

Java

- Collections & Maps
- Threads
- Kritische Abschnitte
- Koordinierung

Verteilte Ausführung

Versionsverwaltung mit Git



Maps

java.util.Map<K,V>

- 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);
```



Collections

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



The Java Tutorials, Trail: Collections

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

Algorithmen-Bibliothek

java.util.Collections

- 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]
```



Java

- Collections & Maps
- Threads
- Kritische Abschnitte
- Koordinierung

Verteilte Ausführung

Versionsverwaltung mit Git



Erzeugung von Threads

Runnable

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 MRunnableTest implements Runnable {  
    @Override  
    public void run() {  
        System.out.println("Test");  
    }  
}  
  
Runnable test = new MRunnableTest();  
Thread thread = new Thread(test);  
thread.start();
```



Erzeugung von Threads

Thread

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 MWThreadTest extends Thread {  
    @Override  
    public void run() {  
        System.out.println("Test");  
    }  
}  
  
Thread test = new MWThreadTest();  
test.start();
```



Pausieren von Threads

`sleep()`, `yield()`

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 den 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

Ausführung auf unbestimmte Zeit aussetzen

Mittels `yield()`-Methode

```
static void yield();
```

- Aussetzen der eigenen Ausführung zugunsten anderer Threads
- Keine Informationen über die Dauer der Pause



- 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-Variablen, die mit isInterrupted() abgefragt werden kann, sonst.
- Auf die Terminierung eines Threads warten mittels join()-Methode

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



Identifizierung kritischer Abschnitte

Beispiel

```
public class MWCounter 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 {
        MWCounter value = new MWCounter();
        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);
    }
}
```



Java

Collections & Maps

Threads

Kritische Abschnitte

Koordinierung

Verteilte Ausführung

Versionsverwaltung mit Git



Identifizierung kritischer Abschnitte

Beispiel

- Ergebnisse einiger Durchläufe: 1732744, 1378075, 1506836
- 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 T₁ und T₂ beteiligt sind

0. a = 0;
1. T₁-LOAD: a = 0, Reg₁ = 0
2. T₂-LOAD: a = 0, Reg₂ = 0
3. T₁-ADD: a = 0, Reg₁ = 1
4. T₁-STORE: a = 1, Reg₁ = 1
5. T₂-ADD: a = 1, Reg₂ = 1
6. T₂-STORE: a = 1, Reg₂ = 1

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



Identifizierung kritischer Abschnitte

- Code, der zu jedem Zeitpunkt nur von einem einzigen Thread ausgeführt wird, muss nicht synchronisiert werden
- Synchronisieren nötig, 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);
```



Synchronisierte Methoden

- Ersatzschreibweise für einen methodenweiten synchronized-Block
 - Sperrobject
 - Statische Methoden: class-Objekt der entsprechenden Klasse
 - Sonst: this
- ```
class MWExample {
 synchronized public void foo() {
 [...] // kritischer Abschnitt
 }
 public void bar() {
 synchronized(this) {
 [...] // kritischer Abschnitt
 }
 }
}
```
- Beachte
    - Alle synchronized-Methoden einer Klasse nutzen dasselbe Sperrobject
    - Ansatz nur sinnvoll, falls Methoden tatsächlich in Konflikt stehen



## Definition kritischer Abschnitte

- Standardansatz in Java
  - Kennzeichnung eines kritischen Abschnitts mittels synchronized-Block
  - Verknüpfung eines kritischen Abschnitts mit einem *Sperrobject*
  - Ein Sperrobject kann nur von jeweils einem Thread gehalten werden

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

- Hinweise
  - Jedes java.lang.Object kann als Sperrobject dienen
  - Ein Thread kann dasselbe Sperrobject mehrfach halten (rekursive Sperre)
- Mögliche Lösung für das Zähler-Beispiel

```
synchronized(this) { a = a + 1; }
```
- Alternativen: Semaphore, ReentrantLock



## Synchronisierte Datenstrukturen

- 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
  - Löst Fall 1, jedoch nicht Fall 2 von Folie 2–13



## 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);
```



# Koordinierung in Java

## 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



## Java

Collections & Maps

Threads

Kritische Abschnitte

Koordinierung

## Verteilte Ausführung

## Versionsverwaltung mit Git



# Koordinierung in Java

## 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();
}
```



## Java

- Collections & Maps
- Threads
- Kritische Abschnitte
- Koordinierung

## Verteilte Ausführung

## Versionsverwaltung mit Git



## Verteiltes Debugging

- „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
- Debugger
  - Einzelne(n) Java-Prozess(e) im Debugger starten
  - Restliche Prozesse normal starten
  - Achtung:** 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?



## Kompilieren von Java-Programmen

```
> javac -cp lib1.jar:lib2.jar -d bin File1.java ...
```

- Klassenpfad (-cp) muss verwendete Bibliotheken beinhalten
  - Besteht aus jar-Dateien und Ordnern mit class-Dateien
  - Pfade durch „:“ getrennt
- Ausgabeverzeichnis -d bin für kompilierte class-Dateien
- Quellcodedateien übergeben

## Ausführen von Java-Programmen

```
> java -cp lib1.jar:lib2.jar:bin [-Dparam=value] 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 Entrypoint
- Restliche Parameter werden an das Java-Programm übergeben



## Secure Shell (SSH)

- 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 aktuellem Benutzernamen

```
> ssh <host>
```
  - Zugriff auf Rechner host unter Benutzernamen user

```
> ssh <user>@<host>
```
  - Befehl cmd auf Rechner host ausführen

```
> ssh <host> <cmd>
```
  - Authentifizierung mit SSH-Schlüssel gegenüber dem entfernten Rechner

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

    - Standard: Verwendung von SSH-Schlüssel unter ~/.ssh/id\_rsa
    - Öffentlicher Teil des Schlüssels (~/.ssh/id\_rsa.pub) muss auf entferntem Rechner in ~/.ssh/authorized\_keys eingetragen sein



## Secure Copy (SCP)

scp(1)

- Kopieren von Dateien zwischen Rechnern

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

Für entfernte Pfade: [<user>@]<host>:<path\_remote>, Beispiele:

```
> scp faui0ad:/tmp/srcfile .
> scp /tmp/srcfile user@faui0ad: # Ziel: Home von user
> scp -r faui0ad:srcdir faui0ad:/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
```

- **Hinweis:** Innerhalb des CIP-Pool-Netzes sind einfache Hostnamen wie 'faui00a' ausreichend. Ansonsten muss der **Domänenname** mit angegeben werden, z. B. 'faui00a.cs.fau.de'.



## Screen: Terminal-Multiplexer

screen(1)

- Programm zum Verwalten mehrerer virtueller Terminals
  - Erlaubt beliebiges Trennen und Fortsetzen von Sitzungen
  - Wichtige Tastatur-/Screen-Befehle
- |               |            |                                                 |
|---------------|------------|-------------------------------------------------|
| Ctrl+a c      | screen     | Erstelle neues Fenster und wechsle zu diesem    |
| Ctrl+a Ctrl+a | other      | Zwischen letzten aktiven Fenstern wechseln      |
| Ctrl+a <num>  | select     | Springe zu Fenster <num>                        |
| Ctrl+a "      | windowlist | Liste mit offenen Fenstern anzeigen             |
| Ctrl+a k      | kill       | Schließe aktuelles Fenster                      |
| Ctrl+a \      | quit       | Schließe alle Fenster und beende Screen-Instanz |
| Ctrl+a d      | detach     | Screen-Sitzung trennen                          |
| Ctrl+a [      | copy       | Kopiermodus zum Scrollen, Verlassen: <ESC>      |
| Ctrl+a ?      | help       | Tastaturbelegung mit Befehlen zeigen            |



## Bash-Skripte

bash(1)

- Automatisieren häufiger Vorgänge

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

```
#!/bin/bash
echo "Starte Anwendung mit Parametern $@"
java -cp <classpath> vsue.bank.VSBankServer "$@"
```

- 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>



## Screen: Terminal-Multiplexer

screen(1)

- Programme laufen auch bei getrennter Sitzung weiter

- Screen-Sitzung im aktuellen Terminal fortsetzen

(wird gegebenenfalls von anderem Terminal getrennt)

```
$ screen -dr
```

- Bei mehreren Screen-Sitzungen

- Auflisten laufender Sitzungen

```
$ screen -ls
There are screens on:
 16656.pts-145.faui48f (27.10.2017 15:10:06) (Attached)
 16457.pts-123.faui48f (27.10.2017 15:27:59) (Attached)
 2 Sockets in /var/run/screen/S-eischer.
```

- Bestimmte Sitzung fortsetzen

```
$ screen -dr 16457.pts-123.faui48f
```



- Screen-Konfiguration (.screenrc): Textdatei mit Befehlen

```
startup_message off # Keine Hilfeseite beim Start
defscrollback 1500 # Max. 1500 Zeilen puffern
Fenster starten
screen ssh faui04e java -cp <classpath> ReplicationServer 1
screen ssh faui04f java -cp <classpath> ReplicationServer 2
screen ssh faui04g java -cp <classpath> ReplicationServer 3
```

- Konfiguration beim Starten von Screen laden

```
$ screen -c screen-config.txt
```

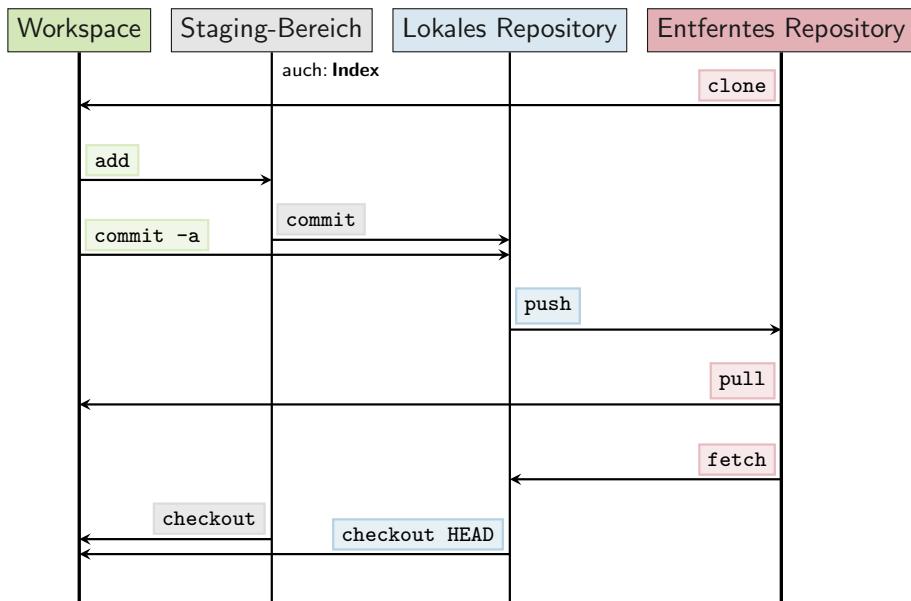
- Fernsteuern aus anderem Terminal oder Shell-Script

```
screen -X <Screen-Befehl> ...
$ screen -X screen ssh faui04g
```

- Dokumentation: man 1 screen



## Überblick über den Git-Arbeitsablauf



## Java

- Collections & Maps
- Threads
- Kritische Abschnitte
- Koordinierung

## Verteilte Ausführung

## Versionsverwaltung mit Git



## Einrichten eines Repository

git clone/git config

- Erstellen einer lokalen Arbeitskopie über ein entferntes Repository
  - Befehl: `> git clone <URL>`
  - Beispiel: `git clone` über SSH (SSH-Schlüssel nötig, siehe Folie 0-6)
 

```
> git clone git@gitlab.cs.fau.de:mustermann/vsue.git
```
  - URL des GitLab-Repository steht auf der jeweiligen Projektübersichtsseite
- Konfiguration
  - Befehl: `> git config`
  - E-Mail-Adresse und Name für Benutzer (`--global`) festlegen
 

```
> git config --global user.name "Max Mustermann"
> git config --global user.email max@mustermann.de
```
  - Ohne `--global` erfolgt die Konfiguration Repository-spezifisch
  - Dokumentation: `git config --help`
- Weitere Informationen zu Git: <https://git-scm.com/book/en/v2>



### Betrachten von Commits im lokalen Repository

- Befehl (nur Commit-Nachrichten): > git log

```
commit f8ceebcd8d581cab736350c055b072db148987cd
Author: Michael Eischer <eischer+gitlab@cs.fau.de>
Date: Wed Apr 24 11:11:11 2018 +0200
```

```
Add initial README file
```

```
[...]
```

- Ausgeben der Änderungen eines Commits > git log -p

### Git-GUIs mit graphischer Darstellung

- git-cola
- gitk
- gitg



# Änderungen überprüfen (1)

### Auswirkungen des nächsten Commits überprüfen: > git status

```
On branch master
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)
#
new file: README.md
#
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: application.java
#
Untracked files:
(use "git add <file>..." to include in what will be committed)
#
Makefile
```



### Dateien werden zunächst nur dem Index (→ Staging-Bereich) hinzugefügt oder davon entfernt

- Es wird nur der **aktuelle** Zustand hinzugefügt
- Änderungen werden erst beim nächsten Commit wirksam (d. h. in das lokale Repository übertragen) → siehe nächste Folie ff.
- Einzelne Änderungen durch Option -p bzw. --patch auswählbar

### Änderung(en) hinzufügen

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

### Änderung(en) aus Index entfernen

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

### Änderung(en) verwerfen

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

### Datei(en) entfernen

```
> git rm <file(s)-to-remove>
```



# Änderungen überprüfen (2)

### Unterschiedliche Ausprägungen

- Standardverhalten: Diff zwischen Workspace und Index

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

- Diff zwischen Index 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 u. remote\_branch := origin/master

→ ! Vorheriges git fetch (siehe Folie 2-38) ratsam.



- Mit commit übernommene Änderungen sind zunächst nur im **lokalen** Repository sichtbar
  - Kompletten Index oder nur bestimmte Datei(en) übernehmen  
 > `git commit [<file(s)-to-commit>]`
  - Alle modifizierten Dateien übernehmen  
 > `git commit -a`
  - Commit-Nachricht direkt per Kommandozeile übergeben  
 > `git commit -m <commit_message>`
  - Vorherigen Commit modifizieren  
 > `git commit --amend`
- Commits vom lokalen in das **entfernte** Repository einprüfen  
 > `git push [[remote_name] [branch_name]]`  
 → Lokales Repository muss vorher aktualisiert werden, wenn entferntes Repository weitere, noch nicht lokal vorhandene Commits enthält



## Konfliktbewältigung

- Konflikt feststellen  
 > `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`  
 <<<<< HEAD  
 TODO: Structure and fill this README.  
 =====  
 ## Synopsis  
  
 ## Installation  
 >>>> 39efa77d814d4aebfec37da8d252fc80091907
- Konflikt in Datei manuell auflösen und Ergebnis einprüfen  
 > `git add README.md`  
 > `git commit`

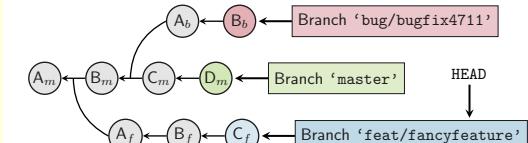


- Lokal
  - > `git checkout <branch>`  
 → Aktuellen Stand aus dem Zweig <branch> übernehmen  
 → Übernahme in den Workspace **und** in den Index  
 → Operation ist „safe“, verwirft also keine Änderungen
- Entfernt
  - > `git fetch --all`  
 → Aktualisierung der Remote-Tracking-Branches (refs/remotes/)
  - > `git pull [[remote_name] [branch_name]]`  
 → Zustand aus entferntem Repository holen und in aktuellen lokalen Branch integrieren (≈ git fetch, gefolgt von git merge)  
 → eventuell Konfliktauflösung notwendig, siehe nächste Folie



## Verzweigungen (1)

- Für jedes neue Feature wird üblicherweise ein neuer Branch erstellt
- Anzeigen aller (auch entfernter) Branches  
 > `git branch -a`  
 master  
 \* feat/fancyfeature  
 bug/bugfix4711  
 remotes/origin/master
- Neuen lokalen Branch erstellen (ausgehend vom gegenwärtigen HEAD)  
 > `git checkout -b <new_branch_name>`
- Branch wechseln
  - Wechsel in bereits existierenden Branch  
 > `git checkout <local_branch>`
  - Lokalen Branch basierend auf Remote-Branch erstellen und ausprüfen  
 > `git checkout -b <local_branch> <remote_branch>`



## Verzweigungen (2)

HEAD

- Bedeutung von HEAD



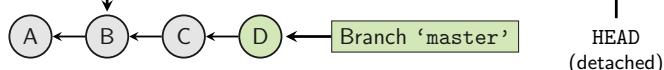
- Eine Art Zeiger auf den aktuell ausgewählten Branch

```
> cat .git/HEAD
ref: refs/heads/master
```

- Branch ist wiederum Zeiger auf aktuellsten Commit einer Verzweigung

- Losgelöster HEAD

- Git sorgt normalerweise dafür, dass der HEAD-Zeiger automatisch weitergerückt wird (d.h., immer auf den aktuellen Branch zeigt)
- Ausnahme: HEAD zeigt auf bestimmten Commit → detached HEAD



- „detached“ bedeutet, dass folgende Commits keinem (benannten) Branch zugeordnet sind und bei Wechsel zu anderem Commit/Branch verlorengehen



## Git in Eclipse

- Eclipse enthält Unterstützung für Git

- Schritte zum Einrichten

1. Lokale Kopie des Repository erstellen

- Entweder mit `git clone`, siehe Folie 2-32
- oder direkt in Eclipse
  - \* „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 und diesen **merken**
  - \* „Import using the New Project wizard“ auswählen

2. Als Projekt in Eclipse hinzufügen

- \* Neues „Java“ → „Java Project“ auswählen
- \* „**Use default location**“ deaktivieren
- \* **Pfad des lokalen Repository eingeben**  
→ Eclipse erkennt das Git-Repository anschließend automatisch
- \* Rest wie ohne Git

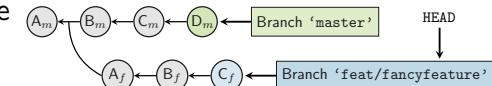
- Git-Befehle sind nach Rechtsklick auf das Projekt über das „Team“-Untermenü verfügbar



## Verzweigungen (3)

git merge vs. git rebase

- Irgendwann müssen verschiedene Zweige (wieder) vereint werden



- Prinzipiell zwei unterschiedliche Wege

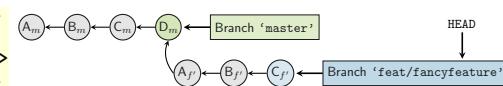
- Das klassische Mergen → Mergen von `<branch_src>` in `<branch_dst>`:

```
> git checkout <branch_dst>
> git merge <branch_src>
```

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

- Rebase

```
> git checkout <branch_dst>
> git rebase [-i] <branch_src>
```



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



## Git in Eclipse

Bekannte Probleme

- „Push to Upstream“ ist nicht verfügbar

- Tritt bei bislang leerem Repository auf
  - Dateien hinzufügen und einprüfen (`commit`)
  - Einmalig „Push Branch 'master'...“ mit Standardeinstellungen verwenden

