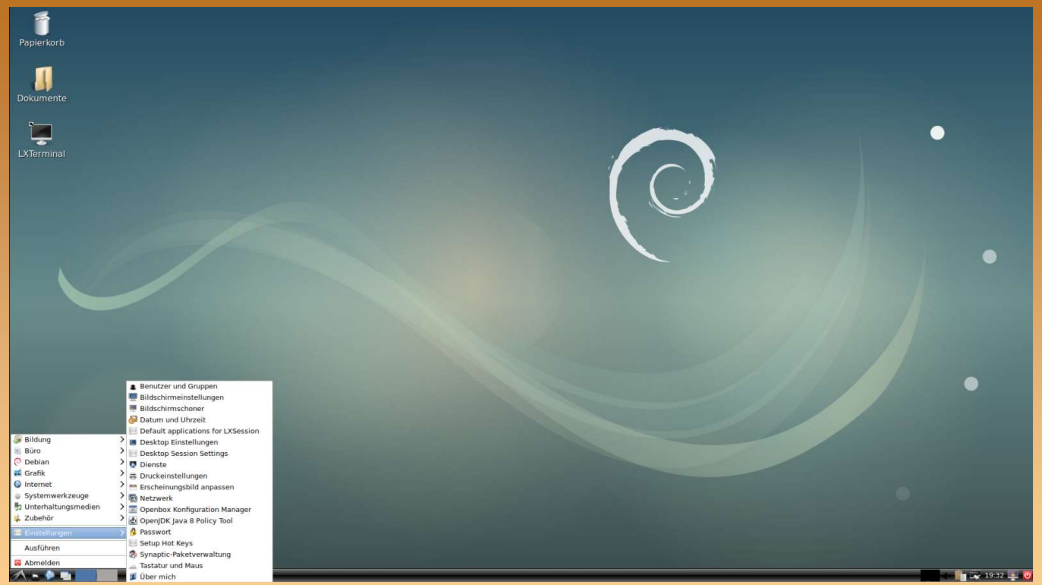
# Skript-Erläuterungen

1. Shebang
2. Bildschirm löschen mit clear
3. Skript-Überschrift
4. Die for-Schleife zählt von 1 bis 25, mit Semikolon ist das nachfolgende do in der gleichen Zeile möglich
5. Mit echo –n wird Gleichheitszeichen ausgegeben (25x)
6. Mittels done wird die for-Schleife beendet
7. Leerzeile mit echo
8. Eingabeaufforderung mit echo
9. Einlesen des Mitarbeiternamens in die Variable MA_NAME mittels read
10. Kontrollausgabe des Inhalts der Variable MA_NAME
11. Das home-Verzeichnis in der Variable MA_HOME anlegen
12. Kontrollausgabe des Platzhalters $MA_HOME
13. Aufforderung zur Eingabe einer der Abteilungen
14. Mit read wird die Variable HAUPTGRUPPE erstellt (einkauf bzw. verkauf sind Hauptgruppen)
15. Kontrollausgabe des Platzhalters $HAUPTGRUPPE

# Skript-Erläuterungen

16. Variable GRUPPE mit Inhalt kaufleute erstellen (feste Zuweisung)
17. Kontrollausgabe des Platzhalters $GRUPPE
18. Mittels openssl passwd wird ein verschlüsseltes Passwort aus dem Benutzernamen erstellt
19. Mit useradd werden alle Platzhalter richtig zusammengefasst und ergänzt. Der Backslash (\) hinter der UND-Verknüpfung (&&) markiert einen Zeilenumbruch. Die Zeile 20 gehört eigentlich zur Zeile 19.
    Der Inhalt der Zeile 20 wird nur ausgegeben, wenn useradd erfolgreich war.
20. Ausgabe, dass der Benutzer angelegt wurde
21. Textausgabe für die darunter folgenden Zeilen
22. Kontrollausgabe: Der neue Nutzer in der Datei /etc/passwd
23. Kontrollausgabe: Das verschlüsselte Passwort in der Datei /etc/shadow
24. Kontrollausgabe: Die Gruppenzuordnung des Benutzers in /etc/group

Die Kontrollausgaben können entweder kommentiert oder gelöscht werden, wenn das Skript funktioniert




# Shell Scripting

## Teil 3 (Nützliche Erweiterungsmöglichkeiten)

# Vorbemerkungen

- Nachdem das Skript zur Erstellung von neuen Benutzern unserer imaginären Firma in den Grundzügen funktioniert, tauchen aber schon die ersten Fragen nach dem Motto auf – Was passiert, wenn, …
- Ein Beispiel wäre: Was passiert, wenn der Abteilungsname falsch eingegeben wird?
- Nach den bisher behandelten Themen sollten Sie auf die Bedingungsabfrage mit If…then…else kommen.
- Aber es gibt auch noch andere Möglichkeiten

# case-Struktur

- Mit dem case-Kommando werden Fallentscheidungen getroffen und zwar in Abhängigkeit vom Wert einer oder mehrerer Variablen.
- Hier die allgemeine Syntax:

```
case $INHALT in

Muster 1) Kommando(s)1;;
Muster 2) Kommando(s)2;;
...
Muster n) Kommando(s)n;;

esac
```

# case-Erläuterungen

- Wenn eine Übereinstimmung zwischen dem Inhalt der Variable (hier: INHALT) und einem Eintrag in einer Musterzeile gefunden wird, dann wird das Kommando oder werden die Kommandos in der Zeile neben dem Muster ausgeführt, bis zu den beiden Semikola am Anweisungsende
- Die Kommandos können sich also auch über mehrere Zeilen erstrecken
- Danach wird der Programmablauf unterhalb von esac ausgeführt

# Ein- und Verkauf

- Für unser Skript aus dem Workshop, könnte man verschiedene Eingabe-Variationen für die beiden Abteilungen als gültig erklären z.B. Einkauf oder E oder einkauf oder e. Das sind die gültigen Muster. Genauso gibt es die gleichen Muster-Variationen für den Verkauf.
- Falscheingaben werden durch das * abgefangen und beenden das Skript.
- Hier der komplette Skriptauszug:

# Skript case.sh

- Die Mustereinträge bis zur schließenden runden Klammer (keine öffnende runde Klammer) findet man alle Einträge, die als gültig angesehen werden, die also in $INHALT vorkommen dürfen
- Das trifft sowohl für den Einkauf, als auch für den Verkauf zu.
- Wird keine Übereinstimmung mit dem Inhalt der Variable $INHALT gefunden, was mit *) abgefangen wird, wird das Skript beendet (exit 0)

```
1  #!/bin/bash
2  (clear)
3  echo -n "Einkauf oder Verkauf: "
4  read INHALT
5
6  case $INHALT in
7  Einkauf|E|einkauf|e) echo "Einkauf gewählt";;
8  Verkauf|V|verkauf|v) echo "Verkauf gewählt";;
9  *) echo "Falscheingabe. Starten Sie neu!"; exit 0;;
10 esac
```

# Das komplette Skript nu1.sh

- Verändern Sie das Skript nu1.sh mit nano in die abschließende Form (siehe Abb.).
- Jetzt ist es Ihnen möglich neue Benutzer der beiden Abteilungen komfortabel mit allen nötigen Einstellungen zu erstellen.
- Testen Sie alle Funktionen ausgiebig, um ggf. Fehler zu erkennen und zu beseitigen.
- Fügen Sie an den markanten Stellen Kommentarzeilen ein, die die Funktionen für Sie verständlich erklären.

```
#!/bin/bash
(clear)
echo "     Neuen Mitarbeiter anlegen"
for i in {1..30}; do
        echo -n "="
done
echo
echo -n "Name des neuen Mitarbeiters: "
read MA_NAME
MA_HOME="/home/$MA_NAME"
echo -n "Abteilungen: 1 - einkauf, 2 - verkauf "
read ABT
case $ABT in
1) HAUPTGRUPPE="einkauf";;
2) HAUPTGRUPPE="verkauf";;
*) echo "Falscheingabe. Das Skript wird beendet. Neustart mit ./nu1.sh"; exit 0;;
esac
GRUPPE="kaufleute"
MA_PASSWD=$(openssl passwd -5 $MA_NAME)
(useradd -m -d $MA_HOME -g $HAUPTGRUPPE -G $GRUPPE -p $MA_PASSWD -s /bin/bash $MA_NAME) && \
(chmod 700 $MA_HOME) && (chage -d 0 -M 90 -W 8 $MA_NAME)
PW=$MA_NAME
(echo $PW; echo $PW) | smbpasswd -s -a $MA_NAME
echo "Der Mitarbeiter $MA_NAME wurde erstellt."
```

# Das fertige Skript nu1.sh bearbeiten

- **Übungsaufgaben:**

- Verändern Sie das Skript, indem Sie jedem neuen Mitarbeiter das generische Linux-Passwort „ChangeMe“ zuweisen.
- In der nächsten Lektion lernen Sie, wie die User ihr vorgegebenes Passwort bei der Ersteinwahl ändern müssen.
- Ergänzen Sie das Skript durch die Kopie des Inhalts vom Ordner /etc/skel in den /home-Ordner jedes neuen Mitarbeiters, wodurch u. a. auch wichtige Firmendokumente kopiert werden können
- Testen Sie das Skript ausführlich und notieren Sie die derzeitigen Schwächen. Überlegen Sie, wie diese Schwächen beseitigt werden können.