Programmiermethoden und -werkzeuge 1

Woche 7 - Bash Scripting

Jochen Hosenfeld

jochen.hosenfeld@informatik.hs-fulda.de

Fachbereich Angewandte Informatik

December 5, 2025

Heutige Inhalte

- Wiederholung: RANDOM
- Variablen
- Arithmetik
- Eingabe und Ausgabe
- Vergleichsoperatoren
- Kontrollstrukturen

RANDOM

- Spezielle Shell-Variable
- Generiert (pseudo)zufällige Zahl zwischen 0 und 32767
- Bis zur maximalen Zahl für ein 15-Bit-Integer
- *pseudo random number generator* (PRNG)

Verwendung von RANDOM

```
1  echo $RANDOM                # Zufallszahl zwischen 0 und 32767
2  echo $(($RANDOM % 10))      # Zufallszahl zwischen 0 und 9 durch Modulo-Operator
3
4  RANDOM=42                   # Setzen des Seeds
5
6  # Zufallszahl in einem benutzerdefinierten Bereich  (z. B. zwischen min und max)
7  min=-100
8  max=100
9  echo $((RANDOM % (max - min + 1) + min))
```

((...)) - Compound Command (Zusammengesetzter Befehl)

Die Klammern `((...))` in Bash ermöglichen die arithmetische Auswertung, und das `$`-Zeichen davor sorgt dafür, dass das Ergebnis dieses Ausdrucks im Befehl verwendet wird.

Variablen

- Variablendeklaration:
  - → Syntax: `variableName=value`
  - → Kein Leerzeichen um das Gleichheitszeichen
- Verwendung von Variablen:
  - → Zugriff mit `$variableName`
  - → Beispiel: `echo $variableName`

Variablennamen

- Können aus alphanumerischen Zeichen (Buchstaben und Zahlen) und Unterstrichen bestehen
- Erstes Zeichen muss entweder ein Buchstabe oder ein Unterstrich sein
- Leerzeichen und Satzzeichen sind nicht erlaubt

Best Practices

- Großbuchstaben, um Konstanten zu kennzeichnen: `PI=3.1415`
- Kleinbuchstaben für tatsächliche Variablen

Arithmetik

Grundrechenarten, Operationen mit Zahlen

- Einfache Arithmetik mit `$(( ))`
  - → Syntax: `result=$((num1 + num2))`
  - → Unterstützt Operatoren wie `+`, `-`, `*`, `/`, `%`.
- Alternative für einfache Rechnungen mit `let`
  - → Syntax: `let result=num1+num2`
- `bc` (Basic Calculator) für erweiterte Mathematik
  - → `echo "a(1)" | bc -l`
  - → Präzise und komplexe Berechnungen wie z. B. Fließkommazahlen sowie Funktionen wie Sinus, Kosinus und Quadratwurzel

Basic Calculator

- Installieren mit `sudo apt install bc`

```
num1=10
num2=3
angle_degrees=30

PI=$(echo "4 * a(1)" | bc -l)                                  # Ermittlung von π
float_result=$(echo "$num1 / $num2" | bc -l)                   # Fließkommazahlen
angle_radians=$(echo "$angle_degrees * $PI / 180" | bc -l)     # Grad zu Bogenmaß
sin_result=$(echo "s($angle_radians)" | bc -l)                 # Sinus
cos_result=$(echo "c($angle_radians)" | bc -l)                 # Kosinus
sqrt_result=$(echo "sqrt($num1)" | bc -l)                      # Quadratwurzel
```

Ausgabe mit `echo`

- Einfacher Text: `echo "Hallo Welt"`
- Variablen in der Ausgabe: `echo "Der Wert ist $variable"`

```
# Optionen
-n          # Kein Zeilenumbruch
-e          # Erlaubt Escape-Sequenzen

echo -n "Dies ist Teil 1"
echo ", und dies ist Teil 2 in derselben Zeile."

echo -e "Zeile 1\nZeile 2"
```

Ausgabe mit `printf`

- printf -> *print formatted*
- Formatierter Text: `printf "Hallo %s\n" "Welt"`
- Erlaubt im Vergleich zu `echo` präzisere Kontrolle über die Ausgabe

```
1  printf FORMAT [ARGUMENT]...
```

- `FORMAT`: Zeichenkette, die Platzhalter wie `%s`, `%d`, `%f` usw. enthält, um anzugeben, wie die Argumente formatiert werden sollen

  → Linksbündig, rechtsbündig, führende Nullen usw.
- `ARGUMENT`: Werte, die in den Platzhaltern eingefügt werden

Ausgabe mit `printf`

```
# Format-Spezifikatoren:
# %s - String
# %d - Ganzzahl
# %f - Gleitkommazahl (z.B. %.2f für zwei Dezimalstellen)

name="Alice"
punktzahl=95
durchschnitt=92.75

printf "Name: %s\n" "$name"
printf "Punktzahl: %d\n" "$punktzahl"
printf "Durchschnitt: %.1f\n" "$durchschnitt"

# Mehrere Variablen mit einem Aufruf
printf "Student: %s | Punktzahl: %d | Durchschnitt: %.1f\n" "$name" "$punktzahl" "$durchschnitt"
```

Vergleich `echo` zu `printf`

Merkmal	`echo`	`printf`
Einfachheit	Sehr einfach zu verwenden	Komplexer aufgrund der Formatangaben
Zeilenumbruch	Fügt automatisch einen hinzu	Kein automatischer Zeilenumbruch
Escape-Sequenzen	Variiert, benötigt oft `-e`	Konsistente Unterstützung
Portabilität	Kann sich auf verschiedenen Systemen unterscheiden	Konsistentes Verhalten über Systeme hinweg
Formatierbarkeit	Begrenzt	Umfangreich
Verwendung	Einfache, schnelle Ausgaben	Präzise, komplexe Ausgaben
Datentypangaben	Keine expliziten Datentypen	Unterstützt Platzhalter wie `%s`, `%d`

Eingabe mit `read`

- Syntax: `read variableName`
- Beispiel: `echo "Bitte eingeben:"; read userInput`

```
# Optionen
-p          # Zeigt eine Eingabeaufforderung an
-r          # Backslashes werden nicht als Escape-Zeichen behandelt
-s          # Unterdrückt die Anzeige der Eingabe (nützlich für Passwörter)

read -p "Name: " name
read -s -p "Password: " password
echo        # Fügt eine neue Zeile hinzu, um die Ausgabe zu trennen
echo "Hallo, $name!"
```

Übergeben von Argumenten

- Positionsparameter nutzen:
  - → Zugriff mittels `$1`, `$2`, …, `$N` für Argumente
  - → Beispiel: `echo "Erstes Argument: $1"`
- Skripte aufrufen mit Argumenten:
  - → Syntax: `./scriptname.sh arg1 arg2`

```
1 echo "Skriptname: $0"
2 echo "Anzahl der übergebenen Argumente: $#"
3 echo "Alle Argumente als separate Strings: $@"
4 echo "Alle Argumente als einzelner String: $*"
```

tput

- Terminalsteuerung über terminfo-Datenbank
- Generiert Escape-Codes für Farben und Attribute
- `tput setaf N`: Vordergrundfarbe setzen (0-255)
- `tput sgr0`: Alle Attribute zurücksetzen

tput – Anwendung

- Farbvariablen definieren
- `red=$(tput setaf 1)`
- `RESET=$(tput sgr0)`
- Praktische Beispiele

```
1  echo "${red}Fehler${RESET}"
2
3  tput bold; echo "Fett"; tput sgr0
```

Kontrollstrukturen

Kontrollstrukturen – Übersicht

- Bedeutung von Kontrollstrukturen
  - → Ermöglichen bedingte Ausführung von Code
  - → Wesentlich für Entscheidungsfindung und Schleifen
- Typen von Kontrollstrukturen
  - → Bedingte Anweisungen: `if`, `else`, `elif`
  - → Schleifen: `for`, `while`
  - → Fallunterscheidungen: `case`

`if`-Anweisung – Syntax

- Syntax der `if`-Anweisung:

```
1  if [[ Bedingung ]]; then
2      # Codeblock bei erfüllter Bedingung
3  fi
```

- Beispielhafte Bedingung:

```
1  # Überprüfung, ob eine Datei existiert
2  if [[ -f myfile.txt ]]; then
3      echo "Datei existiert"
4  fi
```

Vergleichsoperatoren für Zahlen

```
1  # Zahlenvergleich
2  -eq         # Prüft auf Gleichheit von Zahlen ("equal")
3  -ne         # Prüft auf Ungleichheit von Zahlen ("not equal")
4  -gt         # Prüft, ob eine Zahl größer als eine andere ist ("greater than")
5  -lt         # Prüft, ob eine Zahl kleiner als eine andere ist ("less  than")
6
7
8  zahl1=5
9  zahl2=5
10
11 # Vergleich
12 if [[ $zahl1 -eq $zahl2 ]]; then
13   echo "Die Zahlen sind gleich."
14 fi
```

Vergleichsoperatoren für Zeichen und Dateien

```
# Zeichenkettenvergleich
if [[ "$str1" = "$str2" ]]; then
  echo "Die Zeichenketten sind gleich."
fi


# Dateioperationen
-d          # Prüft, ob ein Verzeichnis existiert.
-e          # Prüft, ob eine Datei oder ein Verzeichnis existiert.

if [[ -d "$verzeichnis" ]]; then
  echo "Das Verzeichnis $verzeichnis existiert."
fi
```

`if`-Anweisung mit `else`

- Fügt eine Alternative hinzu, wenn die Bedingung nicht erfüllt ist.

```
1 if [[ -d mydir ]]; then
2     echo "Verzeichnis existiert"
3 else
4     echo "Verzeichnis existiert nicht"
5 fi
```

`if`-Anweisung mit `elif`

- `if`-Anweisungen können innerhalb von `if`- oder `else`-Blöcken verschachtelt werden.

```
1 if [[ $zahl -gt 10 ]]; then
2     echo "Zahl ist größer als 10"
3 elif [[ $zahl -eq 10 ]]; then
4     echo "Zahl ist gleich 10"
5 else
6     echo "Zahl ist kleiner als 10"
7 fi
```

`if`-Anweisung mit `elif`

- `if`-Anweisungen können innerhalb von `if`- oder `else`-Blöcken verschachtelt werden.

```
1 if [[ $tier = "Hund" ]]; then
2     if [[ $alter -lt 3 ]]; then
3         echo "Junger Hund"
4     else
5         echo "Erwachsener Hund"
6     fi
7 fi
```

`if`-Anweisung - Best Practices

Best Practices

- Vermeidung unnötig verschachtelter Bedingungen
  → besser logische Strukturierung
- Achten auf korrekte Verwendung von eckigen Klammern und deren Abstände

`for`-Schleifen – Syntax 1

- Syntax der `for`-Schleife:

```
for variable in liste; do
    # Codeblock für jedes Element
done
```

`for`-Schleifen – Syntax 2

- Iteration über eine Liste von Werten:

```
1 for farbe in rot blau grün; do
2     echo "Farbe: $farbe"
3 done
```

`for`-Schleifen – Syntax 3

- Bearbeitung von Dateien in einem Verzeichnis:

```
1 for datei in *.txt; do
2     echo "Datei: $datei"
3 done
```

`for`-Schleifen – Syntax 4

- Iteration über die Ausgabe eines Befehls:

```
1  for benutzer in $(cut -f1 -d: /etc/passwd); do
2          echo "Benutzer: $benutzer"
3      done
```

Praktische Anwendungen von `for`-Schleifen

- Iteration über ein Verzeichnis:

  → Operation auf mehreren Dateien:

```
for datei in /path/to/directory/*.jpg; do
    echo "Bearbeite $datei"
    # Beispiel-Befehl zur Bearbeitung
    convert "$datei" -resize 800x600 "resized-$datei"
done
```

`while`-Schleifen – Syntax 1

- Syntax der `while`-Schleife:

```
1 while [[ Bedingung ]]; do
2     # Codeblock, solange die Bedingung wahr ist
3 done
```

`while`-Schleifen – Syntax 2

- Beispiel für eine einfache `while`-Schleife:

  → Schrittweises Verringern bis zum Schwellenwert:

```
1  zahl=5
2  while [[ $zahl -gt 0 ]]; do
3      echo "Nummer: $zahl"
4      zahl=$((zahl - 1))
5  done
```

`while`-Schleifen – Syntax 3

- Warten auf Benutzereingabe:

  → Schleife bis zur Eingabe eines bestimmten Wertes:

```
1 eingabe=""
2 while [[ "$eingabe" != "exit" ]]; do
3     echo "Tippe 'exit' zum Beenden"
4     read eingabe
5 done
```

Fallunterscheidung mit `case`

- Syntax des `case`-Statements:

```
case variable in
    Muster1)
        # Codeblock für Muster1
        ;;
    Muster2)
        # Codeblock für Muster2
        ;;
    *)
        # Standardfall
        ;;
esac
```

Beispiel für ein Menüsystem

```
1  echo "Wählen Sie eine Aktion: start, stop, status"
2  read aktion
3  case $aktion in
4      start)
5          echo "Dienst starten..."
6          ;;
7      stop)
8          echo "Dienst stoppen..."
9          ;;
10     status)
11         echo "Dienststatus anzeigen..."
12         ;;
13     *)
14         echo "Ungültige Eingabe"
15         ;;
16 esac
```

Fallunterscheidung mit `case`

Tipps

- Nutzung von Wildcards: `*` für alle anderen Fälle
- Mehrere Optionen in einem Fall mit logischem ODER: `option1|option2`
- Beachten der Platzierung der `;;` am Ende jedes Fallblocks