

## Introduction to Git and Version Control

2021-09-09

Introduction to Git and Version Control  
Lecture 1: Git ready!

Christopher Buckley

Okinawa Institute of Science and Technology

August 16, 2021

Original Slides by James Schloss, 2016

# Overview



## 1 Why Git?

- Traditional vs. Git Versioning

## 2 What is Git

- Three Main Parts (Working Directory, Stage, Repository)

## 3 How to Use Git

- The Terminal
- Create a repo
- Commits
- The Stage (Staging Area, or Index)
- Making Commits
- Checking Out Past Commits
- Revert
- Reset
- Rebase

## 4 Working Online

## 5 Wrap Up

# Introduction to Git and Version Control

2021-09-09

## └ Overview

- Git is different from github
- Why Git? I'll tell you what my motivations are, but what are your motivations for being here?

- Version control
- Easily compare and merge changes between any version
- Organize your work items



# Why Git?

- Version control
- Easily compare and merge changes between any version
- Organize your work items



Before

After

## Introduction to Git and Version Control

### └ Why Git?

#### └ Why Git?

Imagine I asked you to remove the red sharpie marker from the left hand side?

- Difficulty finding it
- Could dive right in, but might get poked by lot of sharp things on the way in
- Or you could dump everything out and start all over

# Why Git?



A photograph showing a massive, chaotic pile of discarded wooden chairs stacked against the side of a multi-story building. A small figure of a person stands at the bottom left, highlighting the enormous scale of the waste.



Introduction to Git and Version Control

2021-09-09

Introduction to Git and Version Control

└ Why Git?

└ Why Git?

Imagine I asked you to remove this chair. What difficulties would you face?

- How to access it safely
- Can't remove it without fearing everything will fall

Why Git?

Christopher Buckley (OIST)

Introduction to Git and Version Control

August 16, 2021

4 / 56

# Traditional vs. Git Versioning

- What changed when
- Not limited to file name length to inform user of changes

```
christopher@christopher-ThinkPad-W541:~/git/mythesis$ ls -ltr
total 4
-rw-rw-r-- 1 christopher 51 Nov 17 18:22 thesis
-rw-rw-r-- 1 christopher 0 Nov 18 12:07 thesis_v1
-rw-rw-r-- 1 christopher 0 Nov 18 12:07 thesis_v2
-rw-rw-r-- 1 christopher 0 Nov 18 12:07 thesis_v3
-rw-rw-r-- 1 christopher 0 Nov 18 12:07 thesis_v4
-rw-rw-r-- 1 christopher 0 Nov 18 12:07 thesis_final
-rw-rw-r-- 1 christopher 0 Nov 18 12:07 thesis_final1
-rw-rw-r-- 1 christopher 0 Nov 18 12:07 thesis_final2
-rw-rw-r-- 1 christopher 0 Nov 18 12:07 thesis_final3
-rw-rw-r-- 1 christopher 0 Nov 18 12:07 thesis_final4
christopher@christopher-ThinkPad-W541:~/git/mythesis$
```

```
christopher@christopher-ThinkPad-W541:~/git/mythesis$ git log --reverse
commit e393a47e257310df071ac8290cdcf0c4a60bb8944 (master)
Author: Christopher Buckley <15166572+topherbuckley@users.noreply.github.com>
Date: Tue Nov 17 18:14:52 2020 +0900

    Added empty thesis template

commit e017bf79743fa7724d4c35f430e1e78064823e1a
Author: Christopher Buckley <15166572+topherbuckley@users.noreply.github.com>
Date: Tue Nov 17 18:19:14 2020 +0900

    Added initial title

commit 750455880517cbdd5db25054e0e9815ed67da185
Author: Christopher Buckley <15166572+topherbuckley@users.noreply.github.com>
Date: Tue Nov 17 18:20:38 2020 +0900

    Added initial summary section

commit a83135a00c134372a7973a879e2431008b5a466
Author: Christopher Buckley <15166572+topherbuckley@users.noreply.github.com>
Date: Tue Nov 17 18:21:09 2020 +0900

    Added initial bulk of main body section

commit 98c17b7472ef14bd75804d4ddb990de92f211ba
Author: Christopher Buckley <15166572+topherbuckley@users.noreply.github.com>
Date: Tue Nov 17 18:21:31 2020 +0900

    Added initial conclusions

commit aa3034ff24084d3f95e37ae222f265da07d3590
Author: Christopher Buckley <15166572+topherbuckley@users.noreply.github.com>
Date: Tue Nov 17 18:22:03 2020 +0900

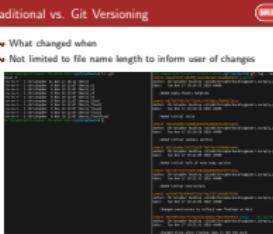
    Changed conclusions to reflect new findings on Mars

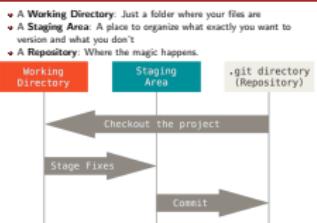
commit d918fbfe43cf474a34c6cd86ccf845f63e9cb4 (HEAD -> new_versioning)
Author: Christopher Buckley <15166572+topherbuckley@users.noreply.github.com>
Date: Tue Nov 17 18:22:31 2020 +0900

    Changed title after finding typo in teh the word
```

Introduction to Git and Version Control  
└ Why Git?  
 └ Traditional vs. Git Versioning  
 └ Traditional vs. Git Versioning

2021-09-09





# What is git?

2021-09-09

- A **Working Directory**: Just a folder where your files are
- A **Staging Area**: A place to organize what exactly you want to version and what you don't
- A **Repository**: Where the magic happens.

The diagram shows the three main components of Git:

- Working Directory**
- Staging Area**
- .git directory (Repository)**

Arrows indicate the flow between these components:

- A large arrow points from the Working Directory to the Staging Area, labeled "Checkout the project".
- A large arrow points from the Staging Area to the .git directory, labeled "Stage Fixes".
- A large arrow points from the .git directory back to the Working Directory, labeled "Commit".

Introduction to Git and Version Control

- └ What is Git
- └ Three Main Parts (Working Directory, Stage, Repository)
- └ What is git?

This chart might not make much sense now, but I hope it will before the end of these slides

Whether you realize it or not, you are already familiar with the "Working Directory". I'll save the staging area for a bit later, but first lets talk about what a repo is.

# Repositories

- A **repository** is a container for both your project data and all the items that allow interactions with git commands.
  - There are many sites to host your repository on (github, bitbucket), including your own local machine.
  - All of the essential parts of your repository can be found in the `.git` directory
  - GitHub (a website hosting Git repositories)  $\neq$  Git (a set of tools for creating and managing those repositories).



## Introduction to Git and Version Control

### └ What is Git

- └ Three Main Parts (Working Directory, Stage, Repository)
  - └ Repositories

2021-09-09

Repositories

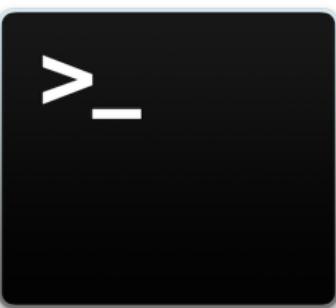


- A **repository** is a container for both your project data and all the items that allow interactions with git commands.
  - There are many sites to host your repository on (github, bitbucket), including your own local machine.
  - All of the essential parts of your repository can be found in the `.git` directory
  - GitHub (a website hosting Git repositories)  $\neq$  Git (a set of tools for creating and managing those repositories).

# Terminal Talk



- There are multiple GUIs available for Git, such as one from GitHub called the **GitHub Desktop**. We will not be using this for religious perfectly scientific reasons.
- These reasons primarily revolve around flexibility and improved understanding of the Git tools.
- Everything we do will be usable on Deigo.
- The **Pro Git** book is available online at [git-scm.com/book](http://git-scm.com/book)
- There is a cheatsheet for Git available here: <https://www.git-tower.com/learn/cheatsheets/git>



2021-09-09

## Introduction to Git and Version Control

- └ How to Use Git
  - └ The Terminal
    - └ Terminal Talk

- There are multiple GUIs available for Git, such as one from GitHub called the **GitHub Desktop**. We will not be using this for religious perfectly scientific reasons.
- These reasons primarily revolve around flexibility and improved understanding of the Git tools.
- Everything we do will be usable on Deigo.
- The **Pro Git** book is available online at [git-scm.com/book](http://git-scm.com/book)
- There is a cheatsheet for Git available here: <https://www.git-tower.com/learn/cheatsheets/git>

- I personally struggled with the terminal interface at first because most of the man pages use so much vocab I don't know to explain terms I don't know. Hopefully by the end of this mini-course you'll have the basic vocab down so you can help yourselves more efficiently going forward.

# Create a Repo(sitory)



Let's **git** started.

- To initialize a git repository, simply type **git init** in a directory (preferably empty for now)
- This creates a folder **.git/**, where all your repository information is held.



Introduction to Git and Version Control  
└ How to Use Git  
  └ Create a repo  
    └ Create a Repo(sitory)

2021-09-09

Create a Repo(sitory)

Let's **git** started:

- To initialize a git repository, simply type **git init** in a directory (preferably empty for now)
- This creates a folder **.git/**, where all your repository information is held.





## EXERCISE

- ① Open a terminal
- ② Create a new directory called **myFirstRepo** and enter it.
- ③ This is your Working Directory. Thats it!
- ④ Run **git init** in your Working Directory.
- ⑤ Take a peak in the newly created .git directory but don't touch anything quite yet.

## Introduction to Git and Version Control

### └ How to Use Git

#### └ Create a repo

##### └ Create an Empty Repo

2021-09-09

Create an Empty Repo

### EXERCISE

- ① Open a terminal
- ② Create a new directory called **myFirstRepo** and enter it.
- ③ This is your Working Directory. Thats it!
- ④ Run **git init** in your Working Directory.
- ⑤ Take a peak in the newly created .git directory but don't touch anything quite yet.

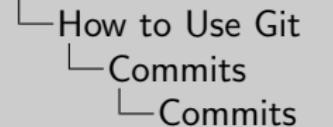
# Commits

- Conceptually similar to "versions"
- The more effort you put into crafting these the more helpful they are in the future.

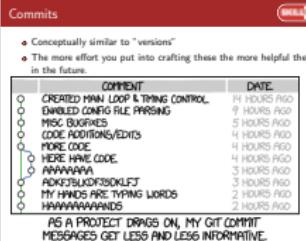
COMMENT	DATE
CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
ENABLED CONFIG FILE PARSING	9 HOURS AGO
MISC BUGFIxes	5 HOURS AGO
CODE ADDITIONS/EDITS	4 HOURS AGO
MORE CODE	4 HOURS AGO
HERE HAVE CODE	4 HOURS AGO
AAAAAAA	3 HOURS AGO
ADKFJSLKDFJSOKLFJ	3 HOURS AGO
MY HANDS ARE TYPING WORDS	2 HOURS AGO
HAAAAAAAAANDS	2 HOURS AGO

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

## Introduction to Git and Version Control



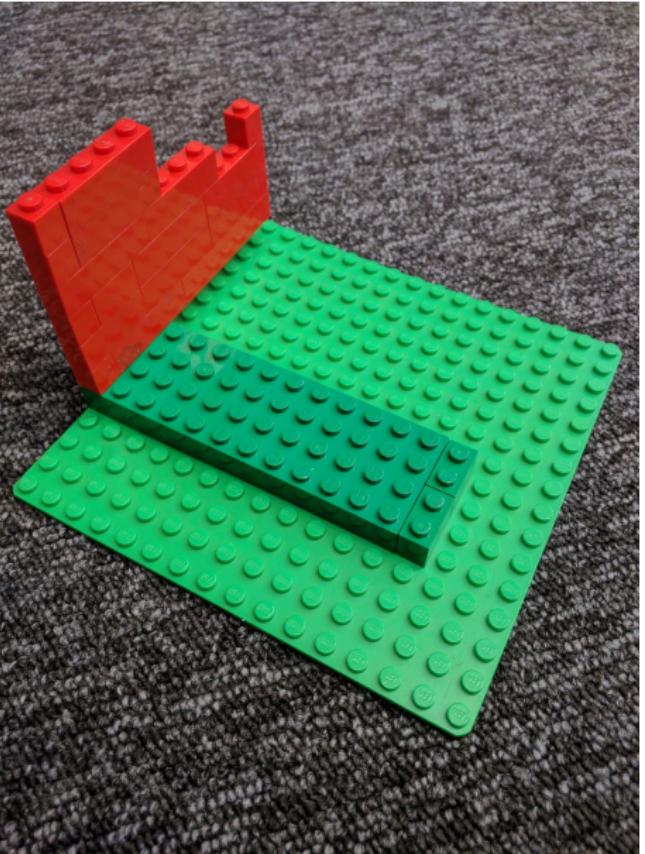
2021-09-09



- Using git in and of itself does not mean you will have better versioning.
- We need to make GOOD commits for git to have any significant benefit.
- By "versions" I mean what you may be used to seeing in dropbox or similar backup schemes.
- Git at its worst acts quite similar to this, but if used properly can add so much more.
- To clarify with a more visual example, I made a house using legos. (Lego photos on next slides).



# Commits



## Introduction to Git and Version Control

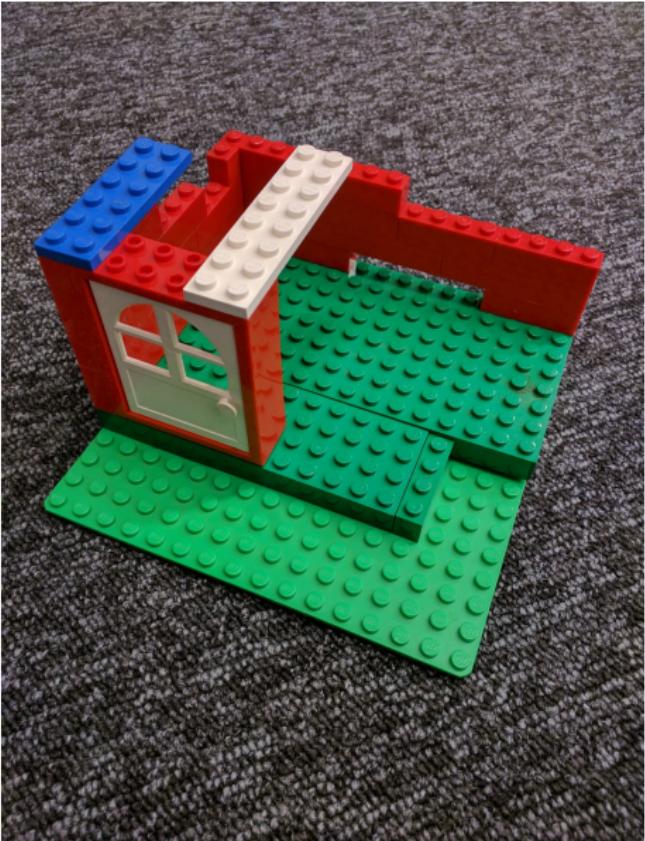
- └ How to Use Git
- └ Commits
- └ Commits

2021-09-09

Added some stuff



# Commits



## Introduction to Git and Version Control

- └ How to Use Git
- └ Commits
- └ Commits

2021-09-09

Added some more stuff



# Commits



## Introduction to Git and Version Control

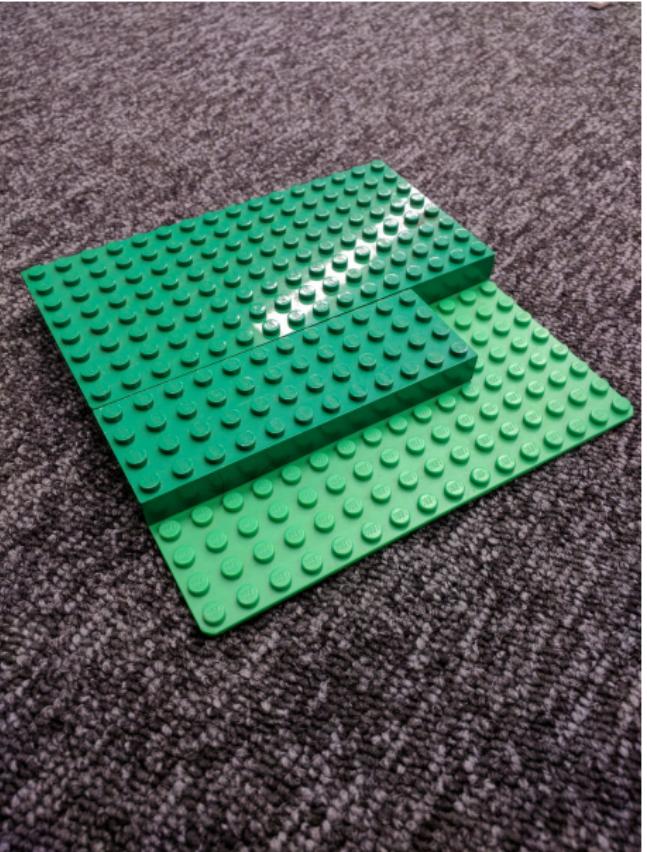
- └ How to Use Git
- └ Commits
- └ Commits

2021-09-09

Done! Now lets say we found an error in the western wall. Which commit was that added in?



# Commits



## Introduction to Git and Version Control

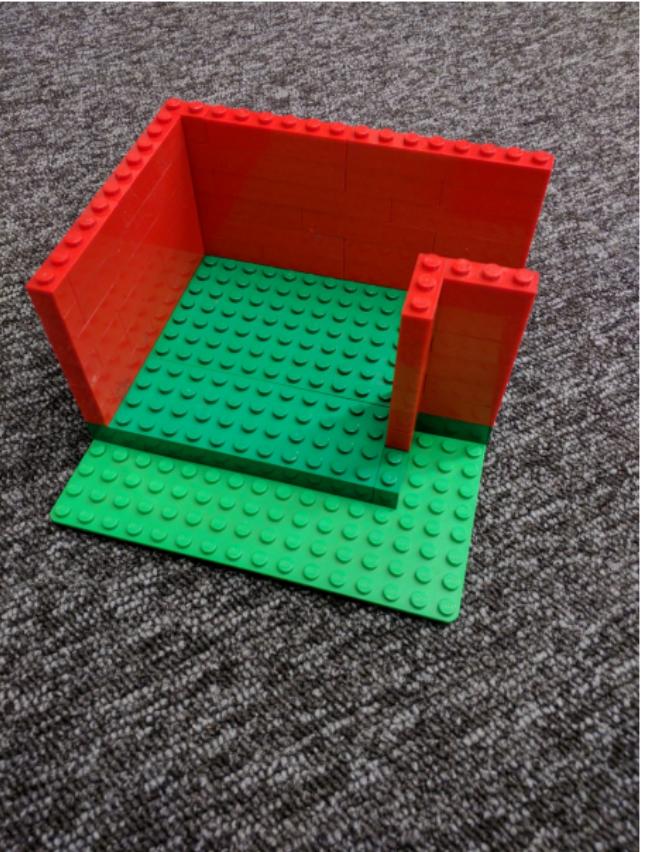
- └ How to Use Git
- └ Commits
- └ Commits

2021-09-09

Added the foundation



# Commits



## Introduction to Git and Version Control

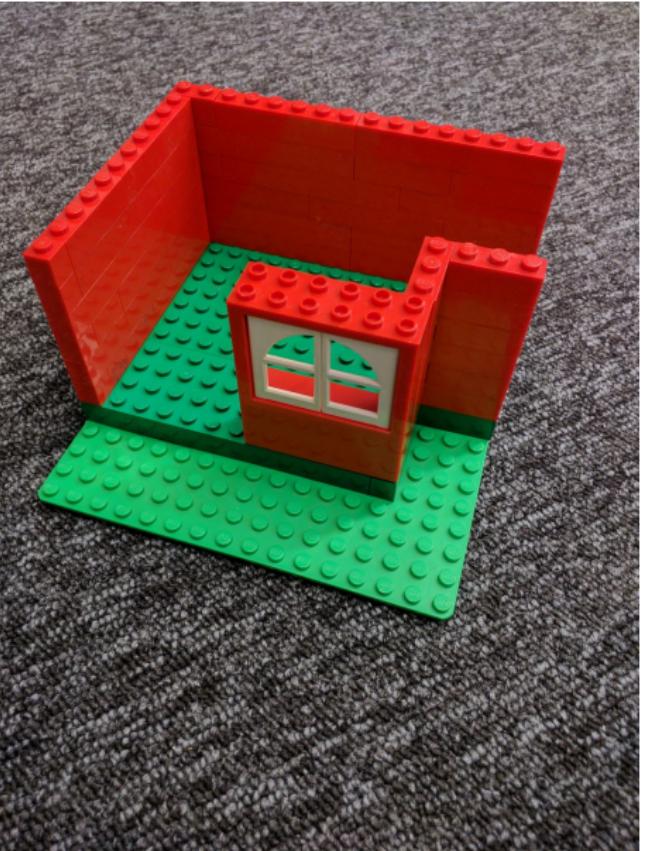
- └ How to Use Git
- └ Commits
- └ Commits

2021-09-09

Added the walls



# Commits



## Introduction to Git and Version Control

- └ How to Use Git
- └ Commits
- └ Commits

2021-09-09

Added the window



# Commits



## Introduction to Git and Version Control

- └ How to Use Git
- └ Commits
- └ Commits

2021-09-09

Added the door



# Commits



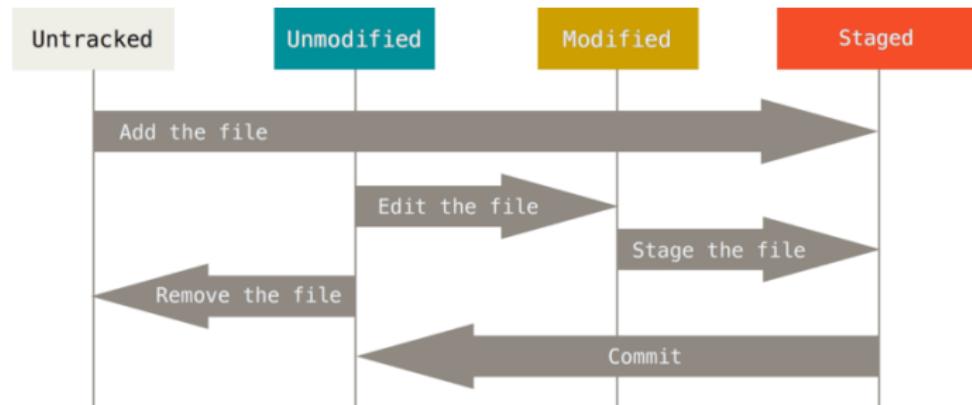
## Introduction to Git and Version Control

- └ How to Use Git
- └ Commits
- └ Commits

2021-09-09

Added the roof

# Staging Changes



- A new file is initially **untracked**
- When you use **git add**, it moves to the staging area and becomes **staged**
- After being committed (using **git commit**), a file is up-to-date and considered **unmodified**
- Changing a file makes it modified, but doesn't add it to the staging area

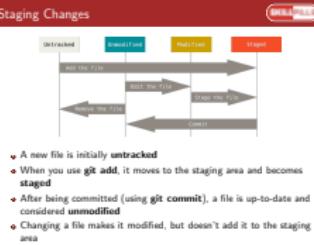
## Introduction to Git and Version Control

### How to Use Git

#### The Stage (Staging Area, or Index)

##### Staging Changes

2021-09-09



So how do we construct or make these GOOD commits? By using the Staging Area or Index. Next we'll discuss what this is and how to use it to make GOOD commits.

# Currying the Stage before Committing



- Check what is on the stage with **git status**. Anything in **green** is staged.
- If you wish to unstage all changes, simply type **git reset**. This will remove everything from the stage, but keep your working directory untouched.
- **git reset** will work for individual files as well

---

```
git reset <file>
```

---



2021-09-09

## Introduction to Git and Version Control

- └ How to Use Git
  - └ The Stage (Staging Area, or Index)
    - └ Currying the Stage before Committing

Currying the Stage before Committing

- Check what is on the stage with **git status**. Anything in **green** is staged.
- If you wish to unstage all changes, simply type **git reset**. This will remove everything from the stage, but keep your working directory untouched.
- **git reset** will work for individual files as well  
`git reset <file>`



# Try out the Stage



## EXERCISE

- ① Create a new empty file **myfile.txt**
- ② Check the status of everything with **git status**
- ③ Add **myfile.txt** to the stage via **git add myfile.txt**
- ④ Check the status of everything again with **git status**. What changed?
- ⑤ Unstage the changes with **git reset myfile.txt**
- ⑥ Check the status of everything again with **git status**. What changed?

## Introduction to Git and Version Control

- └ How to Use Git
  - └ The Stage (Staging Area, or Index)
    - └ Try out the Stage

2021-09-09

### EXERCISE

- ① Create a new empty file **myfile.txt**
- ② Check the status of everything with **git status**
- ③ Add **myfile.txt** to the stage via **git add myfile.txt**
- ④ Check the status of everything again with **git status**. What changed?
- ⑤ Unstage the changes with **git reset myfile.txt**
- ⑥ Check the status of everything again with **git status**. What changed?



- Git keeps track of **commits**. Check these commits with **git log**.  
There's plenty of options to show only what you want or everything under the sun.
- **git status** checks any changes since the last commit.
- **git commit** commits everything in the *staging area* - git status shows these files in **green** by default.

## Introduction to Git and Version Control

### └ How to Use Git

#### └ The Stage (Staging Area, or Index)

##### └ Committing from the Stage

2021-09-09

- Git keeps track of **commits**. Check these commits with **git log**. There's plenty of options to show only what you want or everything under the sun.
- **git status** checks any changes since the last commit.
- **git commit** commits everything in the *staging area* - git status shows these files in **green** by default.



## EXERCISE

- ① Reopen your **myFirstRepo** from before
- ② Add the **myFile.txt** back to the stage with **git add myFile.txt**
- ③ Check the status of the stage with **git status**
- ④ Once satisfied with what is in the stage and you're ready to commit, go ahead and do so with **git commit** to add your new file to the git repository. Be sure to add a meaningful commit message!
- ⑤ Check the **git log**.
- ⑥ Check the **git status**
- ⑦ Add a line of text to **myFile.txt** and save it.
- ⑧ Check the status of the stage with **git status**
- ⑨ Check the differences in the file with **git diff**
- ⑩ Once satisfied with your changes, add it back to the stage and commit.

## Introduction to Git and Version Control

- └ How to Use Git
  - └ Making Commits
    - └ Making Commits

2021-09-09

Making Commits

EXERCISE

- ➊ Reopen your **myFirstRepo** from before
- ➋ Add the **myFile.txt** back to the stage with **git add myFile.txt**
- ➌ Check the status of the stage with **git status**
- ➍ Once satisfied with what is in the stage and you're ready to commit, go ahead and do so with **git commit** to add your new file to the git repository. Be sure to add a meaningful commit message!
- ➎ Check the **git log**.
- ➏ Check the **git status**
- ➐ Add a line of text to **myFile.txt** and save it.
- ➑ Check the status of the stage with **git status**
- ➒ Check the differences in the file with **git diff**
- ➓ Once satisfied with your changes, add it back to the stage and commit.

# Checking out your past commits



- **git checkout** allows you to view the repository at any commit (found with **git log**).

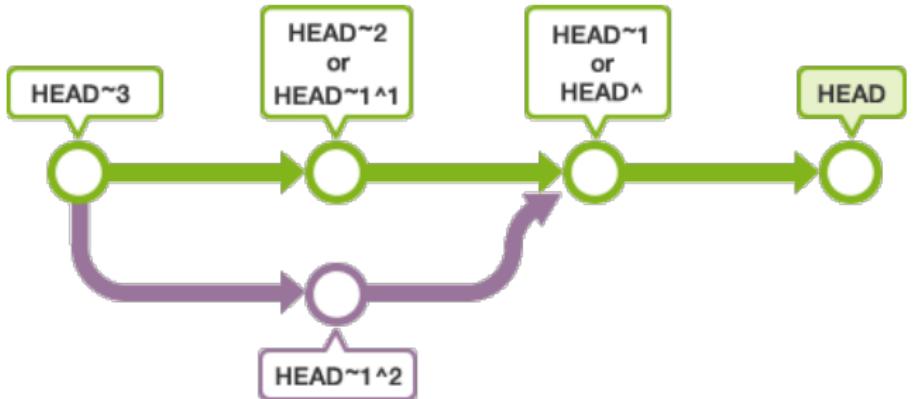
- You may also checkout specific files like so:

---

```
git checkout a1e8fb5 hello.py
```

---

- Note that the most recent commit is **HEAD** and the one just before that is **HEAD~1**



## Introduction to Git and Version Control

### └ How to Use Git

#### └ Checking Out Past Commits

##### └ Checking out your past commits

2021-09-09

## Checking out your past commits

- **git checkout** allows you to view the repository at any commit (found with **git log**).

- You may also checkout specific files like so:

```
git checkout a1e8fb5 hello.py
```

- Note that the most recent commit is **HEAD** and the one just before that is **HEAD~1**



# Checkout Your History



## EXERCISE

- ① Add a second line of text to **myFile.txt** and save it.
- ② Add these changes to the stage with **git add myFile.txt** and check the status with **git status**
- ③ Once satisfied with what is in the stage and you're ready to commit, use **git commit** to add your new file to the git repository.
- ④ Check the **git log**. You should have three commits by now.
- ⑤ Go checkout each of the commits with **git checkout <HASH>**, **git checkout HEAD~1**, or **git checkout HEAD~2**
- ⑥ See whats different with **ls -al** or **git status** or just open **myfile.txt** in your favorite text editor
- ⑦ When you are satisfied that your commit history is as expected you can return to the most recent commit with **git checkout master** (Note this could be **git checkout main** depending on your version of git.)

## Introduction to Git and Version Control

### └ How to Use Git

#### └ Checking Out Past Commits

##### └ Checkout Your History

2021-09-09

Checkout Your History

EXERCISE

- ① Add a second line of text to **myFile.txt** and save it.
- ② Add these changes to the stage with **git add myFile.txt** and check the status with **git status**
- ③ Once satisfied with what is in the stage and you're ready to commit, use **git commit** to add your new file to the git repository.
- ④ Check the **git log**. You should have three commits by now.
- ⑤ Go checkout each of the commits with **git checkout <HASH>**, **git checkout HEAD~1**, or **git checkout HEAD~2**
- ⑥ See whats different with **ls -al** or **git status** or just open **myfile.txt** in your favorite text editor
- ⑦ When you are satisfied that your commit history is as expected you can return to the most recent commit with **git checkout master** (Note this could be **git checkout main** depending on your version of git.)

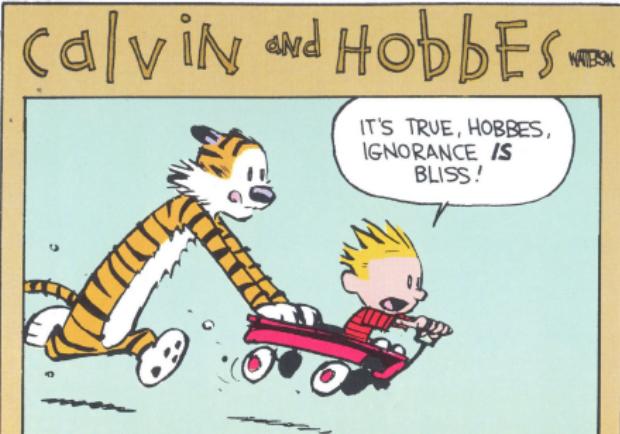
# Git Generally Only Adds

- After you commit something it is fairly difficult to remove it.
- This is a double edged sword. Low risk of losing anything permanently. High risk of creating a **HUGE** repo.
- Keep your repository clean! Do your best to commit as few images and data files as possible!
- You can do this by ignoring certain file extensions in a **.gitignore** file.
- Great templates for projects of many types found at  
<https://github.com/github/gitignore>

---

```
# Example gitignore configuration
```

```
*.log  
*.tar  
*.gz  
*.exe  
*.dat  
*.lvp
```



## Introduction to Git and Version Control

### How to Use Git

#### Checking Out Past Commits

##### Git Generally Only Adds

2021-09-09

Git Generally Only Adds

- After you commit something it is fairly difficult to remove it.
- This is a double edged sword. Low risk of losing anything permanently. High risk of creating a **HUGE** repo.
- Keep your repository clean! Do your best to commit as few images and data files as possible!
- You can do this by ignoring certain file extensions in a **.gitignore** file.
- Great templates for projects of many types found at <https://github.com/github/gitignore>

---

```
# Example gitignore configuration
*.log
*.tar
*.gz
*.exe
*.dat
*.lvp
```





## EXERCISE

- ① Touch multiple files with various extensions, one of which should be **.dat**.
- ② Ignore the **.dat** file, but commit all the others.
- ③ Be sure to write a clear message describing what you did.
- ④ Check the **git log**

## Introduction to Git and Version Control

- └ How to Use Git
  - └ Checking Out Past Commits
    - └ Quick Exercise

2021-09-09

EXERCISE

- Touch multiple files with various extensions, one of which should be **.dat**.
- Ignore the **.dat** file, but commit all the others.
- Be sure to write a clear message describing what you did.
- Check the **git log**

# Modifying Previous Commits



- If you commit something that turns out to be a mistake, don't worry! There are plenty of tools to rework commits.
- Some are more powerful (and potentially destructive than others)
- Non-destructive: (Leaves history intact)
  - **revert**
- Potentially destructive: (Changes history)
  - **reset**
  - **rebase**
- Danger Zone: (Can erase history)
  - **reflog**
- Note: These tools may take some time to conceptualize, so I encourage you to play around with them in your toy environments (e.g. `myFirstRepo`) a fair bit before trying them out on your actual code base.

## Introduction to Git and Version Control

### └ How to Use Git

#### └ Checking Out Past Commits

##### └ Modifying Previous Commits

2021-09-09

Modifying Previous Commits

- If you commit something that turns out to be a mistake, don't worry! There are plenty of tools to rework commits.
- Some are more powerful (and potentially destructive than others)
- Non-destructive: (Leaves history intact)
  - **revert**
- Potentially destructive: (Changes history)
  - **reset**
  - **rebase**
- Danger Zone: (Can erase history)
  - **reflog**
- Note: These tools may take some time to conceptualize, so I encourage you to play around with them in your toy environments (e.g. `myFirstRepo`) a fair bit before trying them out on your actual code base.



- **git revert <HASH>** makes a new commit showing what you reverted.
- Pro: This is very useful for public repos where you want to show exactly what you've undone to others.
- Con: Can make your commit history messy if used too often.

## Introduction to Git and Version Control

- └ How to Use Git
  - └ Revert
    - └ Using Revert

2021-09-09

- **git revert <HASH>** makes a new commit showing what you reverted.
- Pro: This is very useful for public repos where you want to show exactly what you've undone to others.
- Con: Can make your commit history messy if used too often.

# Using Revert



## EXERCISE

- ① Make a few commits to your **myFirstRepo** if you don't have any already. Make sure they are simple for now (just single line additions). Use **git log** to understand your current git history and **cat myfile.txt** to understand the contents of your file before reverting.
- ② Find a commit you want to revert using **git log** and **git show** or **git diff**. Ensure the commit you choose is just the addition of a single line.
- ③ Revert that commit with **git revert <HASH>** or **git revert HEAD~<#>** Use **git log** to understand your current git history. Use **cat myfile.txt** to again understand the contents of your file after reverting.

## Introduction to Git and Version Control

- └ How to Use Git
  - └ Revert
    - └ Using Revert

2021-09-09

Using Revert

### EXERCISE

- ① Make a few commits to your **myFirstRepo** if you don't have any already. Make sure they are simple for now (just single line additions). Use **git log** to understand your current git history and **cat myfile.txt** to understand the contents of your file before reverting.
- ② Find a commit you want to revert using **git log** and **git show** or **git diff**. Ensure the commit you choose is just the addition of a single line.
- ③ Revert that commit with **git revert <HASH>** or **git revert HEAD~<#>** Use **git log** to understand your current git history. Use **cat myfile.txt** to again understand the contents of your file after reverting.



- You may remember the **git reset** command to remove things from the Staging Area (undoing staging)
- We can also use **git reset** to undo commits.
- This is a great way to undo fairly recent commits (one or two before), but not the ideal tool for doing anything further than that.
- When you use **git reset HEAD^1** you undo your most recent commit but retain all the changes the commit made. This is useful if you accidentally included a file in a commit and want to go back, and remake the commit.

- └ How to Use Git
  - └ Reset
    - └ Using Reset

2021-09-09

- You may remember the **git reset** command to remove things from the Staging Area (undoing staging)
- We can also use **git reset** to undo commits.
- This is a great way to undo fairly recent commits (one or two before), but not the ideal tool for doing anything further than that.
- When you use **git reset HEAD^1** you undo your most recent commit but retain all the changes the commit made. This is useful if you accidentally included a file in a commit and want to go back, and remake the commit.

# Using Reset

## EXERCISE

- ① Make a new commit with a single line addition to myfile.txt and add a new file called "BIGBINARY".
- ② Make sure you have a clean working directory (no local changes compared to most recent commit) with **git status**
- ③ Use **git show** to show the changes made by your most recent commit. Hopefully this should show the single line addition to myfile.txt and the newly added BIGBINARY file.
- ④ Use **git reset HEAD^1** to undo this commit but keep the single line addition and BIGBINARY in the working directory (Your work is not undone, just the commit).
- ⑤ Use **git status** to see that this line addition is still in your text file, but **git log** no longer shows the commit
- ⑥ Make a new GOOD commit by adding only myfile.txt to the stage and not BIGBINARY.



## Introduction to Git and Version Control

- └ How to Use Git
  - └ Reset
    - └ Using Reset

2021-09-09

Using Reset

EXERCISE

- ① Make a new commit with a single line addition to myfile.txt and add a new file called "BIGBINARY".
- ② Make sure you have a clean working directory (no local changes compared to most recent commit) with **git status**
- ③ Use **git show** to show the changes made by your most recent commit. Hopefully this should show the single line addition to myfile.txt and the newly added BIGBINARY file.
- ④ Use **git reset HEAD^1** to undo this commit but keep the single line addition and BIGBINARY in the working directory (Your work is not undone, just the commit).
- ⑤ Use **git status** to see that this line addition is still in your text file, but **git log** no longer shows the commit
- ⑥ Make a new GOOD commit by adding only myfile.txt to the stage and not BIGBINARY.

# Using Rebase



- **git rebase** rewrites the commit history by starting from specified base commit and choosing what to do with each commit all the way to the current HEAD.
- Pro: Great for removing WIP commits or otherwise meaningless commits. Use it to clean up your local history before pushing to a public repo.
- Con: This has the possibility to create a lot of conflicts if used in a shared repo (as one person's history will differ from another). Generally rebase should not be used to modify any commits you have pushed to a public repo.
- Generally I recommend using **git rebase -i** for beginners as this shows you what is going on each step of the way.

## Introduction to Git and Version Control

- └ How to Use Git
  - └ Rebase
    - └ Using Rebase

2021-09-09

Using Rebase

- **git rebase** rewrites the commit history by starting from specified base commit and choosing what to do with each commit all the way to the current HEAD.
- Pro: Great for removing WIP commits or otherwise meaningless commits. Use it to clean up your local history before pushing to a public repo.
- Con: This has the possibility to create a lot of conflicts if used in a shared repo (as one person's history will differ from another). Generally rebase should not be used to modify any commits you have pushed to a public repo.
- Generally I recommend using **git rebase -i** for beginners as this shows you what is going on each step of the way.



## EXERCISE

- ① Make sure you have at least 5 commits to your **myFirstRepo**.
- ② Making at least one of these a meaningless WIP commit.
- ③ Use **git log** to find the earliest commit that was bad (first WIP commit). Copy the HASH from the commit **just prior** to this WIP commit. We want to rebase our current HEAD not on the WIP commit, but just before it so we can remove or fix the WIP commit.
- ④ Use **git rebase -i <HASH>** and follow the instructions

2021-09-09

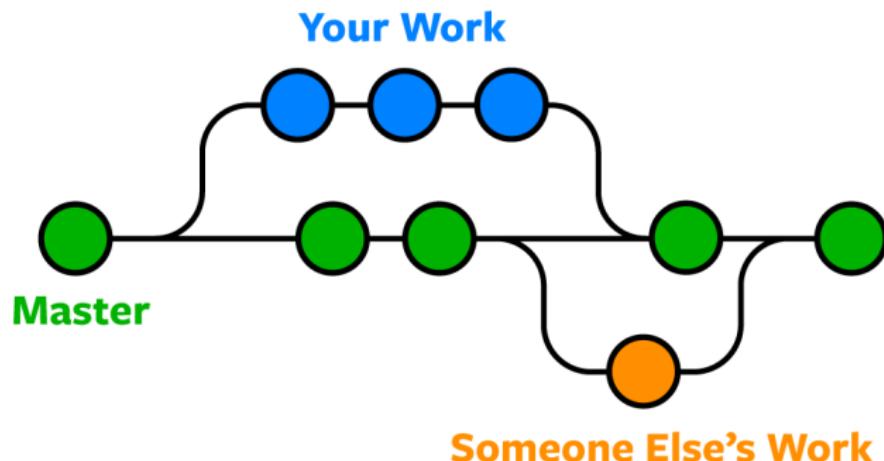
- └ How to Use Git
  - └ Rebase
    - └ Using Rebase

### EXERCISE

- ① Make sure you have at least 5 commits to your **myFirstRepo**.
- ② Making at least one of these a meaningless WIP commit.
- ③ Use **git log** to find the earliest commit that was bad (first WIP commit). Copy the HASH from the commit **just prior** to this WIP commit. We want to rebase our current HEAD not on the WIP commit, but just before it so we can remove or fix the WIP commit.
- ④ Use **git rebase -i <HASH>** and follow the instructions



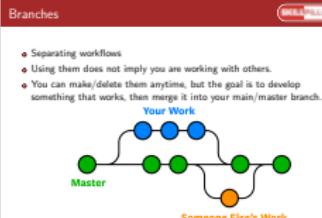
- Separating workflows
- Using them does not imply you are working with others.
- You can make/delete them anytime, but the goal is to develop something that works, then merge it into your main/master branch.



## Introduction to Git and Version Control

- └ How to Use Git
  - └ Rebase
  - └ Branches

2021-09-09



- When you want to develop something separately (New features, refactoring, hacking away) but don't want to affect your main workflow.
- Make a new branch before doing something you think will break the code, or do something you think will be nontrivial (more than one commit).
- With the house example, think separation of the development of garage or porch. Or think about remodeling the interior while someone else works on the exterior.



## EXERCISE

- ① **git branch my2ndBranch** Create a new branch at HEAD
- ② **git checkout my2ndBranch** to check it out
- ③ Make 2 WIP commits (empty files, single line additions, etc).
- ④ **git log --all** or **git log --pretty='format:%C(auto)%h %d %s, %C(green)%ad'** --all --graph --abbrev-commit --notes --date=relative for a prettier log/tree.
- ⑤ **git branch -d my2ndBranch**: Delete a branch (Git warns you)
- ⑥ **git checkout master**: Checkout master branch before deleting.
- ⑦ **git branch -d my2ndBranch**: Delete a branch (Git warns you)
- ⑧ **git branch -D my2ndBranch**: Delete a branch
- ⑨ Shortcut: **git checkout -b my2ndBranch**: Create a new branch @ HEAD and then check it out.

## Introduction to Git and Version Control

- └ How to Use Git
  - └ Rebase
    - └ Branches: Creating and Deleting Them

2021-09-09

Lets take a look at how to create, delete and play around with branches. Note git warns you about a LOT. Its not perfect, but I highly encourage you to read the warnings carefully and understand them before moving on. The initial learning curve is a bit steeper, but it will save you SOOO much hassle trying to fix something you did despite git warning you it is likely a bad idea.

Branches: Creating and Deleting Them

### EXERCISE

- ① **git branch my2ndBranch** Create a new branch at HEAD
- ② **git checkout my2ndBranch** to check it out
- ③ Make 2 WIP commits (empty files, single line additions, etc).
- ④ **git log --all** or **git log --pretty='format:%C(auto)%h %d %s, %C(green)%ad'** --all --graph --abbrev-commit --notes --date=relative for a prettier log/tree.
- ⑤ **git branch -d my2ndBranch**: Delete a branch (Git warns you)
- ⑥ **git checkout master**: Checkout master branch before deleting.
- ⑦ **git branch -D my2ndBranch**: Delete a branch (Git warns you)
- ⑧ **git branch -D my2ndBranch**: Delete a branch
- ⑨ Shortcut: **git checkout -b my2ndBranch**: Create a new branch @ HEAD and then check it out.



## EXERCISE

- ① **git checkout -b my2ndBranch** Create a new branch at HEAD and check it out
- ② Make 2 GOOD commits (single line additions for now).
- ③ **git log --all** or **git log --pretty=format:'%C(auto)%h %d %s, %C(green)%ad'** --all --graph --abbrev-commit --notes --date=relative for a prettier log/tree.
- ④ **git checkout master**: Checkout master branch before deleting.
- ⑤ **git merge my2ndBranch**: Merges my2ndBranch into HEAD (master)
- ⑥ **git merge my2ndBranch --no-ff**: Retains branch-like history. (I prefer this)
- ⑦ **git branch -d my2ndBranch**: Delete a branch (git doesn't warn you)

Introduction to Git and Version Control

- └ How to Use Git
  - └ Rebase
    - └ Branches: Merging Them

2021-09-09

Note you may get merge conflicts. I don't want to spend too much time on these, as it can be a VERY complex topic, but just want to illustrate the basic idea here.

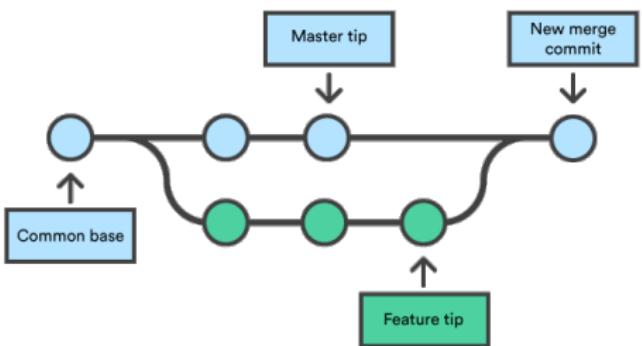
## EXERCISE

- ➊ **git checkout -b my2ndBranch** Create a new branch at HEAD and check it out
- ➋ Make 2 GOOD commits (single line additions for now).
- ➌ **git log --all** or **git log --pretty=format:'%C(auto)%h %d %s, %C(green)%ad'** --all --graph --abbrev-commit --notes --date=relative for a prettier log/tree.
- ➍ **git checkout master**: Checkout master branch before deleting.
- ➎ **git merge my2ndBranch**: Merges my2ndBranch into HEAD (master)
- ➏ **git merge my2ndBranch --no-ff**: Retains branch-like history. (I prefer this)
- ➐ **git branch -d my2ndBranch**: Delete a branch (git doesn't warn you)

# Review

## Git history:

- commits
- git log --all --graph --online
- git checkout
- git reset
- git revert
- git rebase
- git branch
- git merge



## Introduction to Git and Version Control

### How to Use Git

#### Rebase

##### Review

2021-09-09

- commits
- git log --all --graph --online
- git checkout
- git reset
- git revert
- git rebase
- git branch
- git merge

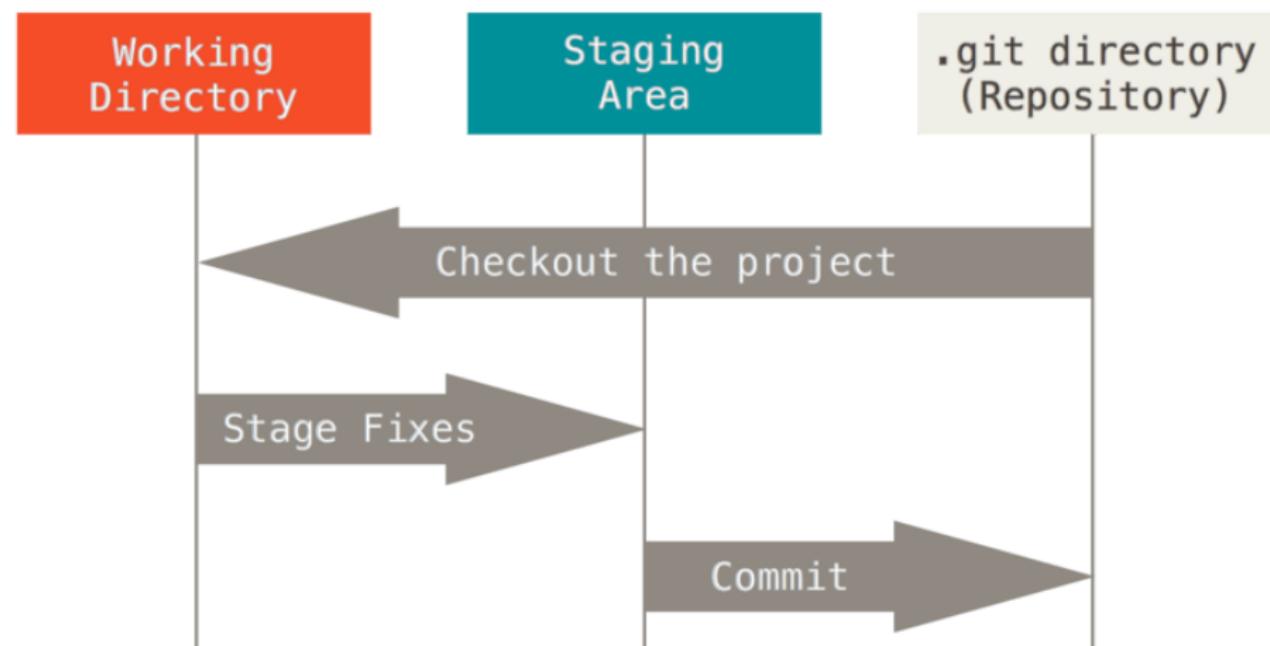




# Review

Three main parts:

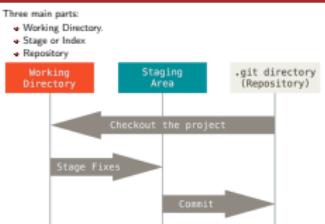
- Working Directory.
- Stage or Index
- Repository

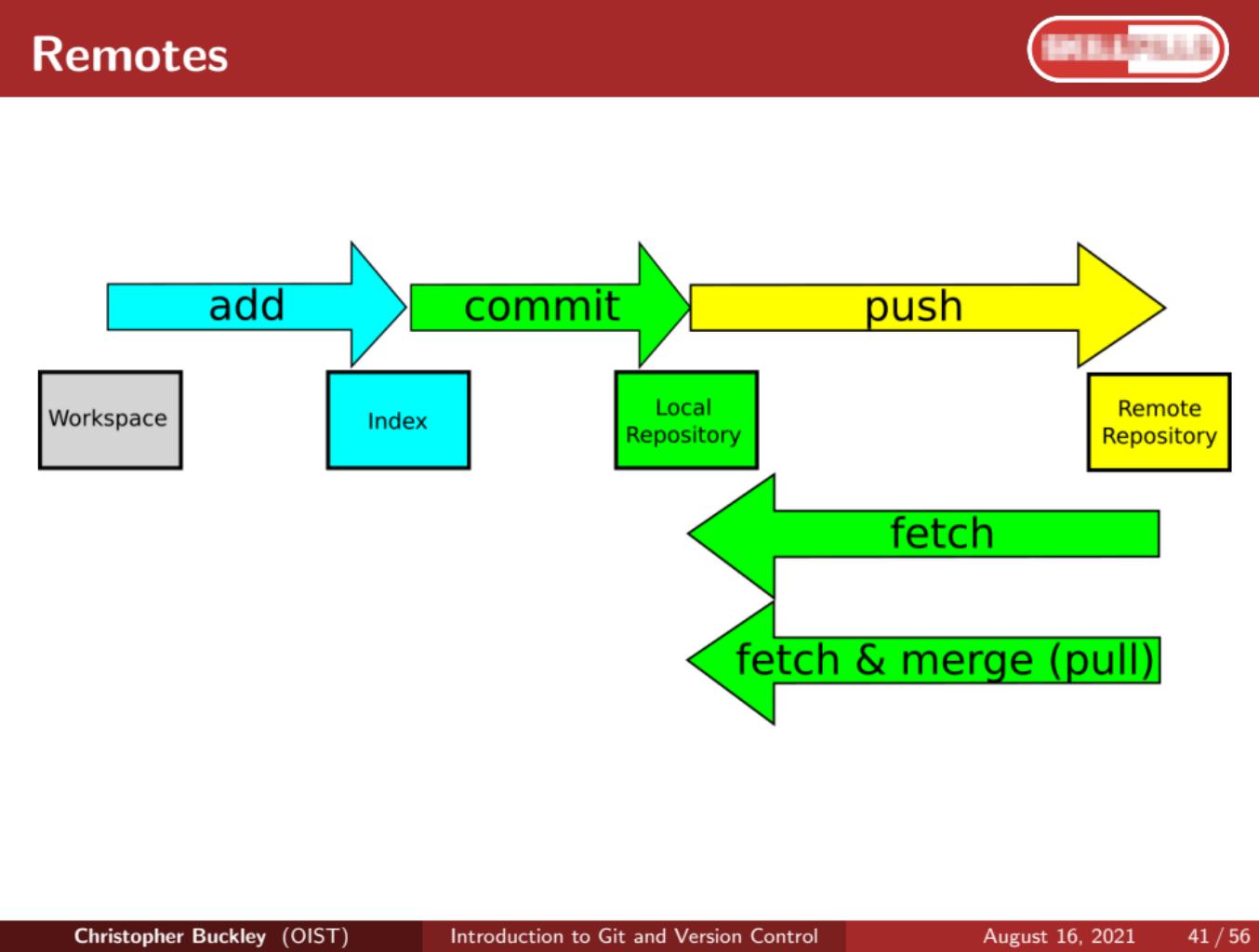


## Introduction to Git and Version Control

- How to Use Git
  - Rebase
  - Review

2021-09-09





2021-09-09

## Introduction to Git and Version Control

- Working Online
- Remotes

A small diagram titled 'Remotes' located in the top right corner. It shows a sequence of four boxes: 'Workspace' (grey), 'Index' (cyan), 'Local Repository' (green), and 'Remote Repository' (yellow). Arrows indicate a linear flow from 'Workspace' to 'Index' to 'Local Repository' to 'Remote Repository'. A green arrow labeled 'push' points from 'Local Repository' to 'Remote Repository'. Another green arrow labeled 'fetch' points from 'Remote Repository' back to 'Local Repository'.

# Your First? Remote

## Introduction to Git and Version Control

### Working Online

2021-09-09

#### ↳ Your First? Remote

Your First? Remote

EXERCISE

- ① Create a new repo using github (e.g. myfirstremote) (<https://docs.github.com/en/get-started/quickstart/create-a-repo>)
- ② Add a README, .gitignore and a license.
- ③ Create a new repo from the command line with an identical name
- ④ Make some simple commits and test the process of **pushing**, **fetching**, and **pulling** stuff from that repo.
- ⑤ Check what is going on with each step via **git log --all --graph --oneline**"

## EXERCISE

- ① Create a new repo using github (e.g. myfirstremote)  
(<https://docs.github.com/en/get-started/quickstart/create-a-repo>)
- ② Add a README, .gitignore and a license.
- ③ Create a new repo from the command line with an identical name
- ④ Make some simple commits and test the process of **pushing**, **fetching**, and **pulling** stuff from that repo.
- ⑤ Check what is going on with each step via **git log --all --graph --oneline**"

# Working with Forks and Remotes



- A fork is a copy (with extras) of some other repo.
- Pushing your changes to a shared repo can cause chaos (unless done in a VERY structured way)
- Forks enable multiple people to freely push/pull their changes without messing up the original repo.
- We want to keep our fork up to date with the original repo so we can merge changes we make to the original repo eventually.
- Keep track of both the original and the fork(s) via **git remote -v**
- Using **git log --all --graph --oneline** we can see the status of each repo. Remotes in RED, branches in GREEN.
- One or a few people act as the gate keeper to the original repo by reviewing and approving/denying pull requests.

## Introduction to Git and Version Control

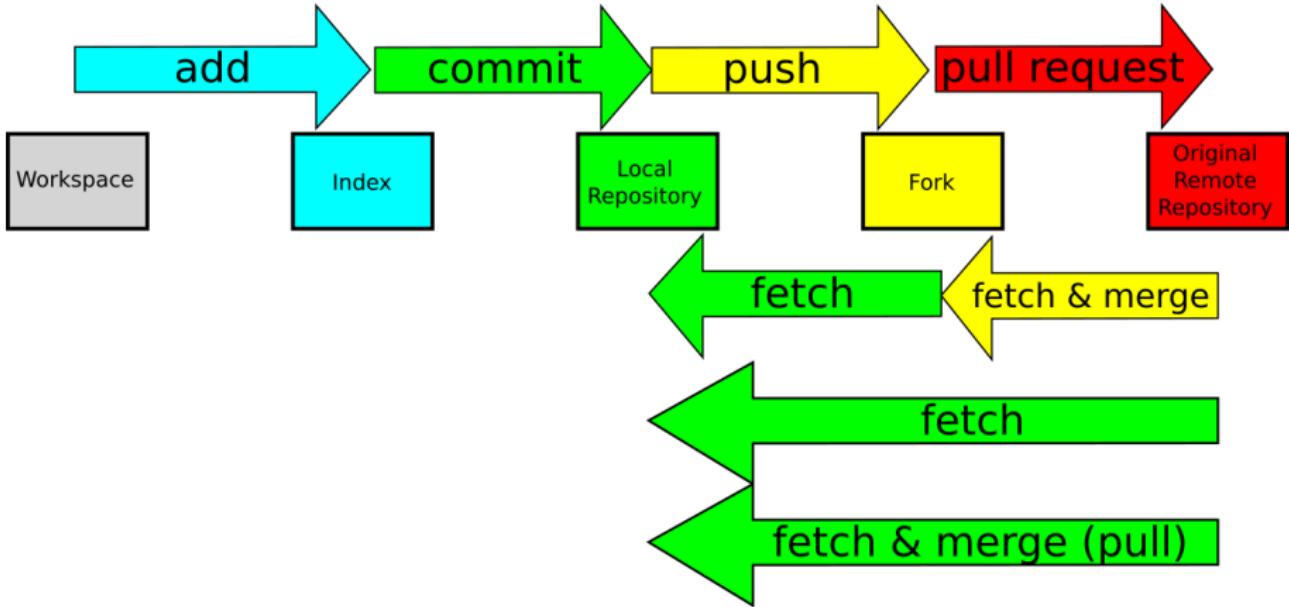
### └ Working Online

2021-09-09

#### └ Working with Forks and Remotes

- A fork is a copy (with extras) of some other repo.
- Pushing your changes to a shared repo can cause chaos (unless done in a VERY structured way)
- Forks enable multiple people to freely push/pull their changes without messing up the original repo.
- We want to keep our fork up to date with the original repo so we can merge changes we make to the original repo eventually.
- Keep track of both the original and the fork(s) via **git remote -v**
- Using **git log --all --graph --oneline** we can see the status of each repo. Remotes in RED, branches in GREEN.
- One or a few people act as the gate keeper to the original repo by reviewing and approving/denying pull requests.

# Forks

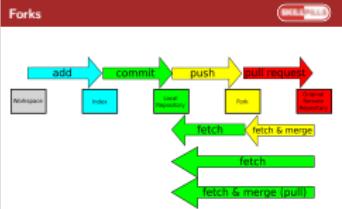


## Introduction to Git and Version Control

### Working Online

#### Forks

2021-09-09



# Cloning Your First Fork



## EXERCISE

- ① Go to <https://github.com/oist/myfirstfork>
- ② Click "Fork" in the top-right and select your account.
- ③ Click Code and clone the repo via HTTPS or SSH. Click the ? if you are unsure which you would like to use.
- ④ In some directory (e.g. /git/) use **git clone**  
**git@github.com:topherbuckley/myfirstfork.git** to clone into your fork.
- ⑤ **cd myfirstfork** and **git status**
- ⑥ **git log --all --oneline --graph**
- ⑦ **git remote -v**
- ⑧ **git remote rename origin fork** and **git remote -v**

## Introduction to Git and Version Control

### Working Online

2021-09-09

### Cloning Your First Fork

#### Cloning Your First Fork

##### EXERCISE

- ➊ Go to <https://github.com/oist/myfirstfork>
- ➋ Click "Fork" in the top-right and select your account.
- ➌ Click Code and clone the repo via HTTPS or SSH. Click the ? if you are unsure which you would like to use.
- ➍ In some directory (e.g. /git/) use `git clone git@github.com:topherbuckley/myfirstfork.git` to clone into your fork.
- ➎ `cd myfirstfork` and `git status`
- ➏ `git log --all --oneline --graph`
- ➐ `git remote -v`
- ➑ `git remote rename origin fork` and `git remote -v`



## EXERCISE

- ① **git remote add oist git@github.com:oist/myfirstfork.git** or **git remote add oist https://github.com/oist/myfirstfork.git**
- ② **git remote -v**
- ③ **git remote update** to download any changes from the remotes.
- ④ **git log --all --oneline --graph** to see how they all overlap.
- ⑤ Allow me to push a new commit.
- ⑥ **git fetch oist** or **git remote update** to fetch these changes
- ⑦ **git merge oist/main** to merge these changes
- ⑧ Note **git fetch oist** followed by **git merge oist/main** is roughly equivalent to **git pull oist main** but this can be hazardous!

## Introduction to Git and Version Control

### Working Online

2021-09-09

### Keeping in Sync with the Original Repo

#### EXERCISE

- ① **git remote add oist git@github.com:oist/myfirstfork.git** or **git remote add oist https://github.com/oist/myfirstfork.git**
- ② **git remote -v**
- ③ **git remote update** to download any changes from the remotes.
- ④ **git log --all --oneline --graph** to see how they all overlap.
- ⑤ Allow me to push a new commit.
- ⑥ **git fetch oist** or **git remote update** to fetch these changes
- ⑦ **git merge oist/main** to merge these changes
- ⑧ Note **git fetch oist** followed by **git merge oist/main** is roughly equivalent to **git pull oist main** but this can be hazardous!



- A pull request is making a request to some github repo to merge in **your** changes.
- Github provides several ways to edit and discuss these PRs before merging.
- Lets take a look at how to make one, and how to interact with them.
- If time allows lets look at using rebase to update our PRs with more current changes in the upstream repo.

## Introduction to Git and Version Control

### └ Working Online

#### └ Pull Requests (PRs)

2021-09-09

- A pull request is making a request to some github repo to merge in **your** changes.
- Github provides several ways to edit and discuss these PRs before merging.
- Lets take a look at how to make one, and how to interact with them.
- If time allows lets look at using rebase to update our PRs with more current changes in the upstream repo.

# Your First? Pull Request

## EXERCISE

- ① Add your name to README and make a commit with this change
- ② **git log --all --oneline --graph** to see this new commit on your local repo. DO THIS AFTER EVERY STEP to see what is happening from each command.
- ③ **git push fork main** (or just **git push** in this case)
- ④ Go to your fork on github, "Contribute" then "Open Pull Request"
- ⑤ Make sure you are requesting to pull changes from the "main" branch of your fork into the "main" branch of the oist repo.
- ⑥ Check your changes in the git diff below. When happy with what you see, "Create pull request"
- ⑦ Add a message to explain/summarize what your PR contains.
- ⑧ Inform me that you've done so and I will merge your changes
- ⑨ **git remote update** or **git fetch oist** after I've merged the changes.

2021-09-09

## Introduction to Git and Version Control

### Working Online

### Your First? Pull Request

Your First? Pull Request

EXERCISE

- Add your name to README and make a commit with this change
- `git log --all --oneline --graph` to see this new commit on your local repo. DO THIS AFTER EVERY STEP to see what is happening from each command.
- `git push fork main` (or just `git push` in this case)
- Go to your fork on github, "Contribute" then "Open Pull Request"
- Make sure you are requesting to pull changes from the "main" branch of your fork into the "main" branch of the oist repo.
- Check your changes in the git diff below. When happy with what you see, "Create pull request"
- Add a message to explain/summarize what your PR contains.
- Inform me that you've done so and I will merge your changes
- `git remote update` or `git fetch oist` after I've merged the changes.

Christopher Buckley (OIST)

Introduction to Git and Version Control

August 16, 2021

48 / 56



- Remember: pull = fetch + merge
- A pull request is no different.
- When you try to merge conflicting changes git requests that you make take the wheel
- Who should resolve the conflicts?
- How to resolve conflicts?

<https://docs.github.com/en/github/collaborating-with-pull-requests/addressing-merge-conflicts/about-merge-conflicts>

### Working Online

#### Merge Conflicts

2021-09-09

- The author of the PR typically should resolve the conflicts as they are the ones who are proposing to CHANGE something from how it was. They should be able to explain WHY it should be changed to the person maintaining the main branch.

- Remember: pull = fetch + merge
  - A pull request is no different.
  - When you try to merge conflicting changes git requests that you make take the wheel
  - Who should resolve the conflicts?
  - How to resolve conflicts?
- <https://docs.github.com/en/github/collaborating-with-pull-requests/addressing-merge-conflicts/about-merge-conflicts>



2021-09-09

- Resolving them on Github (New?)
  - Does this require push permissions? (i.e. PR requestor cannot do this?)
- Resolving them in the terminal
  - Conflicting lines
  - Rebasing

# Conflicting Line Merge Conflicts



Example: A Pull Request from branch-a to main.

If you have questions, please

<<<<< HEAD

[open](#) an issue

=====

ask your question [in](#) IRC.

>>>>> branch-a

From <<<<< HEAD to ===== are the lines that currently  
exist [in](#) the main branch

From ===== to >>>>> branch-a are the lines with  
proposed changes [from](#) branch-a.

## Introduction to Git and Version Control

### Working Online

2021-09-09

#### Conflicting Line Merge Conflicts

##### Conflicting Line Merge Conflicts

Example: A Pull Request from branch-a to main.

If you have questions, please

<<<<< HEAD

[open](#) an issue

[ask your question](#) [in](#) IRC.

>>>>> branch-a

From <<<<< HEAD to ===== are the lines that currently  
exist [in](#) the main branch

From ===== to >>>>> branch-a are the lines with  
proposed changes [from](#) branch-a.



- Manually remove the lines you don't want and keep those you do.
- Remove all git markup
- **git add myFile**
- **git commit** to conclude merge commit
- **git push** to push resolving changes
- PR is automatically updated to reflect the tip of the branch you are trying to merge. Thus conflicts should be gone.

### └ Working Online

#### └ Conflicting Line Merge Conflicts

2021-09-09

- 

- Manually remove the lines you don't want and keep those you do.
- Remove all git markup
- **git add myFile**
- **git commit** to conclude merge commit
- **git push** to push resolving changes
- PR is automatically updated to reflect the tip of the branch you are trying to merge. Thus conflicts should be gone.

# Refactored Merge Conflicts



Sometimes it can be easier to understand what changed if you do a rebase rather than a merge. e.g.:

```
-/git/myfirstfork$ gt
*   3b24005  (HEAD -> oistMain, oist/main) Merge pull request #5 from topherbuckley/main, 48 seconds ago
|\ 
| * 8f5dc70  (fork/main, fork/HEAD, forkMain) Changed inputs to floats rather than ints, 11 minutes ago
|/
* f9df705  refactored c->e and d->f, 8 minutes ago
*  51cb2f4  Merge pull request #4 from topherbuckley/main, 13 minutes ago
|\ 
| * 7a41840  Merge remote-tracking branch 'oist/main' into main, 14 minutes ago
| |
| |
|/
*  4340344  refactored a->c and b->d, 19 minutes ago
| * 5bdaa7b  Fixed * instead of +, 21 minutes ago
|/
* 9e0e43e  Initial commit of onePlusOne.py, 22 minutes ago
* cc7608c  Added last line to README, 19 hours ago
* c4fc768  Added another line to readme, 7 days ago
* 58aa063  a new commit, 8 days ago
* ef76ffd  Added Christopher to README, 8 days ago
* 9d6db44  Update README.md, 8 days ago
* fe2f804  Initial commit, 8 days ago
```

## Introduction to Git and Version Control

### Working Online

#### Refactored Merge Conflicts

2021-09-09

- Let me demonstrate what this looks like before we move to an exercise.

Refactored Merge Conflicts

Sometimes it can be easier to understand what changed if you do a rebase rather than a merge. e.g.:

```
git clone https://github.com/topherbuckley/oist.git
cd oist
git checkout main
git pull origin main
git merge -X theirs oist/main
git push -u origin main
```

Initial commit of onePlusOne.py, 22 minutes ago

Added last line to README, 19 hours ago

Added another line to readme, 7 days ago

a new commit, 8 days ago

Added Christopher to README, 8 days ago

Update README.md, 8 days ago

Initial commit, 8 days ago

# Rebasing a PR



## Excercise

- ① Note: This exercise has caused CHAOS in the past, so no gaurantees.
- ② Add a line to the end of README
- ③ Make a commnit and push it to your fork
- ④ Make a PR with this new commit
- ⑤ Wait for me to create another commit in oist/main before proceeding
- ⑥ Refresh your PR to notice the conflict
- ⑦ Fetch oist changes locally
- ⑧ **git rebase oist/main** and resolve the conflicts locally
- ⑨ **git rebase --continue**
- ⑩ **git push -f**
- ⑪ Note PR now is conflict free.

## Introduction to Git and Version Control

### Working Online

2021-09-09

### Rebasing a PR

Rebasing a PR

Exercise

- ➊ Note: This exercise has caused CHAOS in the past, so no gaurantees.
- ➋ Add a line to the end of README
- ➌ Make a commit and push it to your fork
- ➍ Make a PR with this new commit
- ➎ Refresh your PR to notice the conflict
- ➏ Fetch oist changes locally
- ➐ git rebase oist/main and resolve the conflicts locally
- ➑ git rebase --continue
- ➒ git push -f
- ➓ Note PR now is conflict free.

# What it will feel like...

- git is not intuitive to start with, but it's a powerful tool for storing and restoring history, and working collaboratively with other people.
- The more you use it, the more you will like it. Think Stockholm syndrome.
- Operations that you use frequently will become easy.
- Operations you use infrequently, you can Google!



## Introduction to Git and Version Control

### └ Wrap Up

#### └ What it will feel like...

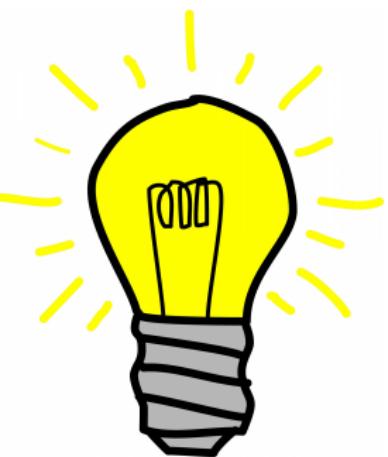
2021-09-09

- git is not intuitive to start with, but it's a powerful tool for storing and restoring history, and working collaboratively with other people.
- The more you use it, the more you will like it. Think Stockholm syndrome.
- Operations that you use frequently will become easy.
- Operations you use infrequently, you can Google!





- git is weird. It's not intuitive, but it's the best way to collaborate with people on open projects.
- It's also great even if you don't collaborate!
- Whenever you are using git, think about other people and how they will perceive your comments. **Would you be able to understand your own cryptic commit messages?**
- You will make mistakes. Don't worry about it. Your entire history is backed up already. Learn from your mistakes and don't make them again!
- Read error messages carefully - they can be useful/informative/instructive.



### └ Wrap Up

#### └ Final Comments

2021-09-09

- git is weird. It's not intuitive, but it's the best way to collaborate with people on open projects.
- It's also great even if you don't collaborate!
- Whenever you are using git, think about other people and how they will perceive your comments. Would you be able to understand your own cryptic commit messages?
- You will make mistakes. Don't worry about it. Your entire history is backed up already. Learn from your mistakes and don't make them again!
- Read error messages carefully - they can be useful/informative/instructive.

