# Intro to git

CS14 - S26

# Version Control Systems

Writing software is complex. Managing software projects is also complex!

We want easy ways to:

- Develop new features without breaking the main project
- Revert to an old version when changes break something
- Keep track of contributions
- Visualize the changes a project has gone through over time

Version Control Systems (VCSs) are designed to solve these problems.

git is the most commonly used VCS.

# git vs GitHub

## git

- Free, open source software
- Specific, popular data/history model
- Ability to collaborate remotely over the internet
- Integrates with many different front ends
- Other VCSs exist, but git is most popular

## GitHub

- Proprietary hosting platform
- Additional functionality beyond git software (e.g., forking, CI/CD tools, issue tracking)
- GitHub API allows for interacting with GitHub programmatically.
- Competitors exist, but GitHub is most popular for open source projects

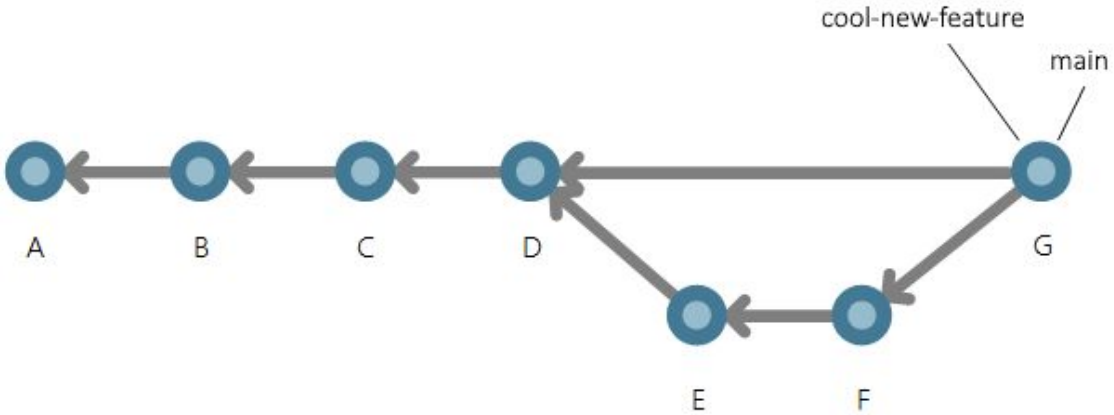
# git Data Model

- file = “blob”
- folder/directory = “tree”
- commit = snapshot of blobs and trees at a point in time
- repository (repo) = collection of blobs, trees, and commits

- blobs, trees, and commits (objects) each have an associated SHA-1 hash.

  A SHA-1 hash is basically a long unique string.

- History is a Directed Acyclic Graph of commits. Each commit points to its parent(s).
- Outside of special (often discouraged) commands, history is immutable.




```
λ git log --oneline --graph --color --all --decorate
*   04b26ba (HEAD -> main) Merge feature3
|\
| * ae59408 (feature3) Commit G
| * 854dc3e Commit F
* |   1da0602 Merge feature1
|\ \
| |/
|/|
| * f0525d5 (feature1) Commit B
| * d6237f5 Commit A
* | 1c2bf32 (feature2) Commit E
* | 9ab6898 Commit D
* | fc6a971 Commit C
|/
* 729eccd Initial commit
```

Source: https://learn.microsoft.com/en-us/devops/develop/git/understand-git-history

# Common git commands

Create a repository – `git init`

Add files/directories to be tracked by git – `git add <name>`

Make a commit – `git commit [-m <message>]`

# Branches and Merging

A branch is a human-readable label for a series of commits.

Each repository typically has a “main” branch (also sometimes called “master” or “trunk”).

Having multiple branches allows for developing multiple features/bug fixes/etc. from a single parent commit.

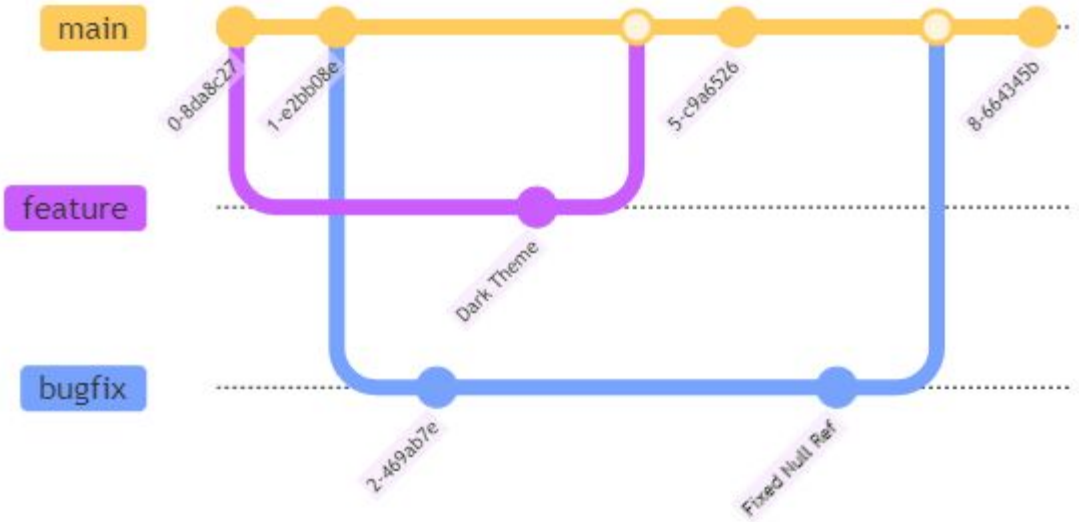

Source: https://newdevsguide.com/2023/04/11/illustrating-git-branching-with-markdown-mermaid-js/

# Branches and Merging

Changes from one branch are incorporated into another by “merging”.

Merging can typically be done automatically by git. git can automatically merge so long as different files were modified and/or different parts of the same file were modified.

If multiple different commits have happened on two different branches, then a special merge commit is needed to combine them.




Source: https://newdevsguide.com/2023/04/11/illustrating-git-branching-with-markdown-mermaid-js/

# Common git commands

Create a branch – `git branch <name>`

Switch to a branch – `git switch <name>`

Merge a branch – `git merge <name>`

- Merging applies changes from another commit onto the current one.
- E.g., suppose you are on the `main` branch. `git merge bug_fix` will apply the changes from the `bug_fix` branch onto the `main` branch.

Delete a branch – `git branch -d <name>`

# Staging Area

git only tracks blobs and trees that are explicitly added.

You can check what untracked changes there are (and other details) with `git status`

# Adding Files and gitignore

You can add all of the files and directories in a repository at once using `git add .`

However, this is bad practice because you may add unwanted files!

…unless you have a comprehensive `.gitignore`

# Adding Files and gitignore

If you add and commit a `.gitignore`, any files listed there will not be added.

- gitignore supports wildcards.
- Comments starts with `#`.

```
#Example C project .gitignore
kennedy_space_center
*.o
*.a
a.out
```

# Adding Files and gitignore

There are more fine grained ways to add multiple files/directories at once.

E.g. `git add -u` adds all files that have been committed before and have updates.

See `git help add` for more details.

# Tags

You can give arbitrary commits a human readable “tag”. These are often used to mark versions and feature releases.

Apply a tag to current commit – `git tag <tag name>`

There is a special tag called `HEAD`.

`HEAD` is always a tag of whatever branch/commit that the repository is currently on. It can be used to refer to the current commit in other commands.

# Diff

git allows you to easily compare files between different commits.

```
git diff <commit/tag 1> <commit/tag 2> <file>
```

You can also see all of the changes to a file historically.

```
git log -p <file>
```

# Remote Repositories

git supports collaboration among multiple developers through remote repositories.

One common workflow is to create a remote repository (e.g. through GitHub) and then make a linked, local copy of the repository with `git clone`. By default, clone will make a default (main) branch that “tracks” the same branch on the remote repository.

It is possible to have local branches that are not on the remote repository and vice versa.

History on tracked branches is shared between the remote and all of its copies.

You can use the `git remote` to list details about the remote repository, add a remote repository if one isn’t set, etc.. See `git help remote` for more information.

# Remote Repositories

`git pull` is used to fetch and merge any commits on the current branch that exist in the remote repository, but not the local one.

`git push` is used to send objects to a branch on the remote repository.

# Remote Repositories

- `git remote`: list remotes
- `git remote add <name> <url>`: add a remote
- `git push <remote> <local branch>:<remote branch>`: send objects to remote, and update remote reference
- `git branch --set-upstream-to=<remote>/<remote branch>`: set up correspondence between local and remote branch
- `git fetch`: retrieve objects/references from a remote
- `git pull`: same as `git fetch; git merge`
- `git clone`: download repository from remote

Source: https://missing.csail.mit.edu/2026/version-control/

# Fast-forwarding, divergence, and merge conflicts

- If a user pulls from a branch that has more recent commits than the local version, and there are no new local commits, git will do a “fast-forward”, where the remote and local branches are synched up.
- If a local branch and the corresponding remote branch both have commits that the other doesn’t have, they are “divergent”.

  Normally, git will force the local user to pull and merge the remote commits before pushing anything.

  - If git can automatically merge the differences, it will do so and prompt the user for a commit message.
  - Otherwise, there is a “merge conflict” that the local user must resolve.

# Merge Conflicts

A merge conflict occurs when two versions of a file in different commits cannot be automatically merged by git. Usually, this happens when the same line(s) of code are modified in two commits.

git will modify the conflicting file to have both versions of the conflicting text, and will mark each version with its tag or commit hash.

```
<<<<<<< HEAD
// HEAD code version
=======
// bug_fix code version
>>>>>>> bug_fix
```

# Merge Conflicts

Merge conflicts can be resolved in multiple ways.

The simplest (but not necessarily easiest) is to just open the conflicting file and modify it so that contains the code you want.

Then, to resolve the conflict, add and commit the modified file.

# Merging Exercise

Run the command:

```
git config --global pull.rebase false
```

Clone merging-practice-<your name> from GitHub Enterprise into two different directories named `1/` and `2/`.

```
git clone <link> 1
```

```
git clone <link> 2
```

# Merging Exercise

1. In `1/`, add “starfruit” alphabetically to `fruits.txt`, then add-commit-push.
   In `2/`, add “bokchoy” alphabetically to `vegetables.txt`, then add-commit.

   What happens when you try to push from `2/`?
   Try pulling inside of `2/`. Once you have resolved the pull, then push.

2. In `1/`, add “longan” alphabetically to `fruits.txt`, then add-commit-push.
   In `2/`, add “pricklypear” to alphabetically `fruits.txt`, then add-commit.

   What happens when you try to push from `2/`?
   Try pulling inside of `2/`. Once you have resolved the pull, then push.

# Merging Exercise

3. In `1/`, add “yuzu” to the bottom of `fruits.txt`, then add-commit-push.
   In `2/`, add “watermelon” to the bottom of `fruits.txt`, then add-commit.

   What happens when you try to push from `2/`?
   Try pulling inside of `2/`. Once you have resolved the pull, then push.

# Stash

Sometimes, you may want to quickly revert some local changes to the latest commit, e.g., when you want to pull from a remote repository.

`git stash push` saves uncommitted changes to a special stash entry and changes the files and directories to match the last commit.

`git stash pop` will attempt to apply the most recently stashed changes to the current files and directories, removing the stash entry.

If `pop` fails due to a merge conflict, you must merge the changes then call `git stash drop` to manually delete the stash entry.

# Forking and Pull Requests

Typically, in open source projects, maintainers will not allow strangers to push changes directly to the project repository.

Instead, contributors must make a fork and then submit a pull request.

# Forking and Pull Requests

A fork is a remote copy of a repository.

As opposed to cloning, the fork is an entirely separate repository with its own copy of the history and objects.

Anyone can make a fork and push changes to their fork without affecting the original project’s repository.

# Forking and Pull Requests

A pull request is a request for the original project to incorporate changes from a fork into the original repository.

You are requesting that the maintainers “pull” from your forked repository.

This allows the maintainers of the original project to review the changes, then approve or deny the request.

# Additional Resources

https://git-scm.com/

- Online git documentation
- Also hosts the “Pro Git” book which has excellent explanations of common and advanced git features
- Learning about rebasing is ***highly recommended***

https://github.com/tpope/vim-fugitive

- A popular git integration plugin for vim

https://code.visualstudio.com/docs/sourcecontrol/overview

- An overview of git integration in VScode