Einführung in das Betriebssystem Linux

Markus Näther

19. Dezember 2018

Albert-Ludwigs-Universität Freiburg

1. Recap

2. Heute

3. Dateien und Ordner

4. Graphical User Interface

5. Kommandozeile

6. Bash Scripting

7. Abschluss

Was war

Letzte Vorlesung

- Was ist Linux, wo kommt es her, was wird daraus
- Dateisystem
- Gui vs Konsole

Themen

- Graphical User Interface (GUI)
- Konsole (CLI)
- Bash Skripting für Anfänger

GUI vs. CLI

Warum die CLI?

- GUI macht es einfach Aufgaben auszuführen
- Was ist jedoch mit sich wiederholenden Aufgaben?

$\Rightarrow$ Wenn möglich: Vergleich von GUI und CLI

GUI vs. CLI

Warum die CLI?

- GUI macht es einfach Aufgaben auszuführen
- Was ist jedoch mit sich wiederholenden Aufgaben?

$\Rightarrow$ Wenn möglich: Vergleich von GUI und CLI

Best Practice

Aktiv mitmachen und:

- wenn etwas nicht verstanden wird einfach nachfragen
- keine Angst von der Konsole

... dann klappt's auch mit den Übungsblättern.

Best Practice

Aktiv mitmachen und:

- wenn etwas nicht verstanden wird einfach nachfragen
- keine Angst von der Konsole

... dann klappt's auch mit den Übungsblättern.

Bisher

- Beschränkungen von Dateinamen
- Was ist home, was ist das Elternverzeichnis?
- Verschiedene Typen von Pfaden: absolut und relativ

Zugriffsrechte

Wie erhält man Auskunft über die Zugriffsrechte?

```
ls -l
```

Zugriffsrechte

Beispiel

```
-rw-rw-r--  1 naetherm student    1180 Mär  9 09:21 caffe.cloc
drwxrwxr-x  5 naetherm student    4096 Mär  9 09:22 cmake
-rw-rw-r--  1 naetherm student    2939 Mär  9 09:22 CMakeLists.txt
drwxrwxr-x  5 naetherm student    4096 Mär  9 09:21 data
drwxrwxr-x  4 naetherm student    4096 Mär  9 09:22 Dockerfile
drwxrwxr-x  6 naetherm student    4096 Mär  9 09:22 docs
-rw-rw-r--  1 naetherm student  101863 Mär  9 09:21 .Doxyfile
```

Zugriffsrechte

Was soll -rw-rw-r– bedeuten?

	Benutzer	Gruppe	Andere
-	rw-	rw-	r–
d	rwx	rwx	r-x

Rechte

- -/d beschreibt den Typ
- r steht für Leserechte
- w steht für Schreiberechte
- x steht für Ausführungsrechte

Zugriffsrechte

Was soll -rw-rw-r– bedeuten?

	Benutzer	Gruppe	Andere
-	rw-	rw-	r–
d	rwx	rwx	r-x

Rechte

- **-/d** beschreibt den Typ
- **r** steht für Leserechte
- **w** steht für Schreiberechte
- **x** steht für Ausführungsrechte

Zugriffsrechte

Was soll -rw-rw-r– bedeuten?

	Benutzer	Gruppe	Andere
-	rw-	rw-	r–
d	rwx	rwx	r-x

Rechte

- **-/d** beschreibt den Typ
- **r** steht für Leserechte
- **w** steht für Schreiberechte
- **x** steht für Ausführungsrechte

Zugriffsrechte

Was soll -rw-rw-r– bedeuten?

	Benutzer	Gruppe	Andere
-	rw-	rw-	r–
d	rwx	rwx	r-x

Rechte

- **-/d** beschreibt den Typ
- **r** steht für Leserechte
- **w** steht für Schreiberechte
- **x** steht für Ausführungsrechte

Zugriffsrechte

Was soll -rw-rw-r– bedeuten?

	Benutzer	Gruppe	Andere
-	rw-	rw-	r–
d	rwx	rwx	r-x

Rechte

- **-/d** beschreibt den Typ
- **r** steht für Leserechte
- **w** steht für Schreiberechte
- **x** steht für Ausführungsrechte

Zugriffsrechte

Wofür das alles?

- Linux ist ein Multi-User System!
- Immer auf Zugriffsrechte achten, sonst können Dateien und Ordner „verschwinden“
  drwxrwxrwx .... myBachelorThesis
- Bleibt die Frage: Wie können wir die Berechtigungen ändern?

Zugriffsrechte

Wofür das alles?

- Linux ist ein Multi-User System!
- Immer auf Zugriffsrechte achten, sonst können Dateien und Ordner „verschwinden“
  drwxrwxrwx .... myBachelorThesis
- Bleibt die Frage: Wie können wir die Berechtigungen ändern?

Zugriffsrechte

Wofür das alles?

- Linux ist ein Multi-User System!
- Immer auf Zugriffsrechte achten, sonst können Dateien und Ordner „verschwinden“
  drwxrwxrwx .... myBachelorThesis
- Bleibt die Frage: Wie können wir die Berechtigungen ändern?

Zugriffsrechte ändern

Hierfür können wir einfach das Programm **chmod** verwenden, müssen uns aber andere Kodierungen für rwx merken.

Codierungen

Leserechte	Schreibrechte	Ausführung
r	w	x
4	2	1

Zugriffsrechte ändern

Dateirechte ändern

Wir hatten unseren Ordner „myBachelorThesis“mit den Rechten 777, wir wollen aber das nur wir darin lesen und schreiben können, anderen Studenten sollen unser Arbeit lesen können aber nicht darin schreiben dürfen und alle anderen sollen gar nichts damit machen dürfen:
chmod 740 -R myBachelorThesis

Ohne Kodierung etwas umständlich

chmod g-wx myBachelorThesis; chmod o-rwx myBachelorThesis

Zugriffsrechte ändern

Dateirechte ändern

Wir hatten unseren Ordner „myBachelorThesis“mit den Rechten 777, wir wollen aber das nur wir darin lesen und schreiben können, anderen Studenten sollen unser Arbeit lesen können aber nicht darin schreiben dürfen und alle anderen sollen gar nichts damit machen dürfen:
chmod 740 -R myBachelorThesis

Ohne Kodierung etwas umständlich

chmod g-wx myBachelorThesis; chmod o-rwx myBachelorThesis

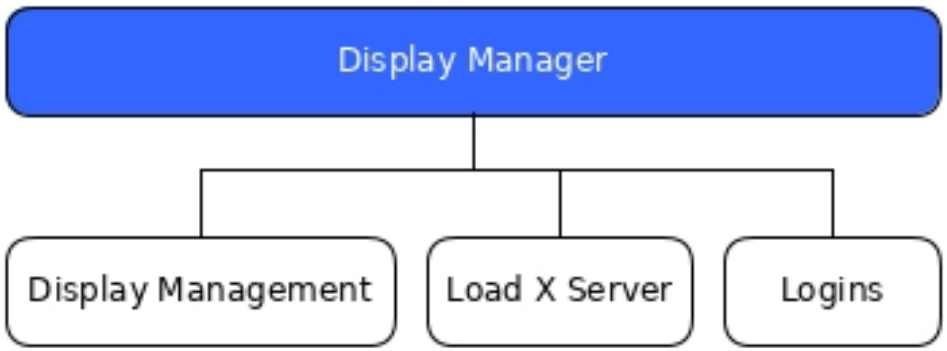
Graphical User Interface

Lernziele

- Umgang mit der GUI
- Anfänglicher Umgang mit dieser
- Die GUI ändern und an eigene Bedürfnisse anpassen

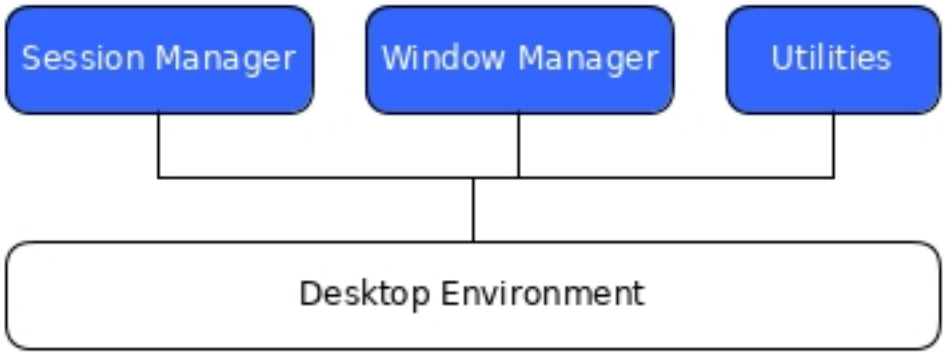
X Window

- X Window wird am Schluss des Bootvorgangs gestartet
- Startet the *Desktopumgebung*
- *Display Manager* kümmert sich um alle *Displays*
- Kümmert sich um Login, Logout, etc.
- Sehr alt: Entwicklungsbeginn in den 1980s
- Wird heutzutage immer mehr durch *Wayland* ersetzt



Desktopumgebung

- Besteht aus *Session Manager*, *Window Manager* und *Utilities*
- *Session Manager*: Startet und verwaltet alles für die aktuelle Session
- *Window Manager*: Verwaltet Platzierung, Bewegung, etc von Fenstern und Widgets
- *Utilities*: Zusätzliche Software



Desktopumgebung

- Ein einzelnes Programm, sondern ein Bundle verschiedener Programme, wir verwenden *GNOME*
- Beschreibt das Aussehen, und *Loook'n'Feel* des Desktops
- Gibt noch viele andere wie KDE, XFCE, LXDE, etc.

Desktophintergrund

- Erste Personalisierung: Der Desktophintergrund
- Nicht das wichtigste, wir verwenden aber schon mal die GUI
- Können ein vordefiniertes Bild verwenden, ein eigenes, oder einfach eine Farbe

Systemkonfigurationen

gnome-tweak-tools

- Konfigurationen für gewöhnlich rechts-oben als Zahnrad o.ä. Symbol auffindbar oder Kontextmenu
- Auswahl jedoch nicht so groß (wie es früher einma war)
- **gnome-tweak-tools** schafft hier Abhilfe

Systemeinstellungen

Lernziele

- Einstellungen, wie Diplay und Datum/Uhrzeit über die GUI verwalten
- Netzwerkeinstellungen vornehmen und verwalten (nächstes mal)
- Software installieren und updaten

Standardsoftware

- Internet
- Produktivität und Entwicklung
- Multimedia

Standardsoftware

Internet

- Browser
- EMails
- Anderes

Standardsoftware

Produktivität und Entwicklung

- Office-Anwendungen
- Entwicklungsumgebungen

Standardsoftware

Multimedia

- Musikplayer
- Videoplayer
- Videoeditoren
- Grafikeditoren

Warum eigentlich die Konsole?

- Die GUI ist zwar schön und nett, aber
- die Konsole ist sehr mächtig
- sehr gut für sich wiederholende Aufgaben.
- *GUIs machen einfache Aufgaben einfacher, CLIs machen schwere Aufgaben möglich.*

Warum eigentlich die Konsole?

- Die GUI ist zwar schön und nett, aber
- die Konsole ist sehr mächtig
- sehr gut für sich wiederholende Aufgaben.
- *GUIs machen einfache Aufgaben einfacher, CLIs machen schwere Aufgaben möglich.*

Warum eigentlich die Konsole?

- Die GUI ist zwar schön und nett, aber
- die Konsole ist sehr mächtig
- sehr gut für sich wiederholende Aufgaben.
- *GUIs machen einfache Aufgaben einfacher, CLIs machen schwere Aufgaben möglich.*

Warum eigentlich die Konsole?

- Die GUI ist zwar schön und nett, aber
- die Konsole ist sehr mächtig
- sehr gut für sich wiederholende Aufgaben.
- *GUIs machen einfache Aufgaben einfacher, CLIs machen schwere Aufgaben möglich.*

Kommandozeile

Demo

- Programm braucht Bilder als jpg, habe aber nur png
- PDFs kombinieren

Kommandozeile

Demo

- Programm braucht Bilder als jpg, habe aber nur png
- PDFs kombinieren

Kommandozeile

Lernziele

- Verwendung der Konsole
- Alles mögliche im Dateisystem machen
- Software installieren und updaten

Kommandozeile

CLI öffnen

- Aktivität
- ⇒ Suche: *Terminal*
- Enter

Kommandozeile

Einfache Programme

cd	Den aktuellen Pfad wechseln
pwd	Den aktuellen Pfad anzeigen
ls	Dateien und Ordner in Pfad anzeigen

Mit Dateien arbeiten

Viele Commandos um mit Dateien zu arbeiten:

- Erstellen und Löschen von Dateien
- Inhalt anschauen
- Verschieben, Umbennen, etc von Dateien

Dateien erstellen

touch	Erstellt leere Datei

Inhalt betrachten

Command	Beschreibung
cat	Ganzen Inhalt einer Datei ausgeben
tac	Wie *cat* nur andersrum
less	Gibt den Inhalt einer Datei Stück für Stück aus
head	Zeit die ersten x Zeilen einer Datei
tail	Zeit die letzten x Zeilen einer Datei

Ordner erstellen

mkdir	Erstellt leeren Ordner

Kommandozeile

Bewegen, Umbenennen, etc

mv	Verschiebt Datei oder Ordner
cp	Kopiert Datei oder Ordner
rm	Löscht Datei oder Ordner
rmdir	Löscht Ordner

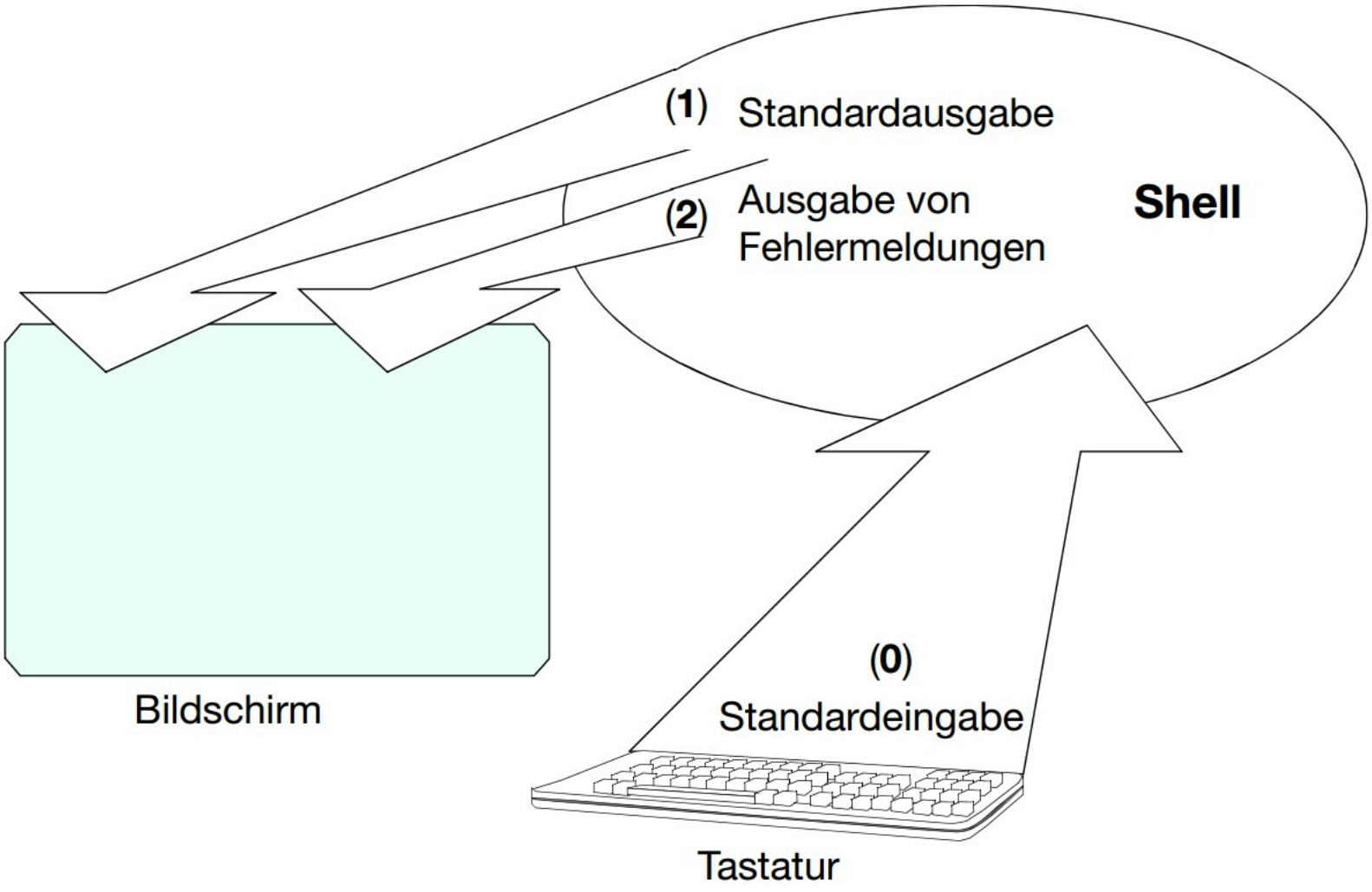
Suchen und mit [Out,In]put arbeiten

- Output speichern, Input verwenden
- Pipes (Einführung)
- Wildcards und RegEx

Standardein- und Ausgabe

Ein- und Ausgabe sind nur **Text**.
⇒ Tastatur und Maus werden durch (temporäre) *Textdateien* ersetzt

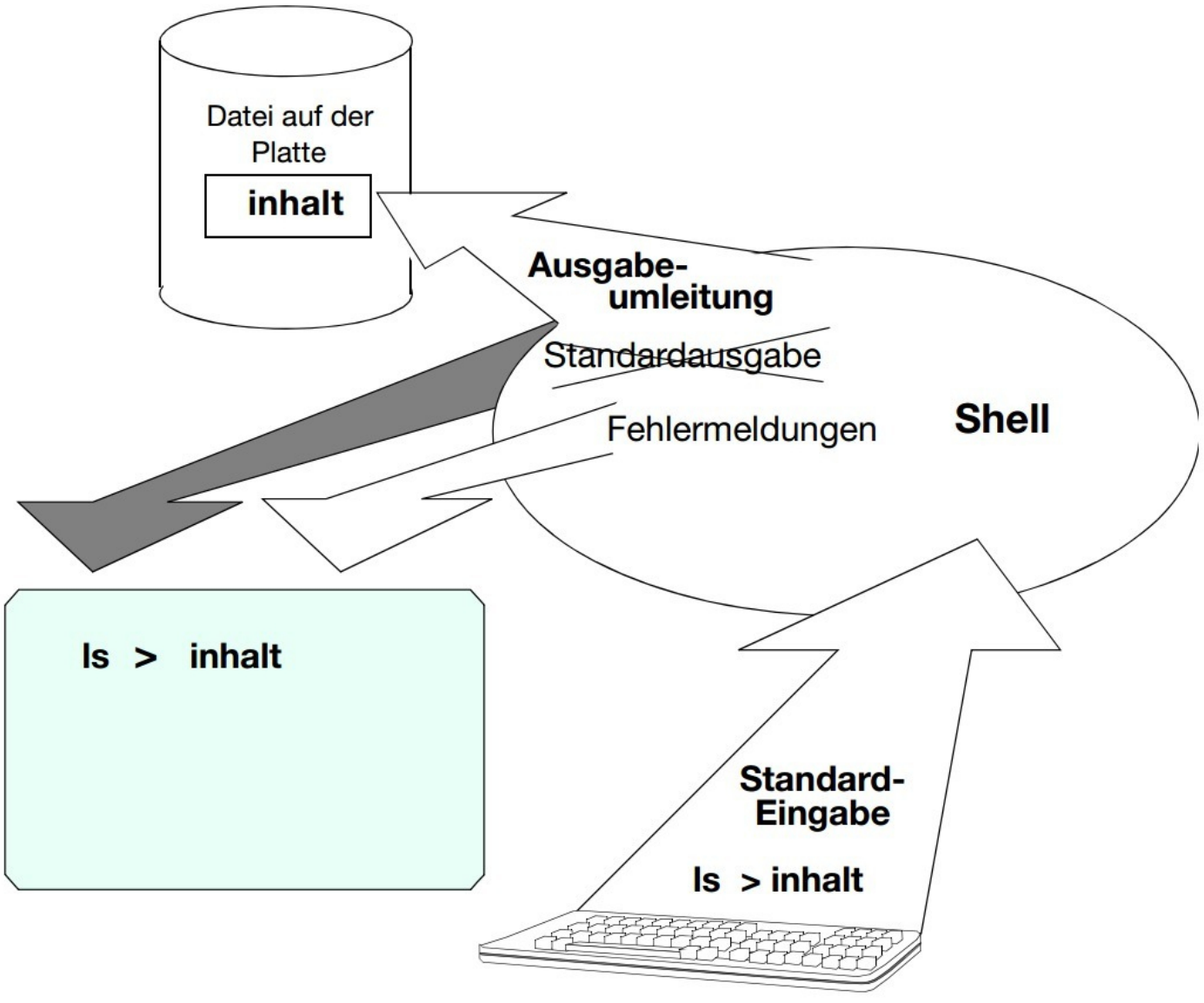
Standardein- und Ausgabe



Umleiten der Standardausgabe

Was soll gemacht werden?		*Wohin?*
Kommando	>	Ausgabe

Geänderte Standardein- und Ausgabe



Beispiel

Liste direkt an Drucker schicken

```
ls > /dev/lp
```

Warnung

Das ist nicht zu empfehlen! Lieber einen Spooler dafür verwenden (mehr dazu später). Für jetzt einfach: Don’t do it!

Beispiel

Liste direkt an Drucker schicken

```
ls > /dev/lp
```

Warnung

Das ist nicht zu empfehlen! Lieber einen Spooler dafür verwenden (mehr dazu später). Für jetzt einfach: Don’t do it!

Geänderte Standardein- und Ausgabe

Umleiten der Standardausgabe

Was soll gemacht werden?		*Wohin?*
Kommando	>	Ausgabe

Wichtig

Hierdurch wird „Ausgabe “immer neu erstellt. Alte Inhalte der Datei gehen somit verloren!

Geänderte Standardein- und Ausgabe

Umleiten der Standardausgabe

Was soll gemacht werden?		*Wohin?*
Kommando	>	Ausgabe

Wichtig

Hierdurch wird „Ausgabe “immer neu erstellt. Alte Inhalte der Datei gehen somit verloren!

Geänderte Standardein- und Ausgabe

Umleiten der Standardausgabe

Was soll gemacht werden?		*Wohin?*
Kommando	>>	Ausgabe

Wichtig

Hierdurch wird der Inhalt aus dem Kommando der Datei „Ausgabe“ **angehängt**. Alte Informationen bleiben somit erhalten.

Geänderte Standardein- und Ausgabe

Umleiten der Standardausgabe

Was soll gemacht werden?		*Wohin?*
Kommando	>>	Ausgabe

Wichtig

Hierdurch wird der Inhalt aus dem Kommando der Datei „Ausgabe“ **angehängt**. Alte Informationen bleiben somit erhalten.

Geänderte Standardeingabe

Recht intuitiv

Was soll gemacht werden?		*Womit?*
Kommando	<	Eingabe

Stadardeingabe - Stanardausgabe

Beispiel 1

Wie kann man die Ausgabe von `ls` in eine Datei `ausgabe.txt` umleiten?

Pipe-Mechnismus

Was ist Pipe

Pipe ist ein sehr effizientes Verfahren verschiedene Kommandos zu verschachteln.

Moment

Hatten wir das nicht schon mal und das geht einfach durch ein Semikolon?

Pipe > ;

Ja, einfache Konkatenationen hatten wir, aber der Vorteil von Pipes ist das wir Ausgaben an Programme weiterleiten können!

Pipe-Mechnismus

Was ist Pipe

Pipe ist ein sehr effizientes Verfahren verschiedene Kommandos zu verschachteln.

Moment

Hatten wir das nicht schon mal und das geht einfach durch ein Semikolon?

Pipe > ;

Ja, einfache Konkatenationen hatten wir, aber der Vorteil von Pipes ist das wir Ausgaben an Programme weiterleiten können!

Pipe-Mechnismus

Was ist Pipe

Pipe ist ein sehr effizientes Verfahren verschiedene Kommandos zu verschachteln.

Moment

Hatten wir das nicht schon mal und das geht einfach durch ein Semikolon?

Pipe > ;

Ja, einfache Konkatenationen hatten wir, aber der Vorteil von Pipes ist das wir Ausgaben an Programme weiterleiten können!

Pipe-Mechanismus

Beispiel: Pipe

Wie viele Dateien und Ordner gibt es im aktuellen Verzeichnis:

```
ls -l | wc -l
```

- **ls -l**: Gibt den Inhalt zeilenweise aus
- **wc -l**: `man` befragen

Ohne Pipe

```
ls -l > inhalt
wc -l < inhalt
rm inhalt
```

Pipe-Mechanismus

Beispiel: Pipe

Wie viele Dateien und Ordner gibt es im aktuellen Verzeichnis:

```
ls -l | wc -l
```

- **ls -l**: Gibt den Inhalt zeilenweise aus
- **wc -l**: `man` befragen

Ohne Pipe

```
ls -l > inhalt
wc -l < inhalt
rm inhalt
```

Pipe-Mechanismus

Beispiel: Pipe

Wie viele Dateien und Ordner gibt es im aktuellen Verzeichnis:

```
ls -l | wc -l
```

- **ls -l**: Gibt den Inhalt zeilenweise aus
- **wc -l**: `man` befragen

Ohne Pipe

```
ls -l > inhalt
wc -l < inhalt
rm inhalt
```

Pipe-Mechanismus

Beispiel: Pipe

Wie viele Dateien und Ordner gibt es im aktuellen Verzeichnis:

```
ls -l | wc -l
```

- **ls -l**: Gibt den Inhalt zeilenweise aus
- **wc -l**: `man` befragen

Ohne Pipe

```
ls -l > inhalt
wc -l < inhalt
rm inhalt
```

Pipe-Mechanismus

Eigenschaften

- Verbindung von zwei Kommandos über *temporären Buffer*
- *Sofortige Weiterleitung* des Buffers an das zweite Kommando
- Schnelle Verarbeitung aufgrund der internen Buffer
- Es können beliebig viele Pipes aneinandergehängt werden

Pipe-Mechanismus

Eigenschaften

- Verbindung von zwei Kommandos über *temporären Buffer*
- *Sofortige Weiterleitung* des Buffers an das zweite Kommando
- Schnelle Verarbeitung aufgrund der internen Buffer
- Es können beliebig viele Pipes aneinandergehängt werden

Pipe-Mechanismus

Eigenschaften

- Verbindung von zwei Kommandos über *temporären Buffer*
- *Sofortige Weiterleitung* des Buffers an das zweite Kommando
- Schnelle Verarbeitung aufgrund der internen Buffer
- Es können beliebig viele Pipes aneinandergehängt werden

Pipe-Mechanismus

Eigenschaften

- Verbindung von zwei Kommandos über *temporären Buffer*
- *Sofortige Weiterleitung* des Buffers an das zweite Kommando
- Schnelle Verarbeitung aufgrund der internen Buffer
- Es können beliebig viele Pipes aneinandergehängt werden

Wildcards

Wiederholung

- Wofür steht `k*.txt`?
- Wofür steht `k?.txt`?

Weitere Wildcards

- **[...]** Die Klammer wird durch **ein** Zeichen aus der Klammer ersetzt
- **[!...]** Die Klammer wird durch **ein** Zeichen das **nicht** aus der Klammer kommt ersetzt
- \ Das „Fluchtsymbol“ hebt das Ersetzungsmuster für das nachfolgende Metazeichen auf (etwa *)
  $\Rightarrow$ Wir wollen also z.B. wirklich nach $ suchen

Wildcards

Wiederholung

- Wofür steht `k*.txt`?
- Wofür steht `k?.txt`?

Weitere Wildcards

- **[...]** Die Klammer wird durch **ein** Zeichen aus der Klammer ersetzt
- **[!...]** Die Klammer wird durch **ein** Zeichen das **nicht** aus der Klammer kommt ersetzt
- \ Das „Fluchtsymbol“ hebt das Ersetzungsmuster für das nachfolgende Metazeichen auf (etwa *)
  $\Rightarrow$ Wir wollen also z.B. wirklich nach $ suchen

Wildcards

Wiederholung

- Wofür steht `k*.txt`?
- Wofür steht `k?.txt`?

Weitere Wildcards

- **[...]** Die Klammer wird durch **ein** Zeichen aus der Klammer ersetzt
- **[!...]** Die Klammer wird durch **ein** Zeichen das **nicht** aus der Klammer kommt ersetzt
- \ Das „Fluchtsymbol“ hebt das Ersetzungsmuster für das nachfolgende Metazeichen auf (etwa *)
  $\Rightarrow$ Wir wollen also z.B. wirklich nach $ suchen

Wildcards

Wiederholung

- Wofür steht `k*.txt`?
- Wofür steht `k?.txt`?

Weitere Wildcards

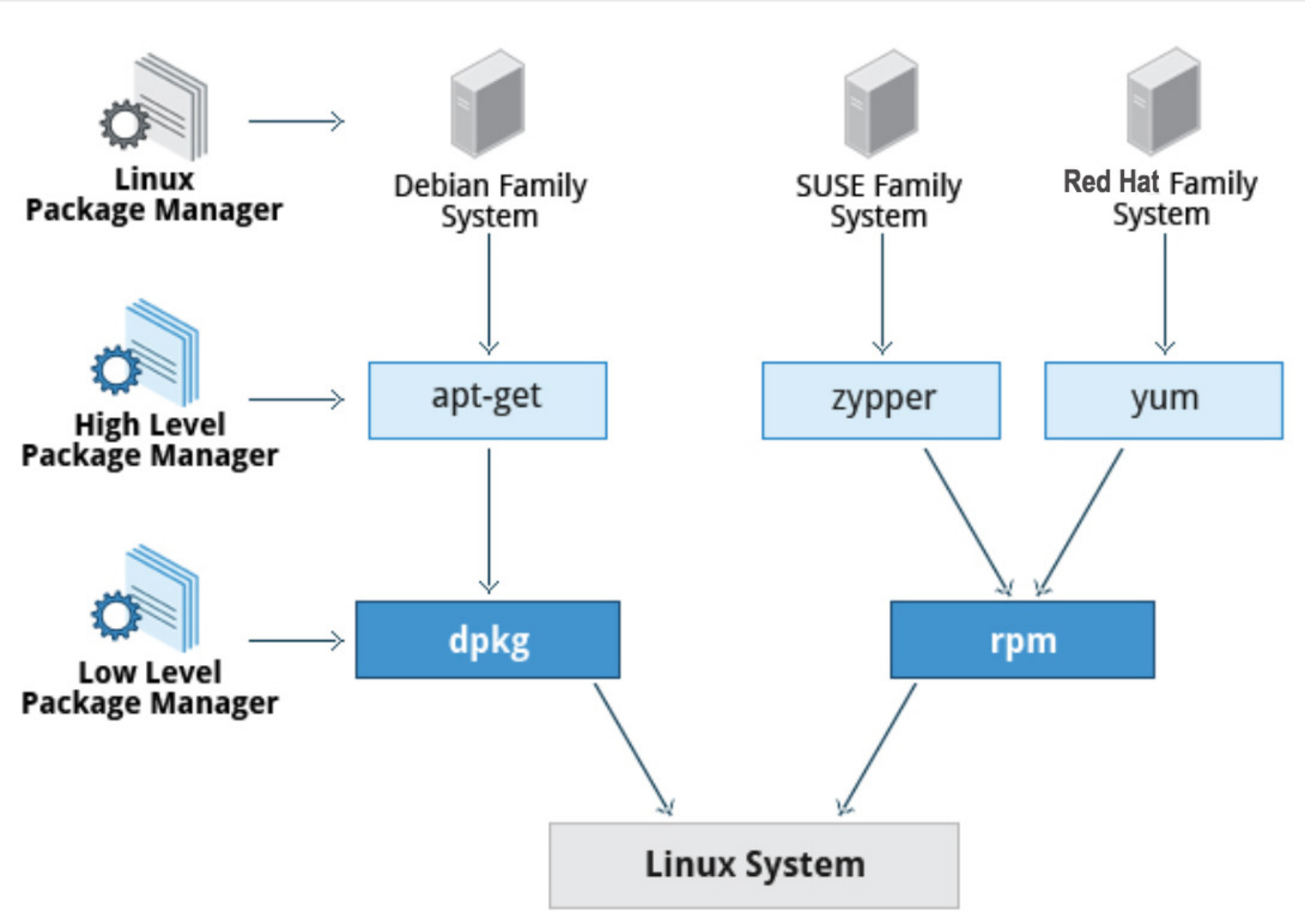
- **[...]** Die Klammer wird durch **ein** Zeichen aus der Klammer ersetzt
- **[!...]** Die Klammer wird durch **ein** Zeichen das **nicht** aus der Klammer kommt ersetzt
- \ Das „Fluchtsymbol“ hebt das Ersetzungsmuster für das nachfolgende Metazeichen auf (etwa *)
  ⇒ Wir wollen also z.B. wirklich nach $ suchen

Nach Dateien suchen

Alle Logs unter /var/log/ finden: `find /var/log -name "*.log"`
Nur bestimmten Typ suchen: `find /var/log -type d -name "*.log"`

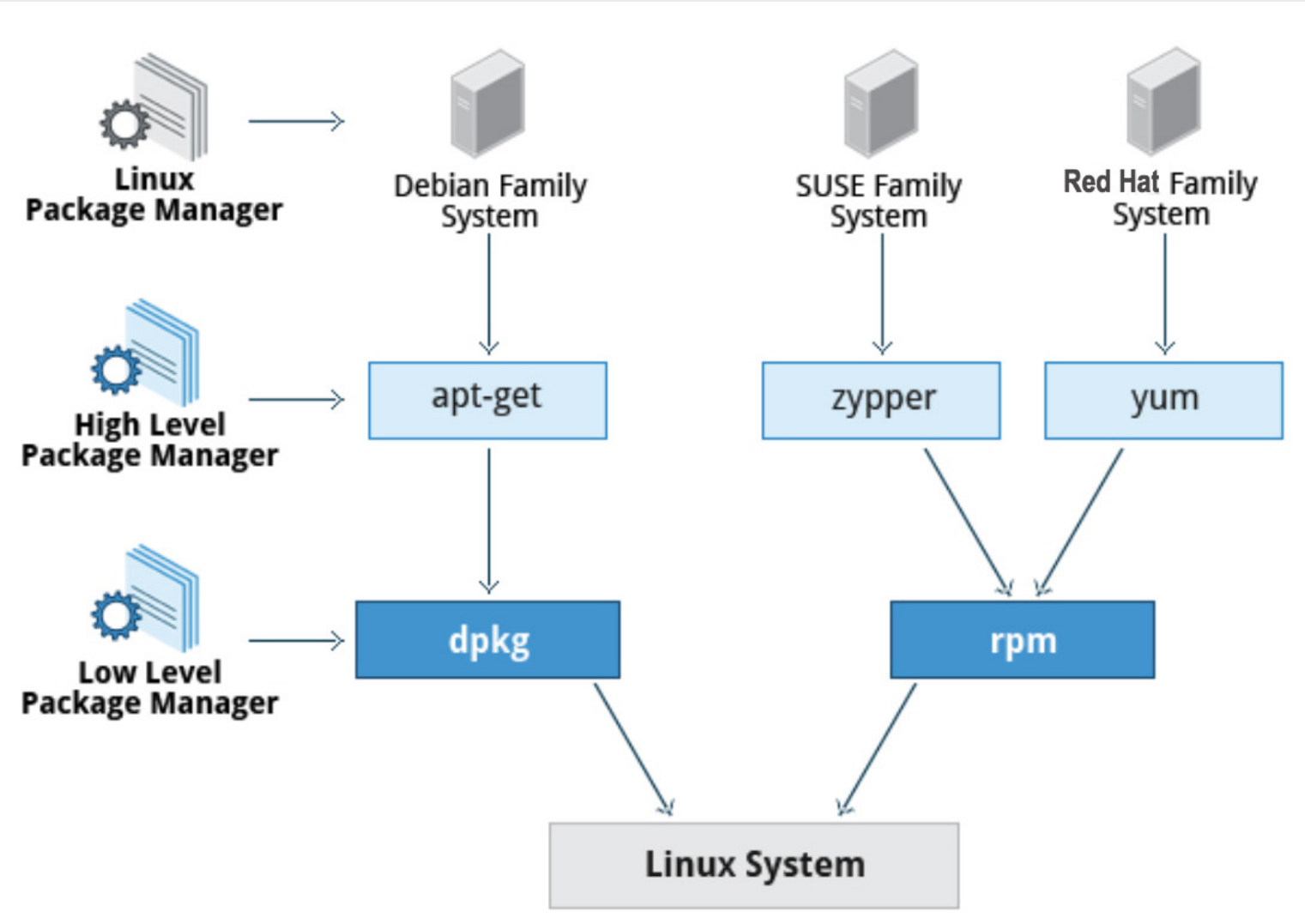
Software installieren

- Zwei große Paketsysteme: **DEB** und **RPM**
- Beide haben High- und Low-Level Paketmanager



Software installieren

- Zwei große Paketsysteme: **DEB** und **RPM**
- Beide haben High- und Low-Level Paketmanager



Kommandozeile

apt

- `sudo apt install <PACKAGENAME>`
- `sudo apt remove <PACKAGENAME>`

yum

- `sudo yum install <PACKAGENAME>`
- `sudo yum remove <PACKAGENAME>`

Wo findet man all das?

- Linux hat große Dokumentation
- `man` wer mit der Konsole auskommt
- `yelp` ist eine Gui-Oberfläche für die gleiche Dokumentation

Bash Scripting

Lernziele

- Features und Funktionumfang von Bash Skripten
- Syntax von Skripten
- Umgang mit Methoden und Konstrukten
- Tests für Eigenschaften (von Dateien) und anderen Objekten
- Programmfluss manipulieren

Features und Funktionumfang

- Suche eine Datei um dann etwas damit zu machen
- Wenn man es nur bei einer Datei machen muss: kein Problem
- Was aber wenn man 10.000 mal gemacht werden muss?
- Was wenn die Datei mehrere GB groß ist?

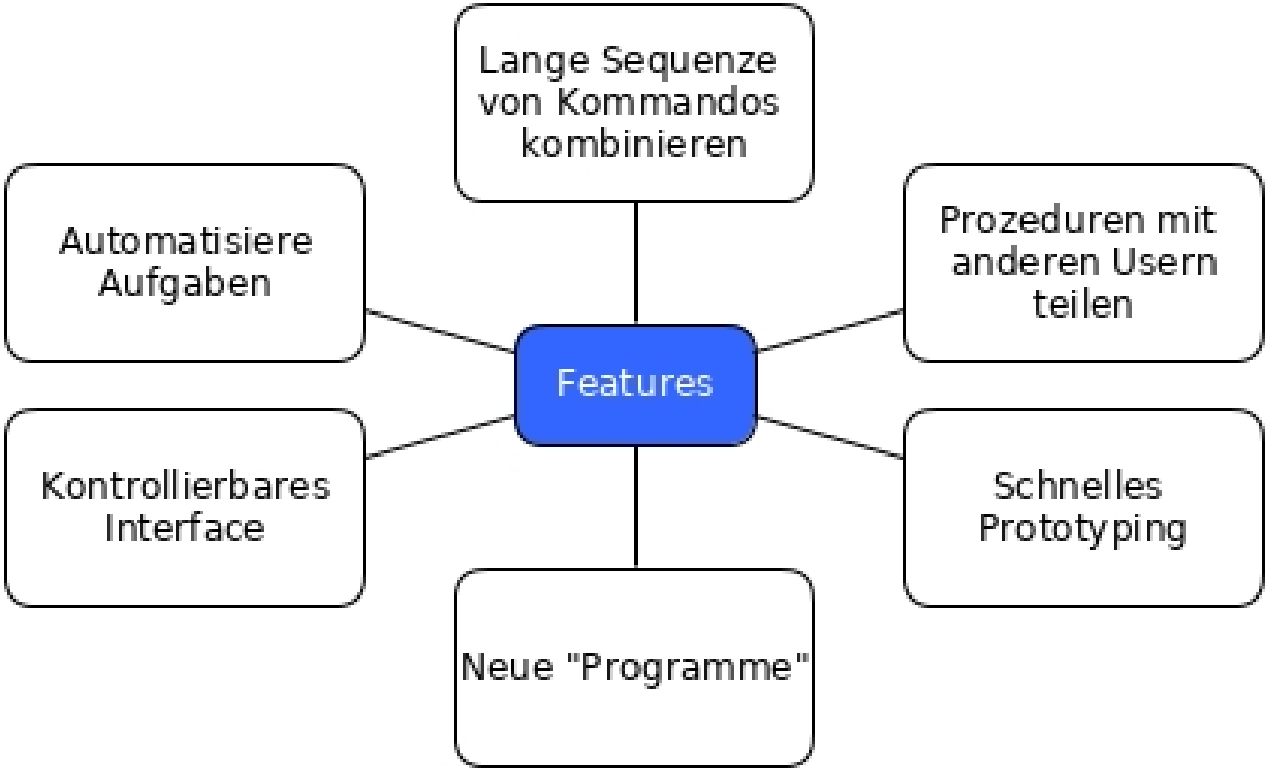
$\Rightarrow$ Automatisierung mit Bash Skripten

Features und Funktionumfang

- Suche eine Datei um dann etwas damit zu machen
- Wenn man es nur bei einer Datei machen muss: kein Problem
- Was aber wenn man 10.000 mal gemacht werden muss?
- Was wenn die Datei mehrere GB groß ist?

$\Rightarrow$ Automatisierung mit Bash Skripten

Bash Scripting



Beispiele

- Dateien komprimieren und automatisch als Backup in eine Cloud hochladen
- Aktuelles Wetter anzeigen lassen
- Aktuelle Aktienkurse laden, parsen, und automatisch benachrichten wenn bestimmte Kriterien zustimmen

Beispiele

- Dateien komprimieren und automatisch als Backup in eine Cloud hochladen
- Aktuelles Wetter anzeigen lassen
- Aktuelle Aktienkurse laden, parsen, und automatisch benachrichten wenn bestimmte Kriterien zustimmen

Beispiele

- Dateien komprimieren und automatisch als Backup in eine Cloud hochladen
- Aktuelles Wetter anzeigen lassen
- Aktuelle Aktienkurse laden, parsen, und automatisch benachrichten wenn bestimmte Kriterien zustimmen

Konsole != Konsole

- Wie immer: Eine einzige Konsole ist langweilig
- Mehrere mit unterschiedlichen Vor- und Nachteilen
- Beispiele: **sh**, **bash**, **csh**, **zsh**

⇒ Die Standardshell reicht für gewöhnlich, später kann auch spezialisierte Shell verwendet werden.

Konsole != Konsole

- Wie immer: Eine einzige Konsole ist langweilig
- Mehrere mit unterschiedlichen Vor- und Nachteilen
- Beispiele: **sh**, **bash**, **csh**, **zsh**

⇒ Die Standardshell reicht für gewöhnlich, später kann auch spezialisierte Shell verwendet werden.

Erstes Skript: hello_world.sh

```
#!/bin/bash
echo "Hello Linux Students"
```

Syntax

Command	Beschreibung
#	Kommentar einfügen
\	Zeilenumbruch um lange Commandos aufzuteilen
;	Alles weitere als neues Commando interpretieren
$	Auf Variable zugreifen
>	Redirect Output
>>	Append Output
<	Redirect Input
|	Pipe

Und noch ein paar weitere wie (...), ..., [...], &&, ||, ’, ", $((...)), zu ein paar später mehr

Parameter übergeben

Parameter

```
#!/bin/bash
echo "Erstes : $1"
echo "Zweites: $2"
echo "Anzahl : $#"
echo "Alle : $*"
```

Zusätzliche Parameter

```
#!/bin/bash
echo "Programmname: $0"
echo "Exit Status: $?"
echo "Prozessnummer der Shellprozedur: $$"
```

Parameter übergeben

Parameter

```
#!/bin/bash
echo "Erstes : $1"
echo "Zweites: $2"
echo "Anzahl : $#"
echo "Alle : $*"
```

Zusätzliche Parameter

```
#!/bin/bash
echo "Programmname: $0"
echo "Exit Status: $?"
echo "Prozessnummer der Shellprozedur: $$"
```

Parameter übergeben

Direkt über die Konsole

```
head -2 data/planeten.txt; tail -n +3 data/planeten.txt | sort
```

Als Parameter planet.sh

```
#!/bin/bash
head -2 $1; tail -n +3 $1 | sort
```

Parameter übergeben

Direkt über die Konsole

```
head -2 data/planeten.txt; tail -n +3 data/planeten.txt | sort
```

Als Parameter planet.sh

```
#!/bin/bash
head -2 $1; tail -n +3 $1 | sort
```

Aufruf über Konsole

```
./planet.sh data/planeten.txt
```

Beim Aufruf

```
head -2 $1; tail -n +3 $1 | sort
```

```
head -2 data/planeten.txt; tail -n +3 data/planeten.txt | sort
```

Aufruf über Konsole

```
./planet.sh data/planeten.txt
```

Beim Aufruf

```
head -2 $1; tail -n +3 $1 | sort
```

```
head -2 data/planeten.txt; tail -n +3 data/planeten.txt | sort
```

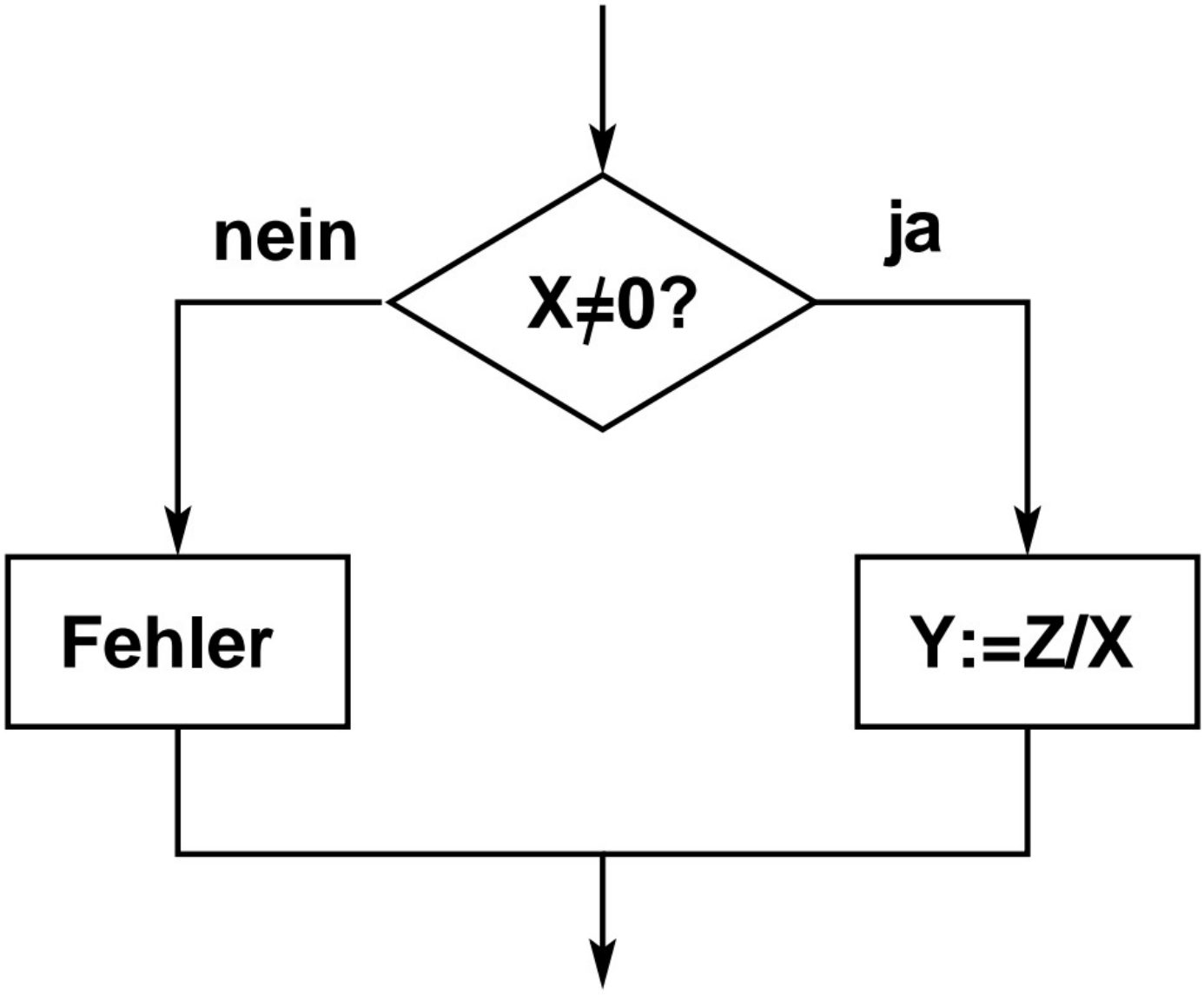
Aufruf über Konsole

```
./planet.sh data/planeten.txt
```

Beim Aufruf

```
head -2 $1; tail -n +3 $1 | sort
```

```
head -2 data/planeten.txt; tail -n +3 data/planeten.txt | sort
```


nein
X≠0?
ja
Fehler
Y:=Z/X

If Verzweigungen

Bedingte Ausführung: if ... then ... else ...

Beispiel eines If-Else-Blocks

```
if test $1 == „eins“
then
        echo „$1 ist gleich eins“
else
        echo „$1 ist ungleich eins“
fi
```

Anzahl der Parameter abfragen

```
if test $# == 1
then
        echo „Genau ein Parameter“
else
        echo „Weniger oder mehr Parameter“
fi
```

Vorhanden sein einer Datei abfragen

```
if test -r $1
then
        echo „Die Datei $1 existiert“
else
        echo „Die Datei $1 existiert nicht“
fi
```

Vorhanden sein einer Datei abfragen

```
if [ ! -r $1 ]
then
        echo „Die Datei $1 existiert nicht“
        exit 1
fi
        echo „Weiter ... “
```

Was bedeutet [! -r $1]?

test kann auch verkürzt werden. Die Testbedingung muss dann in eckige Klammern geschrieben werden.
Also statt `if test ! -r $1` schreiben wir einfach `if [ ! -r \$1 ]`.
Außerdem wichtig: Niemals die Leerzeichen nach `[` und vor `]` vergessen!

Vorhanden sein einer Datei abfragen

```
if [ ! -r $1 ]
then
        echo „Die Datei $1 existiert nicht“
        exit 1
fi
        echo „Weiter ... “
```

Was bedeutet [! -r $1]?

test kann auch verkürzt werden. Die Testbedingung muss dann in eckige Klammern geschrieben werden.
Also statt `if test ! -r $1` schreiben wir einfach `if [ ! -r \$1 ]`.
Außerdem wichtig: Niemals die Leerzeichen nach `[` und vor `]` vergessen!

Suchen und Ersetzen

Nur das erste Vorkommen

```
> echo "alt alt alt" | sed -e "s/alt/neu/"
> neu alt alt
```

Nur aller Vorkommen

```
> echo "alt alt alt" | sed -e "s/alt/neu/g"
> neu neu neu
```

Suchen und Ersetzen

Nur das erste Vorkommen

```
> echo "alt alt alt" | sed -e "s/alt/neu/"
> neu alt alt
```

Nur aller Vorkommen

```
> echo "alt alt alt" | sed -e "s/alt/neu/g"
> neu neu neu
```

Suchen und Ersetzen

Bielefled aus messungen.csv ersetzen

```
> sed -e "s/Sieker/Senne/g" < data/messungen.csv
```

If Verzweigungen

Einzeiler

```
if test -r datei.txt ; then echo da; else echo fehlt; fi
```

Bei Einzeilern nie die Semikolons hinter den Kommandos vergessen.

If Verzweigungen

Einzeiler

```
if test -r datei.txt ; then echo da; else echo fehlt; fi
```

Bei Einzeilern nie die Semikolons hinter den Kommandos vergessen.

If Verzweigungen

Einzeiler mit verkürzter *test* Umgebung

```
if [ -r datei.txt ]; then echo da; else echo fehlt; fi
```

Wichtig

Die Leerzeile nach [und vor] nicht vergessen, ansonsten wir es nicht richtig interpretiert!

If Verzweigungen

Einzeiler mit verkürzter *test* Umgebung

```
if [ -r datei.txt ]; then echo da; else echo fehlt; fi
```

Wichtig

Die Leerzeile nach [und vor] nicht vergessen, ansonsten wir es nicht richtig interpretiert!

Testmöglichkeiten - Zeichenketten

- Gleichheit: `"$1" == "string"`
  Man beachte das es zwei Gleichheitszeichen sein müssen.
- Ungleichheit: `"$1" != "string"`
- Leer: `test -z "$1"`
- Nicht Leer: `test -n "$1"`

Hinweis

`$1` steht hierbei natürlich für den ersten Parameter, wenn das Shell-Skript über die Konsole aufgerufen wird.

Testmöglichkeiten - Zeichenketten

- Gleichheit: `"$1" == "string"`
  Man beachte das es zwei Gleichheitszeichen sein müssen.
- Ungleichheit: `"$1" != "string"`
- Leer: `test -z "$1"`
- Nicht Leer: `test -n "$1"`

Hinweis

`$1` steht hierbei natürlich für den ersten Parameter, wenn das Shell-Skript über die Konsole aufgerufen wird.

Testmöglichkeiten - Zeichenketten

- Gleichheit: `"$1" == "string"`
  Man beachte das es zwei Gleichheitszeichen sein müssen.
- Ungleichheit: `"$1" != "string"`
- Leer: `test -z "$1"`
- Nicht Leer: `test -n "$1"`

Hinweis

`$1` steht hierbei natürlich für den ersten Parameter, wenn das Shell-Skript über die Konsole aufgerufen wird.

Testmöglichkeiten - Zeichenketten

- Gleichheit: `"$1" == "string"`
  Man beachte das es zwei Gleichheitszeichen sein müssen.
- Ungleichheit: `"$1" != "string"`
- Leer: `test -z "$1"`
- Nicht Leer: `test -n "$1"`

Hinweis

`$1` steht hierbei natürlich für den ersten Parameter, wenn das Shell-Skript über die Konsole aufgerufen wird.

Testmöglichkeiten - Zeichenketten

- Gleichheit: `"$1" == "string"`
  Man beachte das es zwei Gleichheitszeichen sein müssen.
- Ungleichheit: `"$1" != "string"`
- Leer: `test -z "$1"`
- Nicht Leer: `test -n "$1"`

Hinweis

`$1` steht hierbei natürlich für den ersten Parameter, wenn das Shell-Skript über die Konsole aufgerufen wird.

Testmöglichkeiten - Zeichenketten

- Gleichheit: `"$1" == "string"`
  Man beachte das es zwei Gleichheitszeichen sein müssen.
- Ungleichheit: `"$1" != "string"`
- Leer: `test -z "$1"`
- Nicht Leer: `test -n "$1"`

Hinweis

`$1` steht hierbei natürlich für den ersten Parameter, wenn das Shell-Skript über die Konsole aufgerufen wird.

Testmöglichkeiten - Ganzzahlen

- Gleichheit: `test $1 -eq 42`
- Ungleichheit: `test $1 -ne 42`
- Größer: `test $1 -gt 42`
- Größer-Gleich: `test $1 -ge 42`
- Kleiner: `test $1 -lt 42`
- Kleiner-Gleich: `test $1 -le 42`

Testmöglichkeiten - Ganzzahlen

- Gleichheit: `test $1 -eq 42`
- Ungleichheit: `test $1 -ne 42`
- Größer: `test $1 -gt 42`
- Größer-Gleich: `test $1 -ge 42`
- Kleiner: `test $1 -lt 42`
- Kleiner-Gleich: `test $1 -le 42`

Testmöglichkeiten - Ganzzahlen

- Gleichheit: `test $1 -eq 42`
- Ungleichheit: `test $1 -ne 42`
- Größer: `test $1 -gt 42`
- Größer-Gleich: `test $1 -ge 42`
- Kleiner: `test $1 -lt 42`
- Kleiner-Gleich: `test $1 -le 42`

Testmöglichkeiten - Ganzzahlen

- Gleichheit: `test $1 -eq 42`
- Ungleichheit: `test $1 -ne 42`
- Größer: `test $1 -gt 42`
- Größer-Gleich: `test $1 -ge 42`
- Kleiner: `test $1 -lt 42`
- Kleiner-Gleich: `test $1 -le 42`

Testmöglichkeiten - Ganzzahlen

- Gleichheit: `test $1 -eq 42`
- Ungleichheit: `test $1 -ne 42`
- Größer: `test $1 -gt 42`
- Größer-Gleich: `test $1 -ge 42`
- Kleiner: `test $1 -lt 42`
- Kleiner-Gleich: `test $1 -le 42`

Testmöglichkeiten - Ganzzahlen

- Gleichheit: `test $1 -eq 42`
- Ungleichheit: `test $1 -ne 42`
- Größer: `test $1 -gt 42`
- Größer-Gleich: `test $1 -ge 42`
- Kleiner: `test $1 -lt 42`
- Kleiner-Gleich: `test $1 -le 42`

Testmöglichkeiten - Dateien

- Existiert und „reguläre“ Datei: `test -f $1`
- Existiert und lesbar: `test -r $1`
- Existiert und schreibbar: `test -w $1`
- Existiert und ausführbar: `test -x $1`
- Existiert und ist Verzeichnis: `test -d $1`

Hinweis

`$1` steht hierbei natürlich für den ersten Parameter, wenn das Shell-Skript über die Konsole aufgerufen wird.

Testmöglichkeiten - Dateien

- Existiert und „reguläre“ Datei: `test -f $1`
- Existiert und lesbar: `test -r $1`
- Existiert und schreibbar: `test -w $1`
- Existiert und ausführbar: `test -x $1`
- Existiert und ist Verzeichnis: `test -d $1`

Hinweis

`$1` steht hierbei natürlich für den ersten Parameter, wenn das Shell-Skript über die Konsole aufgerufen wird.

Testmöglichkeiten - Dateien

- Existiert und „reguläre“ Datei: `test -f $1`
- Existiert und lesbar: `test -r $1`
- Existiert und schreibbar: `test -w $1`
- Existiert und ausführbar: `test -x $1`
- Existiert und ist Verzeichnis: `test -d $1`

Hinweis

`$1` steht hierbei natürlich für den ersten Parameter, wenn das Shell-Skript über die Konsole aufgerufen wird.

Testmöglichkeiten - Dateien

- Existiert und „reguläre“ Datei: `test -f $1`
- Existiert und lesbar: `test -r $1`
- Existiert und schreibbar: `test -w $1`
- Existiert und ausführbar: `test -x $1`
- Existiert und ist Verzeichnis: `test -d $1`

Hinweis

`$1` steht hierbei natürlich für den ersten Parameter, wenn das Shell-Skript über die Konsole aufgerufen wird.

Testmöglichkeiten - Dateien

- Existiert und „reguläre“ Datei: `test -f $1`
- Existiert und lesbar: `test -r $1`
- Existiert und schreibbar: `test -w $1`
- Existiert und ausführbar: `test -x $1`
- Existiert und ist Verzeichnis: `test -d $1`

Hinweis

`$1` steht hierbei natürlich für den ersten Parameter, wenn das Shell-Skript über die Konsole aufgerufen wird.

Testmöglichkeiten - Dateien

- Existiert und „reguläre“ Datei: `test -f $1`
- Existiert und lesbar: `test -r $1`
- Existiert und schreibbar: `test -w $1`
- Existiert und ausführbar: `test -x $1`
- Existiert und ist Verzeichnis: `test -d $1`

Hinweis

`$1` steht hierbei natürlich für den ersten Parameter, wenn das Shell-Skript über die Konsole aufgerufen wird.

Testmöglichkeiten - Dateien

- Existiert und „reguläre“ Datei: `test -f $1`
- Existiert und lesbar: `test -r $1`
- Existiert und schreibbar: `test -w $1`
- Existiert und ausführbar: `test -x $1`
- Existiert und ist Verzeichnis: `test -d $1`

Hinweis

`$1` steht hierbei natürlich für den ersten Parameter, wenn das Shell-Skript über die Konsole aufgerufen wird.

Logische Verknüpfungen

UND

```
if test "$1" == "rot" && test "$2" == "blau" ; then
        echo wahr
else
        echo falsch
fi
```

Logische Verknüpfungen

ODER

```
if test "$1" == "rot" || test "$2" == "blau" ; then
        echo wahr
else
        echo falsch
fi
```

Logische Verknüpfungen

Komplexe Ausdrücke

```
if test "$1" == "rot" &&
 (test "$2" == "apfel" || test "$2" == "kirsche") ; then
        echo wahr
else
        echo falsch
fi
```

Logische Verknüpfungen

Wahrheitswerte

Verschiedene Befehle wie *grep* haben Wahrheitswerte als Rückgabe

```
#!/bin/bash
if grep -q -i $1 data/planeten.txt; then
        echo $1 ist ein Planet
else
        echo $1 ist kein Planet
fi
```

Was ist grep

- Schauen wir doch mal in die **man**.
- Was macht -i?
- Was macht -q?

Was ist grep

- Schauen wir doch mal in die **man**.
- Was macht -i?
- Was macht -q?

Was ist grep

- Schauen wir doch mal in die **man**.
- Was macht -i?
- Was macht -q?

Variablen

Werte zuweisen

Wir können auch selbst Variablen erstellen und diese verwenden.

```
wort="eins"
```

Variablen verwenden

Und wenn wir die Variable wieder benutzen wollen:

```
echo $wort
```

Variablen

Werte zuweisen

Wir können auch selbst Variablen erstellen und diese verwenden.

```
wort="eins"
```

Variablen verwenden

Und wenn wir die Variable wieder benutzen wollen:

```
echo $wort
```

Variablen

Datentypen

Variablen sind „schwach getypt“.

Variablen

Als Zeichenketten

```
> name=datei
> verz=/home/student
> pfad=$verz/$name.jpg
> echo $pfad
/home/student/datei.jpg
```

Zeichenketten - Sonderfälle

Variablennamen durch Klammern vom Text abtrennen:

```
> name=zeichen
> echo ${name}kette
zeichenkette
```

Zeichenketten - Sonderfälle

Leerzeichen durch Anführungszeichen erhalten

```
> a=eins
> b=zwei
> c="$a $b"
> echo $c
eins zwei
```

Arithmetische Ausdrücke

Über $((...)) auswerten

```
a=9
b=3
echo $((a*b))
```

Arithmetische Ausdrücke

Über $((...)) auswerten

```
a=9
b=3
echo $((a*b))
```

Ergebnis

27

Arithmetische Ausdrücke

Über $((...)) auswerten

```
a=9
b=3
echo $((a*b))
```

In arithmentischen Ausdrücken können wir bei Variablen das vorstehende $ weglassen und einfach nur den Variablennamen verwenden.
⇒ Es wird selbständig ermittelt ob es sich um eine Variable handelt!

Variablen

Arithmetische Operatoren

+	Addition
-	Subtraktion
*	Multiplikation
/	Division
%	Modulo

Beispiele

```
> echo $((23 / 5))
4
> echo $((23 % 5))
3
```

Variablen

Arithmetische Operatoren

+	Addition
-	Subtraktion
*	Multiplikation
/	Division
%	Modulo

Beispiele

```
> echo $((23 / 5))
4
> echo $((23 % 5))
3
```

Variablen

Zufall

```
> echo $RANDOM
35346
```

Modulo

```
> echo $((RANDOM%20))
15
```

Variablen

Zufall

```
> echo $RANDOM
35346
```

Modulo

```
> echo $((RANDOM%20))
15
```

Motivation

- Nicht nur einmalige Ausführung erforderlich
- Mehrmalige Ausführung des/der gleichen Kommandos
- Zusätzlich durch Parameter einstellbar
- Schleifen bestehen aus drei Abschnitten: Schleifenkopf, Schleifenrumpf, Schleifenklammerung

```
for i [in ...] do ... done
```

Motivation

- Nicht nur einmalige Ausführung erforderlich
- Mehrmalige Ausführung des/der gleichen Kommandos
- Zusätzlich durch Parameter einstellbar
- Schleifen bestehen aus drei Abschnitten: Schleifenkopf, Schleifenrumpf, Schleifenklammerung

```
for i [in ...] do ... done
```

Motivation

- Nicht nur einmalige Ausführung erforderlich
- Mehrmalige Ausführung des/der gleichen Kommandos
- Zusätzlich durch Parameter einstellbar
- Schleifen bestehen aus drei Abschnitten: Schleifenkopf, Schleifenrumpf, Schleifenklammerung

```
for i [in ...] do ... done
```

Motivation

- Nicht nur einmalige Ausführung erforderlich
- Mehrmalige Ausführung des/der gleichen Kommandos
- Zusätzlich durch Parameter einstellbar
- Schleifen bestehen aus drei Abschnitten: Schleifenkopf, Schleifenrumpf, Schleifenklammerung

```
for i [in ...] do ... done
```

For-Schleifen

Beispiel

```
#!/bin/bash
for x in y
do
        # Do something
done
```

Einzeiler

```
for i in *.jpg; do echo $i; done
```

For-Schleifen

Beispiel

```
#!/bin/bash
for x in y
do
        # Do something
done
```

Einzeiler

```
for i in *.jpg; do echo $i; done
```

For-Schleifen

Diskrete Werte

```
for i in eins zwei drei; do echo $i; done
```

Ausgabe

```
eins
zwei
drei
```

For-Schleifen

Diskrete Werte

```
for i in eins zwei drei; do echo $i; done
```

Ausgabe

```
eins
zwei
drei
```

For-Schleife

Beispiel

x	x*x
1	1
2	4
3	9
...	...

Jemand eine Idee?

For-Schleife

Beispiel

x	x*x
1	1
2	4
3	9
...	...

Jemand eine Idee?

For-Schleifen

Zahlenfolgen

```
seq 3
```

Von 1 bis 3

```
1
2
3
```

For-Schleifen

Zahlenfolgen

```
seq 3
```

Von 1 bis 3

```
1
2
3
```

For-Schleifen

Zahlenfolgen

```
seq 4 6
```

Ab 4 bis 6

```
4
5
6
```

For-Schleifen

Zahlenfolgen

```
seq 4 6
```

Ab 4 bis 6

```
4
5
6
```

For-Schleifen

Zahlenfolgen

```
seq 4 2 10
```

Ab 4 bis 10 in 2er-Schritten

```
4
6
8
10
```

For-Schleifen

Zahlenfolgen

```
seq 4 2 10
```

Ab 4 bis 10 in 2er-Schritten

```
4
6
8
10
```

For-Schleife

Beispiel

x	x*x
1	1
2	4
3	9
...	...

Mit `seq`: Was müssen wir jetzt ändern?

For-Schleife

Beispiel

x	x*x
1	1
2	4
3	9
...	...

Mit `seq`: Was müssen wir jetzt ändern?

Wertetabelle ausgeben

Tabelle mit seq ausgeben

```
#!/bin/bash

echo -e "x\t x*x"
for i in $(seq 10)
do
        echo -e "$i\t $((i*i))"
done
```

For-Schleifen

For und find

Alle jpg Dateien in einen Unterordner verschieben

```
#!/bin/bash

for i in $(find . -name "*.jpg")
do
        mv $i ~/my_tmp_dir
done
```

Find

Man: *find - search for files in a directory hierarchy*

Was ist den find schon wieder?

```
SYNOPSIS
find [-H] [-L] [-P] [-D debugopts] [-Olevel] [starting-point...]
     [expression]
```

Kommando?	*Startverzeichnis*	*Suchkriterien*	*Ausgabeart*
find	**.**	**-name**	**-print**
		-type	**-exec \{}\;**

Find

Hä, was?

```
find . -iname '*.jpg' -exec echo \{} \;
```

```
find . -iname '*.jpg' -exec mv \{} ~/my_tmp_dir/ \;
```

Find

Hä, was?

```
find . -iname '*.jpg' -exec echo \{} \;

find . -iname '*.jpg' -exec mv \{} ~/my_tmp_dir/ \;
```

For-Schleifen

Dateiendungen umbenennen

```
#!/bin/bash

for i in *.JPG
do
        bn=$(basename $i .JPG)
        mv $i $bn.jpg
done
```

For-Schleifen

Dateiendungen umbenennen 2

```
#!/bin/bash

for i in *.$1
do
        bn=$(basename $i .$1)
        mv $i $bn.$2
done
```

Sachen berechnen

Setzen Sie die Reihe fort

1, 1, 2, 3, 5, 8, 13, ...
Und noch viel interessanter

- wie heißt das überhaupt?
- was hat das eigentlich mit dem goldenen Schnitt zu tun?

Nun versuchen wir das einmal zusammen in der Konsole zu implementieren

Sachen berechnen

Setzen Sie die Reihe fort

1, 1, 2, 3, 5, 8, 13, ...
Und noch viel interessanter

- wie heißt das überhaupt?
- was hat das eigentlich mit dem goldenen Schnitt zu tun?

Nun versuchen wir das einmal zusammen in der Konsole zu implementieren

Motivation

Gegeben sei ...

... eine zweispaltige Datei data/zahlen.txt:

90 17
24 1
25 874
...

Motivation

Wir wollen ...

.. diese Datei nun zeilenweise addieren, also:

90 + 17 =

24 + 1 =

25 + 874 =

...

Hierfür würden sich doch for Schleifen anbieten!

Motivation

Wir wollen ...

.. diese Datei nun zeilenweise addieren, also:

90 + 17 =

24 + 1 =

25 + 874 =

...

Hierfür würden sich doch for Schleifen anbieten!

Unter Einsatz von for Schleifen

```
for i in $(cat data/zahlen.txt); do
        echo $i
done
```

Ausgabe

```
90
17
24
...
```

Unter Einsatz von for Schleifen

- for-Schleifen arbeiten elementweise
- $\Rightarrow$ Helfen uns hier nicht weiter

Unter Einsatz von for Schleifen

- for-Schleifen arbeiten elementweise
- $\Rightarrow$ Helfen uns hier nicht weiter

While Schleifen

```
while steuerbefehl; do
        Befehl1
        Befehl2
        ...
        Befehl
        n
done
```

While Schleifen

Beispiel

```
#!/bin/bash
zaehler=1
while test $zaehler -le 3; do
        echo $zaehler
        zaehler=$((zaehler+1))
done
```

`read` zum zeilenweisem Einlesen

read hilft eine Zeile aus der Eingabe zu lesen

- `read line`
- List eine Zeile aus der Eingabe in die Variable *line* ein
- *line* bekommt den Wert „falsch“ zugewiesen wenn die Eingabe leer ist.

read

`read` zum zeilenweisem Einlesen

read hilft eine Zeile aus der Eingabe zu lesen

- `read line`
- List eine Zeile aus der Eingabe in die Variable *line* ein
- *line* bekommt den Wert „falsch“ zugewiesen wenn die Eingabe leer ist.

read

`read` zum zeilenweisem Einlesen

read hilft eine Zeile aus der Eingabe zu lesen

- `read line`
- List eine Zeile aus der Eingabe in die Variable *line* ein
- *line* bekommt den Wert „falsch“ zugewiesen wenn die Eingabe leer ist.

Read

Beispiel

```
#!/bin/bash
while read line; do
        echo "Zeile: $line"
done < data/zahlen.txt
```

Lösung unter Verwendung von while

Zeilenweises addieren

```
#!/bin/bash
while read line; do
        a=$(echo $line | cut -d\  -f 1) # Nach -d\ kommen zwei
             Leerzeichen
        # Leerzeichen 1 ist Parameter "\ " der -d uebergeben wird
        # Das zweite eine einfache Trennung zwischen den Parametern
        b=$(echo $line | cut -d\  -f 2)
        echo $a + $b = $((a+b))
done < data/zahlen.txt
```

Was macht das cut hier?

```
a=$(echo $line | cut -d\  -f 1)
```

Schauen wir uns das doch mal an einem kleinen Beispiel an.

Beispiele

```
echo "42 4 1234 73 56" | cut -d\  -f 1
echo "42 4 1234 73 56" | cut -d\  -f 2
echo "42 4 1234 73 56" | cut -d\  -f 2,3
echo "42 4 1234 73 56" | cut -d\  -f 2-4
echo "42 4 1234 73 56" | cut -d\  -f 2-4,1
```

Was macht das cut hier?

```
a=$(echo $line | cut -d\  -f 1)
```

Schauen wir uns das doch mal an einem kleinen Beispiel an.

Beispiele

```
echo "42 4 1234 73 56" | cut -d\  -f 1
echo "42 4 1234 73 56" | cut -d\  -f 2
echo "42 4 1234 73 56" | cut -d\  -f 2,3
echo "42 4 1234 73 56" | cut -d\  -f 2-4
echo "42 4 1234 73 56" | cut -d\  -f 2-4,1
```

Verschärfte Bedingungen

1		4	3	
8	10	9	7	
2	8	1		
10	9	12	7	1
1	9			

Unterschiedliche Anzahl an Werten pro Zeile.

Verschärfte Bedingungen

1		4	3	
8	10	9	7	
2	8	1		
10	9	12	7	1
1	9			

Unterschiedliche Anzahl an Werten pro Zeile.

Arrays

Standardfall

- Jeder Variable ist ein Speicherbereich zugeordnet
- n Werte $\Rightarrow$ n Variablen und Speicherbereiche

Arrays

Array/Feld-Variablen

- Jeder Variable sind n Speicherbereiche zugeordnet
- Ein Name für n Speicherbereiche
- Zugriff/Unterscheidung durch Index

Arrays

Wie erstellt man Arrays?

```
a=(1  2  7  9)
```

Ein Element ausgeben

```
> echo ${a[1]}
2
```

Alle Elemente ausgeben

```
> echo ${a[*]}
1 2 7 9
```

Anzahl Elemente ausgeben

```
> echo ${#a[*]}
4
```

Arrays

Wie erstellt man Arrays?

```
a=(1 2 7 9)
```

Ein Element ausgeben

```
> echo ${a[1]}
2
```

Alle Elemente ausgeben

```
> echo ${a[*]}
1 2 7 9
```

Anzahl Elemente ausgeben

```
> echo ${#a[*]}
4
```

Arrays

Wie erstellt man Arrays?

```
a=(1 2 7 9)
```

Ein Element ausgeben

```
> echo ${a[1]}
2
```

Alle Elemente ausgeben

```
> echo ${a[*]}
1 2 7 9
```

Anzahl Elemente ausgeben

```
> echo ${#a[*]}
4
```

Arrays

Wie erstellt man Arrays?

```
a=(1 2 7 9)
```

Ein Element ausgeben

```
> echo ${a[1]}
2
```

Alle Elemente ausgeben

```
> echo ${a[*]}
1 2 7 9
```

Anzahl Elemente ausgeben

```
> echo ${#a[*]}
4
```

Mehrere Elemente gleichzeitig setzen

```
> b=(eins zwei drei vier fuenf)
> echo ${b[*]}
eins zwei drei vier fuenf
```

Lösung unter Verwendung von while

Zeilenweises ohne Array

```
#!/bin/bash
while read line; do
        line=$(echo $line | tr -s " ")
        a=$(echo $line | cut -d\  -f 1)
        b=$(echo $line | cut -d\  -f 2)
        echo $a + $b = $((a+b))
done < data/zahlen.txt
```

Lösung unter Verwendung von while

Zeilenweises addieren mit Array

```
while read line; do
        z=($line)
        echo ${z[0]} + ${z[1]} = $((${z[0]}+${z[1]}))
done < data/zahlen.txt
```

Komplexes Beispiel

Zeilenweises addieren mit Array

```
while read line; do
        a=($line)
        letztes=$((${#a[*]}-1))
        sum=${a[0]}
        echo -n "$sum "
        for i in $(seq 1 $letztes); do
                sum=$((sum+${a[$i]}))
                echo -n "+ ${a[$i]} "
        done
        echo "= $sum"
done < data/zahlen2.txt
```

Abschluss

Tagwerk

- Viel gelernt
- Was ist eine Gui
- Vergleich von Gui und Konsole
- Mit der Bash gearbeitet