

Version Control Systems

CS350 Introduction to Software Engineering

Shin Yoo

Version Control System

- Manages any changes made to software engineering artifacts (source code, documents, web sites, etc). Also known as Source Code Management (SCM).
- Why do we need to manage changes?
 - We want a safe, secure repository of our projects.
 - We may want to go back to previous versions (to roll back mistakes).
 - We may have needs for gate-keeping incoming changes.
 - We may have to maintain multiple versions of the same system.

Requirement for VCS

- Atomic Updates: We call an operation “atomic” if it is always the case that either the entire operation takes place, or not, even when the operation is interrupted in the middle - you do not leave a “Work-In-Progress” state.
 - If a developer starts making changes to a file, others should not make concurrent modifications - otherwise operations become non-atomic.
 - One way of ensuring atomicity is to use file locks: once a file is “checked out” by someone, others cannot modify it. However, lost locks will be a problem.

Requirements for VCS

- Merging: Unless you implement file locks, multiple changes can be made to the same file. The first person will be able to save the changes; later ones will not be able to do so, because the change (the delta) they want to submit is different (the baseline has been moved).
 - Instead of overwriting, merging will pick up only the different parts, and work these parts into the new baseline.
 - However, conflicts are possible: what if the first change and the second change both work on the same part of the file? We cannot overwrite, we cannot accommodate both changes.
- Merging calls for conflict resolution.

Centralized VCS

- Revision Control System (RCS): part of UNIX. Stores the latest version in full (for faster access), and all previous versions are stored as deltas. Manages files, not projects. Uses lock mechanism, so only one user can work on a file at any given moment.
- Concurrent Version System (CVS): GPL improvement of RCS. Client/server architecture. Many people can work on the same file; however, the server only accepts changes to the latest stored version (meaning anyone who checked out files need to periodically update their local copies - conflicts can happen).
- Subversion (SVN): Apache-developed, open source VCS that is mostly compatible with CVS.

Distributed VCS

- Gets rid of the central server: now everyone stores the full repository, including all previous changes and entire history of the project.
- This is the most widely used form of VCS now.



Benefits of being distributed

- Can work when offline.
- Most operations are now faster: no need to talk to the server.
- You can keep private changes that you do not want to publish.
- Working copies all become backups.
- Allows more flexible workflows (will see later).

Downsides of being distributed

- Initial checkouts are slower and bigger: you need to move the entire history.
- Additional storage required.
- More security vulnerability: more copies of the project to attack.

git. *n.* *British Informal.*

an unpleasant or contemptible
person.

git

The most widely used DVCS right now

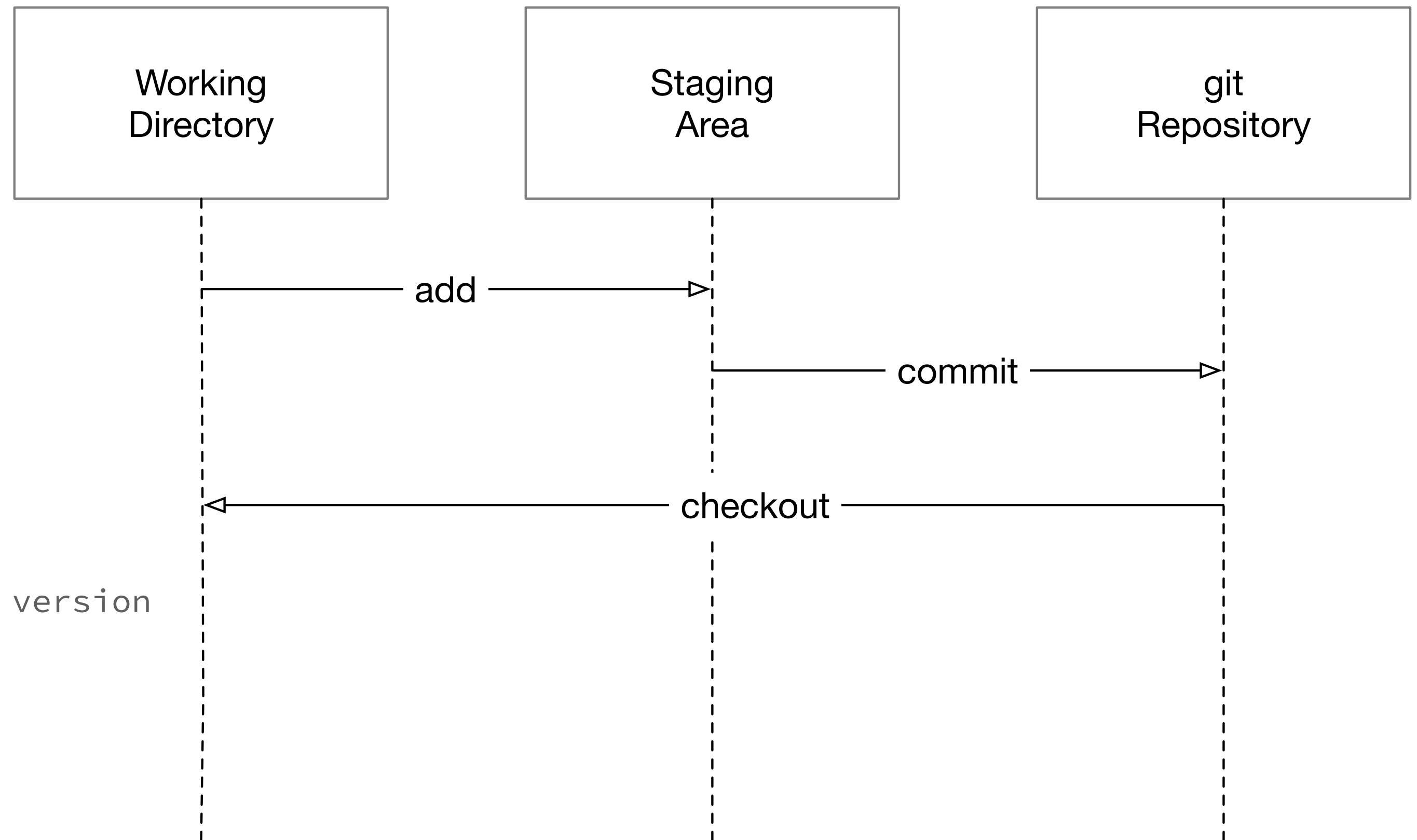
- Developed by Linus Torvalds
- Linux kernel developers used BitKeeper, a proprietary DVCS that provided a special license to linux kernel developers. After a dispute about contract violation, BitKeeper decided to stop the free license.
- Torvalds built a DVCS himself so that Linux kernel development can continue.
 - Began on 3rd April 2005; announced on 6th April; benchmarked on 29th April; adopted to maintain Linux kernel 2.6.12 release on 16th June; version 1.0 release on 21st December 2005.
- Current stable version 2.4.0 (13 March 2023)

git

A repository structure

- A directory becomes a git repository when you initialize it with `git init`
- It will now contain `.git` hidden directory, inside which git organizes repository data (history, other configurations, etc).
- The directory itself is the working directory.

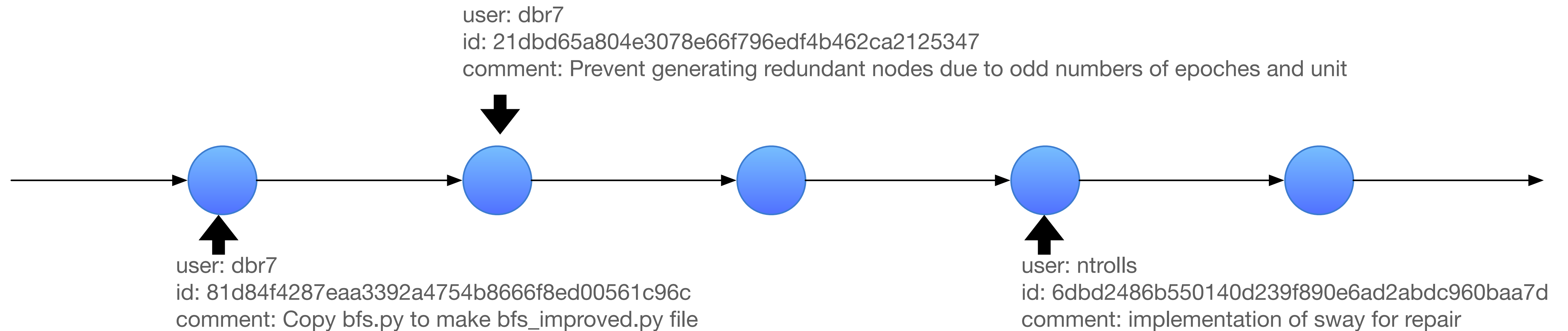
git - staging



```
$ git init  
// ... do some work  
$ git add foo.py # adds foo to the staging area  
$ git commit -m "first commit"
```

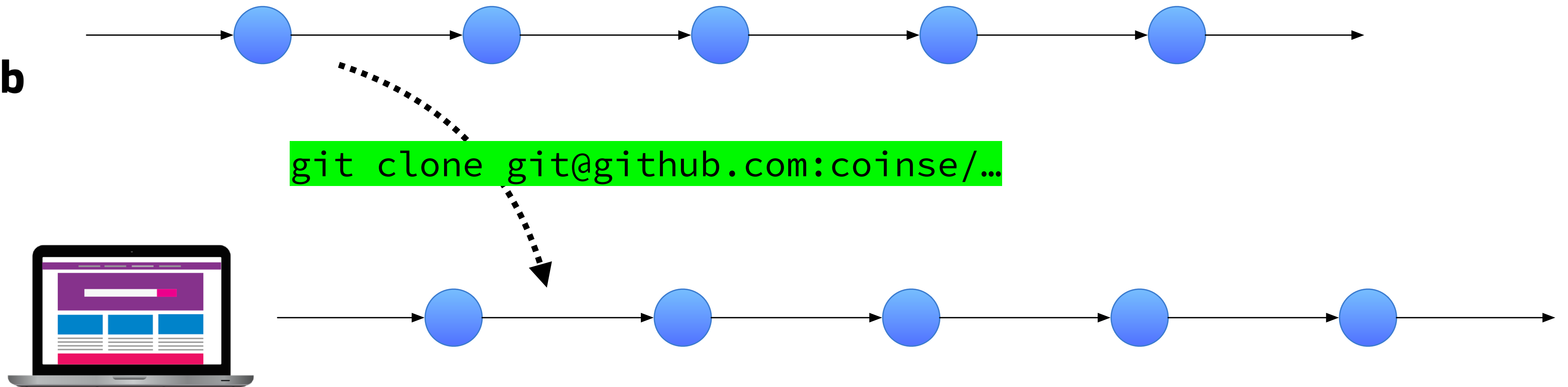
```
// realises that you've made a mistake  
$ git checkout foo.py # brings back the previous version
```

git - commit history



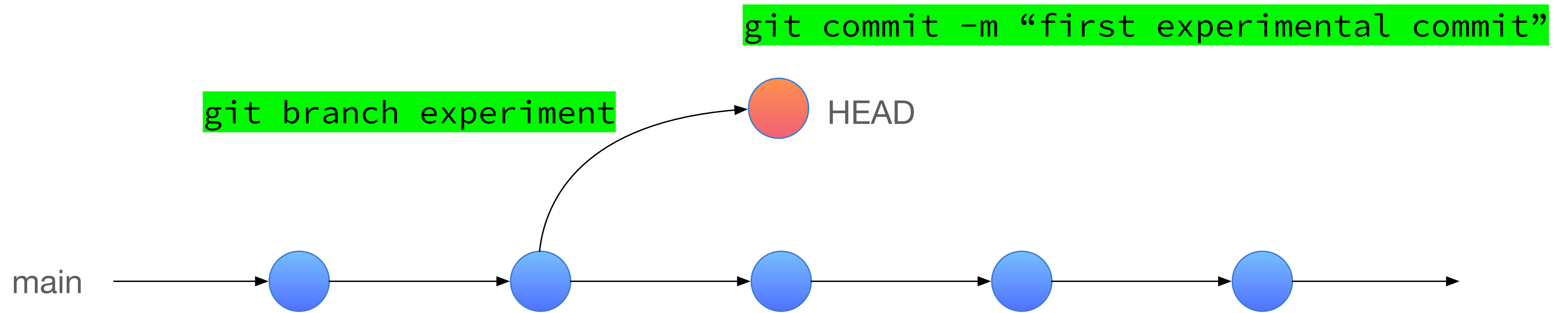
- Commit ID: an SHA-1 hash of a git commit object file, which contains the actual diff, as well as other metadata such as commit/author date, committer/author email, etc
- HEAD: a special index that points to the last effective commit

git - clone



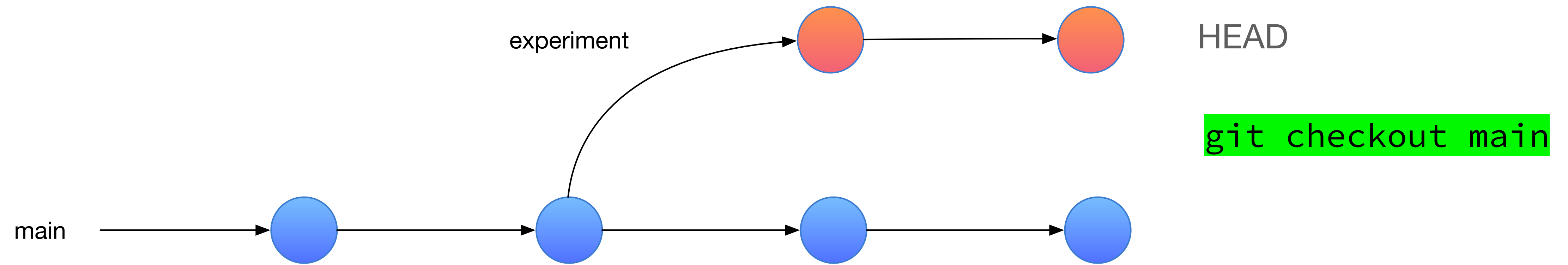
- Clone: you copy an existing repository into your local storage.

git - branching



- Creates a different timeline that branches from a specific point (typically current HEAD)
- The next commit will be made under the new branch
- `git branch --list` gives you the list of all branches.

git - checkout



- Change the contents of the working directory with those of another branch

git - status

```
ntrolls@mbp16 ~/Projects/dl-fla main git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

       modified:   stats.py

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:   sway.py

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

       newfile.py

ntrolls@mbp16 ~/Projects/dl-fla main
```

Current branch and the remote branch that it is tracking

Files that have been modified and are already in the staging area (i.e., they have been added to the staging area with `git add [filename]`)

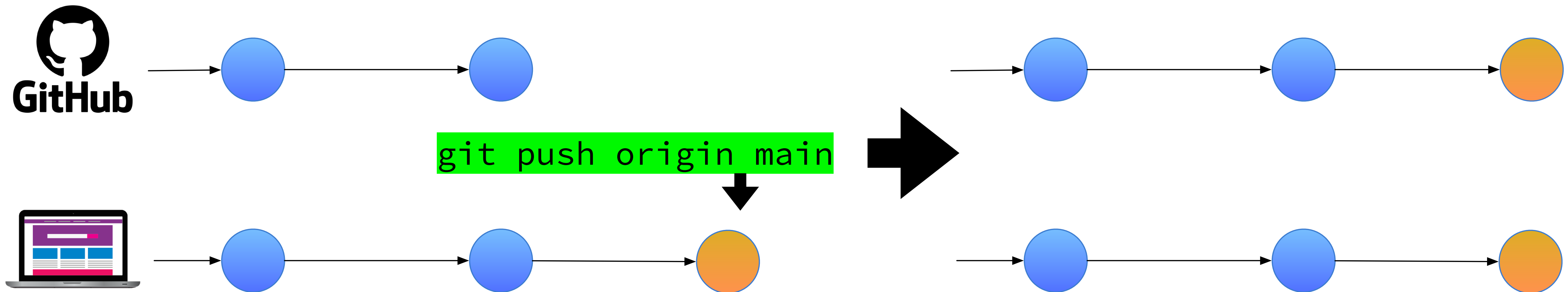
Files that have been modified but are not in the staging area

Files that are not being tracked (i.e., not part of the repository)

git - remote

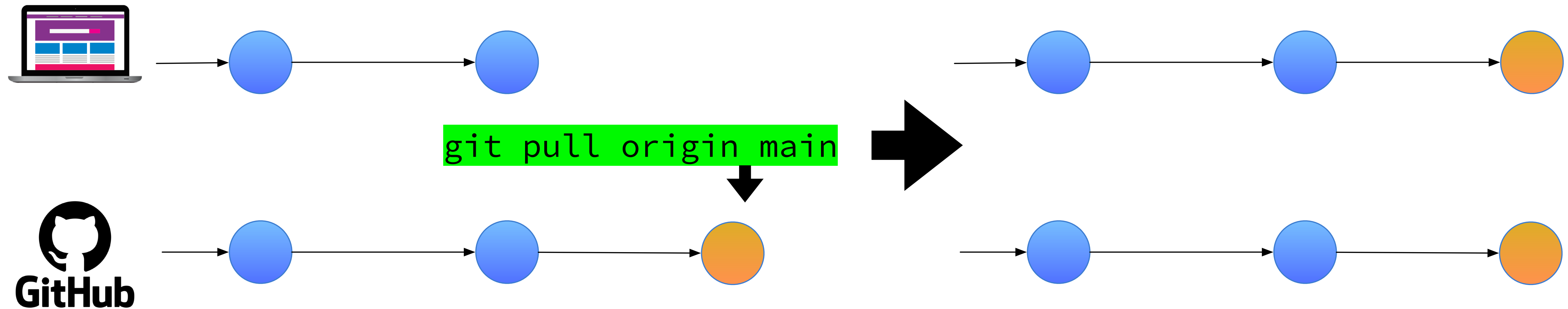
- Manages remote repositories
- `git remote [-v | --verbose]`: list all remote repositories
- `git remote add [name] [URL]`: adds the repo specific by URL under the given name
 - “origin” is the conventional name given to the upstream repo. If you clone a repo from GitHub for example, the local cloned repo will point to the GitHub repository as the upstream with “origin”
- `git remote rename [oldname] [name]`
- `git remote remove [name]`

git - push



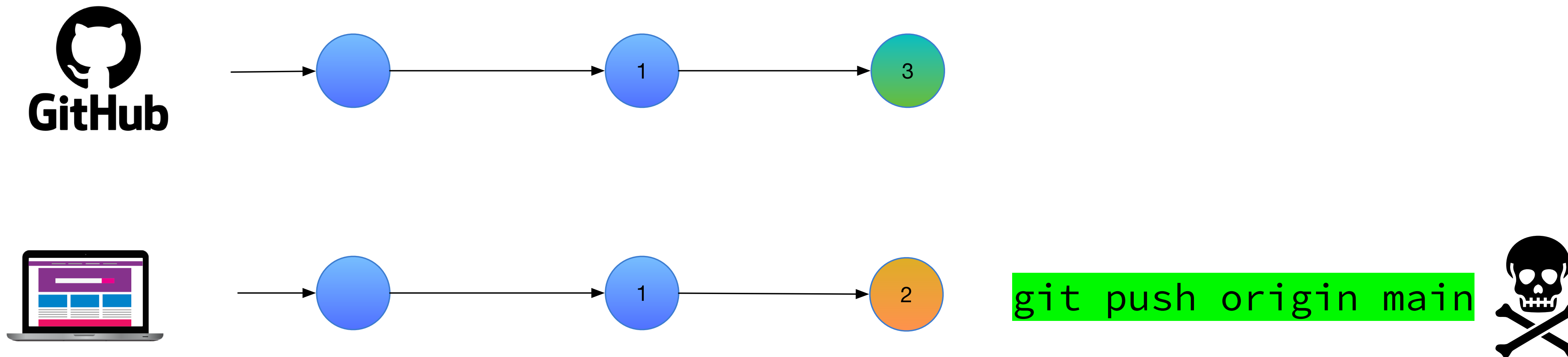
- When your local branch is one or more commits “ahead” then a remote branch, you can “push” the additional commits you made.
- Push can fail due to a conflict: you have to then “pull” the changes and push again.

git - pull



- When your local branch is one or more commits “behind” a remote branch, you can “push” the additional commits you made.
- What if local and remote heads are at different places?

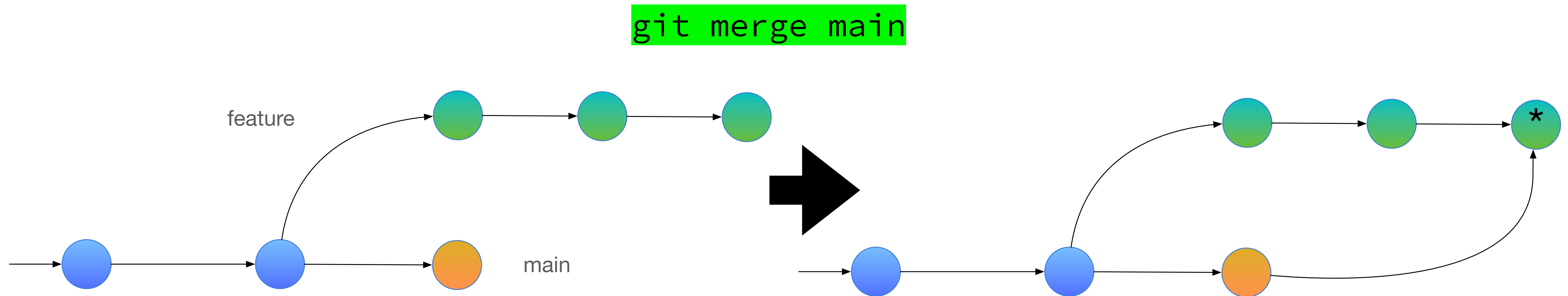
Conflict



- You pulled the remote when its head was at 1; you created changes, resulting in 2.
- Someone else pushed to origin, resulting in 3.
- Your latest change is between 1 and 2, which cannot be applied to 3 because they have different starting points.

Conflict Resolution: Merge

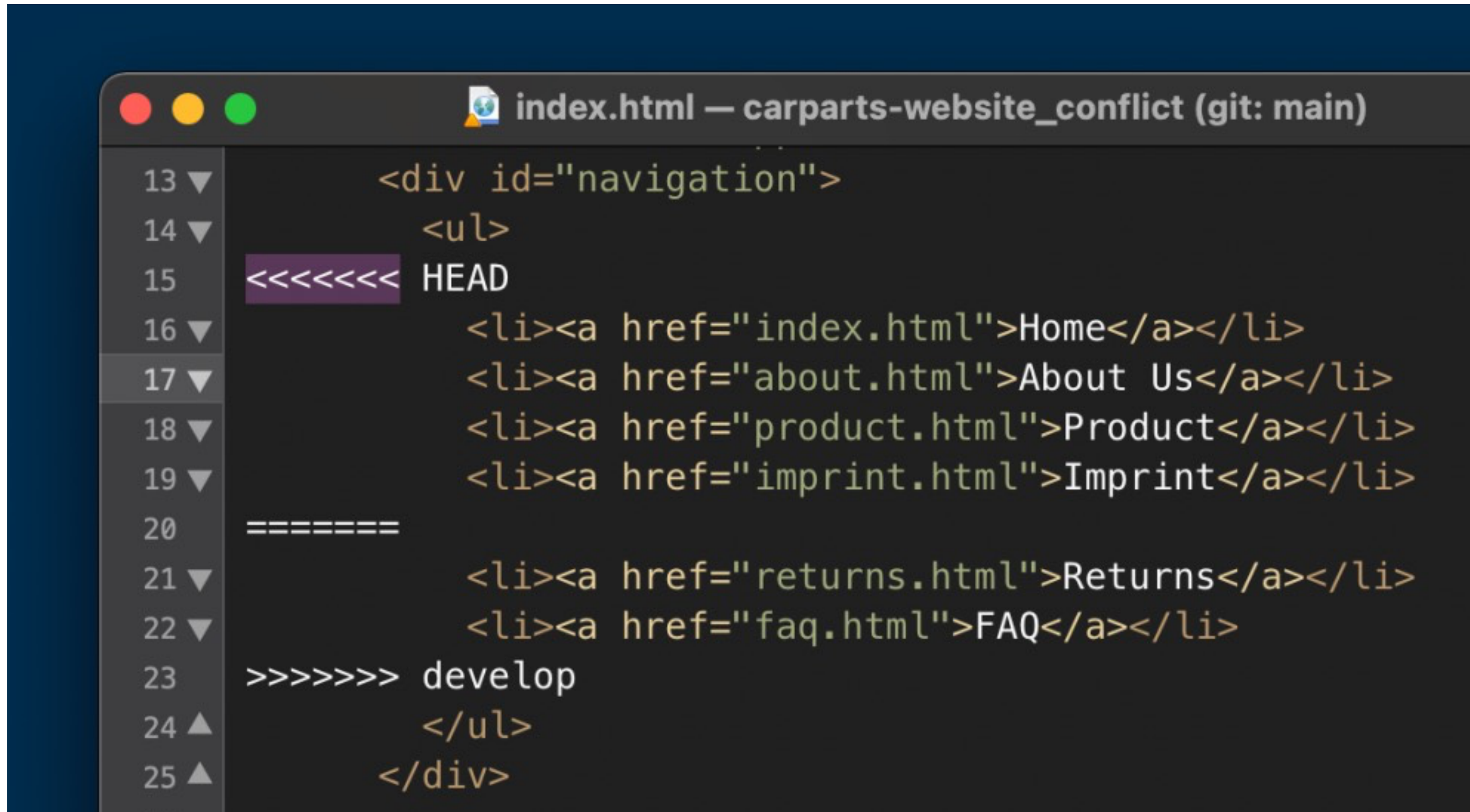
Merge integrates changes made in the other branch.



- Changes in the other branch are merged into the current branch.
- If two change sets do not overlap, merging can be done automatically.
- If they do overlap, you have to manually resolve it.
- Merge leaves a commit.

Merge Resolution

HEAD vs. the other branch: leave only one of them

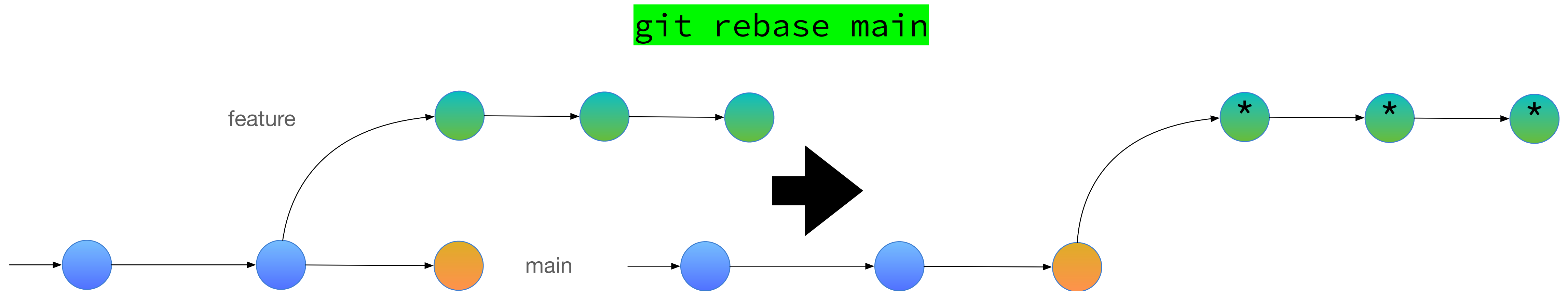


The screenshot shows a code editor window titled "index.html — carparts-website_conflict (git: main)". The code is a navigation menu in HTML. A merge conflict is resolved by choosing the content from the HEAD branch. The conflict markers are as follows:

```
13 <div id="navigation">
14 <ul>
15 <<<<<<<< HEAD
16 <li><a href="index.html">Home</a></li>
17 <li><a href="about.html">About Us</a></li>
18 <li><a href="product.html">Product</a></li>
19 <li><a href="imprint.html">Imprint</a></li>
20 =====
21 <li><a href="returns.html">Returns</a></li>
22 <li><a href="faq.html">FAQ</a></li>
23 >>>>>>> develop
24 </ul>
25 </div>
```

Conflict Resolution: Rebase

Rebase re-calculates your changes against a new baseline.



- Moves the baseline (i.e., the branching point) and applies the accumulated changes.
- Newly applied changes now have different commit IDs.
- No additional commit; however, conflicts are still possible.

git - fetch

- Your local repo can store information about remote branches: `git fetch` updates these to the latest information.
- `git pull` is in fact two commands combined: `git fetch`, followed by:
 - `merge`: conflicts are either automatically merged and committed, or you have to manually resolve
 - `rebase`: no conflicts, so your branch is simply updated

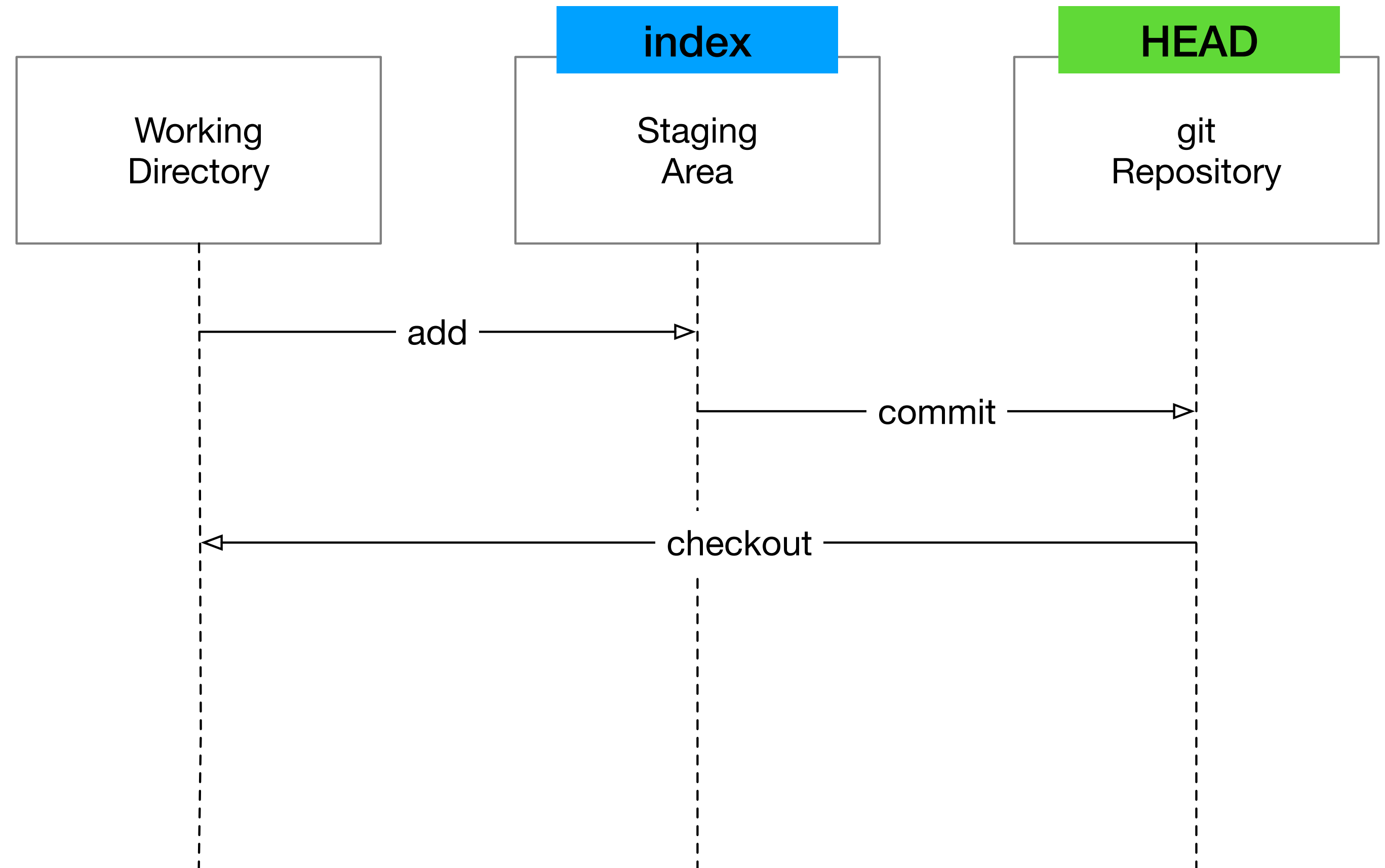
git - log

- Gives you the timeline.
- `git log --oneline --graph` will give you ascii graph representation.

```
[Terminal] $ git log --oneline --graph
* 7d4a5d1 (HEAD -> main) Merge branch 'test' into main
|\
| * 8a533f2 (test) commit in test branch
* | 2e762b0 edit in main branch
|/
* 9e5c798 Merge branch 'test' into main
|\
| * cb0d131 edit in test branch
* | 66f8e28 edit in main branch
|/
* 15177dd (origin/main, origin/HEAD) modified
* 05a1018 Create test_file_y.py
* b0ce84d Create test_file_x.py
```

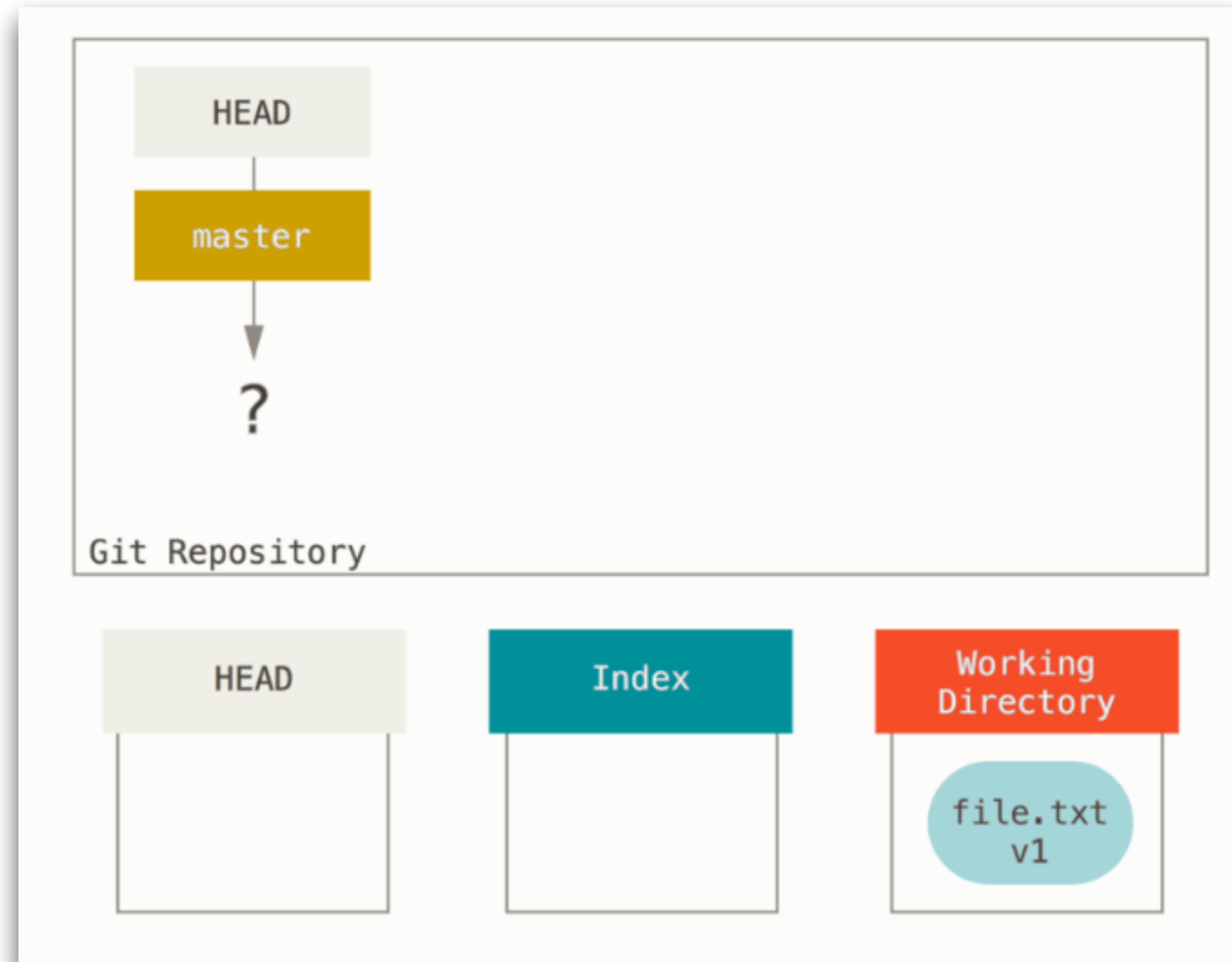
git - reset

- Moves the HEAD pointer



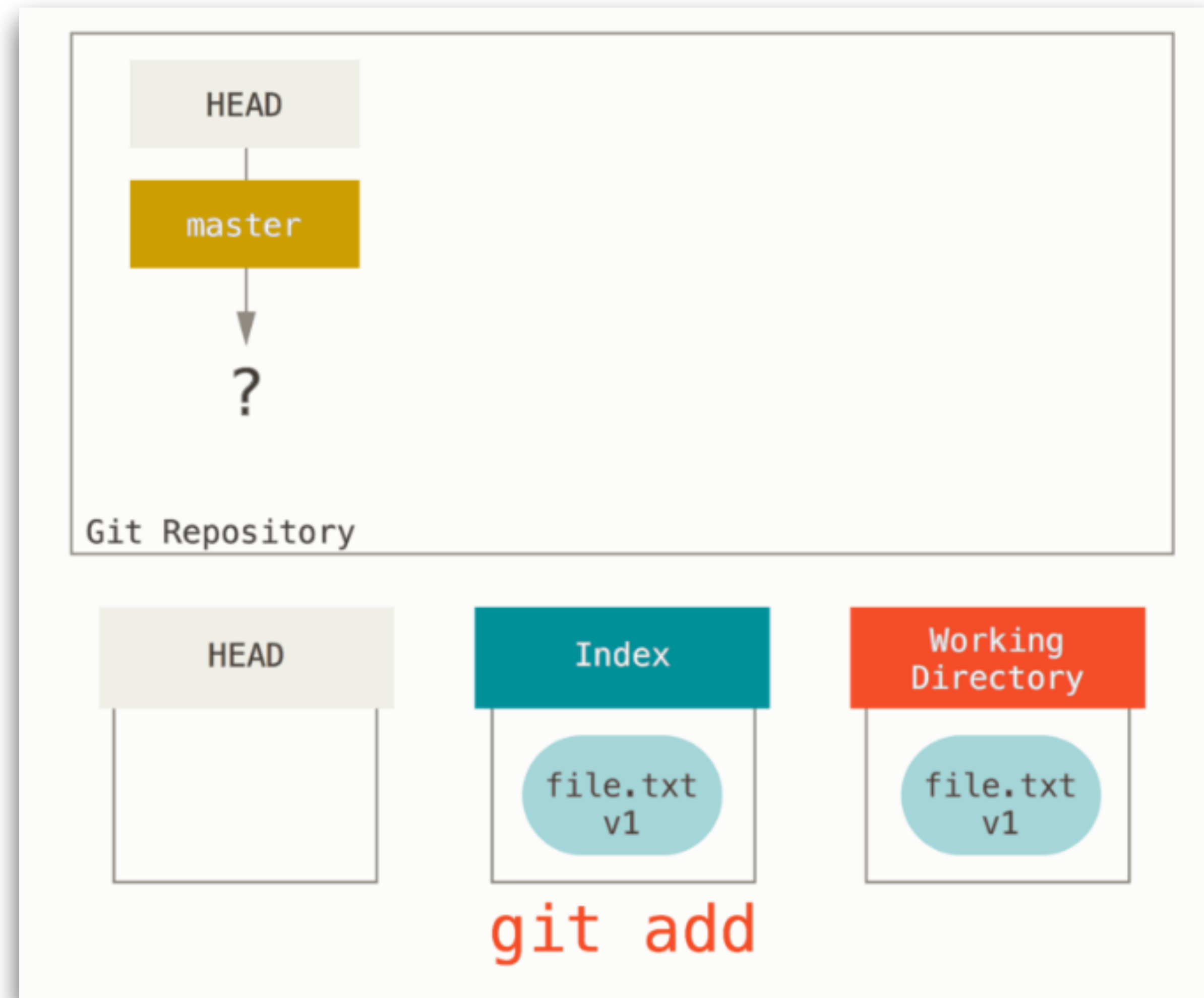
git - reset

(borrowed from <https://git-scm.com/book/ko/v2/Git-도구-Reset-명확히-알고-가기>)



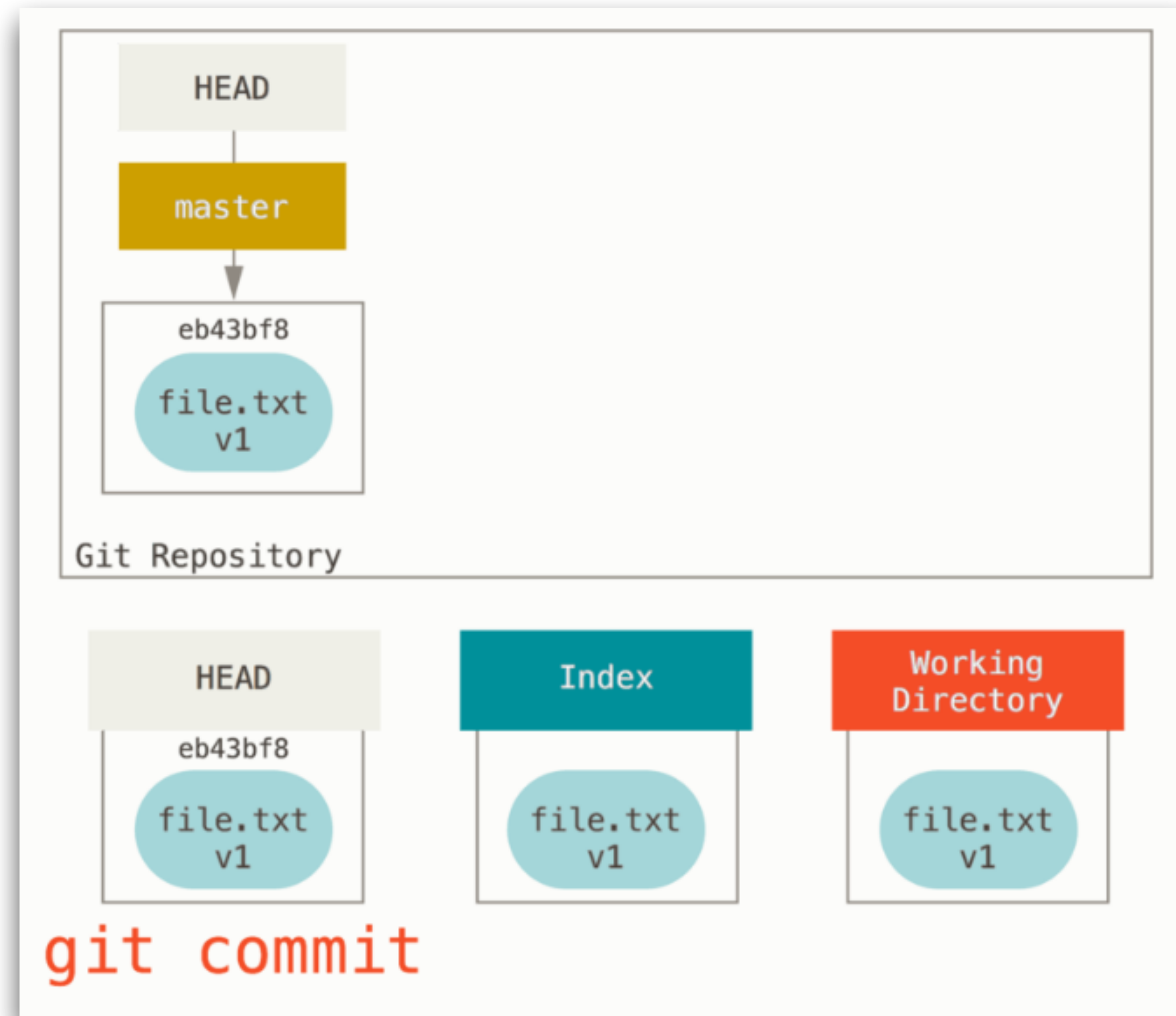
git - reset

(borrowed from <https://git-scm.com/book/ko/v2/Git-도구-Reset-명확히-알고-가기>)



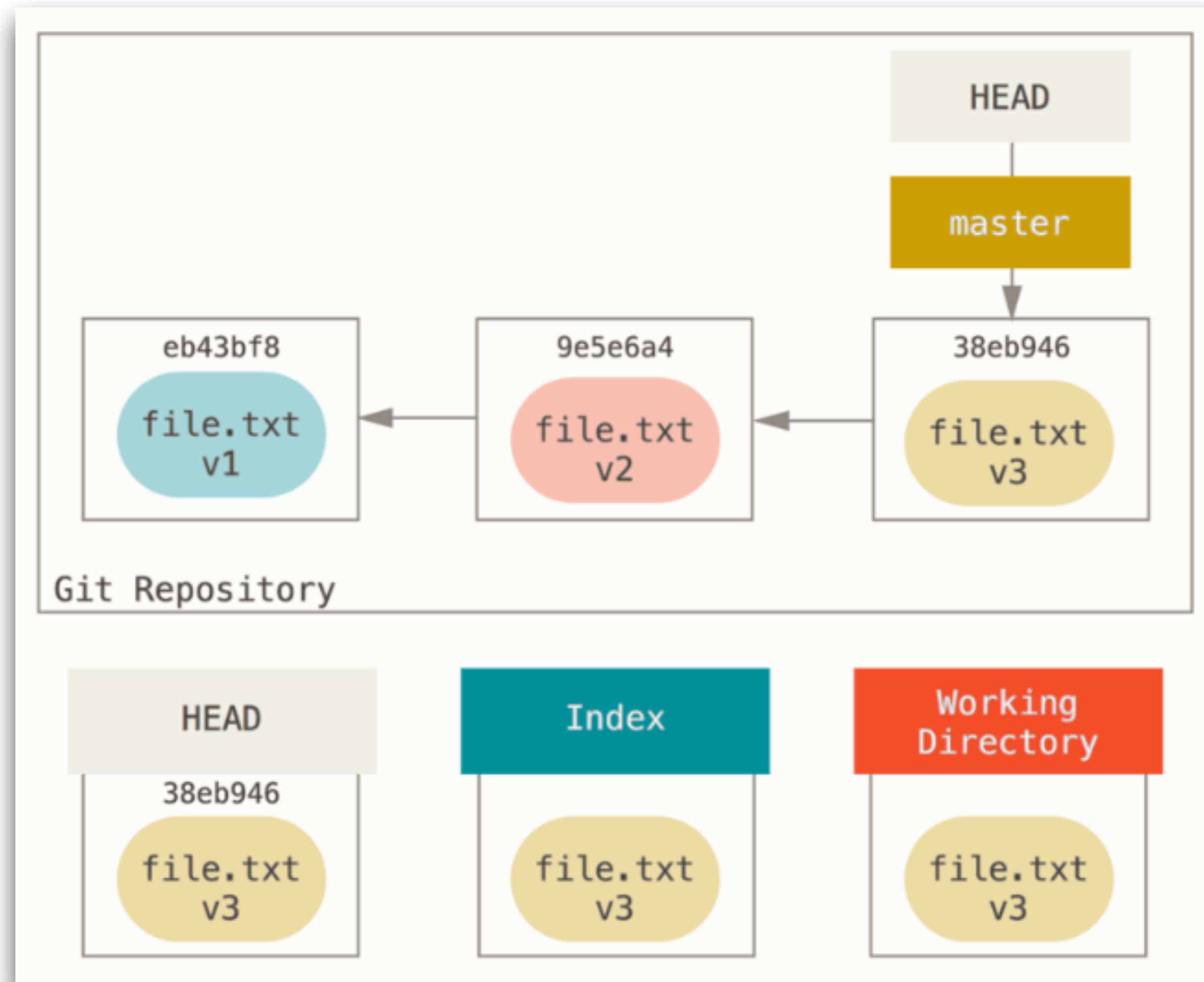
git - reset

(borrowed from <https://git-scm.com/book/ko/v2/Git-도구-Reset-명확히-알고-가기>)



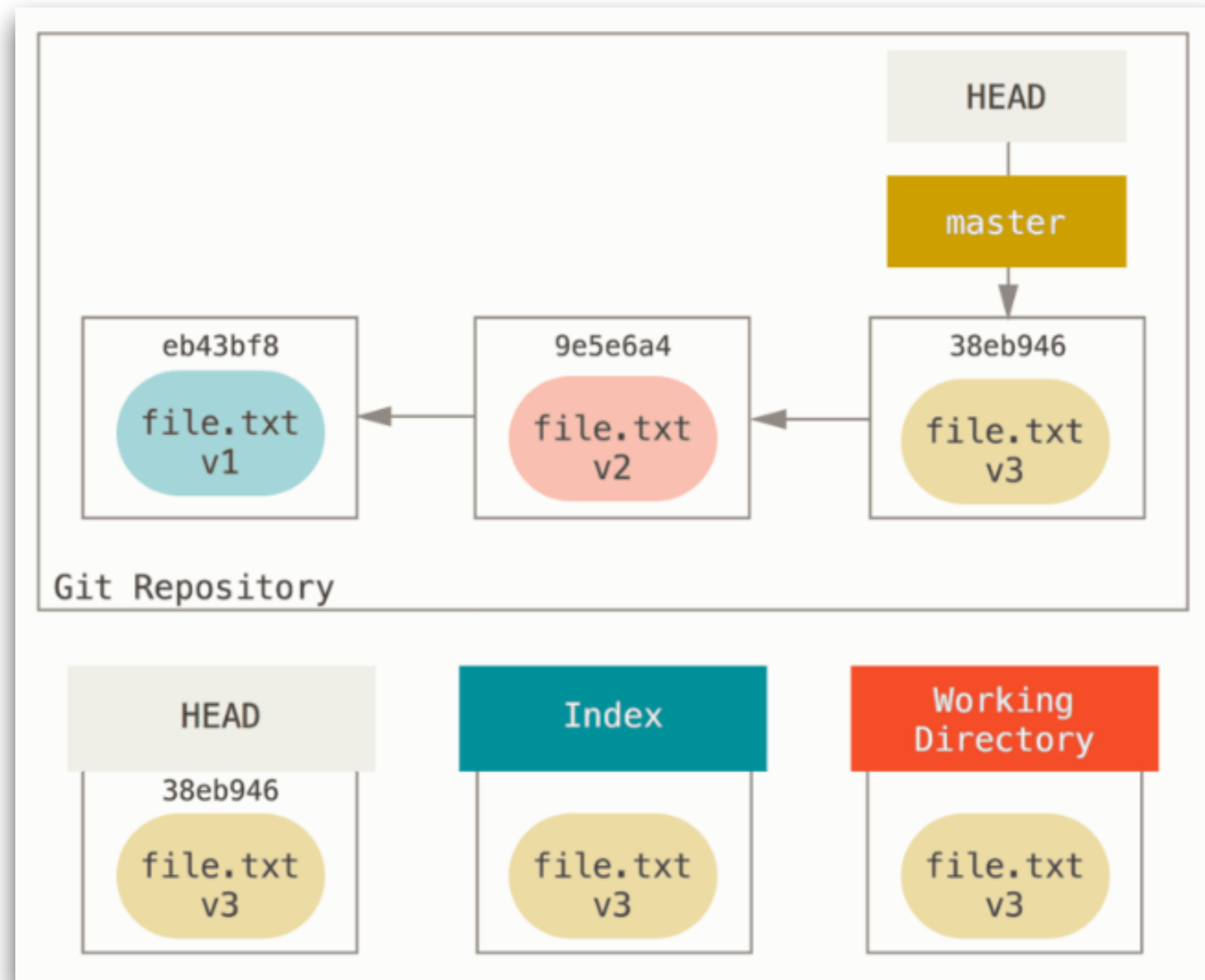
git - reset

(borrowed from <https://git-scm.com/book/ko/v2/Git-도구-Reset-명확히-알고-가기>)



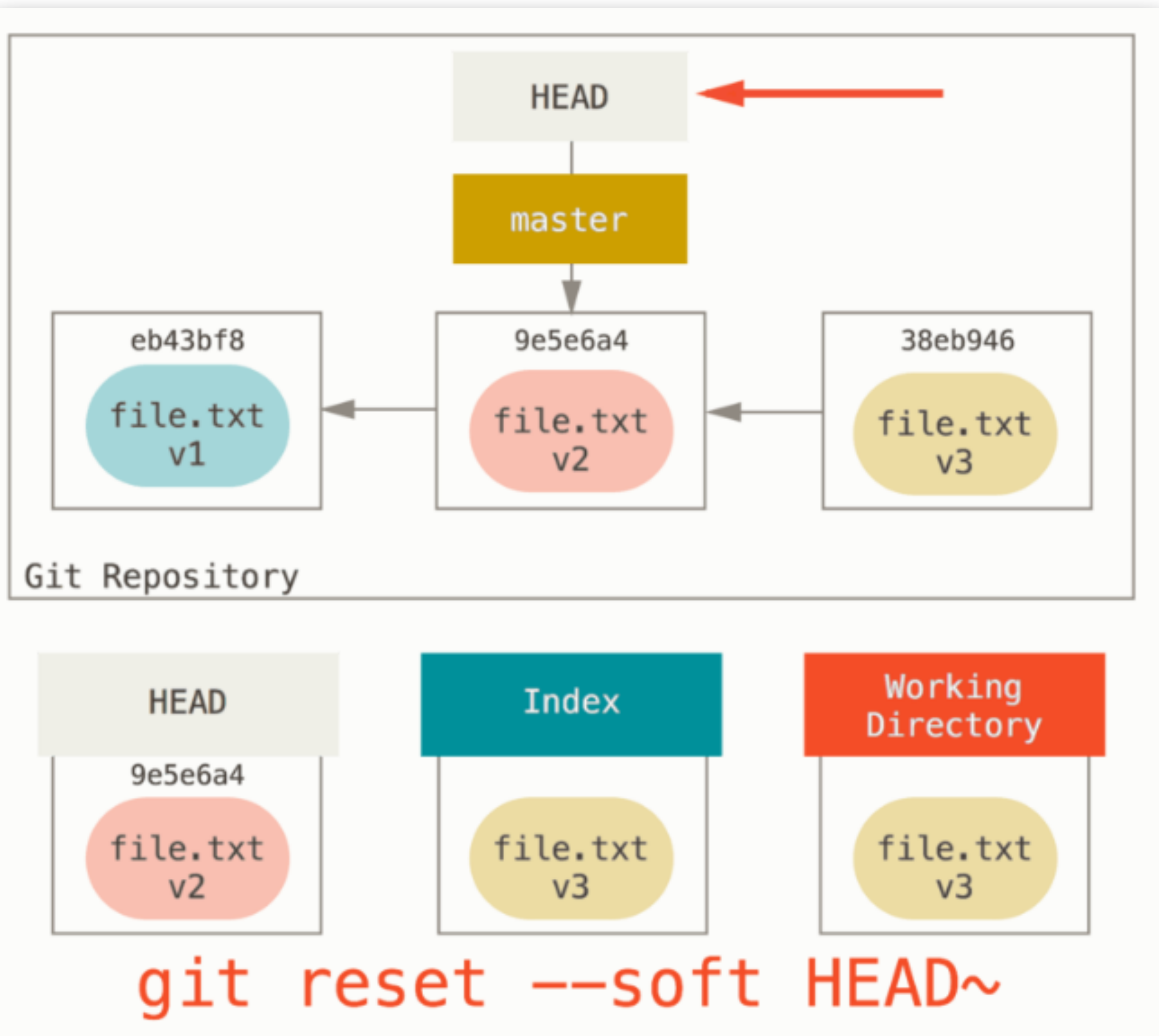
git - reset

(borrowed from <https://git-scm.com/book/ko/v2/Git-도구-Reset-명확히-알고-가기>)



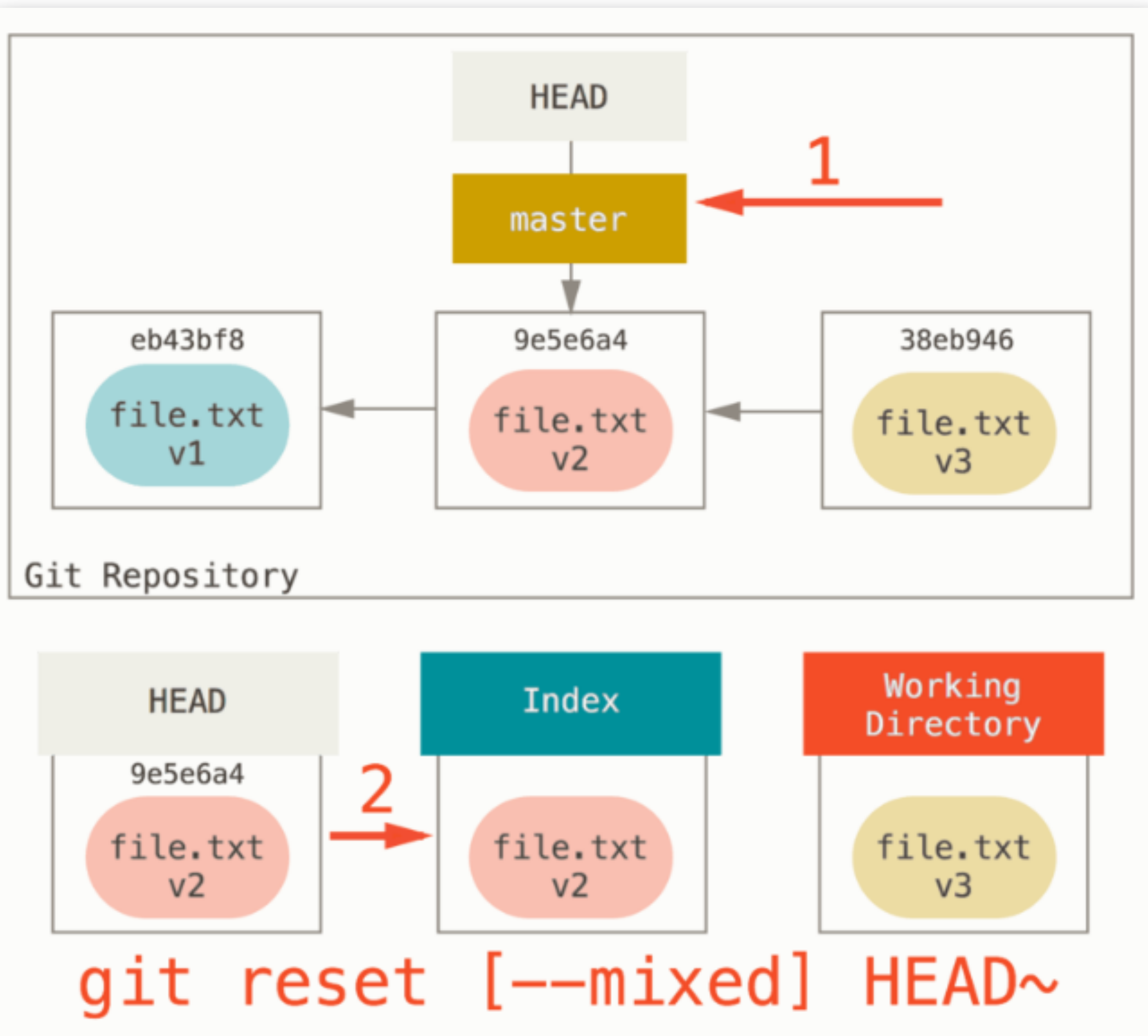
git - reset

(borrowed from <https://git-scm.com/book/ko/v2/Git-도구-Reset-명확히-알고-가기>)



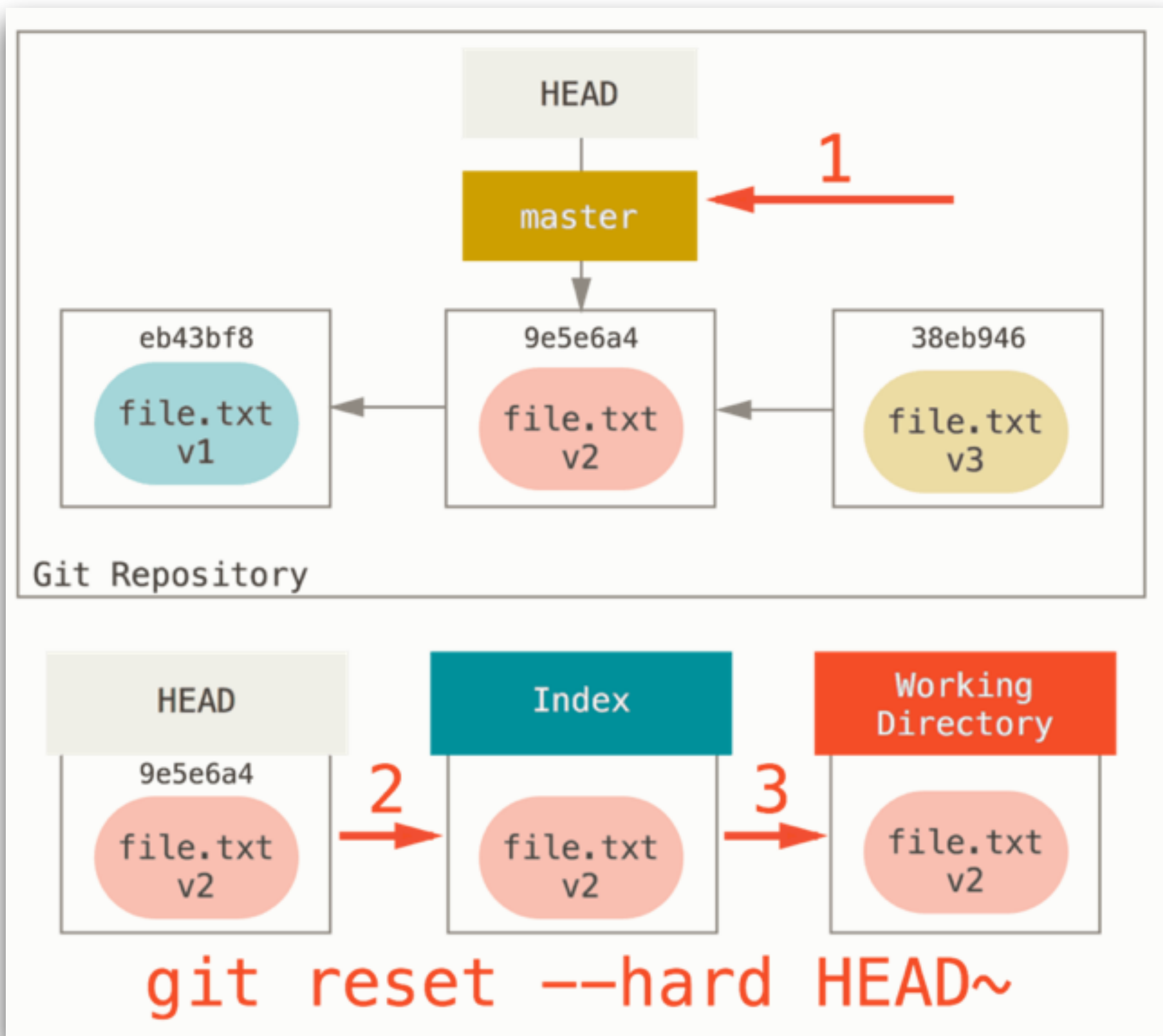
git - reset

(borrowed from <https://git-scm.com/book/ko/v2/Git-도구-Reset-명확히-알고-가기>)



git - reset

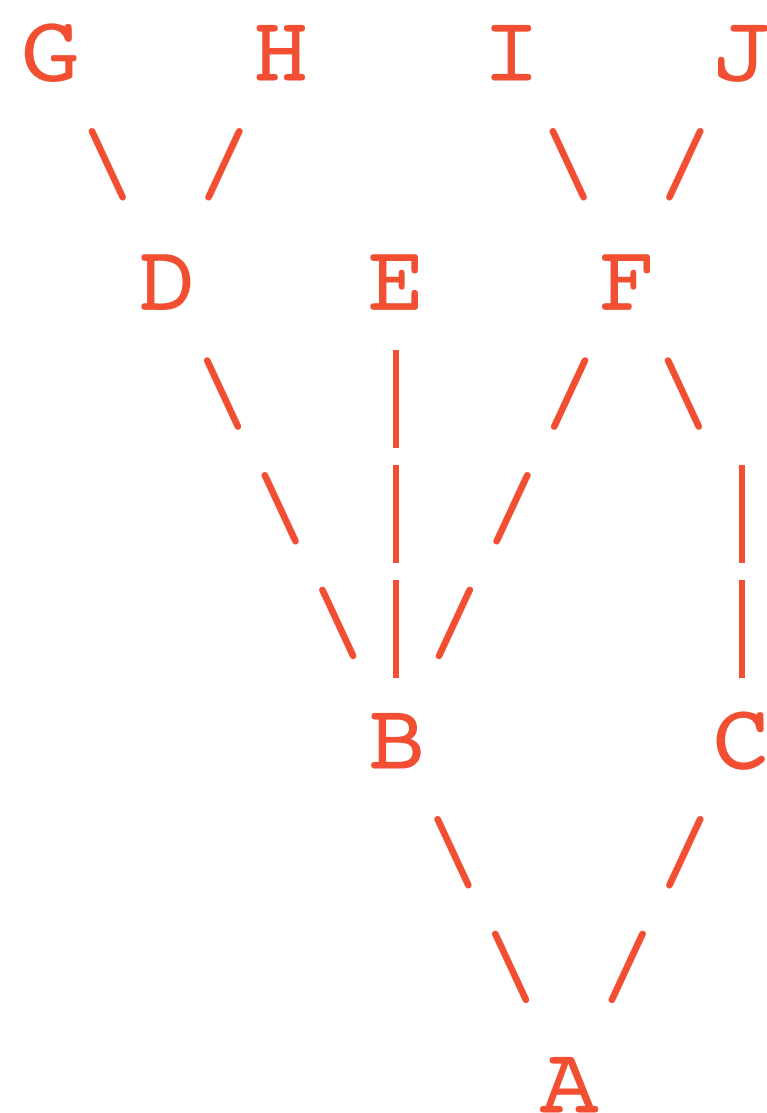
(borrowed from <https://git-scm.com/book/ko/v2/Git-도구-Reset-명확히-알고-가기>)



HEAD notation

(example from https://git-scm.com/docs/git-rev-parse#specifying_revisions)

- HEAD~n: *n*-th ancestor from HEAD (vertical)
- HEAD^n: *n*-th immediate parent of HEAD when HEAD has multiple parents (horizontal)



A =	=	A^0		
B = A^	=	A^1	=	A~1
C =	=	A^2		
D = A^^	=	A^1^1	=	A~2
E = B^2	=	A^^2		
F = B^3	=	A^^3		
G = A^^^	=	A^1^1^1	=	A~3
H = D^2	=	B^^2	=	A^^^2 = A~2^2
I = F^	=	B^3^	=	A^^3^
J = F^2	=	B^3^2	=	A^^3^2

git - blame

- Identifies commits that touched specific parts of a file.
- `git blame [filename]`: shows all commits that modified this file.
- `git blame [filename] -L <start>, <end>`: shows all commits that modified lines between <start> and <end>

git - bisect

- Allows you to pinpoint the moment (=the commit) that something bad happened for the first time.
 - First, initiate the bisection process: `git bisect start`
 - Second, mark the bad commit e.g., the current commit: `git bisect bad`
 - Third, mark the last good commit that you know of: `git bisect good <id>`
- Now git will essentially perform binary search between good and bad, and check-out a commit that you need to mark either good or bad
 - Good/bad can be anything: code inspection, test results, etc...

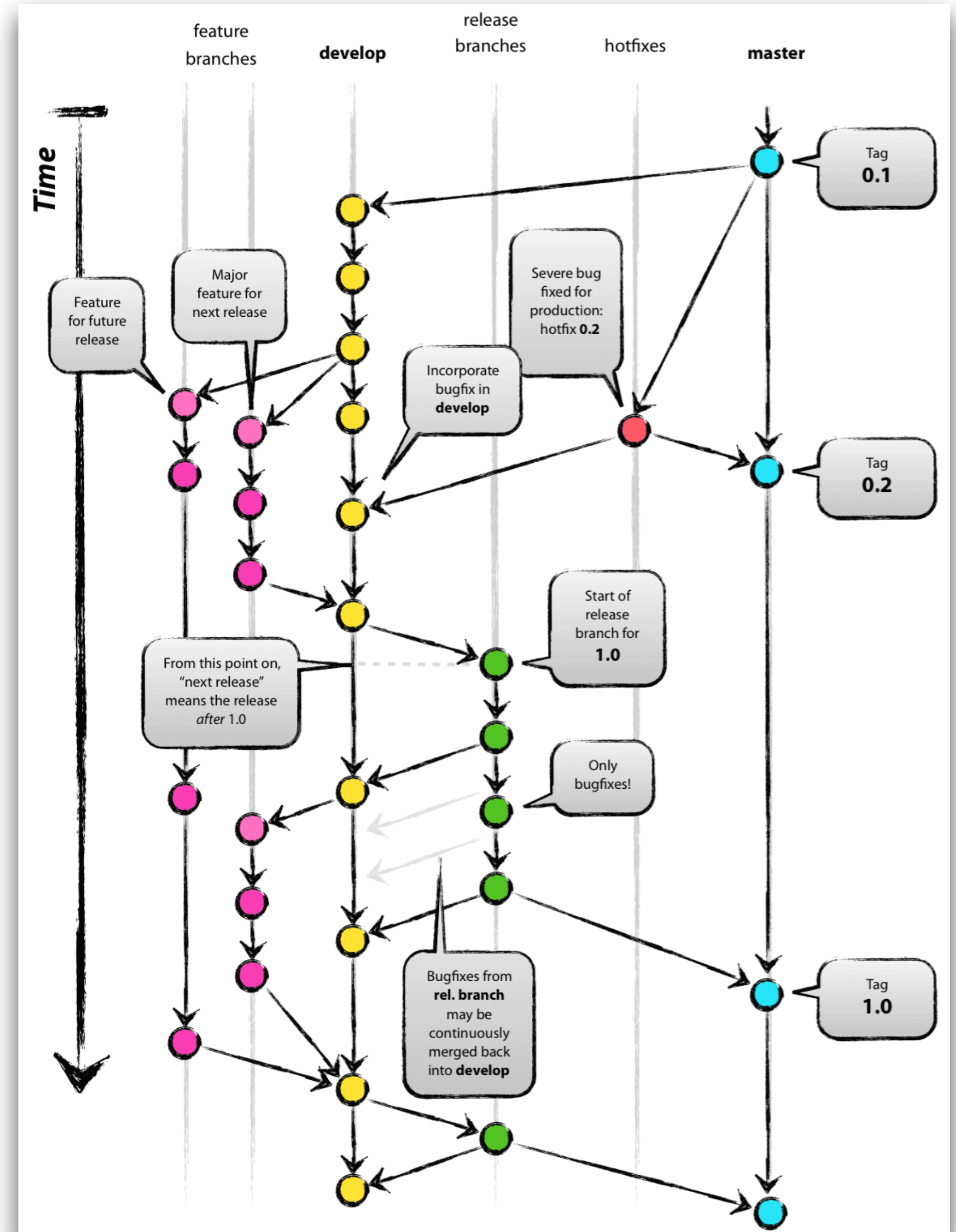
Different Workflows Using Git

- Git allows us to have branches, as well as to push/pull changes.
- You can implement different workflows using git as the VCS tool.
- Major workflows
 - GitFlow
 - GitHub Flow
 - Trunk-based Development

GitFlow

Vincent Driessen, 2010

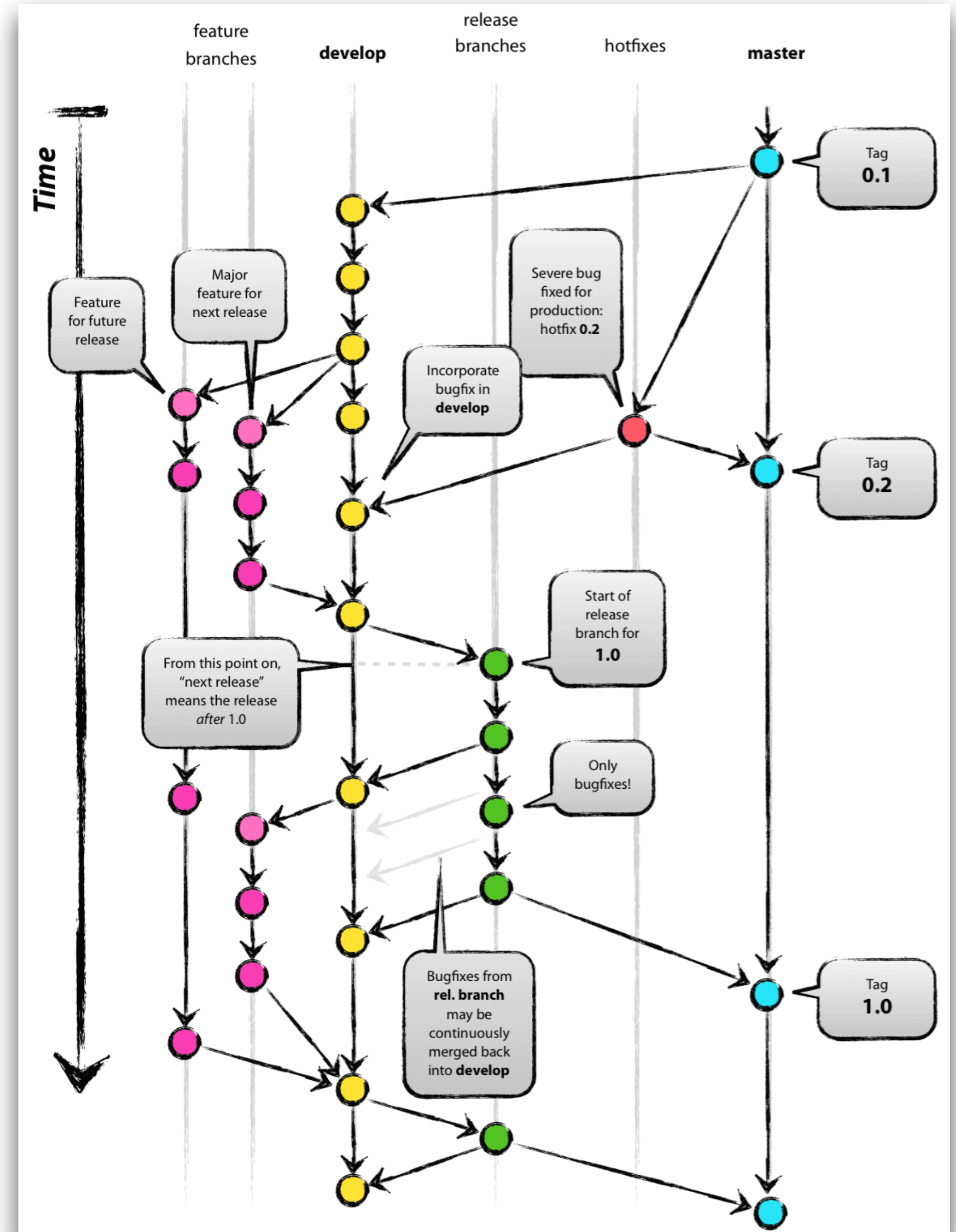
- Trunk (main, master) tags release versions.
- Main development branch is where code changes are pushed.
- Each feature is developed in its own branch, and merged with the development branch.
- Release/hotfix branches for only those type of changes.



GitFlow

Vincent Driessen, 2010

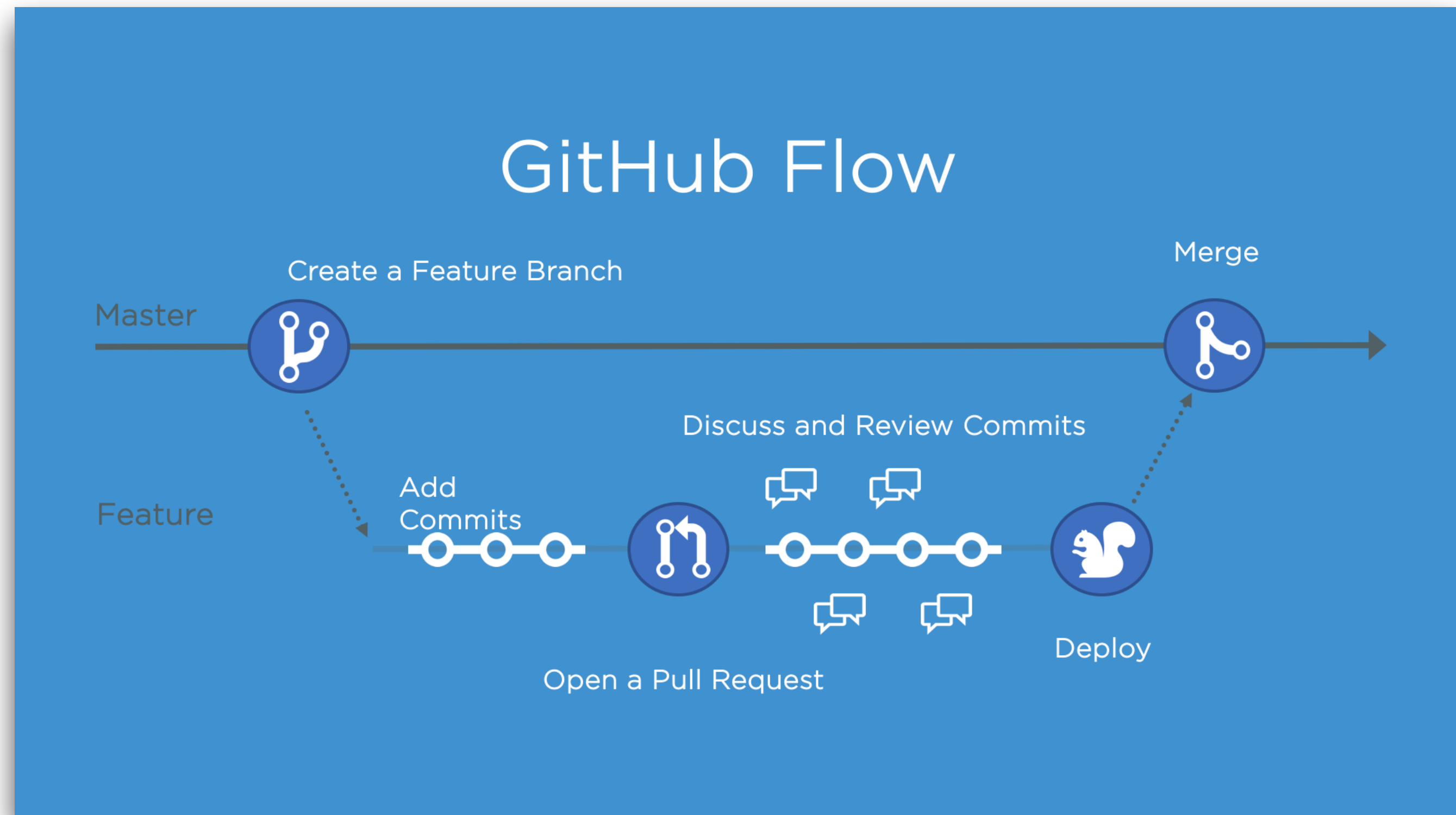
- Uses git branches to their maximum potential
- However, no single branch available for continuous integration/testing
- Can run into incredibly messy merge conflicts if not careful



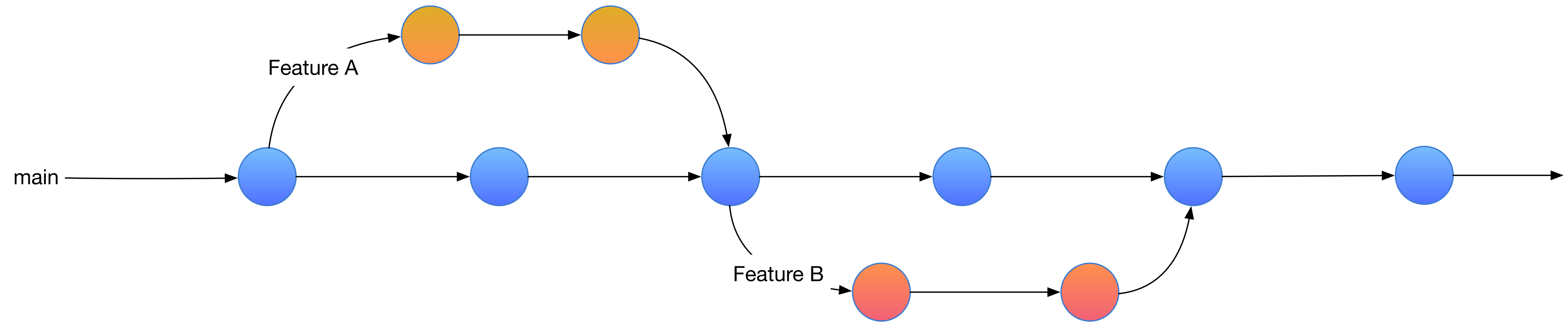
GitHub Flow

Simplified version of GitFlow

- Proposed by GitHub
- Each new feature is developed in its own feature branch.
 - Once ready, a Pull Request is made to the main branch.
 - PR is reviewed, the merged.
- Release is done from the main branch.

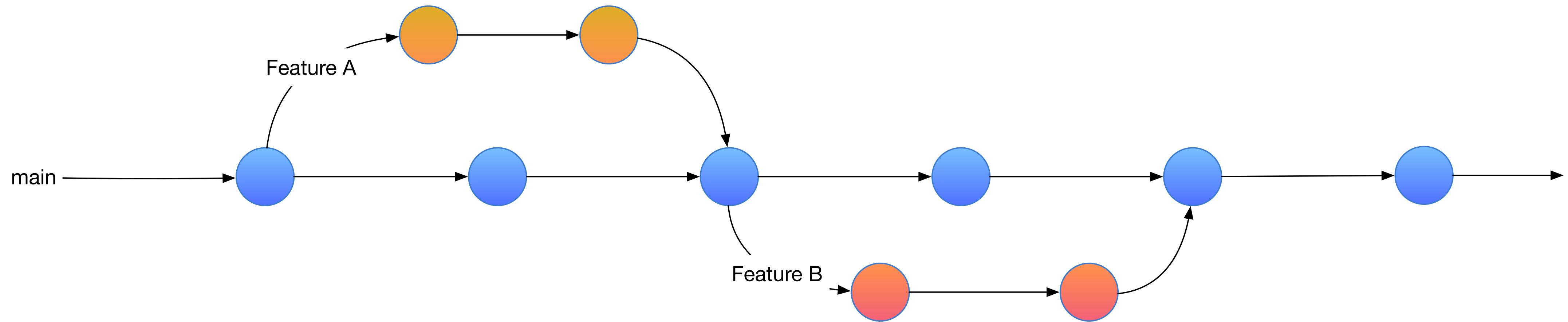


Trunk Based Development



- Developers work on features on separate branches, but...
- Everyone has to merge to the main branch at least once a day.
- Main branch should always be releasable!

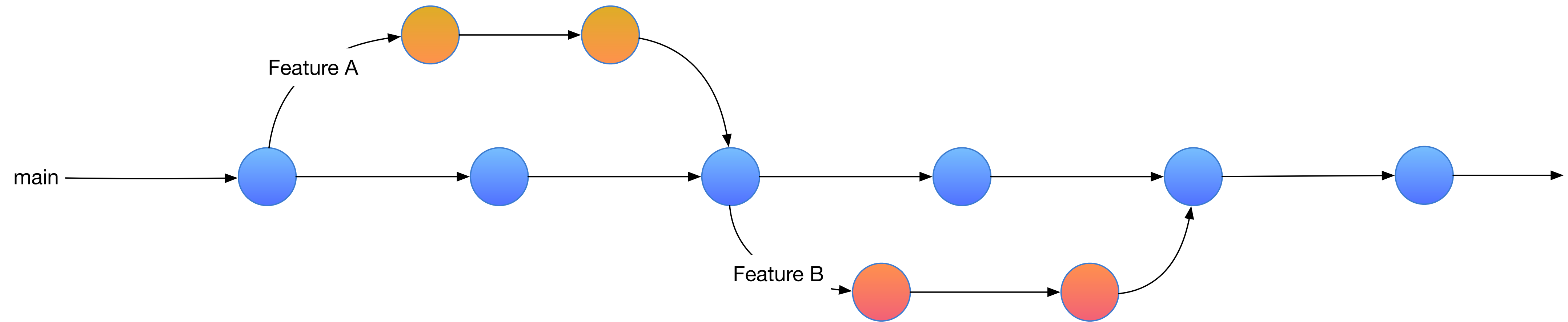
Feature Flag



- How do you develop a feature that may take more than a day?
- Add a flag that turns off the feature, and implement under the disabled flag.
- The main trunk still continues to pass, as the new feature is disabled.
- On the day that you are ready, turn on the feature and merge :)

Trunk Based Development

Why do this?



- Merges are much easier, as we only merge small changes.
- Main branch is more readily available for CI/CD.
- May require more experienced developers.

Summary

- VCS, especially DVCS, is a foundation of modern day SE practices.
- You really, really need to be familiar with basics of git.
 - It is like a big Swiss army knife: so many different features, so many different ways of doing the same thing.
 - Still, basic branching/merging operations should be your second nature.
- Assignment 3 will require you to be reasonably comfortable with the basics :)