Introduction to Git and Version Control

Zhou Jiming

March 13, 2021

西北工業大學
NORTHWESTERN POLYTECHNICAL UNIVERSITY

Slides forked from GIT

Overview I

- Part 1: Why Git?
  - Traditional vs. Git Versioning
- Part 2: What is Git
  - Three Main Parts (Working Directory, Stage, Repository)
- Part 3: How to Use Git
  - The Terminal
  - Create a repo
  - Commits
  - The Stage (Staging Area, or Index)
  - Making Commits
  - Checking Out Past Commits
- Part 4: Working Online
- Part 5: Wrap Up
- Part 6: Extra Credit
  - Revert

Overview II

Rebase

- Part 7: Recap
- Part 8: Working with Remotes
  Remotes
  Branches
  Merging
- Part 9: Pull Requests on GitHub
- Part 10: More Advanced Topics
  Rebasing and Rewriting history

Part 1: Why Git?

Why Git?

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

Before

After

Why Git?


T
30
TUTUCU
TAKIMLARI
KESICI
TAKIMLARI
ÖLCÜ
ALETLERI
KESME
SIVILARI
CNC
KATERLERI
SERT
MADEN
UÇLARI
SERT
MADEN
FIREZELERI

Traditional vs. Git Versioning

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

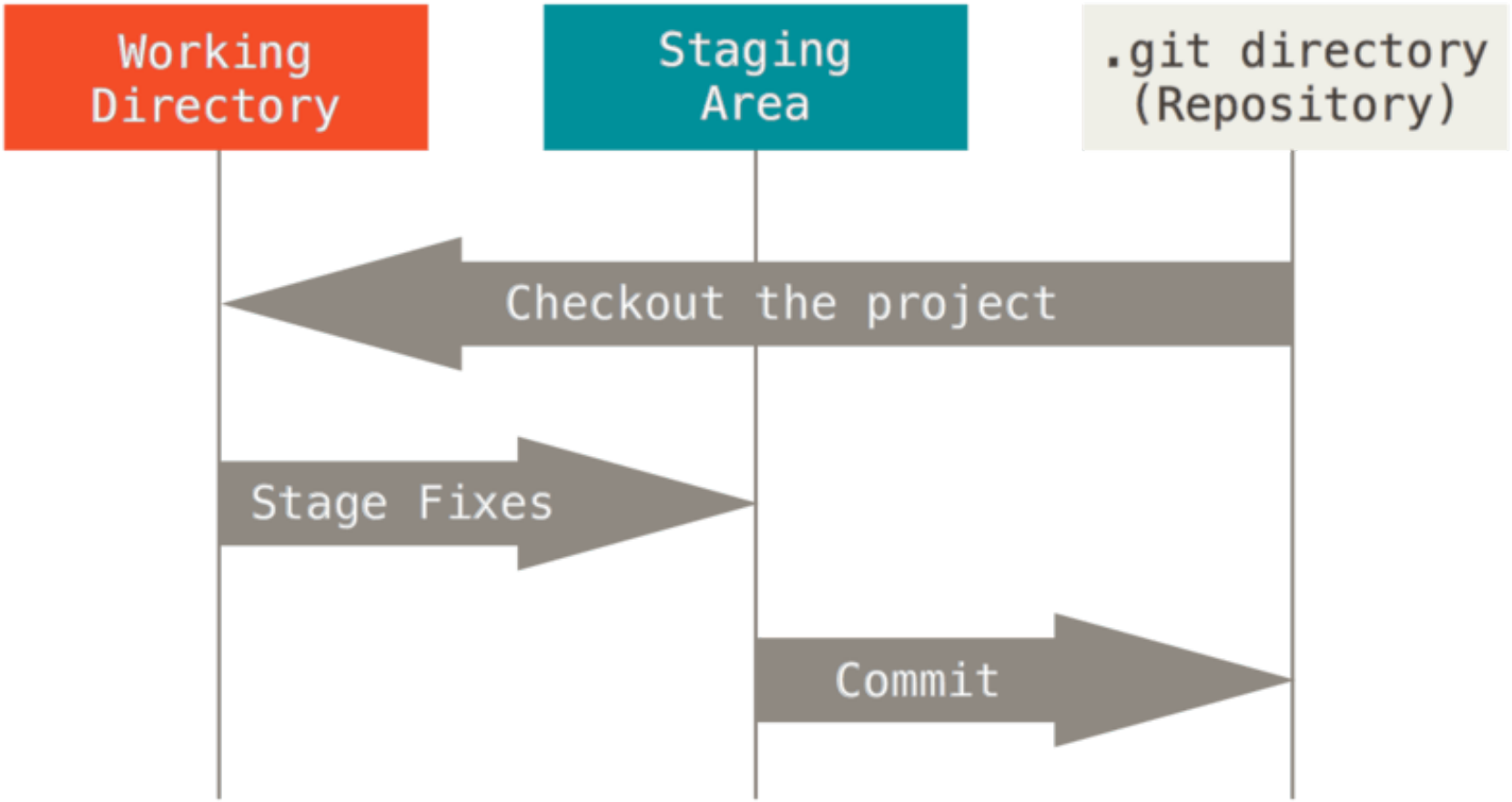
```
christopher@christopher-ThinkPad-W541:~/git/mythesis$ ls -gtr
total 4
-rw-rw-r-- 1 christopher 51 Nov 17 18:22 thesis
-rw-rw-r-- 1 christopher  0 Nov 18 12:07 thesis_v1
-rw-rw-r-- 1 christopher  0 Nov 18 12:07 thesis_v2
-rw-rw-r-- 1 christopher  0 Nov 18 12:07 thesis_v3
-rw-rw-r-- 1 christopher  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_finalfinal
christopher@christopher-ThinkPad-W541:~/git/mythesis$
```

```
christopher@christopher-ThinkPad-W541:~/git/mythesis$ git log --reverse
commit 839a476257310df071ac8290cdefc64a60b86944 (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 a83135a08c1343723a7973a879e2431008b5a466
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 98c17b7472ef14bd75804d4dddb990de92f211ba
Author: Christopher Buckley <15166572+topherbuckley@users.noreply.github.com>
Date:   Tue Nov 17 18:21:31 2020 +0900

    Added initial conclusions

commit aa3034ffb24084d3f95e37ae222f265da07d3590
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 d918fbfe436cf474a34c6cde86ccf845f63e9cb4 (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
```

Part 2: What is Git

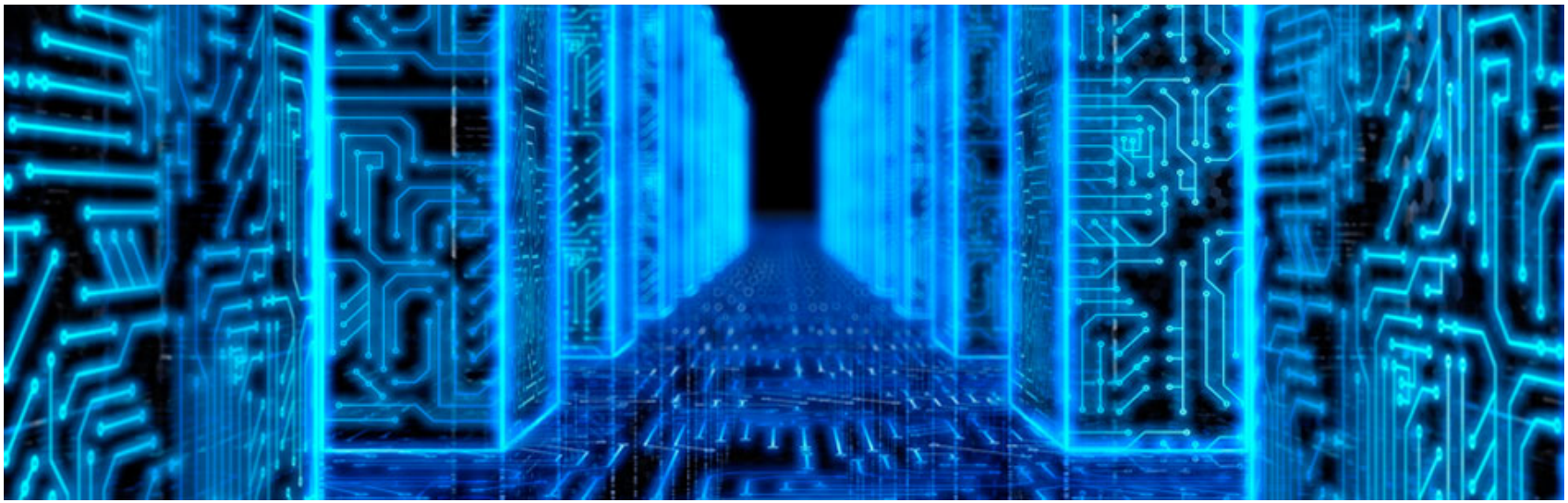
What is git?

- 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.



Repositories

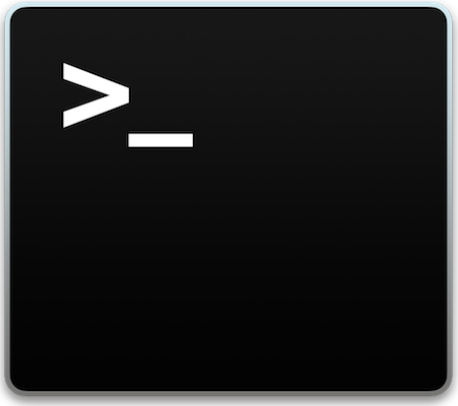
- A **repository** is a container for both your project data and all the items that allow interactions with git commands.
  - There are many sites to host your repository on (github, bitbucket), including your own local machine.
  - All of the essential parts of your repository can be found in the **.git**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



Part 3: How to Use Git

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**
- There is a cheatsheet for Git available here: **https://www.git-tower.com/learn/cheat-sheets/git**



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.

Create an Empty Repo

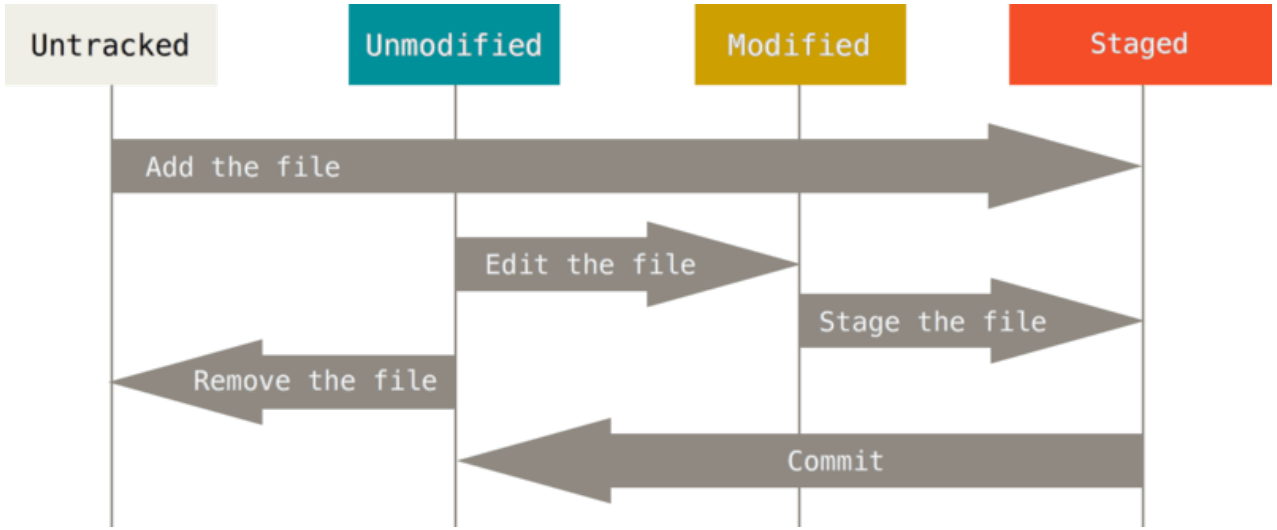
EXERCISE

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 using the **staging area** the more helpful they are in the future.

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

Currating 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 emtpy file **myfile.txt**
2. Check the status of everything with **git status**
3. Add **myfile.txt** to the stage vis **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?

Committing from the Stage

- 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.

Making Commits

EXERCISE

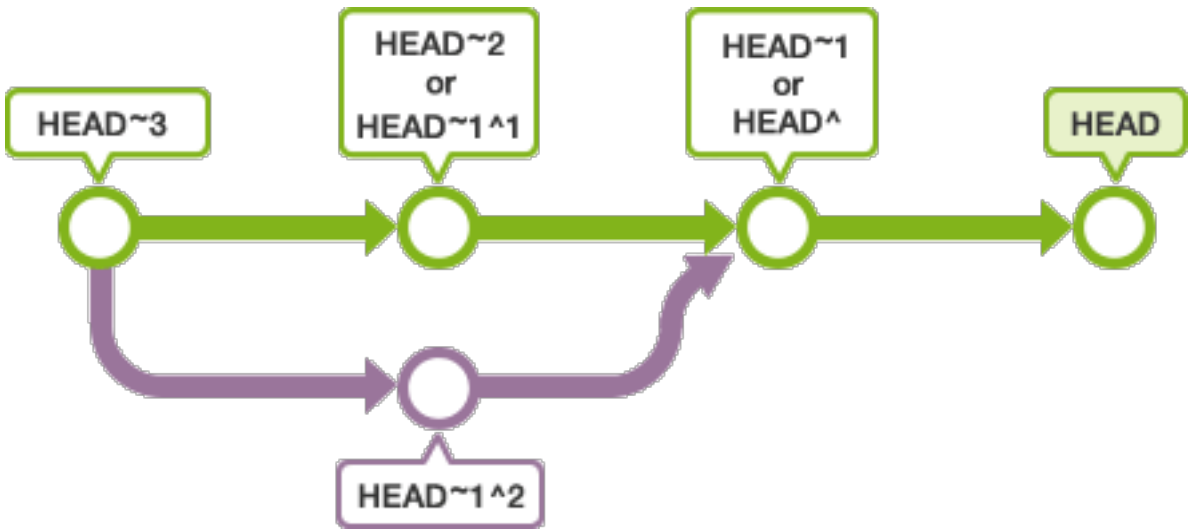
1. Repoen your **myFirstRepo** from before
2. Add the **myFile.txt** back to the stage with **git add myFile.txt**
3. Check the status of the stage with **git status**
4. Once satisfied with what is in the stage and you're ready to commit, go ahead and do so with **git commit** to add your new file to 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.

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

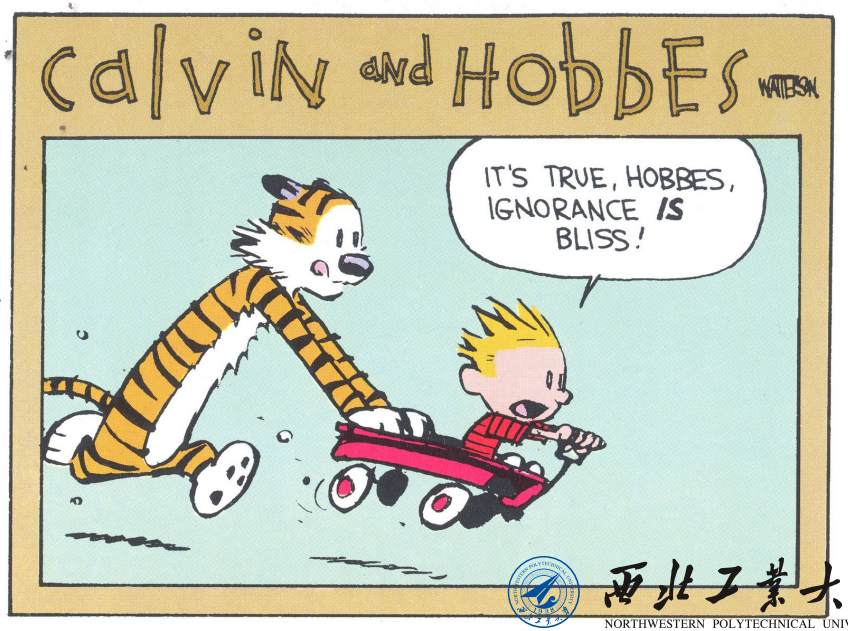
EXERCISE

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.)

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
*.lvlps
```



Quick Exercise

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**

Part 4: Working Online

git with it!

Now we move to the fun* stuff: working with **online repositories**.

- For this, we will be using **github**.
- We'll begin by creating a GitHub repository using the website.
  - If we're working on a project that's already hosted on a remote Git server, we can skip this step.
- Next, we use **git clone** to download a copy.
- From here, you can do the following:
  - **git push** to push any changes you may have to the online repository.
  - **git pull** to take any changes from the repository.

*Here, the word *fun* is subject to interpretation.

Quick Exercise

EXERCISE

1. Fork the **https://github.com/oist/skillpill-git** repository using a browser.
2. Clone the forked repository* to your local disk:

```
git clone
    git@github.com:<git_user_name>/skillpill-git.git
```

or

```
git clone
    https://github.com/<git_user_name>/skillpill-git.git
```

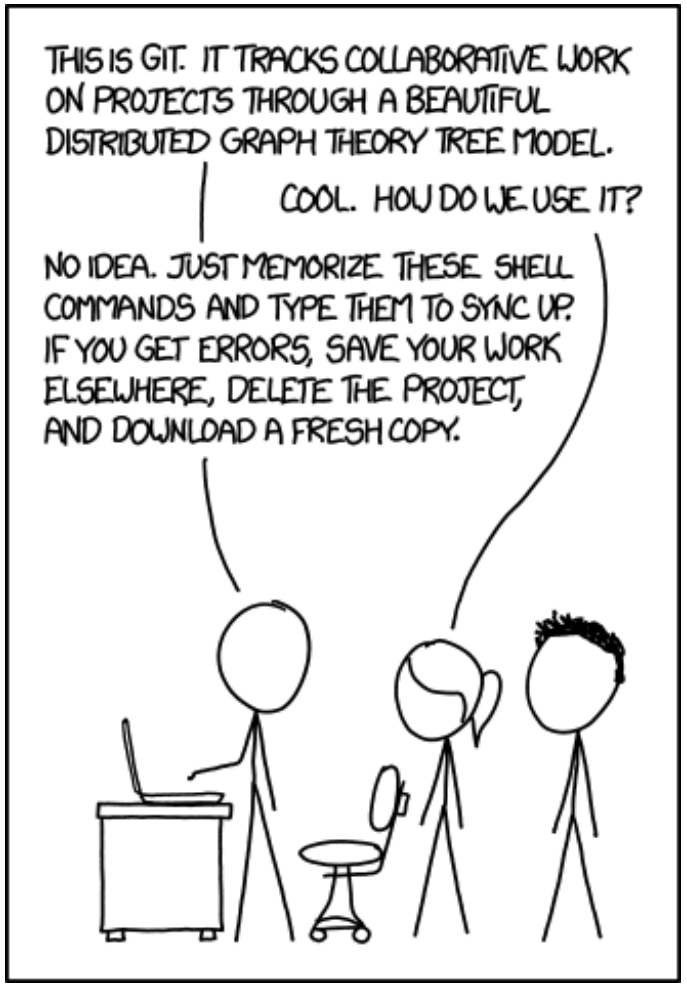
3. Make some simple commits and test the process of **push**ing and **pull**ing stuff from that repo.

*The examples here show cloning the SkillPill Git repository - replace the links as appropriate!

Part 5: Wrap Up

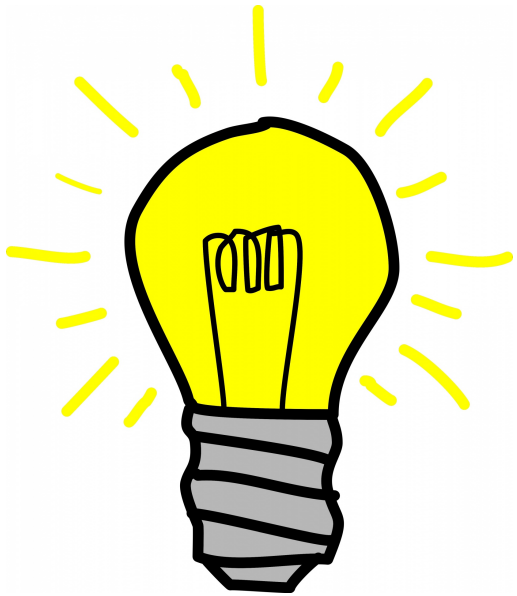
What it will feel like...

- 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!



Final Comments

- 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.



Part 6: Extra Credit

(If Time Allows) 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 distructive than others)
- Non-destructive: (Leaves history intact)
  - **revert**
- Potentially destructive: (Changes history)
  - **rebase**
- Danger Zone: (Can erase history)
  - **reflog**

Using Revert

- Revert makes a new commit showing that you reverted a previous commit.
- Pro: This is very useful for public repos (next session) where you want to show exactly what you've undone to others.
- Con: This can make your commit history quite messy if used too often.

EXERCISE

1. Make a few commits to your **myFirstRepo** or just use the existing history of your skill-pill fork
2. Find a commit you want to revert using **git log** and **git show** or **git diff**
3. Revert that commit with **git revert ⟨HASH⟩** or **git revert HEAD~⟨#⟩**

Using Rebase

- 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 possiblity 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 on any commits you have pushed to a public repo.

EXERCISE

1. Make a few commits to your **myFirstRepo** or just use the existing history of your skill-pill fork
2. Making a meaningless WIP commit
3. Use **git rebase -i ⟨HASH⟩** and follow the instructions

Part 7: Recap

Recap

Last week we covered (don't forget to prefix with **git**):

- **clone** : Cloning a repository into a new directory
- **add** : Add file contents to the *index*. This makes git track the file.
- **commit** : Record changes. Store the staged files as a new part of the history!
- **push** : Update remote *refs* and objects.

There's also a **pull** command.

- **pull** : Update from a *remote*. Technicially a combination of **fetch** and **merge** by default.

Some definitions/descriptions

remote	Another git repository. We used GitHub to provide this.
index	A single, large, binary file listing all files in the current branch with some extra information. Reflects the "proposed next commit"
refs	Short for references. Can point to almost anything in git.

Tips so far...

- You can use `git help <command>` or `git <command> --help` to get information about a command, like `clone`.
- `git add -p` uses *patch mode* to interactively add parts of a file. **-i** is interactive without patch mode.
- `git rm` can be used to remove files from the index (and optionally working directory), whilst `git mv` can help you move files within the repository.
- `git commit --amend` opens an editor to alter the previous commit's message. Don't do this if you already **push**ed the commit!

Reset and Checkout

We also considered **reset** and **revert**.

- Reset is a fairly complicated tool, which modifies the three 'trees' we have briefly mentioned/considered - `HEAD` (your last commit), `index` (the staging area) and the `working directory`.
- If you're interested to know more about this tool, there is a long and informative guide at `https://git-scm.com/book/en/v2/Git-Tools-Reset-Demystified`.
- This content is really beyond the scope of our Skill Pill. :(

Part 8: Working with Remotes

Remotes

Last week we introduced **GitHub**. GitHub is a service that offers you a solution to remotely store your repositories.

- Git is Distributed Version Control System (DVCS). Every copy of your repository, may it be remote or local, is independent of each other. There is no central master repository.
- In order to synchronize these distributed copies we introduce the concept of a remote.

git **remote**

- There can be as many remotes as you want each with different names. When you clone a repository there will be one default remote called **origin**.

Branches

Since there git is decentralized there is no one state of the repository that is correct. To manage this complexity git has the notion of a branch.

- git **branch** Manages branches.
- git **checkout** Switch between branches.

- Most repositories have a default branch called **master** (this is changing to "main" in some cases). Branches are just names for points in the history.
- Once we start working with branches we have to ask ourselves how are we going to join them back up? We can do this by performing a merge.
- You can also associate a local branch with a remote branch by setting it as upstream. git push -**u**.

Exercise

1. Create a new branch, based of master
2. Add a few commits to your branch
3. Change back onto master
4. Check the contents of the file(s) you changed on your other branch whilst you're on the master branch

Merging - An Introduction

- We perform *merges* to "join two or more development histories together".
- It is most commonly performed invisibly by `git pull` and performs by default a "fast-forward" merge.
- We usually see this first when we try to pull some changes and we cannot perform a fast-forward merge.
- In that case, we have to resolve the *merge conflict*.

Merging

Merging is the act of joining two branches together or to join two different branches. You will always merge from a branch/remote into a branch.

- git **fetch** Gets remote changes
- git **merge** Merge changes (ff by default)
- git **add** Resolve merge-conflict

Options for merge:

–no-commit Performs the merge, but doesn't commit yet. Gives you a chance to edit the merge commit.

–ff-only Aborts when we can't perform a fast-forward merge.

–abort Aborts current conflict-resolution and reset to previous state.

You can visualize your history in many different ways, but a nice way from the command line is

Exercise

1. **Clone** a fork of the repository at `https://github.com/oist/skillpill-git` (you may have this available from last week)
2. **Checkout** the 'merge-main' branch
3. **Merge** the 'merge-AddNameToGreeting' branch. Optionally use "–no-ff" to force a merge commit. This will succeed without conflict.
4. Attempt to merge the 'merge-TimeOfDayGreeting' branch. This will cause a merge conflict!

Merge conflict contents

```
def main(username, timeValue):
<<<<<<< HEAD
    print ("Hello " + username)

def callFctn(args):
    if len(args) > 1:
        username = args[1]
    else:
        username = "World"

    timeValue = ""
=======
    greeting = getGreeting(timeValue)
    print (greeting + " " + username)

def callFctn(args):
    username = "World"

    if len(args) > 1:
        try:
            timeValue = time.strptime(args[1], "%H:%M")
        except:
            timeValue = ""
    else:
        timeValue = ""

>>>>>>> merge—TimeOfDayGreeting
```

Exercise continued

5. Use a text editor to resolve the conflict
6. Commit the resolved file (don't forget to **add**)
7. Push your branch to your forked repository

This brings us on to "Pull Requests"...

Part 9: Pull Requests on GitHub

Pull Requests

- Pull Requests are a GitHub-specific feature (also implemented on other platforms, but not a git feature) used to allow contributing code to a repository.
- They are typically used when you don't have write access to a repository
- They can also be used to allow review of your code, perhaps by a coworker, even if you could directly push your changes
- Without using extensions, you must use the website to use them

Demo + Exercise

- Demonstration...
- Practice:
  1. In the last exercise we pushed commits to forks of the OIST repository
  2. Open a pull request on GitHub against the original repository

Part 10: More Advanced Topics

Rewriting History

Rebases are a way to create fast-forward merges, by altering history. Each branch has a root commit from which it diverged from the original commit. By rebasing we change this root. This has a couple of side effects.

- Linear commit history.
- No merge commits within a branch.
- commit-ids change.

- git **pull –ff-only** Don't merge if there are conflict with the remote
- git **rebase** Perform a rebase
- git **rebase -i** Perform a interactive rebase
- git **push -f** Force push your changes
- git **pull –rebase** Perform a pull with a rebase

Exercise

1. create a branch, with some commits
2. go back to master and do some additional work
3. rebase your branch onto master
4. merge your branch onto master

Secrets! I

Stash

When you are moving between branches you sometines want to keep your non-commited changes associated with the branch you where doing them one.

- git **stash**
- git **stash pop**

- git commit –**amend** Amend the last commit.
- git add -**i** Interactive add
- git add -**p** Interactive add in patch mode.
- git **rm** Removes file.
- git **mv** Move file within repository

Secrets! II

Autosquash

- git config rebase.autosquash true
- git commit –**squash**=some-hash
- git commit –**fixup**=some-hash

Autosquash will reorder the commits appropriatly before you perform a git **rebase -i**.

Blame

Secrets! III

There is no such thing as *good* code. If you are using git with people, chances are that something will break at some time and you need someone to blame. That's what git blame is for:

```
git blame -L 1,3 file
```