

Verteilte Systeme – Übung

Grundlagen

Sommersemester 2023

Laura Lawniczak, Harald Böhm, Tobias Distler

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
Lehrstuhl Informatik 16 (Systemsoftware)

<https://sys.cs.fau.de>



Lehrstuhl für Verteilte Systeme
und Betriebssysteme



FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Java

- Collections & Maps

- Threads

- Kritische Abschnitte

- Koordinierung

Verteilte Ausführung

Versionsverwaltung mit Git

- Grundlagen

- Branches

- Konflikte

- Git in Eclipse

Java

Collections & Maps

- Package: `java.util`
- Gemeinsame Schnittstelle: `Collection`
- Datenstrukturen
 - Menge
 - Schnittstelle: `Set`
 - Implementierungen: `HashSet`, `TreeSet`, ...
 - Liste
 - Schnittstelle: `List`
 - Implementierungen: `LinkedList`, `ArrayList`, ...
 - Warteschlange
 - Schnittstelle: `Queue`
 - Implementierungen: `PriorityQueue`, `LinkedBlockingQueue`, ...
- Tutorial



The Java Tutorials, Trail: Collections

<http://docs.oracle.com/javase/tutorial/collections/index.html>

- Verfügbare Algorithmen (Beispiele)
 - Maximums- (`max()`) bzw. Minimumsbestimmung (`min()`)
 - Sortieren (`sort()`)
 - Überprüfung auf Existenz gemeinsamer Elemente (`disjoint()`)
 - Erzeugung zufälliger Permutationen (`shuffle()`)
- Beispiel
 - Implementierung

```
Integer[] values = { 1, 2, 3, 4, 5, 6 };  
  
List<Integer> list = new ArrayList<Integer>(values.length);  
Collections.addAll(list, values);  
  
System.out.println("Before: " + list);  
Collections.shuffle(list);  
System.out.println("After: " + list);
```

- Ausgabe eines Testlaufs

```
Before: [1, 2, 3, 4, 5, 6]  
After:  [4, 2, 1, 6, 5, 3]
```

- Allgemeine Schnittstelle für Datenstrukturen zur Verwaltung von Schlüssel-Wert-Paaren
- Eigenschaften
 - Maximal ein Wert pro Schlüssel (→ keine Duplikate)
 - Interner Aufbau bestimmt durch gewählte Implementierung
 - HashMap
 - TreeMap
 - ...
- Beispiel

```
Map<String, Integer> telBook = new HashMap<String, Integer>();  
telBook.put("Alice", 123456789);  
telBook.put("Bob" , 987654321);  
[...]
```

```
Integer aliceNumber = telBook.get("Alice");  
System.out.println("Alice's number: " + aliceNumber);
```

Java

Threads

Variante 1: Unterklasse von `java.lang.Thread`

■ Vorgehensweise

1. Unterklasse von `Thread` erstellen
2. `run()`-Methode überschreiben
3. Instanz der neuen Klasse erzeugen
4. An dieser Instanz die `start()`-Methode aufrufen

■ Beispiel

```
class VSThreadTest extends Thread {  
    @Override  
    public void run() {  
        System.out.println("Test");  
    }  
}
```

```
Thread test = new VSThreadTest();  
test.start();
```

Variante 2: Implementieren von `java.lang.Runnable`

■ Vorgehensweise

1. `run()`-Methode der `Runnable`-Schnittstelle implementieren
2. `Runnable`-Objekt erstellen
3. Instanz von `Thread` mit Hilfe des `Runnable`-Objekts erzeugen
4. Am neuen `Thread`-Objekt die `start()`-Methode aufrufen

■ Beispiel

```
class VSRunnableTest implements Runnable {  
    @Override  
    public void run() {  
        System.out.println("Test");  
    }  
}
```

```
Runnable test = new VSRunnableTest();  
Thread thread = new Thread(test);  
thread.start();
```

Variante 3: Java Lambda Ausdrücke [seit Java 8]

■ Vorgehensweise:

1. Erzeugung von `Thread`-Instanz und Beschreibung der `run()`-Methode mittels Lambda
2. Am neuen `Thread`-Objekt die `start()`-Methode aufrufen

■ **Einschränkung:** Kein Zustand (z.B. globale Variablen) möglich

■ Beispiel

```
class VSLambdaTest {
    private int x = 10;

    public void lambdaTest() {
        Thread test = new Thread(() -> {
            System.out.println("Test " + this.x);
        });
        test.start();
    }
}
```

- Ausführung für einen bestimmten Zeitraum aussetzen

- Mittels sleep()-Methoden

```
static void sleep(long millis) throws InterruptedException;
```

```
static void sleep(long millis, int nanos) throws InterruptedException;
```

- Legt aktuellen Thread für millis Millisekunden (und nanos Nanosekunden) „schlafen“

- **Achtung:**

- Es ist nicht garantiert, dass der Thread exakt nach der angegebenen Zeit seine Ausführung fortsetzt
- Von Präzision der Systemzeit/des Schedulers abhängig (Linux: 1ms, Windows (default): 15ms)

- Synchronisierung mit anderen Threads (siehe Kapitel „Koordinierung“ ab Folie 17)

■ Regulär

- return aus der run()-Methode
- Ende der run()-Methode

■ Abbruch nach expliziter Anweisung

- Aufruf der interrupt()-Methode (durch einen anderen Thread)

```
public void interrupt();
```

- Führt zu

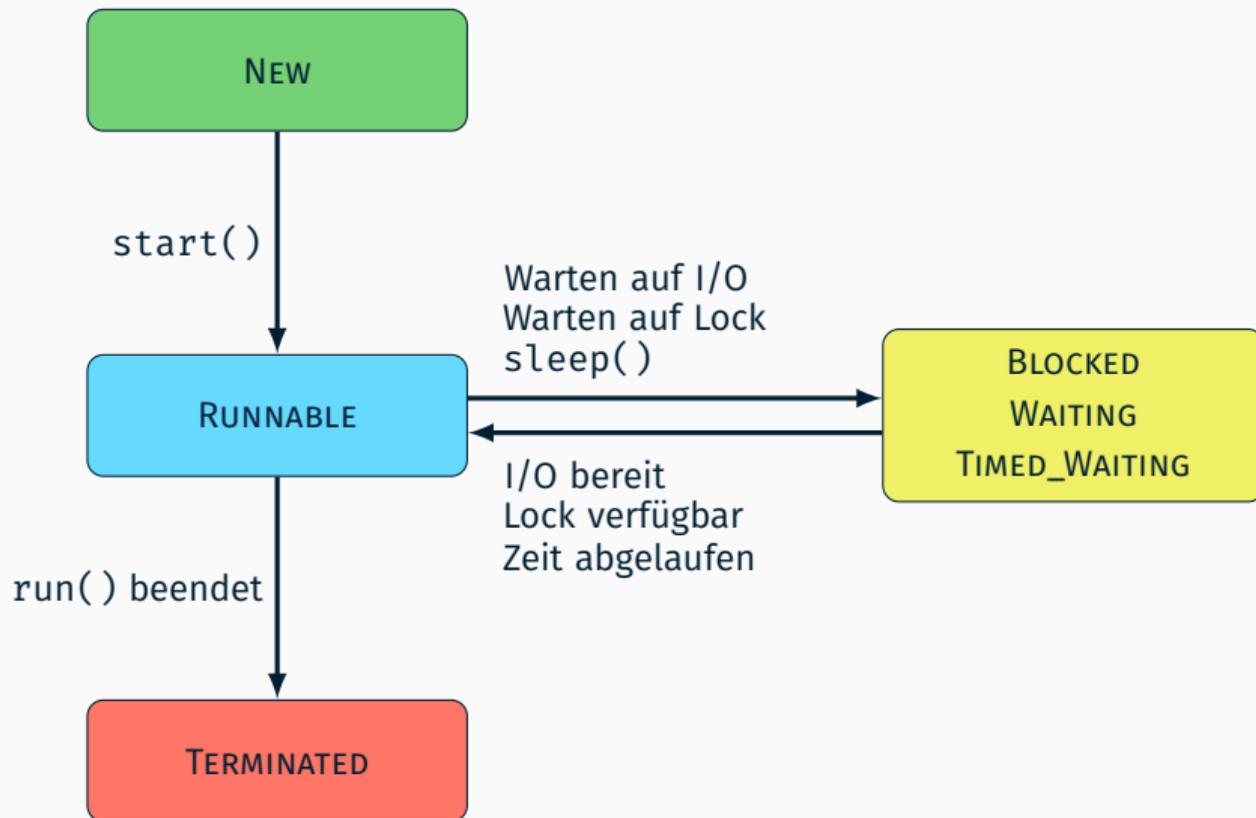
- einer InterruptedException, falls sich der Thread gerade in einer unterbrechbaren blockierenden Operation befindet
- einer ClosedByInterruptException, falls sich der Thread gerade in einer unterbrechbaren I/O-Operation befindet
- dem Setzen einer Interrupt-Status-Variable, die mit isInterrupted() abgefragt werden kann, sonst.

Wichtig: Threads können sich in Java aktiv der Unterbrechung *widersetzen* (z.B. Fangen & Ignorieren von InterruptedExceptions). Man kann sie von außerhalb also nicht zum Beenden *zwingen*.

■ Auf die Terminierung eines Threads warten mittels join()-Methode

```
public void join() throws InterruptedException;
```

Thread-Zustände in Java



Java

Kritische Abschnitte

```
public class VSCounter implements Runnable {
    public int a = 0;

    public void run() {
        for(int i = 0; i < 1000000; i++) {
            a = a + 1;
        }
    }

    public static void main(String[] args) throws Exception {
        VSCounter value = new VSCounter();
        Thread t1 = new Thread(value);
        Thread t2 = new Thread(value);

        t1.start();
        t2.start();

        t1.join();
        t2.join();
        System.out.println("Expected a = 2000000, " +
            "but a = " + value.a);
    }
}
```

- Ergebnisse einiger Durchläufe: 1 732 744, 1 378 075, 1 506 836
- Was passiert, wenn $a = a + 1$ ausgeführt wird?

```
LOAD a into Register
ADD 1 to Register
STORE Register into a
```

- Mögliche Verzahnung wenn zwei Threads T_1 und T_2 beteiligt sind

0. $a = 0$;

1. T_1 -LOAD: $a = 0$, $Reg_1 = 0$

2. T_2 -LOAD: $a = 0$, $Reg_2 = 0$

3. T_1 -ADD: $a = 0$, $Reg_1 = 1$

4. T_1 -STORE: $a = 1$, $Reg_1 = 1$

5. T_2 -ADD: $a = 1$, $Reg_2 = 1$

6. T_2 -STORE: $a = 1$, $Reg_2 = 1$

⇒ Die drei Operationen müssen jeweils **atomar** ausgeführt werden!

Synchronisieren ist notwendig, falls Atomizität erforderlich

1. Der Aufruf einer (komplexen) Methode muss atomar erfolgen
 - Eine Methode enthält mehrere Operationen, die auf einem konsistenten Zustand arbeiten müssen
 - Beispiele:
 - „a = a + 1“
 - Listen-Operationen (add(), remove(),...)
2. Zusammenhängende Methodenaufrufe müssen atomar erfolgen
 - Methodenfolge muss auf einem konsistenten Zustand arbeiten
 - Beispiel:

```
List list = new LinkedList();  
[...]  
int lastObjectIndex = list.size() - 1;  
Object lastObject = list.get(lastObjectIndex);
```

■ Standardansatz in Java

- Kennzeichnung eines kritischen Abschnitts mittels `synchronized`-Block
- Verknüpfung eines kritischen Abschnitts mit einem *Sperrobjekt*
- Ein Sperrobjekt kann nur von jeweils einem Thread gehalten werden

```
public void foo() {  
    [...] // unkritische Operationen  
    synchronized(<Sperrobjekt>) {  
        [...] // kritischer Abschnitt  
    }  
    [...] // unkritische Operationen  
}
```

■ Hinweise

- Jedes `java.lang.Object` kann als Sperrobjekt dienen
- Ein Thread kann dasselbe Sperrobjekt mehrfach halten (rekursive Sperre)

■ Mögliche Lösung für das Zähler-Beispiel

```
synchronized(this) { a = a + 1; }
```

■ Alternativen: Semaphore, ReentrantLock

- Ersatzschreibweise für einen methodenweiten `synchronized`-Block
- Sperrobjekt
 - Statische Methoden: `Class`-Objekt der entsprechenden Klasse
 - Sonst: `this`

```
class VExample {  
    synchronized public void foo() {  
        [...] // kritischer Abschnitt  
    }  
    public void bar() {  
        synchronized(this) {  
            [...] // kritischer Abschnitt  
        }  
    }  
}
```

- Beachte
 - Alle `synchronized`-Methoden einer Klasse nutzen dasselbe Sperrobjekt
 - Ansatz nur sinnvoll, falls Methoden tatsächlich in Konflikt stehen

- Klasse `java.util.Collections`
 - Statische Wrapper-Methoden für `Collection`-Objekte
 - Synchronisation kompletter Datenstrukturen

- Methoden

```
static <T> List<T> synchronizedList(List<T> list);  
static <K,V> Map<K,V> synchronizedMap(Map<K,V> map);  
static <T> Set<T> synchronizedSet(Set<T> set);  
[...]
```

- Beispiel

```
List<String> list = new LinkedList<String>();  
List<String> syncList = Collections.synchronizedList(list);
```

- Beachte

- Synchronisiert **alle** Zugriffe auf eine Datenstruktur
- Kein Schutz von zusammenhängenden Methodenaufrufen

■ Ansatz

- Ersatz-Klassen für problematische Datentypen
- Atomare Varianten häufig verwendeter Operationen
- Operation für atomares *Compare-and-Swap* (CAS)

■ Verfügbare Klassen

- Versionen für primitive Datentypen: `Atomic{Boolean,Integer,Long}`
- Arrays: `AtomicIntegerArray`, `AtomicLongArray`
- Referenzen: `AtomicReference`, `AtomicReferenceArray`
- ...

■ Beispiel

```
AtomicInteger ai = new AtomicInteger(47);
int newValueA = ai.incrementAndGet();
int newValueB = ai.getAndIncrement();
int oldValue = ai.getAndSet(4);
boolean success = ai.compareAndSet(oldValue, 7);
```

Java

Koordinierung

■ Problemstellung

- Rollenverteilung zwischen Threads (z. B. Produzent/Konsument)
 - Threads müssen sich abstimmen, um eine gemeinsame Aufgabe zu lösen
- Mechanismen zur Koordinierung erforderlich

■ Standardansatz in Java

- Ein Thread wartet darauf, dass ein Ereignis eintritt
- Der Thread wird mittels einer *Synchronisationsvariable* benachrichtigt

■ Hinweise

- Jedes `java.lang.Object` kann als Synchronisationsvariable dienen
- Um andere Threads per Synchronisationsvariable zu benachrichtigen, muss ein Thread innerhalb eines `synchronized`-Blocks dieser Variable sein

■ Methoden

`wait()` Auf eine Benachrichtigung warten

`notify()` Benachrichtigung an **einen** wartenden Thread senden

`notifyAll()` Benachrichtigung an **alle** wartenden Threads senden

■ Variablen

```
Object syncObject = new Object(); // Synchronisationsvariable
boolean flag = false;             // Ereignis-Flag
```

■ Auf Erfüllung der Bedingung wartender Thread

```
synchronized(syncObject) {
    while(!flag) {
        syncObject.wait();
    }
}
```

■ Bedingung erfüllender Thread

```
synchronized(syncObject) {
    flag = true;
    syncObject.notify();
}
```

Verteilte Ausführung

■ Kompilieren von Java-Programmen

```
> javac -cp 'lib1.jar:libs/*' -d bin File1.java ...
```

- Klassenpfad (-cp) muss verwendete Bibliotheken beinhalten
 - Besteht aus jar-Dateien und Ordnern mit class-Dateien
 - Platzhalter * expandiert zu allen .jar-Dateien im jeweiligen Ordner
 - Pfade durch „:“ getrennt
- Ausgabeverzeichnis -d bin für kompilierte class-Dateien
- Quellcodedateien übergeben

■ Ausführen von Java-Programmen

```
> java -cp 'bin:lib1.jar:libs/*' [-Dparam=value] package.name.Entrypoint [args ...]
```

- Klassenpfad um Ausgabeverzeichnis für kompilierte Klassen ergänzen
- Systemeigenschaften mit -Dparam=value übergeben
 - Abfrage per `System.getProperty("param", "default");`
- Ausführung startet in der Klasse `package.name.Entrypoint`
- Restliche Parameter werden an das Java-Programm übergeben

■ „printf“-Debugging

- An unterschiedlichen Stellen im Programm Debugausgaben erzeugen
 - Zuordnung von Ausgabe zu Programmzeilen sollte möglich sein
 - Bei großen Ausgabemengen in Dateien umleiten
 - Ausgaben mit Zeitstempeln versehen
- Achtung:** Uhren der Rechner können im verteilten Fall voneinander abweichen

Wichtig: Ausgaben verändern ggf. Programmverhalten (*I/O ist langsam!*)

■ Debugger

- Einzelne(n) Java-Prozess(e) im Debugger starten
- Restliche Prozesse normal starten

Wichtig: Pausieren im Debugger hält nur den zugehörigen Prozess an. Restliche Prozesse laufen normal weiter.

→ **Gefahr von unerwartetem Verhalten durch Timeouts**

■ Läuft **überall** der aktuelle Programmcode?

- Protokoll für sichere Kommunikation über unsichere Netzwerke
 - SSH-Clients kommunizieren mit SSH-Servern über TCP (meist Port 22)
 - Public-Key-Verfahren für Verschlüsselung und Authentifizierung
- Anwendungen
 - Zugriff auf Rechner `host` unter Benutzernamen `user`

```
> ssh [<user>@]<host>
```

Hinweis: Innerhalb des CIP-Pool-Netzes sind einfache Hostnamen wie `cip2a0` ausreichend. Ansonsten muss der **Domänenname** mit angegeben werden, z. B. `cip2a0.cip.cs.fau.de`.

- Befehl `cmd` auf Rechner `host` ausführen

```
> ssh [<user>@]<host> <cmd>
```

- Authentifizierung mit SSH-Schlüssel gegenüber dem entfernten Rechner

```
> ssh [-i <ssh-key>] [<user>@]<host>
```

- Standard: Verwendung von SSH-Schlüssel unter `~/.ssh/id_rsa`
- Erstellung des Keys mittels `ssh-keygen`
- Übermittlung an entfernten Rechner am Besten mit `ssh-copy-id`

- Kopieren von Dateien zwischen Rechnern

```
> scp <path_src> <path_dst>
```

Für entfernte Pfade: [`<user>@<host>:<path_remote>`], Beispiele:

```
> scp cip2a0:/tmp/srcfile .
> scp /tmp/srcfile user@cip2a0:      # Ziel: Home von user
> scp -r cip2a0:srcdir cip2a0:/tmp  # Rekursiv, Ordner kopieren
```

- **Hinweis:** Die Verzeichnisse `/home` und `/proj` auf CIP-Pool-Rechnern werden per NFS (Network File System) bereitgestellt. Dadurch enthalten diese auf allen Rechner dieselben Dateien

```
> scp README faui00a:
> ssh faui00b cat README
```

■ Automatisieren häufiger Vorgänge

- Skript zum Starten der Anwendung (Dateiname: start-server.sh)

```
#!/bin/bash
echo "Starte Anwendung mit Parametern $@"
java -cp <classpath> vs.queue.VSQueueServer "$@"
```

- Skript ausführen

```
> chmod +x start-server.sh # einmalig als ausfuehrbar markieren
> ./start-server.sh param1 param2 ...
Starte Anwendung mit Parametern param1 param2 ...
```

■ Bash-Skripte debuggen

- Hinzufügen von echo-Anweisungen
- Starten mit bash -x

```
> bash -x start-server.sh param1 param2 ...
```

■ Wiki / Tutorialsammlung



The Bash Hackers Wiki

<http://wiki.bash-hackers.org/start>

- Aus- und wieder einhängbare Terminals
- Programme laufen auch bei getrennter Sitzung weiter
- Verwendung:
 - Starten eines Screens:

```
> screen
```

Aushängen (*detach*) eines Screens mittels 'Ctrl+a d'

- Auflisten aller laufenden Sitzungen

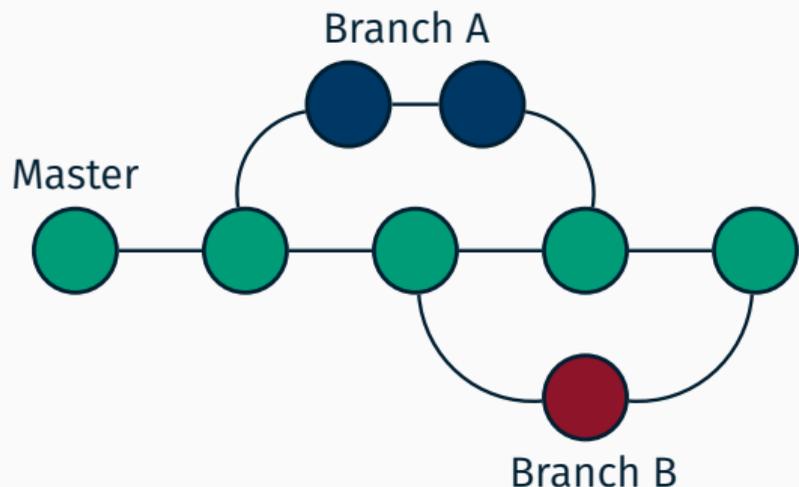
```
> screen -ls
There are screens on:
16656.pts-145.faii48f (25.10.2019 12:10:06) (Attached)
16457.pts-123.faii48f (25.10.2019 12:27:59) (Attached)
2 Sockets in /var/run/screen/S-lawniczak.
```

- Bestimmte Sitzung fortsetzen

```
> screen -dr 16457.pts-123.faii48f
```

Alternative: tmux

- Vorteile eines Versionskontrollsystems
 - Ermöglicht Zusammenarbeit mit mehreren Entwicklern
 - Einfaches Zusammenführen von Code und Erkennen von Konflikten
 - Parallele Entwicklung mehrerer Features
 - Fehlersuche uvm. durch „zurückspringen“ zu alten Versionen
- Weit verbreitet und öffentliche Hosting Plattformen (z.B. Gitlab, GitHub)



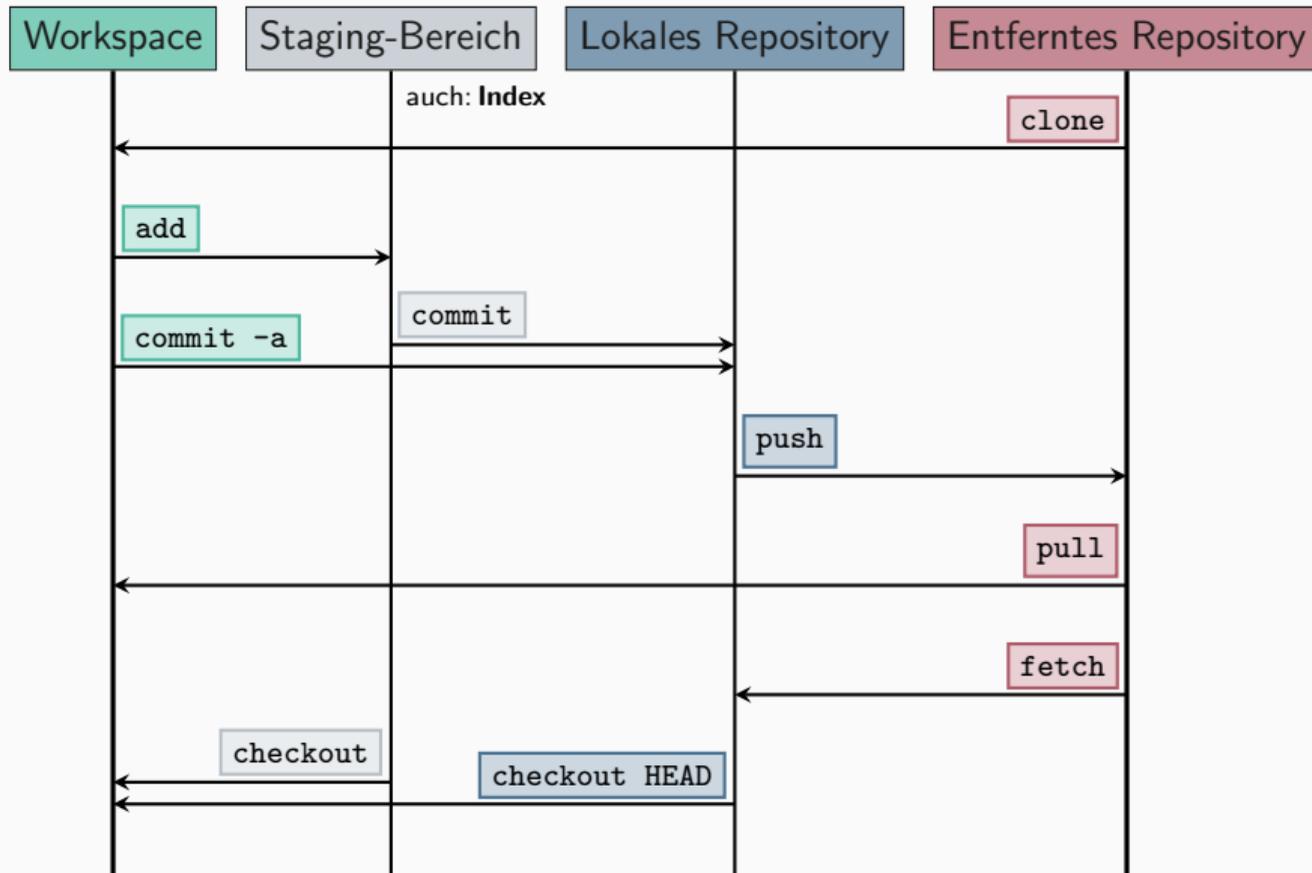
- Übungsaufgaben sollten im Git bearbeitet werden
- Benötigt ein Benutzerkonto bei <https://gitlab.cs.fau.de>
 - Konto werden automatisch mit dem IdM Single-Sign-On verknüpft
→ "Sign-in with *FAU Single Sign-On*"
 - Öffentliche(n) SSH-Schlüssel hinzufügen:
 - Oben rechts auf das Profil-Logo und auf „Profile Settings“ klicken
 - Reiter „SSH Keys“ auswählen
 - Einen oder mehrere SSH-Schlüssel hinzufügen
(siehe auch: <https://gitlab.cs.fau.de/help/ssh/README>)
- Repositories werden von uns **anhand der Übungsgruppen** erstellt.
Jeder Gruppenteilnehmer erhält automatisch Zugriff zum Repository seiner Gruppe.



Versionsverwaltung mit Git

Grundlagen

Überblick über den Git-Arbeitsablauf



- Erstellen einer **lokalen** Arbeitskopie über ein **entferntes** Repository

- Befehl: `> git clone <URL>`

- Beispiel: `git clone` über SSH (SSH-Schlüssel nötig!)

```
> git clone git@gitlab.cs.fau.de:i4-exercise/vs/ss21/vs-gruppe-42.git
```

(URL des GitLab-Repository steht auf der jeweiligen Projektübersichtsseite)

```
> git clone git@gitlab.cs.fau.de:i4-exercise/vs/ss21/vs-gruppe-42.git
Cloning into 'vs-gruppe-42'...
X11 forwarding request failed
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 1), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (6/6), done.
Resolving deltas: 100% (1/1), done.

> ls vs-gruppe-42/
README.md
```

- Dateien werden zunächst nur dem Staging-Bereich hinzugefügt oder davon entfernt
 - Es wird nur der **aktuelle** Zustand hinzugefügt
 - Änderungen haben erst beim nächsten Commit Auswirkungen auf das Repository
 - Einzelne Änderungen durch Option `-p` bzw. `--patch` auswählbar
- Änderung(en) zu Staging-Bereich hinzufügen (bzw. Datei(en) entfernen)

```
> git add [-p] <file(s)-to-add>  
> git rm <file(s)-to-remove>
```

- Änderung(en) aus Staging-Bereich entfernen

```
> git reset HEAD [-p] <file(s)-to-reset>
```

- Änderung(en) im **Workspace** verwerfen

```
> git checkout -- [-p] <file(s)-to-checkout>
```

Seit Version 2.23 gibt es 'git restore', das 'git reset HEAD' und 'git checkout --' ersetzt.
Laden der neuen git Version im CIP: 'load module git'

■ Auswirkungen des nächsten Commits überprüfen

```
> git status
```

```
On branch master
```

```
Your branch is up to date with 'origin/master'.
```

```
Changes to be committed:
```

```
(use "git reset HEAD <file>..." to unstage)
```

```
new file:   src/Application.java
```

```
Changes not staged for commit:
```

```
(use "git add <file>..." to update what will be committed)
```

```
(use "git checkout -- <file>..." to discard changes in working directory)
```

```
modified:   README.md
```

```
Untracked files:
```

```
(use "git add <file>..." to include in what will be committed)
```

```
Makefile
```

Unterschiedliche Ausprägungen von diff:

- *Standardverhalten*: Diff zwischen Workspace und Staging-Bereich

```
> git diff [<filename>]
```

- Diff zwischen Staging-Bereich und aktuellem Commit

```
> git diff --cached [<filename>]
```

- Diff zwischen Workspace und einem bestimmten Commit

```
> git diff <commit> [<filename>]
```

- Unterschiede zu Dateien in einem Remote-Branch

```
> git diff <local_branch> <remote_branch>
```

Zum Beispiel:

Unterschied von lokalem Branch 'master' zu Zustand von 'master' im entfernten Repository
(local_branch := master und remote_branch := origin/master)

→  Vorheriges git fetch (siehe Folie 2-13) ratsam.

- Änderungen vom Staging-Bereich ins lokale Repository übernehmen

```
> git commit [<file(s)-to-commit>]
```

Nützliche Parameter:

- a Alle modifizierten Dateien übernehmen
- m <message> message als Commit-Nachricht verwenden
- amend Vorherigen Commit modifizieren

- Commits vom lokalen in das **entfernte** Repository einprüfen

```
> git push [[remote_name] [branch_name]]
```

Wenn das entfernte Repository **zusätzliche, noch nicht lokal vorhandene** Commits enthält, muss das lokale Repository **zuerst** aktualisiert werden.

- Zustand aus entferntem Repository holen und integrieren

```
> git pull [[remote_name] [branch_name]]
```

Eventuell Konfliktauflösung notwendig, siehe „Konflikte“ ab Folie 19

- Aktualisierung der lokalen Sicht auf das entfernte Repository

```
> git fetch --all
```

- Änderungen werden nur gelesen, noch nicht eingespielt
- Ermöglicht Vergleich von lokalem und entferntem Stand, z.B.

```
> git diff master origin/master
```

■ Betrachten von Commits im lokalen Repository

```
> git log

commit f8ceebed8d581cab736350c055b072db148987cd
Author: Laura Lawniczak <lawniczak@cs.fau.de>
Date:   Fri Oct 25 13:11:11 2019 +0200

Add initial README file

[...]
```

- Aufbau: Commit-ID, Autor, Datum, Commit-Nachricht
- Ausgeben der Änderungen eines Commits: `> git log -p [<commit-id>]`

■ Graphische Aufarbeitung im Terminal

```
> tig [<file(s)-to-view-log-for>]
```

■ Git-GUIs mit graphischer Darstellung

- git-cola
- gitk

Kompilierte Dateien (z. B. `.class`-Dateien) sollten nicht ins Repository!

- Zu ignorierende Dateien in `.gitignore` eintragen

```
# Ignore class files
*.class
```

- Sollte in das Repository eingecheckt werden
- Greift nicht für bereits eingecheckte Dateien
→ ggf. die entsprechende Datei explizit mit `git rm <file>` löschen

- Lokale Änderungen inklusive ignorerter Dateien anzeigen

```
> git status --ignored

[...]
```

Ignored files:
(use "`git add -f <file>...`" to include in what will be committed)
application.class

- E-Mail-Adresse und Name für Commits festlegen

```
> git config --global user.email max@mustermann.de  
> git config --global user.name "Max Mustermann"
```

- Alle gesetzten Variablen ansehen

```
> git config --list  
  
user.name=Max Mustermann  
user.email=max@mustermann.de  
[...]
```

- Dokumentation: `man 1 git-config`

Versionsverwaltung mit Git

Branches

- Für jedes neue Feature wird üblicherweise ein neuer Branch erstellt

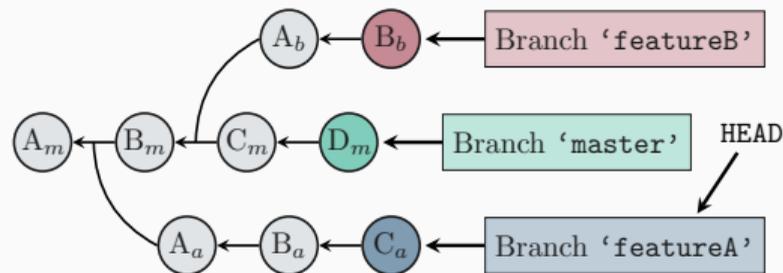
```
> git checkout -b <new_branch_name>
```

- Wechseln zwischen Branches (Workspace und Staging-Bereich bleiben erhalten)

```
> git checkout <branch_name>
```

- Anzeigen aller Branches (-a inkludiert entfernte Branches)

```
> git branch [-a]
master
* featureA
featureB
```



- Irgendwann müssen verschiedene Zweige vereint werden

- Prinzipiell zwei unterschiedliche Wege

- Klassischer Merge:

→ Mergen von <branch> in <other_branch>:

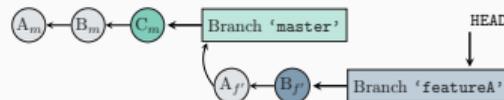
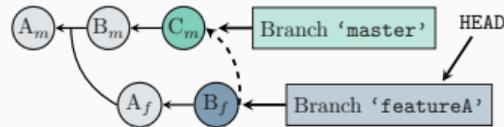
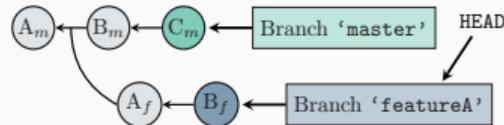
```
> git checkout <other_branch>
> git merge <branch>
```

- Einfacher Fall: *fast-forward merge*
- Fall mit eventuell notwendiger Konfliktauflösung: *3-way merge*

- Rebase:

```
> git checkout <other_branch>
> git rebase [-i] <branch>
```

- Interaktives Rebase (-i): Historie neu schreiben
-  Sollte nicht auf öffentlichem Branch angewendet werden



Versionsverwaltung mit Git

Konflikte

- Es gibt Konflikte, die git nicht selbstständig auflösen kann

```
> git pull
[...]
1b09b5d..39efa77  master -> origin/master
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.

> cat README.md
Das ist meine README Datei!

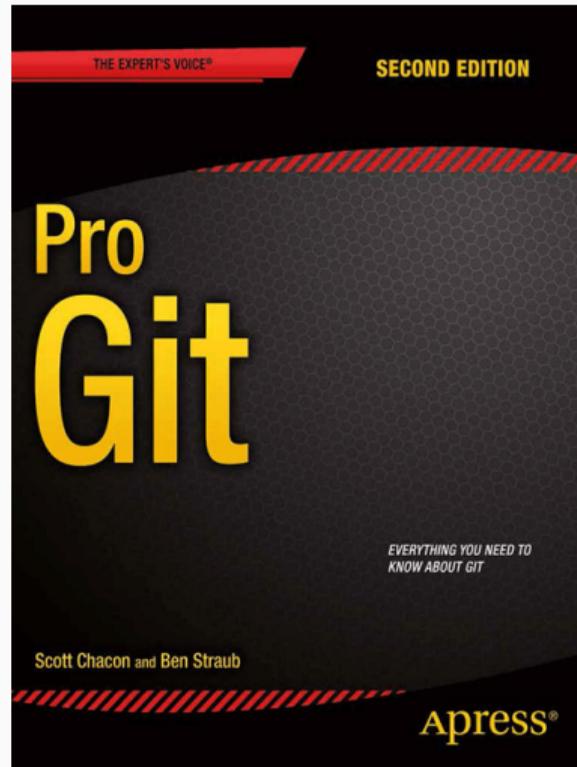
<<<<<<< HEAD
Meine neue Änderung
=====
Änderung aus dem gepullten Commit
>>>>>> 39efa77d814d4aebfecfd37da8d252cfc80091907
```

- Konflikt muss manuell gelöst werden und Ergebnis committed werden

```
> git add README.md
> git commit
```



<https://xkcd.com/1597/>



<https://git-scm.com/book/en/v2>

Versionsverwaltung mit Git

Git in Eclipse

- Eclipse enthält Unterstützung für Git
- Schritte zum Einrichten
 1. Lokale Kopie des Repositories erstellen (wenn nicht schon per 'git clone')
 - „File“ → „Import...“ → „Git“ → „Projects from Git“
 - Anschließend „Clone URI“ auswählen und URL aus Gitlab einfügen
 - Bei „Branch Selection“ auf weiter klicken
 - Bei „Local Destination“ ggf. **Pfad** anpassen
 - „Import using the New Project wizard“ auswählen
 2. Als Projekt in Eclipse einfügen
 - Neues „Java“ → „Java Project“ auswählen
 - **„Use default location“ deaktivieren**
 - **Pfad des lokalen Repositories eingeben**
 - ⇒ Eclipse erkennt das Git-Repository automatisch
 - Rest wie ohne Git
- Git-Befehle sind nach Rechtsklick auf das Projekt über das „Team“-Untermenü verfügbar