Introduction to Git and Version Control

Lecture 1: Git ready!

Christopher Buckley

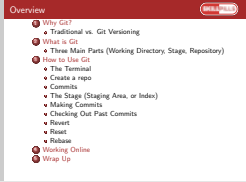
Okinawa Institute of Science and Technology

August 16, 2021

Original Slides by James Schloss, 2016

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

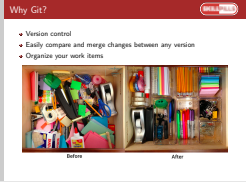


- Git is different from github
- Why Git? I'll tell you what my motivations are, but what are your motivations for being here?

Why Git?

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

Before | After



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?

Why Git?

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

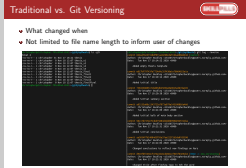
- How to access it safely
- Can't remove it without fearing everything will fall

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$
```

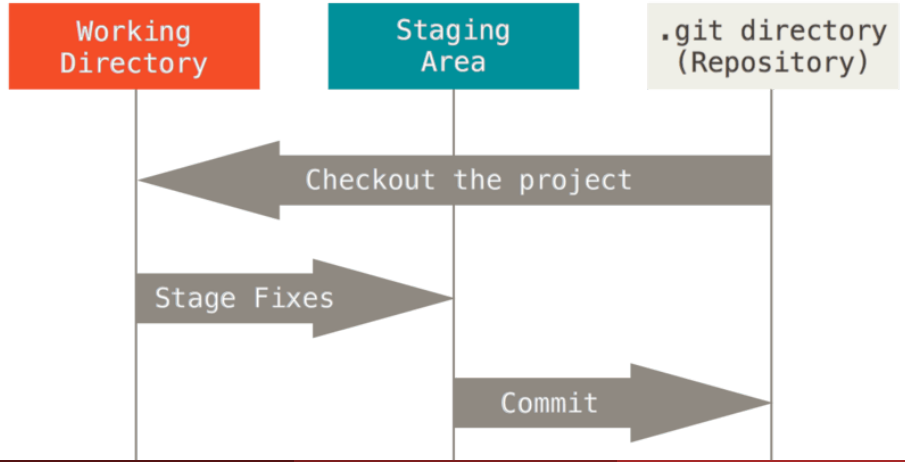
```
christopher@christopher-ThinkPad-W541:~/git/mythesis$ git log --reverse
commit 839a47625710dfd71ac8290cdefc64a60b8d944 (master)
Author: Christopher Buckley <15166572+topherbuckley@users.noreply.github.com>
Date:   Tue Nov 17 18:14:52 2020 +0900

    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 758455880517cbdd5db25054e0e9815ed67da185
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 98c17b7472ef14bd75804d4ddb990de92f211ba
Author: Christopher Buckley <15166572+topherbuckley@users.noreply.github.com>
Date:   Tue Nov 17 18:21:31 2020 +0900

    Added initial conclusions

commit aa3034ffb24084d3f95e37ae222f265da87d3590
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
```

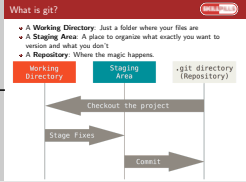



This is just a VERY limited toy example, but to just want to show how I versioned in my master's thesis compared to now using git. We'll get to the actual useful stuff later.

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.



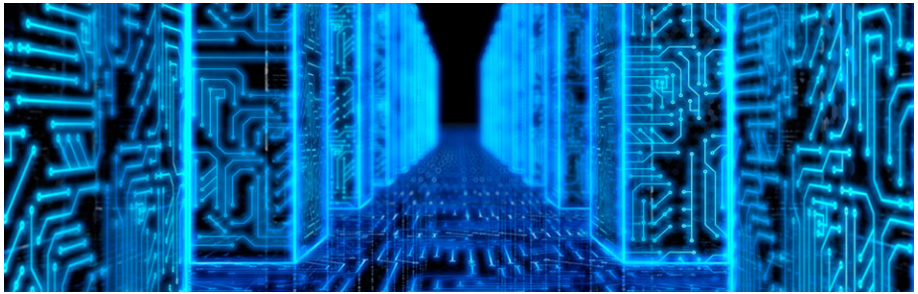


This chart might not make much sense now, but I hope it will before the end of these slides

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

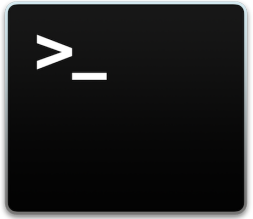
- A **repository** is a container for both your project data and all the items that allow interactions with git commands.
  - There are many sites to host your repository on (github, bitbucket), including your own local machine.
  - All of the essential parts of your repository can be found in the **.git**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

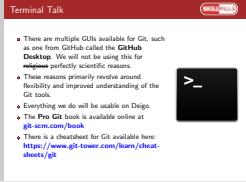


Repositories
- A **repository** is a container for both your project data and all the items that allow interactions with git commands.
  - There are many sites to host your repository on (github, bitbucket), including your own local machine.
  - All of the essential parts of your repository can be found 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**
- There is a cheatsheet for Git available here: **https://www.git-tower.com/learn/cheat-sheets/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.

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.
git

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.

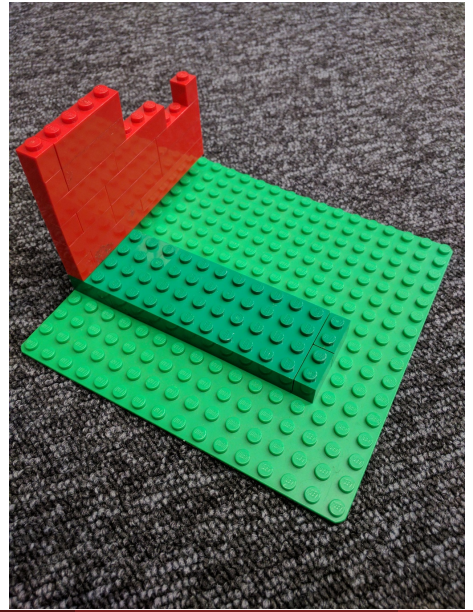
Create an Empty Repo
EXERCISE
1. Open a terminal
2. Create a new directory called **myFirstRepo** and enter it.
3. This is your Working Directory. Thats it!
4. Run **git init** in your Working Directory.
5. Take a peak in the newly created .git directory but don't touch anything quite yet.

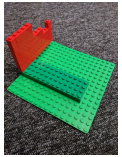
Commits

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

- 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).

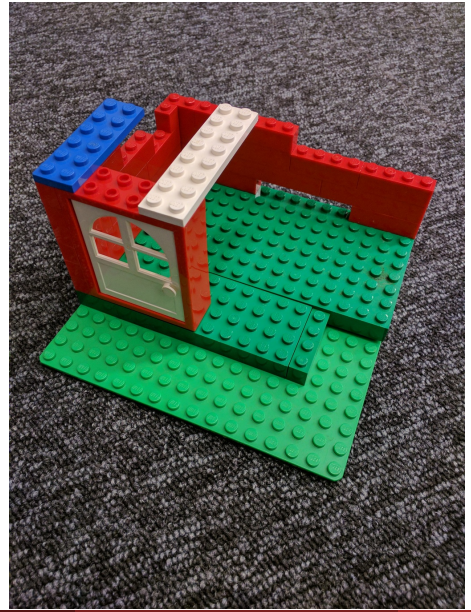
Commits



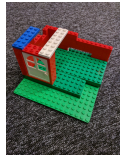


Added some stuff

Commits

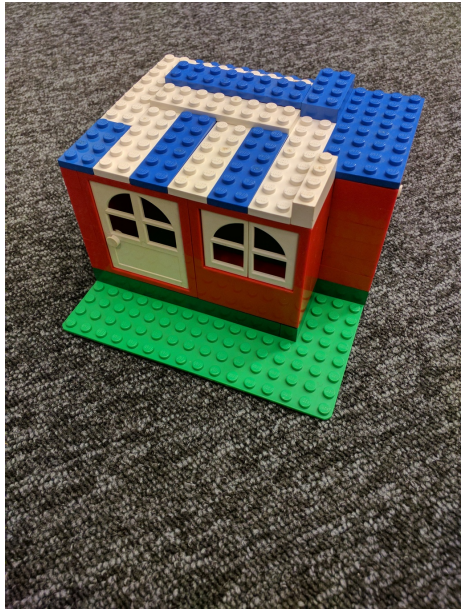


Commits



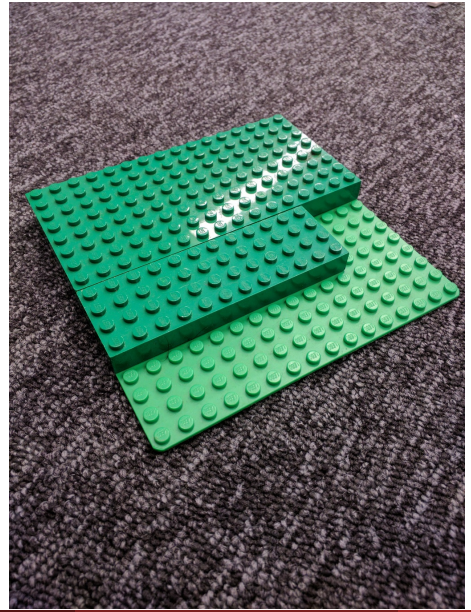
Added some more stuff

Commits

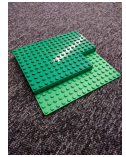


Done! Now lets say we found an error in the western wall. Which commit was that added in?

Commits

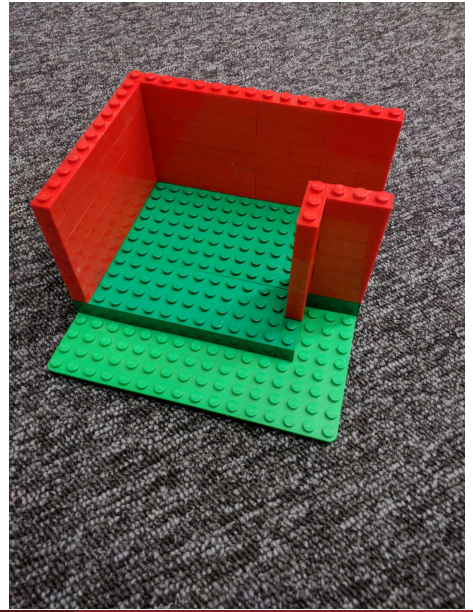


Commits

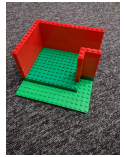


Added the foundation

Commits

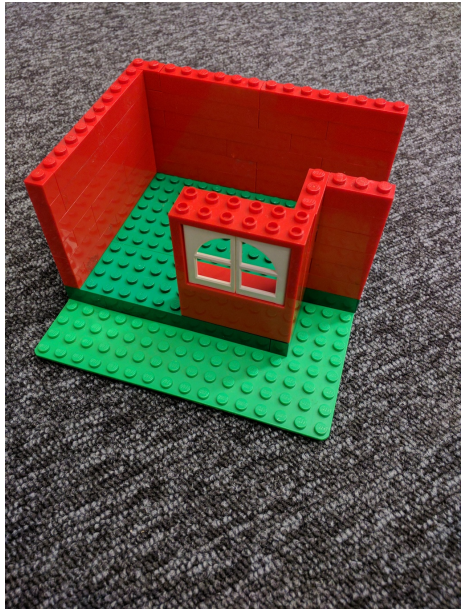


Commits



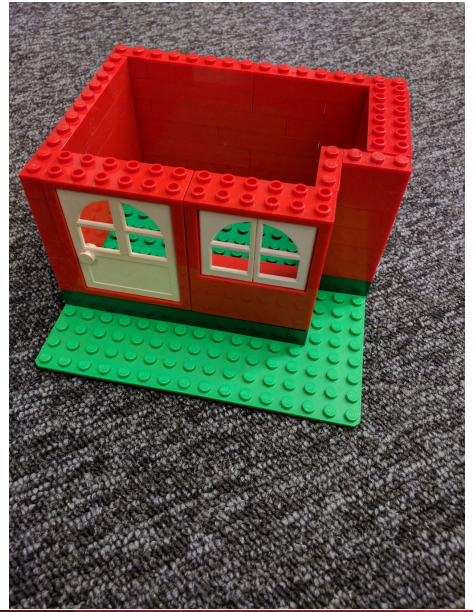
Added the walls

Commits



Added the window

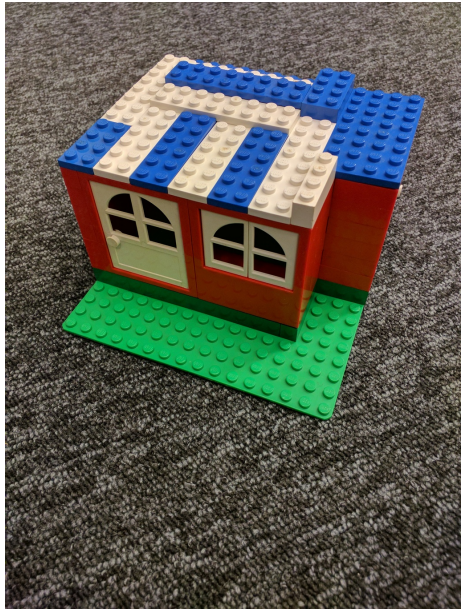
Commits



Commits

Added the door

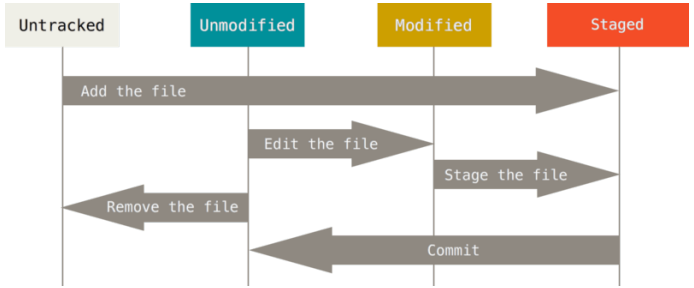
Commits



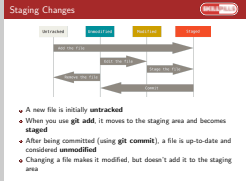
Commits

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



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.

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>
```

- As you saw a few slides ago. Adding messy or unnecessary commits doesn't help anyone. Lets discuss how to make meaningful helpful commits using the stage. There are various ways to fix bad commits, but it takes a whole lot less work to do it right the first time.
- Git works best with the workflow Prof. Doya often and rightly reminds of; start with something simple and work our way up.

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?

Try out the Stage
EXERCISE
Create a new emtpy file **myfile.txt**
Check the status of everything with **git status**
Add **myfile.txt** to the stage vis **git add myfile.txt**
Check the status of everything again with **git status**. What changed?
Unstage the changes with **git reset myfile.txt**
Check the status of everything again with **git status**. What changed?

Committing from the Stage

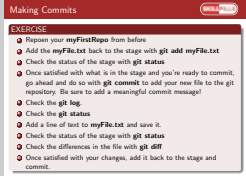
- Git keeps track of **commits**. Check these commits with **git log**. There's plenty of options to show only what you want or everything under the sun.
- **git status** checks any changes since the last commit.
- **git commit** commits everything in the *staging area* - git status shows these files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

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.

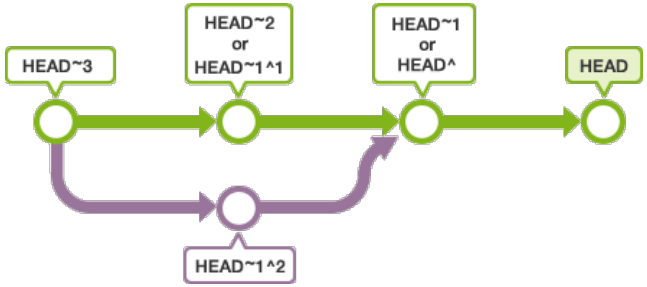


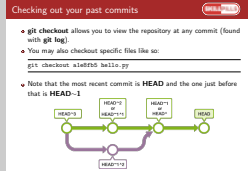
Checking out your past commits

- **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**

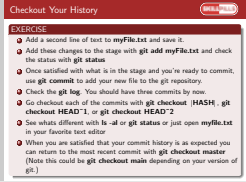




Checkout Your History

EXERCISE

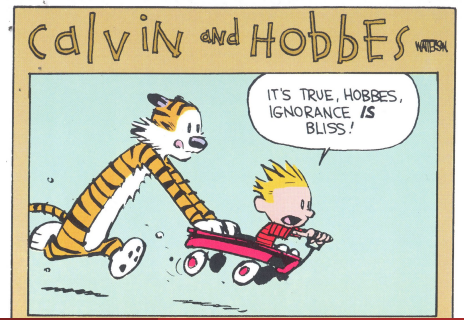
1. Add a second line of text to **myFile.txt** and save it.
2. Add these changes to the stage with **git add myFile.txt** and check the status with **git status**
3. Once satisfied with what is in the stage and you're ready to commit, use **git commit** to add your new file to the git repository.
4. 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.)

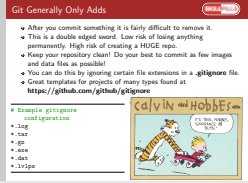


Git Generally Only Adds

- 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
```





My robot repo is now 500GB due to huge Solidworks CAD files.

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**

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

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)
  - **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.

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)
  - 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.

Using Revert

- **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
- **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.

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.

Using Reset

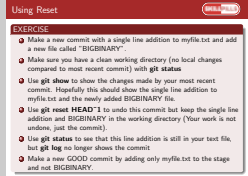
- You may remember the **git reset** command to remove things from the Staging Area (undoing staging)
- We can also use **git reset** to undo commits.
- This is a great way to undo fairly recent commits (one or two before), but not the ideal tool for doing anything further than that.
- When you use **git 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
- 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.



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 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 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.

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 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 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.

Using Rebase

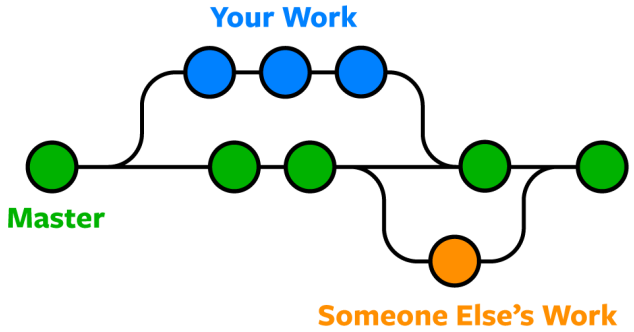
EXERCISE

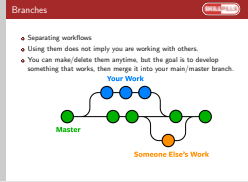
1. Make sure you have at least 5 commits to your **myFirstRepo**.
2. Making at least one of these a meaningless WIP commit.
3. Use **git log** to find the earliest commit that was bad (first WIP commit). Copy the HASH from the commit **just prior** to this WIP commit. We want to rebase our current HEAD 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

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

Branches

- 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.



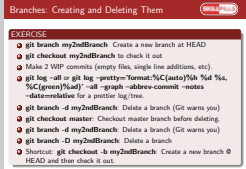


- When you want to develop something separately (New features, refactoring, hacking away) but don't want to affect your main workflow.
- Make a new branch before doing something you think will break the code, or do something you think will be nontrivial (more than one commit).
- With the house example,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

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.

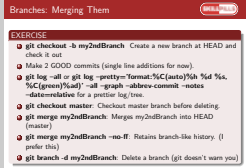


Lets take a look at how to create, delete and play around with branches. Note git warns you about a LOT. Its not perfect, but I highly encourage you to read the warnings carefully and understand them before moving on. The initial learning curve is a bit steeper, but it will save you SOOO much hassle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

Branches: Merging Them

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)

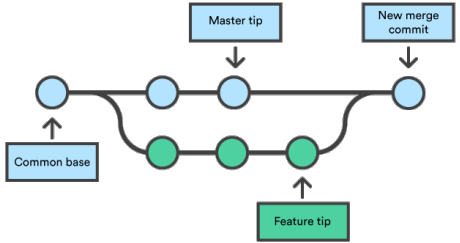


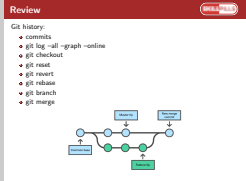
Note you may get merge conflicts. I don't want to spend too much time on these, as it can be a VERY complex topic, but just want to illustrate the basic idea here.

Review

Git history:

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

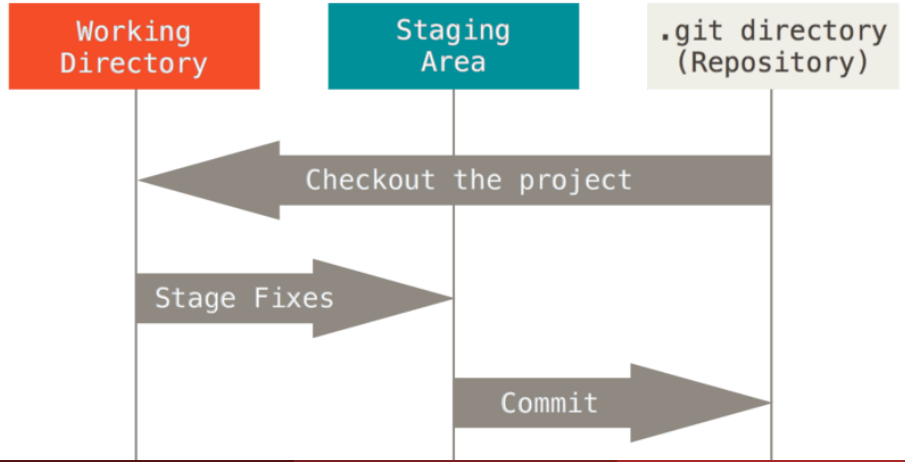


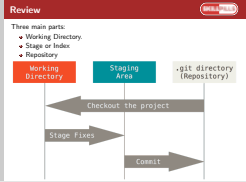


Review

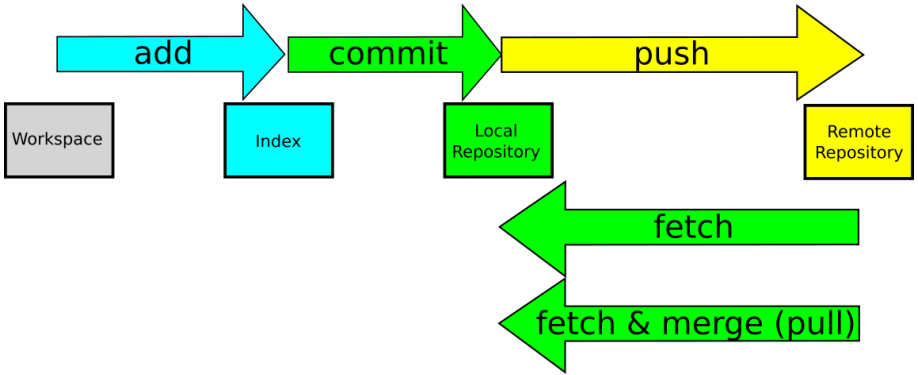
Three main parts:

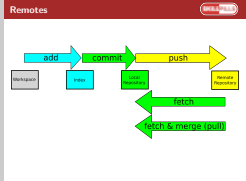
- Working Directory.
- Stage or Index
- Repository





Remotes


add
commit
push
Workspace
Index
Local Repository
Remote Repository
fetch
fetch & merge (pull)


Remotes
add
commit
push
fetch
fetch & merge (pull)

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 idential name
4. Make some simple commits and test the process of **push**ing, **fetch**ing, and **pull**ing stuff from that repo.
5. Check what is going on with each step via **git log –all –graph –oneline"**

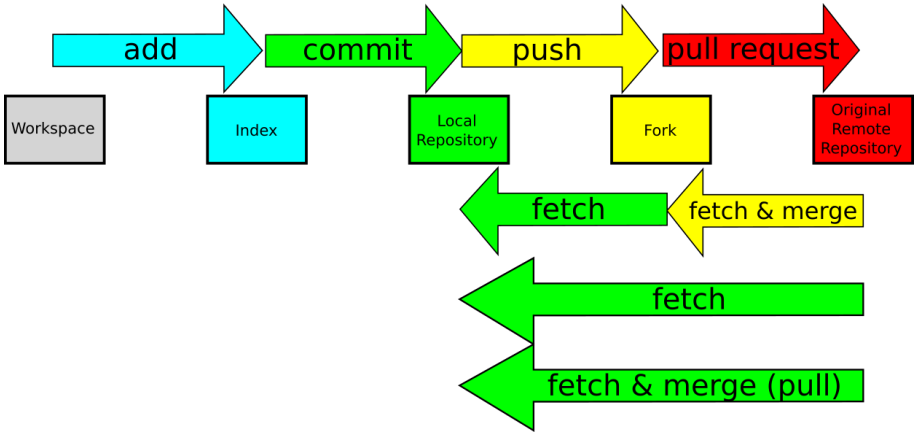
Your First? Remote
EXERCISE
1. Create a new repo using github (e.g. myfirstremote) (https://docs.github.com/en/get-started/quickstart/create-a-repo)
2. Add a README, .gitignore and a license.
3. Create a new repo from the command line with an idential name
4. Make some simple commits and test the process of **push**ing, **fetch**ing, and **pull**ing stuff from that repo.
5. Check what is going on with each step via **git log –all –graph –oneline"**

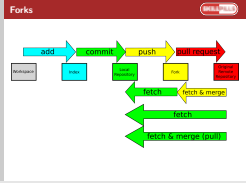
Working with Forks and Remotes

- 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.

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


add
commit
push
pull request
Workspace
Index
Local Repository
Fork
Original Remote Repository
fetch
fetch & merge
fetch
fetch & merge (pull)



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**

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**

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!

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!

Pull Requests (PRs)

- 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.

Pull Requests (PRs)
- 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.

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.

Rebasing a PR

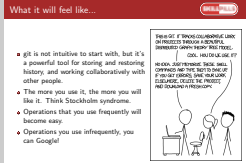
EXERCISE

1. TBD

Rebasing a PR
EXERCISE
TBD

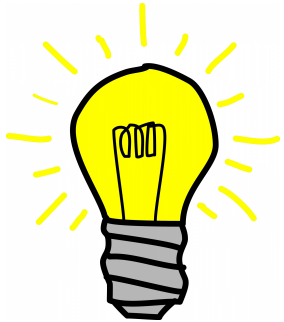
What it will feel like...

- 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.



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.