Git

17 Oct 2013

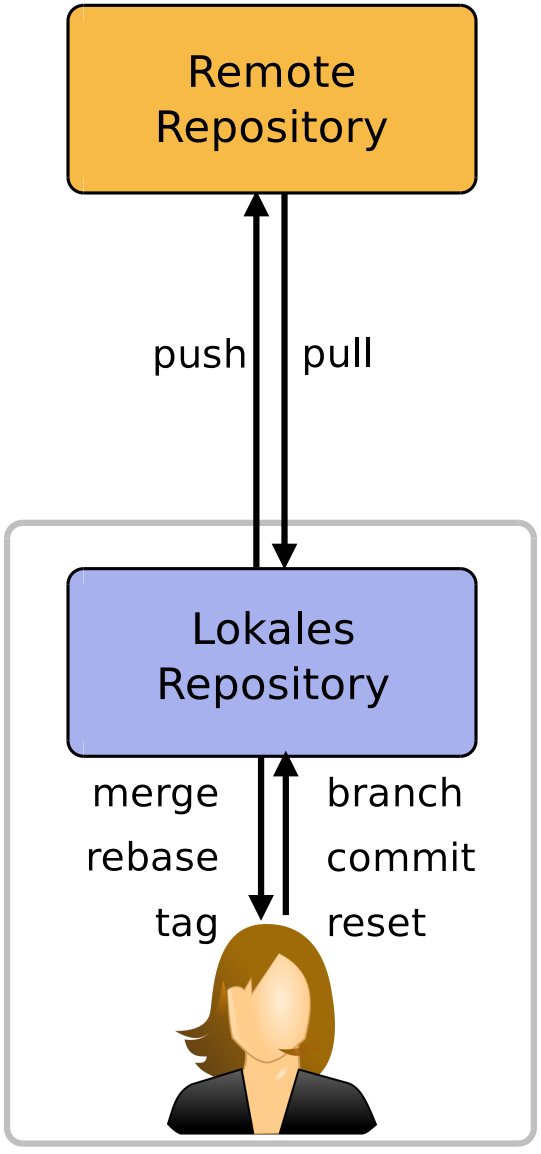
Goals

Goals

- Learn the basics of Git
- Useful for everyday work
- Mainly using local repositories
- Learn the basics of Github

Overview

Autonomy of Repositories



- *Remote* and local repository have equal rights
- Synchronisation between repositories via pull/push
  - *Push*: Upload own changes
  - *Pull*: Download others' changes
- Everything else happens locally

Advantages and Disadvantages of Distributed Version Control Systems (DVCS)

Advantages:

- Every developer has a complete copy of the public history
  - Commands are executed quickly
  - Enables working offline
  - Implicit protection against manipulation
- No «single point of failure»
  - Server offline, disgruntled developer, security breach ...

Advantages and Disadvantages of Distributed Version Control Systems (DVCS)

Advantages:

- No conflicts regarding commit-access
- Delegation of tasks becomes easier
- Flexible workflows

Advantages and Disadvantages of Distributed Version Control Systems (DVCS)

Disadvantages:

- Lots of freedom, appropriate policies must be established
- Slightly more complex setup

Overview

Vocabulary

- **Commit**: A change in one more files; includes meta-data such as author, date and description
- **Commit-ID**: every **Commit** is identified by a unique SHA-1-Sum, its **ID**
- **Repository**: «Container» for saved **Commits**
- **Working-Tree**: the current files in the working directory
- **Branch**: A bifurcation in the development, e. g. to add a new feature
- **Reference**: A reference «points» to a specific commit, e. g. a branch

- **Index/Staging-Area**: Area between **Working-Tree** and **Repository** where changes for the next **Commit** are collected

Configure Git

- Using `git config` the configuration can be queried and changed
- Usually just for the current project
  - Saved in `.git/config`
- With the `--global` switch for the current user
  - Saved in the file `~/.gitconfig`

Wo am I? – Set Name and Email

- Before we can use Git, we must introduce ourselves
- Information will be used when creating a commit
- The defaults are `$USER` and `hostname`

For the current user

```
git config --global user.name "John Doe"
git config --global user.email john@doe.com
```

... alternatively just for the current project

```
git config user.email maintainer@cool-project.org
```

Colorize Output

Colors for the output

```
git config --global color.ui auto
```

Inspect configuration

```
git config color.ui
```

Warning! *do not* use =

```
git config --global color.ui = auto
```

Overview

Import or Create a New Project

Creating a new project

```
git init project
```

To *import* an existing project, you «clone» it

```
git clone git://gitschulung.de/projekt
```

A Typical Workflow

Modify a *file* and «check-in» the changes:

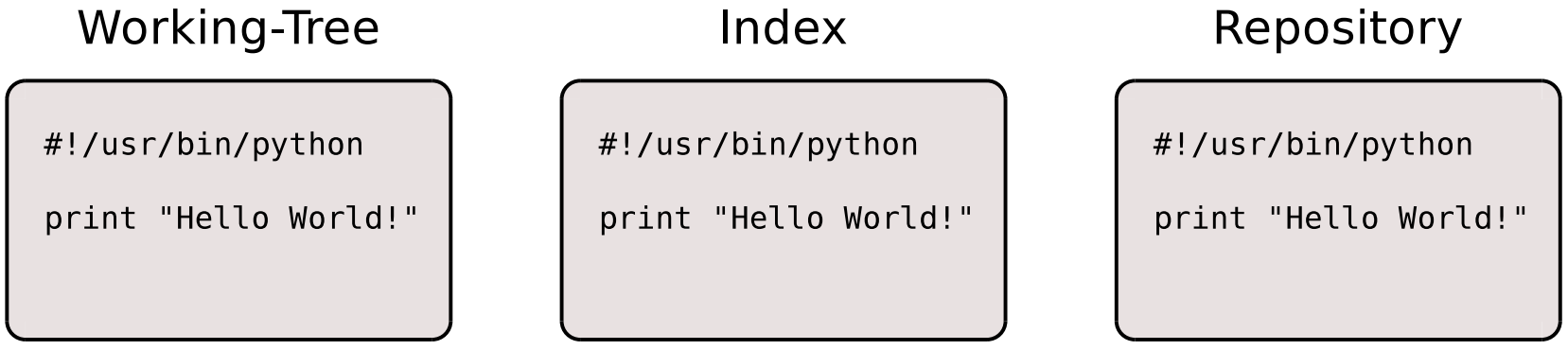
1. `$EDITOR` *file*
2. `git status`
3. `git add` *file*
4. `git commit -m 'modified` *file*`'`
5. `git show`

Index / Staging Area

- The *index/staging arae* is used to «stage» changes for the next commit
- Hence the commit can be assembled piece by piece from single changes
- After a commit, the index contains exactly the changes of the last commit

Initial State

- All have the same state



Making Changes

- Changes are made to the working-tree

Working-Tree

```
 #!/usr/bin/python
+# Autor: Valentin
+
 print "Hello World!"
```

Index

```
#!/usr/bin/python

print "Hello World!"
```

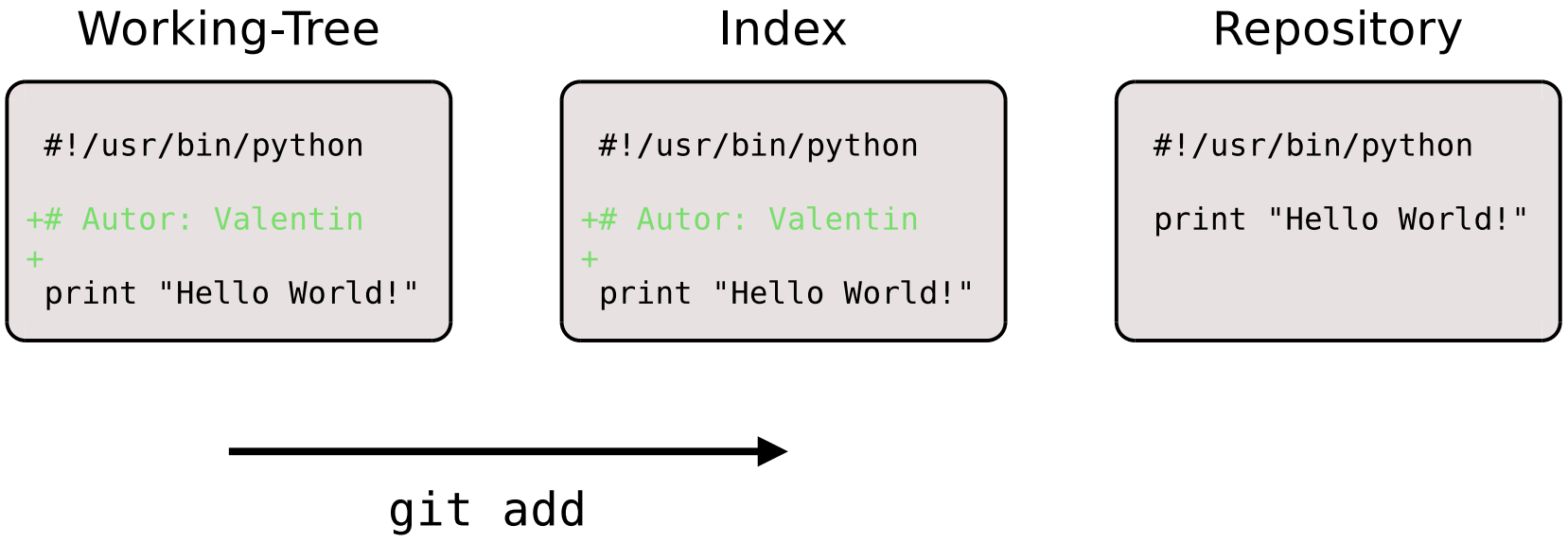
Repository

```
#!/usr/bin/python

print "Hello World!"
```

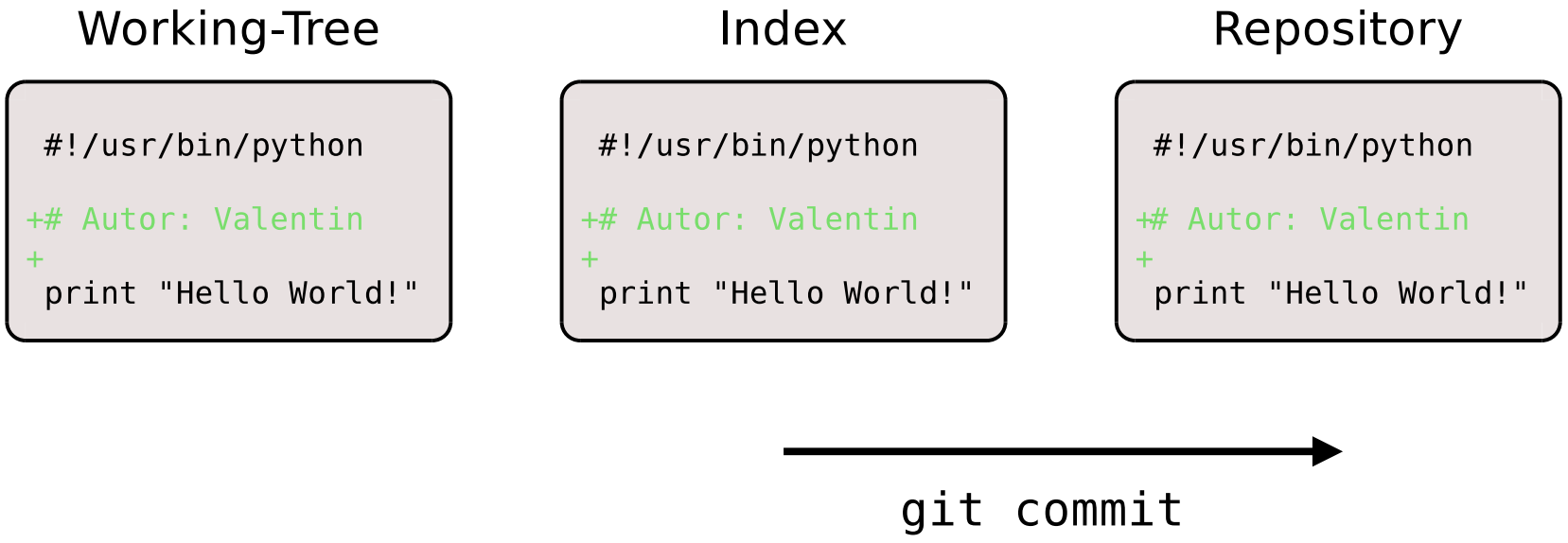
Adding to the Index – `git add`

- Changes to the working-tree → index



Creating a Commit – `git commit`

- All changes in the index → commit



Result

- All have the same state again

Working-Tree

```
#!/usr/bin/python

# Autor: Valentin

print "Hello World!"
```

Index

```
#!/usr/bin/python

# Autor: Valentin

print "Hello World!"
```

Repository

```
#!/usr/bin/python

# Autor: Valentin

print "Hello World!"
```

HEAD

HEAD

The most recent commit in the history is the `HEAD`

`git status` – What is the Status?

Query Status

```
git status
```

- Which files were modified?
- Which changes are in the index?
- Are there untracked files?

Adding Files to the Index

Adding all changes in a file

```
git add file
```

Interactive adding

```
git add -p
```

Interactive adding, just for a single file

```
git add -p file
```

Index: Differences and Resetting

Changes between working-tree and index

```
git diff
```

Changes between index and HEAD

```
git diff --staged
git diff --cached
```

Reset the Index

```
git reset
```

`git commit` – Creating commits

- The most often used command
- Changes in the index are «bundled» together into a *commit*

Create a commit

```
git commit
```

A commit message on the command line

```
git commit -m "message"
```

All changes in the working-tree

```
git commit -a
```

Advanced Usage

Amend the most recent a commit

```
git commit --amend
```

Create an empty commit

```
git commit --allow-empty
```

Committing with different Author information

```
git commit --author="Maxine Mustermann \
    maxine@mustermann.de"
```

Including the line `Signed-off-by:`

```
git commit -s
```

Commit-Message

- The first line of the commit message should not exceed 50 characters
- Should be short and concise – but still informative!
- Explain *why* something was changed
  - *what* was changed is evident from the `diff`

Example

```
commit 95ad6d2de1f762f20edb52d139d3cc19529a581a
Author: Matthieu Moy <Matthieu.Moy@imag.fr>
Date:   Fri Sep 24 18:43:59 2010 +0200

update comment and documentation for :/foo syntax

The documentation in revisions.txt did not match the
implementation, and the comment in sha1_name.c was
incomplete.
```

Viewing History

History since the beginning

```
git log
```

History including patches

```
git log -p
```

The current commit only

```
git show
```

One-line summary

```
git log --oneline
```

Overview

Branching – A Piece of Cake

- Branches in Git are fast and intuitive
  - Revolutionizes your workflow
- A branch is *not* a complete copy of the project
  - «Branches are cheap»
- More like: a «label» that is attached to a commit

Listing branches

```
git branch [-v]
```

- We've been working on the `master` branch all the time

Creating Branches

- No data is copied
  - Creation takes only a few milliseconds

Creating a branch

```
git branch name
```

Explicitly name the starting commit

```
git branch name start
```

Switching Branches

- To switch to a branch, it must be «checked-out»

Checking out a branch

```
git checkout branch
```

- For people coming from Subversion
  - The current directory is *not* changed
  - Instead: content of the branch $\rightarrow$ working-tree

Create a branch and switch to it

```
git checkout -b name
```

Manipulating Branches

Renaming branches

`git branch -m` *old new*

`git branch -M` *old new* (force)

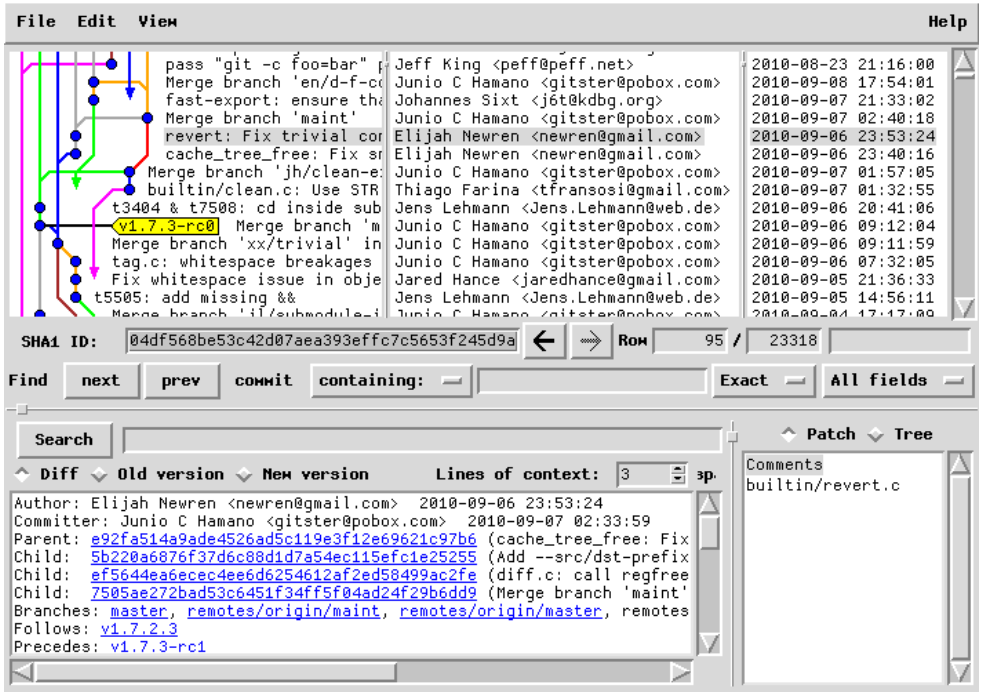
Deleting Branches

`git branch -d` *name*

`git branch -D` *name* (force)

- Forcing (using `-M` or `-D`) is required if branches are to be over-written or commits potentially lost

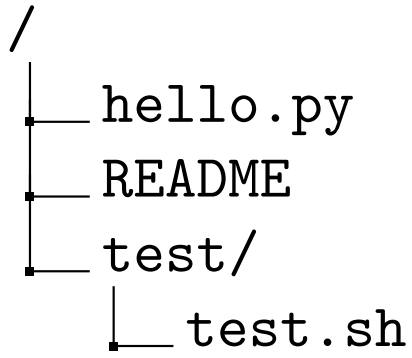
gitk: the Graphical Repository Browser



`gitk --all`

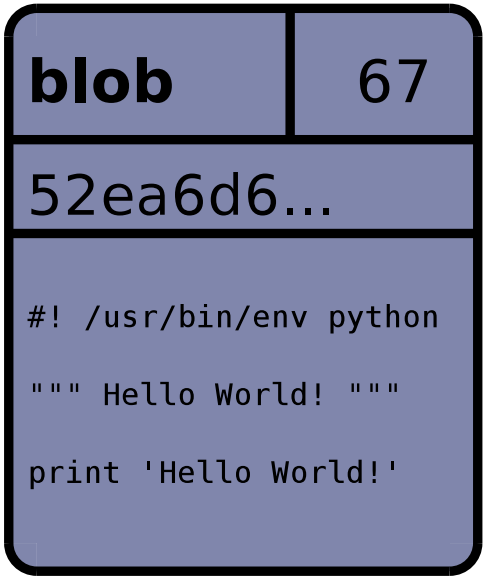
What do we Want to Store?

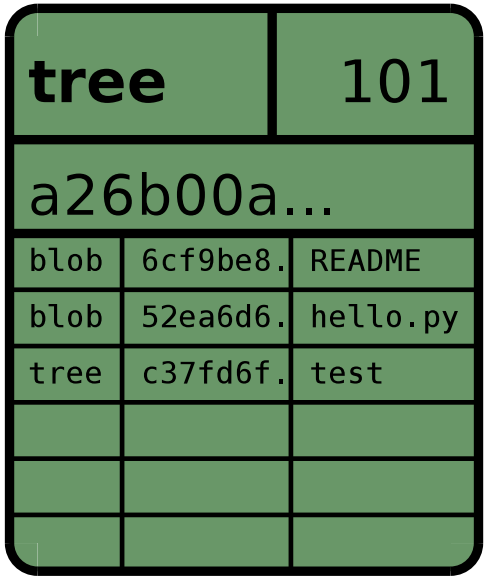
Suppose we want to store the following directory:



Object Model

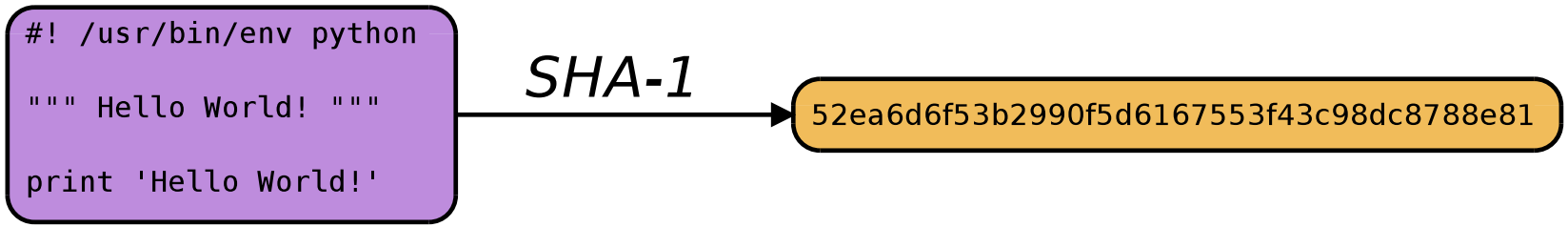
- *Blob*: Contains the contents of a file
- *Tree*: A collection of tree and blob objects
- *Commit*: Consists of a reference to a tree with additional information
  - *Author* and *Commiter*
  - *Parents*
  - *Commit-Message*





SHA-1 IDs

- Objects are identified with *SHA-1 IDs*
- This is the *object-name*
- It is derived from the content
- *SHA-1* is a hash function, which converts a bit sequence with a maximum length of 2^{64} -1 bits ($\approx$ 2 Exbibyte) a hexadecimal number of length 40 (i.e. 160-bit)
- The resulting number is one of 2^{160} ($\approx 1.5x10^{49}$) possible numbers and quite unique



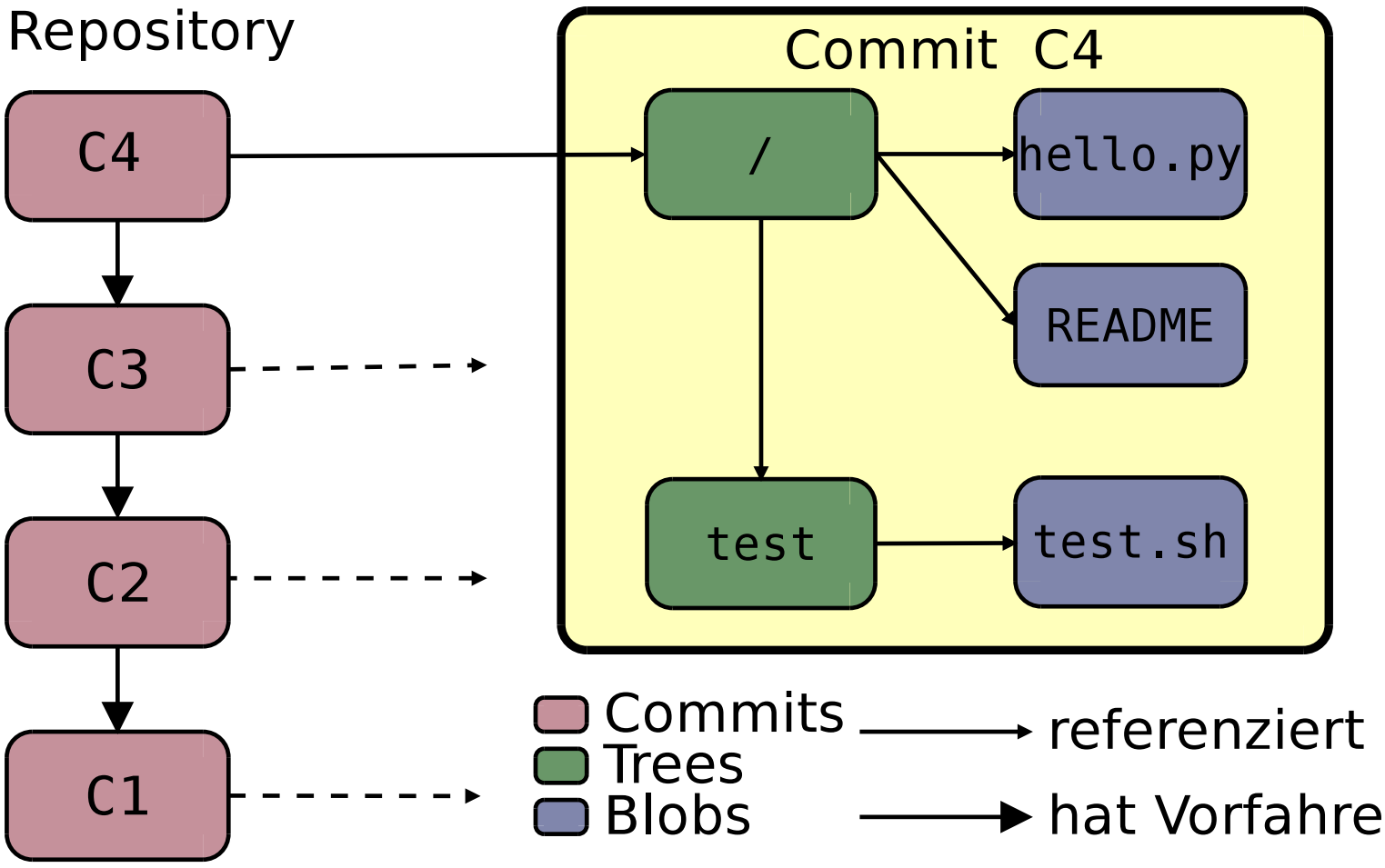
Object Management

- All objects are stored in the Git *object database* (a.k.a. the repository)
- The objects are identified by their uniquely addressable SHA-1-ID

- For each file, Git will create a blob object
- For each directory, Git will create a tree object
- A tree object contains references (SHA-1 IDs) to the files contained in the directory

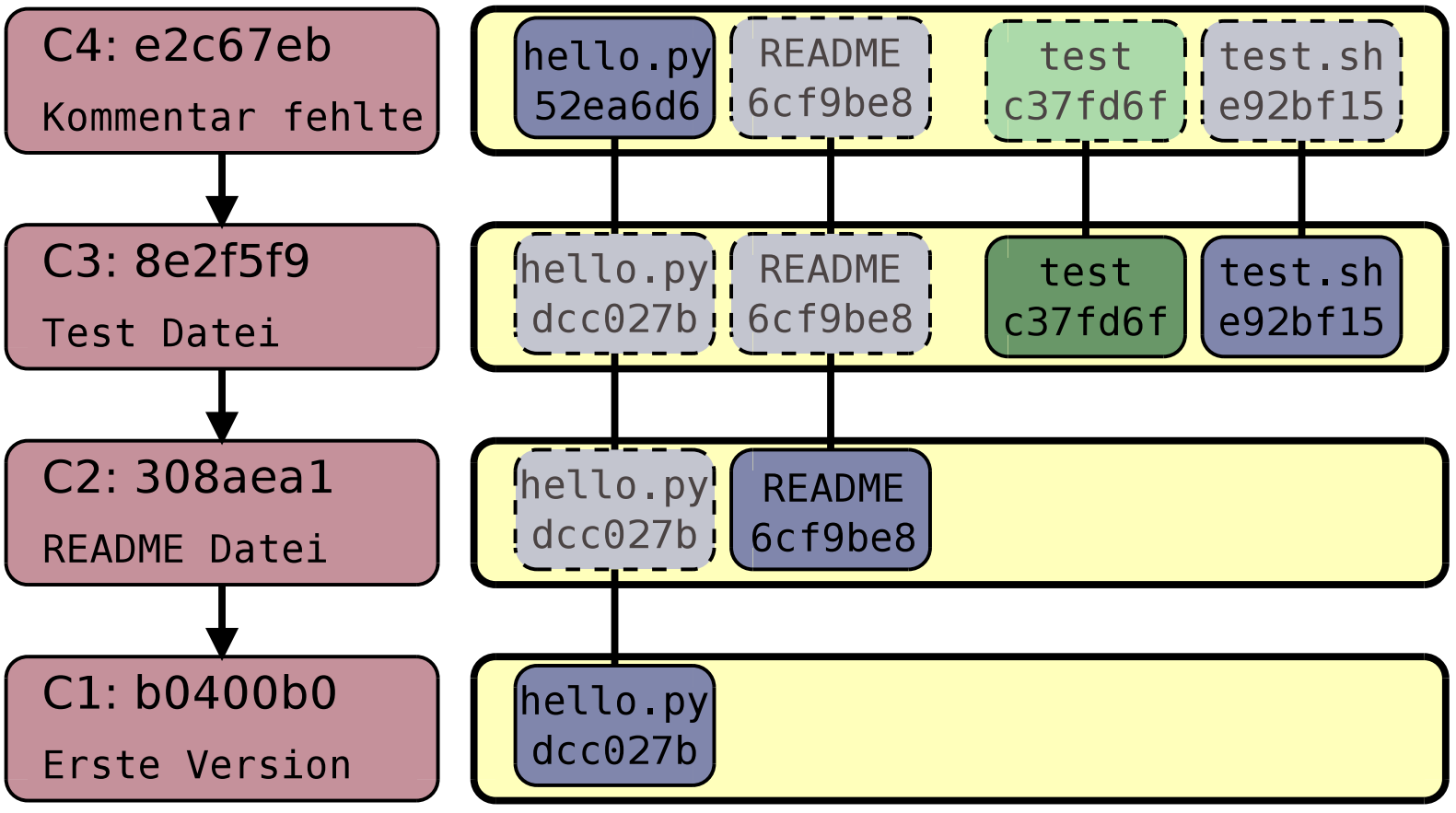
Summary

A Git repository contains commits; these then reference trees and blobs, and their direct predecessors



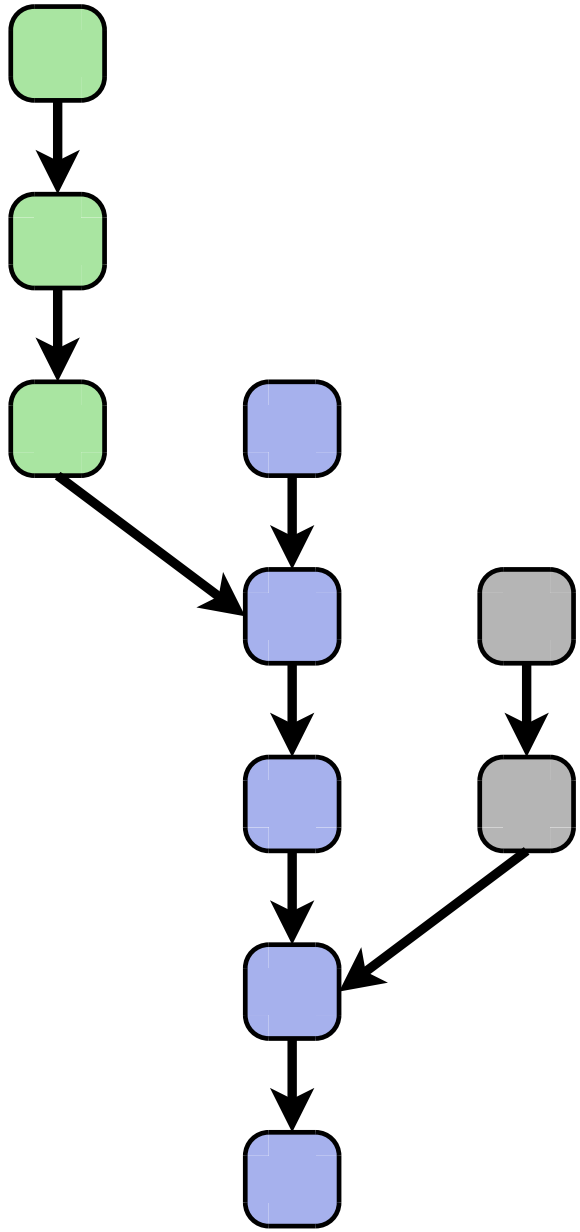
Commit = File Tree

A commit references the state of *all* tracked files at any given time.
(Even those that have not changed as part of the commit!)



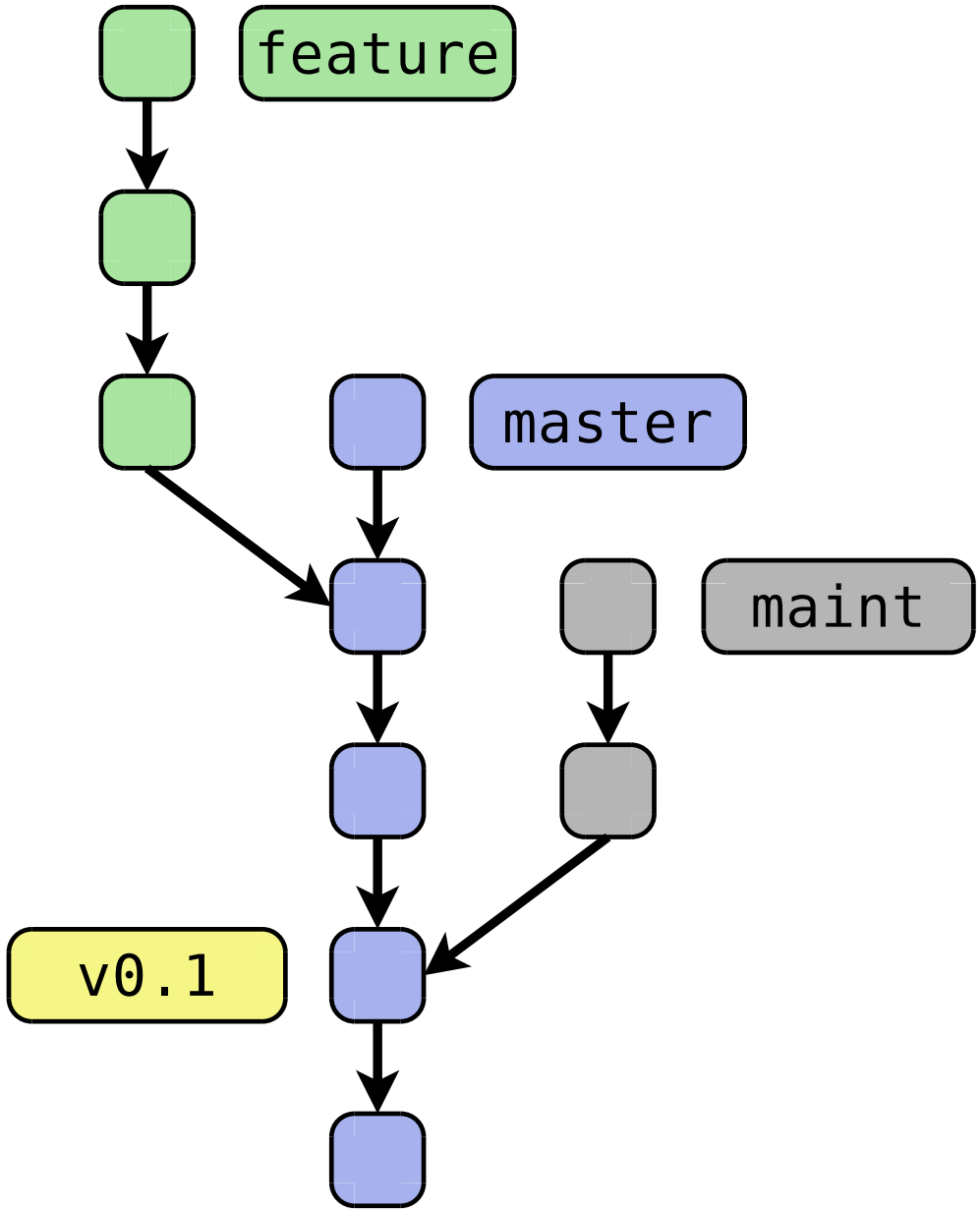
Commit Graph

The repository is a *Directed Acyclic Graph (DAG)*



Branches and Tags

Branches and tags are pointers



Graph Structure

- The directed graph structure arises because each commit contains references to its direct ancestors
- Integrity is cryptographically secured
- Git commands simply manipulate the graph structure

Tags

- Tags are used to mark important commits in the development history, e. g. releases

Lightweight Tags

Just a reference to a commit

Annotated Tags

Contain additional information (author, date) a message, and can be digitally signed. (This is the preferred way)

Tag Commands

Show all tags

```
git tag
```

Create a lightweight tag

```
git tag v1.0
```

Create an annotated tag

```
git tag -a v1.0 -m "tag message"
```

Delete tag

```
git tag -d v1.0
```

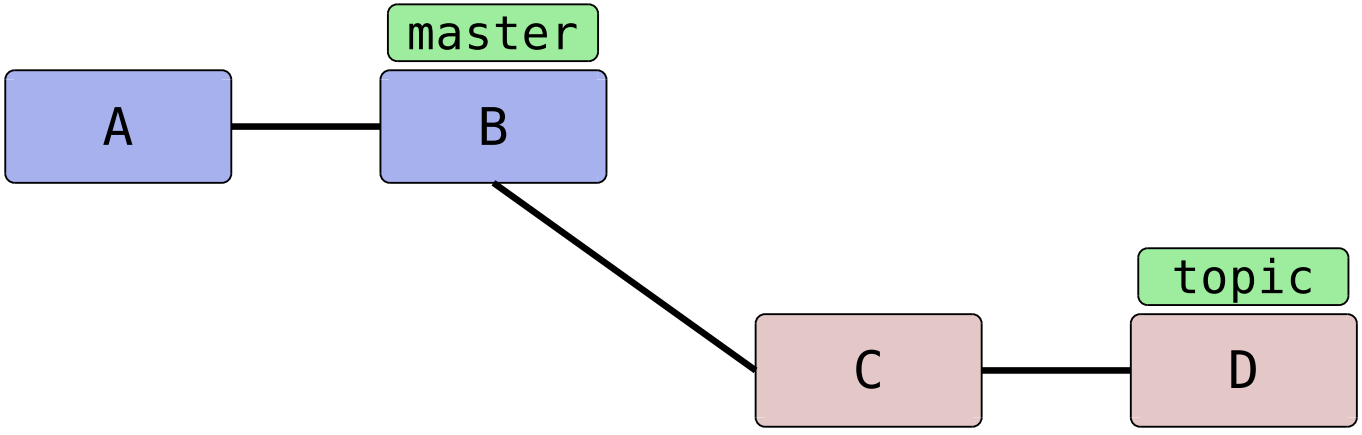
Overview

Merge

`git merge` merges two or more branches

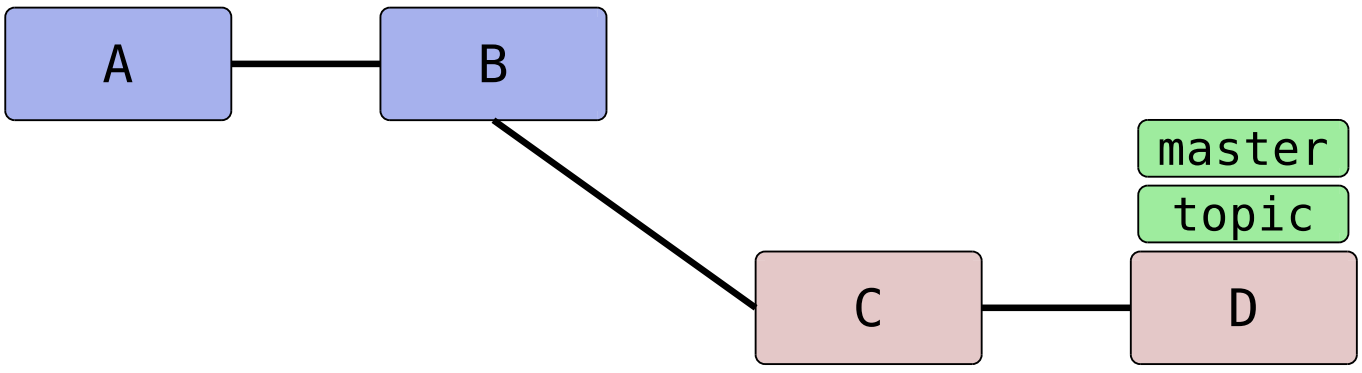
- *Fast-Forward*: No bifurcations exists, the branch can be «moved forward»
- Otherwise, a *merge commit* is created which contains both branches as parents
  - Conflicts are resolved as part of the merge commit
  - Otherwise the merge commit is «empty»
- Future development will be based on the commits of *both* branches
  - Thus, the branches can not be decoupled anymore

Before the Fast-Forward



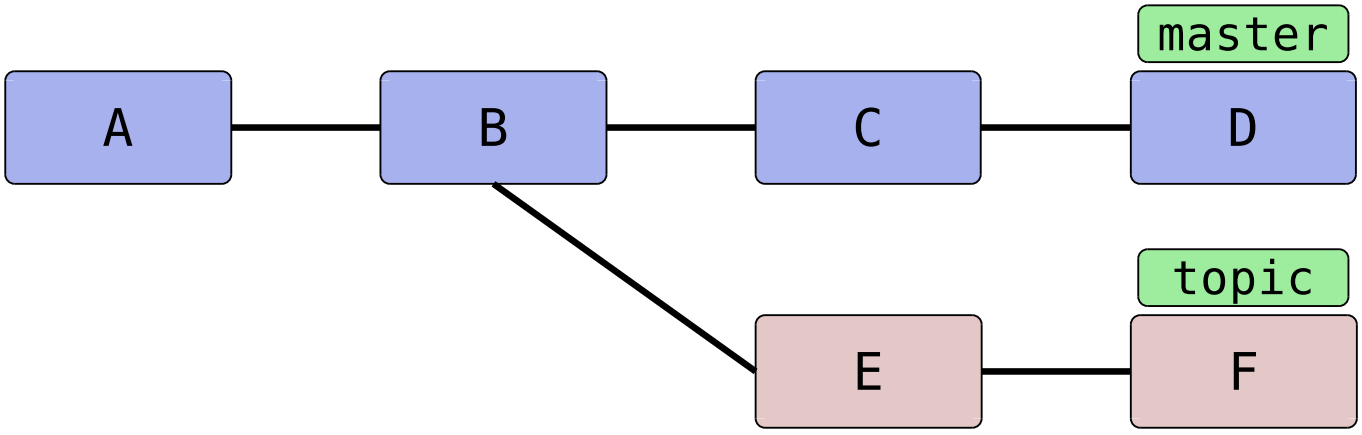
- `master is unchanged`, `topic` almost done

Afer the Fast-Forward



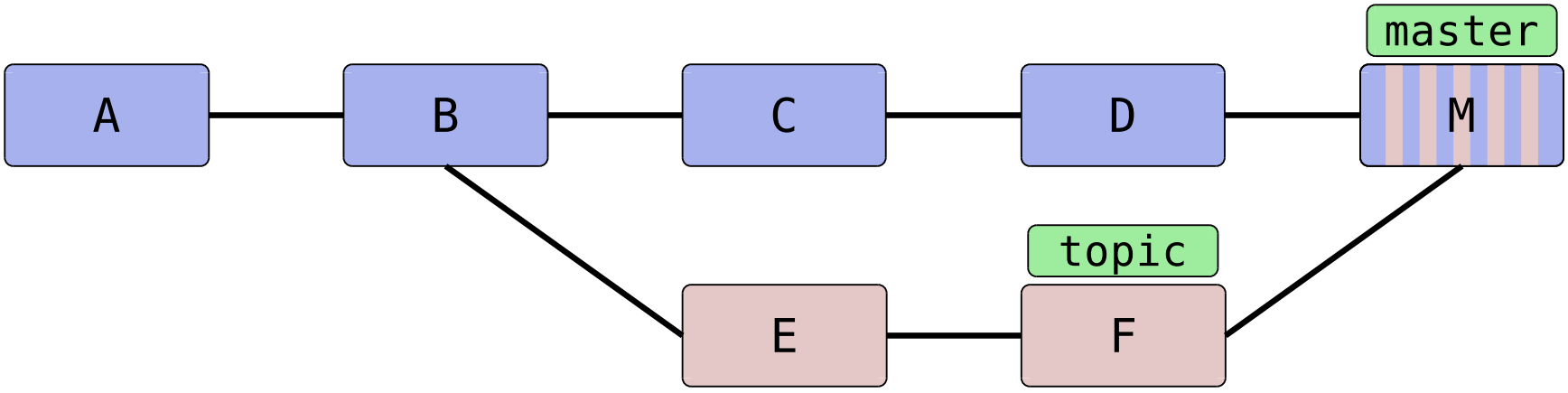
- `master` is fast-forwarded

Before the Merge



- `topic` is almost ready, and should be integrated into `master`

After the merge



- Execute in `master`: `git merge topic`

Merge-Commit

- No conflicts: merge commit does not introduce changes
  - But it has two or more «parents»
- Description is created automatically
  - Alternative commit messages can be supplied with `git merge -m` *`message`* `...`

Include a summary of all commits bt default

```
git config --global merge.log true
```

Merge Strategies

Git has many strategies to perform merges, the two most important are:

- **recursive**
  - Standard strategy: a 3-way merge, will place markers into files in case of conflicts
- **octopus**
  - Strategy to merge multiple branches, will abort in case of conflicts

Conflict Resolution: Manually

- `git checkout master`
- `git merge` *`topic`*
  - Will abort with a conflict in *`file`*
- `$EDITOR` *`file`*
  - Search for the markers >>>>>, <<<<< and =====
  - Resolve the conflict
- `git add` *`file`*
- `git commit`
  - Describe the conflict and the resolution

Conflict Resolution: Mergetool

Configure a mergetool

```
git config --global merge.tool vimdiff
```

- `git merge` *`topic`*
  - Will abort with a conflict in *`file`*
- `git mergetool`
  - Resolve conflict
- `git commit`

Conflict Resolution: Automatic

- The merge strategy *recursive* knows two options to automatically resolve a merge
- In case of conflicts...
  - **ours** takes the changes from the current branch
  - **theirs** takes the changes from the branch being merged

Use changes from `master`

```
git checkout master
git merge -X ours topic
```

use changes from `topic`

```
git checkout master
git merge -X theirs topic
```

Overview

Remote-Repositories

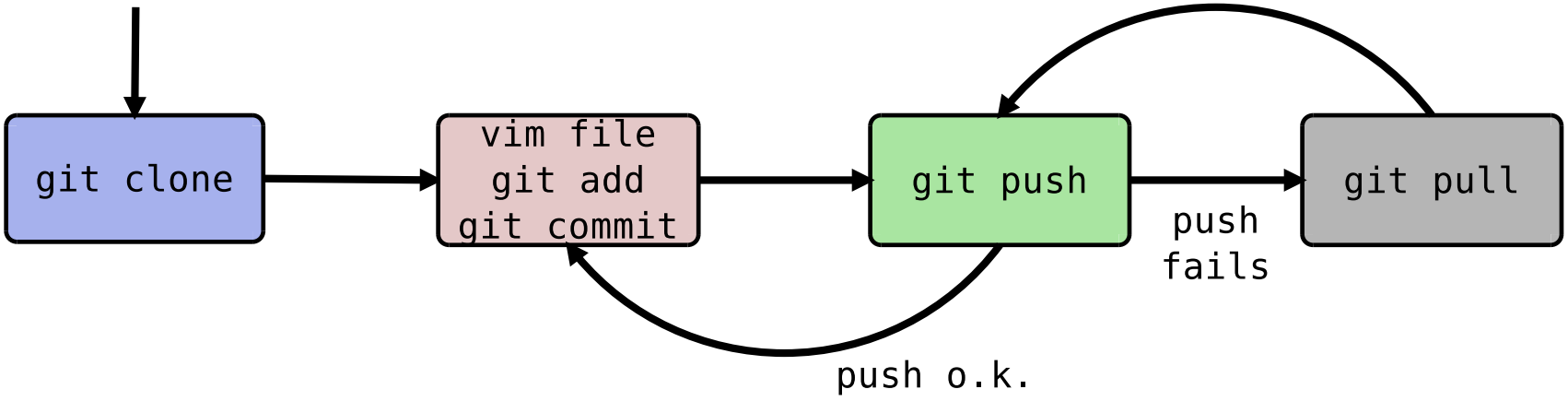
- All «other» repositories are known as *remote repositories*
  - The central (blessed) repository
  - Repositories belonging to other developers
  - Copies (clones) of the repository
- Abbreviation: **Remotes**

- In the simplest case (e. g. after a `git clone`) only a single remote by the name of `origin` is configured

Clone an existing project

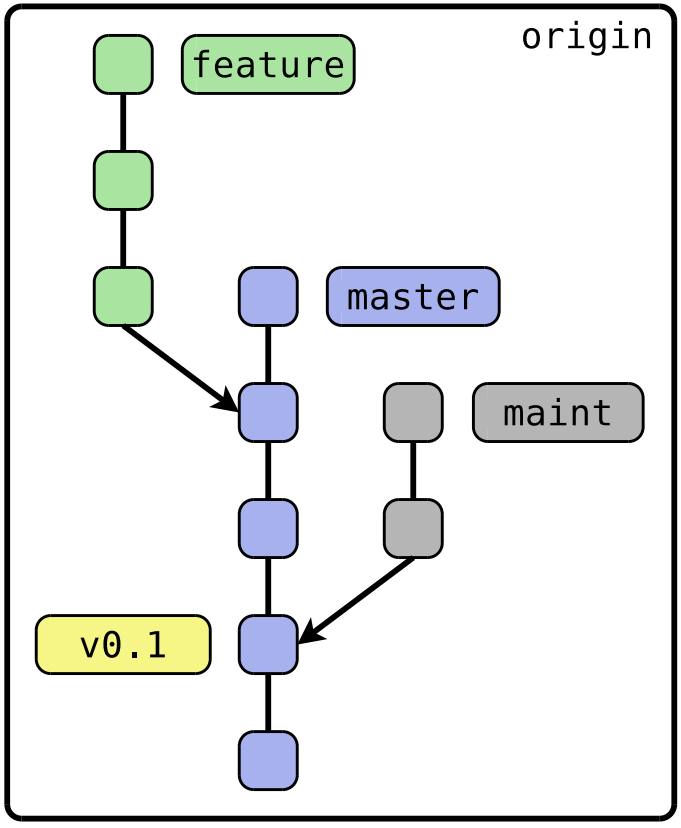
```
git clone https://github.com/fcl13/week3.git
```

Typical Workflow

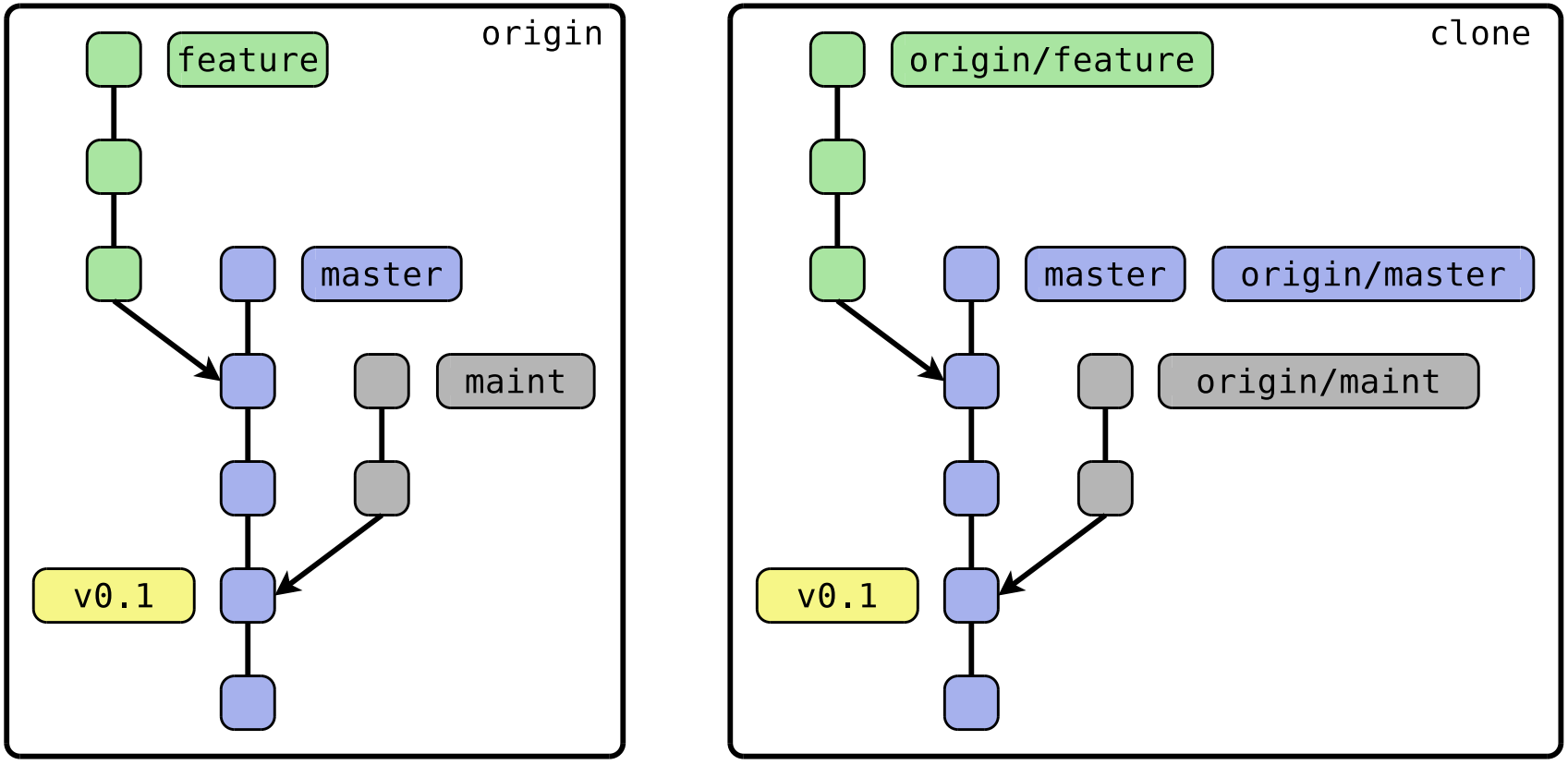


1. Local changes
   - `vim` *`file`*
   - `git add` *`file`*
   - `git commit -m "`*`msg`*`"`
2. Publish changes
   - `git push`
   - If `push` fails
   - `git pull`, then `git push`

Before `git clone`



After `git clone`



`git push` – Pushing Changes

Pushing changes from *branch* to *remote*

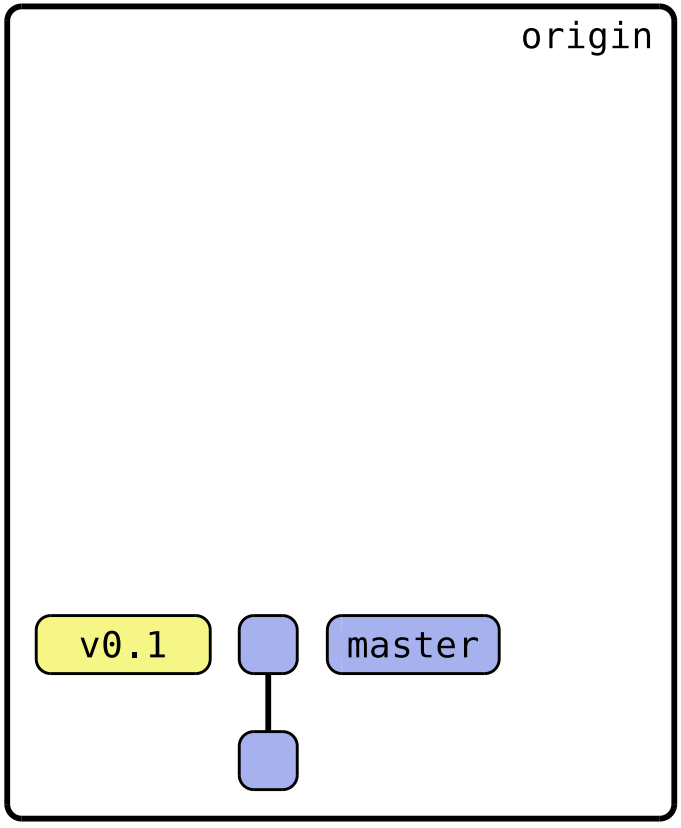
```
git push remote branch
git push origin master
```

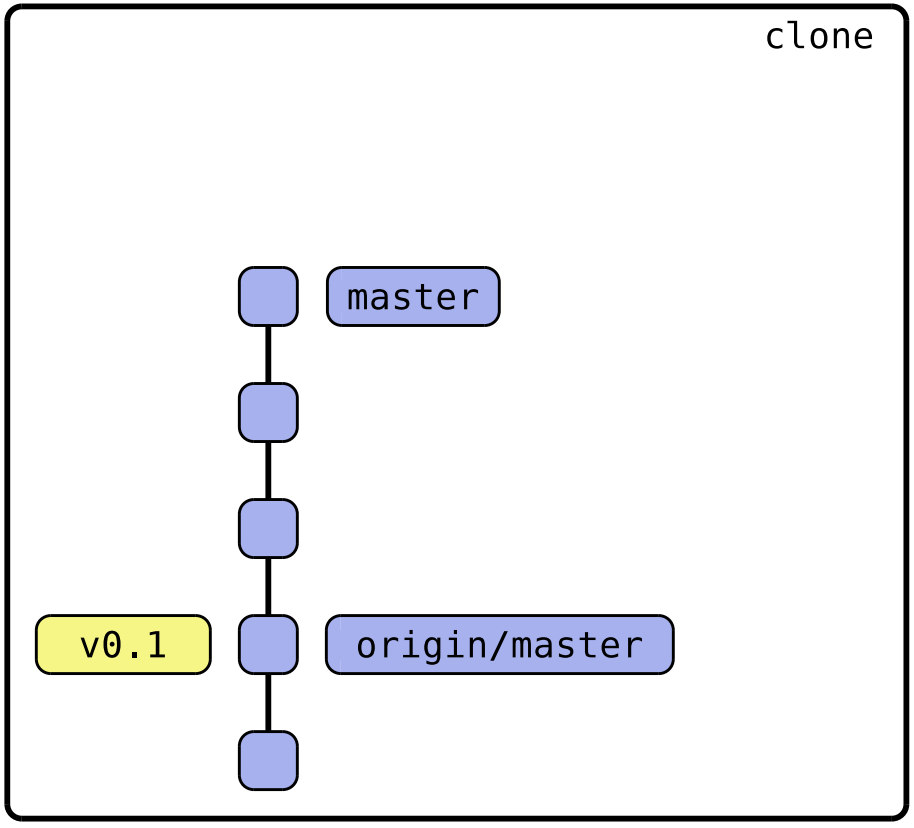
Pushing a local branch to a differnt remote branch

```
git push remote branch-local:branch-remote
```

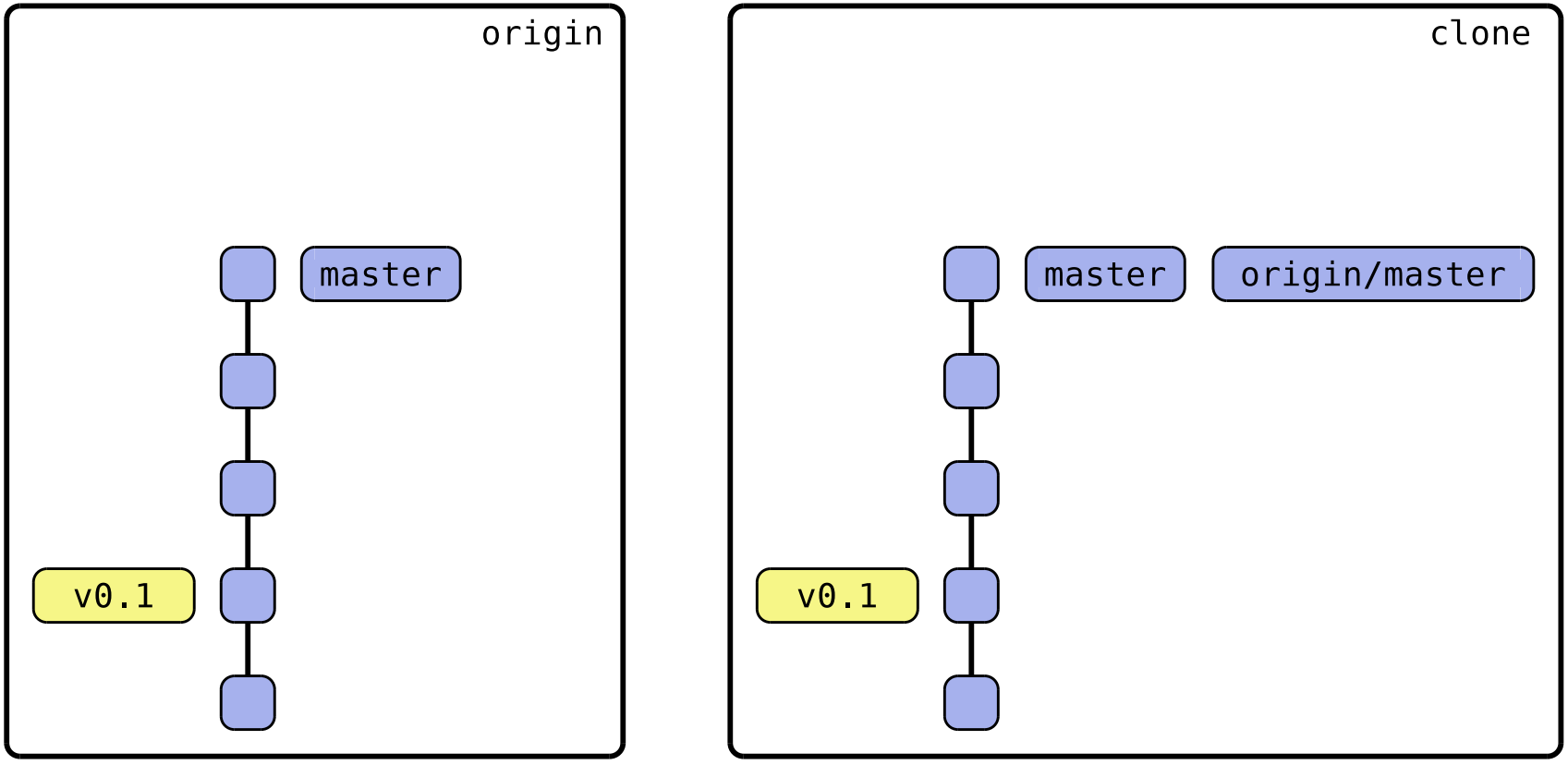
- Git will attempt to «fast-forward» the remote branch if it exists
- When this fails, an error message will be displayed
- Git will create the branch, if it does not exist

Before `git push`





After `git push`



`git pull` – Pulling Changes

Pull changes from *branch* in *remote*

```
git pull remote branch
git pull origin master
```

- Changes will be merged into the currently checked out branch

`git fetch` – Update Remote-Tracking-Branches

- Synchronize local repository with remote repository
  1. Download changes
  2. Remote-tracking-branches are updates automatically

Fetch changes from a single remote repository

```
git fetch remote
```

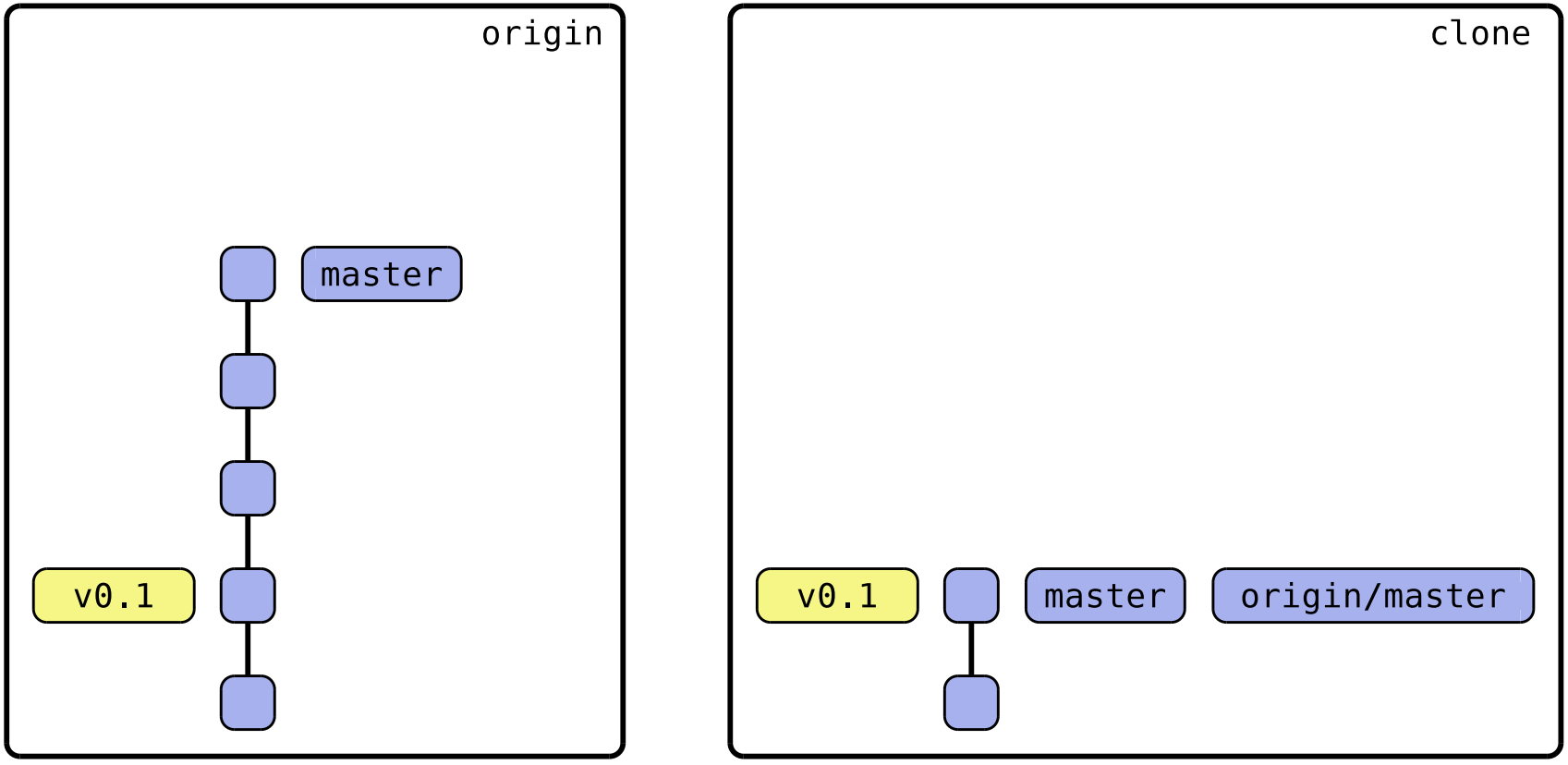
Fetch changes from all remotes

```
git remote update
```
(alternatively)

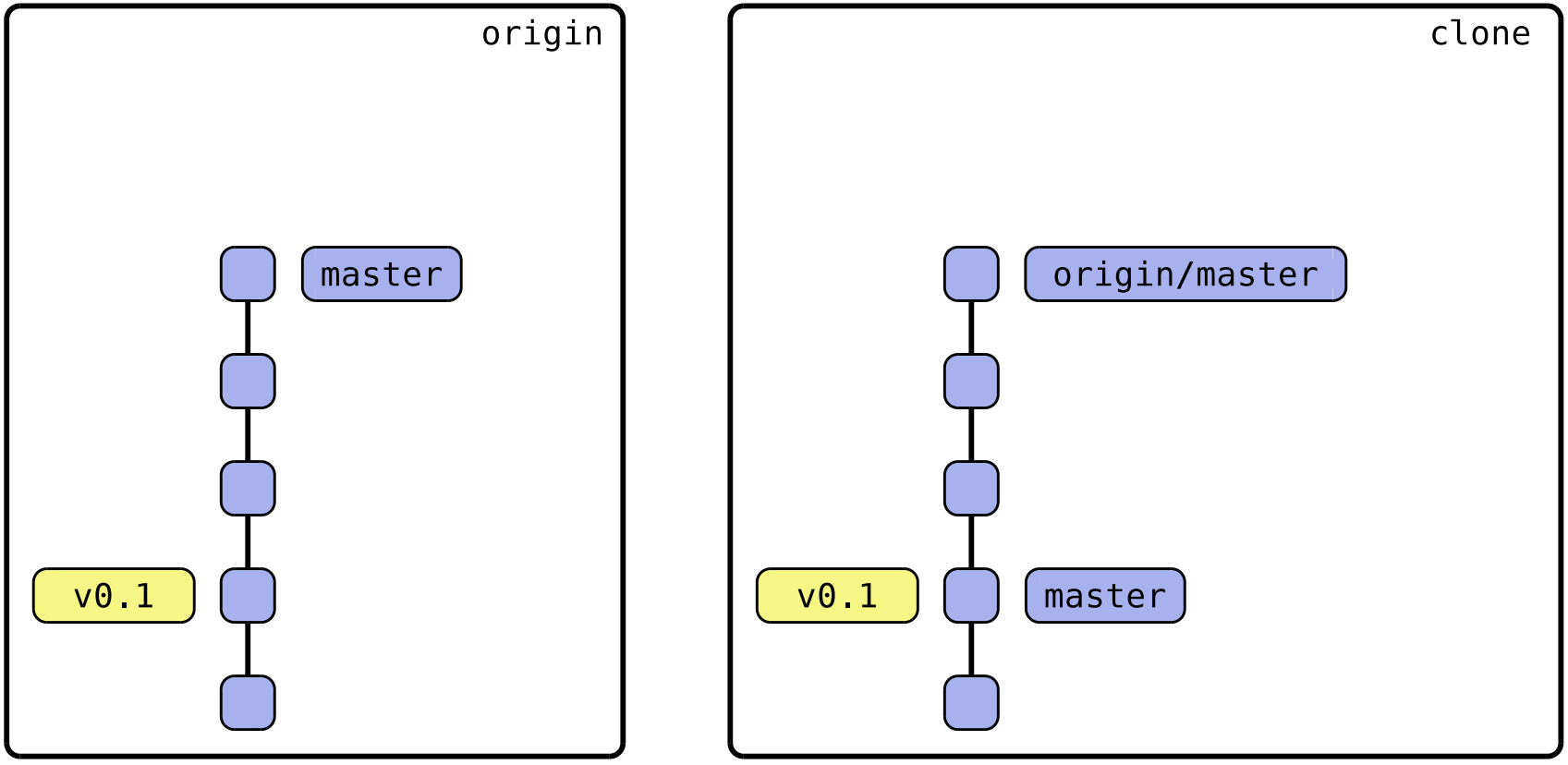
```
git fetch --all
```

- After the update, changes may have to be merged into the local branches
  - → `git merge` or `git rebase`

Before `git fetch`



After `git fetch`



`git pull = fetch + X`

`git pull` combines two commands:

1. fetch changes and update tracking branches
   - `git fetch`
2. merge tracking branches
   - `git merge` or `git rebase`

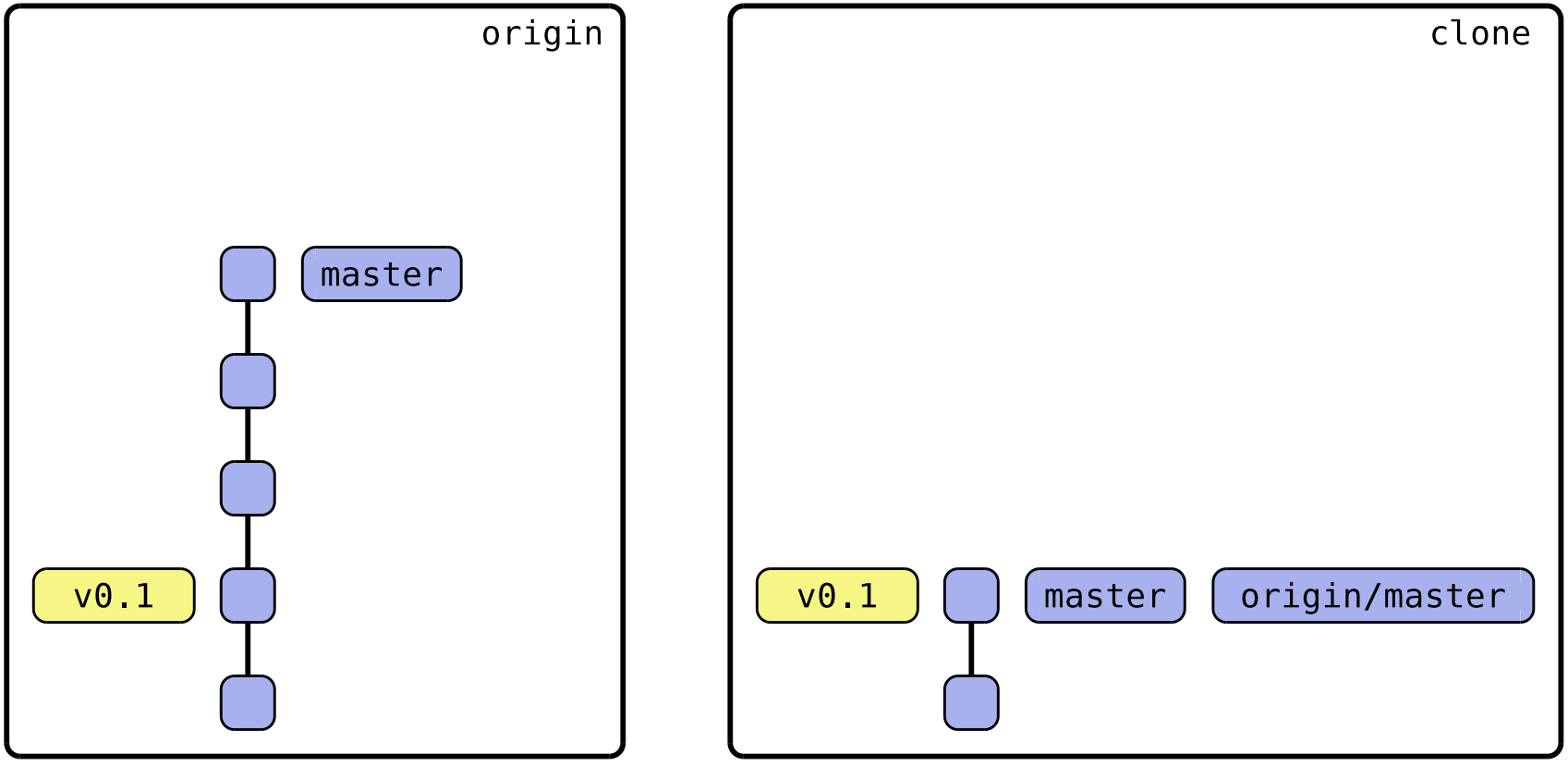
Fetch + Merge

```
git pull
```

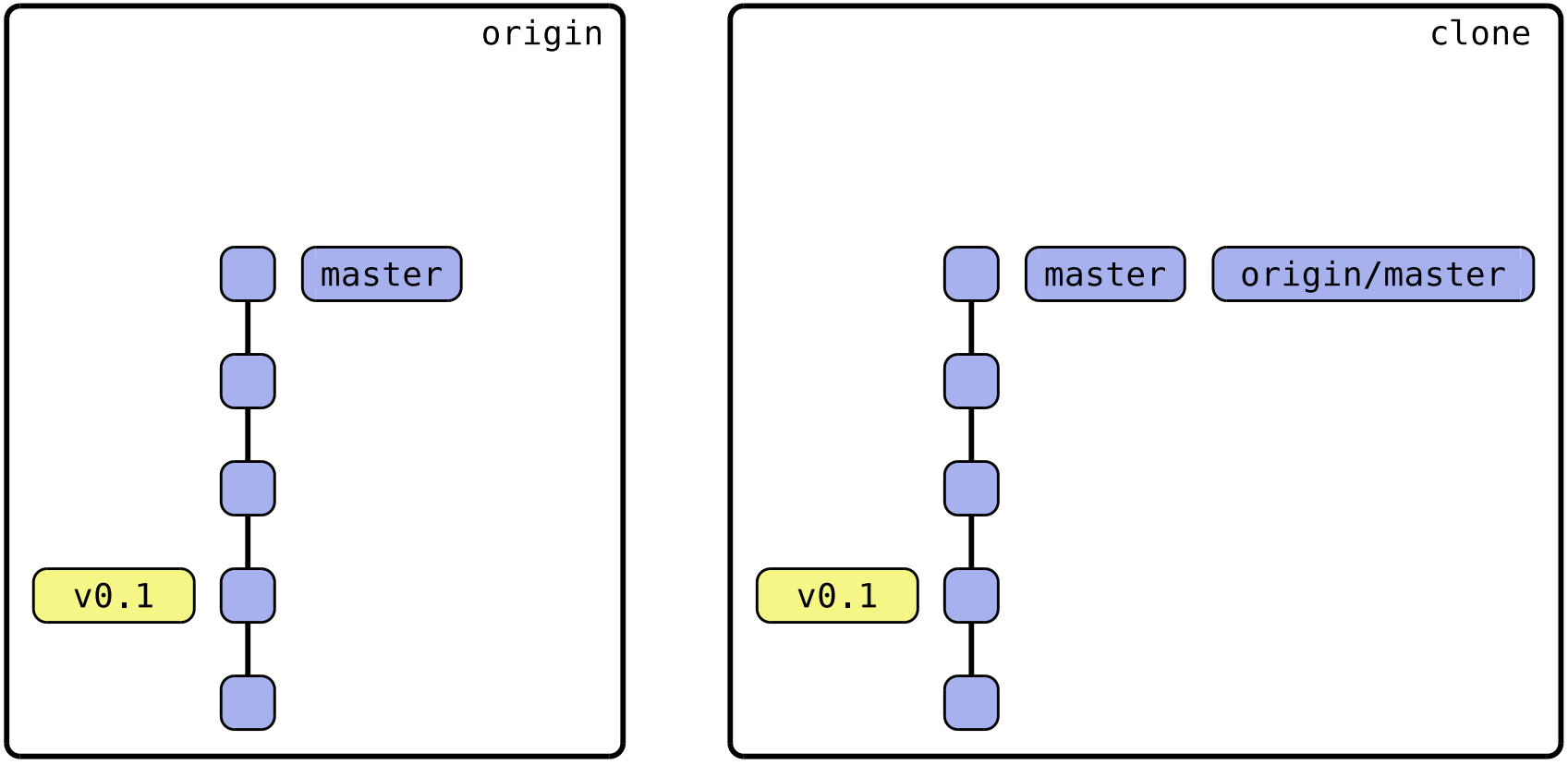
Fetch + Rebase

```
git pull --rebase
```

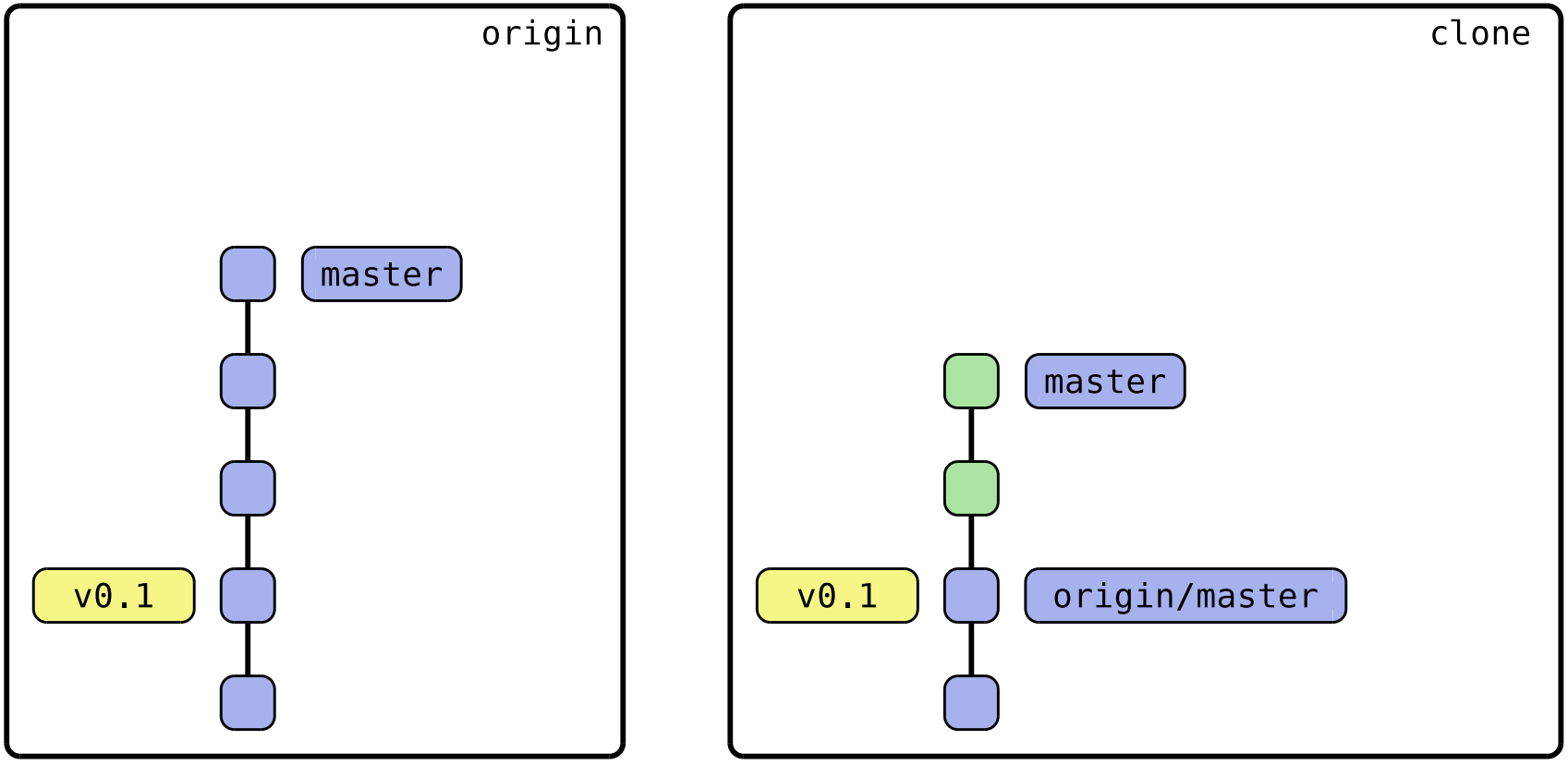
Before Fast-Forward Pull



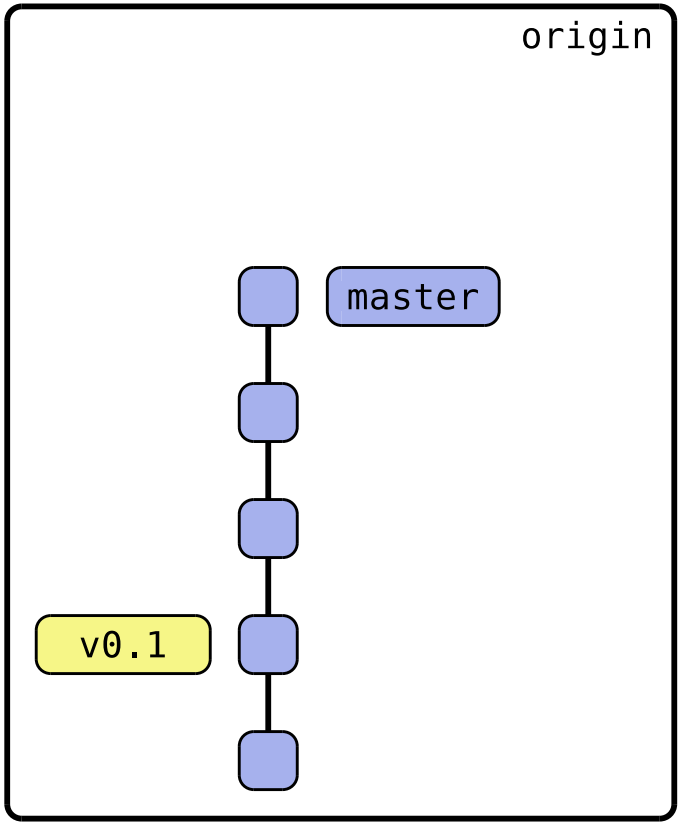
After Fast-Forward Pull

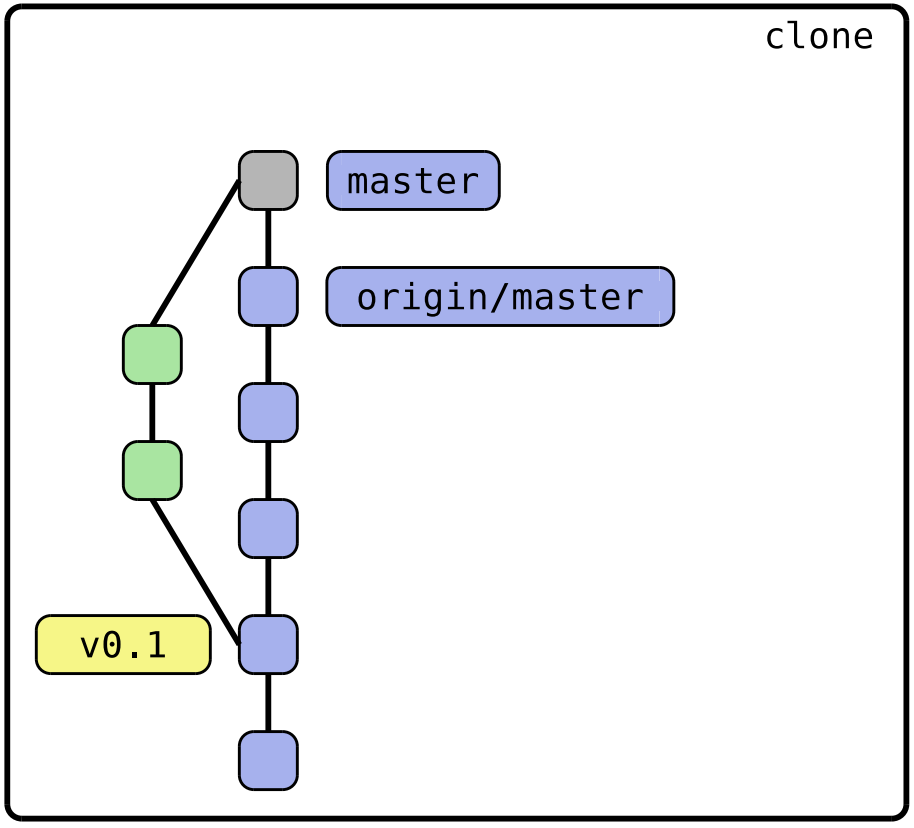


Before Merge Pull

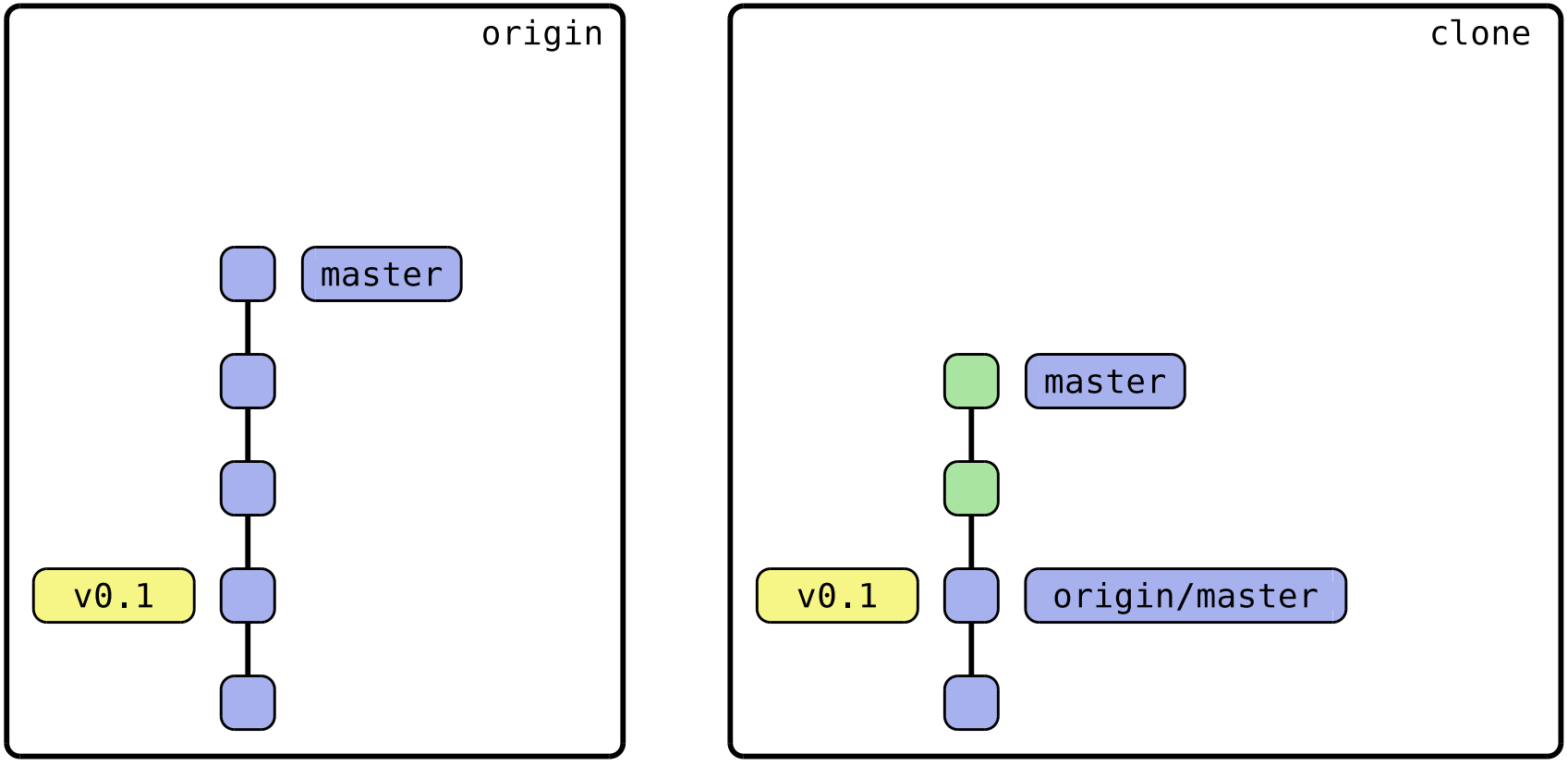


After Merge Pull

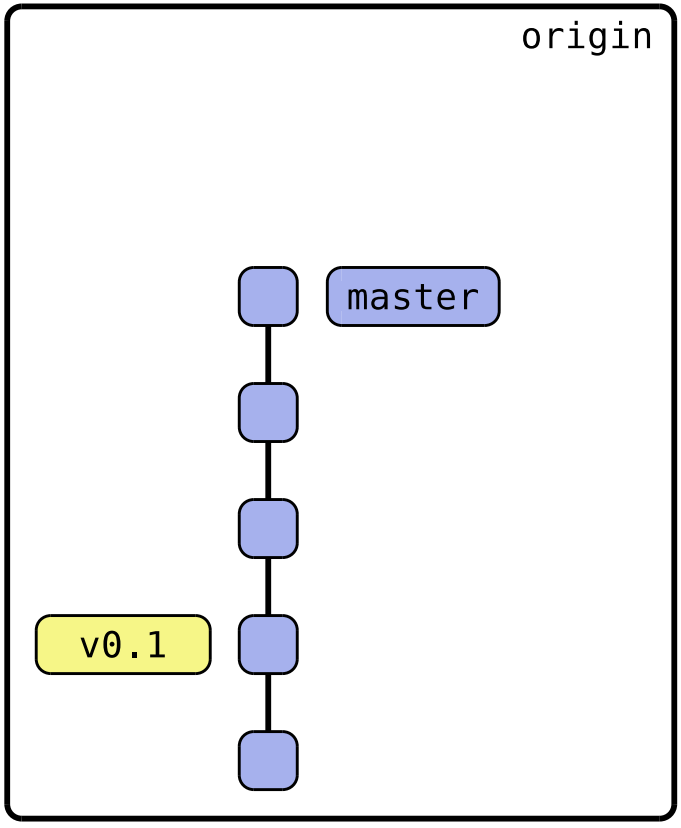


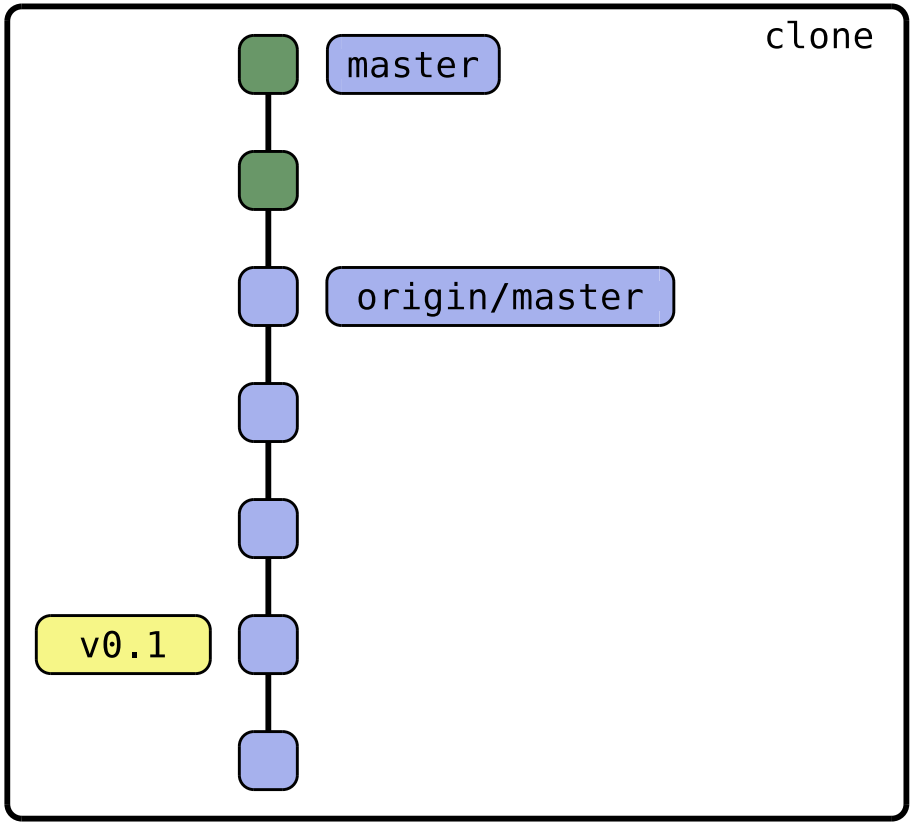


Before Pull + Rebase



After Pull + Rebase

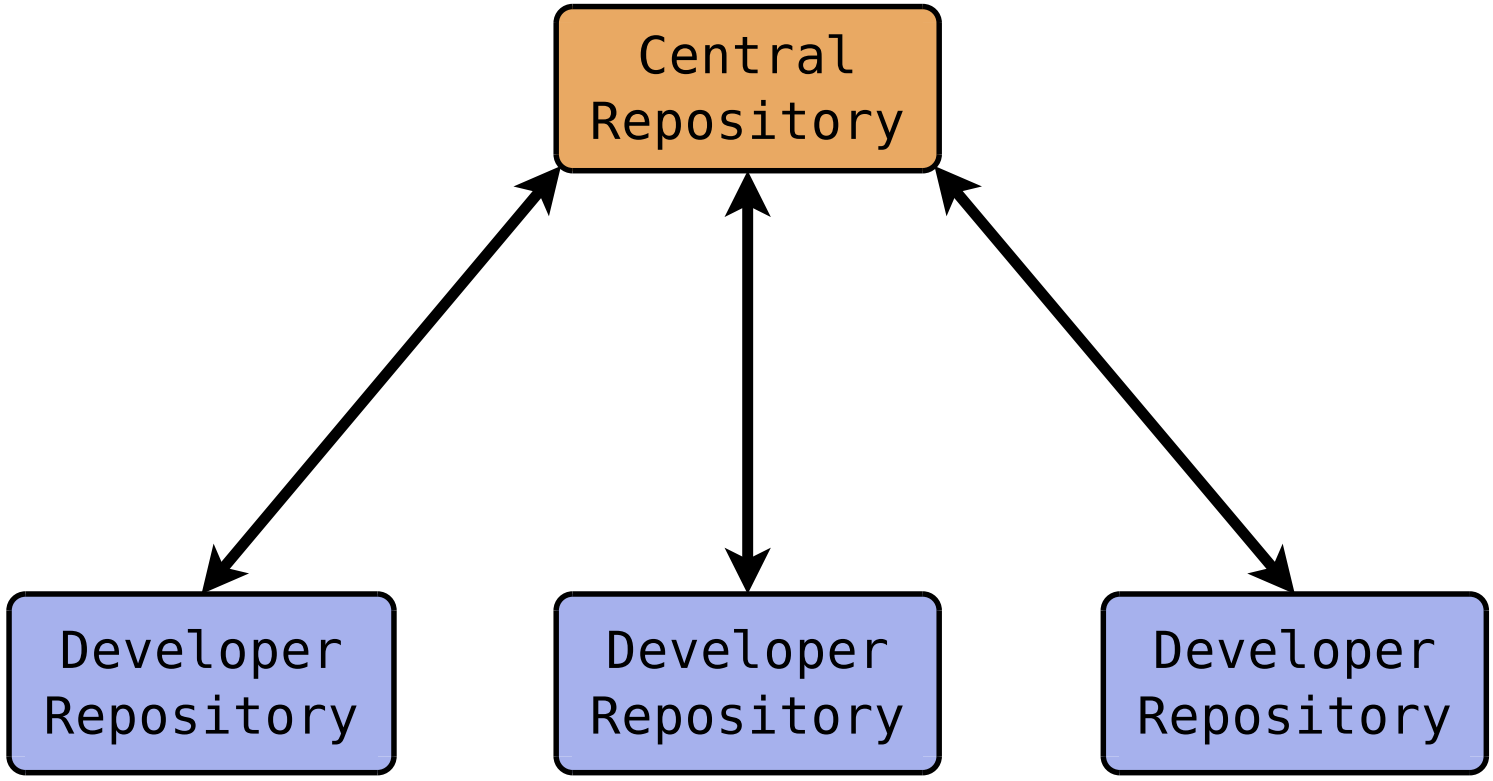




Out into the Wild World!

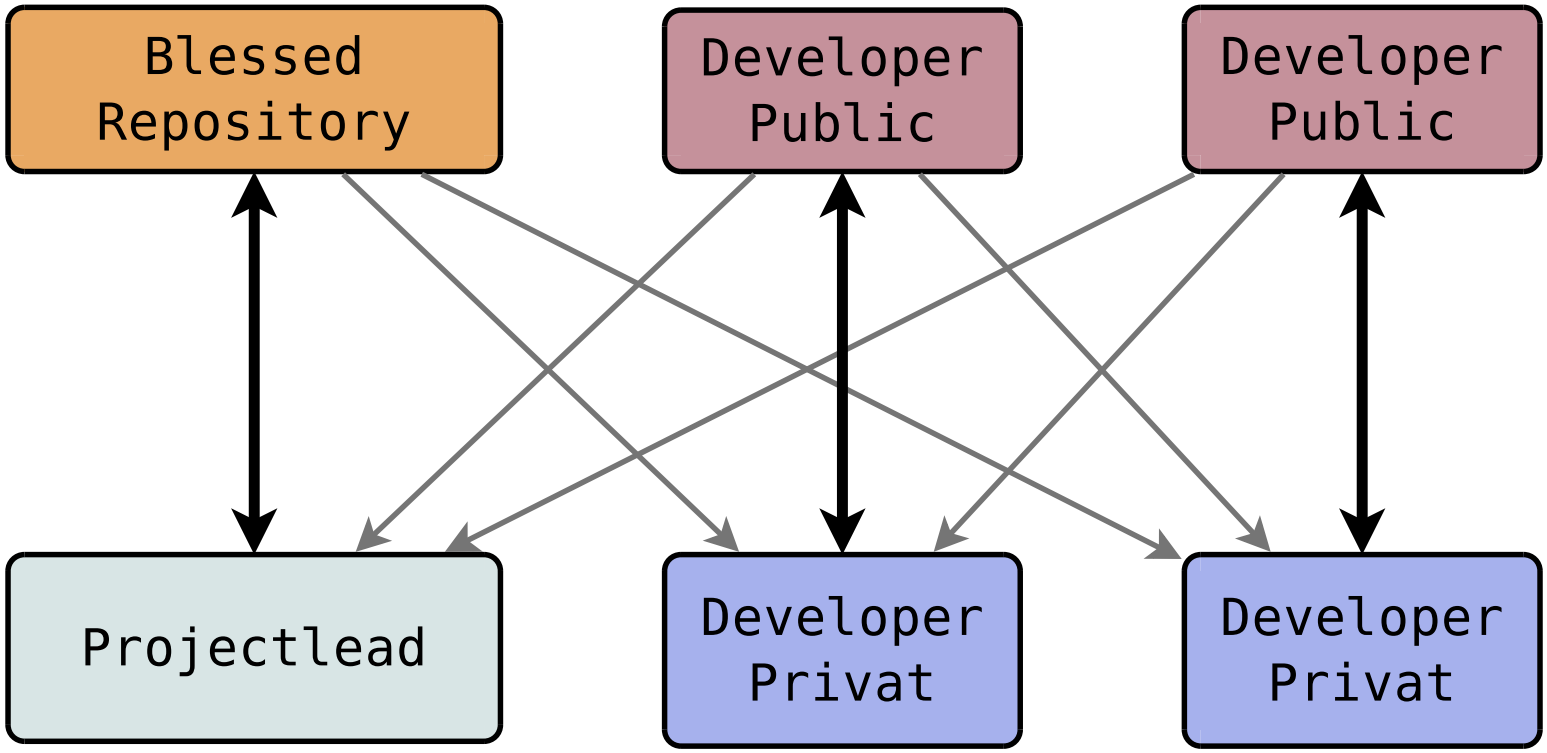
- Goal: Exchange work with other developers!
- There is no need for a *central* server due to the distributed architecture of `git`
- The developers needs to agree on a *Workflow*
  - Centralized
  - Public developer repositories
  - Patch-queue by email
  - ... or everything mixed up :)

Centralized



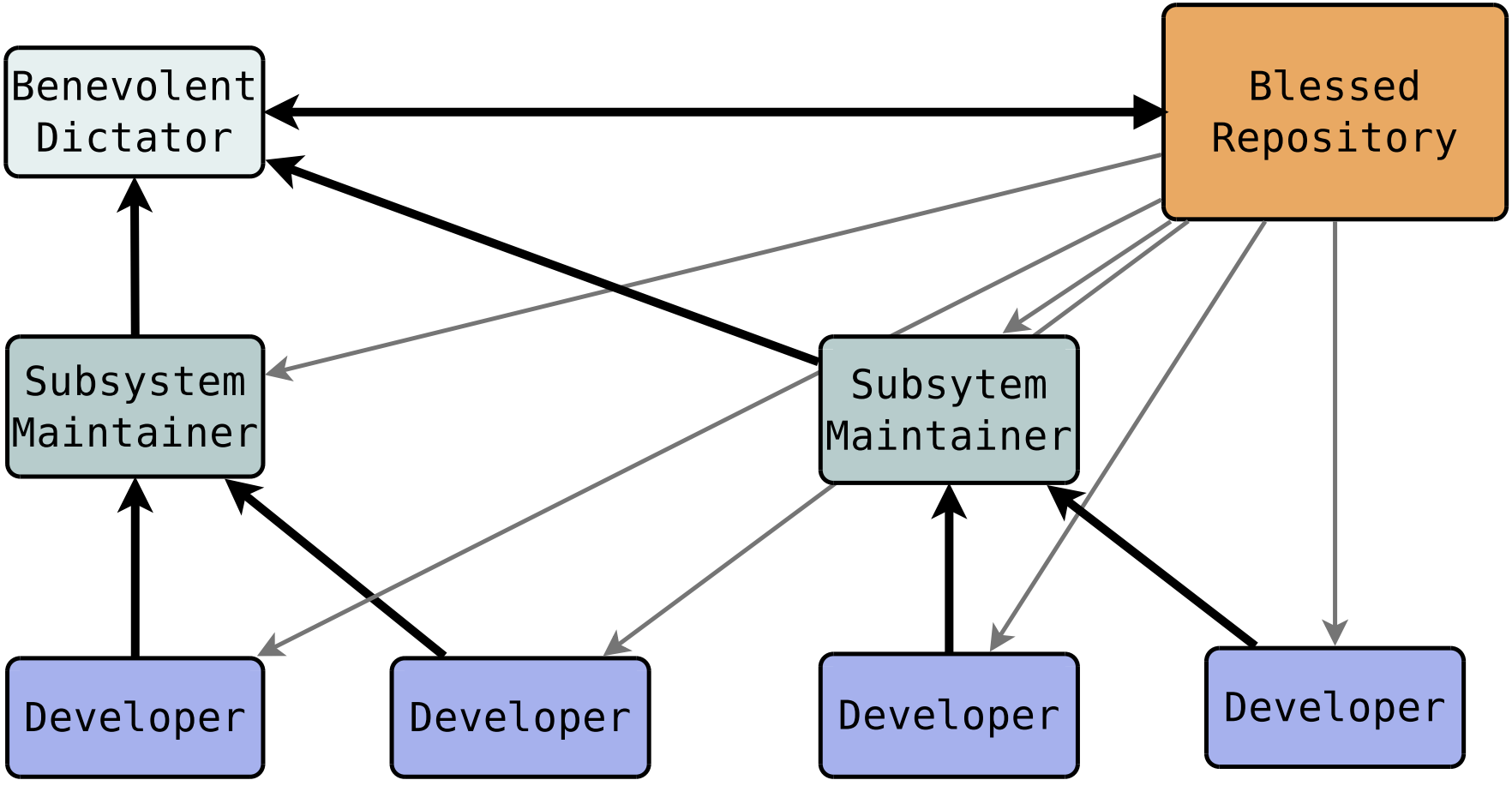
- A single central repository
- All developers have write access

Public Developer-Repositories



- One public repository per developer
- The project leader(s) integrate(s) improvements

Patch-Queue by Email



- Extensively used by the Kernel and Git itself

Github

Github Concepts

- Github is **developer centric**
- ... but there are **organizations** too

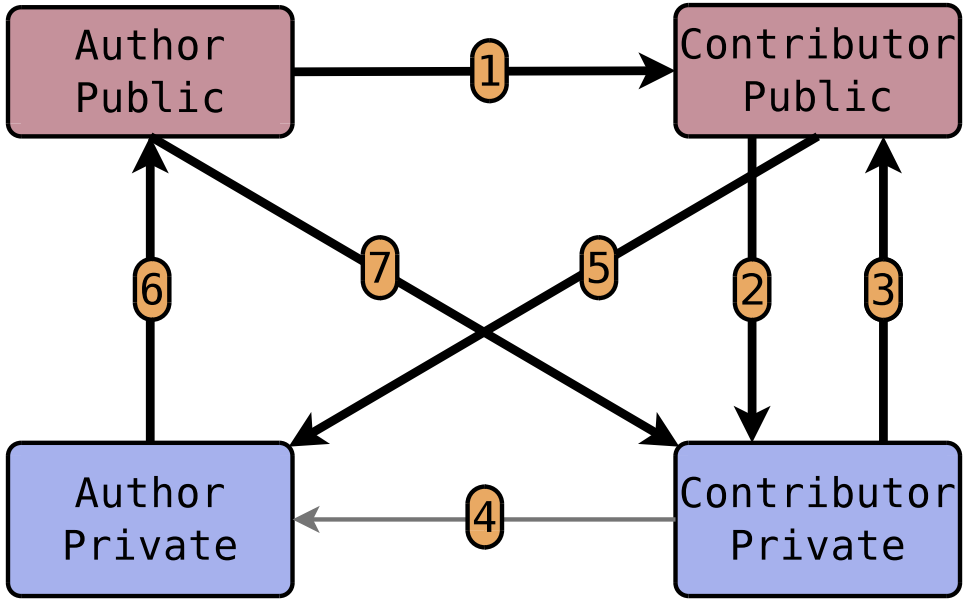
- **Fork**: your own clone on Github
  - Can be considered a public developer repository
- **Pull-Request (PR)**: used to contribute changes back
  - Including online code-review

- Social features:
  - **follow** others and be **followed**
  - **star** and **watch** repositories

Github Features

- Repositories can be equipped with
  - **issue-tracker**
  - **wiki**
  - **RSS feeds**
- Various analytical tools incl. graphs:
  - Contributors over time
  - Commit activity
  - Code frequency

The Github Workflow



Contributor:

1. .. **forks** a repository
2. .. **clones** the repository and makes some changes
3. .. **pushes** the changes to his fork
4. .. Issues a **pull-request**

Author:

5. .. **pulls** changes from contributor's fork, reviews the changes and merges them
6. .. **pushes** to his public repository

7. Contributor updates his clone (and fork)

Obtaining a free educational repository on GitHub

- For future exercises and project work
- Github offers educational accounts for students
- Micro plan: 5 private repositories

`https://github.com/edu`

- Please obtain one for next week

Overview

Exercise: Configure Git

1. Use `git config` to set username and e-mail address
2. Activate colorized output (`color.ui auto`)
3. Query the settings from the command line
4. Check the contents of `~/.gitconfig` to see the effects of your commands

Exercise: Initialize a Repository

1. Use `git init` to create an empty repository
2. Create two files with content, and observe the output of `git status`
3. Add them to the index using `git add`
4. Use `git commit` to create you first commit
5. Observe again the output of `git status`

Exercise: Familiarize Yourself with the Index

1. Create another file and view the output of `git status`
2. Change both files, preferably at the beginning and the end
3. View the output of `git diff`
4. Attempt to use `git add -p`
5. View the output of `git diff --staged`
6. Reset the index using `git reset`
7. View the output of `git status` again

Exercise: Creating Commits

1. Create a couple of commits
2. Familiarize yourself with `git commit -m 'message'` and `git commit -a`
3. Attempt to modify a commit using `git commit --amend`
4. Create an empty commit using `git commit --allow-empty`

Exercise: View History

1. View the work you've done so far using `git log`
2. Create a new commit and view `git log` again
3. View only the *n* newest commits using `git log -`*n*
4. View single commits using `git show` e. g. `git show HEAD^^`
5. Modify the output of `git log`, use:
   - `git log --oneline`
   - `git log --stat`
   - `git log -p`
6. Execute the following command:
   `git log --oneline --graph --decorate --all`

Exercise: Creating and Deleting Branches

1. Create two branches from the same commit
2. Create commits on each of those branches
3. View the result using

   ```
   git log --oneline --graph --decorate --all
   ```

4. Call `gitk --all`
5. List all branches with `git branch -av`
6. Create the annotated tag `v1.0`

Exercise: Merging Branches

1. Merge two branches
2. Merge multiple branches into `master` (*octopus*)

Exercise: Resolving Conflicts

1. Create conflicting changes on different branches and resolve them with a `merge`
2. Define a mergetool and uses that to resolve a conflict
3. Resolve a conflict using the strategy option *ours* and/or *theirs*