



Git for Version Control

These slides are heavily based on slides created
by Ruth Anderson for CSE 390a. Thanks, Ruth!

<http://www.cs.washington.edu/403/>

About Git

- Created by Linus Torvalds, creator of Linux, in 2005
 - Came out of Linux development community
 - Designed to do version control on Linux kernel
- Goals of Git:
 - Speed
 - Support for non-linear development (thousands of parallel branches)
 - Fully distributed
 - Able to handle large projects efficiently
 - (A "git" is a cranky old man. Linus meant himself.)

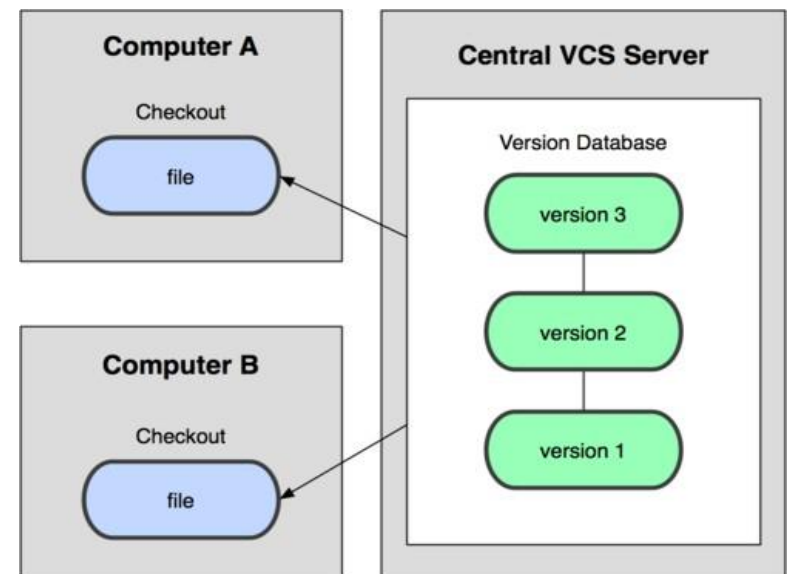


Installing/learning Git

- Git website: <http://git-scm.com/>
 - Free on-line book: <http://git-scm.com/book>
 - Reference page for Git: <http://gitref.org/index.html>
 - Git tutorial: <http://schacon.github.com/git/gittutorial.html>
 - Git for Computer Scientists:
 - <http://eagain.net/articles/git-for-computer-scientists/>
- At command line: (where verb = config, add, commit, etc.)
 - `git help verb`

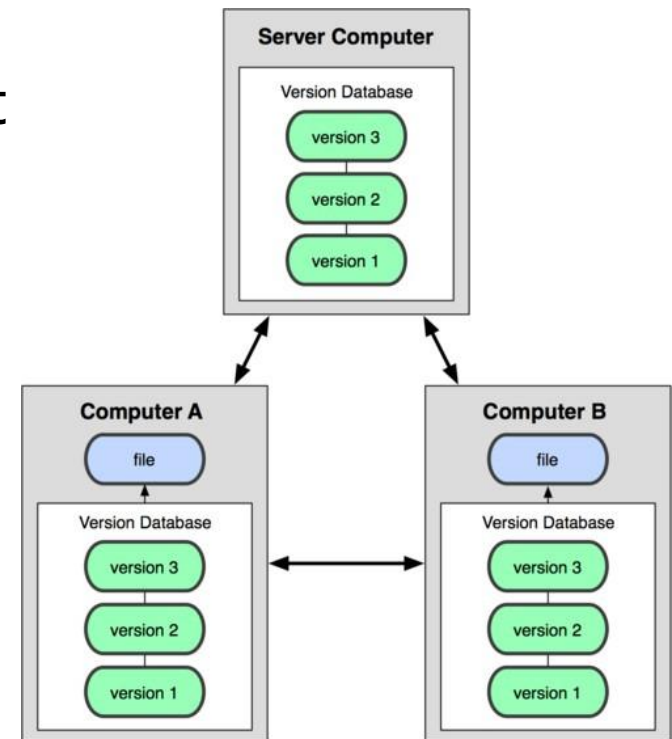
Centralized VCS

- In Subversion, CVS, Perforce, etc.
A central server repository (repo) holds the "official copy" of the code
 - the server maintains the sole version history of the repo
- You make "checkouts" of it to your local copy
 - you make local modifications
 - your changes are not versioned
- When you're done, you "check in" back to the server
 - your checkin increments the repo's version



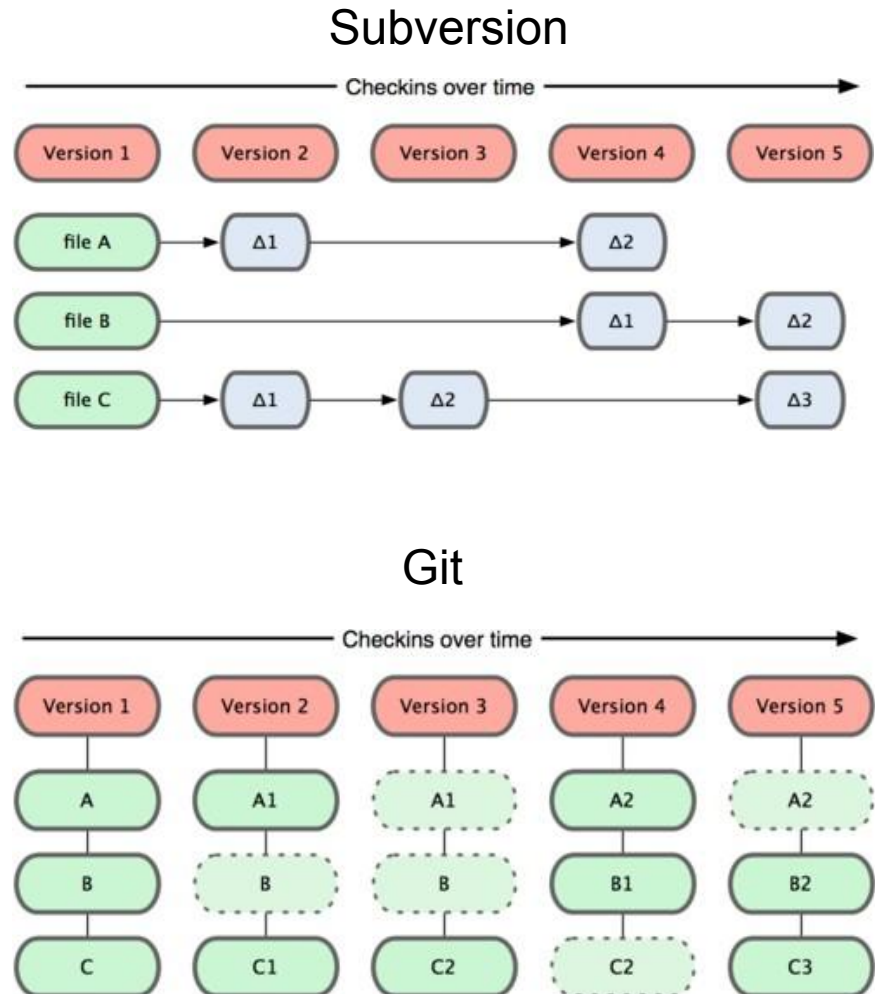
Distributed VCS (Git)

- In git, mercurial, etc., you don't "checkout" from a central repo
 - you "clone" it and "pull" changes from it
- Your local repo is a complete copy of everything on the remote server
 - yours is "just as good" as theirs
- Many operations are local:
 - check in/out from local repo
 - commit changes to local repo
 - local repo keeps version history
- When you're ready, you can "push" changes back to server



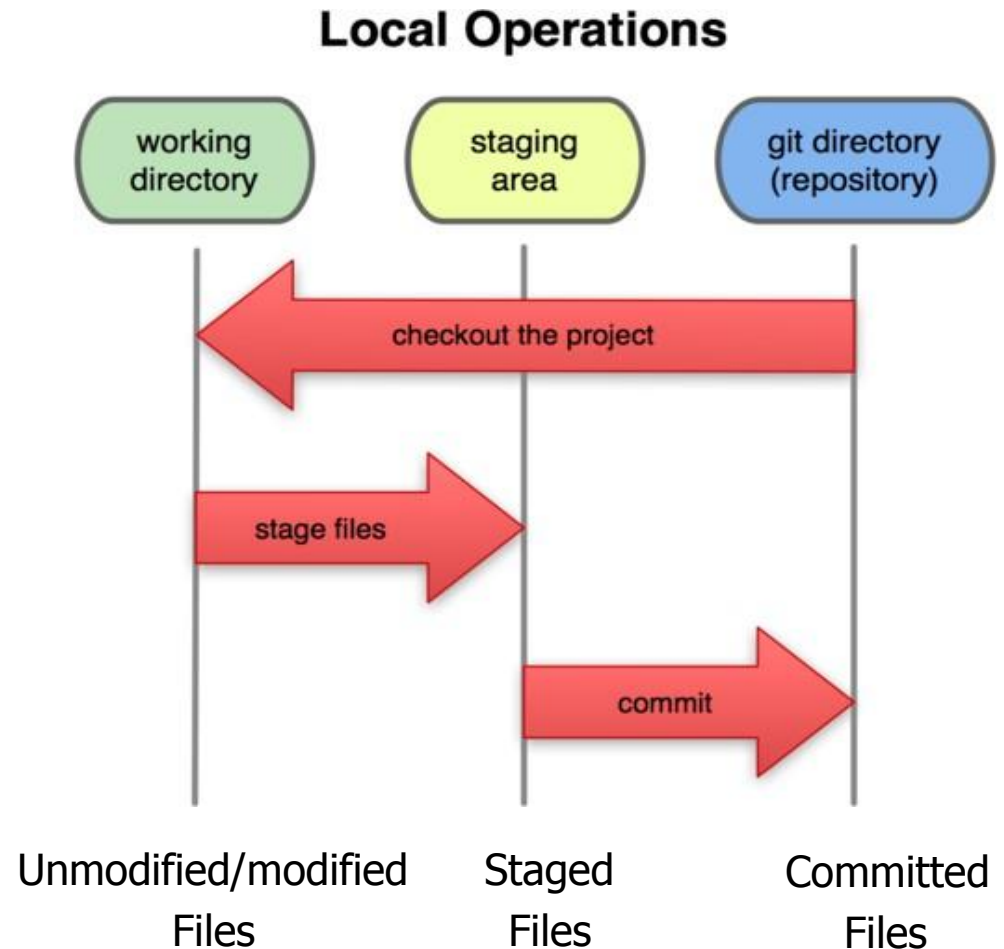
Git snapshots

- Centralized VCS like Subversion track version data on each individual file.
- Git keeps "snapshots" of the entire state of the project.
 - Each checkin version of the overall code has a copy of each file in it.
 - Some files change on a given checkin, some do not.
 - More redundancy, but faster.



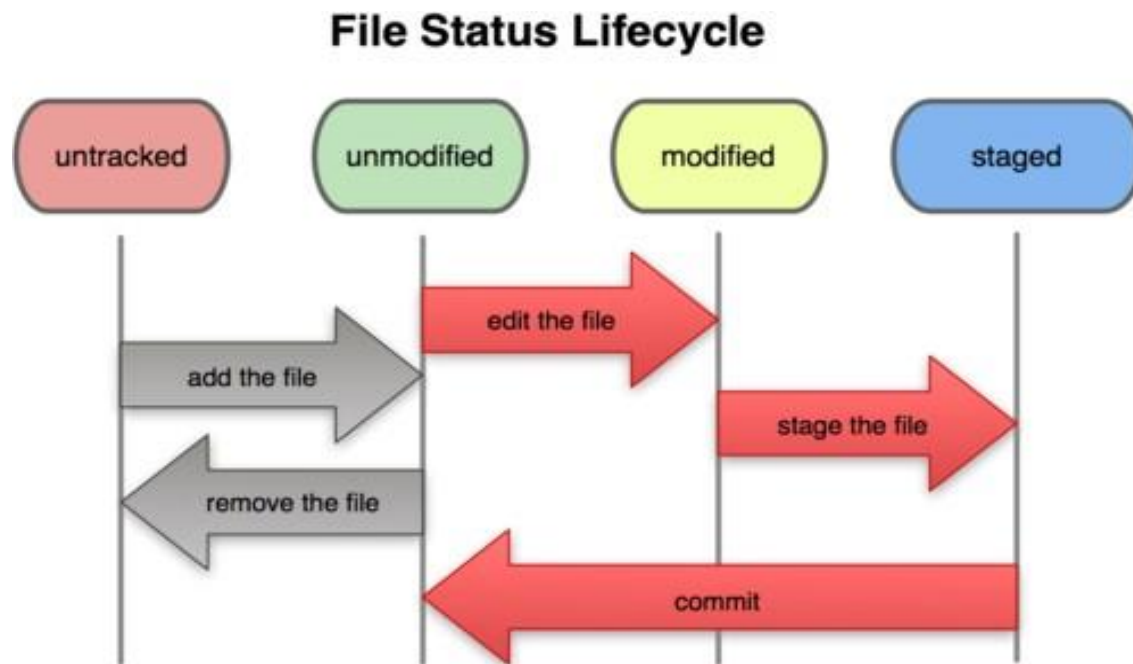
Local git areas

- In your local copy on git, files can be:
 - In your local repo
 - (committed)
 - Checked out and modified, but not yet committed
 - (working copy)
 - Or, in-between, in a **"staging" area**
 - Staged files are ready to be committed.
 - A commit saves a snapshot of all staged state.



Basic Git workflow

- **Modify** files in your working directory.
- **Stage** files, adding snapshots of them to your staging area.
- **Commit**, which takes the files in the staging area and stores that snapshot permanently to your Git directory.



Git commit checksums

- In Subversion each modification to the central repo increments the version # of the overall repo.
 - In Git, each user has their own copy of the repo, and commits changes to their local copy of the repo before pushing to the central server.
 - So Git generates a unique **SHA-1 hash** (40 character string of hex digits) for every commit.
 - Refers to commits by this ID rather than a version number.
 - Often we only see the first 7 characters:
 - 1677b2d Edited first line of readme
 - 258efa7 Added line to readme
 - 0e52da7 Initial commit

Initial Git configuration

- Set the name and email for Git to use when you commit:
 - `git config --global user.name "Bugs Bunny"`
 - `git config --global user.email bugs@gmail.com`
 - You can call `git config -list` to verify these are set.
- Set the editor that is used for writing commit messages:
 - `git config --global core.editor nano`
 - (it is vim by default)

Creating a Git

Two common scenarios: (only do one of these)

- To create a new **local Git repo** in your current directory:

- `git init`

- This will create a `.git` directory in your current directory.

- Then you can commit files in that directory into the repo.

- `git add filename`

- `git commit -m "commit message"`

- To **clone a remote repo** to your current directory:

- `git clone url localDirectoryName`

- This will create the given local directory, containing a working copy of the files from the repo, and a `.git` directory (used to hold the staging area and your actual local repo)

Git commands

command	description
<code>git clone <i>url</i> [<i>dir</i>]</code>	copy a Git repository so you can add to it
<code>git add <i>file</i></code>	adds file contents to the staging area
<code>git commit</code>	records a snapshot of the staging area
<code>git status</code>	view the status of your files in the working directory and staging area
<code>git diff</code>	shows diff of what is staged and what is modified but unstaged
<code>git help [<i>command</i>]</code>	get help info about a particular command
<code>git pull</code>	fetch from a remote repo and try to merge into the current branch
<code>git push</code>	push your new branches and data to a remote repository
others: <code>init</code> , <code>reset</code> , <code>branch</code> , <code>checkout</code> , <code>merge</code> , <code>log</code> , <code>tag</code>	

Add and commit a file

- The first time we ask a file to be tracked, and every time before we commit a file, we must add it to the staging area:
 - `git add Hello.java Goodbye.java`
 - Takes a snapshot of these files, adds them to the staging area.
 - In older VCS, "add" means "start tracking this file." In Git, "add" means "add to staging area" so it will be part of the next commit.
- To move staged changes into the repo, we commit:
 - `git commit -m "Fixing bug #22"`
- To undo changes on a file before you have committed it:
 - `git reset HEAD -- filename` (unstages the file)
 - `git checkout -- filename` (undoes your changes)
 - All these commands are acting on your local version of repo.

Viewing/undoing changes

- To view status of files in working directory and staging area:
 - `git status` or `git status -s` (short version)

Note: Short status flags are:

- ?? - Untracked files
 - A - Files added to stage
 - M - Modified files
 - D - Deleted files
-
- To see what is modified but unstaged:
 - `git diff`
 - To see a list of staged changes:
 - `git diff --cached`
 - To see a log of all changes in your local repo:
 - `git log` or `git log --oneline` (shorter version)
 - 1677b2d Edited first line of readme 258efa7
 - Added line to readme 0e52da7 Initial commit
 - `git log -5` (to show only the 5 most recent updates) etc

An example

```
[rea@attul superstar]$ vim rea.txt
[rea@attul superstar]$ git status
  no changes added to commit
  (use "git add" and/or "git commit -a")
[rea@attul superstar]$ git status -s
  M rea.txt
[rea@attul superstar]$ git diff
  diff --git a/rea.txt b/rea.txt
[rea@attul superstar]$ git add rea.txt
[rea@attul superstar]$ git status
  #modified: rea.txt
[rea@attul superstar]$ git diff --cached
  diff --git a/rea.txt b/rea.txt
[rea@attul superstar]$ git commit -m "Created new text file"
```

Branching and merging

Git uses branching heavily to switch between multiple tasks.

- To create a new local branch:
 - `git branch name`
- To list all local branches: (* = current branch)
 - `git branch`
- To switch to a given local branch:
 - `git checkout branch_name`
- To merge changes from a branch into the local master:
 - `git checkout master`
 - `git merge branch_name`

Merge conflicts

- The conflicting file will contain <<< and >>> sections to indicate where Git was unable to resolve a conflict:

```
<<<<<< HEAD:index.html
<div id="footer">todo: message here</div>
=====
<div id="footer">
  thanks for visiting our site
</div>
>>>>>> SpecialBranch:index.html
```

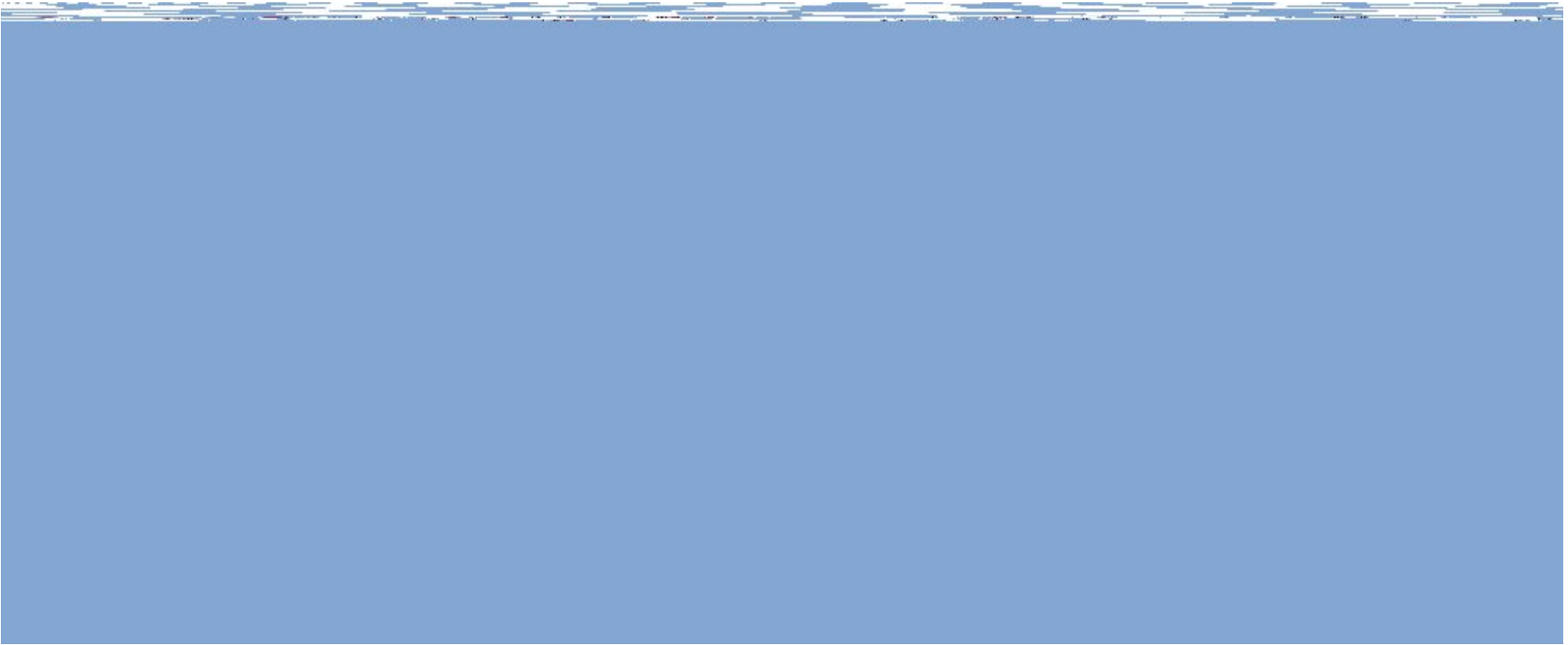
} branch 1's version

} branch 2's version

- Find all such sections, and edit them to the proper state (whichever of the two versions is newer / better / more correct).

Interactionw/ remote repo

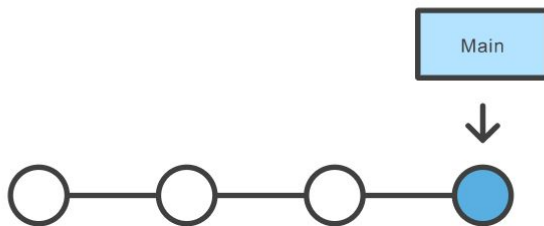
- **Push** your local changes to the remote repo.
- **Pull** from remote repo to get most recent changes.
 - (fix conflicts if necessary, add/commit them to your local repo)
- To fetch the most recent updates from the remote repo into your local repo, and put them into your working directory:
 - `git pull origin master`
- To put your changes from your local repo in the remote repo:
 - `git push origin master`



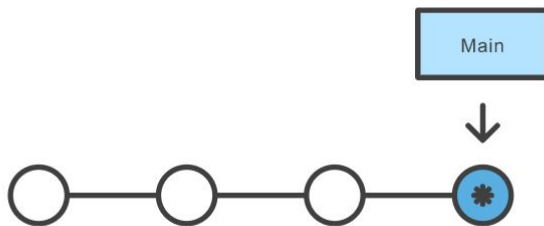
Changing the Last Commit

The `git commit --amend` command is a convenient way to modify the most recent commit.

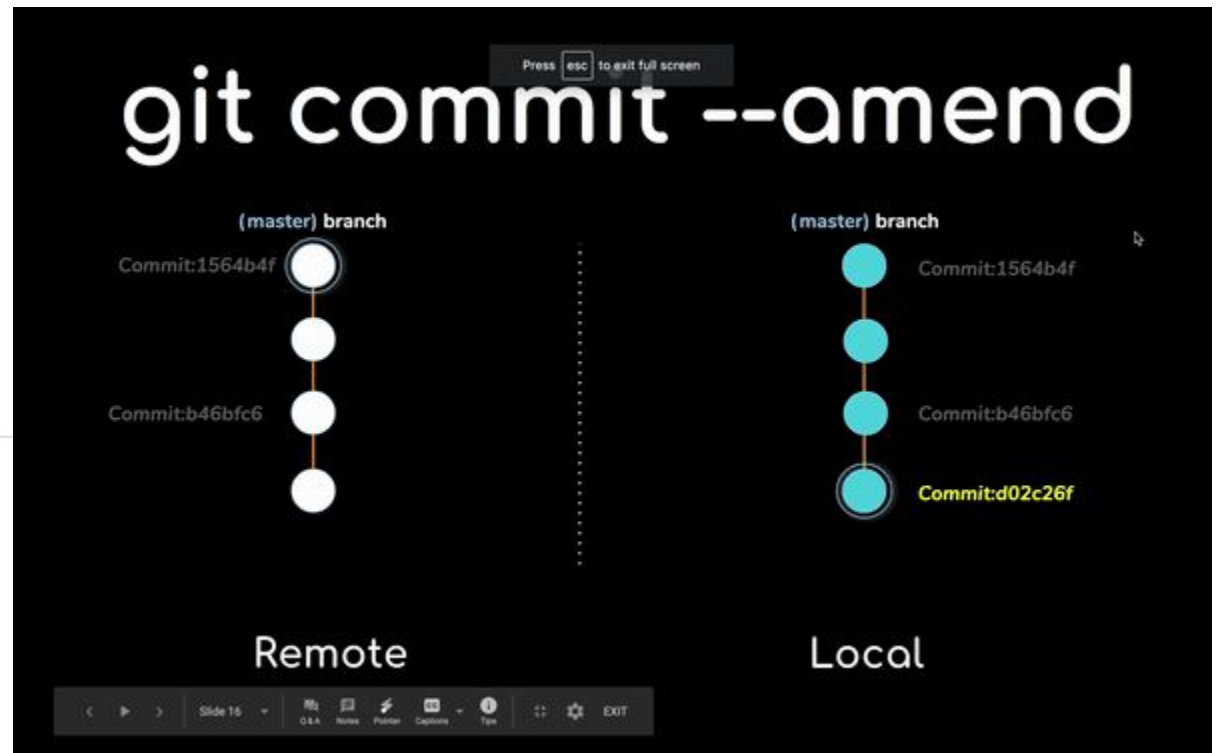
Initial History



Amended History



★ Brand New Commits



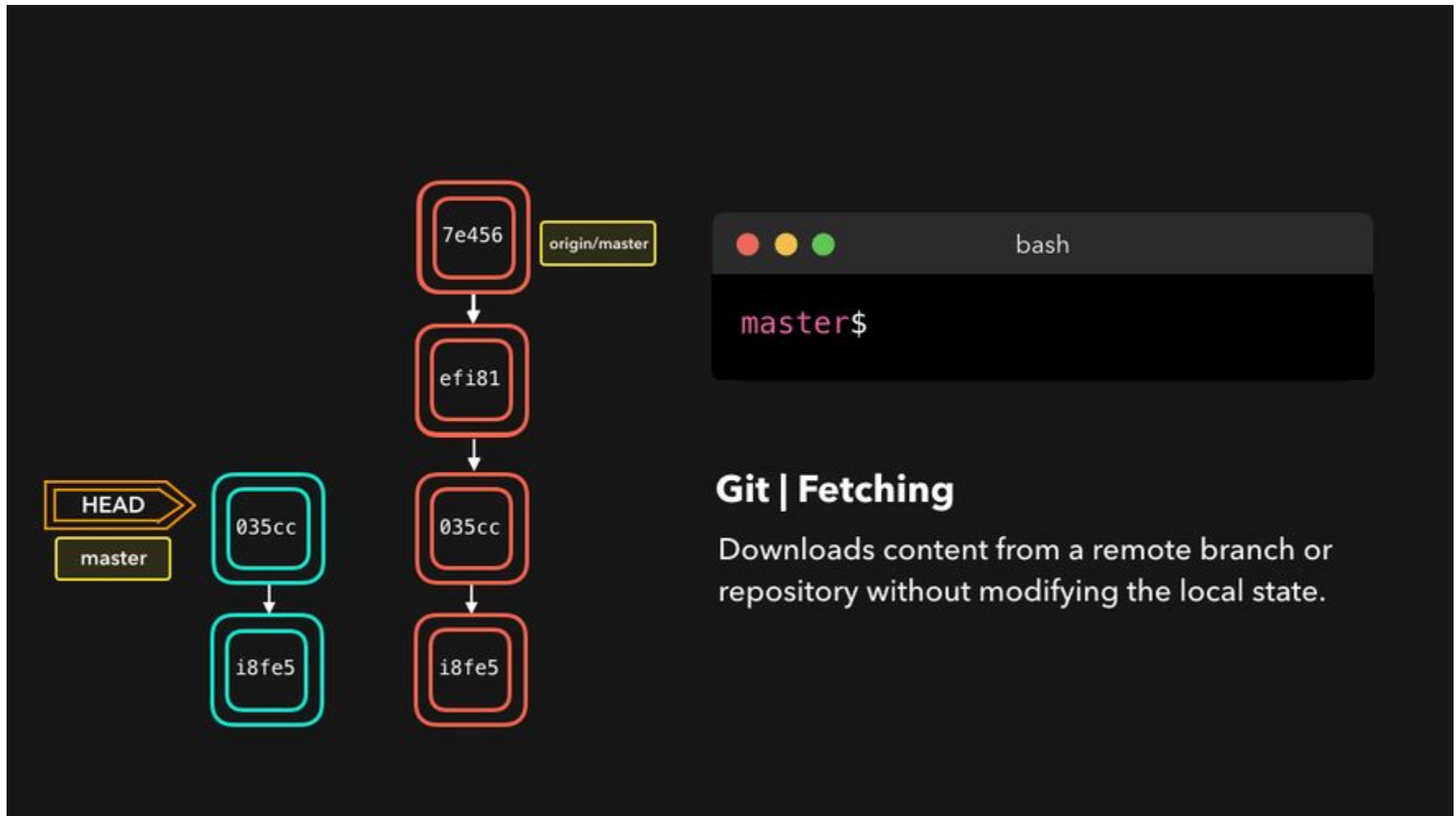
Undo with: `git commit --amend`

or

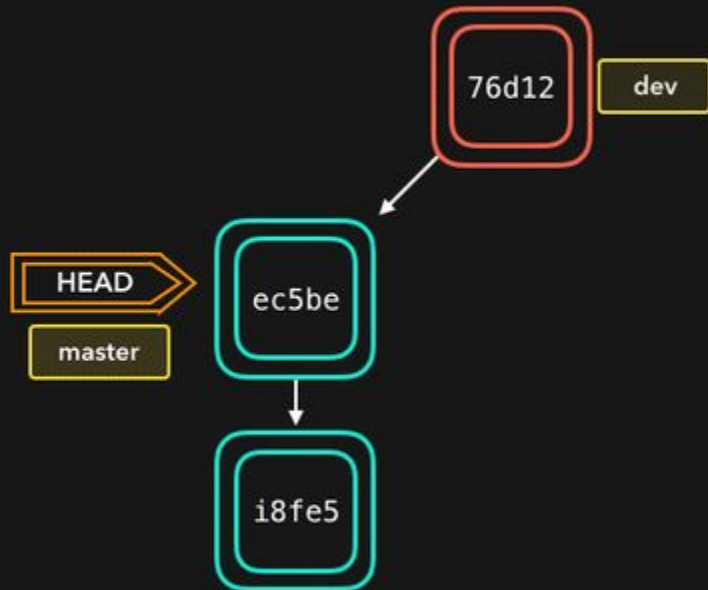
`git commit --amend -m "Fixes bug #42"`

Git Fetch

In the simplest terms, git pull does a git fetch followed by a git merge.



Merging (fast-forward)



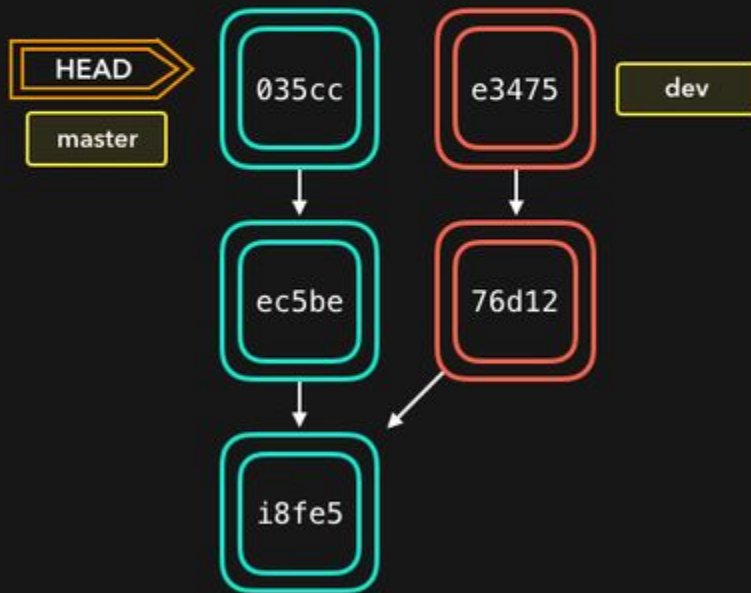
```
bash
master$
```

Git | Merging (fast-forward)

Default behavior when the merging branch has all of the current branch's commits

Doesn't create a new commit, thus doesn't modify existing branches

Merging (no-fast-forward)



bash

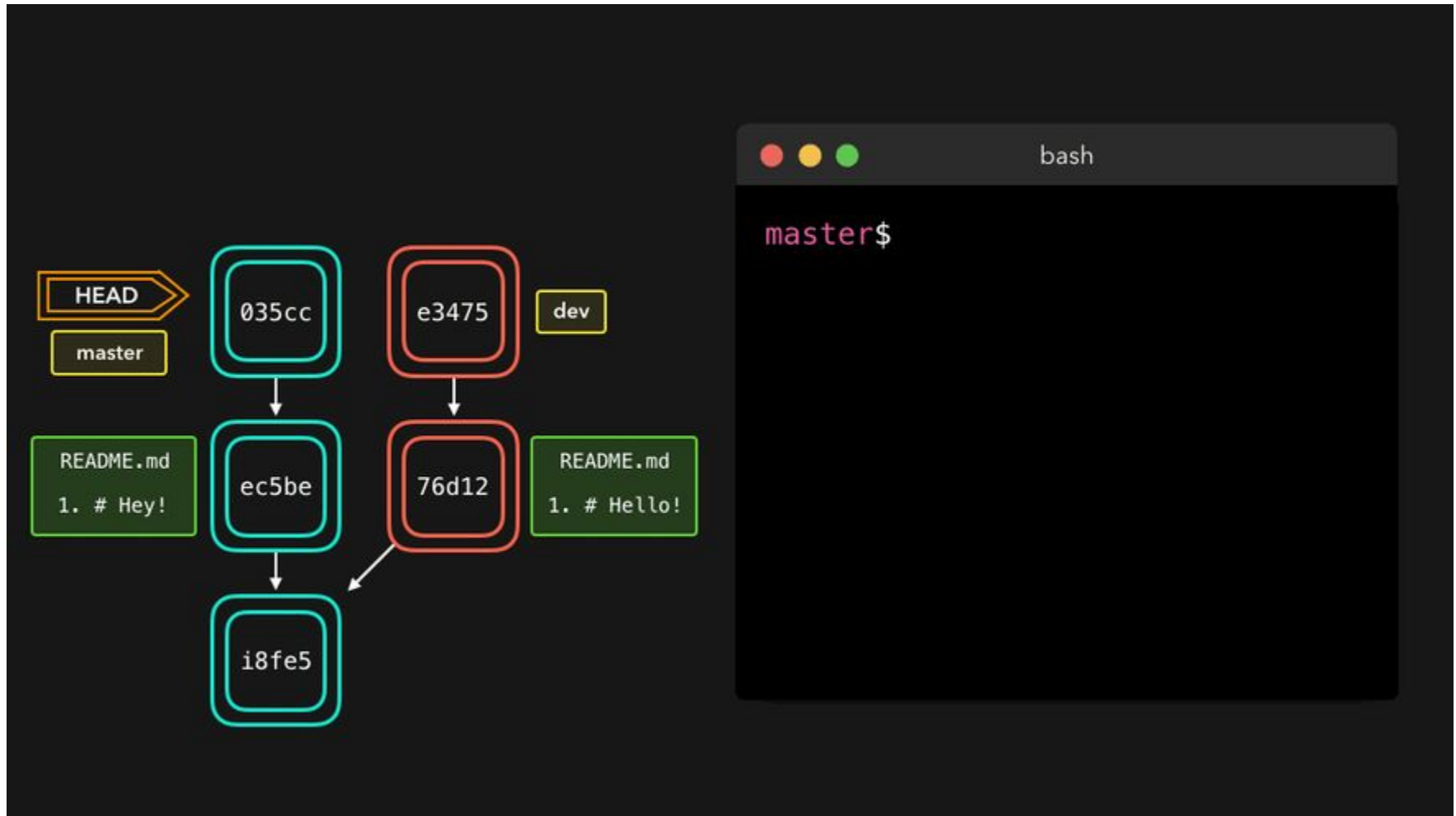
master\$

Git | Merging (no-fast-forward)

Default behavior when current branch contains commits that the merging branch doesn't have

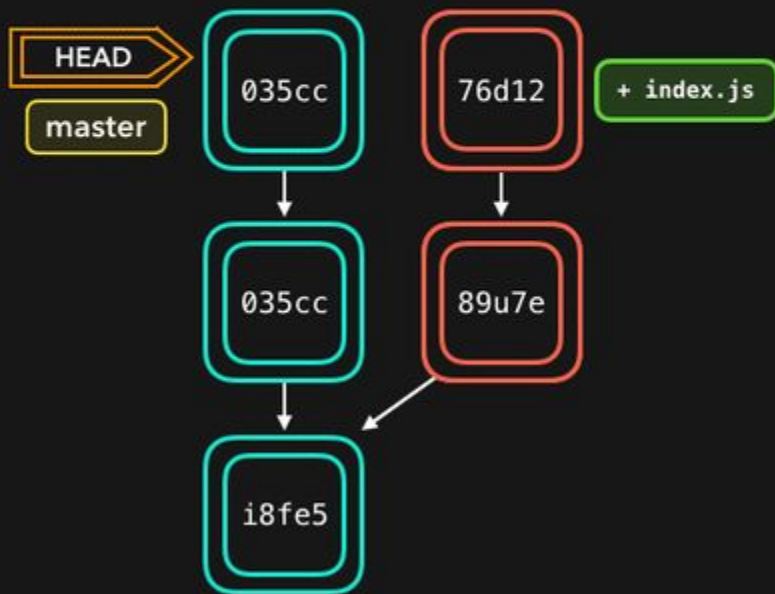
Creates a new commit which merges two branches together without modifying existing branches

Merge Conflict



Cherry-pick

Cherry picking is the act of picking a commit from a branch and applying it to another. `git cherry-pick` can be useful for undoing changes. For example, say a commit is accidentally made to the wrong branch. You can switch to the correct branch and cherry-pick the commit to where it should belong.



```
bash
master$
```

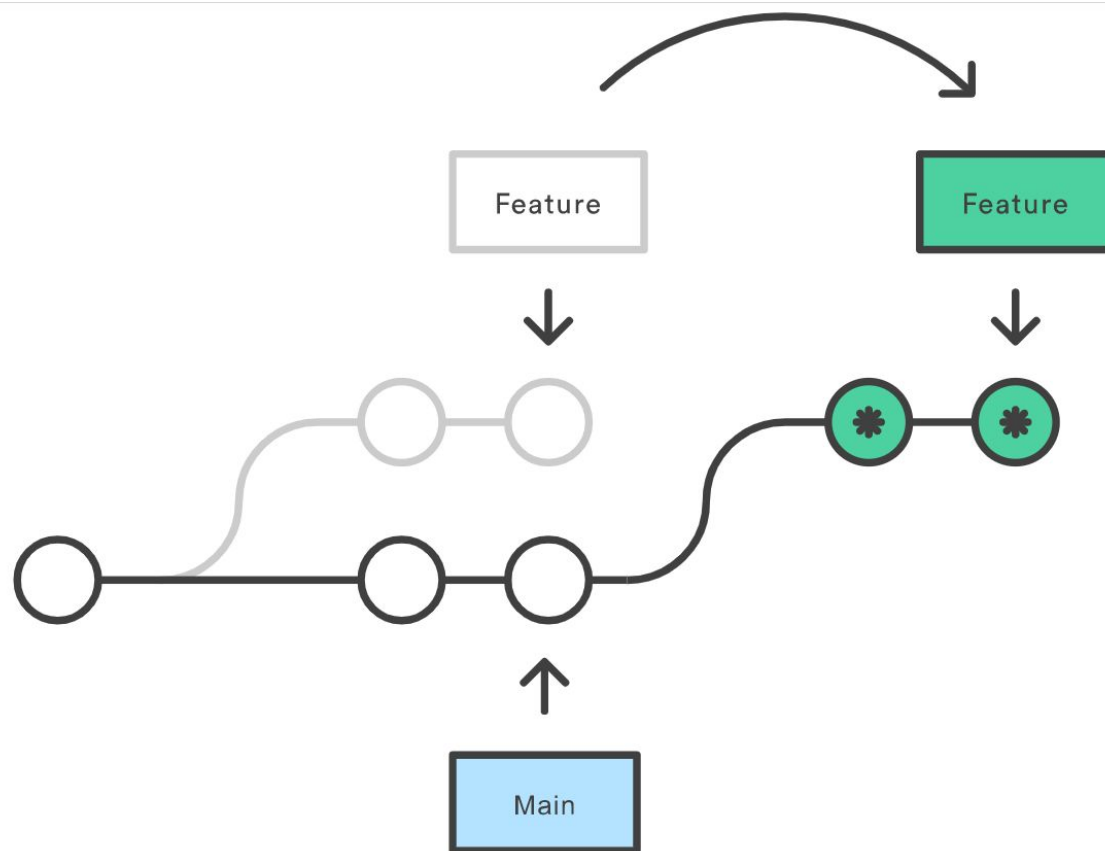
Git | Cherry-picking

Creates a new commit with the changes that the cherry-picked commit introduced.

By default, Git will only apply the changes if the current branch does not have these changes in order to prevent an empty commit.

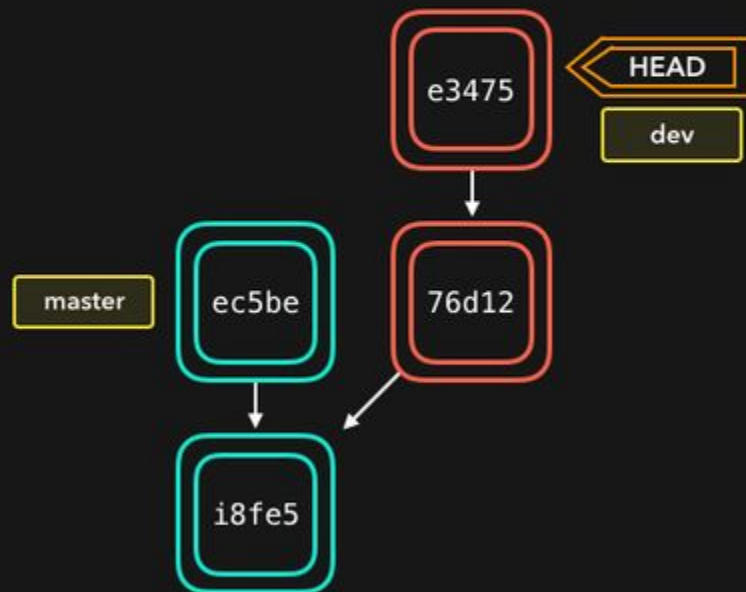
Rebase

Rebasing is the process of moving or combining a sequence of commits to a new base commit. Rebasing is most useful and easily visualized in the context of a feature branching workflow. The general process can be visualized as the following:



* Brand New Commits

Rebase



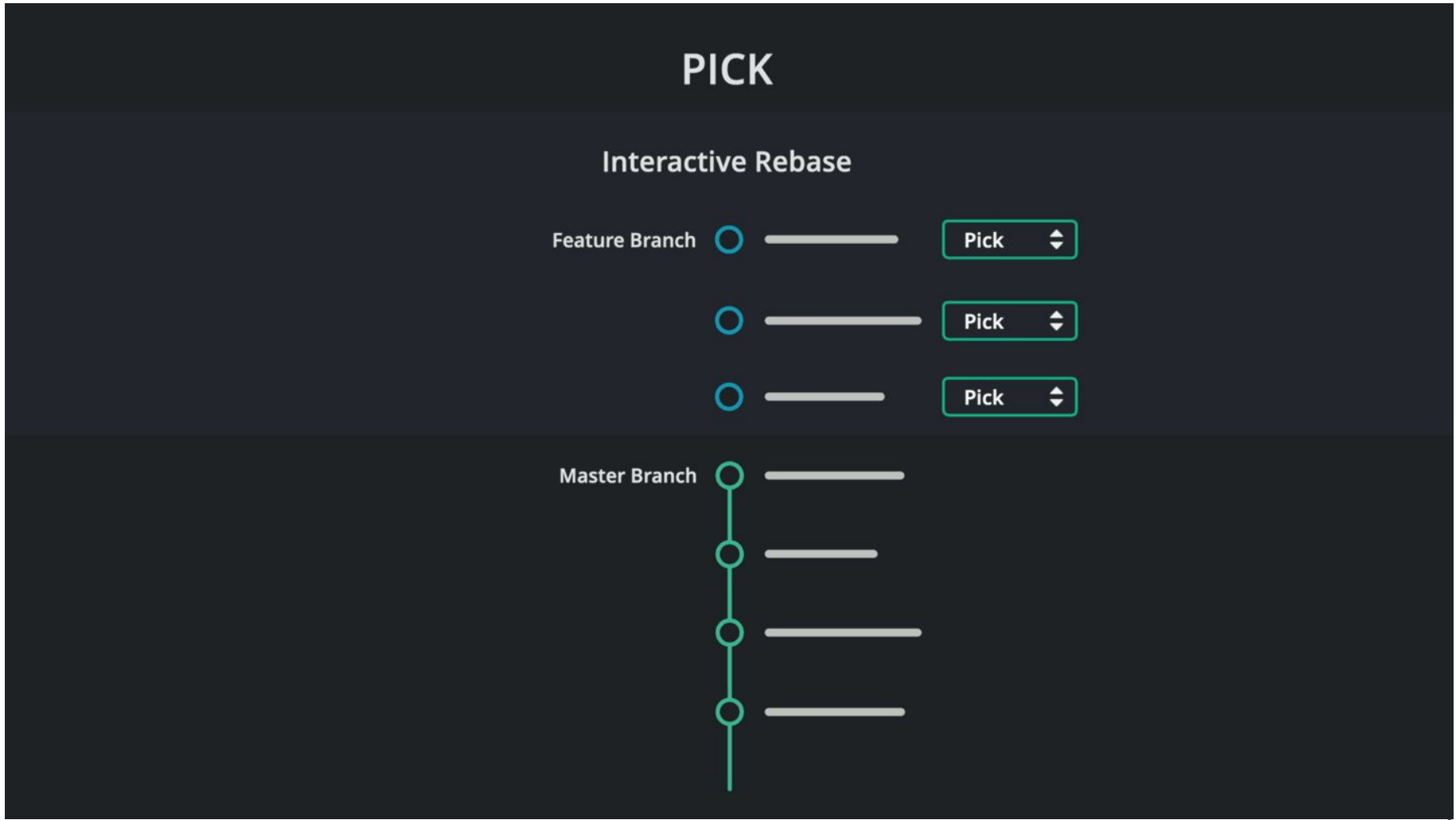
bash

dev\$

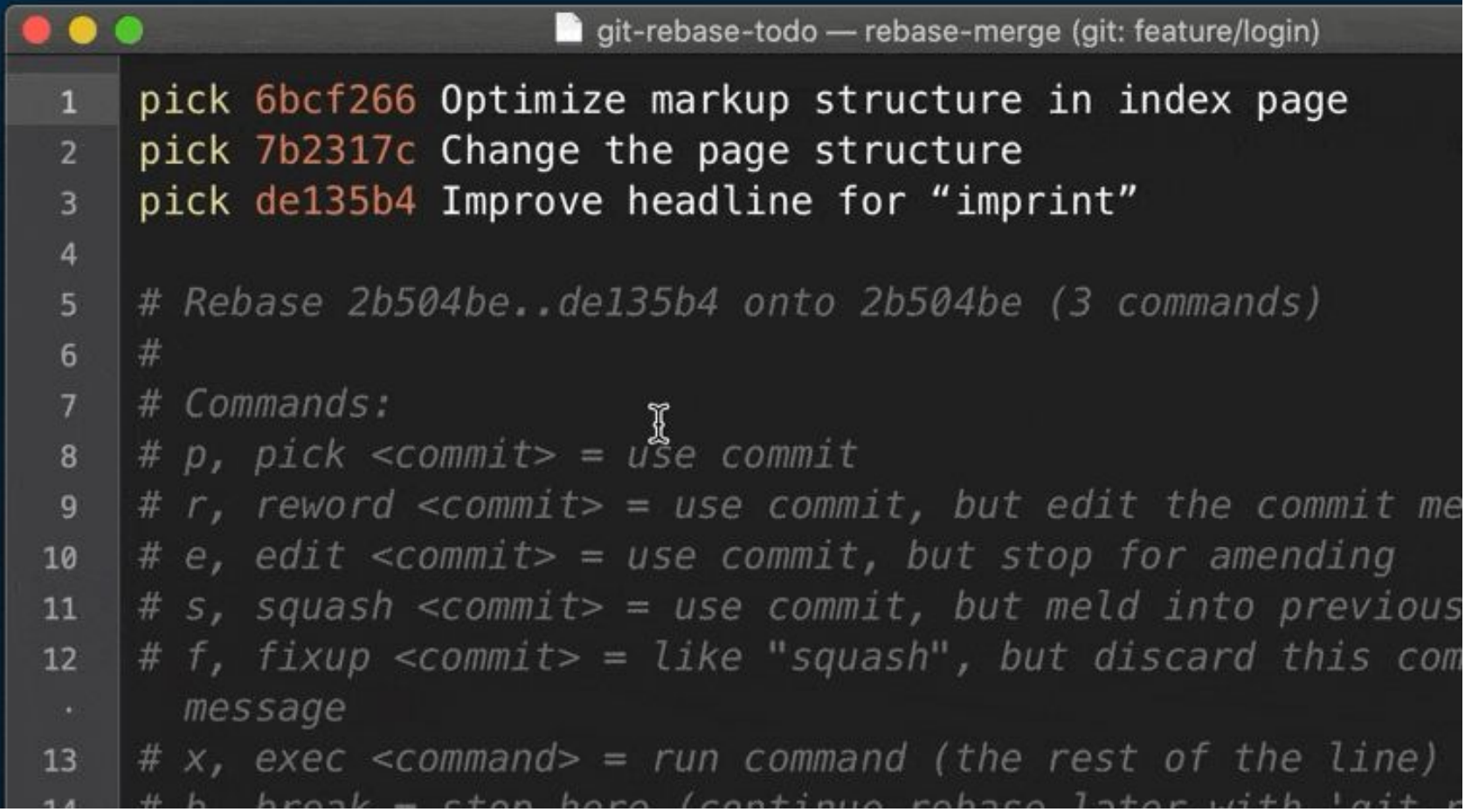
Git | Rebasing

Copies commits on top of another branch without creating a commit, which keeps a linear history

Changes the history as new hashes are created for the copied commits



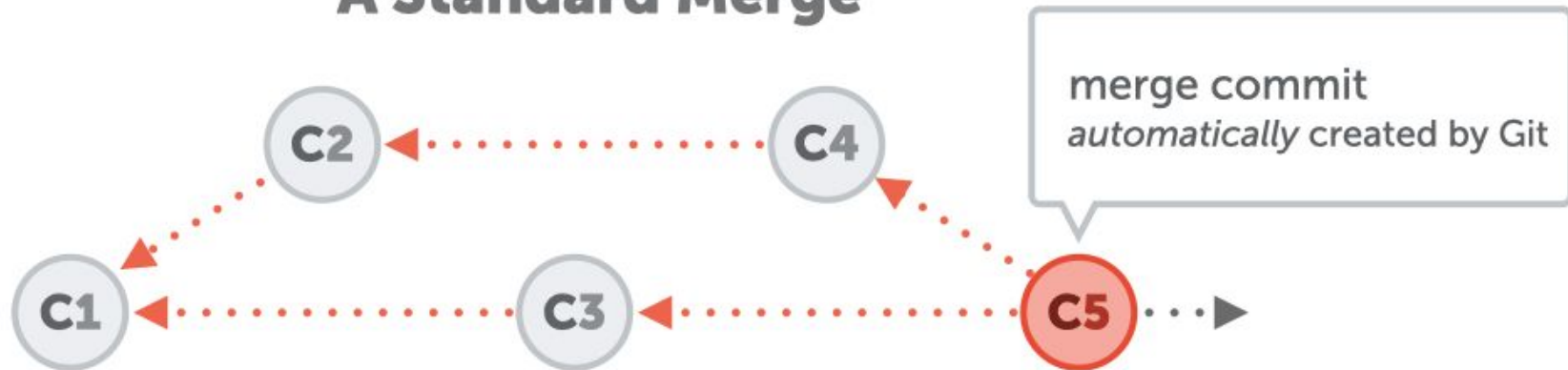
git rebase -i HEAD~3



```
git-rebase-todo — rebase-merge (git: feature/login)
1  pick 6bcf266 Optimize markup structure in index page
2  pick 7b2317c Change the page structure
3  pick de135b4 Improve headline for "imprint"
4
5  # Rebase 2b504be..de135b4 onto 2b504be (3 commands)
6  #
7  # Commands:
8  # p, pick <commit> = use commit
9  # r, reword <commit> = use commit, but edit the commit me
10 # e, edit <commit> = use commit, but stop for amending
11 # s, squash <commit> = use commit, but meld into previous
12 # f, fixup <commit> = like "squash", but discard this com
   message
13 # x, exec <command> = run command (the rest of the line)
14 # b, break = stop here (continue rebase later with 'git r
```

```
git merge --squash feature/login
```

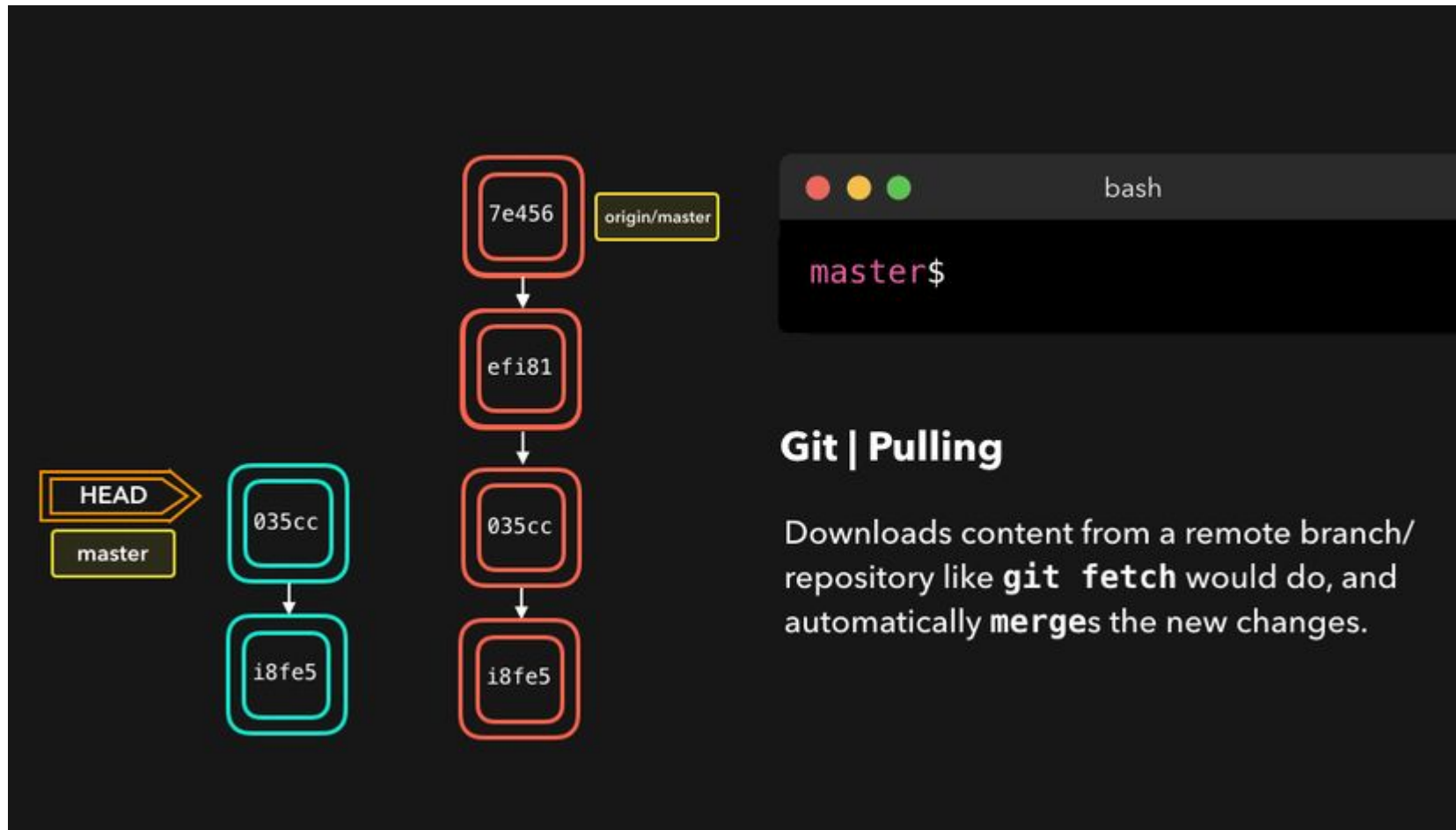
A Standard Merge



A Merge using "--squash"



Pulling



Push

git push

(master) branch
Commit:1564b4f



Remote

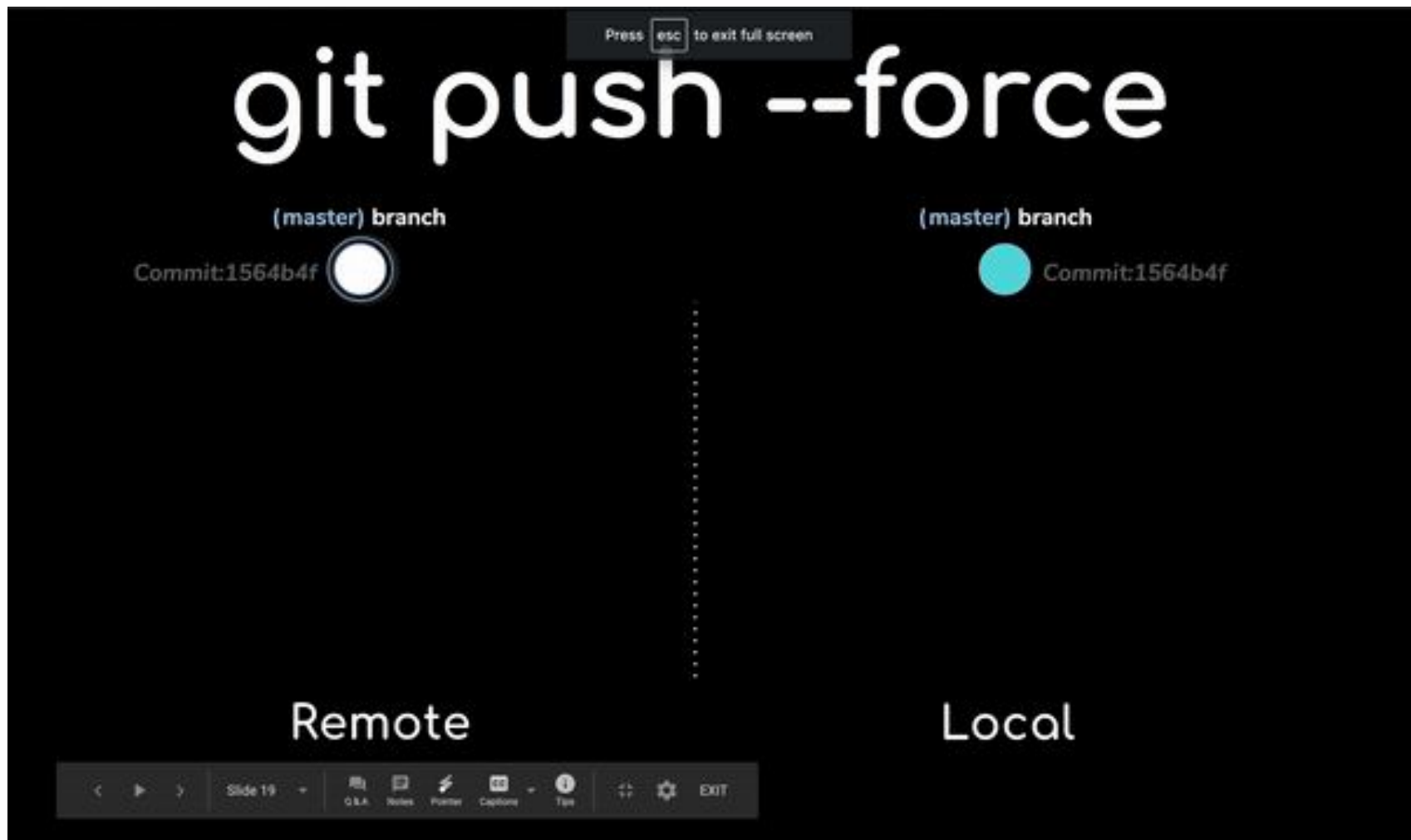
(master) branch
Commit:1564b4f



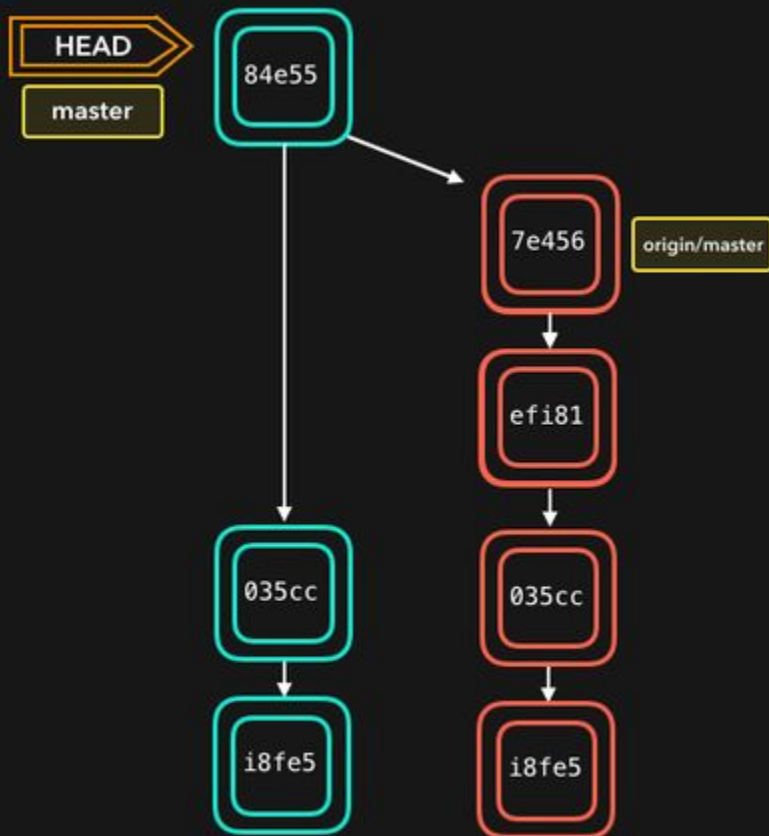
Commit:d023360

Local

push --force



Reflog



```
bash

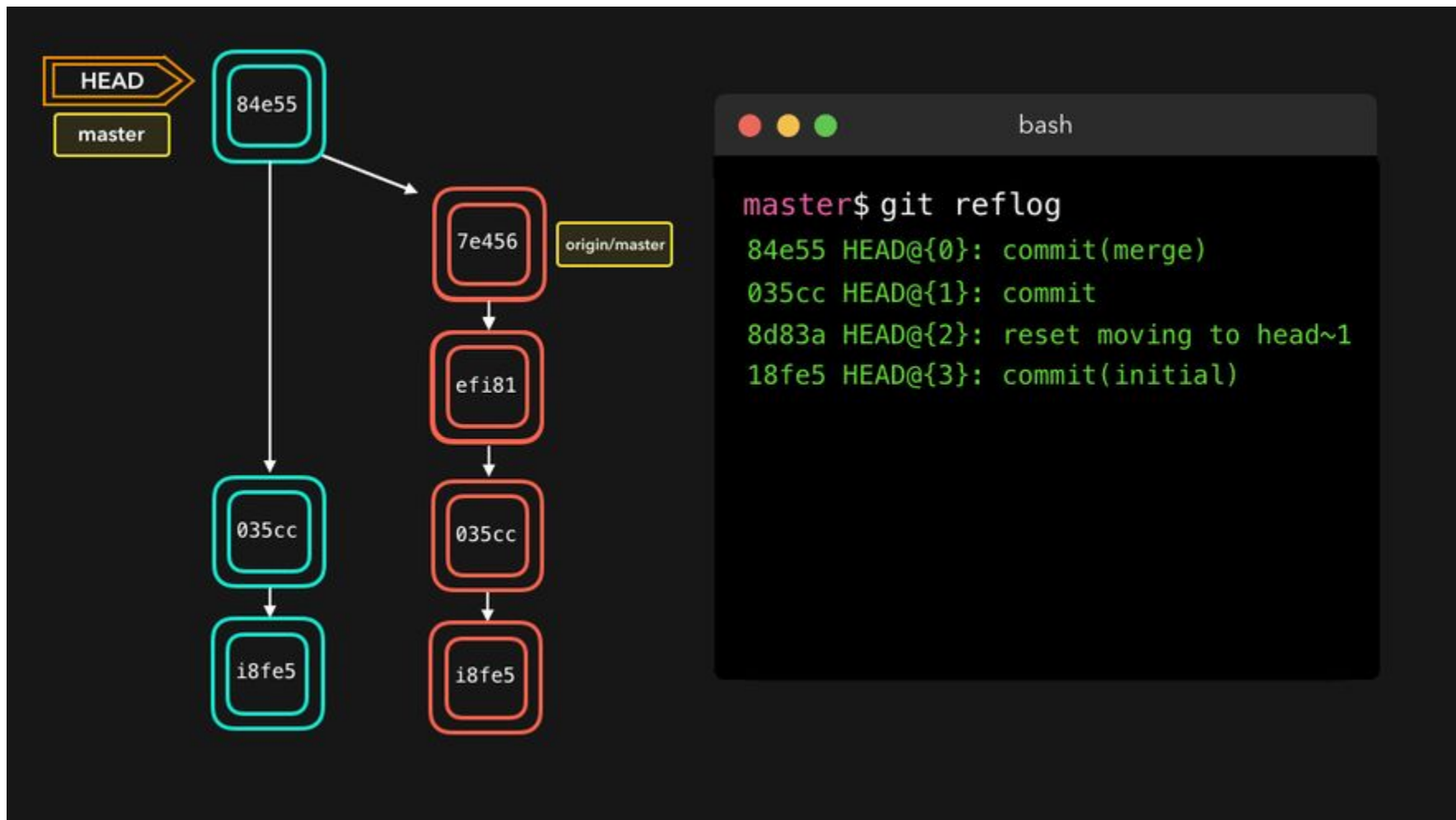
master$
```

Git | Reflog

Shows the history of actions in the repo.

With this information, you can easily undo changes that have been made to a repository with **git reset**

reflog and reset

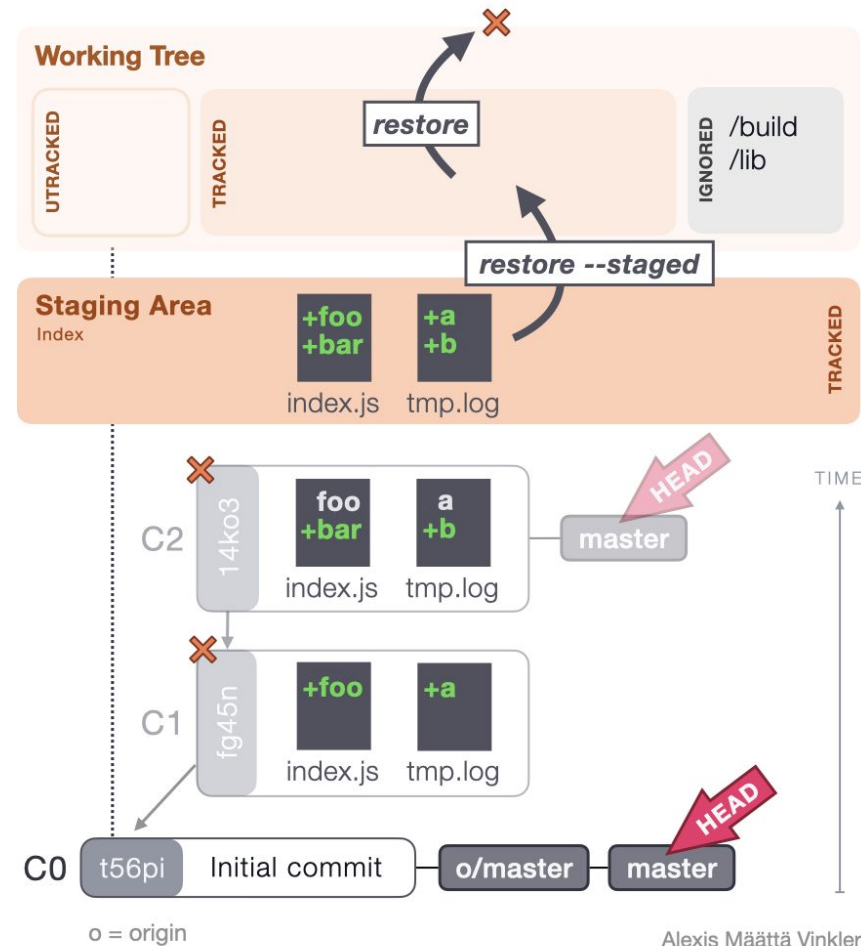


Reset

Git | reset

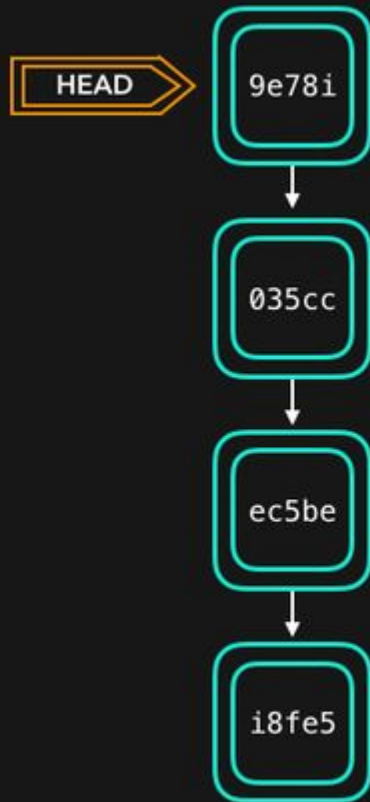
1. Undo mistakes by soft resetting the branch to a commit prior to the faulty commits, leaving the compounded changes in the *Staging Area*.
2. Discard the unwanted changes with **restore** and create a new commit.

```
(zsh)
$ git status
On branch master
nothing to commit, working tree clean
$ git reset --soft t56pi
$ git status
On branch master
To be committed ("restore --staged <f>" to unstage):
  modified: index.js
  modified: tmp.log
$
```



Alexis Määttä Vinkler
git-init.com

Reset --hard



```
bash
master$
```

Git | Hard reset

Points **HEAD** to the specified commit

Discards changes that have been made since the new commit that **HEAD** points to, and deletes changes in working directory

Restore

The "restore" command helps to unstage or even discard uncommitted local changes.

Usage Examples

To only *unstage* a certain file and thereby undo a previous `git add`, you need to provide the `--staged` flag:

```
$ git restore --staged index.html
```

You can of course also remove multiple files at once from the Staging Area:

```
$ git restore --staged *.css
```

If you want to discard uncommitted local changes in a file, simply omit the `--staged` flag. Keep in mind, however, that you cannot undo this!

```
$ git restore index.html
```

Another interesting use case is to restore a specific historic revision of a file:

```
$ git restore --source 7173808e index.html  
$ git restore --source master~2 index.html
```

The first example will restore the file as it was in commit #7173808e, while the second one will restore it as it was "two commits before the current tip of the master branch".

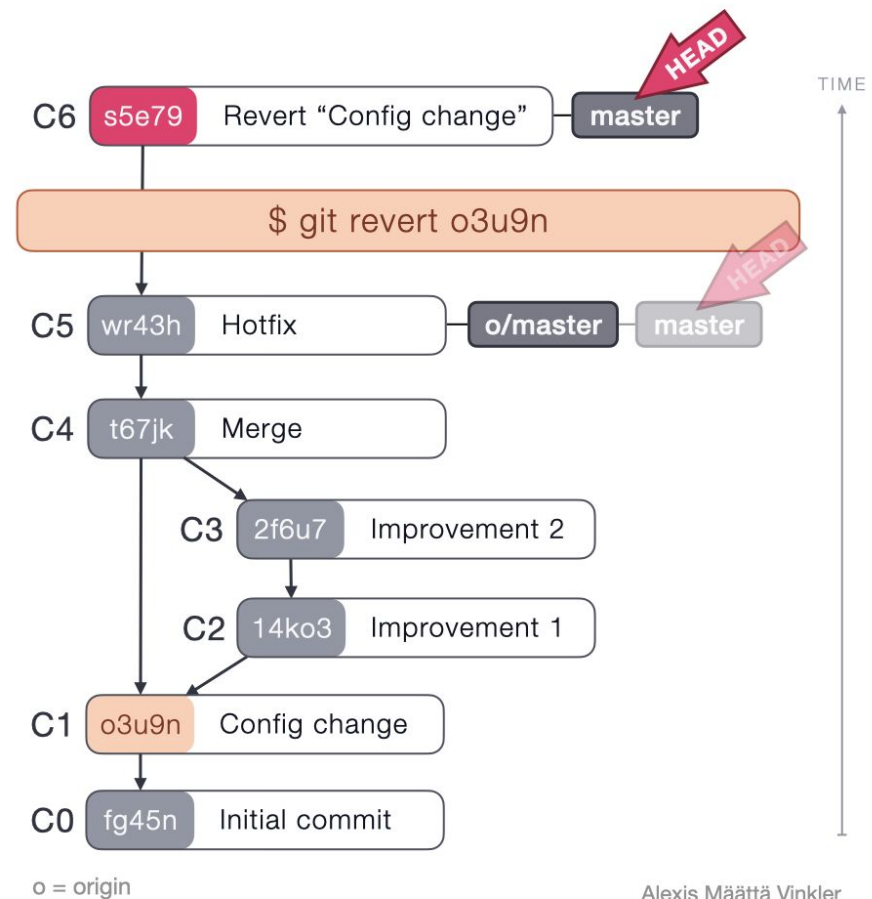
Revert

Git | revert

Revert creates a new commit that reverts the changes made by another commit.

The command is essentially a reverse git **cherry-pick**, as it creates a new commit that applies the exact opposite of the change introduced in the target commit, essentially undoing or reverting it.

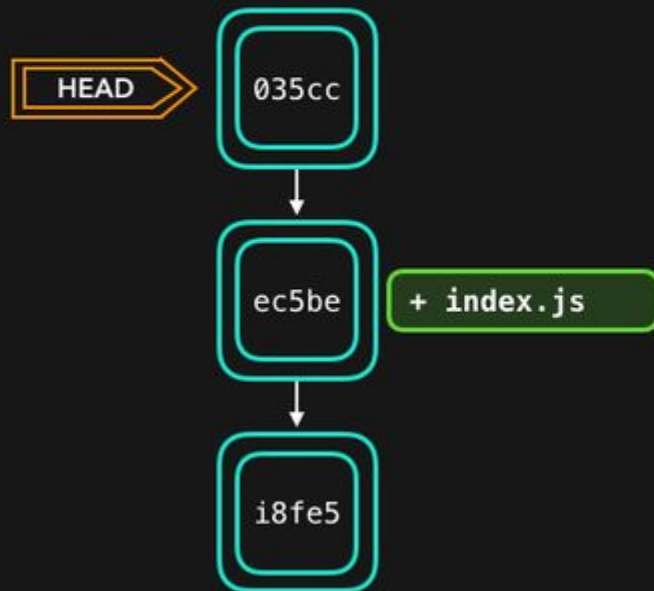
```
(zsh)
$ git revert o3u9n
[master s5e79] Revert "Config change"
1 file changed, 1 deletion(-)
$
```



Alexis Määttä Vinkler
git-init.com

Revert

<https://stephencharlesweiss.com/git-restore-reset-revert>



```
bash
dev$
```

Git | Reverting

Reverts the changes that commits introduce.
Creates a new commit with the reverted changes.

Flow

Git Flow



git remote

\$ git remote add remote_name remote_url

\$ git remote -v

origin https://github.com/OWNER/REPOSITORY.git (fetch)

origin https://github.com/OWNER/REPOSITORY.git (push)

\$ git remote set-url --add --push origin https://github.com/OWNER/REPOSITORY.git

\$ git remote set-url --add --push origin ssh://abc123@APP-OWNER.rhcloud.com/~git/APP

\$ git remote -v

origin https://github.com/OWNER/REPOSITORY.git (fetch)

origin https://github.com/OWNER/REPOSITORY.git (push)

origin ssh://abc123@APP-OWNER.rhcloud.com/~git/APP.git/ (push)

Code Flow Branches

These branches which we expect to be permanently available on the repository follow the flow of code changes starting from development until the production.

- **Development** (*dev*)
All new features and bug fixes should be brought to the development branch. Resolving developer codes conflicts should be done as early as here.
- **QA/Test** (*test*)
Contains all codes ready for QA testing.
- **Staging** (*staging* , Optional)
It contains tested features that the stakeholders wanted to be available either for a demo or a proposal before elevating into the production. Decisions are made here if a feature should be finally brought to the production code.
- **Master** (*master*) - **Main** (*main*)
The production branch, if the repository is published, this is the default branch being presented.

Except for Hotfixes, we want our codes to follow a one-way merge starting from development > test > staging > production.

Temporary Branches

As the name implies, these are disposable branches that can be created and deleted by need of the developer or deployer.

- **Feature**

Any code changes for a new module or use case should be done on a feature branch. This branch is created based on the current development branch. When all changes are Done, a Pull Request/Merge Request is needed to put all of these to the development branch.

Examples:

- feature/integrate-swagger
- feature/JIRA-1234
- feature/JIRA-1234_support-dark-theme

It is recommended to use all lowercase letters and hyphen (-) to separate words unless it is a specific item name or ID. Underscore (_) could be used to separate the ID and description.

- **Bug Fix**

If the code changes made from the feature branch were rejected after a release, sprint or demo, any necessary fixes after that should be done on the bugfix branch.

Examples:

- bugfix/more-gray-shades
- bugfix/JIRA-1444_gray-on-blur-fix

- **Hot Fix**

If there is a need to fix a blocker, do a temporary patch, apply a critical framework or configuration change that should be handled immediately, it should be created as a Hotfix. It does not follow the scheduled integration of code and could be merged directly to the production branch, then on the development branch later.

Examples:

- hotfix/disable-endpoint-zero-day-exploit
- hotfix/increase-scaling-threshold

Temporary Branches

- **Experimental**

Any new feature or idea that is not part of a release or a sprint. A branch for playing around.

Examples:

- experimental/dark-theme-support

- **Build**

A branch specifically for creating specific build artifacts or for doing code coverage runs.

Examples:

- build/jacoco-metric

- **Release**

A branch for tagging a specific release version

Examples:

- release/myapp-1.01.123

- Git also supports **tagging** a specific commit history of the repository. A release branch is used if there is a need to make the code available for checkout or use.

- **Merging**

A temporary branch for resolving merge conflicts, usually between the latest development and a feature or Hotfix branch. This can also be used if two branches of a feature being worked on by multiple developers need to be merged, verified and finalized.

Examples:

- merge/dev_lombok-refactoring
- merge/combined-device-support

GitHub

- [GitHub.com](https://github.com) is a site for online storage of Git repositories.
 - You can create a **remote repo** there and push code to it.
 - Many open source projects use it, such as the Linux kernel.
 - You can get free space for open source projects, or you can pay for private projects.
 - Free private repos for educational use: github.com/edu
- Question: Do I always have to use GitHub to use Git?
 - Answer: No! You can use Git locally for your own purposes.
 - Or you or someone else could set up a server to share files.
 - Or you could share a repo with users on the same file system, as long everyone has the needed file permissions).