Comic Credit: Randall Munroe, xkcd.com

Lab 5/6: Version Control

CS 2112 Fall 2025

September 29 / October 8, 2025

Why Version Control?

You're emailing your project back and forth with your partner. An hour before the deadline, you and your partner both find different bugs and work feverishly to correct them. When you try to submit, you find that you have two different versions of the code, and you don't have enough time to figure out who changed what, how to merge them together into one final project, and what, if any, bugs were introduced along the way!

Download Git

First, if necessary download and install Git (follow the instructions for your OS) https://git-scm.com/downloads

The Git CLI is a powerful tool, but you prefer a graphical interface, IntelliJ comes with a Git client built-in, and is sufficient for most use-cases covered in this class.

Note Cornell requires the use of SSH. If you are on Windows, Git Credential Manager will automatically let you authenticate via the browser, but on macOS and Linux, if you are not comfortable with SSH keys, we recommend using IntelliJ instead.

Configure Git (Terminal)

If you're using git on the terminal, enter the following two commands to set up your name and email:

```
git config --global user.name "<Your Name>"
git config --global user.email <netid>@cornell.edu
```

Configure IntelliJ Git

If you're using IntelliJ, choose File > New > Project from Version Control

On the following screen, choose GitHub Enterprise. Enter `github.coecis.cornell.edu` as the server, and click "Generate" next to the token field.

Your browser should open to the Cornell GitHub page. Set an expiration date late enough to cover the end of this class (December should work) and click "Generate token."

Copy the token from the resulting page and paste it back into IntelliJ.

SSH

For CLI users, the newest version of Git on Windows should come with Git Credential Manager, which will automatically prompt you to log in.
Otherwise, please follow these instructions to add an SSH key to your Cornell GitHub account.

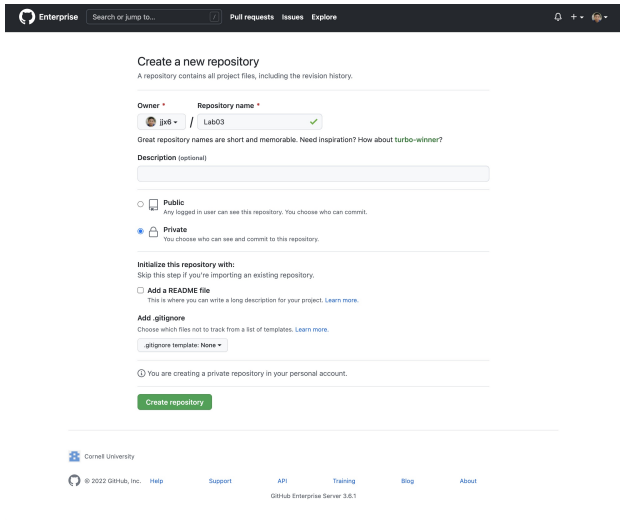
Repository

Git calls a project a "repository".
Go to https://github.coecis.cornell.edu and login with your netid to get started.

Note: for future assignments in this class, we will be assigning you a repository.

Creating the Repository

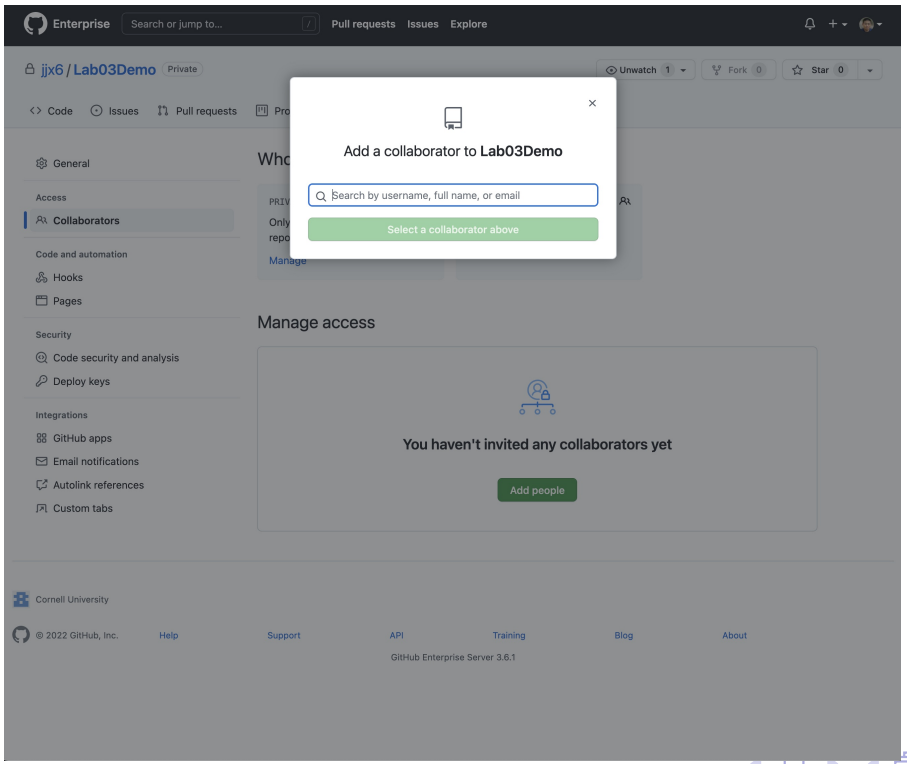
Click the plus sign in the top right to make a new repository. Enter a repository name, choose "Private" for privacy, and check "Initialize with a README."



VERY IMPORTANT: You MUST remember to make your repositories private. Academic integrity is enforced very strictly at Cornell and you do not want to open yourself to potential liability.

Adding Collaborators

From the repository's main page, choose "Settings" → "Collaborators" and enter their Cornell emails. Your partner should then be able to see the repo listed on the home page if they refresh.

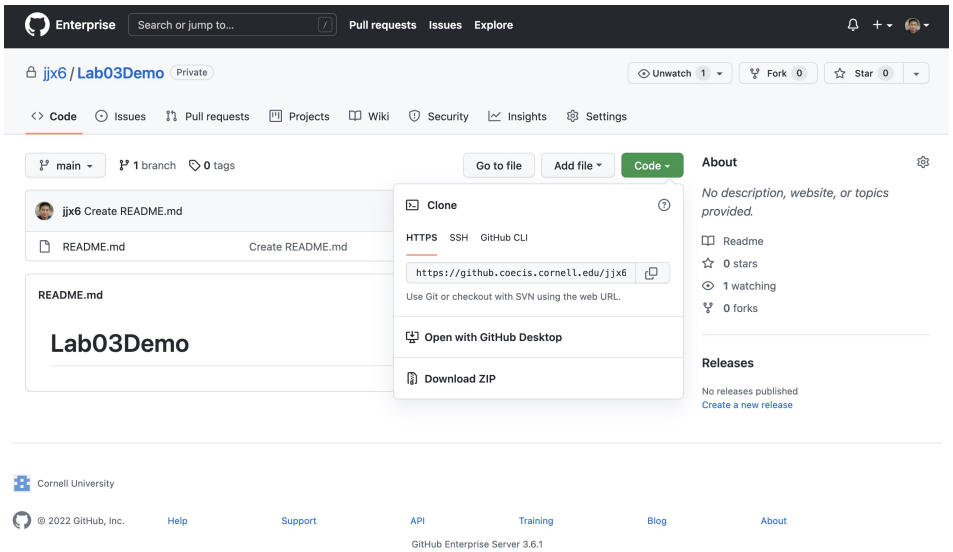


Cloning a Repo (IntelliJ)

Both partners should clone the repo.
Choose File > New > Project from Version Control.
Select GitHub Enterprise, choose your repo name, and then click Clone.

Cloning a Repo (CLI)

From the repository's main page, click the green "Code" button. Copy the provided link.



`cd` to the directory you want and then run `git clone <PASTE>` (where `<PASTE>` is where you paste the link from the last slide). Then, open the resulting folder in IntelliJ as a normal project like any other.

The Workflow

To get your code to your partner's computer, there are three main steps.

1) **Commit** - Tell Git about your changes
2) **Push** - Send your changes up to the server
3) **Pull** - Your partner gets changes down from the server

1) Commit

To demonstrate, create a file with some text in it.
On terminal, run `git add <filename>` followed by
`git commit -m "Commit Message"`.
In IntelliJ, when asked if you want to add the file, click yes. Then, in the Git tab, check the box next to your file under the "Changes" list, type a commit message, and choose Commit (or Commit and Push).

1) Commit (Add vs. Commit)

Why two separate steps?

By default, Git doesn't see anything until you tell it to.
The first step of "adding" (or "staging") your changes tells Git to notice that you've changed a file (and moves it from what's called your "Working Directory" to the "Staging Area").
Then, when you "commit" the change, Git confirms those changes into your repository.
This way, you can pick and choose what you want to commit, adding only finished work and leaving work-in-progress files in your working directory.

2) Push

Send your changes to the server.

On terminal, run `git push`
In IntelliJ, choose Git > Push (if you didn't click "Commit and Push" in the last step) and then click the Push button.

3) Pull

Your partner can now get your changes on their machine.

On terminal, run `git pull`
In IntelliJ, choose Git > Pull and then click the Pull button.

Basic Workflow

You now have the bare minimum needed to get work done with Git. If you remember nothing else from this presentation, at least remember this.

Knowledge Check
Say you just came back from dinner and you sit down at your computer to resume work on a project you and your partner have been working on. What is the first step you should do?

Answer: Pull! You want to make sure you have your partner's latest changes before you start working again.

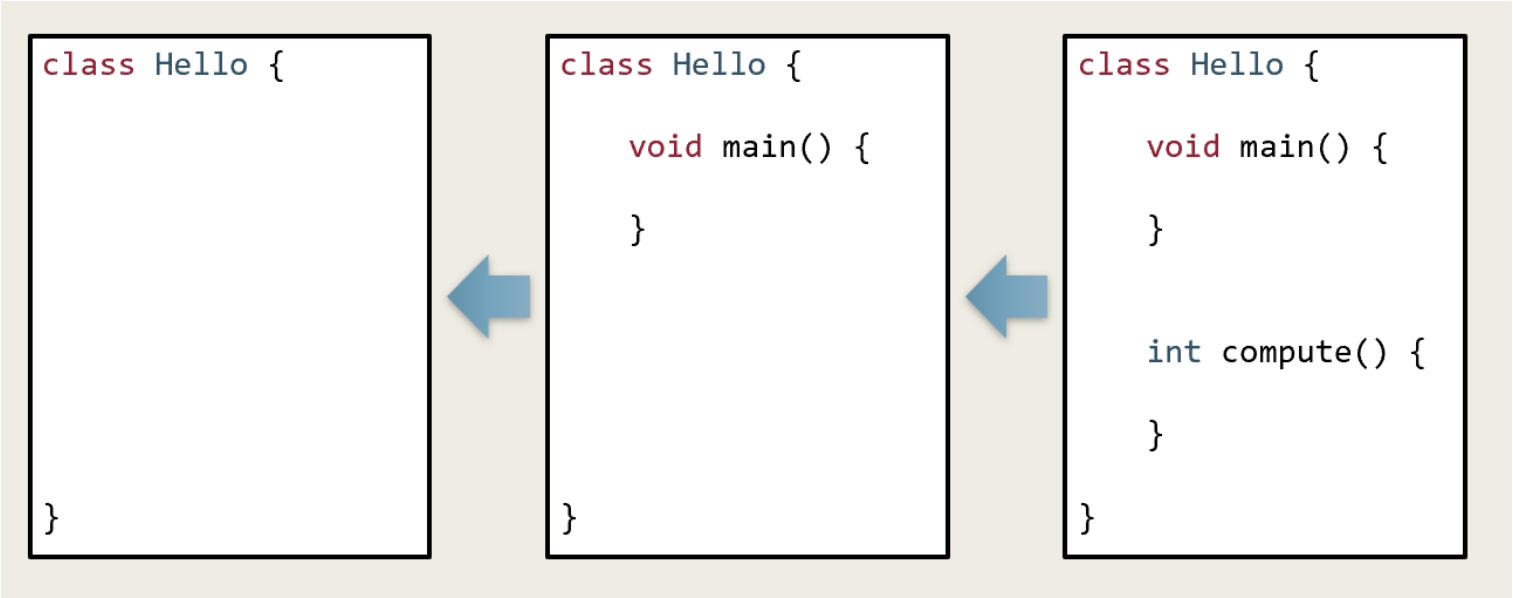
What is Version Control?

Pulling back a bit, Git is software that keeps track of and **controls** various **versions** of your code (hence, "Version Control").
Version Control allows multiple people to work on a project simultaneously by keeping versioned copies of each file in your project for each edit that you make. This makes it easy to:

- Revert files back to a previous state if you make a mistake.
- Look over any changes made by you or your collaborators.
- Recover any files if they are lost.
- Merge changes between multiple people's contributions

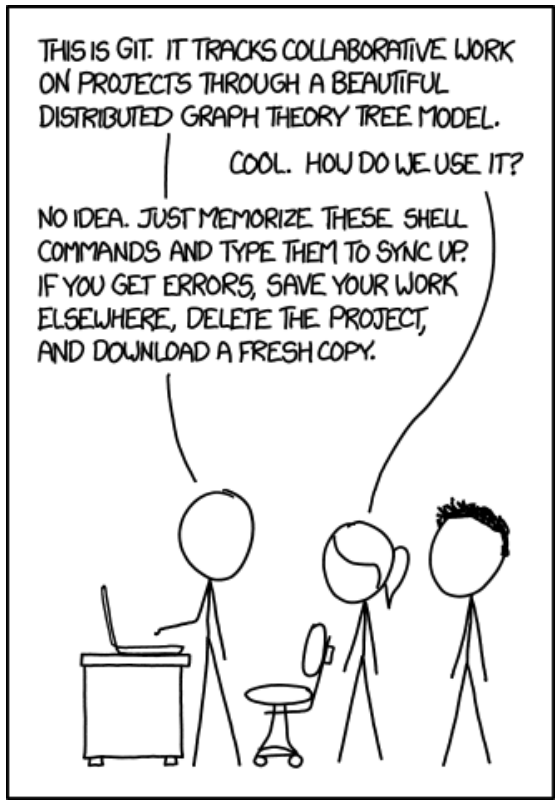
What's a Commit?

Every time you commit your code, you create a snapshot of your code in time.
These snapshots form a linear chain of history, like a linked list.



Git is Complicated

It's actually a bit more complicated, because when you start branching, it turns into a tree, and then you start merging and it's a graph, and the whole thing is beautiful and also not something you need to worry about right this moment.



Comic Credit: Randall Munroe, xkcd.com

Automatic Merge

If two people edit **different files** or **different parts of the same file**, the second person to push will fail. Instead, they will first need to pull the changes.
Git will automatically merge the changes from both users by creating a new "Merge Commit."
Then, push again. This time, it should go through.

CLI users, run `git pull` first.
If it complains about a merge strategy, run `git config --global pull.rebase false` then try again.
After the merge, run `git push`.

Merge Conflicts

If two people edit the **same parts of the same file**, then Git will not be able to automatically merge and you will have a merge conflict.

- **IntelliJ**: a window will pop up which displays your changes compared to those on the server. Click the >> arrows to accept changes from one side or the other (local or remote).

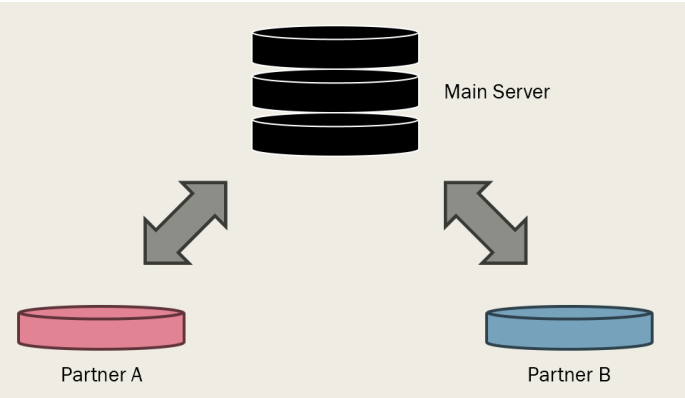
Or, open the conflicting file and you will see something like this:

```
1 <<<<<<< HEAD
2 Your changes
3 =======
4 Partner changes
5 >>>>>>> 123456789
```

Git is marking which parts of the file conflict. Just delete what you don't want, keep what you do, and then run `git commit` again.

Centralized Version Control

So far, you've seen pushing and pulling code from a server.
As such, you may think that version control works by keeping a "main" version and history on a server somewhere, and having individual users work on their portions of it on their machines.
This model is called **Centralized Version Control** and used to be popular. Examples of software include Subversion (previously used by 2112) and Perforce (previously used by Amazon).



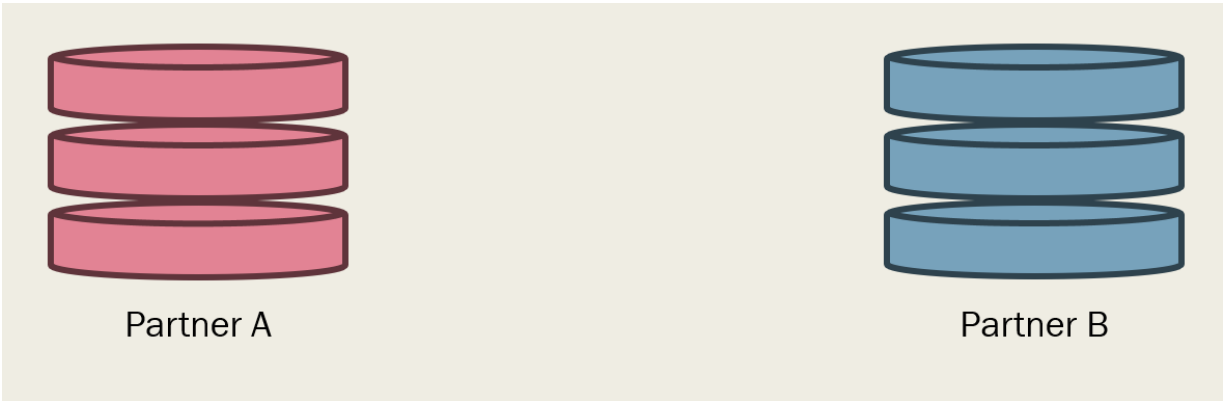
Distributed Version Control

But as you may have guessed by the use of past tense, a new model called **Distributed Version Control** has become much more popular.

In a distributed model, there is no concept of a "main" server.

Everyone keeps a full and complete copy of the entire repository on their own machines.

Examples include Mercurial (used by Facebook) and Git (used by almost everyone else).



Sharing Code w/ Distributed Version Control

In a distributed model, you can set any other repository accessible over the internet as your "remote" and then you can push and pull from it to send changes back and forth.
However, especially with larger groups, sending changes between individual contributors gets hard to manage, so it's conventional to designate one particular repository as the one everyone pushes and pulls from.
GitHub is a service that hosts a repository for you, allowing everyone on your team a convenient place to push and pull from.

Benefits of Distributed Version Control

While this setup ends up looking a lot like Centralized VC, there's multiple benefits to distributed version control.

- You can work offline b/c the whole repo is on your machine
- Commands run faster because nothing talks to a server until you ask it to
- If the server goes down, every individual user has a full backup
- Branching is easier, allowing for better separation of work

Branching

Since a commit in Git is just a snapshot of your code and a pointer to the previous snapshot in history, it's very easy to create a "branch" where you diverge from the main history (in fact, every time you had a merge conflict, a branch was automatically made behind the scenes).

We don't require their use in this class, but branches are very useful especially on larger teams so that different people can work on different features and commit changes without messing with the work of others.

In Git CLI, create a new branch with `git switch -c <branch name>`.

In IntelliJ, click the plus button in the Git window.

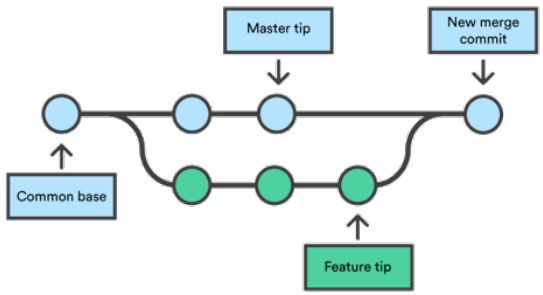
Pull Requests

On larger teams, when someone is done with their branch and wants to **request** that it be **pulled** back into the main branch, it's customary to create a **Pull Request**.
This is a feature of systems like GitHub that lets others review the code written and make comments and suggestions before approving.
When the PR is **merged**, a new merge commit is added on the main branch that also points to your branch commits.
(After this, other people with their own branches can either merge these changes into their branch too, or **rebase** their branch to move their commits on top of the new changes).

Merging (details not covered in lab)

You can merge a branch back into another (such as your main master branch). This creates a new merge commit with two parent commits.

- **IntelliJ**: click the branch icon in the bottom right. In the menu, select the branch to be worked with. A secondary menu will appear with options to merge.
- **Terminal**: move to the destination branch, then run `git merge <branch>`.

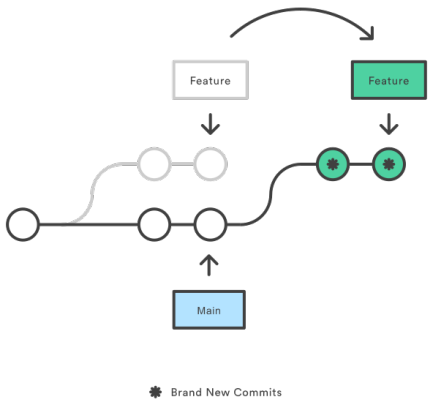


Rebasing (details not covered in lab)

Alternatively, a branch can be rebased, which moves the base of the branch to a new location.

- **IntelliJ**: select Git > Rebase and choose which branch to rebase onto
- **Terminal**: on the branch to rebase, run

  `git rebase <destination_branch>`.



Git Log

Git Log displays a running chronological log of all the commits in your repository and the changes made.

- **IntelliJ**: navigate to "Git" in the menu bar, then "Show Git log".
- **Terminal**: you can run `git log` to see the same information.

Commit Hash

Each commit is identified by a unique hash (or any unambiguous prefix of the hash).

- **IntelliJ**: once the desired commit is selected, the hash is displayed on the bottom right (it will list: `<hash> <author> <email> <date and time>`).
- **Terminal**: `git log` lists the hash next to the word "commit."

Checkout

Your current Git state is just a pointer to a node in the graph of all commits. You can point to a different node by **checking out** the other node.

- **IntelliJ**: Right click a node in the git log and choose "Checkout"
- **Terminal**: `git checkout <hash>`

You can also use this to switch branches. Checkout the latest commit in your branch to go back to normal.

Revert

You can revert a bad commit to undo its changes.
Note that this is an undo, not a rewind. A revert does not rewind a repository to its previous state. Instead, it creates a new commit that does the opposite of the commit you're reverting.

- **IntelliJ**: right click the commit in the git log and choose "Revert Commit"
- **Terminal**: run `git revert <hash>`.

Git Ignore

- Files and folders to not add to version control
  - eg: IDE temporary files (.idea folder)
- .gitignore File
- Goes inside a directory; applies recursively to all subdirectories
- Supports wildcards (*)

Stash

Temporarily store changes for later.

- **IntelliJ**: navigate to "Git" in the menu bar, then "Uncommitted Changes". Options to stash will be displayed.
- **Terminal**: `git stash`.

Diff

Run `git diff <commit1> <commit2>` to get the difference between two commits. Use `HEAD` to represent the current commit.

- **IntelliJ**: click the green and blue rectangles and the gray triangles next to the line numbers on the left of the code to see changes. Or for the full diff, in the toolbar, click the icon with two blue arrows facing one another (next to the push button).

```
dhcp-rhodes-252:lab02 benjamingillott$ git diff 09157c44b7cefd41967941d57d453b68df33463e HEAD
diff --git a/README.md b/README.md
index 2bcfca7..4f2ce3f 100644
--- a/README.md
+++ b/README.md
@@ -2,4 +2,4 @@
 Useful information is contained within this sentence.

 This line was written by B.
-THis line added by mx57
+Testing
dhcp-rhodes-252:lab02 benjamingillott$ git diff 09157c44b7cefd41967941d57d453b68df33463e HEAD > git.txt
```

Blame

Check the last person to edit a particular line in a file.

- **IntelliJ**: use the "Git" menu, then "Current file", then select "Annotate with Git blame".
- **Terminal**: `git blame <file>`.
- Alternatively, go to the GitHub website, find your file in the repo, and click the "Blame" button.

Further Reading

Official documentation: https://www.git-scm.com/docs

GitHub cheatsheet: https://github.github.com/training-kit/downloads/github-git-cheat-sheet.pdf

Turn on Autoformatting

If you have not yet already, turn on Autoformatting in IntelliJ with the following steps:

- Open File > Settings
- Select Tools > Actions on Save
- Check the box "Reformat Code"
- Change the dropdown "Whole File" to "Changed Lines" (you must have a Git repo open to see this option)

Command Line

The following slides were not shown in lab,
but are provided here as a reference.

The shell: a lower-level interface

A *shell* is a command-line interface to your computer's operating system. Less pretty than the GUI interface, more functional.

- Windows: PowerShell, WSL (Windows Subsystem for Linux), Cygwin (Unix on Windows), or cmd
- Unix (Mac OS X, Linux): sh (bash), zsh

Shell commands

A shell command consists of a program name followed by some optional command-line arguments. Spaces separate the command and the arguments from each other.
Examples:

- "rm" ⟨filename⟩ (Windows: "del" ⟨filename⟩) : remove a file
- "ls" ⟨directory⟩ (Windows: "dir" ⟨directory⟩): list contents of directory
- "echo" ⟨message⟩ : print the message to standard output
- "cat" ⟨filename⟩ (Windows: "type" ⟨filename⟩) : print the contents of the file.
- "cd" ⟨directory⟩ : change the current directory (shell built-in command)

Command context

Every command is executed with some context provided by the operating system:

- The *current directory* in which the command runs. All files accesses are relative to this directory. Note: folders are known as *directories* at the operating system level.
- A set of *environment variables* that can be looked up by the running program.[1]
- Standard input and output devices. Normally standard input reads from the shell's input and output goes to the shell. However, it is possible to override these. The syntax "$>$ ⟨filename⟩" *redirects* output to the specified file.

[1] The ShellShock vulnerability exploits a bug in how the "bash" shell handles environment variables.

Pathnames and directories

- Files and directories are specified by *pathnames*: names separated by slashes (Unix, Cygwin) or backslashes (Windows).
- Pathnames starting with a slash (backslash) are *absolute* and start from the root of the file system.
- The special directory "." means the current directory, and ".." means the parent directory of the current directory.
  - To go up a directory: "cd .."
  - To go to the root directory: "cd /"
  - To list the current directory: "ls .", or just "ls".