Programmieren Vorkurs

Tag 4 - Bedingungen und Git

Nils-André Forjahn, 04.04.2019

Über mich

Nils-André Forjahn

- Studiere Informatik
- Java-HiWi am Institut für Softwaretechnik und Fahrzeuginformatik
- Lehr-HiWi am Institut für Anwendungssicherheit
- Mail: `n.forjahn@tu-braunschweig.de`

Gliederung

- **Bedingungen**
  - Einführung
  - Aussagenlogischer Hintergrund
  - Bedingungen in Java
  - if-Klauseln
  - Schleifen
- **Git**
  - Warum Git?
  - Aufbau
  - Arbeiten mit Git

Einführung

Motivation

- Präzise Bestimmung des Wahrheitsgehalts von gewissen Bedingungen
- Nicht jeder Teil eines Algorithmus soll immer auf die selbe Art ausgeführt werden
  Beispiel: Alternativen und Portionenskalierung bei Kochrezepten, Lichtschalter im Haus, ...

$\Rightarrow$ Konstrukt zum Auswerten von Bedingungen und Steuerung der Codeausführung benötigt!

Aussagenlogik

Aussagen

- Eine Aussage ist eine Wahrheitsaussage, dessen Ergebnis zum Zeitpunkt der Auswertung eindeutig bestimmt werden kann
- Sie ist entweder **wahr** oder **falsch**!
- Beispiele:
  - Das Wasser ist nass
  - *Alien* ist besser als *Aliens*
  - 10 ist kleiner als 5
  - Ich bin männlich

Aussagenlogik

Aussagen

- Eine Aussage ist eine Wahrheitsaussage, dessen Ergebnis zum Zeitpunkt der Auswertung **eindeutig bestimmt** werden kann
- Sie ist entweder **wahr** oder **falsch**!
- Beispiele:
  - Das Wasser ist nass
  - *Alien* ist besser als *Aliens*
  - 10 ist kleiner als 5
  - Ich bin männlich (Did you just assume my gender?!)

$\Rightarrow$ In der Informatik geht es um präzise formulierte Aussagen!

Aussagenlogik

Logische Operatoren

- Einfache, elementare Aussagen reichen oft nicht aus, stattdessen werden sie mit Junktoren zusammengesetzt
- Beispiele:
  - Braunschweig liegt in Niedersachen **und** hat über 250.000 Einwohner
  - Bring mir ein Erdbeer- **oder** Vanille-Cornetto mit
  - Geld **oder** leben!
  - **Wenn** ich das machen muss, **dann** hilfst du mir! (Implikation)

Bedingungen in Java

Aussagen in Java

- Aussagen werden in Java auch **Bedingungen** genannt
- Ihre Ergebnisse sind entweder **true** oder **false**
- Nicht umgangssprachliche, **präzise Formulierung** gefordert
- Durch den Datentyp **boolean** repräsentiert

Bedingungen in Java

```
boolean fun = false;

boolean asleep = true;
```

Bedingungen in Java

Komplexe Bedingungen

- Wie schon bei Aussagen lassen sich auch Bedingungen miteinander kombinieren
- Eine Kombination einer Bedingung mit einem **Operator** ergibt wieder eine Bedingung

Bedingungen in Java

```
boolean fun = false;
boolean awake = true;
boolean leave = !fun && awake;

boolean complexCondition =
        (!true && !false) || (true || false);
```

Bedingungen in Java

Operatorenübersicht

Name	Formal	Java	wahr, falls
and	$\wedge$	&&	beide Bedingungen wahr sind.
or	$\vee$	||	eine der Bedingungen wahr ist.
exclusive or	xor	^	genau eine der Bedingungen wahr ist.
not	$\neg$	!	die Bedingung falsch ist.
implies	$\Longrightarrow$	!(A || B)	wenn A wahr ist auch B wahr ist.

Bedingungen in Java

Zahlenvergleiche

- Vergleiche zwischen Zahlen sind auch Bedingungen
- Ihr Ergebnis kann genauso wie andere Bedingungen zusammengesetzt werden
- Beispiel: Braunschweig hat **mehr als** 250.000 Einwohner **und weniger als** Berlin.

Bedingungen in Java

```
int einwohnerBs = 251364;
int einwohnerBer = 3575000;

boolean aussage = (einwohnerBs > 250000)
                  && (einwohnerBs < einwohnerBer);
```

Bedingungen in Java

Zahlenvergleichsoperatoren

Name	Formal	Java	wahr, falls
equals	$=$	==	beide Zahlen gleich
not equals	$\neq$	!=	eine Zahl unterschiedlich
less than	$<$	<	erste Zahl kleiner
greater than	$>$	>	erste Zahl größer
less than or equal or	$\leq$	<=	erste Zahl kleiner oder gleich
greater then or equal	$\geq$	>=	erste Zahl größer oder gleich

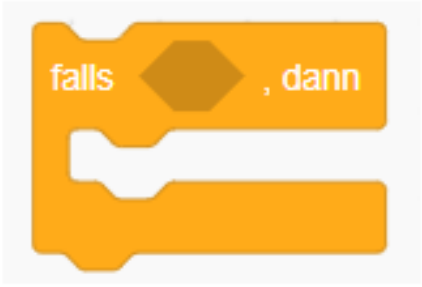
if-Klauseln

Anwendung: if

- if-Klauseln führen einen Codeblock **genau dann** aus, wenn ihre Bedingung wahr ist
- Dienen als „Schalter“ im Programmcode

if-Klauseln

Scratch



Java

```
if(<Bedingung>) {
        ...
}
```

if-Klauseln

```
boolean fun = false;
boolean awake = true;
boolean leave = !fun && awake;

if(leave) {
   System.out.println("Ich verschwinde!");
}
```

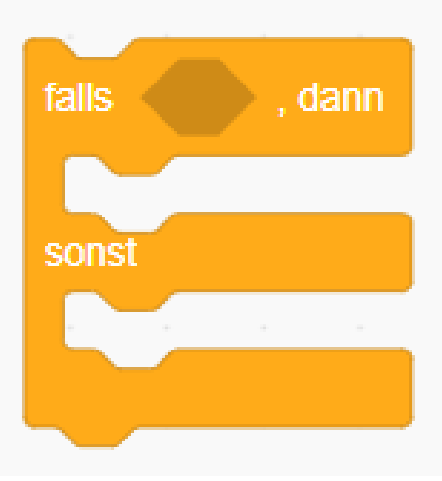
if-Klauseln

else

- Wird verwendet wenn ein anderer Codeteil ausgeführt werden soll, falls die Bedingung **falsch** ist
- Steht direkt nach der if-Klausel und führt den eingeschlossenen Code bei falscher Bedingung aus

if-Klauseln

Scratch



Java

```
if(<Bedingung>) {
        ...
} else {
        ...
}
```

if-Klauseln

```
int einwohnerDierfeld = 10;
int grossstadtGrenze = 100000;

if(einwohnerDierfeld >= grossstadtGrenze) {
    System.out.println("Df. ist eine Grossstadt!");
} else {
    System.out.println("Df. ist KEINE Grossstadt!");
}
```

Schleifen

Motivation

- Menge an Input meist nicht im voraus bekannt
  Beispiel: Überprüfung, ob alle Zahlen einer Menge gerade sind.
- Terminierung des Programms nach Output oft nicht erwünscht
  Beispiel: Internetbrowser, Computerspiele, Betriebssysteme, ...

$\Rightarrow$ Konstrukt zur wiederholten, steuerbaren Ausführung von Code benötigt!

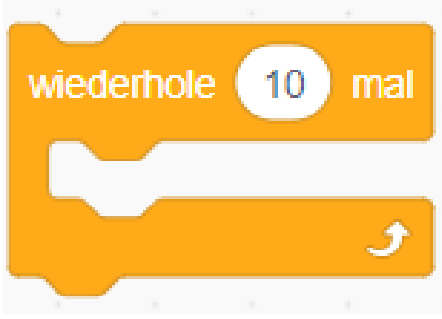
Schleifen

for

- Führt den eingeschlossenen Codeblock so lange aus, wie eine Bedingung erfüllt ist
- Stellt einen Zähler zur Verfügung, der nach jedem Schleifendurchlauf verändert werden kann, z.B. um 1 hochgezählt
- Ideal zum Arbeiten mit Datenmengen wie Arrays

Schleifen

Scratch



Java

```
for(int i = 1; i <= 10; i++)

}
```

Schleifen

```
for(int i = 1; i < 100; i++) {
   System.out.println(i);
}
```

Schleifen

while

- Führt den eingeschlossenen Codeblock so lange aus, wie eine Bedingung erfüllt ist
- Stellt keinerlei Zähler oder andere Variablen zur Verfügung
- Eignet sich besonders zur Realisierung von nicht-terminierenden Programmen

Schleifen

Scratch

Java

```
while(<Bedingung>) {

}
```

Schleifen

```
boolean laufen = true;

while(laufen) {
    System.out.println("Ich␣wiederhole␣mich!");
}
```

Fazit

Zusammenfassung

- Bedingungen sind die Umsetzung von Aussagenlogik in Java
- Sie spielen eine große Bedeutung in der **Steuerung des Programmablaufs**
- if-Klauseln erlauben das Steuern der Codeausführung anhand von Bedingungen
- Schleifen erlauben die steuerbare Mehrfachausführung von Codeblöcken

Git

Verteiltes Arbeiten im Team gehört zum Alltag in der Softwareentwicklung

- Mitglieder können sich auf verschiedenen Kontinenten befinden, u.U. ohne Internetzugang
- Verschiedene Personen könnten auf den selben Dateien arbeiten
- Falsche Änderungen wie das löschen oder verändern von kritischem Code könnte das gesamte Projekt sabotieren

→ Gelöst durch Git!

Git

Warum Git?

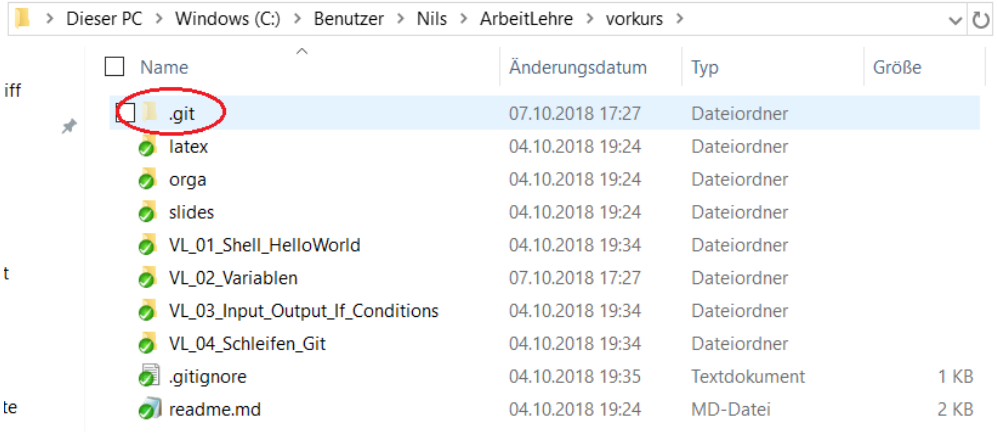
- Freies, verteiltes Versionssystem
- Seit 2005 in der Entwicklung
- Besonders im Open-Source-Bereich verbreitet (siehe Github)
- Erlaubt das Branchen von Projekten
- Wird außerdem zur **Abgabe der Programmierenhausaufgaben** genutzt!

Git

Struktur

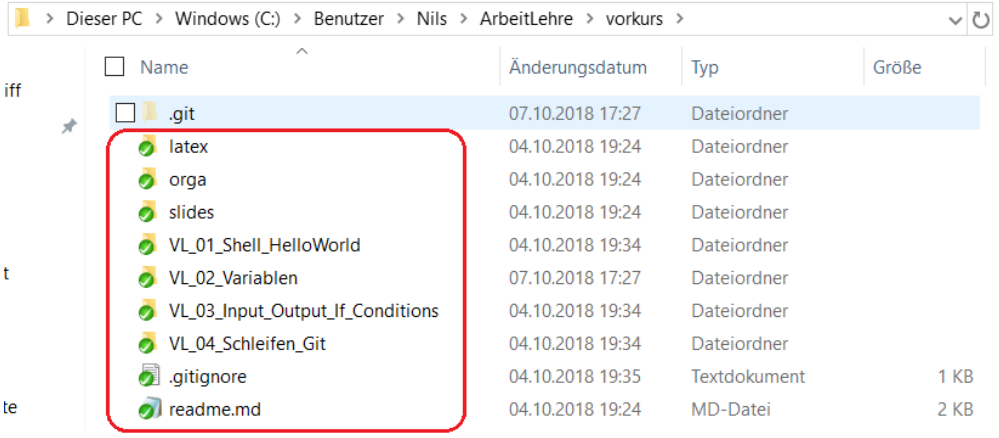
- Der Verlauf eines Projekts wird im `Repository` gespeichert
- Jeder Nutzer hat seine eigene Version des Projekts (`Working Copy`) und Repositories (`Local Repository`)
- Meist gibt es ein zentrales Repository (`Remote Repository`) für das Projekt, das jeder Nutzer mit seinem abgleicht
- Änderungen am Projekt werden in der `Staging Area` erfasst und in `Commits` festgehalten

Git



Local Repository

Git



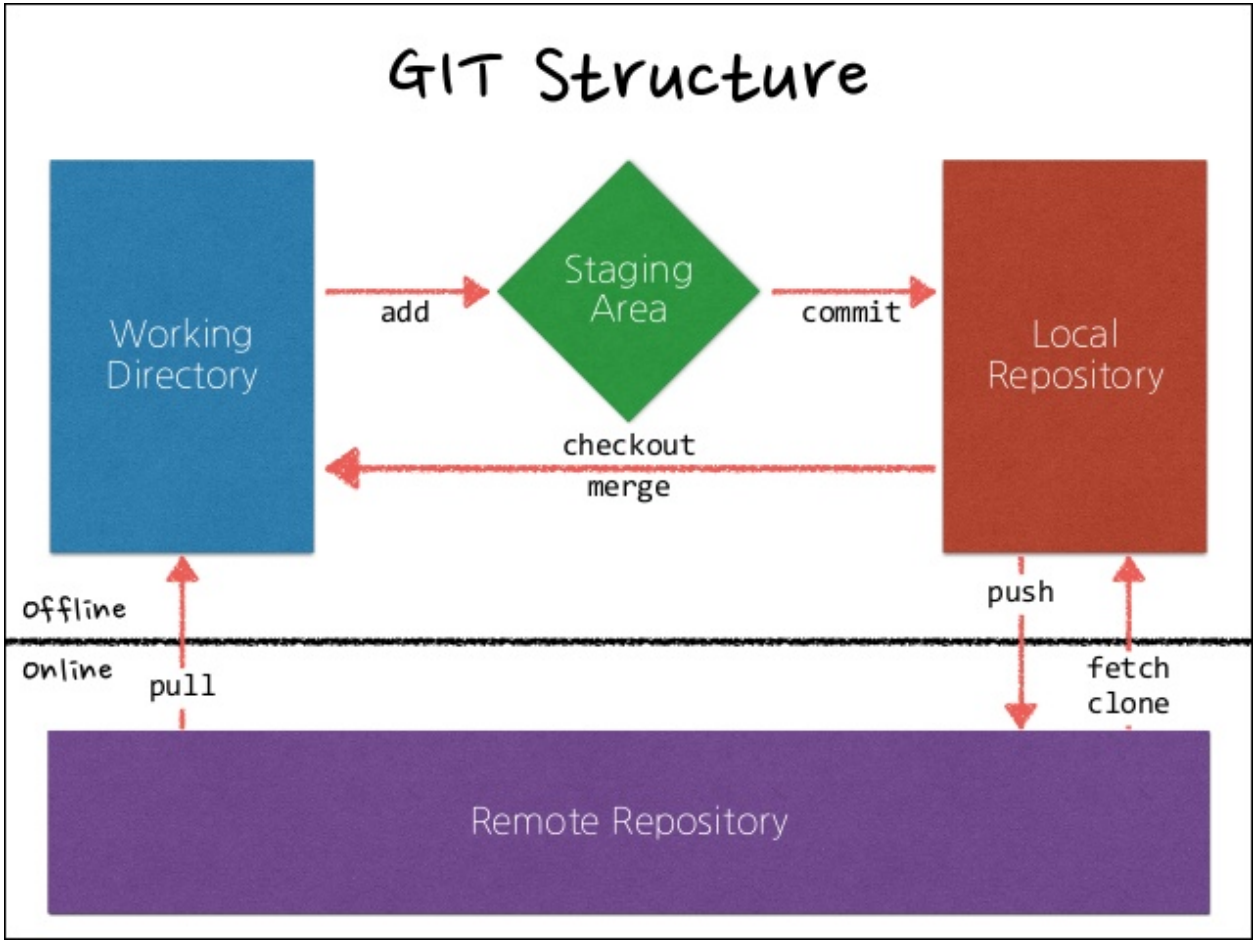
Working Copy

Git

Arbeiten mit Git

- Klonen eines Remote Repositories mittels
  `git clone <PfadZumRemoteRepo>`
- Erfassen von Dateiänderungen mit
  `git add <Dateiname>`
- Festhalten von Änderungen mit
  `git commit -m «Kommentar>"`
- Übertragen von Commits zum Remote Repository durch
  `git push origin master`
- Beziehen und Umsetzen von Commits vom Remote Repository durch
  `git pull`

Git

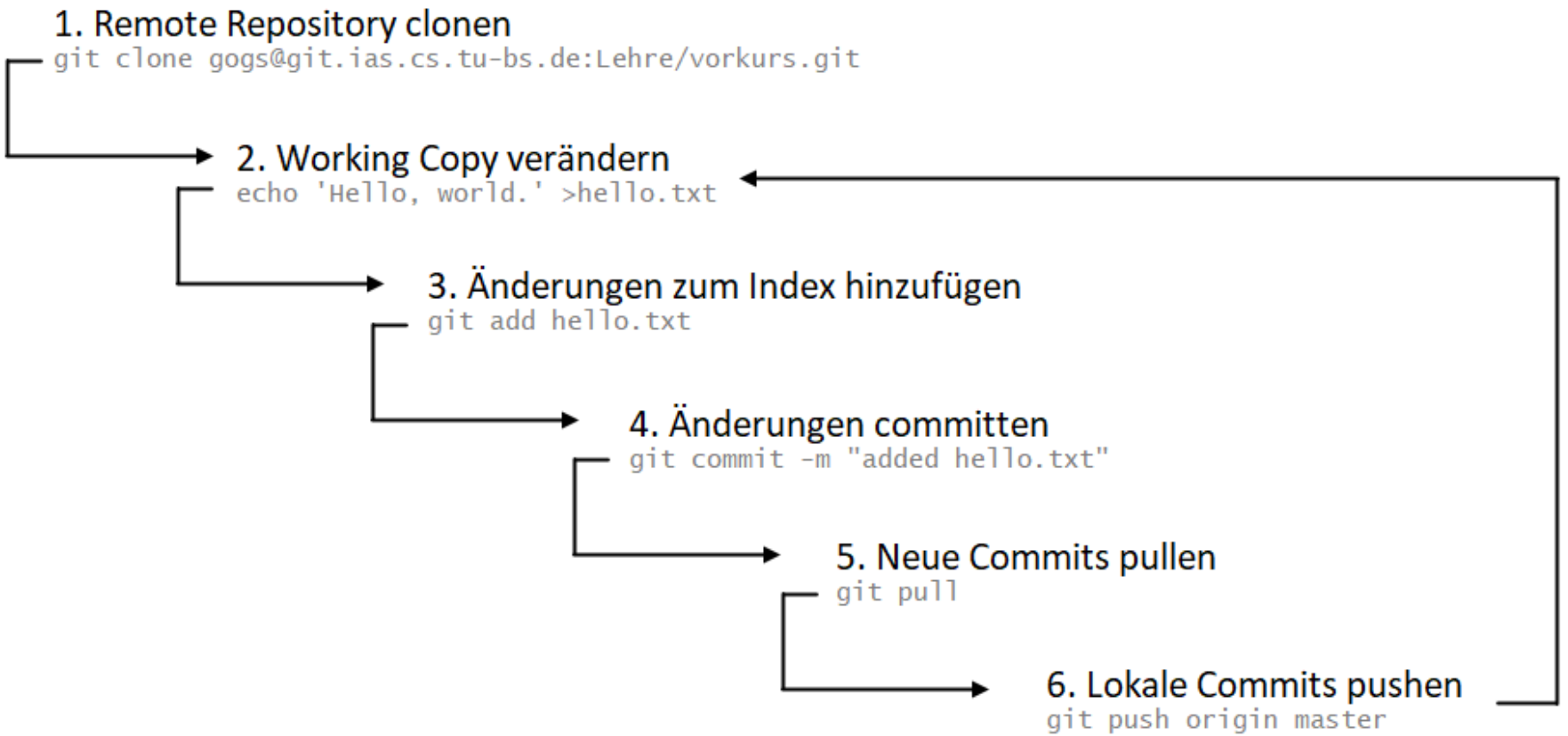

GIT Structure
Working Directory
add
Staging Area
commit
Local Repository
checkout
merge
offline
online
pull
push
fetch
clone
Remote Repository

Git

Weitere Befehle

- Erzeugen eines neuen Respositories
  `git init`
- Updaten des Local Repository ohne Working Copy
  `git fetch origin`
- Einarbeiten der Änderungen im Local Repository in die Working Copy
  `git merge`
- Und wenn man mal nicht weiter weiß…
  `git help`

Git


1. Remote Repository clonen
git clone gogs@git.ias.cs.tu-bs.de:Lehre/vorkurs.git
2. Working Copy verändern
echo 'Hello, world.' >hello.txt
3. Änderungen zum Index hinzufügen
git add hello.txt
4. Änderungen committen
git commit -m "added hello.txt"
5. Neue Commits pullen
git pull
6. Lokale Commits pushen
git push origin master

Git

Konflikte

- Es kann passieren, dass ein Commit Dateien ändert, die man lokal verändert hat
- In vielen Fällen kann Git diesen Konflikt auflösen
- Bei Änderung z.B. der selben Codezeile ist keine automatische Auflösung möglich, manche Konflikte müssen per Hand gelöst werden

$\rightarrow$ Lösung durch Modifikation und erneutem Hinzufügen der Konfliktdatei!

Git

Konfliktbewältigung

1. Konfliktdatei öffnen
2. Konfliktmarkierungen löschen, bis gewünschte Version der Datei vorhanden ist
3. Konfliktdatei adden und commiten

Abschluss

Vielen Dank für eure Aufmerksamkeit!
Ich hoffe ihr seid noch wach. :)