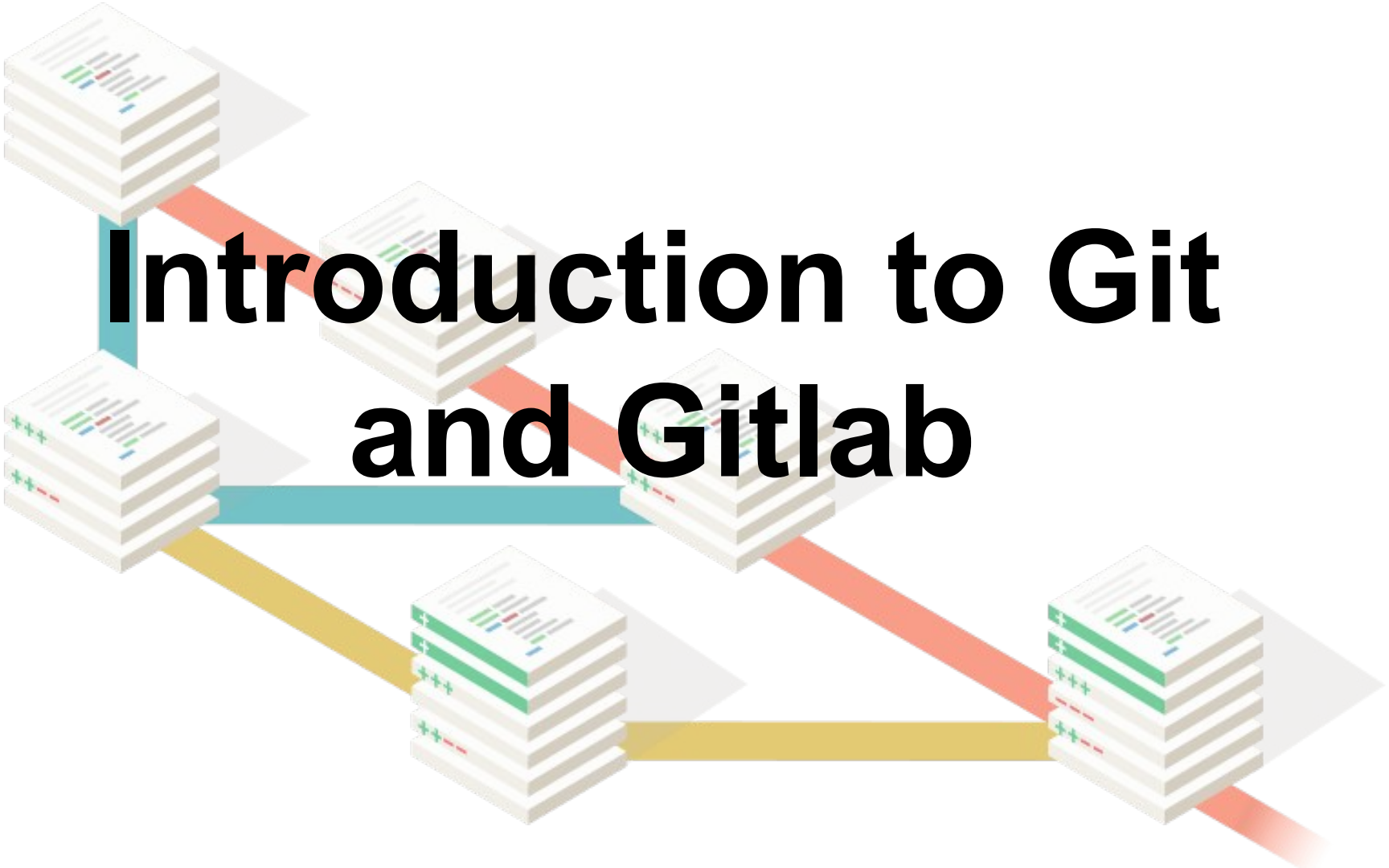
# Introduction to Git and Gitlab



# A note about how to ask for help:

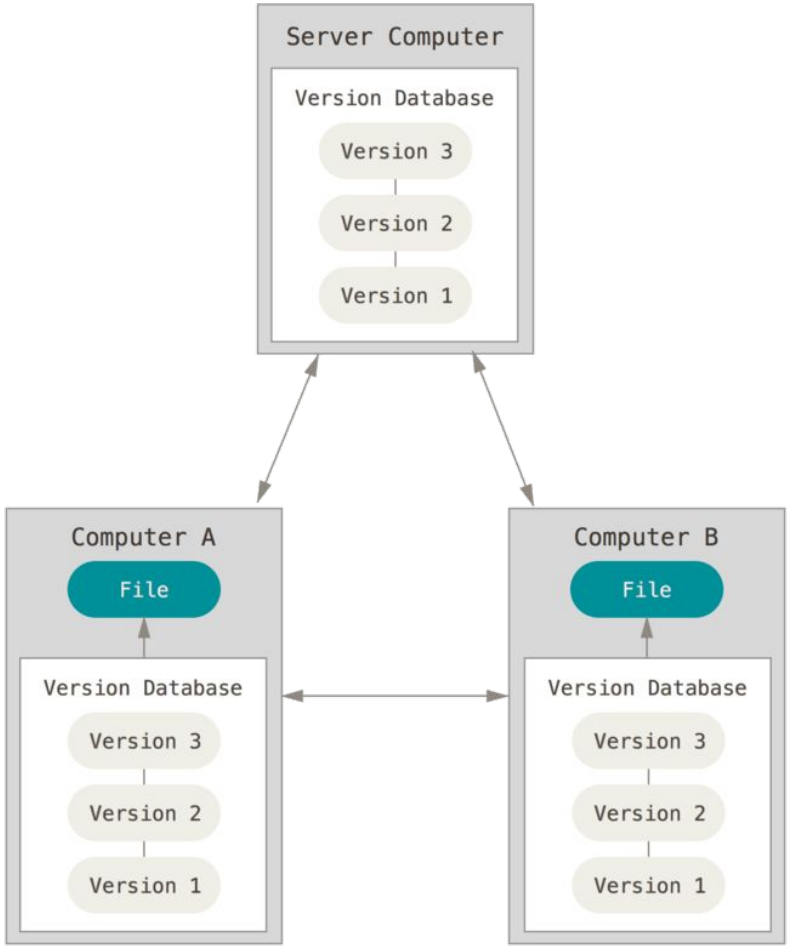
- Always google first
  - Google the error message verbatim
  - Read results even if they're for slightly different problems
  - If it's not an omf/orbit problem, we're probably just going to google it for you
- If you have problems with omf or orbit, always check the wiki first
  - Read the whole page before asking for help
  - If you have a problem while trying a tutorial, check to see if your problem is mentioned somewhere on the tutorial page.
  - If you figure out the solution to your problem, let us know and we can add it to the wiki
- When you email to ask for help:
  - Tell us what machine you are using
  - Tell us what things you did before you got the error
  - Include the full text of the error
  - Include any instructions you were following, and what you were trying to accomplish
- NEVER send an email like "I have a problem with sb5"


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.

# What is Git?

- Git is a version control system
- Git is distributed
- Git is not Github or Gitlab
- Git saves versions of directories, not files

**Version control** is a system that records changes to a file or set of files over time so that you can recall specific versions later.

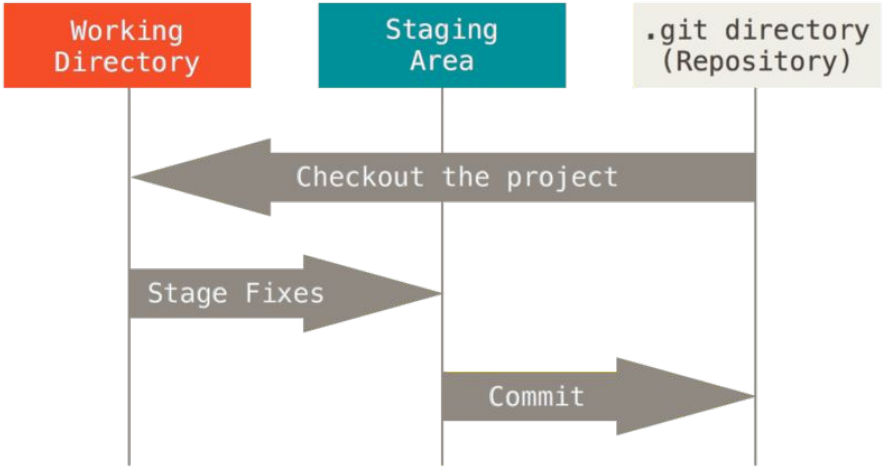


# Some Git Vocabulary

- **Commit**: a recorded snapshot of your work
- **tracked/untracked**: files which are recorded in the snapshots are “tracked”
- **Repository/Repo**: the directory being tracked by git + the full commit history
- **Check out**: retrieve a different version of the directory or a particular file from a previous commit

# How does Git keep track of things?

- Every git commit is a “snapshot” of the state of the directory
- Commits are referenced using a checksum
- You have to tell git which changes you want committed: **`git add`**
- A file being tracked by git can be either modified, staged, or committed.
- There are three main areas of a git project: working directory, staging area, and .git directory




- Basic Git Workflow:
  - Modify files
  - Stage changes
  - Commit changes

## Setting up Git

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

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

$ git config --global core.editor “vim”

## Getting Help

$ git help <verb>

$ man git-<verb>

$git <verb> -h

# Getting a local repository

- Creating a repository from an existing directory:

```
git init
```

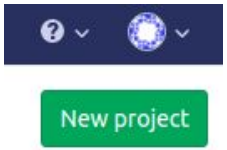
- Cloning an existing repository:

```
git clone https://gitlab.orbit-lab.org/sample/sampleRepo.git
```

```
git clone git@gitlab.orbit-lab.org:sample/sampleRepo.git
```

# Getting a remote (gitlab) repository

- Creating a new repository:




- Forking an existing repository:

# Basic repository usage

- **`git add <files>`**
- **`git commit -m “commit message”`**

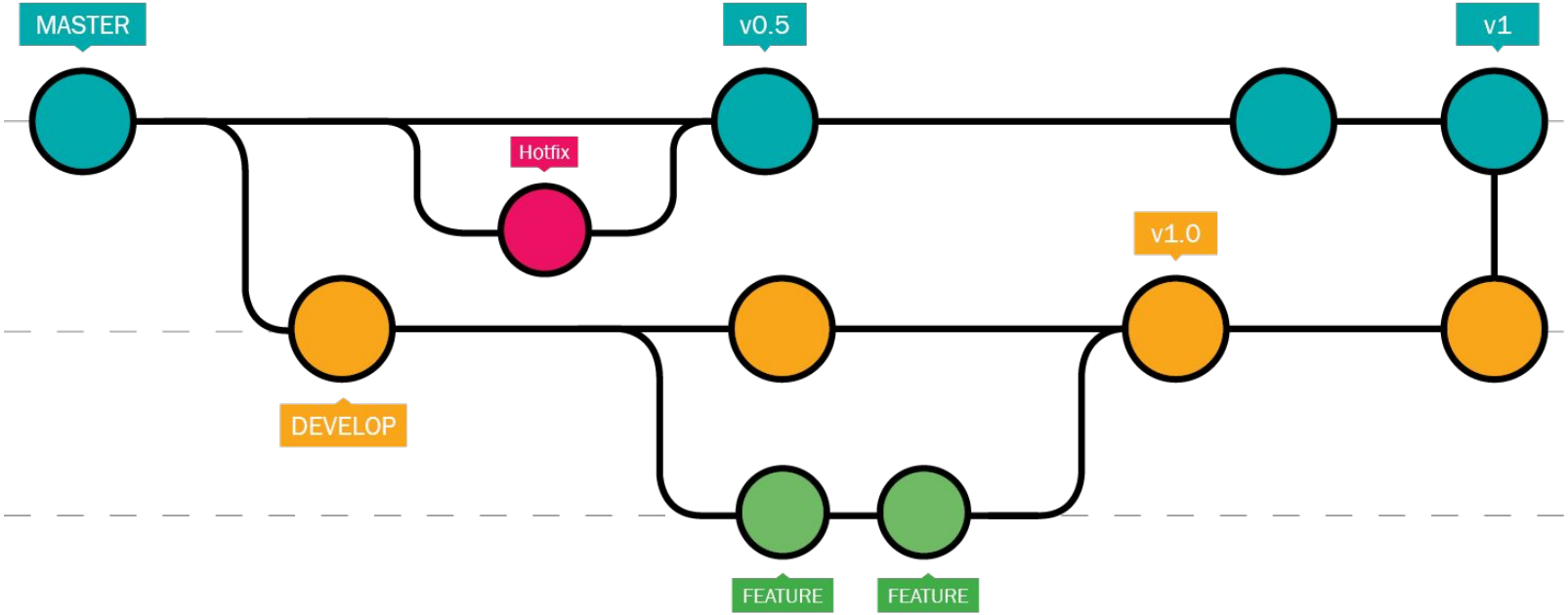
# Checking your repository

- **`git status`** : shows the state of your working directory and staging area
- **`git diff`** : shows the differences between current working directory and staging area
- **`git diff --staged`** : shows difference between staged changes and last commit
- **`git log`** : lists commits made in a repository (check documentation for lots of output options)

# Frequently used terms

- **`master :`** default branch name
- **`origin :`** frequently used name for a remote repository
- **`working directory/tree :`** directory as it currently is, including local changes to the most recently checked out commit
- **`index :`** file that contains information about what's going into the next commit. Also called the staging area
- **`HEAD :`** pointer that refers to the current location of the working directory in the history
- **`SHA :`** sometimes used to refer to the hash number that indicates a particular commit

# How does Git keep track of history?



# How to fix common mistakes

- **`git commit --amend :`** applies current staging area to previous commit, and allows you to redo the commit message
- **`git reset HEAD <filename> :`** unstages a file
- **`git checkout -- <filename> :`** returns a file to the state it was in at last commit

# .gitignore file

- Allows you to specify files and file types for git to ignore automatically
- Make sure you commit it!

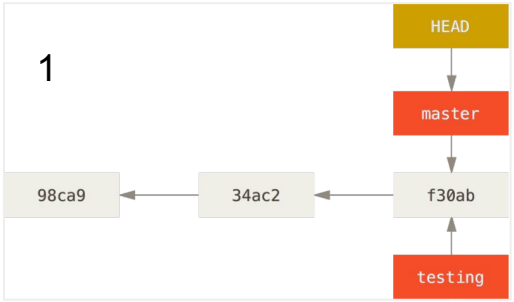
# Using your remote repository

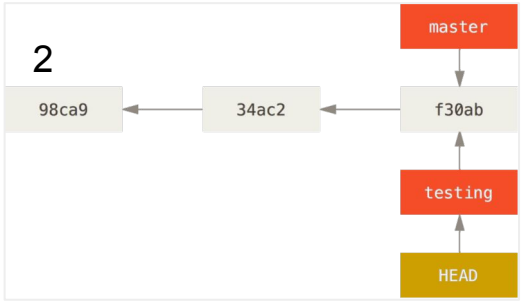
- **`git fetch :`** gets changes from the remote repository, but doesn't integrate them into your local repo
- **`git pull :`** gets changes from the remote repository and automatically integrates them
- **`git push :`** sends your local changes to the remote repository

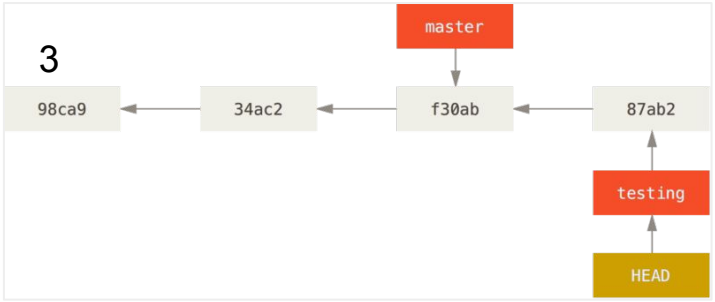
Note: if there are changes on the remote repository that are not in your commit history (if group members have been pushing while you've been working), git will ask you to **`pull`** before you can **`push`**

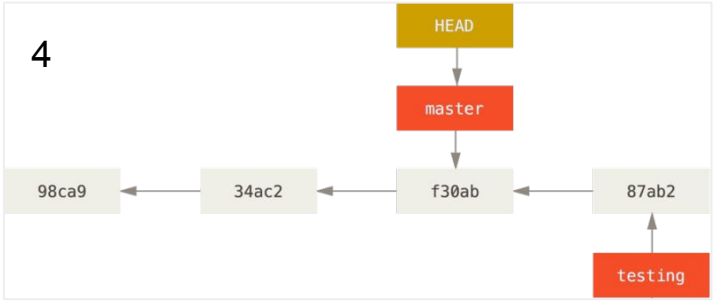
# Branching

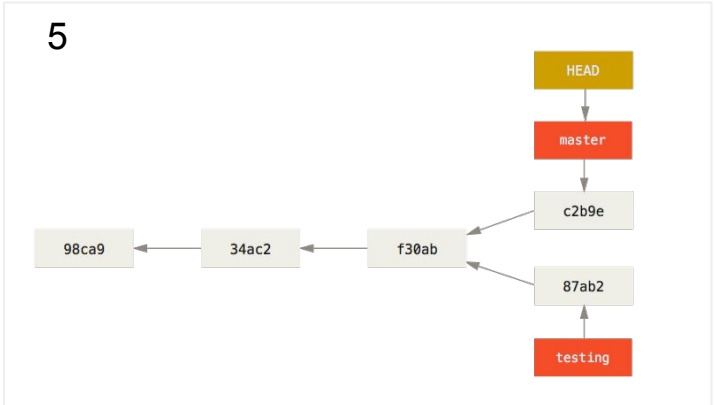
- Git stores commits as a graph, with a pointer to a particular commit indicating where the working directory is (HEAD)
- If you create another pointer, you can begin a new line of commits in the history-- this is a **branch**



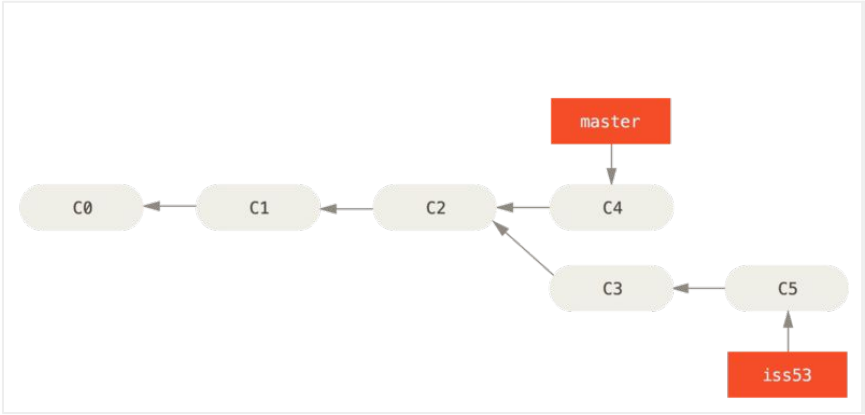



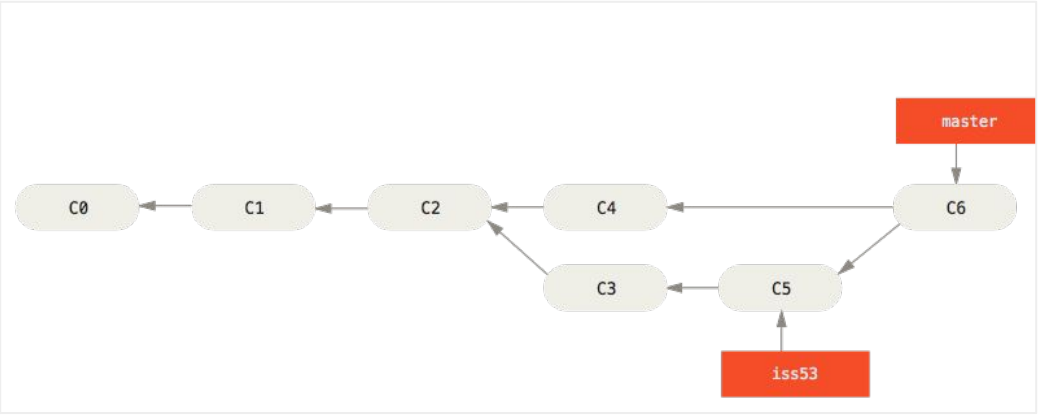



# Branching

- **`git branch`** : lists local branches
- **`git branch <branch name>`** : creates a new branch from the current HEAD
- **`git checkout <branch name>`** : switches HEAD to <branch name>
- **`git checkout -b <branch name>`** : creates a new branch called <branch name> and checks it out
- **`git merge <branch name>`** : merges <branch name> into current HEAD
- **`git branch -d <branch name>`** : deletes the branch (make sure you've merged your changes!)



# Handling Merge Conflicts

- A merge conflict happens when changes on the two branches conflict with each other

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then
commit the result.
```

1. Open the conflicting file and handle the change manually
2. **`git add`** the file
3. **`git commit`** the changes

# Branches on a remote repository

- **`git push origin <branch name>`** : pushes your local branch to the remote repository called origin
- **`git fetch`** : returns any new branches in the remote repo, and updates the pointers old ones
- **`git checkout --track origin/<branch name>`** : creates a local copy of the remote branch that will track remote changes to the branch
- **`git push origin --delete <branch name>`** : delete the remote copy of the branch

# Git reset and git checkout

- **`git reset`** can be used to reset the working directory, the staging area, and the HEAD
- **`git reset --soft`** resets the state of the HEAD pointer, but leaves the working directory and staging area the same
- **`git reset`** or **`git reset --mixed`** resets the HEAD pointer and the staging area, but does not change the working directory
- **`git reset --hard`** changes the HEAD, the working directory, and the staging area. This **will** overwrite your work.
- This can be used to squash your commits
- **`git checkout <commit or branch> -- <filename>`** : places <filename> from <commit or branch> in your working directory. This will overwrite any local changes, so be careful.

# Miscellaneous Gitfalls

- Committing large files
  - Use .gitignore to make sure you don’t commit binaries or datafiles
  - Git LFS
  - Google “unity .gitignore”
- Committing passwords
- Letting personal branches get too out of date with master
- Overwriting local history after it’s been pushed. This can be fixed with a **`git push --force`**, but be very careful

# Using Git LFS

- **`Git lfs install`**
- **`Git lfs track <*.ext>`**
- Add and commit .gitattributes
- Add and commit your files as usual
- **`Git lfs ls-files`**

# Workflow for small collaborative projects

Many git projects involve a small team of contributors who share a remote repository. This is the simplest possible workflow.

- Contributors keep their work in branches, which can be either private, local branches that they want to keep to themselves, or shared branches that are tracked in the remote repository
- Developers regularly fetch and merge changes from the remote to keep up to date.
- When work on a branch is complete, it is merged into master locally and then pushed.
- This relies a bit on the honor system-- nothing stops a contributor from merging whatever they want into master!

# Workflow for contributing to a large github repo

For many large projects, it’s not practical to allow anyone to be a contributor. In these cases, the project is managed by a maintainer, who approves changes submitted by people who want to contribute.

- To contribute to a project, fork it, then clone
- Set the original github repo as an upstream remote
- Identify something you want to fix or add, and submit an issue
- Make your changes in a branch, pulling frequently from the upstream repo
- Merge upstream master into your branch
- Submit a pull request

# Best practices for commit messages and clean history

- Squash messy, frequent commits before pushing
- Keep your commits clear and focused
- Commit messages should have a subject/summary on the first line, followed by a blank line then a brief paragraph describing the changes and what they are intended to solve or add.

# Resources

- https://git-scm.com/book/en/v2 : Pro Git, Scott Chacon and Ben Straub. Available free online.
- https://git-scm.com/docs/gitglossary : glossary of frequently used terms related to git
- https://git-scm.com/docs/ : documentation, including man pages and guides
- https://guides.github.com/ : github guides
- https://www.google.com/ : when you have a problem, always try googling first!