Lecture 2 recap

Clarifications

Clarifications

Terminal != Shell

- (Video) Terminals are old-school I/O devices that serve as a combination display and a keyboard
  - VT100, VT102, VT220 are the gold standard of terminals
  - Additional functionality with ANSI escape codes (colors and cursor movement, anyone?)
  - Predated by the "teletype", the namesake of `tty`, which is a fancy electric typewriter

Clarifications

Terminal != Shell

- (Video) Terminals are old-school I/O devices that serve as a combination display and a keyboard
  - VT100, VT102, VT220 are the gold standard of terminals
  - Additional functionality with ANSI escape codes (colors and cursor movement, anyone?)
  - Predated by the "teletype", the namesake of `tty`, which is a fancy electric typewriter
- Nowadays no one uses *real* terminals, so we have *virtual* terminals to provide our text-based interfaces
  - You can find these as `/dev/tty*` and friends
  - (This is a pretty deep topic with lots of semantics...I'm not going to even talk about pseudo-terminals: `man pty` is left as an exercise for those interested)

Clarifications

Terminal != Shell

- (Video) Terminals are old-school I/O devices that serve as a combination display and a keyboard
  - VT100, VT102, VT220 are the gold standard of terminals
  - Additional functionality with ANSI escape codes (colors and cursor movement, anyone?)
  - Predated by the "teletype", the namesake of `tty`, which is a fancy electric typewriter
- Nowadays no one uses *real* terminals, so we have *virtual* terminals to provide our text-based interfaces
  - You can find these as `/dev/tty*` and friends
  - (This is a pretty deep topic with lots of semantics...I'm not going to even talk about pseudo-terminals: `man pty` is left as an exercise for those interested)
- Terminal emulators are software that serve as visible front-ends to virtual terminals
  - Examples: xterm, GNOME Terminal, Konsole, macOS Terminal, iTerm2, Windows Terminal (Preview)™

Terminal != Shell

- For modern everyday use, "terminal" refers to a terminal emulator
- A terminal will run the user's default shell
  - You can change this with `chsh`
  - `SHELL` environment variable can tell you what yours is
- Shell handles the terminal input, parses, then handles *creating* the processes specified by commands

Terminal != Shell

- For modern everyday use, "terminal" refers to a terminal emulator
- A terminal will run the user's default shell
  - You can change this with `chsh`
  - `SHELL` environment variable can tell you what yours is
- Shell handles the terminal input, parses, then handles *creating* the processes specified by commands
- In a general sense, a shell is a program that serves as an interface between a user and the operating system
- Command line shells: `bash`, `zsh`, `fish`
  - Most common interpretation of "shell"
  - Take in typed-in commands to handle executing them

Terminal != Shell

- For modern everyday use, "terminal" refers to a terminal emulator
- A terminal will run the user's default shell
  - You can change this with `chsh`
  - `SHELL` environment variable can tell you what yours is
- Shell handles the terminal input, parses, then handles *creating* the processes specified by commands
- In a general sense, a shell is a program that serves as an interface between a user and the operating system
- Command line shells: `bash`, `zsh`, `fish`
  - Most common interpretation of "shell"
  - Take in typed-in commands to handle executing them
- Graphical shells: GNOME, KDE Plasma, Windows shell, macOS Quartz + Finder + Dock + other components
  - Handle mice/other input devices to do things like launch a program when its icon is clicked

Recap

- Basic commands: `pwd`, `ls`, `mv`, `rm`, `touch`, `cd`, `man`

Recap

- Basic commands: `pwd`, `ls`, `mv`, `rm`, `touch`, `cd`, `man`
  - If command is a path (i.e. contains a forward slash), will run whatever's at that path
  - e.g. `$ subdir/hello-world`, `$ ./script.sh`, `$ /usr/bin/gedit`

Recap

- Basic commands: `pwd`, `ls`, `mv`, `rm`, `touch`, `cd`, `man`
  - If command is a path (i.e. contains a forward slash), will run whatever's at that path
  - e.g. `$ subdir/hello-world`, `$ ./script.sh`, `$ /usr/bin/gedit`
  - If command is just some word (no forward slash), will search the directories in `PATH` for some executable that matches
  - e.g. `$ gedit`, which will search `PATH`, which has `/usr/bin/`, and finds `/usr/bin/gedit` to run

Recap

- Basic commands: `pwd`, `ls`, `mv`, `rm`, `touch`, `cd`, `man`
  - If command is a path (i.e. contains a forward slash), will run whatever's at that path
  - e.g. `$ subdir/hello-world`, `$ ./script.sh`, `$ /usr/bin/gedit`
  - If command is just some word (no forward slash), will search the directories in `PATH` for some executable that matches
  - e.g. `$ gedit`, which will search `PATH`, which has `/usr/bin/`, and finds `/usr/bin/gedit` to run
  - Note: some commands are built into the shell, like `cd`, `jobs`, `exit`, and are not actual executables

Recap

- Basic commands: `pwd`, `ls`, `mv`, `rm`, `touch`, `cd`, `man`
  - If command is a path (i.e. contains a forward slash), will run whatever's at that path
  - e.g. `$ subdir/hello-world`, `$ ./script.sh`, `$ /usr/bin/gedit`
  - If command is just some word (no forward slash), will search the directories in `PATH` for some executable that matches
  - e.g. `$ gedit`, which will search `PATH`, which has `/usr/bin/`, and finds `/usr/bin/gedit` to run
  - Note: some commands are built into the shell, like `cd`, `jobs`, `exit`, and are not actual executables
- Basic job control: `jobs`, `kill`, Ctrl-Z to suspend

Recap

- Basic commands: `pwd`, `ls`, `mv`, `rm`, `touch`, `cd`, `man`
  - If command is a path (i.e. contains a forward slash), will run whatever's at that path
  - e.g. `$ subdir/hello-world`, `$ ./script.sh`, `$ /usr/bin/gedit`
  - If command is just some word (no forward slash), will search the directories in `PATH` for some executable that matches
  - e.g. `$ gedit`, which will search `PATH`, which has `/usr/bin/`, and finds `/usr/bin/gedit` to run
  - Note: some commands are built into the shell, like `cd`, `jobs`, `exit`, and are not actual executables
- Basic job control: `jobs`, `kill`, Ctrl-Z to suspend
- Basic command synthesis: `&&`, `||`, `;`

Recap

- Basic commands: `pwd`, `ls`, `mv`, `rm`, `touch`, `cd`, `man`
  - If command is a path (i.e. contains a forward slash), will run whatever's at that path
  - e.g. `$ subdir/hello-world`, `$ ./script.sh`, `$ /usr/bin/gedit`
  - If command is just some word (no forward slash), will search the directories in `PATH` for some executable that matches
  - e.g. `$ gedit`, which will search `PATH`, which has `/usr/bin/`, and finds `/usr/bin/gedit` to run
  - Note: some commands are built into the shell, like `cd`, `jobs`, `exit`, and are not actual executables
- Basic job control: `jobs`, `kill`, Ctrl-Z to suspend
- Basic command synthesis: `&&`, `||`, `;`
- File properties: rwx bits, `chmod`, `chown`

Recap

- Basic commands: `pwd`, `ls`, `mv`, `rm`, `touch`, `cd`, `man`
  - If command is a path (i.e. contains a forward slash), will run whatever's at that path
  - e.g. `$ subdir/hello-world`, `$ ./script.sh`, `$ /usr/bin/gedit`
  - If command is just some word (no forward slash), will search the directories in `PATH` for some executable that matches
  - e.g. `$ gedit`, which will search `PATH`, which has `/usr/bin/`, and finds `/usr/bin/gedit` to run
  - Note: some commands are built into the shell, like `cd`, `jobs`, `exit`, and are not actual executables
- Basic job control: `jobs`, `kill`, Ctrl-Z to suspend
- Basic command synthesis: `&&`, `||`, `;`
- File properties: rwx bits, `chmod`, `chown`
- Basic file redirection: `<`, `>`, `>>`, `|`

Recap

- Basic commands: `pwd`, `ls`, `mv`, `rm`, `touch`, `cd`, `man`
  - If command is a path (i.e. contains a forward slash), will run whatever's at that path
  - e.g. `$ subdir/hello-world`, `$ ./script.sh`, `$ /usr/bin/gedit`
  - If command is just some word (no forward slash), will search the directories in `PATH` for some executable that matches
  - e.g. `$ gedit`, which will search `PATH`, which has `/usr/bin/`, and finds `/usr/bin/gedit` to run
  - Note: some commands are built into the shell, like `cd`, `jobs`, `exit`, and are not actual executables
- Basic job control: `jobs`, `kill`, Ctrl-Z to suspend
- Basic command synthesis: `&&`, `||`, `;`
- File properties: rwx bits, `chmod`, `chown`
- Basic file redirection: `<`, `>`, `>>`, `|`
- Environment variables: `PATH`, `HOME`, `USER`, `PWD`

Recap

- Basic commands: `pwd`, `ls`, `mv`, `rm`, `touch`, `cd`, `man`
  - If command is a path (i.e. contains a forward slash), will run whatever's at that path
  - e.g. `$ subdir/hello-world`, `$ ./script.sh`, `$ /usr/bin/gedit`
  - If command is just some word (no forward slash), will search the directories in `PATH` for some executable that matches
  - e.g. `$ gedit`, which will search `PATH`, which has `/usr/bin/`, and finds `/usr/bin/gedit` to run
  - Note: some commands are built into the shell, like `cd`, `jobs`, `exit`, and are not actual executables
- Basic job control: `jobs`, `kill`, Ctrl-Z to suspend
- Basic command synthesis: `&&`, `||`, `;`
- File properties: rwx bits, `chmod`, `chown`
- Basic file redirection: `<`, `>`, `>>`, `|`
- Environment variables: `PATH`, `HOME`, `USER`, `PWD`
  - There's a subtle difference regarding "shell variables" which are managed and used by the shell itself and not the overal execution environment, e.g. `PS1`
  - You can set and echo them like environment variables, but they don't get passed onto processes that get created

Scripting

- Shell scripts just a line by line listing of shell commands
- Saves you the effort of having to enter all of them in again

Scripting

- Shell scripts just a line by line listing of shell commands
- Saves you the effort of having to enter all of them in again
- *running* a script gives it its own instance of the interpreter
  - `$ ./script.sh`
  - `$ bash script.sh`
  - Environment variable changes will stay local to that instance

Scripting

- Shell scripts just a line by line listing of shell commands
- Saves you the effort of having to enter all of them in again
- *running* a script gives it its own instance of the interpreter
  - `$ ./script.sh`
  - `$ bash script.sh`
  - Environment variable changes will stay local to that instance
- *sourcing* a script makes your current shell actually execute each of the lines
  - `source script.sh`
  - As if you typed "HELLO=world" in yourself
  - Just changed your `.bashrc` and want to reload it without running a new instance of `bash`?

Scripting

- Shell scripts just a line by line listing of shell commands
- Saves you the effort of having to enter all of them in again
- *running* a script gives it its own instance of the interpreter
  - `$ ./script.sh`
  - `$ bash script.sh`
  - Environment variable changes will stay local to that instance
- *sourcing* a script makes your current shell actually execute each of the lines
  - `source script.sh`
  - As if you typed "HELLO=world" in yourself
  - Just changed your `.bashrc` and want to reload it without running a new instance of `bash`?
  - `$ source ~/.bashrc`

Announcements

- HW1 due tonight
- ADV1 due next Friday (Jan 31)
- HW2, ADV2 due Feb 3

GITing Started

Week 3

I bet you've been waiting for this lecture

Overview

1. What is version control?
2. Git basic flow
3. Git branches
4. A taste of Git remotes

Version control

- Keep track of changes of files over time, allowing you to roll back to previous versions
- Software to handle this are known as "version control systems" (VCS)

Two paradigms

Centralized (CVCS)

- Central server keeps track of all the changes and history
- Each developer has local copies of files they need, but need to check in with the server to do any versioning
- Sever down? Good luck.
- Examples: CVS, SVN, Perforce

Two paradigms

Centralized (CVCS)

- Central server keeps track of all the changes and history
- Each developer has local copies of files they need, but need to check in with the server to do any versioning
- Sever down? Good luck.
- Examples: CVS, SVN, Perforce

Decentralized (DVCS)

- Each developer has a local copy of the entire codebase and its history
- Developers can perform versioning locally without needing to contact a server
- Server optional
- Examples: Git, Mercurial

Why version control?

- Checkpointing your work

Why version control?

- Checkpointing your work
  - Have you ever made `main.c.backup1`, `main.c.backup2`,…?

Why version control?

- Checkpointing your work
  - Have you ever made `main.c.backup1`, `main.c.backup2`,...?
- Keeping multiple parallel versions of your work

Why version control?

- Checkpointing your work
  - Have you ever made `main.c.backup1`, `main.c.backup2`,...?
- Keeping multiple parallel versions of your work
  - Have you implemented {thing} one way, made another implementation of {thing} but wanted to keep both around?

Why version control?

- Checkpointing your work
  - Have you ever made `main.c.backup1`, `main.c.backup2`,...?
- Keeping multiple parallel versions of your work
  - Have you implemented {thing} one way, made another implementation of {thing} but wanted to keep both around?
- Collaborating around your work

Why version control?

- Checkpointing your work
  - Have you ever made `main.c.backup1`, `main.c.backup2`,...?
- Keeping multiple parallel versions of your work
  - Have you implemented {thing} one way, made another implementation of {thing} but wanted to keep both around?
- Collaborating around your work
  - Have you ever emailed code or sent code in some messaging app?

Why version control?

- Checkpointing your work
  - Have you ever made `main.c.backup1`, `main.c.backup2`,...?
- Keeping multiple parallel versions of your work
  - Have you implemented {thing} one way, made another implementation of {thing} but wanted to keep both around?
- Collaborating around your work
  - Have you ever emailed code or sent code in some messaging app?
  - Have you tried to coordinate people working on the same file?

Enter…

Enter...Git!

Enter...Git!

- Distributed version control system (DVCS)
- Designed by Linus Torvalds to manage the Linux kernel

Enter...Git!

- Distributed version control system (DVCS)
- Designed by Linus Torvalds to manage the Linux kernel
- No server needed, super easy to get started with
  - `git init`
  - `git add`
  - `git commit`

Enter...Git!

- Distributed version control system (DVCS)
- Designed by Linus Torvalds to manage the Linux kernel
- No server needed, super easy to get started with
  - `git init`
  - `git add`
  - `git commit`

That's it, lecture's over!

Git Overview

- Repository: a directory of stuff that Git is versioning
  - `.git` is the directory that holds all this metadata
- Commit: a checkpoint for the files in the repository
  - Given a hash for identification
  - (Unlike other VCS, has actual snapshots of files rather than diffs)
- History is a linked list of commits pointing to their parent

Basic commands

- `git init`
- `git status`
- `git log`
- `git add`
- `git reset`
- `git checkout`
- `git commit`

Some neat resources

- `man git`
- `man git-<command>` or `help git <command>`
- Official Git documentation
- Official Git tutorial
  - `man gittutorial`
- Official Git minimal set of useful commands
  - `man giteveryday`
- *Pro Git* book
  - Free and comprehensive
  - Besides being on the web, has `.pdf`, `.epub`, and `.mobi` formats!
  - A really great read

Files have multiple states

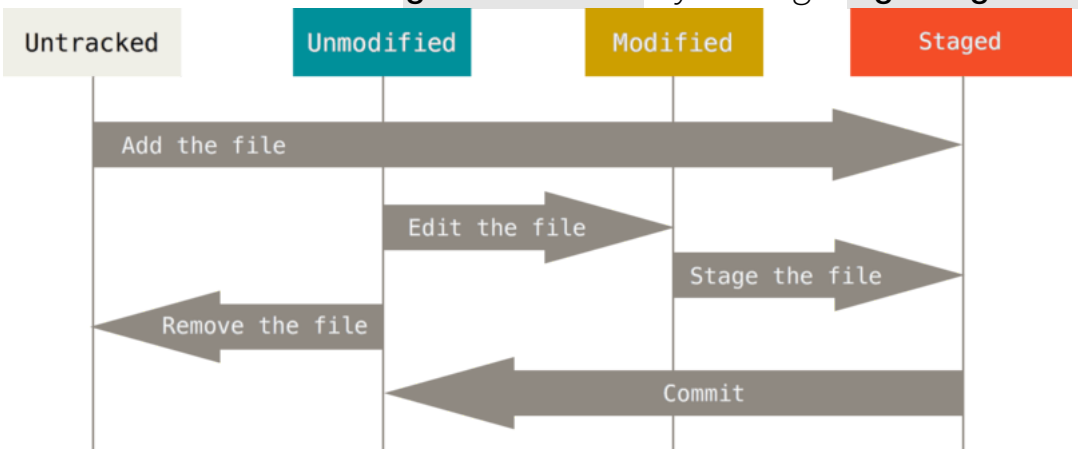
- **Unmodified**: Nothing has happened to this file; no changes compared to current commit

Files have multiple states

- **Unmodified**: Nothing has happened to this file; no changes compared to current commit
- **Modified**: This file differs from the version in the current commit. Can be `git add`ed to be **Staged**

Files have multiple states

- **Unmodified**: Nothing has happened to this file; no changes compared to current commit
- **Modified**: This file differs from the version in the current commit. Can be `git add`ed to be **Staged**
- **Staged**: This file differs, and is set to be in the next commit
- **Untracked**: This file does not exist in the current commit
  - It's pretty similar to **Modified**; it "differs" by existing while the current commit says it doesn't exist
  - You can hide these from `git status` by adding a `.gitignore` file



From *Pro Git*

Ties into the "areas"

- **Working Directory**: the directory as your filesystem sees it, a mess of files which may or may not be changed, or may be even untracked

Ties into the "areas"

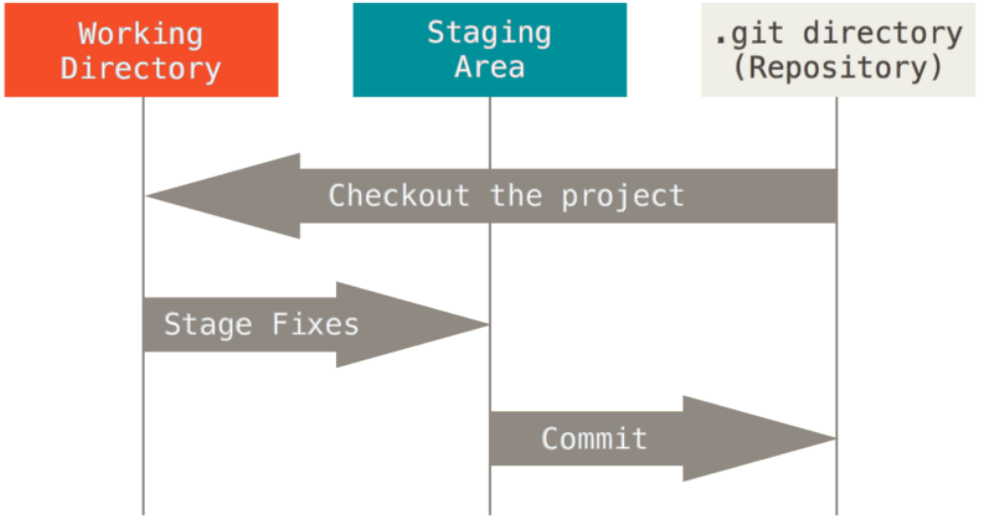
- **Working Directory**: the directory as your filesystem sees it, a mess of files which may or may not be changed, or may be even untracked
- **Staging Area**/**Index**: list of files whose snapshots will be part of the next commit

Ties into the "areas"

- **Working Directory**: the directory as your filesystem sees it, a mess of files which may or may not be changed, or may be even untracked
- **Staging Area**/**Index**: list of files whose snapshots will be part of the next commit
  - You'll see it referred to as either: I'm going to say "**index**" for brevity and to distinguish it from the file state of **Staged**
- **Repository**: What commits Git now has saved

Ties into the "areas"

- **Working Directory**: the directory as your filesystem sees it, a mess of files which may or may not be changed, or may be even untracked
- **Staging Area**/**Index**: list of files whose snapshots will be part of the next commit
  - You'll see it referred to as either: I'm going to say "**index**" for brevity and to distinguish it from the file state of **Staged**
- **Repository**: What commits Git now has saved
- Files and their snapshots will work their way through these three areas



From *Pro Git*

Scenario 1: Untracked file

1. An untracked file chills in the **Working Directory**

Scenario 1: Untracked file

1. An untracked file chills in the **Working Directory**
2. You decide to start versioning it, so you `git add` it, making it **Staged** and putting it into the **Index**

Scenario 1: Untracked file

1. An untracked file chills in the **Working Directory**
2. You decide to start versioning it, so you `git add` it, making it **Staged** and putting it into the **Index**
3. You commit the file in the **Index**, landing it in **Repository**

Scenario 1: Untracked file

1. An untracked file chills in the **Working Directory**
2. You decide to start versioning it, so you `git add` it, making it **Staged** and putting it into the **Index**
3. You commit the file in the **Index**, landing it in **Repository**

Scenario 2: Modified file

1. The file is now **Unchanged** as of the current commit, and is still chilling in the **Working Directory**

Scenario 1: Untracked file

1. An untracked file chills in the **Working Directory**
2. You decide to start versioning it, so you `git add` it, making it **Staged** and putting it into the **Index**
3. You commit the file in the **Index**, landing it in **Repository**

Scenario 2: Modified file

1. The file is now **Unchanged** as of the current commit, and is still chilling in the **Working Directory**
2. You make some changes, so now the file is **Modified**

Scenario 1: Untracked file

1. An untracked file chills in the **Working Directory**
2. You decide to start versioning it, so you `git add` it, making it **Staged** and putting it into the **Index**
3. You commit the file in the **Index**, landing it in **Repository**

Scenario 2: Modified file

1. The file is now **Unchanged** as of the current commit, and is still chilling in the **Working Directory**
2. You make some changes, so now the file is **Modified**
    - Oops, maybe I don't like what I did and want to change it back to the old commited version, let's `git checkout` it

Scenario 1: Untracked file

1. An untracked file chills in the **Working Directory**
2. You decide to start versioning it, so you `git add` it, making it **Staged** and putting it into the **Index**
3. You commit the file in the **Index**, landing it in **Repository**

Scenario 2: Modified file

1. The file is now **Unchanged** as of the current commit, and is still chilling in the **Working Directory**
2. You make some changes, so now the file is **Modified**
    - Oops, maybe I don't like what I did and want to change it back to the old commited version, let's `git checkout` it
3. You `git add` it, making it **Staged** and putting it into the **Index**

Scenario 1: Untracked file

1. An untracked file chills in the **Working Directory**
2. You decide to start versioning it, so you `git add` it, making it **Staged** and putting it into the **Index**
3. You commit the file in the **Index**, landing it in **Repository**

Scenario 2: Modified file

1. The file is now **Unchanged** as of the current commit, and is still chilling in the **Working Directory**
2. You make some changes, so now the file is **Modified**
    - Oops, maybe I don't like what I did and want to change it back to the old commited version, let's `git checkout` it
3. You `git add` it, making it **Staged** and putting it into the **Index**
    - Oops, maybe I added an extra file I didn't want to stage, let's `git reset` it back to **Modified**

Scenario 1: Untracked file

1. An untracked file chills in the **Working Directory**
2. You decide to start versioning it, so you `git add` it, making it **Staged** and putting it into the **Index**
3. You commit the file in the **Index**, landing it in **Repository**

Scenario 2: Modified file

1. The file is now **Unchanged** as of the current commit, and is still chilling in the **Working Directory**
2. You make some changes, so now the file is **Modified**
    - Oops, maybe I don't like what I did and want to change it back to the old commited version, let's `git checkout` it
3. You `git add` it, making it **Staged** and putting it into the **Index**
    - Oops, maybe I added an extra file I didn't want to stage, let's `git reset` it back to **Modified**
4. You commit the file's snapshot, getting that snapshot into the **Repository**

Putting it together, locally

1. Initialize the repository
    - `git init`

Putting it together, locally

1. Initialize the repository
    - `git init`
2. Add the initial files you want to track to the **Index**
    - `git add`

Putting it together, locally

1. Initialize the repository
    - `git init`
2. Add the initial files you want to track to the **Index**
    - `git add`
3. Commit those initial files to the **Repository**
    - `git commit`

Putting it together, locally

1. Initialize the repository
    - `git init`
2. Add the initial files you want to track to the **Index**
    - `git add`
3. Commit those initial files to the **Repository**
    - `git commit`
4. Modify some files
    - Don't like a modification and want to make the file **Unmodified** again? `git checkout <filename>`
    - (`git restore` is a new experimental command that performs this behavior; `git checkout` has been a bit overloaded with functionality and does much more than just this as we'll soon see)

Putting it together, locally

1. Initialize the repository
    - `git init`
2. Add the initial files you want to track to the **Index**
    - `git add`
3. Commit those initial files to the **Repository**
    - `git commit`
4. Modify some files
    - Don't like a modification and want to make the file **Unmodified** again? `git checkout <filename>`
    - (`git restore` is a new experimental command that performs this behavior; `git checkout` has been a bit overloaded with functionality and does much more than just this as we'll soon see)
5. Add **Modified/Untracked** files to the **Index**
    - `git add`
    - Accidentally added a file? `git reset <filename>` to take it out of the **Index**

Putting it together, locally

1. Initialize the repository
    - `git init`
2. Add the initial files you want to track to the **Index**
    - `git add`
3. Commit those initial files to the **Repository**
    - `git commit`
4. Modify some files
    - Don't like a modification and want to make the file **Unmodified** again? `git checkout <filename>`
    - (`git restore` is a new experimental command that performs this behavior; `git checkout` has been a bit overloaded with functionality and does much more than just this as we'll soon see)
5. Add **Modified/Untracked** files to the **Index**
    - `git add`
    - Accidentally added a file? `git reset <filename>` to take it out of the **Index**
6. Commit those files to the **Repository**
    - `git commit`
    - Didn't like your commit message or forgot to include some files? Add them to the **Index**, and `git commit --amend`

Putting it together, locally

1. Initialize the repository
    - `git init`
2. Add the initial files you want to track to the **Index**
    - `git add`
3. Commit those initial files to the **Repository**
    - `git commit`
4. Modify some files
    - Don't like a modification and want to make the file **Unmodified** again? `git checkout <filename>`
    - (`git restore` is a new experimental command that performs this behavior; `git checkout` has been a bit overloaded with functionality and does much more than just this as we'll soon see)
5. Add **Modified/Untracked** files to the **Index**
    - `git add`
    - Accidentally added a file? `git reset <filename>` to take it out of the **Index**
6. Commit those files to the **Repository**
    - `git commit`
    - Didn't like your commit message or forgot to include some files? Add them to the **Index**, and `git commit --amend`
7. Goto 4, rinse and repeat

Commits

- `git commit -m <message>` is a quick and dirty way to make a commit
- Not super ideal when it's a project that you're going to collaborate with others on
- `git commit` will open the configured editor and allow you to fully fill out a commit message

Commits

Title

- Limit to 50 characters
- Capitalize the first letter
- Imperative ("Fix xyz", "Remove abc")
- Summarize the commit

Commits

Title

- Limit to 50 characters
- Capitalize the first letter
- Imperative ("Fix xyz", "Remove abc")
- Summarize the commit

Body

- Limit to 72 characters per line
- Explain what changed and why, not how: your code (ideally) is the "how"
- (Depending on your team/workplace: references to bug/issue number e.g. "Issue #22772", "Bug #1337")

No, I'm not making this up, it's straight from the horse's mouth

Ultimately these are just guidelines, not rules. Do what your team does, but try to keep good habits when you start something yourself

Branching

- Making a linked list of commits is cool, but what can we do with it? Can we go back? Can we split off?

Branching

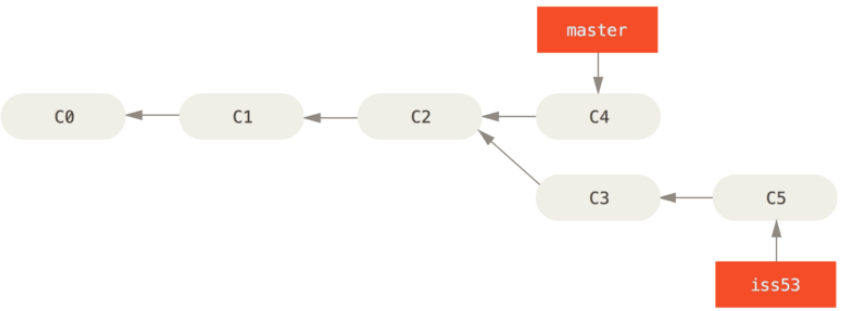
- Making a linked list of commits is cool, but what can we do with it? Can we go back? Can we split off?
- `HEAD` is a pointer pointing to the current commit that's being looked at

Branching

- Making a linked list of commits is cool, but what can we do with it? Can we go back? Can we split off?
- `HEAD` is a pointer pointing to the current commit that's being looked at
- A **branch** in Git is a pointer to a commit
    - Super lightweight compared to other VCS, go wild

Branching

- Making a linked list of commits is cool, but what can we do with it? Can we go back? Can we split off?
- `HEAD` is a pointer pointing to the current commit that's being looked at
- A **branch** in Git is a pointer to a commit
  - Super lightweight compared to other VCS, go wild
- Lots of applications:
  - Make a "backup" of branch
  - Manage a feature ("topic"/"feature" branches)
  - Have a separate line of development (e.g. taking two different approaches)
  - Represent release schedules (e.g. a development branch and a release branch)



From *Pro Git*

Branching

- The default branch is `master`
  - Typically used for production/release

Branching

- The default branch is `master`
  - Typically used for production/release
- `git branch` lists your local branches

Branching

- The default branch is `master`
  - Typically used for production/release
- `git branch` lists your local branches
- `git branch <branch-name>` creates a new branch
  - `<branch-name>` point to wherever `HEAD` is pointing to

Branching

- The default branch is `master`
  - Typically used for production/release
- `git branch` lists your local branches
- `git branch <branch-name>` creates a new branch
  - `<branch-name>` point to wherever `HEAD` is pointing to
- `git checkout <branch-name>` checks out the branch, making your `HEAD` point to where `<branch-name>` is pointing to
  - `git switch` also switches to a branch; added in Git 2.23.0 and is experimental at the moment

Branching

- The default branch is `master`
  - Typically used for production/release
- `git branch` lists your local branches
- `git branch <branch-name>` creates a new branch
  - `<branch-name>` point to wherever `HEAD` is pointing to
- `git checkout <branch-name>` checks out the branch, making your `HEAD` point to where `<branch-name>` is pointing to
  - `git switch` also switches to a branch; added in Git 2.23.0 and is experimental at the moment
  - `git checkout -b <branch-name>` creates **and checks out** the branch

Branching

- The default branch is `master`
  - Typically used for production/release
- `git branch` lists your local branches
- `git branch <branch-name>` creates a new branch
  - `<branch-name>` point to wherever `HEAD` is pointing to
- `git checkout <branch-name>` checks out the branch, making your `HEAD` point to where `<branch-name>` is pointing to
  - `git switch` also switches to a branch; added in Git 2.23.0 and is experimental at the moment
  - `git checkout -b <branch-name>` creates **and checks out** the branch
- `git merge <branch-name>` will replay commits from `<branch-name>` onto the current branch
  - If the branches diverged (`<branch-name>` and the current branch both got new commits before merging), a special "merge commit" will be produced linking the two branches
  - (This is where things get a bit more messy and complicated: we'll take a closer look later)

Remotes

- So far everything we've been looking at has been local
- What if you want to share it?

Remotes

- So far everything we've been looking at has been local
- What if you want to share it?
- A **remote** is a repository is hosted by some server on the Internet or internal network

Remotes

- So far everything we've been looking at has been local
- What if you want to share it?
- A **remote** is a repository is hosted by some server on the Internet or internal network
- `git clone <URL>` will copy the repository from the server to your local machine
  - `origin` is the default name of the **remote** whose URL you cloned from

Remotes

- So far everything we've been looking at has been local
- What if you want to share it?
- A **remote** is a repository is hosted by some server on the Internet or internal network
- `git clone <URL>` will copy the repository from the server to your local machine
  - `origin` is the default name of the **remote** whose URL you cloned from
- `git remote -v` will list your **remotes**

Remotes

- So far everything we've been looking at has been local
- What if you want to share it?
- A **remote** is a repository is hosted by some server on the Internet or internal network
- `git clone <URL>` will copy the repository from the server to your local machine
  - `origin` is the default name of the **remote** whose URL you cloned from
- `git remote -v` will list your **remotes**
- `git fetch` will get the latest commits from the **remote** into the **Repository**

Remotes

- So far everything we've been looking at has been local
- What if you want to share it?
- A **remote** is a repository is hosted by some server on the Internet or internal network
- `git clone <URL>` will copy the repository from the server to your local machine
  - `origin` is the default name of the **remote** whose URL you cloned from
- `git remote -v` will list your **remotes**
- `git fetch` will get the latest commits from the **remote** into the **Repository**
- `git pull` will do a `git fetch` and additionally `git merge`, potentially modifying your **Working Directory**

Remotes

- So far everything we've been looking at has been local
- What if you want to share it?
- A **remote** is a repository is hosted by some server on the Internet or internal network
- `git clone <URL>` will copy the repository from the server to your local machine
  - `origin` is the default name of the **remote** whose URL you cloned from
- `git remote -v` will list your **remotes**
- `git fetch` will get the latest commits from the **remote** into the **Repository**
- `git pull` will do a `git fetch` and additionally `git merge`, potentially modifying your **Working Directory**
- As you work on your locally, you can make commits to your local **Repository**
- `git push` will send your commits to the **remote**

Remotes

- So far everything we've been looking at has been local
- What if you want to share it?
- A **remote** is a repository is hosted by some server on the Internet or internal network
- `git clone <URL>` will copy the repository from the server to your local machine
  - `origin` is the default name of the **remote** whose URL you cloned from
- `git remote -v` will list your **remotes**
- `git fetch` will get the latest commits from the **remote** into the **Repository**
- `git pull` will do a `git fetch` and additionally `git merge`, potentially modifying your **Working Directory**
- As you work on your locally, you can make commits to your local **Repository**
- `git push` will send your commits to the **remote**

Remote hosting services (a.k.a. Git != GitHub)

- GitHub
- BitBucket
- GitLab
  - GitLab is also a Git host server software that you can use to host your own repos

Questions?

Addenda

Core commands

- `git init`
- `git status`
- `git log`
- `git add`
- `git reset`
- `git commit`
- `git branch`
- `git checkout`
  - (`git switch`)
  - (`git restore`)
- `git merge`

Remote and Collaboration commands

- `git clone`
- `git fetch`
- `git pull`
- `git push`
- `git remote`

Additional Commands

- `git help`
- `git stash`
- `git show`
- `git diff`
- `git rebase`
- `git blame`