How to Git

An introduction to Git.

https://github.com/3nol/how-to-git

Overview

Git is a version control system (VCS).

What is a VCS?

Version control systems are tools used to track changes in source code.

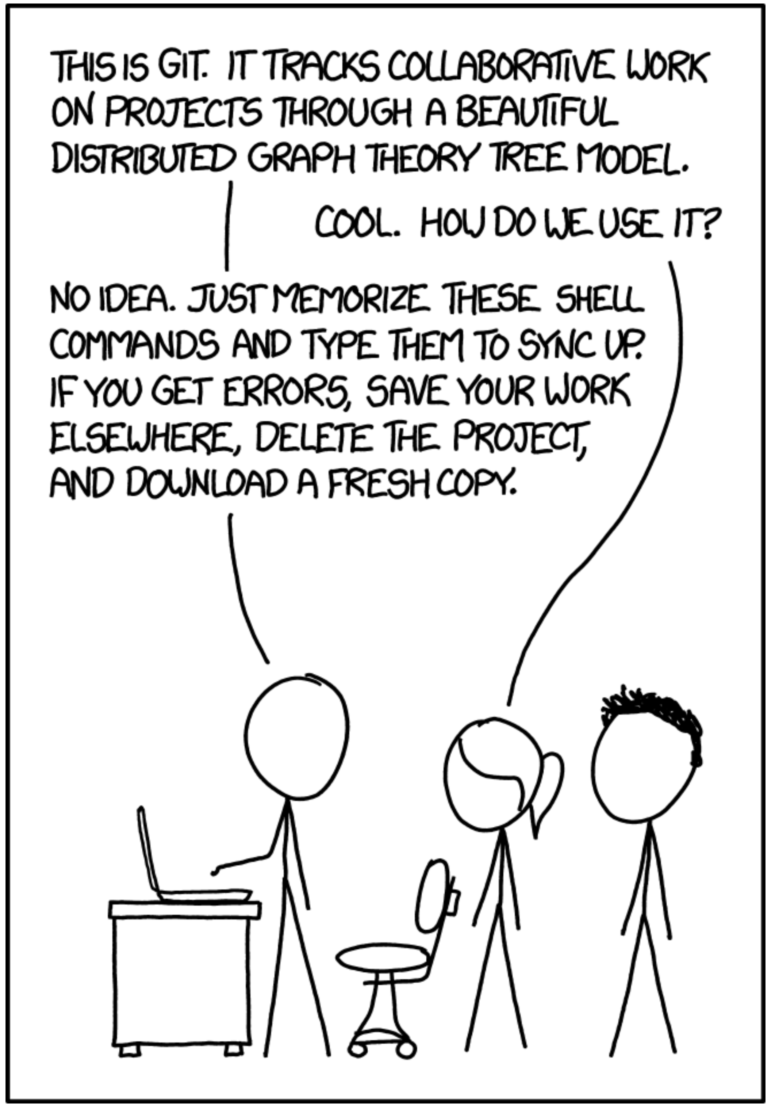
- Series of snapshots, each encapsulating the entire state of a working directory.
- Stored metadata like who created each snapshot, messages, and so on.

Why is Version Control useful?

To look at old snapshots of a project, keep a log of why certain changes were made, work on parallel branches, and much more.

- Who wrote this module?
- When was this particular line of this particular file edited?
- Over the last 1000 revisions, why did a particular unit test stop working?
- Who is to blame for that nasty bug?

While other VCSs exist, **Git** is the de facto standard for version control. However, working with Git can be frustrating at times.


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.

How does Git do versioning?

(1) Snapshots

Git models the history of a collection of files and folders with a series of snapshots. In Git terminology, a file is called a *blob*, and it's just a bunch of bytes. A directory is called a *tree*, and it contains other blobs or trees.

```
<root> (tree)
|
+- foo (tree)
   |
   + bar.txt (blob)
```

(2) References

All snapshots are identified by their *hash codes*. To make this identification more human-readable, special *references* are used. References are mutable pointers to immutable commit hashes.

Special references:

- `master`: latest snapshot in the main branch of development.
- `HEAD`: current position in snapshot history.

(3) Repositories

A Git *repository* is the data objects and references. Git stores the objects and references on the disk in the hidden `.git/` directory.

- Essentially a directory bundling all files.
- Can be local or remote (see later).

Commits

Git calls snapshots *commits*. Visualizing a commit history might look something like this:

```
o <-- o <-- o <-- o
            ^
             \
              --- o <-- o
```

Structure of a Commit

```
# a file is a bunch of bytes
blob: list[bytes] = ...

# a directory contains named files and directories
tree: dict[str, blob | tree] = ...

# a commit has parents, metadata, and the top-level tree
commit = {
    parents: list[commit]
    author: str
    message: str
    snapshot: tree
}
```

Commit History

A merge commit shown in **bold**.

```
o <-- o <-- o <-- o <---- o
            ^            /
             \          v
              --- o <-- o
```

Commits in Git are immutable.

This doesn't mean that mistakes can't be corrected. However, edits to the commit history are actually creating entirely new commits, and references are updated.

Creating Commits

The following command is used to create a commit with a specific commit message:

- `git commit -m "<message>"`

It is important to write *good* commit messages.

Removing Commits

Multiple commands exist around the notion of removing or reverting changes in a commit.

- `git reset <commit-hash> -- <path>`
  Resets a file to another commit hash.
- `git revert <commit-hash>`
  Undoes all changes that have been introduced from the given commit.
- `git commit --amend`
  Edits the last commit's contents and message.

Staging

We want clean snapshots, and it might not always be ideal to make a snapshot from the *entire* current state. Git allows you to specify which modifications should be included into commits.

This partial inclusion is handled by the *staging area*.

- Changes can be added to and removed from the area.
- Only changes that are staged are accepted for commits.

Adding to the Staging Area

The following command is used push files into the staging area:

- `git add <path>`

You can specify one or multiple files and even use wildcards to add multiple files at once. For example, when doing Java development, you can use `git add "*.java"`.

Removing from the Staging Area

Use this command for removing files from the staging area:

- `git restore --staged <path>`

This does not change the content at the path, but only unstages it.

Branches

Here, we will look at parallel versioning. Branches model these parallel version timelines for each developer.

Branching allows you to fork version history. Branches are completely independent of each another and only "interact" when merging them.

```
git branch --show-current
```

Splitting into Branches

The following commands can be used for creating a branch:

- `git branch [-a]`
  Lists all existing branches. Specifying `-a` shows all branches, even on remote hosts (see later).
- `git checkout -b <branch-name>`
  Creates a new branch and switches to it.
- `git checkout <branch-name>`
  Switches to an existing branch.

Re-combining Branches

To following commands allow you to re-combine existing branches.

- `git merge <branch-name>`
  Merges a specified branch into the current one using merging strategy, and prompts you if merge conflicts occur.
- `git rebase <branch-name>`
  Instead of letting both changes "collide", rebasing moves the changes on the current branch *on top* of the other changes.
- `git branch -d <branch-name>`
  Deletes a branch.

Merge conflicts

When merging branches, conflicts can occur. Git does its best effort in combining the changes. But in the scenario of changing a certain file in *both* branches, Git does not decide for you which change to prioritize. In this case, you have to act.

You might get a message like this one:

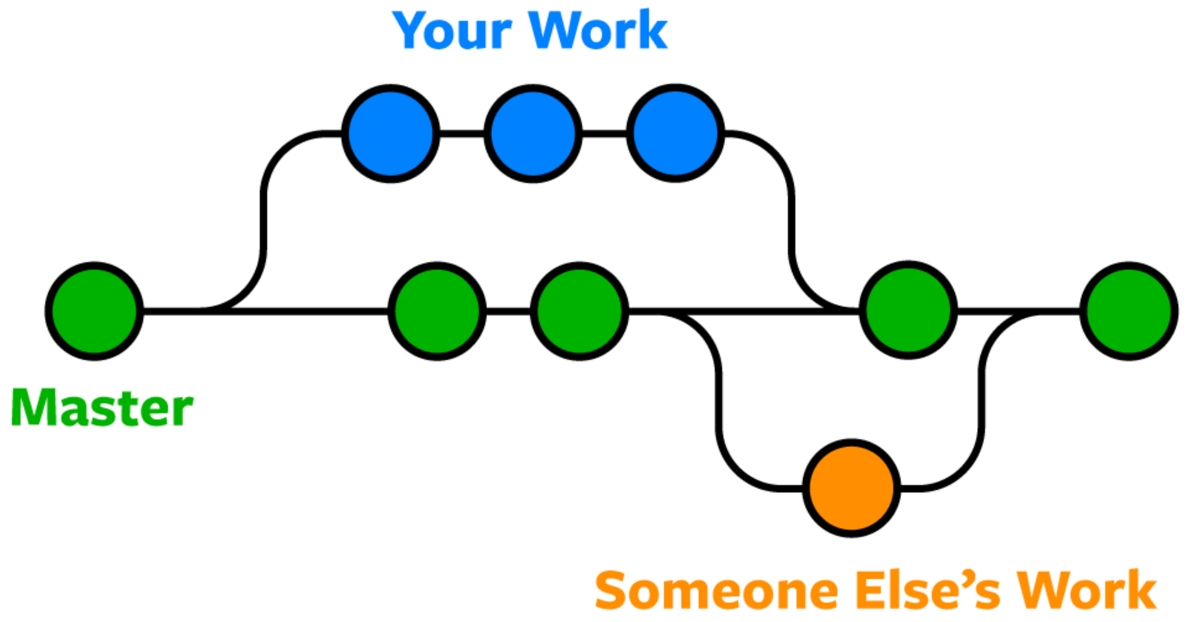
```
Auto-merging HelloWorld.java
CONFLICT (content): Merge conflict in HelloWorld.java
Automatic merge failed; fix conflicts and then commit the result.
```

Remotes

How to collaborate with others using these concepts (branching, merging, ...) while Git only runs on my machine?

Remotes allow you to sync your work and their versions to a server. Git has inbuilt support for uploading and downloading the latest updates.

You can use Git commands for interacting with remotes. The main way of interaction is to fetch, merge, modify, and push changes between *branches*. However, this time, the branches are not living locally on your machine, but are hosted remotely.



Interaction with Remotes

- `git clone <remote-source>`
  Downloads a repository from a remote host, via HTTPS or SSH.
- `git remote`
  Lists all added remotes. If you have `git clone`'d the repository, then its source will be listed here already.
- `git fetch`
  Retrieves all new updates on objects/references from a remote.

CLI

- `git pull`
  Is the same as `git fetch; git merge` where the remote updates are fetched and directly included in your local copy.
- `git branch -u <remote-name>/<branch-name>`
  Sets up the correspondence between local and remote branch.
- `git push`
  Sends the latest changes to remote. For example, if you set up the correspondence, and made a new commit, pushing will upload this new commit to the remote host.

Examples of Remote Hosts

Remote hosts are providers of remote repositories.

- **GitHub**: Git is not GitHub.
- **GitLab**: There are other hosts, like GitLab, which mostly used in research.
- **BitBucket**: A host, that is mostly used in the corporate world, is BitBucket.

Exercises

Installation

1. Go to https://git-scm.com/download/ and download Git for your OS.
2. Complete the installation procedure.
3. Verify that Git is working by running the following command:

```
git --version
```

Alternatively, you can use your package manager of choice. For example: `sudo apt-get install git`

Getting Started I

- `git init`
  Creates a new repository in the current directory, with data stored in the (hidden) `.git/` directory.
- `git status`
  Tells you what's going on.

Getting Started II

- `git log`

  Shows a flattened log of history of commits, incl. their messages.
- `git diff <reference> <path>`

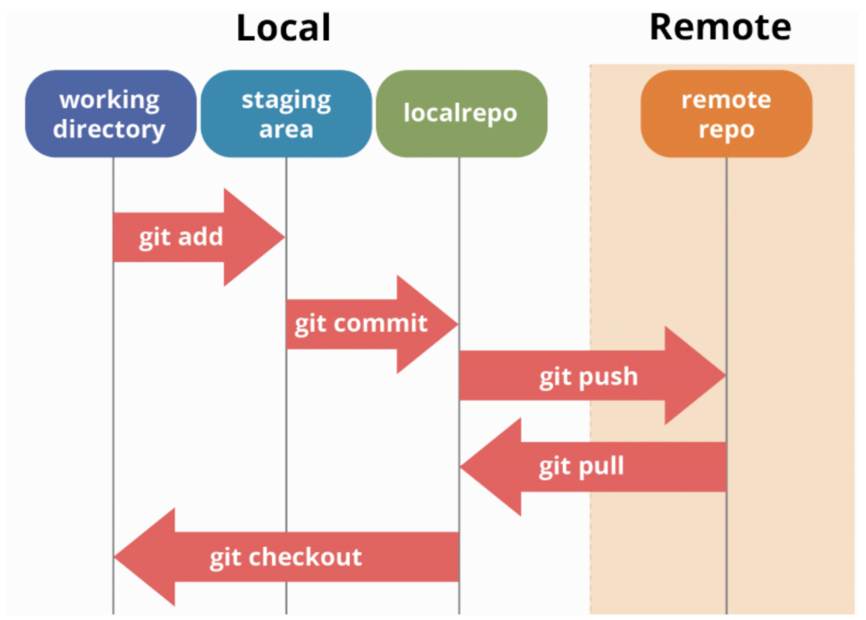
  Shows differences in a file path between snapshots.

Exercise 1

Create a local Git repository, create a simple file, and save it in a commit. Use the commit message "First commit".

Commands you may need:

> `init`, `add`, `commit`, `status`



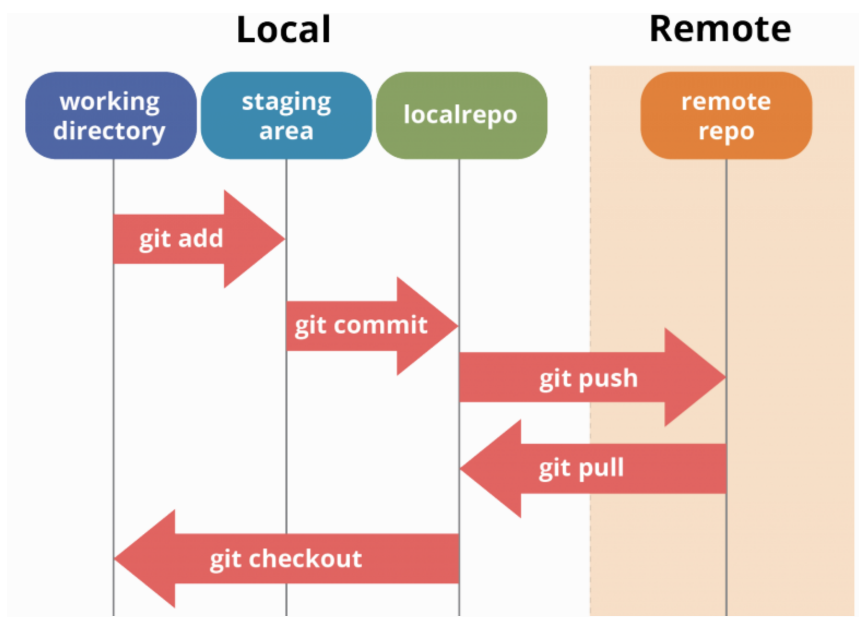
Helpful information for every command can be displayed using `git help <command>`.

Exercise 2

In your newly created repository, switch to a new branch. Then, modify your file and commit the changes.

Commands you may need:

> `checkout`, `add`, `commit`



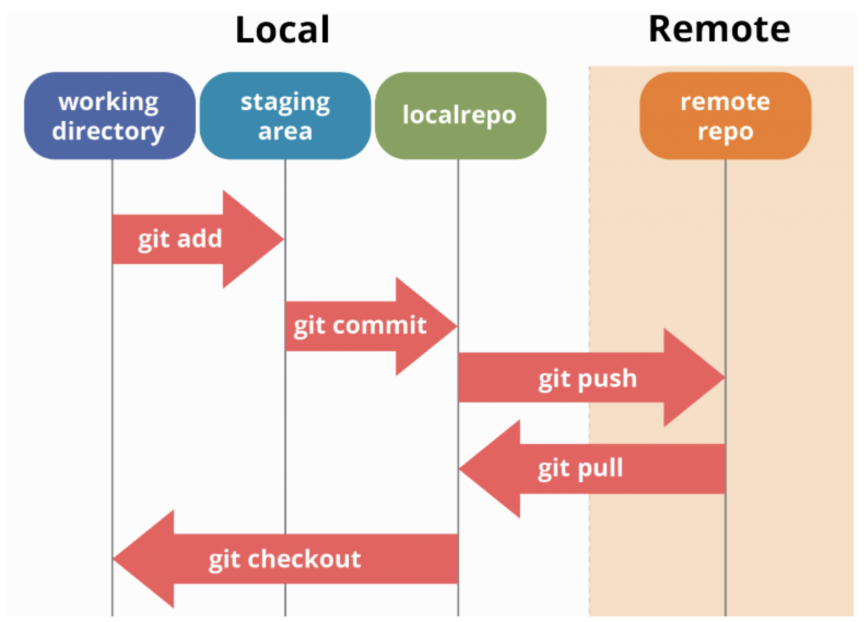
Helpful information for every command can be displayed using `git help <command>`.

Exercise 3

Switch back to the original branch (called `master`). Check the file differences to the other branch. Then, merge your newly created branch into the `master` branch.

Commands you may need:

> `checkout`, `diff`, `merge`



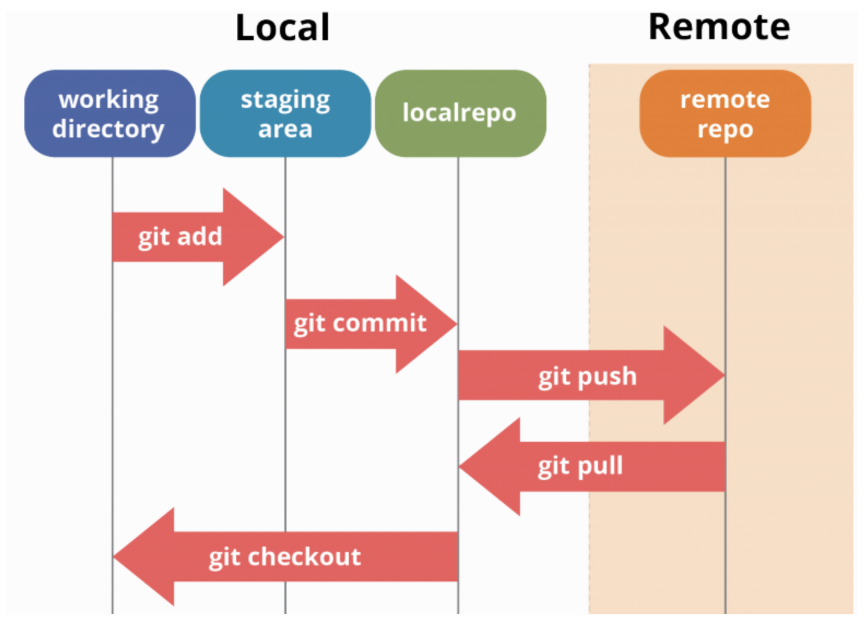
Helpful information for every command can be displayed using `git help <command>`.

Exercise 4

Clone the repository of this presentation from https://github.com/3nol/how-to-git and find this presentation file in one of the branches. Look at the history of commits that have been made.

Commands you may need:

> `clone`, `checkout`, `log`



Helpful information for every command can be displayed using `git help <command>`.

Survival Guide

0. Make changes.
1. `git add <path>`
   Add the path of the changed files.
2. `git commit -m "<message>"`
   Include all added files into a commit, attach a *good* message.
3. `git pull`
   Download the latest changes (if someone else is working on the repo).
4. `git push`
   Upload your changes to the remote host.

With `git status`, you can always check the current state of your files.