

GIT

A deep dive



IT-HÖGSKOLAN

Här startar din IT-karriär.

THIS IS GIT. IT TRACKS COLLABORATIVE WORK
ON PROJECTS THROUGH A BEAUTIFUL
DISTRIBUTED GRAPH THEORY TREE MODEL.

COOL. HOW DO WE USE IT?

NO IDEA. JUST MEMORIZE THESE SHELL
COMMANDS AND TYPE THEM TO SYNC UP.
IF YOU GET ERRORS, SAVE YOUR WORK
ELSEWHERE, DELETE THE PROJECT,
AND DOWNLOAD A FRESH COPY.



<https://xkcd.com/1597/>



IT-HÖGSKOLAN

Här startar din IT-karriär.

Git

- Installera git

Windows: <https://git-scm.com/>

Mac: <https://www.atlassian.com/git/tutorials/install-git#mac-os-x>

Linux: <https://www.atlassian.com/git/tutorials/install-git#linux>

- Online bok: <https://git-scm.com/book/en/v2>

<https://github.com/git-guides>

<https://www.atlassian.com/git/tutorials>

- Cheat sheets:

<https://www.atlassian.com/git/tutorials/atlassian-git-cheatsheet>

<https://education.github.com/git-cheat-sheet-education.pdf>

- Online training with visualizer:

<https://git-school.github.io/visualizing-git/>



IT-HÖGSKOLAN

Här startar din IT-karriär.

Git ...

- is free and open source
- is originally created by Linus Torvalds in 2005.
- <https://www.linuxfoundation.org/blog/10-years-of-git-an-interview-with-git-creator-linus-torvalds/>
- is unlike older centralized version control systems such as CVS and SVN, **distributed**: every developer has the full history of their code repository locally.
- has excellent support for branching, merging, and rewriting repository history.



Version Control

- Version control, also known as source control, is the practice of tracking and managing changes to software code.
- Who made which changes and at what time.
- If a mistake is made, turn back the time to earlier versions of the code, or just remove specific changes which are deemed undesirable (e.g., because they introduced a bug)
- Source code management (SCM)
- Software Configuration Management (SCM).



GIT BASICS

<code>git init <directory></code>	Create empty Git repo in specified directory. Run with no arguments to initialize the current directory as a git repository.
<code>git clone <repo></code>	Clone repo located at <repo> onto local machine. Original repo can be located on the local filesystem or on a remote machine via HTTP or SSH.
<code>git config user.name <name></code>	Define author name to be used for all commits in current repo. Devs commonly use --global flag to set config options for current user.
<code>git add <directory></code>	Stage all changes in <directory> for the next commit. Replace <directory> with a <file> to change a specific file.
<code>git commit -m "<message>"</code>	Commit the staged snapshot, but instead of launching a text editor, use <message> as the commit message.
<code>git status</code>	List which files are staged, unstaged, and untracked.
<code>git log</code>	Display the entire commit history using the default format. For customization see additional options.
<code>git diff</code>	Show unstaged changes between your index and working directory.



Configure git

- `~/.gitconfig`

On Windows systems, Git looks for the `.gitconfig` file in the `$HOME` directory (`C:\Users\%USER` for most people).

- Show all settings and from what file they are.

```
git config --list --show-origin
```

- Your Identity (Global setting, can be overridden, skip `--global` and stand in local gitrepo)

```
git config --global user.name "John Doe"
```

```
git config --global user.email johndoe@example.com
```

Use same email as used on github account for insights to work properly.



IT-HÖGSKOLAN

Här startar din IT-karriär.

Signing our commits

- You can sign your Git commits cryptographically by using a GPG key.
- This will prove the commit came from you.
- This is needed because it is pretty easy to add anyone as the author of a commit, `--author="Author Name <email@address.com>"`
- Trusting Other People's Code.
- <https://binx.io/2021/12/06/why-you-should-start-signing-your-git-commits-today/>
- <https://docs.github.com/en/authentication/managing-commit-signature-verification/about-commit-signature-verification>

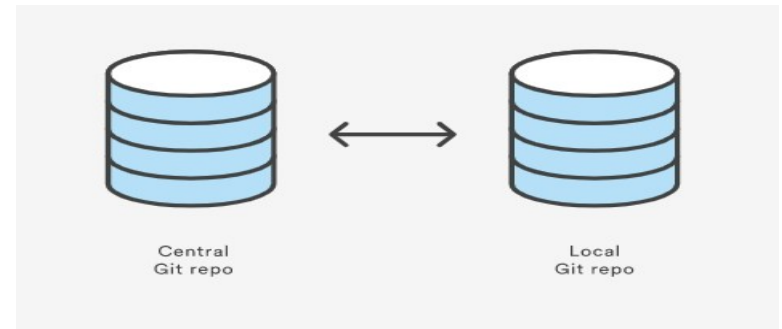


git init git clone

- Create a repository with git init
- Will create a hidden folder named .git
- Changes in one repo can be transferred to another repo
- Git clone to make a copy of another repository.

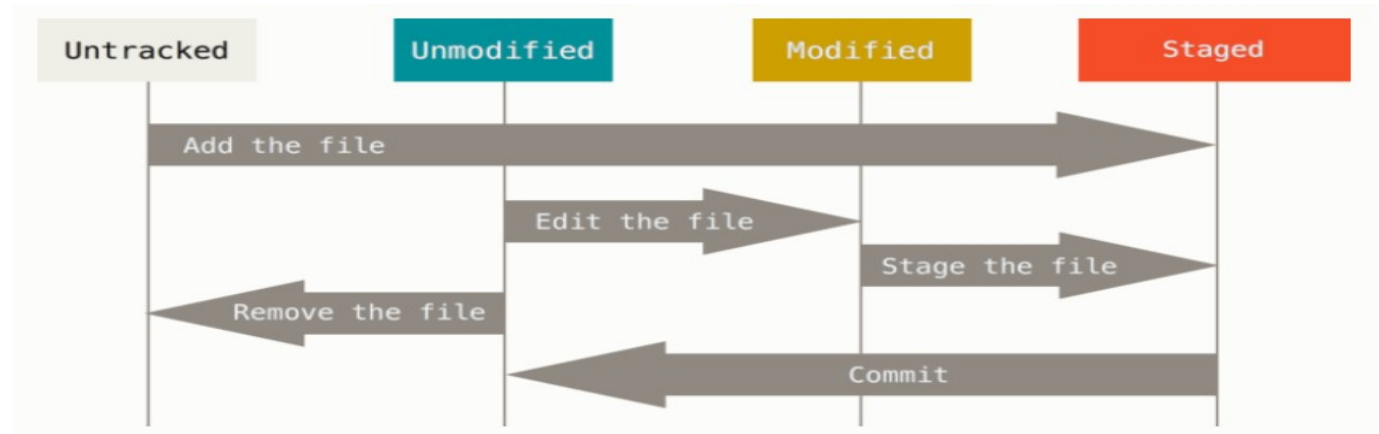
Clone repo from github.com

- Github fork, serverside clone



What does git know?

- Files can be tracked or untracked.
- Tracked files are files that were in the last snapshot, (snapshot is a capture of a file. they can be unmodified, modified, or staged.)
- Check status with **git status**



Git tracking new files and staging modified files

`git add filename`

- Ignoring files, git won't ask about them anymore, add them to the file: `.gitignore`
- `git reset HEAD <file>` unstages any modifications made to the file since the last commit.
- Remove a file from index, git stops tracking changes but file is still on disk.
 - `git rm --cached [file]`
- Remove file from index and disk
 - `git rm [file]`

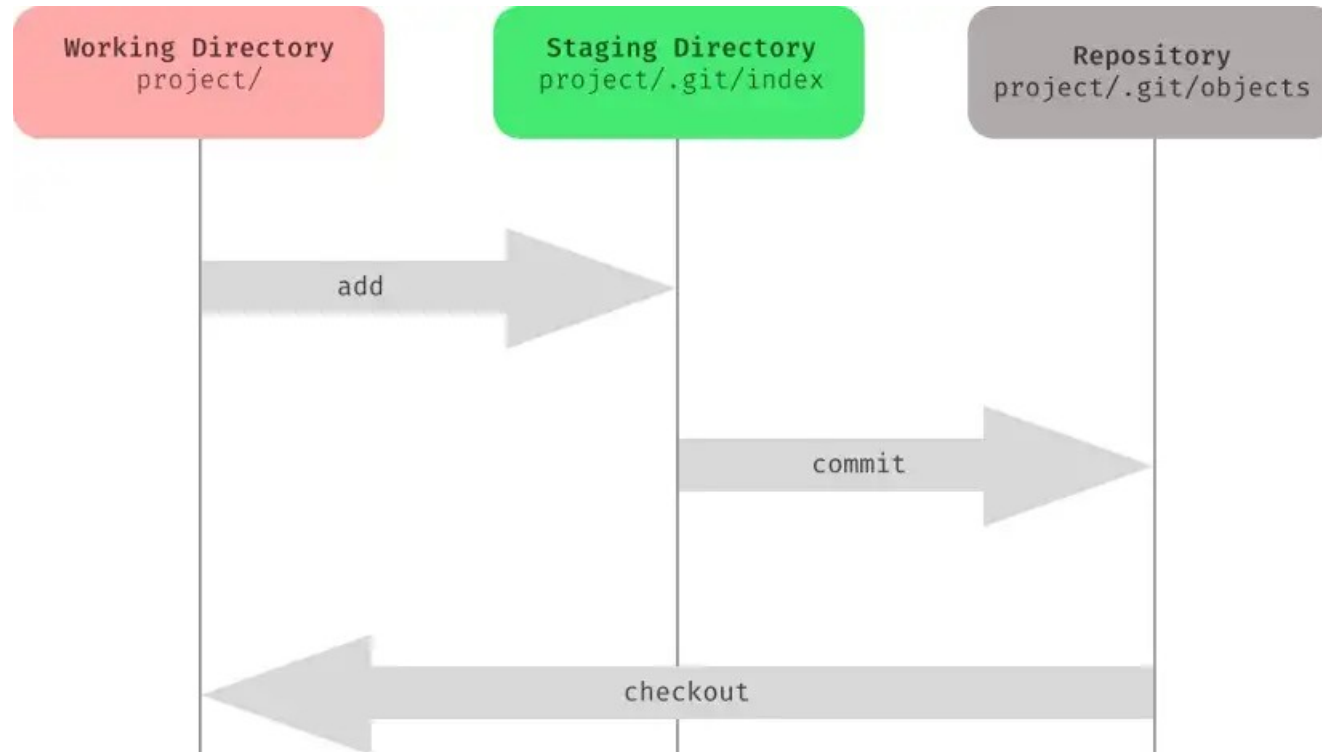


Committing Your Changes, take a snapshot

- Now that your staging area is set up the way you want it, you can commit your changes.
- Remember that anything that is still unstaged — any files you have created or modified that you haven't run **git add** on since you edited them — won't go into this commit. They will stay as modified files on your disk
- `git commit -m "Message describing the commit"`
- `git commit --amend`
- `git commit -a` (Git automatically stages every file that is already tracked)



Git index



<https://konrad126.medium.com/understanding-git-index-4821a0765cf>



IT-HÖGSKOLAN

Här startar din IT-karriär.

- Working Tree - Staging Area(index) - History
- Add files to staging area with `git add`.
- Only files in staging area will be snapshotted in next commit.
- Check with **git status** for status of working tree and staging area.
- **git diff** shows difference between tracked files in working tree and staging area
- **git diff --staged** shows difference between staging area and latest commit
- **git log** shows us the commits with latest on top.
- **git log -p** shows what was changed for each commit



Undo changes

- We have changed a file in working tree and want to undo that. Get latest version from staging area.
- **git checkout -- filename**
- We have staged a change to be committed and want to undo that.
- **git reset HEAD filename**
- Will only change the staging area, use checkout to update working tree.



Restore (deleted) file from earlier commit

- **git log -- <filename>**
- Will show commits with changes to the file.
- **git checkout commithash -- <filename>**
- Will update both working tree and index.
- **More later...**



Stop tracking a tracked file

- Scenario: You accidentally added application.log to the repository, now every time you run the application, Git reports there are unstaged changes in application.log. You put *.log in the .gitignore file, but it's still there—how do you tell git to “undo” tracking changes in this file?
- Undo with: `git rm --cached application.log`
- More scenarios:
- <https://github.blog/2015-06-08-how-to-undo-almost-anything-with-git/>



Git mv

- "Other version control systems (eg. Subversion and Perforce) treat file renames specially. Why doesn't Git?"
- Linus says: "Git tracks exactly what matters, namely "collections of files". Nothing else is relevant, and even thinking that it is relevant only limits your world-view."
- <https://web.archive.org/web/20160304045715/http://permalink.gmane.org/gmane.comp.version-control.git/217>



Renaming files with case

- There's a niche case where `git mv` is very useful: when you want to change the casing of a file name on a case-insensitive file system.
- Both APFS (mac) and NTFS (windows) are, by default, case-insensitive (but case-preserving).
- Suppose you are working on a mac and have a file `Mytest.txt` managed by git. You want to change the file name to `MyTest.txt`.
 - `$ mv Mytest.txt MyTest.txt`
 - `overwrite MyTest.txt? (y/n [n]) y`
 - `$ git status`
- Use `git mv Mytest.txt MyTest.txt` instead.

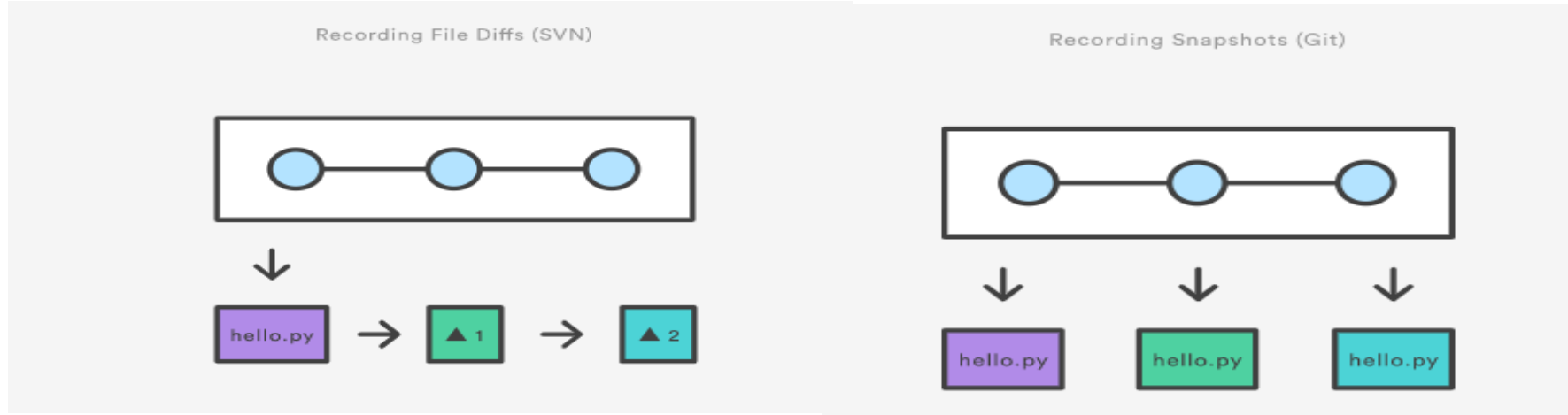


Rename/Move

- Git has a rename command `git mv`, but that is just a convenience. The effect is indistinguishable from removing the file and adding another with different name and the same content.
- `Git mv` is a shorthand for:
 - `mv oldname newname`
 - `git add newname`
 - `git rm oldname`
- Moving the file via `git mv` adds the new path to the index, but not any modified content in the file.



Git uses snapshots when committing



Since git stores the complete version of a file when it's changed it's much faster to restore a specific version of the file.

The seven rules of a great Git commit message

- 1) Separate subject from body with a blank line
- 2) Limit the subject line to 50 characters
- 3) Capitalize the subject line
- 4) Do not end the subject line with a period
- 5) Use the imperative mood in the subject line
- 6) Wrap the body at 72 characters
- 7) Use the body to explain what and why vs. How

<https://cbea.ms/git-commit/>

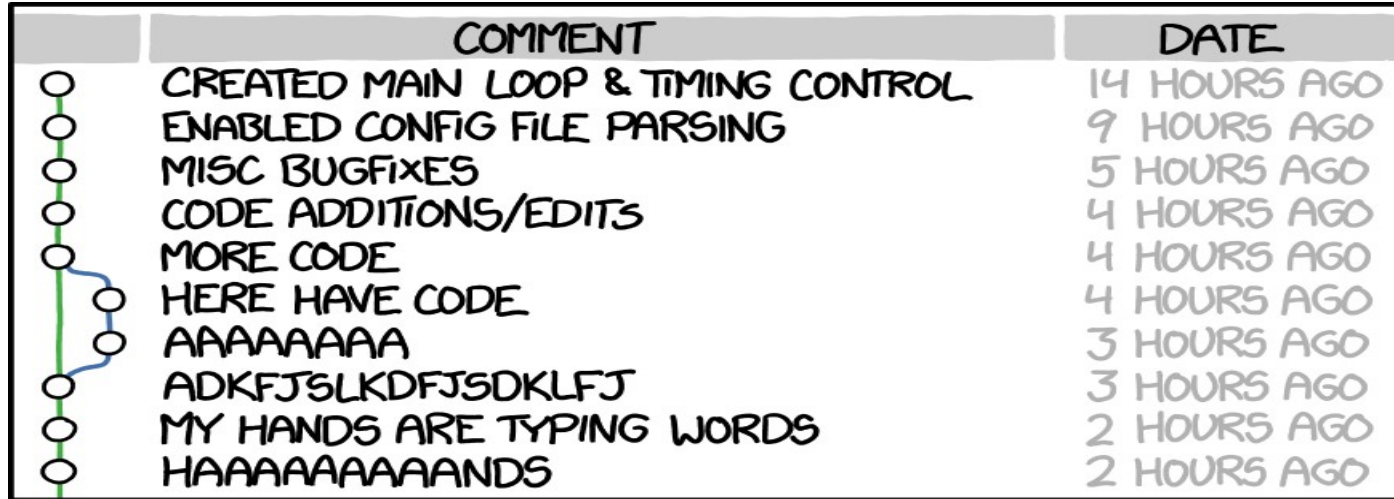
<https://wiki.openstack.org/wiki/GitCommitMessages>



IT-HÖGSKOLAN

Här startar din IT-karriär.

Try to write good commit messages and commit often.



	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE	4 HOURS AGO
○	AAAAAAAAA	3 HOURS AGO
○	ADKFJSLKDFJSDKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT
MESSAGES GET LESS AND LESS INFORMATIVE.

<https://xkcd.com/1296/>



IT-HÖGSKOLAN

Här startar din IT-karriär.

What does commit do?

- Staged files are saved and a commit object is created.
- Commits are snapshots, not diffs!
- Each commit saves metadata:
- Who, what, when, parent(s)
- Message
- SHA-1 hash value (git rev-parse --short HEAD)



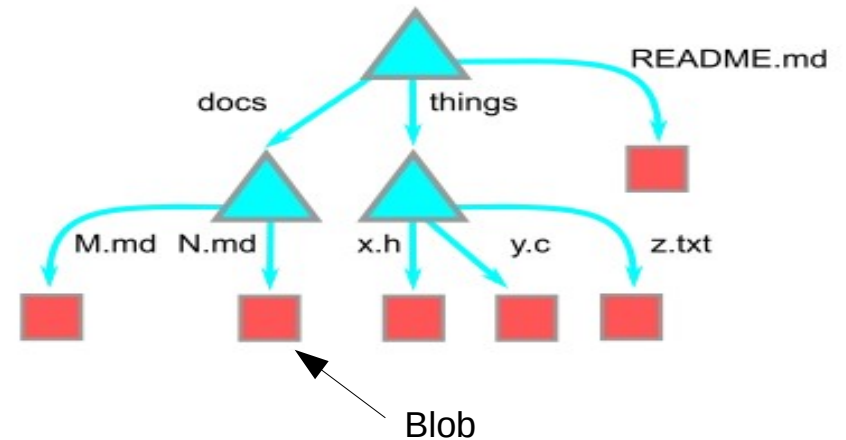
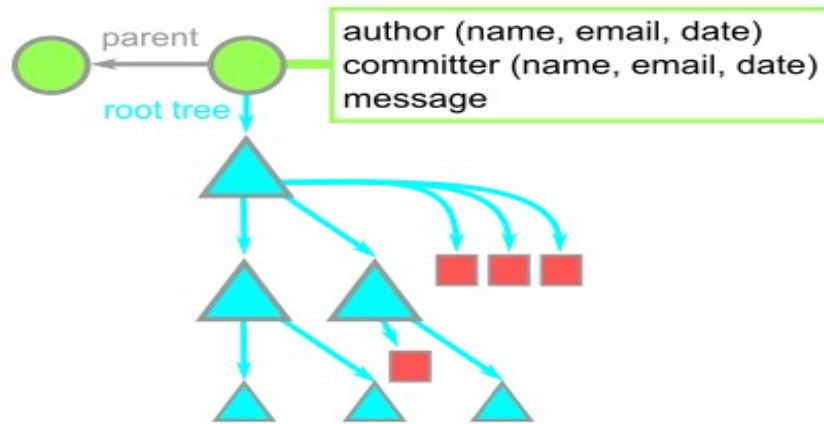
Commit

- A commit is a snapshot in time. Each commit contains a pointer to its root tree, representing the state of the working directory at that time.
- The commit has a list of parent commits corresponding to the previous snapshots.
- A commit with no parents is a root commit and a commit with **multiple parents** is a **merge commit**.
- Commits also contain metadata describing the snapshot such as author and committer (including name, email address, and date) and a commit message.
- The **commit message** is an opportunity for the commit author to describe the purpose of that commit with respect to the parents.



Blobs are file contents

- <https://github.blog/2020-12-17-commits-are-snapshots-not-diffs/>





Branchnames, Tag names and HEAD.
Branchname points to a commit.

You can think of the HEAD as the "current branch". When you switch branches with git checkout, the HEAD revision changes to point to the tip of the new branch.
(cat .git/HEAD)

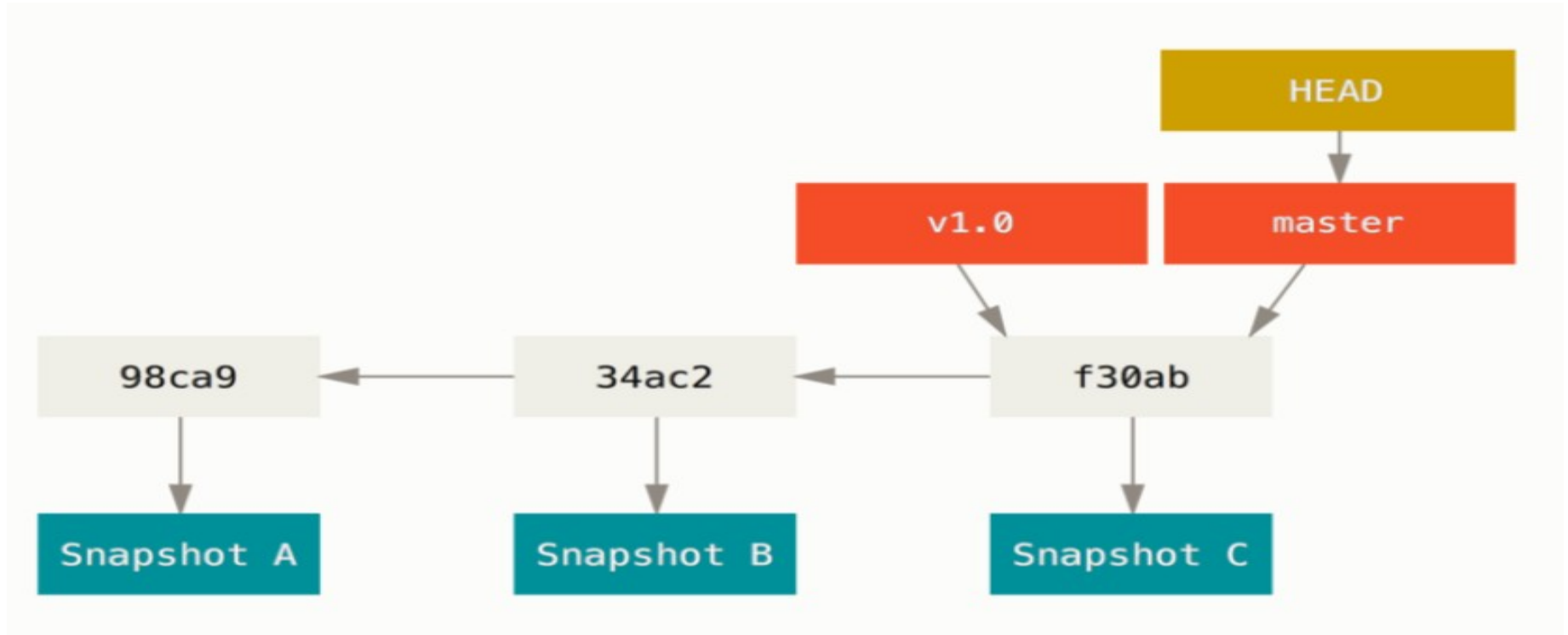
Tags is a way of tagging a specific commit. Often used for adding version numbers to releases.

It is possible for HEAD to refer to a specific revision that is not associated with a branch name. This situation is called a detached HEAD. This happens when you checkout a specific commit, tag, or remote branch.



IT-HÖGSKOLAN

Här startar din IT-karriär.

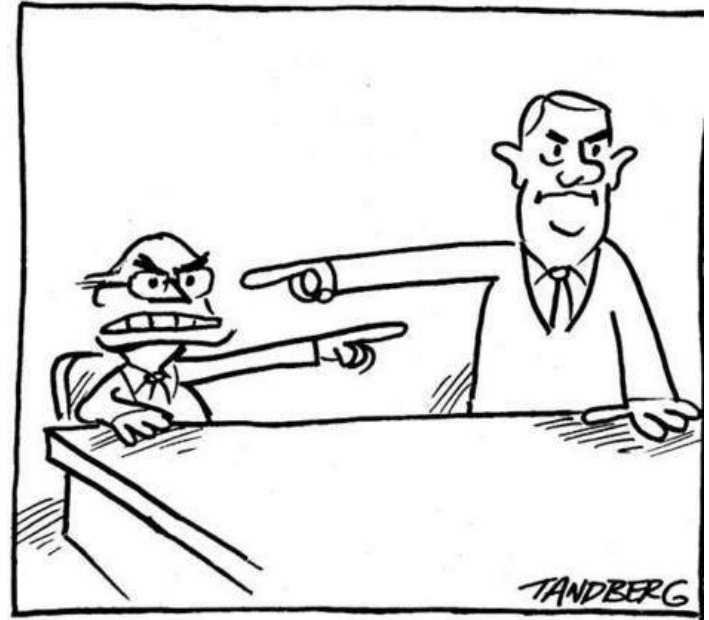


IT-HÖGSKOLAN

Här startar din IT-karriär.

Blame

- Take a look who did changes to the most recent version of the file?
- `git blame <filename>`



$\wedge \sim$

G H I J

\ / \ /

D E F

\ | / \

\ | / |

\ | / |

B C

\ /

\ /

A

$$A = A^0$$

$$B = A^1 = A^1 = A^{\sim 1}$$

$$C = A^2$$

$$D = A^{\wedge 1} = A^1^1 = A^{\sim 2}$$

$$E = B^2 = A^{\wedge 2}$$

$$F = B^3 = A^{\wedge 3}$$

$$G = A^{\wedge \wedge} = A^1^1^1 = A^{\sim 3}$$

$$H = D^2 = B^{\wedge 2} = A^{\wedge \wedge 2} = A^{\sim 2^2}$$

$$I = F^1 = B^3^1 = A^{\wedge 3^1}$$

$$J = F^2 = B^3^2 = A^{\wedge 3^2}$$

Both \sim and \wedge on their own refer to the parent of the commit. But they differ in meaning when they are used with numbers:

~ 2 means up two levels in the hierarchy, via the first parent if a commit has more than one parent

$\wedge 2$ means the second parent where a commit has more than one parent (i.e. because it's a merge)



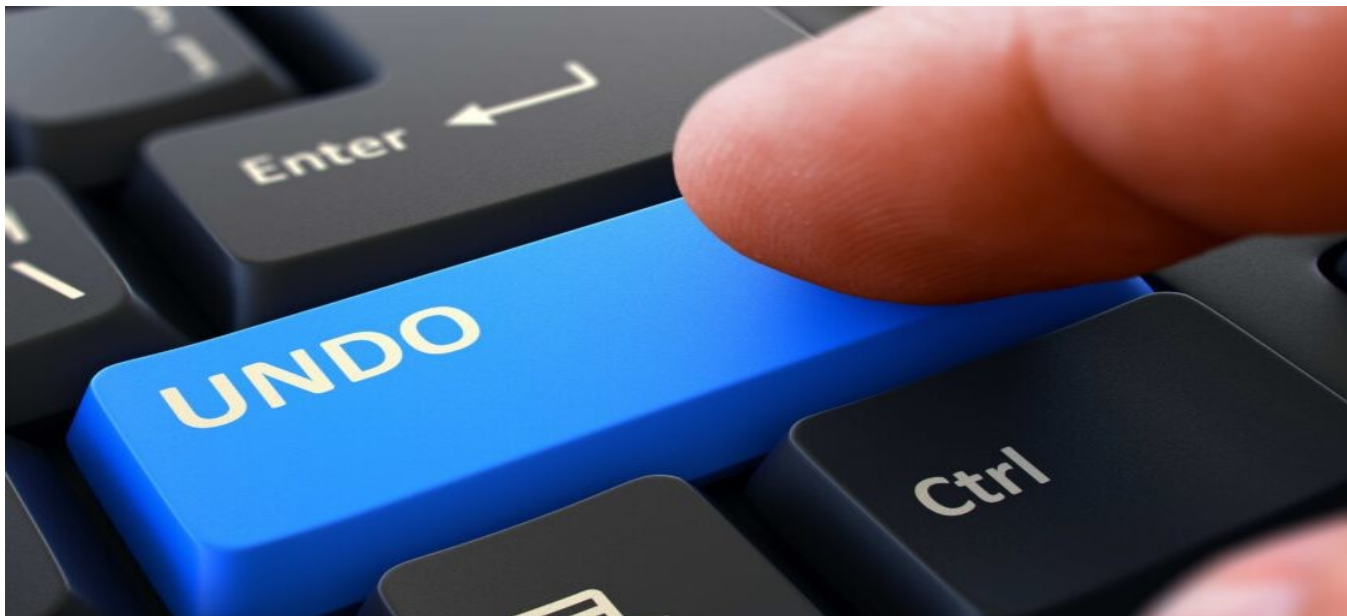
IT-HÖGSKOLAN

Här startar din IT-karriär.

git clean

- Remove untracked files from the working tree
- Normally, only files unknown to Git are removed, but if the -x option is specified, ignored files are also removed. This can, for example, be useful to remove all build products.
- <https://git-scm.com/docs/git-clean>





IT-HÖGSKOLAN

Här startar din IT-karriär.

Rewriting history

- Changing the Last commit (will replace with a new commit)
 - `git commit --amend`
- **Don't amend public commits!!!!**
- Changing multiple commits
 - `git rebase`
 - <https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase>
- Rebase is one of two Git utilities that specializes in integrating changes from one branch onto another.
- The other change integration utility is `git merge`.



Undoing Commits and Changes

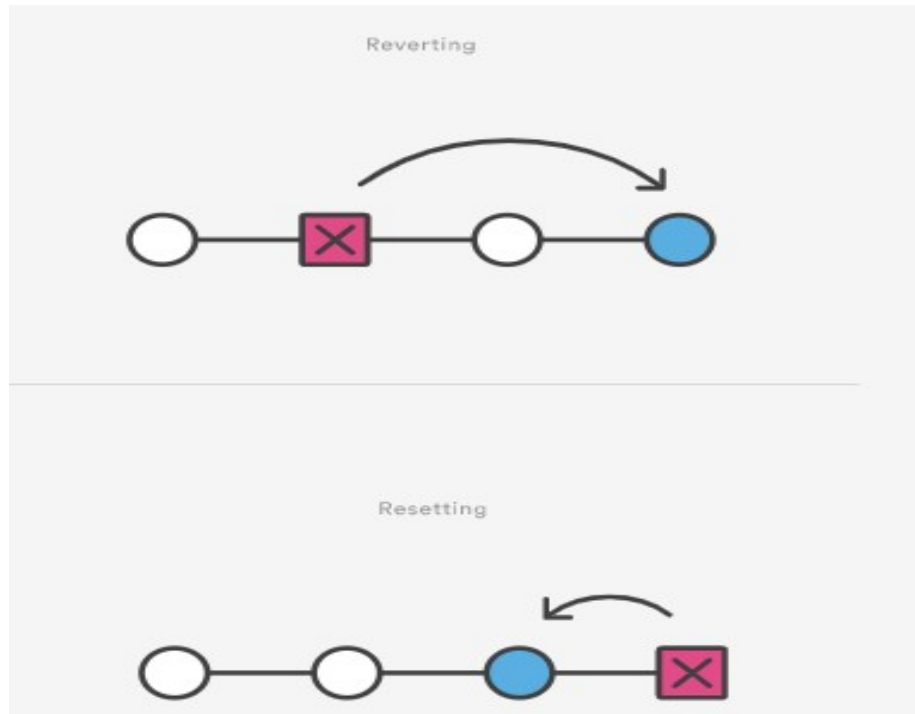
- <https://www.atlassian.com/git/tutorials/undoing-changes>
- `git checkout [file]`
- Checking out a specific commit will put the repo in a "detached HEAD" state. Instead use:

`git checkout -b new_branch_without_crazy_commit`

- Undo **public** commit, (pushed to a remote)
 - `git revert HEAD`
 - `git revert HEAD~`
 - Creates a **new commit** doing the opposite of the commit we are undoing.



Revert

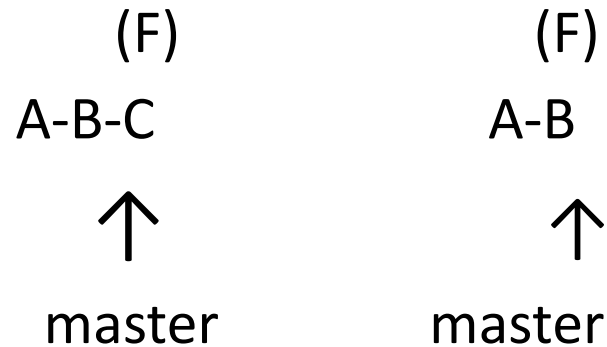


IT-HÖGSKOLAN

Här startar din IT-karriär.

git reset --hard

- Undo **local** commit.
- You want to nuke commit C and never see it again and lose all the changes in locally modified files.
- `git reset --hard`



Use --hard if you don't care about keeping the changes you made



IT-HÖGSKOLAN

Här startar din IT-karriär.

Redo the undo, everything is not lost, yet...

- Scenario: You made some commits, did a git reset --hard to “undo” those changes , and then realized: you want those changes back!
- Undo with: git reflog and git reset or git checkout

git reflog show main

git log --graph --decorate --oneline \$(git rev-list -g --all)

git reset HEAD@{1} or git reset [commithash]

- git reflog doesn't last forever. Git will periodically clean up objects which are “unreachable.” Don't expect to find months-old commits lying around in the reflog forever.



IT-HÖGSKOLAN

Här startar din IT-karriär.

git reset --mixed (default mode)

- Suppose commit C wasn't a disaster, but just a bit off. You want to undo the commit but keep your changes for a bit of editing before you do a better commit. No files are changed but the index is reset to B. All changes are unstaged.

git reset HEAD~1



Use --soft if you want to keep both file and index. Executing a git commit at this point will create a new commit with the same changes as C



Unstaging or undoing file changes

- Unstaging a Staged File
- `git reset HEAD <file>...` to unstage
- Unmodifying a Modified File
- `git checkout -- <file>`

It's important to understand that `git checkout -- <file>` is a dangerous command. Any local changes you made to that file are gone — Git just replaced that file with the last staged or committed version. Don't ever use this command unless you absolutely know that you don't want those unsaved local changes.



git restore

- New command since Git 2.23.0
- `git restore --staged <file>...` to unstage
- `git restore <file>...` to discard changes in working directory

It's important to understand that `git restore <file>` is a dangerous command. Any local changes you made to that file are gone — Git just replaced that file with the last staged or committed version. Don't ever use this command unless you absolutely know that you don't want those unsaved local changes.



Git Reset vs Revert vs Checkout reference

Command	Scope	Common use cases
<code>git reset</code>	Commit-level	Discard commits in a private branch or throw away uncommitted changes
<code>git reset</code>	File-level	Unstage a file
<code>git checkout</code>	Commit-level	Switch between branches or inspect old snapshots
<code>git checkout</code>	File-level	Discard changes in the working directory
<code>git revert</code>	Commit-level	Undo commits in a public branch
<code>git revert</code>	File-level	(N/A)



Branches

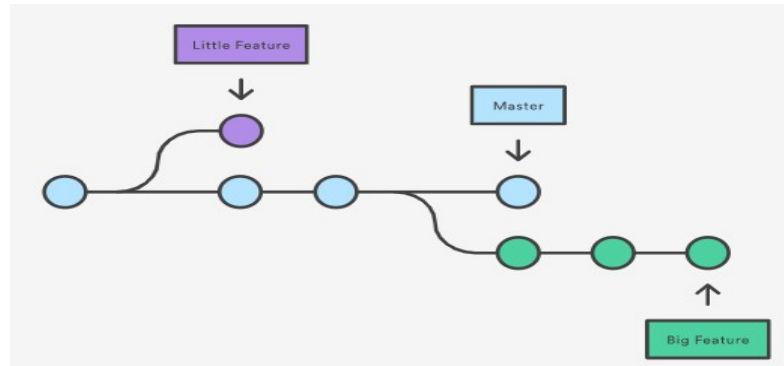


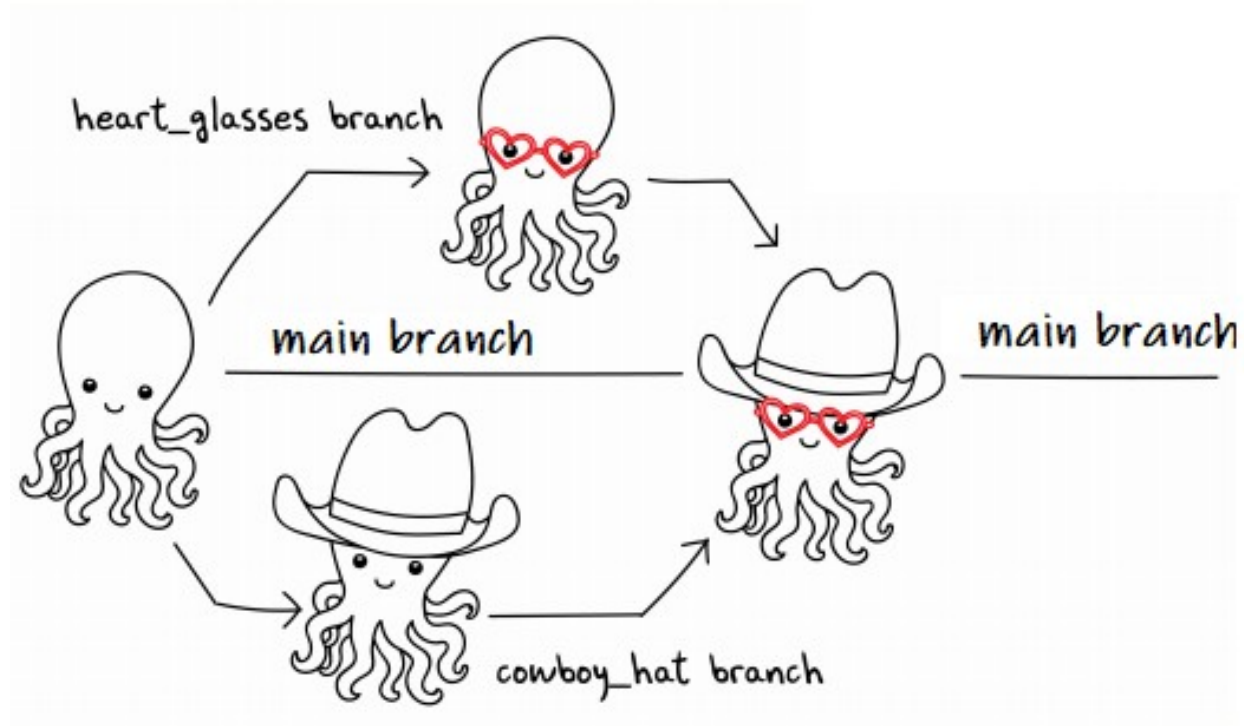
IT-HÖGSKOLAN

Här startar din IT-karriär.

Git Branch

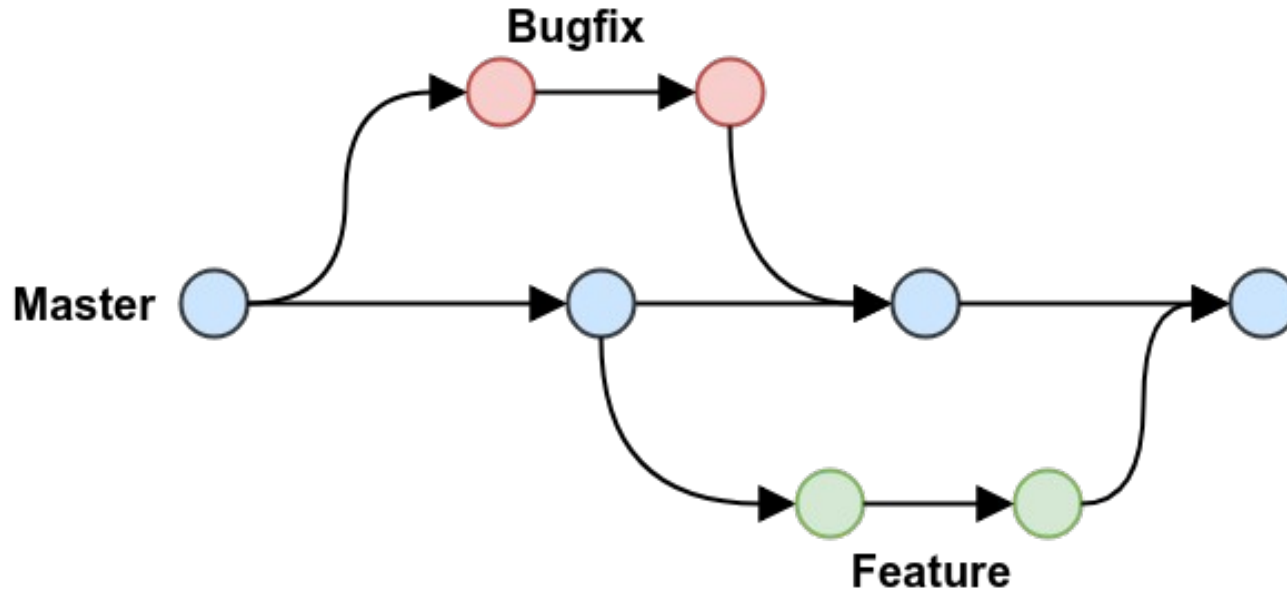
- Git branches are effectively a pointer to a snapshot of your changes.
- Lets us work on different versions of a file in parallel.
- In this sense, a branch represents the tip of a series of commits—it's not a container for commits.
- **git branch**





IT-HÖGSKOLAN

Här startar din IT-karriär.



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

```
git config --global alias.graph 'log --all --decorate --oneline --graph'
```

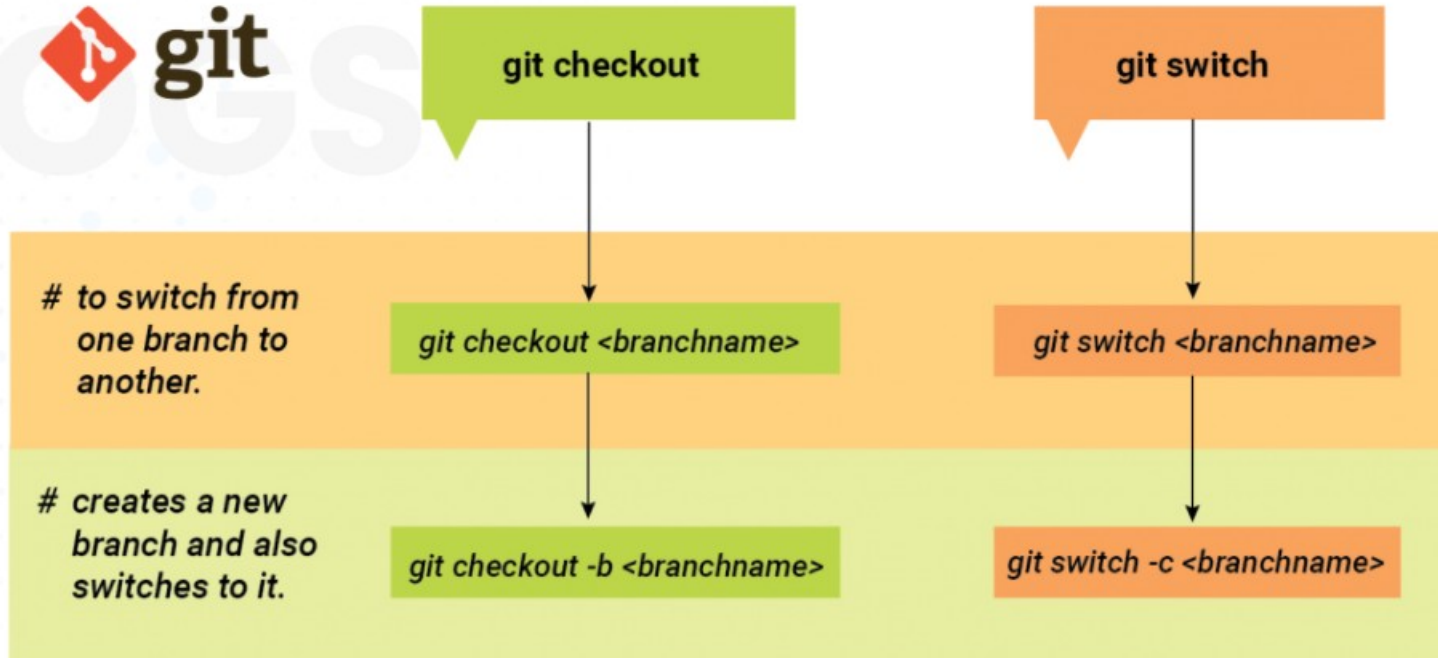


IT-HÖGSKOLAN

Här startar din IT-karriär.

git checkout vs git switch

git switch [branch_name]



<https://bluecast.tech/blog/git-switch-branch/>



IT-HÖGSKOLAN

Här startar din IT-karriär.

Why both checkout and switch?

- The new 'experimental' git switch branch command (git > 2.23) is meant to provide a better interface by having a clear separation, which helps to alleviate the developers' confusion when using git checkout.
- Git checkout can be used to switch branches and also to restore the working tree files.
- `git checkout <filename>` reverses the modifications of an unstaged file.
- <https://stackoverflow.com/questions/3639342/whats-the-difference-between-git-reset-and-git-checkout>
- <https://stackoverflow.com/questions/48508799/how-to-reset-all-files-from-working-directory-but-not-from-staging-area/57066072#57066072>



Can we switch branch with uncommitted changes?

- We can store our changes for later retrieval and restore an unmodified working tree.
- <https://git-scm.com/docs/git-stash>
- Use **git stash push** when you want to record the current state of the working directory and the index, but want to go back to a clean working directory. The command saves your local modifications away and reverts the working directory to match the HEAD commit.
- **git stash pop** or **git stash apply**

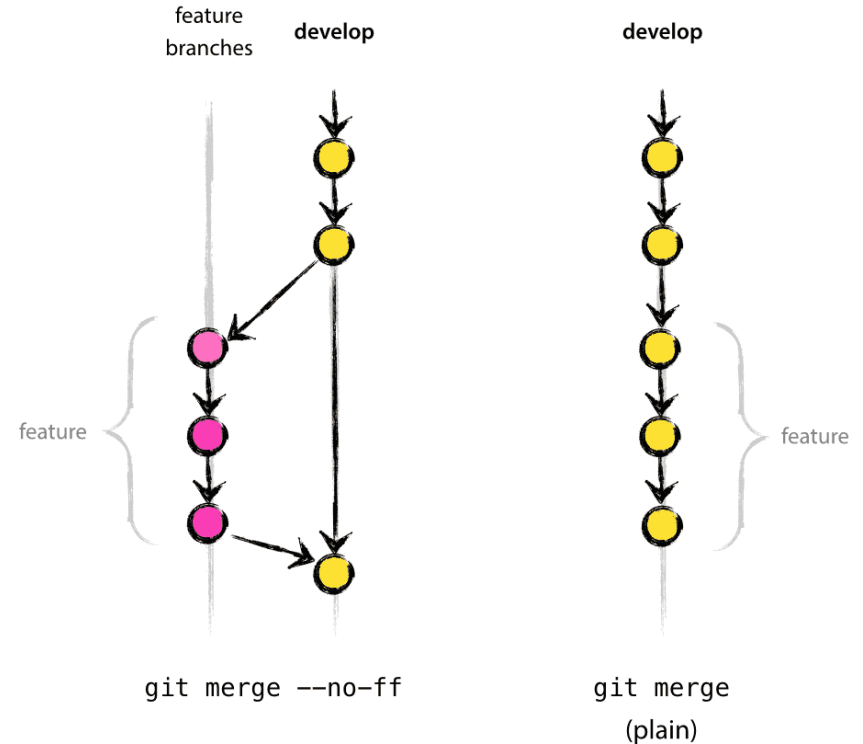


Merge commit

- <https://git-scm.com/docs/git-merge>
- **git switch develop**
- **git merge feature**
- Join two or more development histories together

```
commit 41ea02cc8a1d2964c4f7b46b5f6b11cc04327959
Merge: a8d1421 e71dbf5
Author: Lorna Mitchell <lorna@lornajane.net>
Date: Mon Dec 22 19:48:34 2014 +0000
```

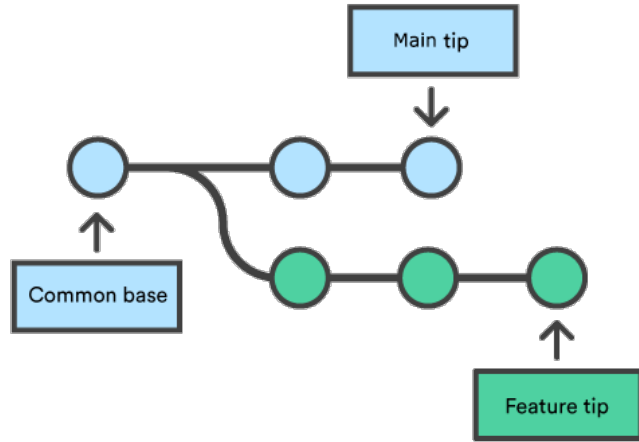
```
Merge branch 'akrobat-update-homepage-with-open-cfps'
```



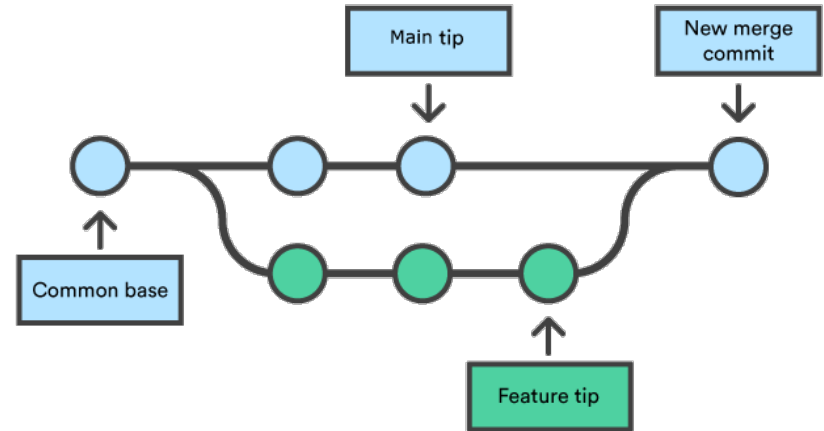
Try a merge without changing anything to see if we have conflicts

- `git merge --no-commit --no-ff`
- No-commit prevents the creation of an actual merge commit.
- Prevents fast forwarding because that doesn't create a commit and would move the pointer.
- We can create an alias:
`git config --global alias.tm "commit --no-commit --no-ff"`
- Use with: **`git tm branch name`**



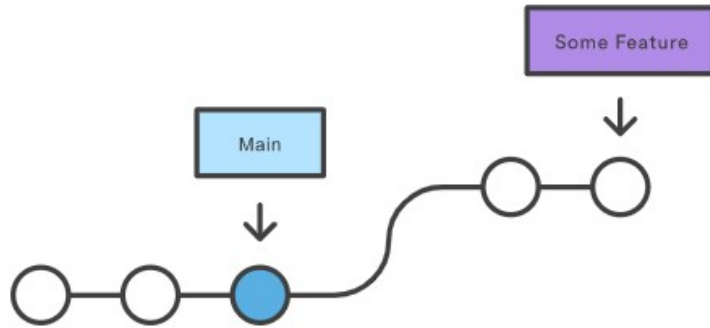


Execute `git status` to ensure that HEAD is pointing to the correct merge-receiving branch.

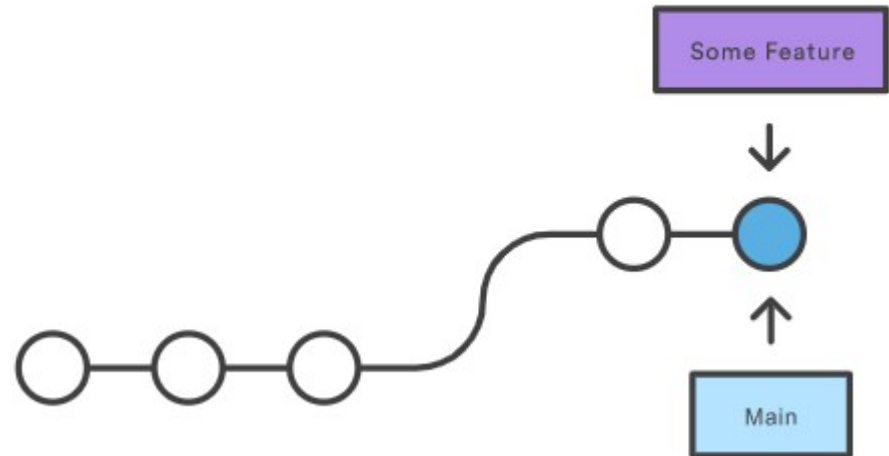


Fast forward merge

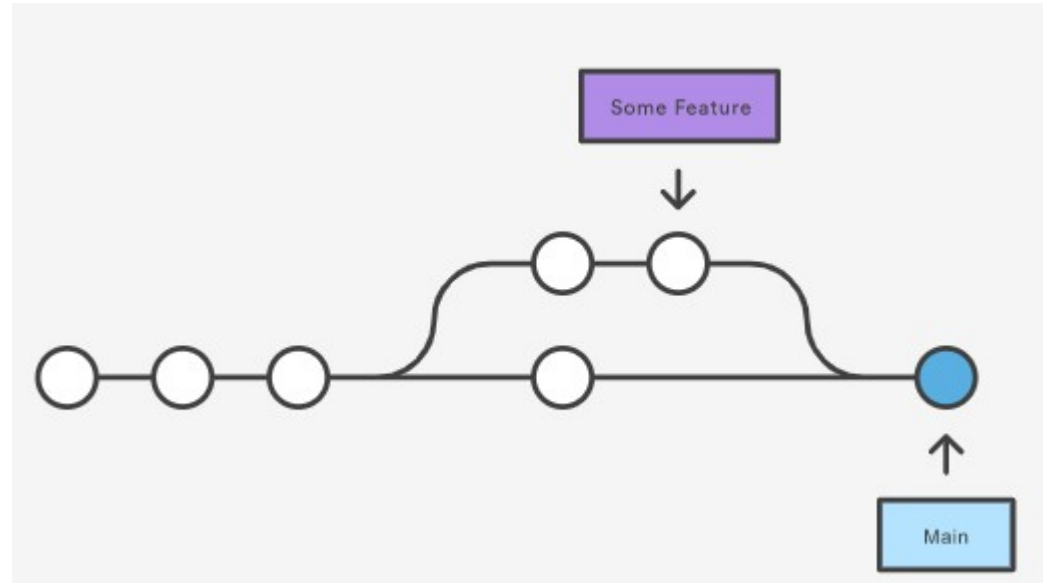
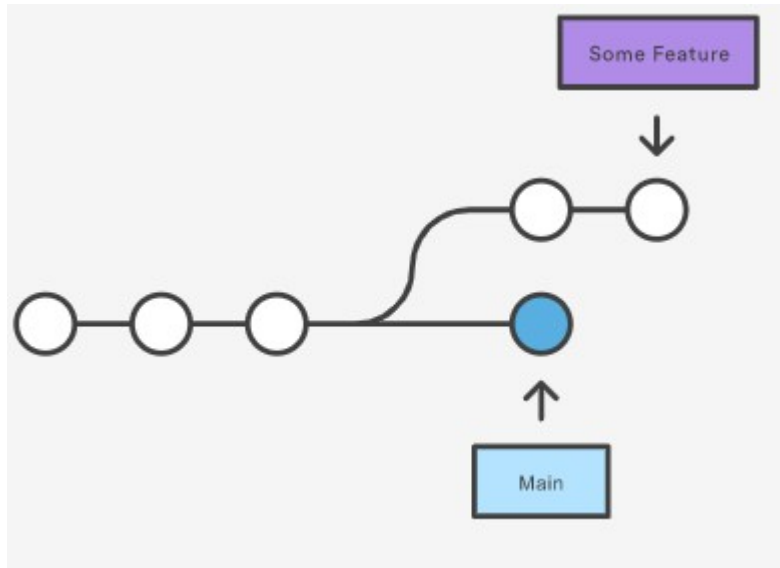
Before Merging



After a Fast-Forward Merge



Not possible with fast forward merge, diverted branches

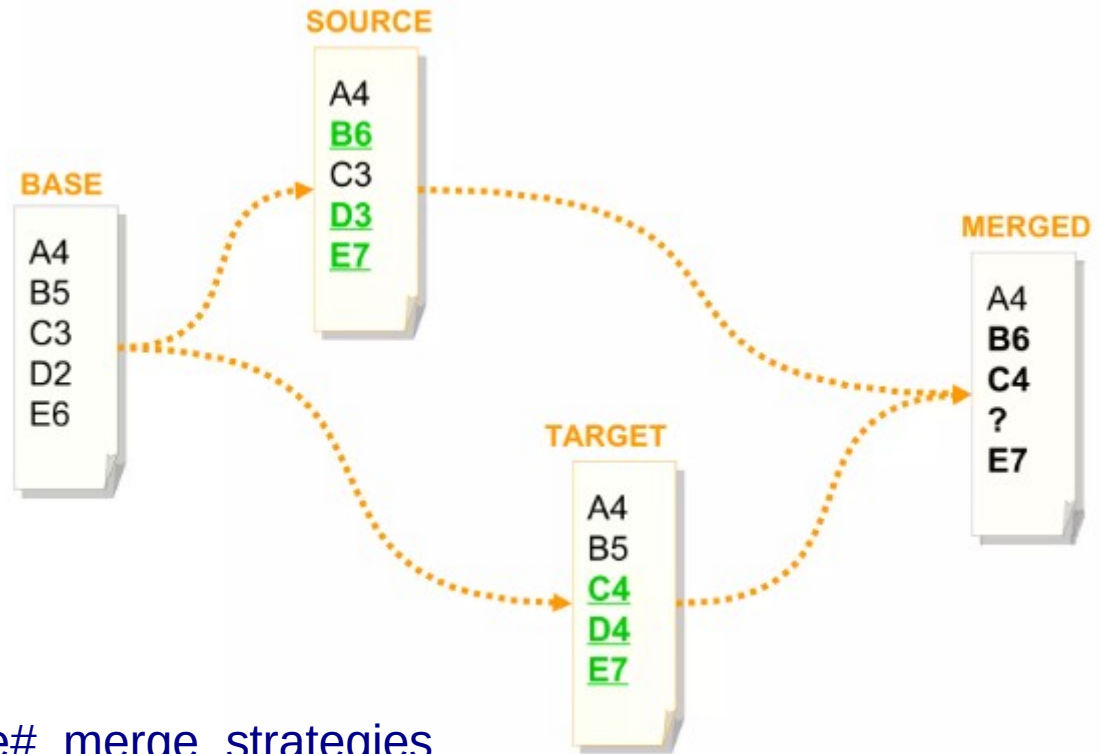


3-Way merge

MERGE STRATEGIES

Only a 3-way merge gives you the ability to know whether or not a chunk is a change from the origin and whether or not changes conflict.

2-way merge only compares source vs target.



https://git-scm.com/docs/git-merge#_merge_strategies

<https://stackoverflow.com/questions/18131515/how-can-i-see-a-three-way-diff-for-a-git-merge-conflict>

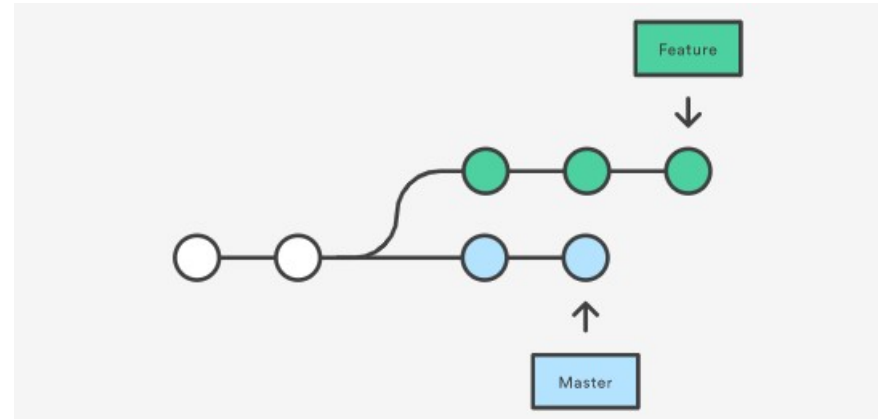


IT-HÖGSKOLAN

Här startar din IT-karriär.

merging-vs-rebasing

- Let's say that the new commits in master are relevant to the feature that you're working on. To incorporate the new commits into your feature branch, you have two options: merging or rebasing.

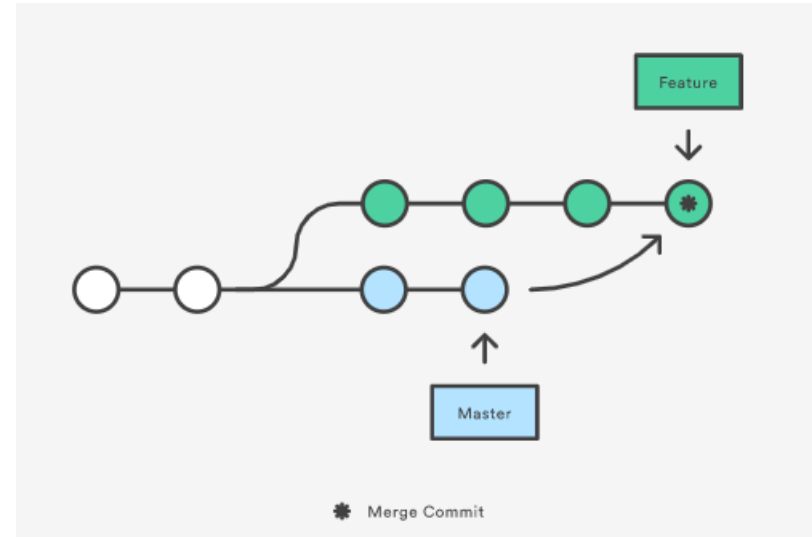


The merge option

- **git checkout feature**
- **git merge master**

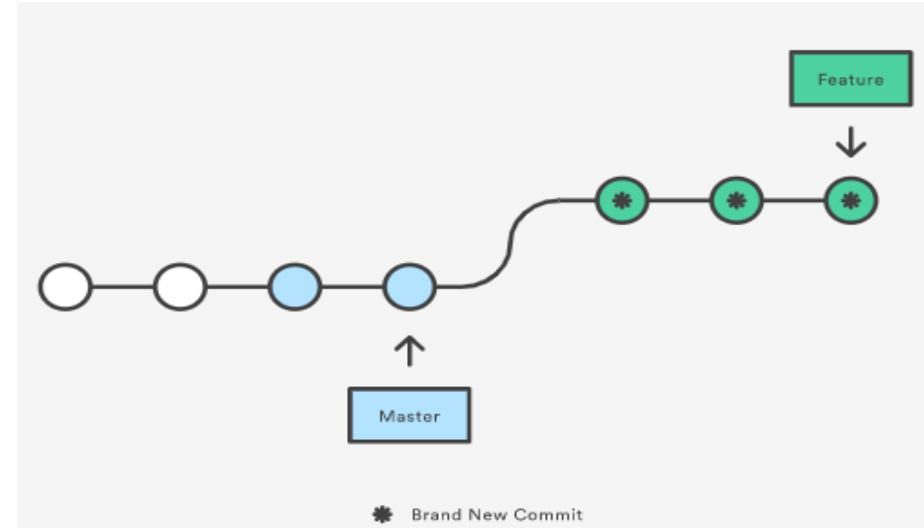
Or:

- **git merge feature master**



The rebase option

- **git checkout feature**
- **git rebase master**
- This moves the entire feature branch to begin on the tip of the master branch, effectively incorporating all of the new commits in master. But, instead of using a merge commit, rebasing rewrites the project history by creating brand new commits for each commit in the original branch.



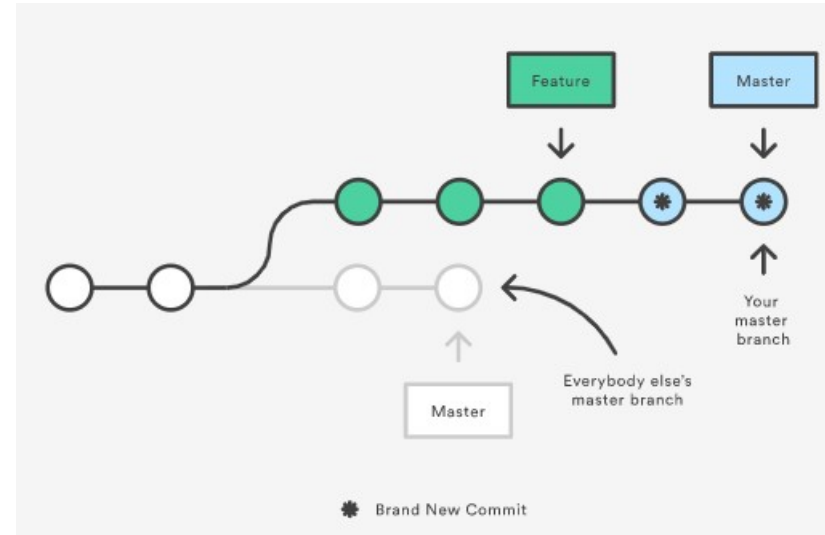
+/- for rebase

- The major benefit of rebasing is that you get a much cleaner project history.
- Eliminates the unnecessary merge commits required by git merge. Rebasing also results in a perfectly linear project history—you can follow the tip of feature all the way to the beginning of the project without any forks.
- Interactive Rebasing lets us rewrite the history **git rebase -i**
- The **Golden Rule of Rebasing** must be followed.



The Golden Rule of Rebasing

- Think about what would happen if you rebased master onto your feature branch
- Since rebasing results in brand new commits, Git will think that your master branch's history has diverged from everybody else's.
- **The golden rule of git rebase is to never use it on public branches.**
- <https://www.atlassian.com/git/tutorials/advanced-overview>



Cherry Picking

- <https://www.atlassian.com/git/tutorials/cherry-pick>
- a - b - c - d Main
 \
 e - f - g Feature
- git cherry-pick f
- a - b - c - d - f Main
 \
 e - f - g Feature



When to use cherry-pick?

- You fixed a critical bug on a feature branch that you'd like to get into your release version asap. Or the other way around.
<https://www.bmc.com/blogs/patch-hotfix-coldfix-bugfix/>
- You accidentally commit to wrong branch. Use cherry pick to copy commit to the right branch.
- Don't do the same changes in different branches. Do it once and then copy the commit.
- Prefer merge if that works instead.
- cherry-picking commits will create a fresh commit with a new hash in the other branch, so don't be confused if you see a different commit hash.



- Count your commits

```
git rev-list --count <branchname>
```

- View a file of another branch

```
git show main:README.md
```



IT-HÖGSKOLAN

Här startar din IT-karriär.

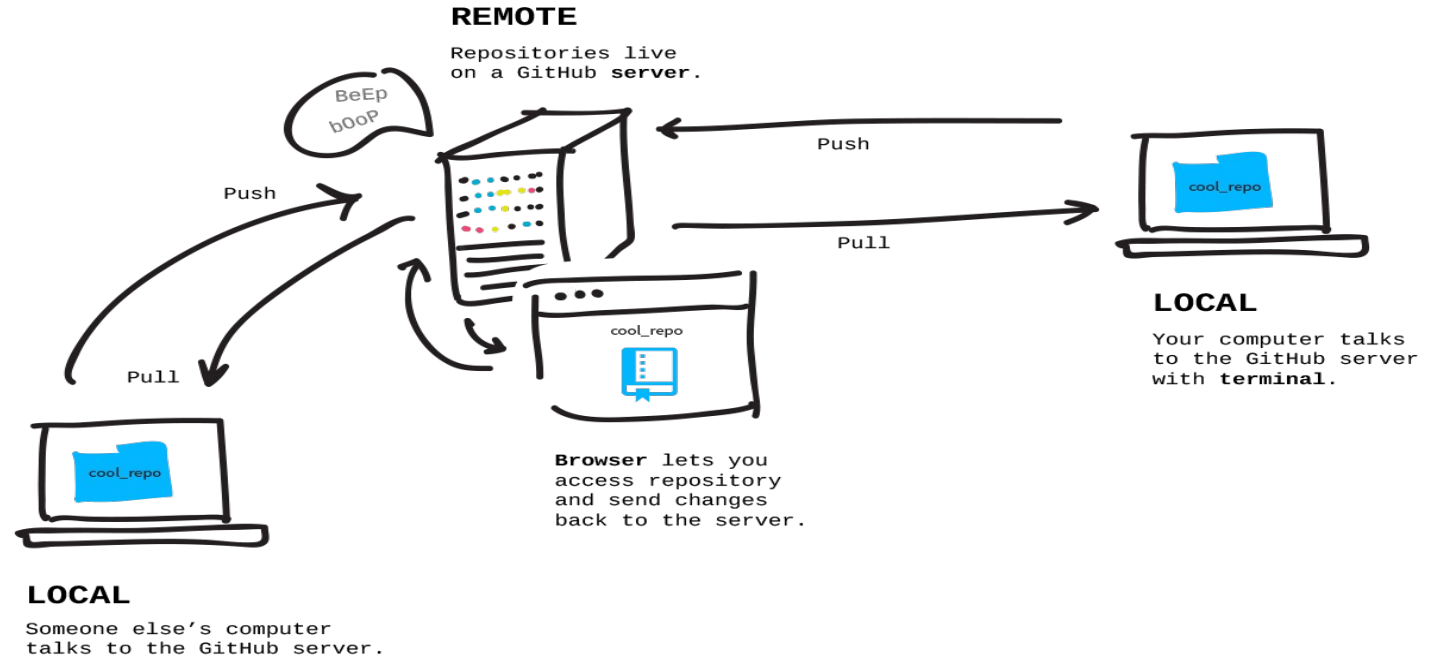
remote/fetch/push/pull

- The **git remote** command lets you create, view, and delete connections to other repositories.
- The **git fetch** command downloads commits, files, and refs from a remote repository into your local repo. Fetching is what you do when you want to see what everybody else has been working on.
- The **git push** command is used to upload local repository content to a remote repository.
 - Pushing is how you transfer commits from your local repository to a remote repo.
 - It's the counterpart to git fetch, but whereas fetching imports commits to local branches, pushing exports commits to remote branches.
- The **git pull** command is used to fetch and download content from a remote repository and immediately update the local repository to match that content.
 - The git pull command is actually a combination of two other commands, git fetch followed by git merge.



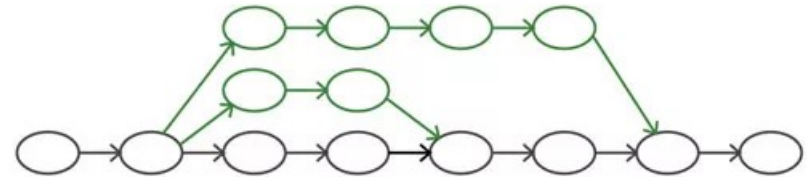
Git remote

- <https://www.atlassian.com/git/tutorials/syncing>



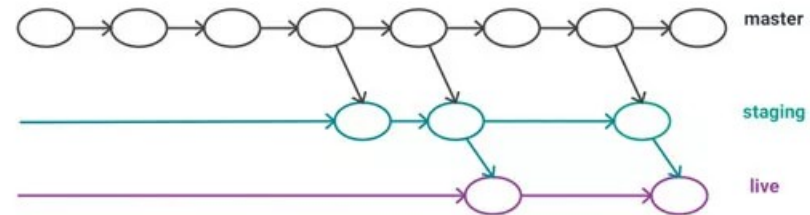
Branching strategies

- GitHub Flow simply states that when working on any feature or bug fix, you should create a branch.
- When it is finished, it is merged back into the master with a merge commit. It's a very easy way for teams and individuals to create and collaborate on separate features safely, and to be able to deliver them when they are done.
- The rule is that the master branch is always deployable, so features must be finished and thoroughly tested before they are merged in.



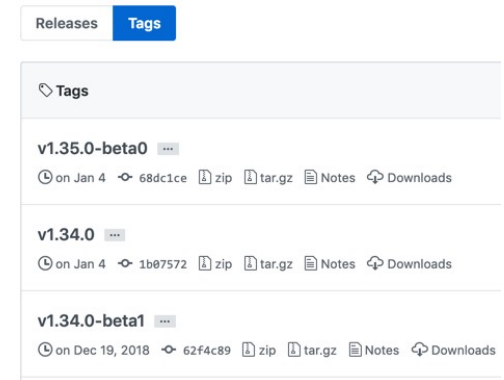
Branching strategies

- 'Branch per Platform' model. In addition to a master branch, there are branches that track the live platform, plus any other platforms you use in your workflow (such as test, staging and integration).
- The master remains at the bleeding edge of your project, but as features complete, you merge to the other platforms accordingly



Tag releases

- Use tags to mark your releases when you make them.
- This way, you can look back at the history of the versions that have been live. In fact, GitHub even has a dedicated page for this. This technique is used especially to mark library releases.
- <https://docs.github.com/en/github/administering-a-repository/releasing-projects-on-github>



IT-HÖGSKOLAN

Här startar din IT-karriär.

Branching strategies

- Git Flow
- More complicated than the easier GitHub Flow.
- <https://nvie.com/posts/a-successful-git-branching-model/>



IT-HÖGSKOLAN

Här startar din IT-karriär.

GitHub pull requests

- Pull requests let you tell others about changes you've pushed to a branch in a repository on GitHub.

Once a pull request is opened, you can discuss and review the potential changes with collaborators and add follow-up commits before your changes are merged into the base branch.

- Perfect for open source projects, except Linux :P
- <https://docs.github.com/en/github/collaborating-with-issues-and-pull-requests/about-pull-requests>



Renaming/Deleting Branches

- <https://www.freecodecamp.org/news/how-to-delete-a-git-branch-both-locally-and-remotely/>
- <https://phoenixnap.com/kb/how-to-rename-git-branch-local-remote>



IT-HÖGSKOLAN

Här startar din IT-karriär.

Git Submodule

- A Git submodule is a separate repository within a repository.
- `git submodule add <url to project to start tracking>`
- Submodules will add the subproject into a directory named the same as the repository.
- `.gitmodules` file. This is a configuration file that stores the mapping between the project's URL and the local subdirectory.
- Version controlled with your other files.
- <https://git-scm.com/book/en/v2/Git-Tools-Submodules>



- To clone a repository with submodules, use:
`git clone --recursive <URL to Git repo>`
- If you've previously cloned a repository and wish to load its submodules, use:
`git submodule update --init`
- If there are nested submodules, do the following:
`git submodule update --init --recursive`
- Specify a branch for a submodule using:
`git submodule set-branch -branch <branch name> -- <submodule path>`
- Or change branch using:
`git submodule change branch`



Improvement, Git subtree

- Consider your Git repository to be a tree, and a subtree is a smaller version of the main tree.
- A subtree can be added to a parent repository.

```
git remote add remote-name <URL to Git repo>
```

```
git subtree add --prefix=folder/ remote-name <URL to Git repo>  
subtree-branch name
```

- Changes to and from the subtree are pushed and pulled using:

```
git subtree push-all
```

```
git subtree pull-all
```



Submodule or Subtree?

- Cloning repositories, which contain submodules, requires downloading the submodules separately.
- The submodule folders will be empty after cloning if the source repository is moved or becomes unavailable.
- git subtree does not add new metadata files like git submodule does (i.e., .gitmodule).
- A new merging approach must be learned with subtree
- It's a little more difficult to contribute code for the sub-projects upstream when using subtree.
- You must be sure that super and sub-project code is not mixed in new commits.
- <https://www.atlassian.com/git/tutorials/git-subtree>



Hooks

- Hooks reside in the `.git/hooks` directory of every Git repository.
 - `pre-commit`
 - `prepare-commit-msg`
 - `commit-msg`
 - `post-commit`
 - `post-checkout`
 - `pre-rebase`
- <https://www.atlassian.com/git/tutorials/git-hooks>



Learn git with visualization

- Visualization of git commands
- <https://git-school.github.io/visualizing-git/>
- Learn Git Branching
- <https://learngitbranching.js.org/>
- <https://github.com/benthayer/git-gud>



Git Katas

- What makes a good practice session?
- You need time without interruptions, and a simple thing you want to try.
- You need to try it as many times as it takes, and be comfortable making mistakes.
- You'll recognize a good practice session because you'll come out of it knowing more than when you went in.
- <https://github.com/eficode-academy/git-katas>

