



Tag 1: Einführung in Git und GitLab, Git-Workflow im Team

17.06.2024, Daniel Krämer & Malte Fischer

© Copyright 2024 anderScore GmbH

HECKER
CONSULTING

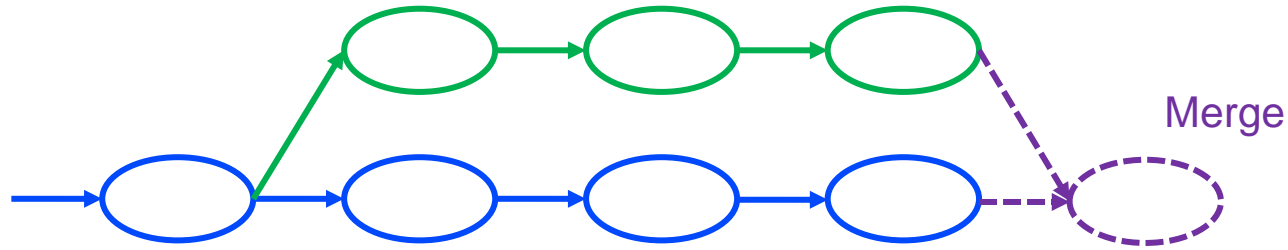
- **Tag 1 – Einführung in Git und GitLab, Git-Workflow im Team**
 - Einführung & Kursüberblick
 - Grundlagen von Git
 - Git Rebase und Merge-Strategien
 - Git Remote
 - Grundlagen von GitLab
 - Git-Workflow im Team
- **Tag 2 – Vertiefung Git-Workflow, CI/CD & GitLab CI**
 - Gitflow-Workflow
 - Tags, Releases & deren Verwaltung
 - GitLab-Runner
 - Einführung in GitLab CI/CD & gitlab.yml
- **Tag 3 – GitOps, Docker in der Entwicklung und Deployment-Strategien**
 - GitOps Grundlagen
 - Lokale Entwicklung mit Docker
 - Container/Docker-Registry
 - Erstellen von Release- und Tagged-Images
 - Möglichkeiten des Deployments & Verwaltung von Konfiguration
 - Abschlussübung & Diskussion

- **Tag 1 – Einführung in Git und GitLab, Git-Workflow im Team**
 - Einführung & Kursüberblick
 - Grundlagen von Git
 - Git Rebase und Merge-Strategien
 - Git Remote
 - Grundlagen von GitLab
 - Git-Workflow im Team
- **Tag 2 – Vertiefung Git-Workflow, CI/CD & GitLab CI**
 - Gitflow-Workflow
 - Tags, Releases & deren Verwaltung
 - GitLab-Runner
 - Einführung in GitLab CI/CD & gitlab.yml
- **Tag 3 – GitOps, Docker in der Entwicklung und Deployment-Strategien**
 - GitOps Grundlagen
 - Lokale Entwicklung mit Docker
 - Container/Docker-Registry
 - Erstellen von Release- und Tagged-Images
 - Möglichkeiten des Deployments & Verwaltung von Konfiguration
 - Abschlussübung & Diskussion

Grundlagen und Konzepte von **Merging in Git**

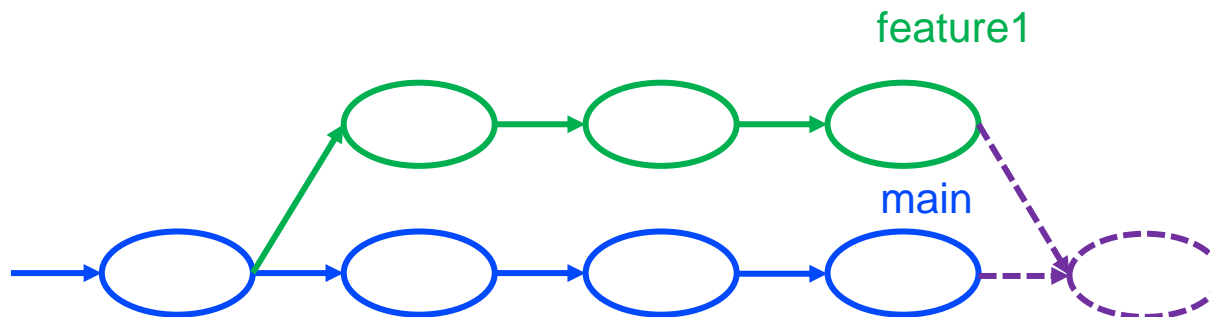
Inhalt

- Merging
 - Konzept
 - Ablauf
 - Konflikte
 - Zusätzliche Befehle
 - Strategien
- Rebase
 - Konzept
 - Befehle und Optionen
 - Konflikte
 - Anwendung
- Merge vs. Rebase



- Merging bildet Gegenstück zum Branching
- Ermöglicht Zusammenführung von Branches
- Beispielhafte Use Cases
 - Feature oder Hotfix abgeschlossen und soll in *main* integriert werden
 - Änderungen von Remote Branches einpflegen (mehr dazu später)

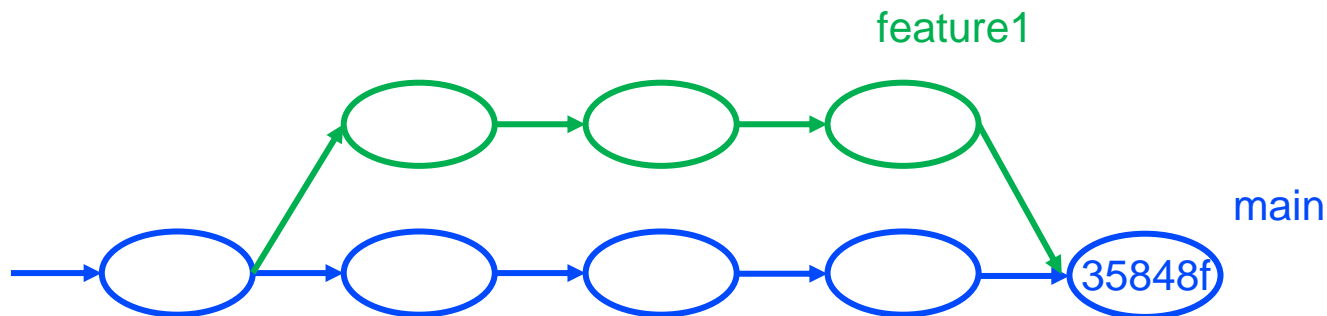
- Branch *branch* in aktiven Branch mergen
`git merge <branch>`
- Merging verändert immer nur aktiven Branch
- Erzeugt auf aktiven Branch einen Commit, um die Änderungen zu integrieren
- Merge-Commit hat *zwei* Vorfahren
- Gemergeter Branch wird *nicht* automatisch gelöscht



Beispiel

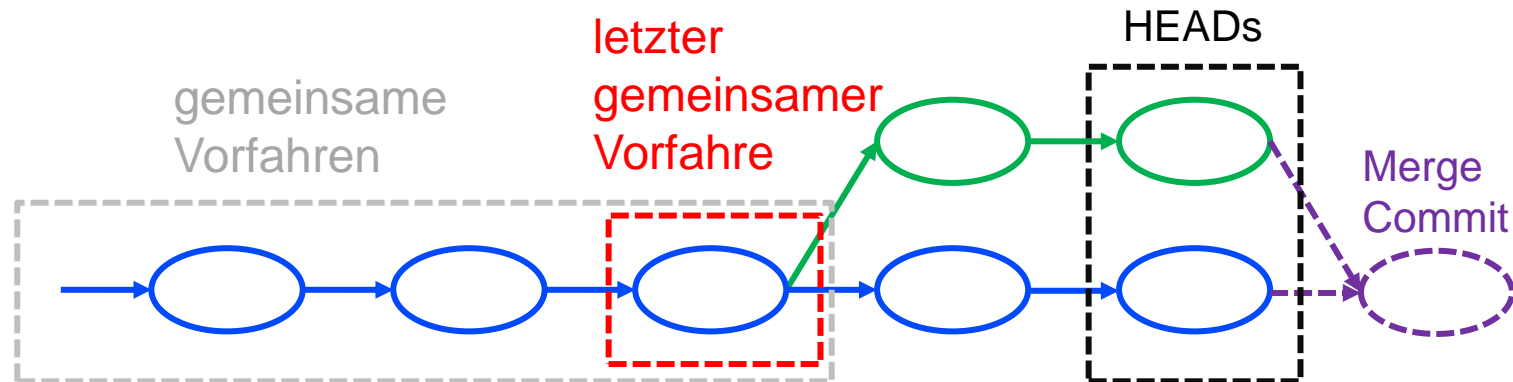
```
$ git merge feature1
Merge made by the 'ort' strategy.
 feature_file1.txt | 0
 feature_file2.txt | 0
 feature_file3.txt | 0
3 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 feature_file1.txt
create mode 100644 feature_file2.txt
create mode 100644 feature_file3.txt
```

```
$ git log --oneline
358548f (HEAD -> main) Merge branch 'feature1'
...
```



Ablauf eines Merges

- Merging von Dateien mit identischem Inhalt am einfachsten
 - Datei auf beiden Branches unverändert
 - Datei auf beiden Branches exakt gleich verändert
- Unterscheiden sich die Inhalte, benötigt man einen anderen Ansatz
- Git nutzt hierzu den *Three-Way-Merge* Algorithmus



- Zusammenfügen von Änderungen auf Dateiebene
- Aufteilung von Dateien in Sektionen
- Vergleich seit letztem gemeinsamem Vorfahren
 - Zeile ist auf beiden Branches unverändert → Zeile ins Ergebnis übernehmen
 - Zeile ist auf beiden Branches gleich verändert → Veränderung übernehmen
 - Zeile ist auf einem beiden Branches geändert, auf dem anderen nicht → Veränderung übernehmen
 - Zeile ist in beiden Branches unterschiedlich verändert → Konflikt, kein automatisches Auflösen möglich

- Git versucht beim zeilenbasierten Mergen zusammenhängende Bereiche zu erkennen

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Person person = (Person) o;
    return Objects.equals(firstName, person.firstName)
        && Objects.equals(lastName, person.lastName);
}

@Override
public int hashCode() {
    return Objects.hash(firstName, lastName);
}
```

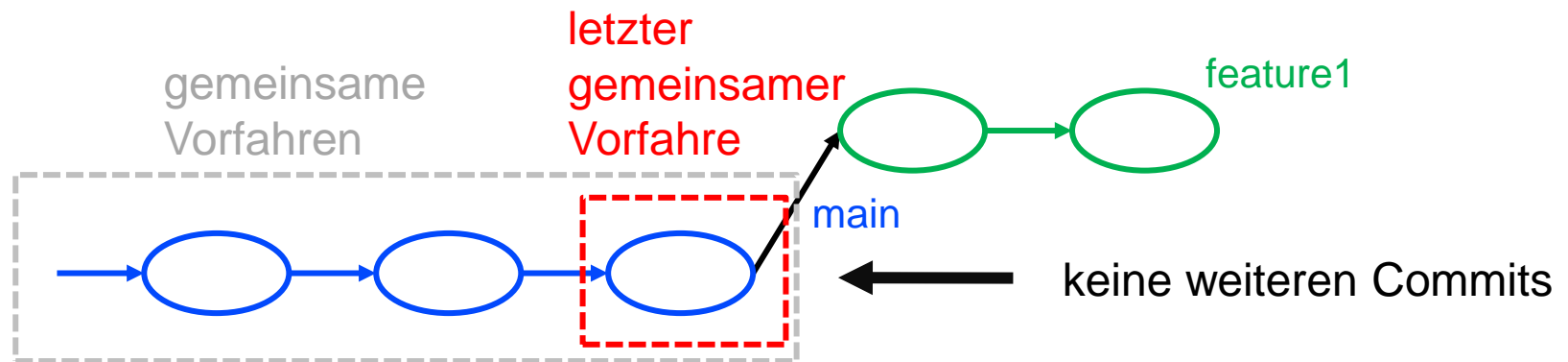
```
35 35
36 36
37 37
>> 38 38 //first comment
39 39
40 40
>> 41 41
42 42 //another comment
43 43
44 44
45 45
46 46
47 47
48 48
49 49
50 50
```

```
@Override
public boolean equals(Object o) {
    //first comment
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Person person = (Person) o;
    //another comment
    return Objects.equals(firstName, person.firstName)
        && Objects.equals(lastName, person.lastName);
}

@Override
public int hashCode() {
    return Objects.hash(firstName, lastName);
}
```

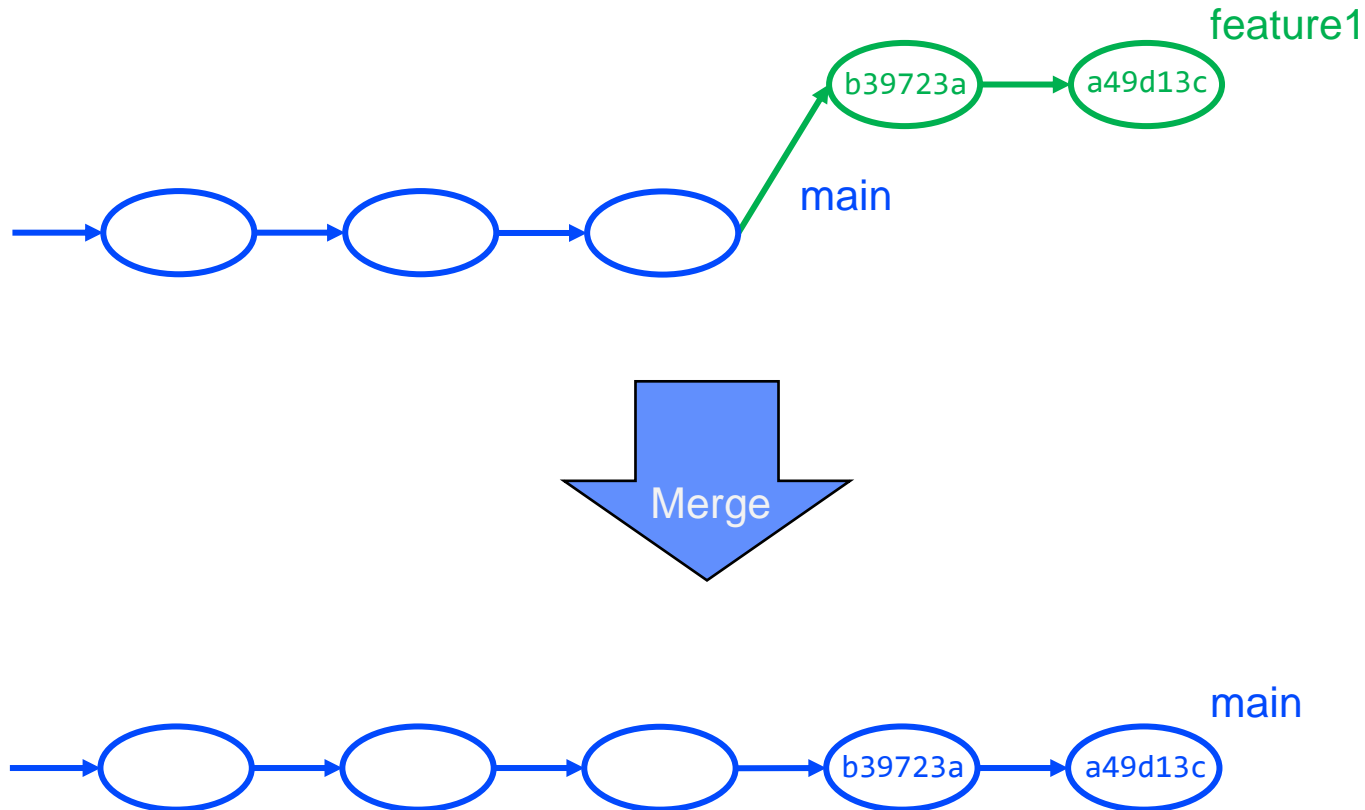
Fast-Forward Merge

- Möglich, falls keine neuen Commits auf Upstream nach Branch-Abzweigung



- Fast-Foward Merge erstellt keinen Merge-Commit
 - Kann mittels `--no-ff` trotzdem erstellt werden
- Commit des weiter fortgeschrittenen Branches werden als Commits des Zielbranches weiterbehandelt

Git – Merge



Aufgabe 8: Fast-Forward Merging

Feature 1 und 2 sind soweit abgeschlossen und die Änderungen sollen nun in den **main** Branch integriert werden.

1. Stellen Sie zunächst sicher, dass Sie sich auf dem **main** Branch befinden. Git folgt bis auf wenige Ausnahmen immer dem Prinzip, dass nur der aktive Branch verändert wird.
2. Mergen Sie den **feature1** Branch zurück in den **main** Branch. Weil nach der Abzweigung von **feature1** (Merge-Base) auf **main** keine neuen Commits erzeugt wurden, sollte Git den Merge als Fast-Forward Merge umsetzen können. Rückwirkend sieht es von den Logs so aus, als wären die **feature1** Commits direkt auf **main** getätigt worden.

Aufgabe 9: Merging

1. Mergen nun auch **feature2** zurück in den **main** Branch.
Nach der Merge-Base von **feature2** und **main** folgen auf **main** noch weitere Commits, daher kann hier kein Fast-Forward Merge durchgeführt werden.
2. Verifizieren Sie über die Logs, dass Git einen Merge-Commit erstellt hat, um **feature2** in **main** zu mergen.
3. Da **feature1** und **2** erfolgreich gemerged wurden, gelten die Branches als abgeschlossen und werden nicht mehr benötigt.
Löschen Sie **feature1** und **feature2**.

Git

Merge-Konflikte

Merge-Konflikte in Git

- Entstehen, wenn eine Zeile in zwei Branches auf verschiedene Art verändert wurde
- Erfordern manuelles Eingreifen
- Dateien ohne Konflikte werden automatisch gemerged und gestaged
- Entwickler muss nach Auflösen des Konfliktes den Merge-Commit selbst ausführen

Beispiel

```
$ git checkout -b conflict-feature  
Switched to a new branch 'conflict-feature'
```

```
$ echo "hello from feature branch" >  
conflict_file.txt
```

```
$ git add conflict_file.txt
```

```
$ git commit -m "Add file on feature"  
[conflict-feature 3984687] Add file on  
feature  
1 file changed, 1 insertion(+)  
create mode 100644 conflict_file.txt
```

```
$ git checkout main  
Switched to branch 'main'
```

```
$ echo "hello from main branch" >  
conflict_file.txt
```

```
$ git add conflict_file.txt
```

```
$ git commit -m "Add file on main"  
[main c9bb037] Add file on main  
1 file changed, 1 insertion(+)  
create mode 100644 conflict_file.txt
```

```
$ git merge conflict-feature  
Auto-merging conflict_file.txt
```

```
CONFLICT (add/add): Merge conflict in  
conflict_file.txt
```

```
Automatic merge failed; fix conflicts and  
then commit the result.
```

Auflösen von Konflikten im Editor

- Indikatoren werden in betroffener Datei hinzugefügt

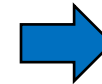
```
$ cat conflict_file.txt
<<<<<< HEAD
hello from main branch
=====
hello from feature branch
>>>>>> conflict-feature
```

- Die Originaldatei wird unter <Dateiname>.orig abgespeichert

Auflösen des Konfliktes

```
$ nano conflict_file.txt
```

```
<<<<<< HEAD
hello from main branch
=====
hello from feature branch
>>>>>> conflict-feature
```

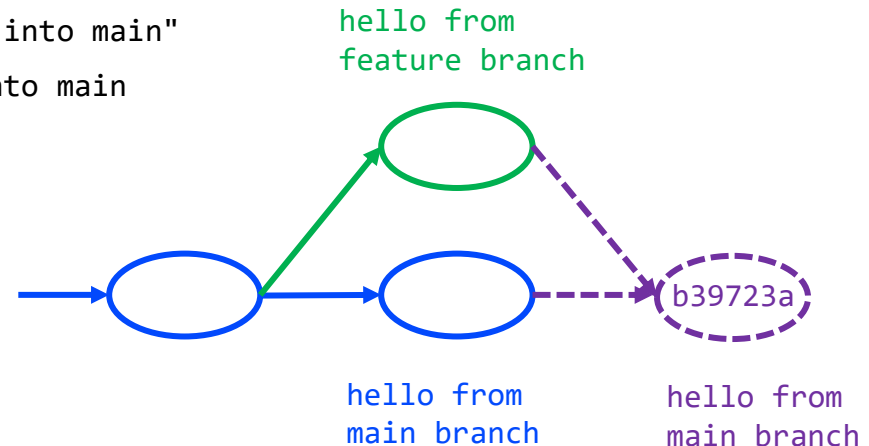


```
<<<<<< HEAD
hello from main branch
=====
hello from feature branch
>>>>>> conflict-feature
```

```
$ git add conflict_file.txt
```

```
$ git commit -m "Merge conflict-feature into main"
[main b39723a] Merge conflict-feature into main
```

```
$ cat conflict_file.txt
hello from main branch
```



Aufgabe 10: Merge-Konflikte

Im Nachfolgenden werden Sie beispielhaft einen Merge-Konflikt verursachen und auflösen.

1. Erstellen Sie einen von **main** neuen Branch **feature3** und wechseln Sie in diesen.
2. Legen Sie im Hauptordner eine Datei **merge_conflict_file.txt** mit dem Inhalt „Hello from feature3“ an und committen Sie diese.
3. Wechseln Sie zurück in den **main** Branch und legen Sie hier ebenfalls eine Datei **merge_conflict_file.txt** im Hauptordner an. In dieser Datei soll jedoch „Hello from main“ stehen. Committen Sie die Datei.

Aufgabe 10: Merge-Konflikte

Damit besitzen sowohl **main** als auch **feature3** eine Datei **conflict_file.txt**, beide jedoch mit unterschiedlichem Inhalt.

4. Versuchen Sie, **feature3** in **main** zu mergen.
Git sollte Ihnen anzeigen, dass es einen Merge-Konflikt bei der Datei **merge_conflict_file.txt** gab.
5. Öffnen Sie die Datei in einem Editor Ihrer Wahl. Mittels der Markierungen wird Ihnen angezeigt, wie die Branches divergieren.
6. Löschen Sie die den Bereich von **feature3** sowie sämtliche Markierungen, speichern Sie die Datei. Damit haben Sie entschieden, welche Änderungen verworfen bzw. übernommen werden.
7. Stagen Sie die Datei und erzeugen Sie nun selbst den Merge-Commit, um den Merge abzuschließen.

Aufgabe 11: Merge-Konflikte

Das vorherige Beispiel war ein sehr simpler Merge-Konflikt. Grundsätzlich führt der Fall, dass zwei Branches eine Datei auf verschiedene Art und Weise ändern, auch nicht zwingend zu einem Konflikt.

Ein Merge kann konfliktfrei funktionieren, jedoch semantische Konflikte produzieren, die Git nicht erkennt.

Beides schauen wir uns im Folgenden an.

1. Erstellen Sie auf dem **main** Branch eine Datei **sum.sh** mit dem Inhalt

```
#!/bin/bash
```

```
echo "Result: $(( $1 + $2 ))"
```

2. Comitten Sie die Datei.

Aufgabe 11: Merge-Konflikte

Als neues Feature soll nun dem Skript noch eine Ausgabe der übergebenen Argumente hinzugefügt werden.

3. Erstellen Sie einen neuen Branch **sum-feature** und wechseln Sie in diesen. Fügen Sie hier mit einem Editor eine Ausgabe **über** der Berechnung ein

```
#!/bin/bash
```

```
echo "Arguments are: $1 and $2"  
echo "Result: $(( $1 + $2 ))"
```

4. Speichern und committen Sie die Änderungen und wechseln Sie zurück auf den **main** Branch.

Aufgabe 11: Merge-Konflikte

Durch ein Versehen werden die geforderten Änderungen ebenfalls direkt auf den **main** Branch committet.

- Öffnen Sie die **sum.sh** Datei im Editor und eine Ausgabe **unter** der Berechnung ein

```
#!/bin/bash
```

```
echo "Result: $(( $1 + $2 ))"  
echo "Arguments were: $1 and $2"
```

- Speichern und committen Sie die Änderungen.
- Mergen Sie nun den **sum-feature** Branch in den **main** Branch. Git sollte hierbei alle Änderungen auflösen können, sodass kein Konflikt entsteht.

Aufgabe 11: Merge-Konflikte

Git merged Dateien immer zeilenbasiert. Dabei werden nicht nur simpel die Zeilennummern verglichen, sondern die intelligenteren Merge-Strategien erkennen neu eingefügte Zeilen.

Git wertet daher die jeweiligen Argumentausgaben als neu hinzugefügte Zeilen.

Diese erzeugen keinen Konflikt, denn wenn eine Zeile nur auf einem Branch existiert, auf dem anderen jedoch nicht, wird die Zeile einfach in das Ergebnis der Merges übernommen.

Semantisch sind die Änderungen jedoch falsch. In anderen Szenarien können auch tiefergreifende Fehler entstehen als eine einfache Textausgabe wie in diesem Fall.

8. (Optional) Führen Sie **sum.sh** aus.

Git – Merges

Zusätzliche Befehle

- Merging von mehr als zwei Branches ebenfalls möglich
- Im normalen Workflow aber meist Merge zwischen zwei Branches
- Liste aller in den aktiven Branch gemergeten Branches

```
git branch --merged
```

```
$ git branch --merged  
feature1  
* main
```

- Liste aller vom aktiven Branch abgezweigten, ungemergeten Branches

```
git branch --no-merged
```

```
$ git branch --no-merged  
feature2
```

- Branch gilt nach Merge oft als abgeschlossen
- Gemergeten Branch löschen mittels

`git branch -d <branch> (oder --delete)`

- Beispiel:

```
$ $ git branch --delete feature1  
Deleted branch feature1 (was edff90f).
```

- Löschen ungemergter Branches erzeugt Fehlermeldung

```
$ git branch --delete feature2
```

```
error: The branch 'feature2' is not fully merged.
```

```
If you are sure you want to delete it, run 'git branch -D feature2'.
```

- Ungemergete Branches löschen mittels einem der folgenden Befehle

```
git branch -D <branch>
```

```
git branch -d -f <branch>
```

```
git branch --delete --force <branch>
```

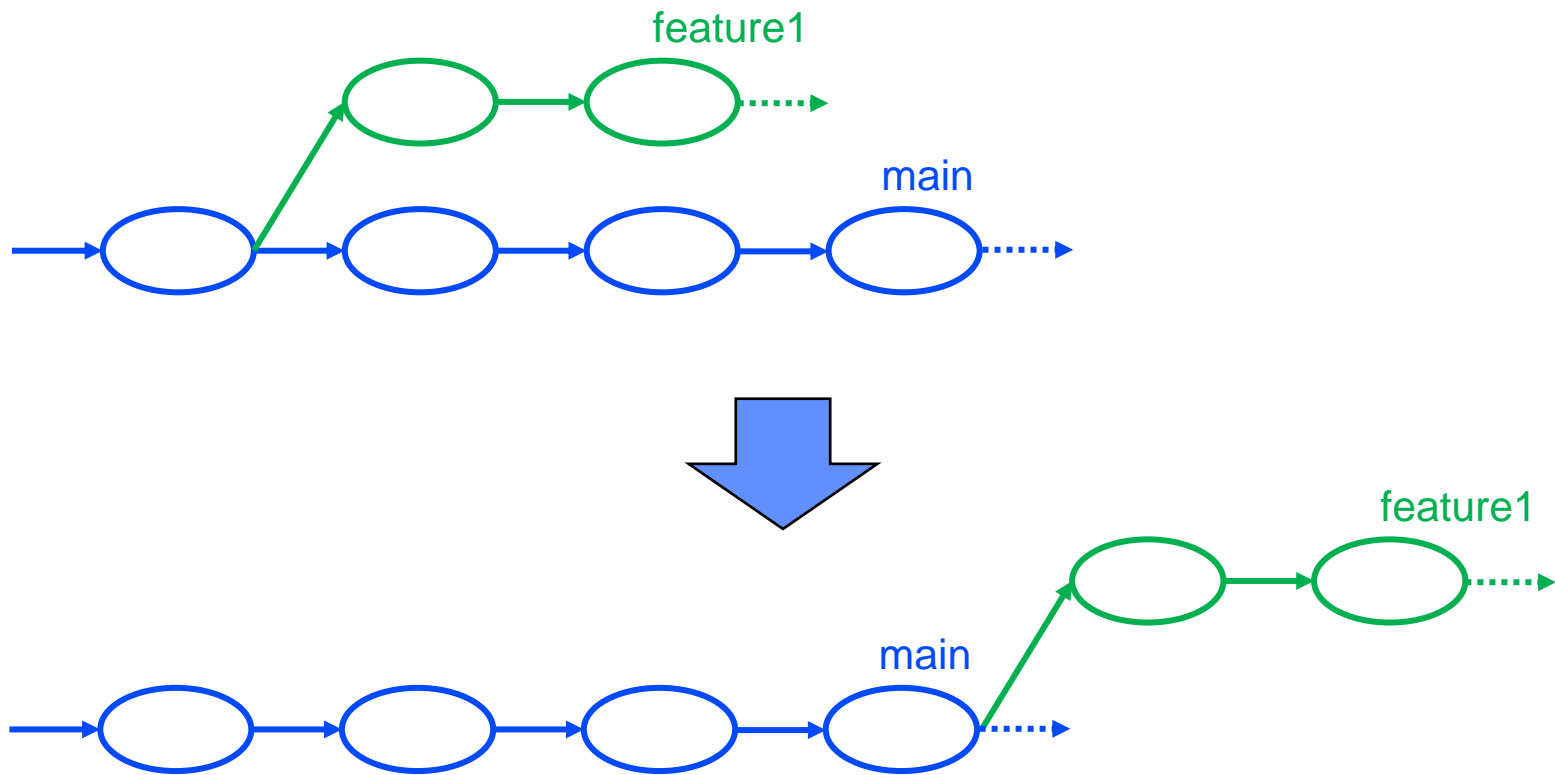
- Beispiel:

```
$ git branch -D feature2
```

```
Deleted branch feature2 (was 609a7d9).
```

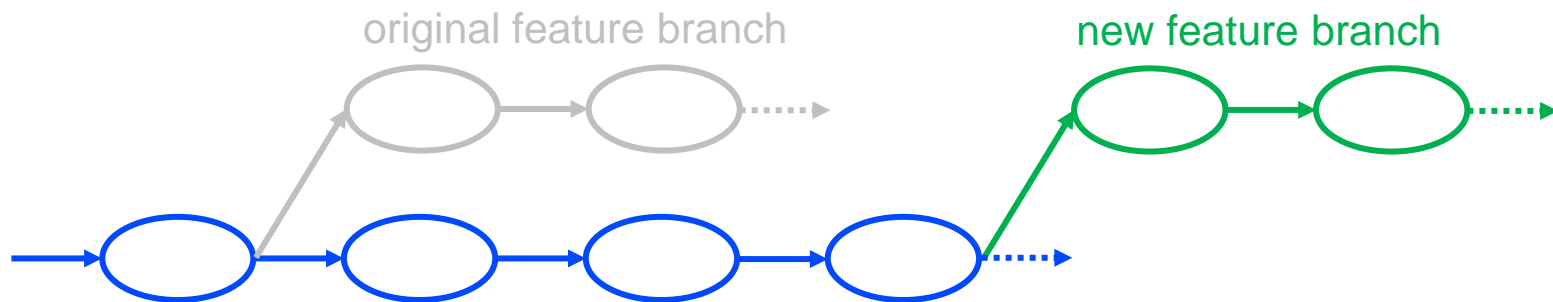
Git Rebase

- Rebase verändert Startpunkt eines Branches



-
- original feature branch
- new feature branch

- Beim Rebase werden der Zielbranch sowie die Commits während des Rebase *neu* erstellt
 - *Unterschiedliche* Commit-IDs
 - Zielbranch wird neu angelegt und ist nach Rebase unter dem Namen des ursprünglichen Branches verfügbar
 - Ursprünglicher Branch bleibt unter anderem Namen vorhanden, gilt aber als „dead“
 - Commits des ursprünglichen Branches bleiben mit alter Commit-ID erhalten



- Rebase des aktiven Branches auf *upstream* mittels

```
git rebase <upstream>
```

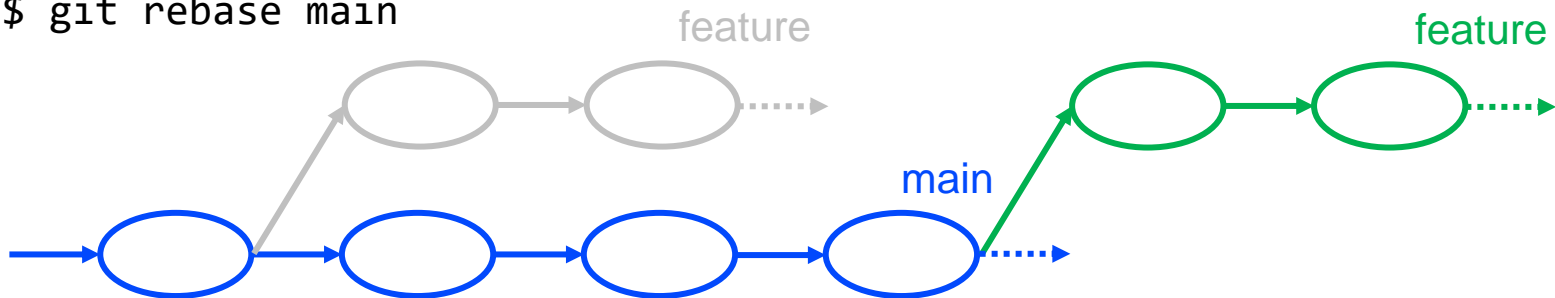
- Rebase eines Branches unabhängig vom aktiven Branch

```
git rebase <upstream> <branch>  
(verwendet intern git switch rebase vor rebase)
```

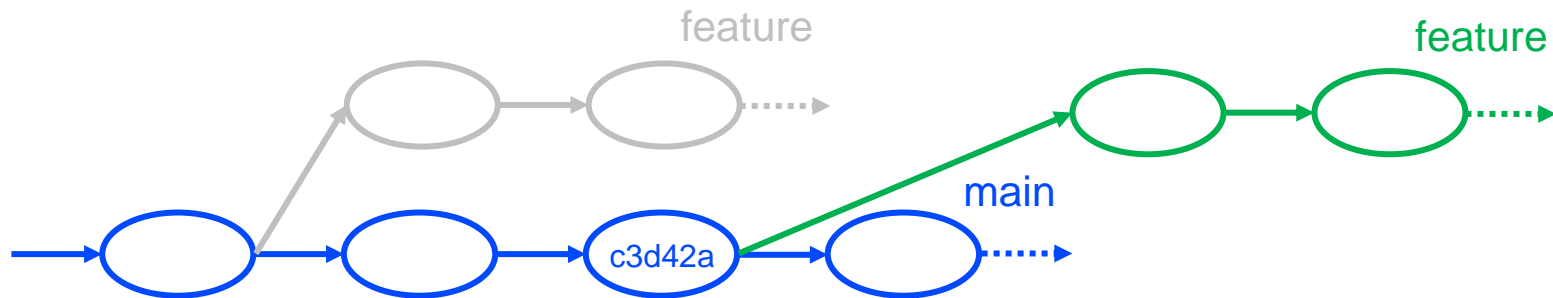
- Rebased auf den aktuellen HEAD des angegebenen Branches

- Beispiel: **feature** auf den HEAD des **main** rebasen

```
$ git checkout feature      oder      $ git rebase main feature  
$ git rebase main
```



- `<upstream>` und `<branch>` können Branches sowie einzelne Commit-IDs sein
- Beispiel: `feature` auf den `c3d42a` Commit des `main` rebasen
\$ git checkout feature
\$ git rebase c3d42a



Rebase Optionen

- Anwenden mittels `git rebase <option> ...`
- Werden beim *Start eines Rebase* angegeben

<code>--onto <newbase></code>	Gibt Startpunkt <newbase> an, von dem neue Commits aus erstellt werden
<code>--keep-base</code>	Behält alten Startpunkt beim Rebase bei
<code>--interactive</code> <code>-i</code>	Startet einen <i>interaktiven</i> Rebase
<code>--empty=(drop keep stop)</code>	Verhalten für Commits, die beim Start Änderungen enthalten, welche beim Rebase durch gleiche Änderungen vom Upstream leer werden

Option --onto

- --onto ermöglicht feinere Einstellung

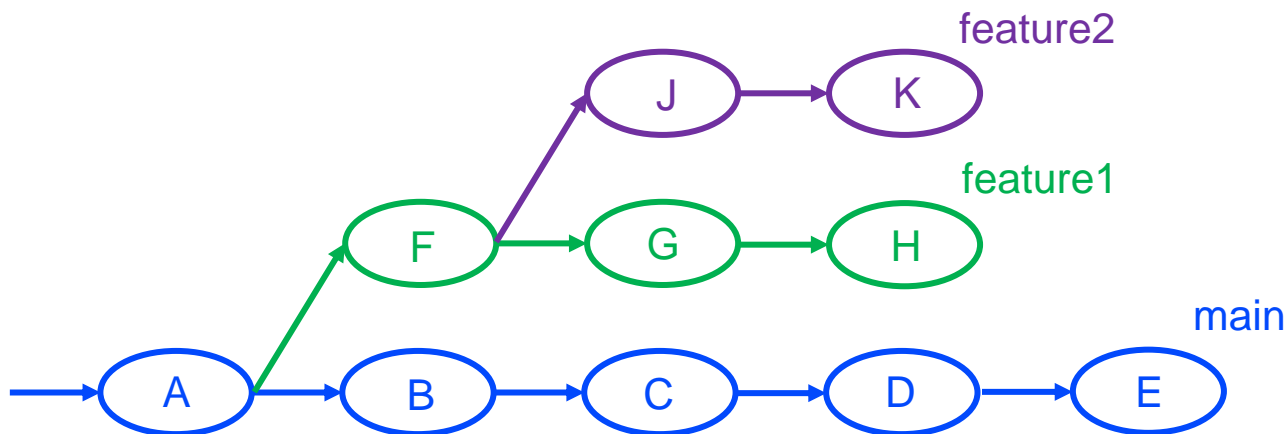
`git rebase --onto <newbase> <upstream> <branch>`

- <newbase> ist neuer Startpunkt
- <upstream> ist Branch oder der Commit, ab dem Commits verschoben werden
- <branch> ist Branch oder Commits, die verschoben werden.
Betrachtung rückwärts bis zum Erreichen des in <upstream> definierten Ausgangspunktes

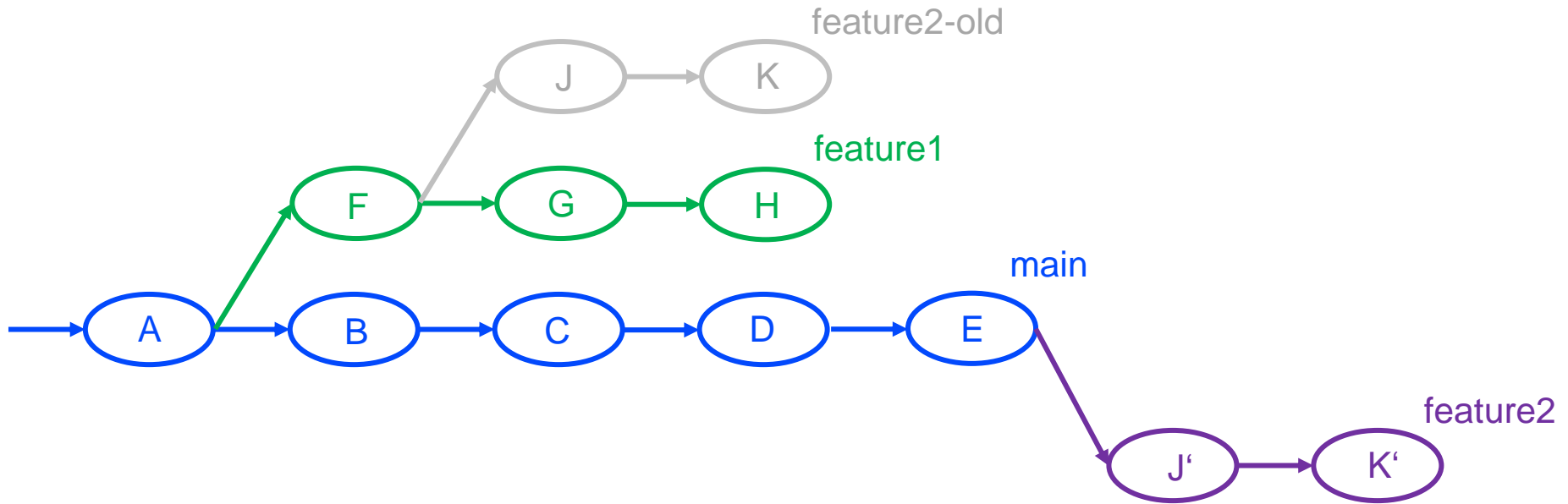
Beispiel: feature2 auf main rebasen

git rebase --onto main feature1 feature2

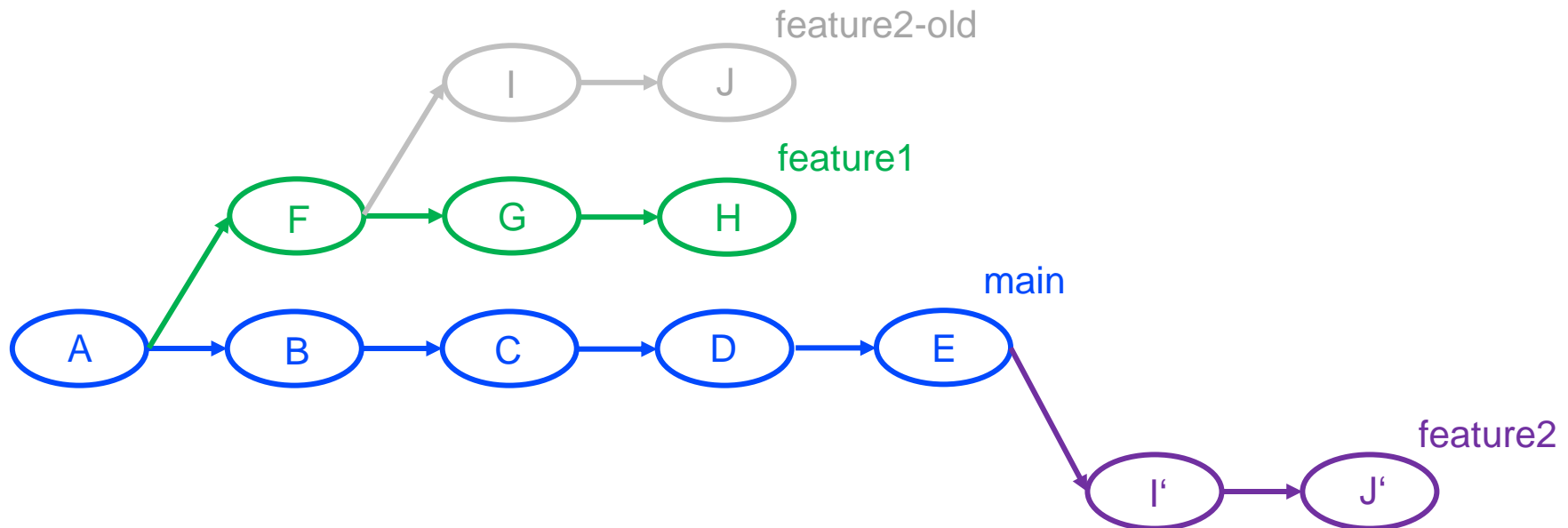
- main ist der neue Startpunkt, in diesem Fall der HEAD
- feature1 ist der Punkt, ab dem die Commits rebased werden.
- feature2 sind die Commits, die neu angewendet werden sollen



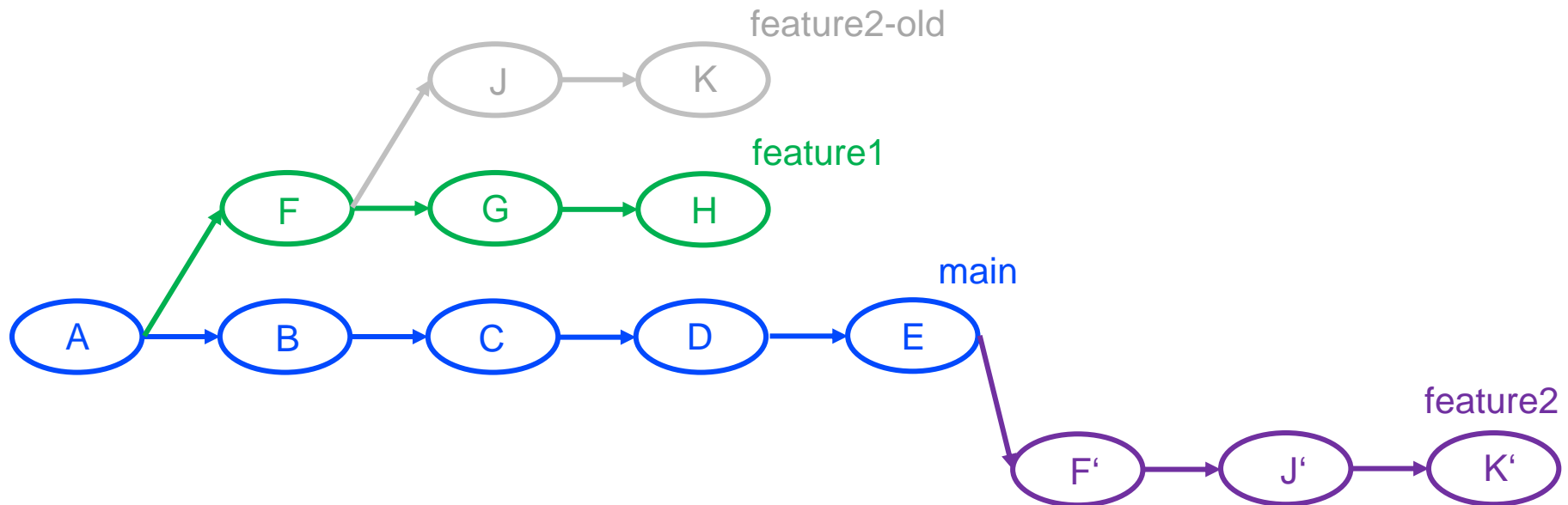
Ergebnis nach Rebase



- **feature2** basiert jetzt auf letzten Stand von **main**
- Commit **F** nicht enthalten
- Möglicher Use Case: **feature2** wurde in Abhängigkeit zu **feature1** geplant, stellt sich aber doch als unabhängig zu **feature1** heraus und soll daher unabhängig von **main** abzweigen



- `git checkout feature2` && `git rebase main` liefert anderes Ergebnis
- Commits ab **main** bis zum HEAD von **feature2** werden rebased
- Commit **F** enthalten



Rebase Mode Options

- Anwenden mittels `git rebase <option> ...`
- Werden *während eines Rebase* verwendet und können nicht mit anderen Optionen kombiniert werden

<code>--continue</code>	Setzt Rebase nach Auflösen von Merge Konflikten fort
<code>--skip</code>	Überspringt aktuellen Commit und setzt Rebase fort
<code>--abort</code>	Bricht Rebase ab und setzt HEAD zurück
<code>--quit</code>	Bricht Rebase ab, HEAD und Workspace werden nicht resettet

Interactive Rebase

- Über Texteditor den Rebase „programmieren“
- Starten mittels Option `--interactive` oder `-i`
- Commands im Editor ermöglichen Änderungen
- Commits können in Reihenfolge vertauscht werden
- Übernehmen durch Abspeichern und Schließen

```
pick b9898cd E
pick b059bc6 F
pick ce204eb G
pick 7b96b56 H
```

```
# Rebase 95e57be..7b96b56 onto 95e57be (4 commands)
```

```
#
```

```
# Commands:
```

```
# p, pick <commit> = use commit
```

```
# r, reword <commit> = use commit, but edit the commit message
```

```
# e, edit <commit> = use commit, but stop for amending
```

```
# s, squash <commit> = use commit, but meld into previous commit
```

```
# f, fixup [-C | -c] <commit> = like "squash" but keep only the previous
```

```
#                               commit's log message, unless -C is used, in which case
```

```
#                               keep only this commit's message; -c is same as -C but
```

```
#                               opens the editor
```

```
# x, exec <command> = run command (the rest of the line) using shell
```

```
# b, break = stop here (continue rebase later with 'git rebase --continue')
```

```
# d, drop <commit> = remove commit
```

```
# l, label <label> = label current HEAD with a name
```

```
# t, reset <label> = reset HEAD to a label
```

```
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
```

```
# .      create a merge commit using the original merge commit's
```

```
# .      message (or the oneline, if no original merge commit was specified); use -c <commit> to
reword the commit message
```

```
#
```

```
# These lines can be re-ordered; they are executed from top to bottom.
```

```
# If you remove a line here THAT COMMIT WILL BE LOST.
```

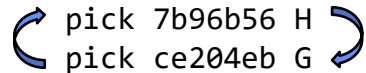
```
# However, if you remove everything, the rebase will be aborted.
```

- Betrachtung der folgenden 4 Commits

```
pick b9898cd E
pick b059bc6 F
pick ce204eb G
pick 7b96b56 H
```

- Vertauschen der Reihenfolge möglich

```
pick b9898cd E
pick b059bc6 F
pick 7b96b56 H
pick ce204eb G
```



- Squashen von Commit E und G möglich

```
pick b9898cd E
squash ce204eb G
pick b059bc6 F
pick 7b96b56 H
```

Interactive Rebase – Fixup

- Fixup beschreibt Situation, bei der ein Commit einen Fehler in einem anderen Commit behebt
- Ziel: Zusammenführung bei Rebase
- Commit-Nachricht von Fixup-Commit wird *verworfen*, außer anders spezifiziert
- Beispiel: Commit H ist Fixup von E

pick b9898cd E		pick b9898cd E
pick b059bc6 F		fixup 7b96b56 H
pick ce204eb G	→	pick b059bc6 F
pick 7b96b56 H		pick ce204eb G

Übungsaufgabe 12: Rebase

Im Folgenden werden wir einen häufigen Use Cases eines Rebase betrachten, indem wir per Rebase Änderungen des **main** Branches in unseren Feature Branch übernehmen

1. Erstellen Sie einen Branch **feature5** und wechseln Sie in diesen.
2. Legen Sie im Ordner **features** eine Datei **feature5_file1.txt** an und committen Sie diese. Legen Sie anschließend eine weitere Datei **feature5_file2.txt** und committen diese ebenfalls.
3. Wechseln Sie nun zurück auf den **main** Branch, legen dort eine Datei **important_changes.txt** an und committen diese.

Übungsaufgabe 12: Rebase

Unser Ziel ist es nun, diese **important_changes.txt** auch im **feature5** Branch zur Verfügung zu haben. Man könnte nun den **main** Branch in unser Feature mergen. Solange man aber alleine auf dem Feature Branch arbeitet, ist ein Rebase die elegantere und sauberere Lösung.

4. Wechseln Sie in den **feature5** Branch, da wir diesen nun verändern möchten.
5. Lassen Sie sich mittels `git log` die Commit-IDs der beiden **feature5** Commits anzeigen und notieren Sie sich die IDs.
6. Rebasen Sie nun **feature5** auf den HEAD des **main** Branches.
7. Verifizieren Sie, ob die Datei **important_changes.txt** nun existiert.
8. Vergleichen Sie die Commit-IDs der aktuellen Ausgabe von `git log` mit der vorherigen Ausgabe. Unterscheiden sich die IDs?

Git

Konflikte beim Rebase

- Ähnlich zu Merge-Konflikten
- Rebase wird pausiert und Konflikte müssen manuell behoben werden
- Nach Auflösen des Konflikts betroffene Dateien mit `git add <file>` stagen
- Danach mittels `git rebase --continue` Rebase fortsetzen
- Ggf. Konflikte bei jedem Commit im Rebase
- Schwierigkeit eines Rebases steigt mit „Unordnung“ in Commit-Historie

Beispiel

```
$ git branch feature
$ echo "hello world from main" > conflict_file.txt
$ git add conflict_file.txt
$ git commit -m "Add file from main"
[main 722eaf0] Add file from main
1 file changed, 1 insertion(+)
create mode 100644 conflict_file.txt

$ git checkout feature
Switched to branch 'feature'

$ echo "hello world from feature" > conflict_file.txt
$ git add conflict_file.txt
$ git commit -m "Add file from feature"
[feature ce692c7] Add file from feature
1 file changed, 1 insertion(+)
create mode 100644 conflict_file.txt
```

```
$ git rebase main
```

```
Auto-merging conflict_file.txt
```

```
CONFLICT (add/add): Merge conflict in conflict_file.txt
```

```
error: could not apply ce692c7... Add file from feature
```

```
hint: Resolve all conflicts manually, mark them as resolved with
```

```
hint: "git add/rm <conflicted_files>", then run "git rebase --continue".
```

```
hint: You can instead skip this commit: run "git rebase --skip".
```

```
hint: To abort and get back to the state before "git rebase", run "git rebase --abort".
```

```
Could not apply ce692c7... Add file from feature
```

```
$ nano conflict_file.txt
```

```
$ git add conflict_file.txt
```

```
$ git rebase --continue
```

```
Successfully rebased and updated refs/heads/feature.
```

Git – Rebase

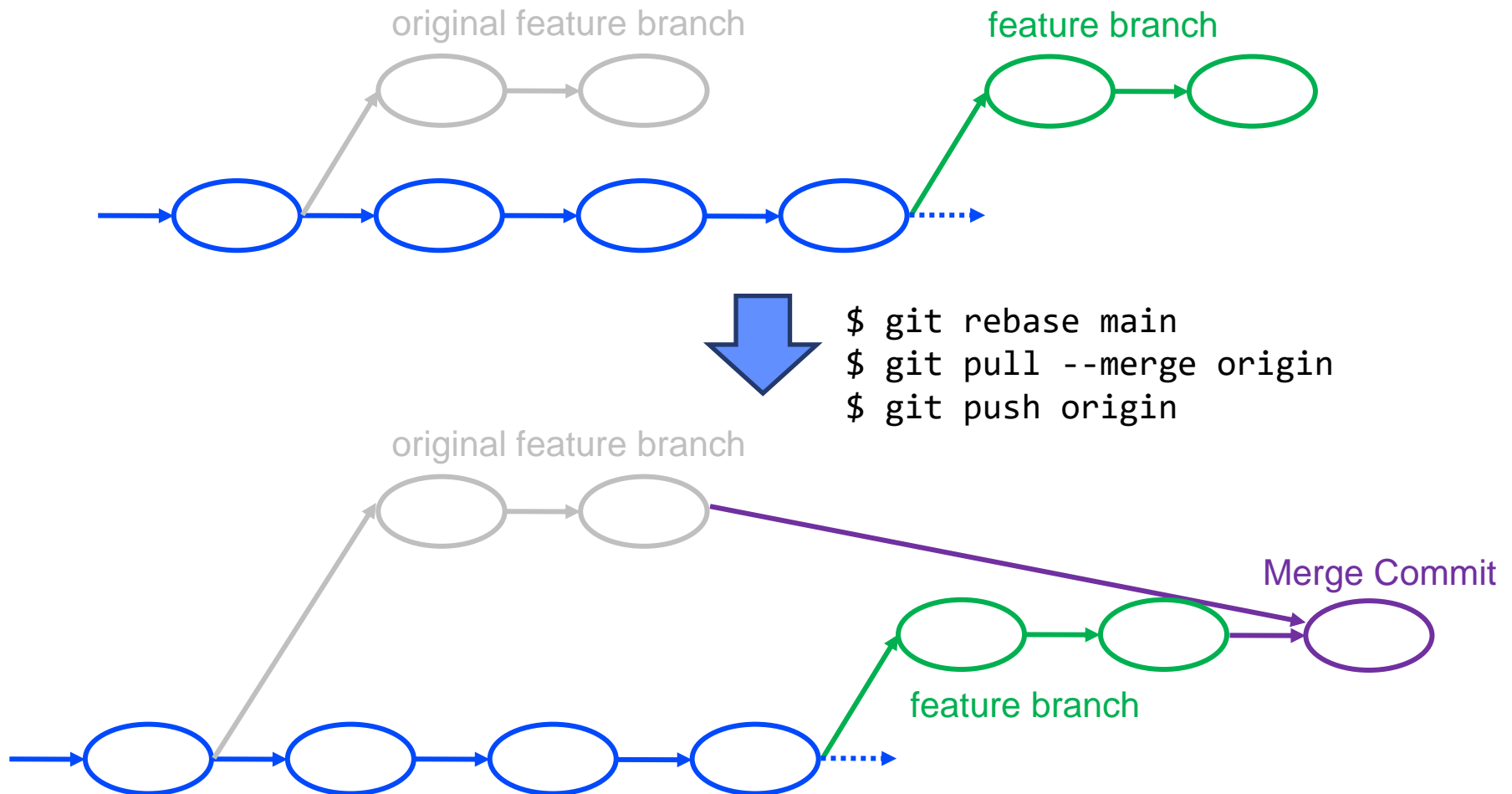
Auswirkungen

Auswirkungen auf Commit Historie

- Rebase verändert die Commit-Historie (im Gegensatz zum Merge)
 - Kann zu Konflikten mit einem Remote-Repository führen
 - Branch folgt nicht der Commit-Historie vom Remote
 - Remote lehnt Push ab
 - Mittels `git push --force` (oder `-f`) alten Branch im Remote Repository überschreiben
 - Andere Entwickler können lokalen Branch nicht weiter verwenden
- ➔ **Auf öffentlichen oder geteilten Branches Rebase unbedingt vermeiden**

- Nach Rebase immer mit --force pushen
- `git pull --merge` mit anschließendem `git push` ohne --force führt zu anderem Verhalten
- Erzeugt zwar keinen Konflikt, dafür aber zusätzlichen Commit

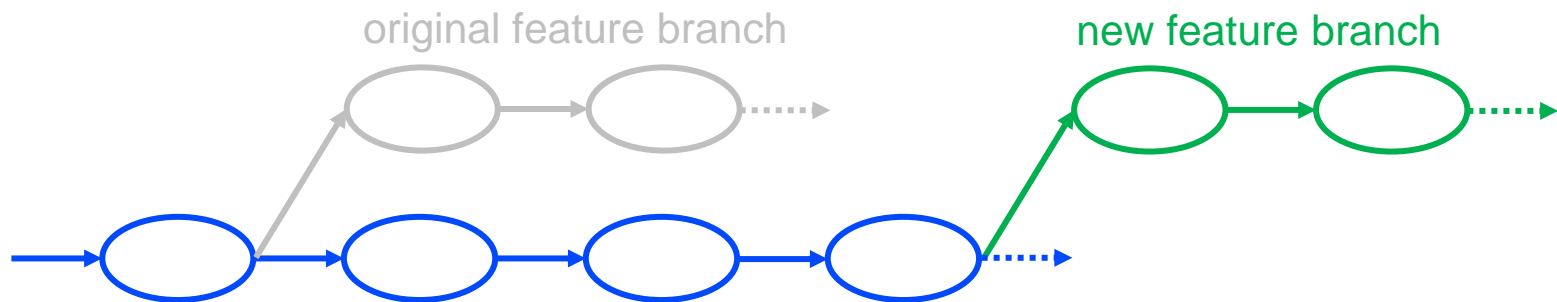
- **Beispiel** zu beschriebenem Szenario (oben Stand nach git rebase)



Git – Rebase

Use Cases

- Rebase kann in vielen Situationen verwendet werden
 - Übernahme von Upstream Änderungen in Feature Branch
 - Feature auf späteres Release verschieben
 - Ein Feature zweigt von anderem Feature ab, ist aber unabhängig. Daher Rebase auf main
 - Strukturänderungen im Repository



Nutzung zur nachträglichen Änderung von Commits

- Nachträgliche Änderung von Commit-Historie auf aktuellem Branch durch interactive Rebase
- Startpunkt des Branches wird nicht verschoben
- Neue Anwendung der letzten n Commits möglich mittels

```
git rebase -i HEAD~n
```

- Alle Commits des aktuellen Branches neu schreiben

```
git rebase --keep-base -i main
```

Übungsaufgabe 13: Rebase

Im Folgenden werden wir einen weiteren Use Cases eines Rebase betrachten, indem wir per Rebase die Commits unseres aktuellen Branches „aufräumen“.

1. Sie sollten sich noch auf dem **feature5** Branch befinden.
2. Füllen Sie die Datei **feature5_file1.txt** mit dem Inhalt „Hallo aus Datei 1 aus feature5 “ und committen Sie Ihre Änderungen.
3. Füllen Sie die Datei **feature5_file2.txt** mit dem Inhalt „Hello from file 2 in feature5 “ und committen Sie Ihre Änderungen.
4. Erstellen Sie einen weiteren Commit, in dem Sie den Inhalt von **feature5_file1.txt** umändern, sodass dieser ebenfalls in Englisch ist.

Übungsaufgabe 13: Interaktiver Rebase

Nun haben Sie auf Ihrem Branch relativ viele einzelne Commits, von denen einer nur eine Reparatur einer vorherigen Änderung ist.

Da Sie die einzige Person sind, die auf dem Branch arbeitet, können Sie die Commits neu schreiben. Dabei soll der Startpunkt des Branches nicht verändert werden.

5. Starten Sie einen interaktiven Rebase.

feature5 soll auf **main** gerebased werden, jedoch ohne die Merge-Base zu verändern.

Sie können hierzu zum einen mittels `git merge-base` den Abzweig-Commit von **feature5** herausfinden und auf diesen Rebasen oder den Rebase mit der Option `--keep-base` starten.

Übungsaufgabe 13: Interaktiver Rebase

6. Ordnen Sie im Editor die angegebenen Commits anders an. Fügen Sie den Commit zum Erstellen von file1 mit dem Commit zum Hinzufügen des Inhalts zu file1 mittels squash zusammen.
7. Squashen Sie ebenfalls die beiden Commits von file2.
8. Fügen Sie den Commit zum Reparieren des Inhalts von file1 als fixup an den ursprünglichen Commit hinzu.

Ihr Editor sollte nun ungefähr so aussehen

```
pick 8dafe28 add feature5_file1
squash 3d45a2 add content to feature5_file1
fixup af422a8 change content language to english
pick 2de828 add feature5_file2
squash bd7f22e add content to feature5_file2
```

9. Speichern Sie Ihre Änderungen ab und passen Sie im nächsten Fenster die Commit Nachrichten entsprechend an.

Übungsaufgabe 13: Interaktiver Rebase

6. Speichern Sie ebenfalls die Änderungen an den Commit Nachrichten. Der interaktive Rebase sollte nun erfolgreich abgeschlossen sein.
7. Verifizieren Sie über `git log`, dass Ihre Änderungen am **feature5** Branch erfolgreich waren.

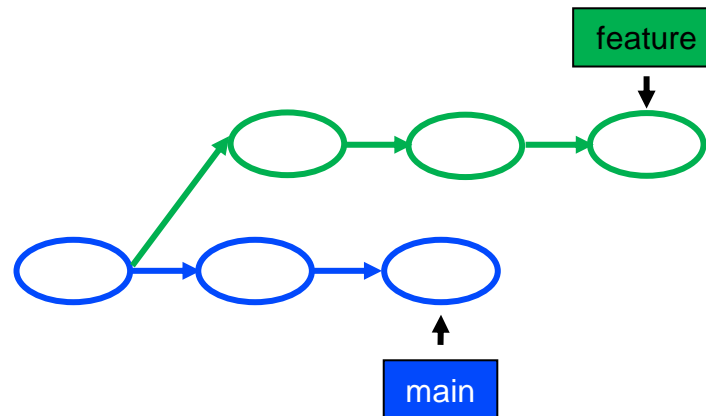
Git – Rebase

Use Cases – Merge vs. Rebase

- Aufgabe von Merge und Rebase: Änderungen in Branch einpflegen
 - Rebase ändert aktiven Branch und kann Änderungen vom Upstream übernehmen
 - Merge zieht Änderungen von anderen Branches in den aktiven Branch
- Daher die Frage „Wann benutze ich Merge, wann Rebase?“
 - Merging führt Commit-Historie fort → weniger Konfliktpotenzial als Rebasing
 - Rebasing erzeugt saubere, lineare aber veränderte Commit-Historie
- Um lokal Änderung aus Upstream einpflegen, ist Rebase oft die bessere Option
 - Merge vom Upstream in Feature kann zu komplizierten Szenarien führen
 - Erzeugt lineare Historie

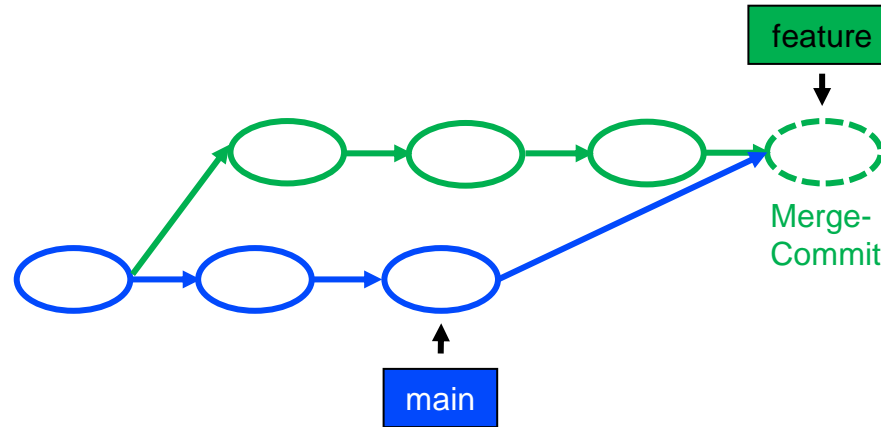
Merge vs. Rebase

- Beispiel: Einfügen von Änderungen von **main** in **feature** Branch

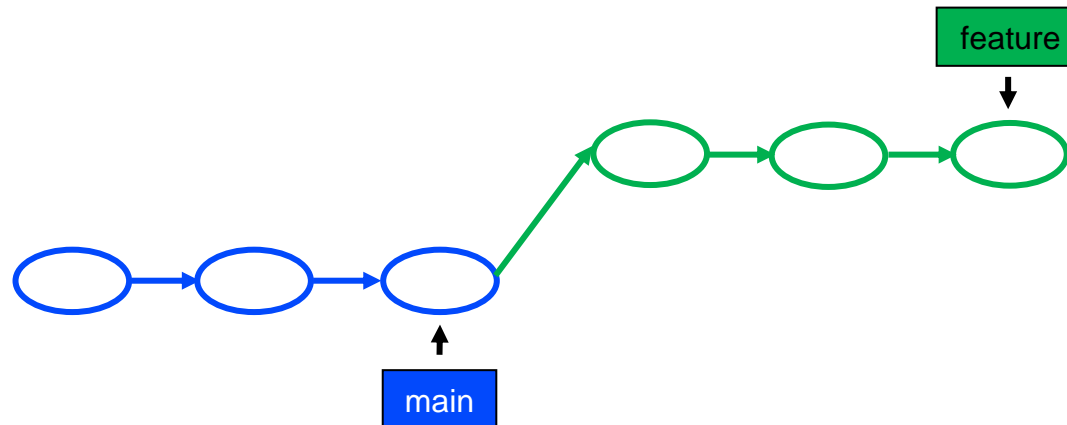


Merge vs. Rebase

Merge



Rebase



- Um Feature Branch in Upstream zurückzuführen, verwendet man Merge
 - Upstream soll fortgeführt werden
 - Upstream meist mit anderen Entwicklern geteilt
 - Ggf. Rebase des Feature Branches auf den HEAD des Upstream vor einem Merge sinnvoll. Dadurch ergibt sich Fast-Forward Merge

