Learn Git & GitHub Online Beginner & Intermediate Concepts

Learn Git & GitHub - Basic & Intermediate Concepts: The Command Line, Git Version Control, GitHub Collaboration, & More!

Mastering git, github, and version control systems

This introductory section demystifies Git, GitHub, and Version Control Systems (VCS), laying the groundwork for your journey into modern software development.

Navigating this course

- Go where you need to go.
    - notice one section is “For beginners”
- To get your certificate, make sure all curriculum items have a checkmark
- learn by doing
    - to learn git, you have to do it
        - watch me, then do
        - otherwise, it’s a bunch of jargon

curriculum item #001-navigating-course

Thinking about versions of files

old way

- file1
- file2
- file3
- file-finished
- file-finished-really

new way

- instead of creating different files, now we create "snapshot" commits of a file at a certain period in time

curriculum item #002-thinking-versions

Thinking about backing up files

We can push all of our files to a remote computer.
Thus if our computer is destroyed, we can still access all of the files.
curriculum item #003-backing-up

Thinking about sharing files

We can have remote repositories

- PUBLIC / PRIVATE
- SHARE WITH OTHERS

Open source

- When the source code for software is available to anyone

curriculum item #004-sharing

Thinking about collaborating

VCS allows different people to work on the same files
curriculum item #005-collaborating

Linus Torvalds git creation story

Git development was started by Torvalds in April 2005 when the proprietary source-control management (SCM) system used for Linux kernel development since 2002, BitKeeper, revoked its free license for Linux development.
curriculum item #006-linus-torvalds

Normalizing working with git

Normalization

- git can be hard
- git was created by a GENIUS
- git has tons of depth/options
- only Linus Torvalds totally understands git
- our goal: WORKING KNOWLEDGE BETTER THAN MANY

Anecdote - no more coding computer stuff - life tangent here - proceed to next video if you're in a rush

In psychology, normalization refers to the process of making something more "normal" or socially acceptable within a specific context or society. This concept is closely related to the idea of social norms, which are the unwritten rules, expectations, and behaviors that are considered typical or acceptable within a particular group or culture.

Normalization can manifest in various ways within the field of psychology:

1. Mental Health: Normalization in the context of mental health refers to reducing the stigma and negative associations surrounding mental health issues. It involves promoting open conversations about mental health struggles and encouraging individuals to seek help without feeling ashamed or isolated.

2. Deviant Behavior: In some cases, normalization can refer to the process by which deviant behaviors or attitudes become accepted or mainstream. Over time, behaviors that were once considered unusual or unacceptable may become normalized due to shifts in societal attitudes, cultural changes, or advocacy efforts.

3. Social Comparison: People often use normalization as a way to gauge their own behaviors and feelings in comparison to those of others. This can influence how individuals perceive their own experiences as well as how they assess their level of conformity or deviation from the perceived norm.

4. Body Image: In discussions of body image, normalization may involve challenging unrealistic beauty standards and promoting a more diverse and inclusive perspective on body types and appearances. This can help individuals develop healthier attitudes toward their bodies.

5. Abnormal Psychology: In the context of abnormal psychology, normalization can involve understanding and treating psychological disorders. It might involve helping individuals with mental disorders feel understood and accepted while working towards managing their symptoms and improving their overall well-being.

6. Human Development: During different stages of human development, certain behaviors and experiences are considered normal or expected. Normalization helps professionals and caregivers understand what is typical for various age groups, ensuring that developmental milestones are met.

Overall, normalization in psychology acknowledges that what is considered "normal" can be highly subjective and influenced by cultural, social, and historical factors. It encourages a broader perspective that respects diversity, challenges stereotypes, and promotes understanding and empathy for individuals with differing experiences and perspectives.

curriculum item #007-normalize

The path to success

- you're learning a new language - give yourself time
    - the more you speak it, the better you will get at it
- persistently, patiently, you are bound to succeed

curriculum item #008-path-success

Terminal basics: a beginner's guide to navigating the command line

Learn about the terminal, shell, and bash, and the difference between GUI and CLI. Understand Unix/Linux/Mac and Windows terminal specifics. Master essential terminal commands.

The terminal / shell / bash

- terminology
    - GUI = graphical user interface
    - CLI = command line interface - **command line**
        - **terminal** = text input/output environment; console = physical terminal
            - unix / linux / mac
                - **shell / bash / terminal**
            - windows
                - **command prompt / windows command / cmd / dos prompt**

This lecture aims to demystify the terminologies and concepts associated with the terminal, shell, and command line interfaces (CLI). We'll break down:

GUI vs CLI: Understand the difference between a Graphical User Interface (GUI) and a Command Line Interface (CLI), and why learning the latter is crucial for many computing tasks.

Terminal and Console: Clarify the terms 'terminal' and 'console,' and how they differ from each other. Gain insights into how a terminal is a text input/output environment while a console refers to the physical terminal.

Unix/Linux/Mac Environments: Dive into the common terminal-related terminologies used in Unix, Linux, and Mac operating systems, such as 'Shell' and 'Bash.'

Windows Environment: Familiarize yourself with the Windows command prompt, its history, and how it's referred to as CMD or DOS prompt.

By the end of this lecture, you'll have a firm grasp of what these terms mean, how they relate to each other, and what they imply for different operating systems. This knowledge is foundational for anyone aiming to become proficient in programming, system administration, or any field requiring advanced computer skills.
curriculum item #009-terminal-bash-shell

Terminal / shell / bash on Windows

- https://git-scm.com/

curriculum item #010-install-git

Terminal / shell / bash on Apple Mac computers

Terminal / shell / bash – navigation

- shell / bash commands
    - **pwd**
    - **ls**
    - **ls -la**
        - permissions
            - owner, group, world
            - r, w, x
            - 4, 2, 1

d = directory
rwxrwxrwx = owner, group, world

```
                     owner            group       bytes   last modification   hidden & name
total 72
drwxr-xr-x+ 20 spockmcleod  staff    680 Jul 31 15:44 .
drwxr-xr-x   5 root         admin    170 Dec 30  2016 ..
-r--------   1 spockmcleod  staff      7 Dec 30  2016 .CFUserTextEncoding
-rw-r--r--@  1 spockmcleod  staff  14340 Aug  1 06:52 .DS_Store
drwx------  19 spockmcleod  staff    646 Aug  1 07:07 .Trash
drwxr-xr-x  14 spockmcleod  staff    476 Jul 26 13:56 .atom
-rw-------   1 spockmcleod  staff  10321 Aug  1 07:16 .bash_history
drwx------  41 spockmcleod  staff   1394 Aug  1 07:16 .bash_sessions
drwx------   3 spockmcleod  staff    102 Aug 15  2016 .cups
-rw-------   1 spockmcleod  staff   1024 Dec 30  2016 .rnd
drwx------   4 spockmcleod  staff    136 Feb 15 14:48 Applications
drwx------+  4 spockmcleod  staff    136 Jul 26 13:16 Desktop
drwx------+  9 spockmcleod  staff    306 Jul 31 16:43 Documents
drwx------+  4 spockmcleod  staff    136 Jul 31 10:12 Downloads
```

    - **cd**
    - **cd ..**
    - **cd ../../..**

curriculum item #011-navigation

Terminal / shell / bash – manage files & folders

    - **mkdir**
    - **touch**
        - touch temp.txt
    - **nano temp.txt**
    - **cat temp.txt**
    - clear
        - **command + k**
        - **clear**
    - **chmod**
        - chmod options permissions filename

        - chmod 777 temp.txt
    - **env**
    - **rm <file or folder name>**
        - **rm -rf <file or folder name>**

here are all of the commands

- **pwd**
- **ls**
    - **ls -la**
- **cd**
    - **cd ..**
    - **cd ../../..**
- **mkdir**
- **touch**
    - touch temp.txt
- **nano temp.txt**
- **cat temp.txt**
- clear
    - **command + k**
    - **clear**
- **chmod**
    - chmod options permissions filename
    - chmod 777 temp.txt
- **env**
- **rm <file or folder name>**
    - **rm -rf <file or folder name>**

curriculum item #012-manage-files-folders

Terminal / shell / bash – pipe and grep

- using the “|” pipe command to pipe output to another function
    - ls -la | grep -i learn
- **grep**
    - cat temp2.txt | grep enter
    - ls | grep -i documents
- **grep --help**
- all commands

In the bash terminal, you can access help documentation for various commands and their options using the `man` command, which stands for "manual." To view the help documentation for the `-i` flag when using the `grep` command, you can follow these steps: 1. Open your terminal. 2. Type the following command and press Enter:

man grep

(note: this doesn't work for me on gitbash in windows)

3. This will open the manual page for the `grep` command. You can navigate through the manual page using the arrow keys, and you can search for specific terms using the `/` key followed by your search term (in this case, `-i`).

4. Scroll through the manual page to find the information about the `-i` flag, which typically stands for "ignore case" in `grep`. The manual page will provide details about how the flag works and what effect it has on the `grep` command's behavior.

5. **To exit the manual page and return to the terminal, press the `q` key.**

Remember that the `man` command can be used for other commands as well, not just `grep`. You can access detailed documentation for various Unix-like commands by using `man` followed by the command name. **If you're looking for a quick summary of available options without diving into the full manual, you can often use the `--help` flag** as well. For example:

grep --help

This will display a brief summary of the available options for the `grep` command without opening the full manual page.

curriculum item #013-pipe-and-grep

Terminal / shell / bash – environment variables

- **env**
- **echo $<env name>**

Environment variables are a feature of operating systems that allow you to set global variables which can be accessed by any program running on the system. These variables can store configuration settings, file paths, options, and other types of data that may be used by multiple programs.

In Unix-like operating systems such as Linux and macOS, environment variables are often set in shell sessions and can be accessed and modified using shell commands. For instance, the `export` command in the bash shell allows you to set an environment variable:

```bash
export VAR_NAME="value"
```

You can read the value of an environment variable using the `$` symbol:

```bash
echo $VAR_NAME
```

In Microsoft Windows, environment variables can be set and accessed through the system properties dialog box, the Windows command prompt using `set`, or through PowerShell using `Get-EnvironmentVariable` and `Set-EnvironmentVariable`.

Environment variables are widely used for various purposes:

1. Configuration: Many software packages and frameworks use environment variables to store configuration settings. For instance, you might set an environment variable to point to the directory where a particular library is stored.

2. Scripting and Automation: Scripts can use environment variables to store temporary values for use in multiple commands, making the script easier to understand and manage.

3. Inter-Process Communication: While not the most efficient method, some applications use environment variables to pass data between different processes.

4. Security: Sensitive information like API keys or database passwords can be stored in environment variables. However, be aware that this is not the most secure method unless additional security measures are taken.

5. Platform Management: They are used to configure settings that are meant to apply to multiple applications running on the same system.

6. Localization and Internationalization: Settings like language or date formats can be set globally and then accessed by various programs.

Note that **environment variables are typically case-sensitive**, especially on Unix-like operating systems. They are also typically global, meaning that once set, they are available to all processes, although some can be set to be only available to the current process or its child processes. Therefore, care should be taken to avoid naming conflicts and to manage values responsibly.

curriculum item #014-environment-variables

Terminal / shell / bash – setting environment variables

- **env**
- **echo $<env name>**
- **nano ~/.bash_profile**
    - **export MY_VARIABLE="example_value"**
- **source ~/.bash_profile**

Both `.bash_profile` and `.bashrc` are configuration files used in Unix-like operating systems, specifically for the Bash shell. They serve different purposes and are loaded under different circumstances.

.bash_profile

- configure **user-specific settings**
    - environment variables

    - configure the user's PATH
- Executed when a user logs into a computer, either locally or remotely using SSH.

curriculum item #015-set-env-vars

Terminal / shell / bash – using bashrc

.bashrc

- **configure the bash shell itself.**
    - set up aliases
    - define functions
    - customize the shell prompt (PS1)
    - configure other shell-specific behaviors.
- executed each time an interactive shell session is started.

To ensure that the settings in `.bashrc` are also applied when you log in, you can source the `.bashrc` file from within your `.bash_profile`. This way, the settings from `.bashrc` will be loaded with the settings from `.bash_profile`

1. Open `.bash_profile` file using a text editor.

```bash
nano ~/.bash_profile
```

2. Add the following line at the end of your `.bash_profile` file to source (execute) the `.bashrc` file:

```bash
source ~/.bashrc
```

Alternatively, you can use the shorthand notation for sourcing:

```bash
. ~/.bashrc
```

3. Save and exit the text editor.

Now, when you log in or start a new shell session, the `.bashrc` settings will be applied in addition to the settings defined in `.bash_profile`.

Keep in mind that some systems might have slightly different configurations or use a different shell. Always consult your system's documentation or consult with an administrator if you're unsure about the best way to set up your shell environment.

curriculum item #016-bashrc

Terminal / shell / bash – customize your bash prompt

- in .bashrc

- **PS1="\W$ "**

To change the prompt displayed in the Bash terminal, modify the `PS1` environment variable. The `PS1` variable controls the format of the prompt you see before entering commands.

1. Open a terminal window.

2. To see your current prompt configuration, you can simply echo the value of the `PS1` variable:

```bash
echo $PS1
```

This will give you an idea of what the current prompt looks like.

3. To change the prompt, set the `PS1` variable to your desired format. For example, you can set it to something like:

```bash
PS1="\[\e[32m\]\u@\h \[\e[34m\]\w\[\e[0m\]\$ "
```

In this example, the prompt will display the username (`\u`), hostname (`\h`), and current working directory (`\w`) in different colors.

The `\[` and `\]` around **color codes** are used to indicate non-printable characters so that the terminal can properly calculate the prompt's length.

4. To make the changes take effect, either enter the modified `PS1` command directly in your terminal session, or add it to your shell configuration file (like `~/.bashrc` or `~/.bash_profile`) so that it's applied each time you start a new terminal session.

5. If you've made changes to a configuration file, you can either restart your terminal or execute the following command to apply the changes immediately without restarting:

```bash
source ~/.bashrc
```

ESCAPED CHARACTERS

Yes, there are several escaped characters that you can use to display various information in your Bash prompt. These escaped characters are surrounded by `\[` and `\]` to ensure proper formatting and prevent issues with cursor positioning. Here are some commonly used escaped characters for customizing your Bash prompt:

- \u username
- \h hostname
- \w current working directory
- \W basename of current working directory
- \d current date in the format "Weekday Month Day"
- \t current time in the format "HH:MM:SS".
- \n newline (line break).
- \$: displays a $ for regular users or # for root user (super user)
- \\ a literal backslash

- \e an escape character (useful for adding color codes)

Here's an example of how you might use some of these escaped characters to create a custom prompt:

```bash
PS1="\u@\h [\w]\$ "
```

This prompt would display something like:

```
username@hostname [/path/to/current/directory]$
```

You can mix and match these escaped characters to create a prompt that suits your preferences and provides the information you find most useful.

ENVIRONMENT VARIABLE

You can include an environment variable in your Bash prompt by referencing the variable within the `PS1` environment variable setting. Here's how you can do it:

Let's say you have an environment variable named `MY_VARIABLE` and you want to include its value in your Bash prompt. You can modify your `PS1` setting like this:

```bash
PS1="\u@\h [\w] \[$MY_VARIABLE\]\$ "
```

In this example, **`\[$MY_VARIABLE\]`** is where the value of the `MY_VARIABLE` environment variable will be inserted into the prompt. The `\[` and `\]` around it are used to properly handle the variable expansion and formatting in the prompt.

When you use this `PS1` configuration, your prompt will look like:

```
username@hostname [/path/to/current/directory] my_variable_value$
```

Just replace `my_variable_value` with the actual value of your `MY_VARIABLE` environment variable.

COLORS

- Black \[\e[30m\]
- Red \[\e[31m\]
- Green \[\e[32m\]
- Yellow \[\e[33m\]
- Blue \[\e[34m\]
- Magenta \[\e[35m\]
- Cyan \[\e[36m\]
- White \[\e[37m\]
- Light Gray \[\e[90m\]

- Light Red \[\e[91m\]

curriculum item #017-bash-prompt

Terminal / shell / bash – explore git bash settings

- explore established settings **for git-bash**
    - /C/Program Files/git/etc/profile.d
        - cat bash_profile.sh
        - cat git-prompt.sh
- the files we created run after the git profile files
    - ~/.bash_profile
    - ~/.bashrc
- start a new bash
    - old files should load

Files with the `.sh` extension are typically shell script files. These scripts are written for a shell, or command-line interpreter, such as `bash` (Bourne-Again SHell), `sh` (Bourne SHell), or `dash` (Debian Almquist Shell). Shell scripts are used to automate tasks, perform file manipulations, execute commands, and more.

Here is a very simple example of a shell script that prints "Hello, world!" to the console:

```bash
#!/bin/bash

echo "Hello, world!"
```

The first line, `#!/bin/bash`, is known as the shebang. It specifies the interpreter for the script, which in this case is `bash`. After that, any commands you put in the script will be executed in order when the script is run.

You can make a `.sh` file executable by running the `chmod` command on it:

```bash
chmod +x my_script.sh
```

Then you can run it like this:

```bash
./my_script.sh
```

Keep in mind that shell scripts are generally platform-specific. They're commonly used on Unix-like operating systems like Linux and macOS but are generally not directly compatible with Windows (though there are workarounds, such as using a Unix-like environment like WSL, Cygwin, or Git Bash on Windows).

curriculum item #018-git-bash-settings

Terminal / shell / bash – hash algorithms

- **shasum /path/to/file**

Hash Algorithm
A hash algorithm, also known as a **hash function,** is a mathematical function that

- **takes an input (often a message or a file) and produces a fixed-size output, usually in the form of a hash value or digest.**
- The output, known as the **hash code** or **hash value**, is **unique to the input data.** Hash functions are designed to be fast and efficient to compute, and **even a small change in the input data will result in a significantly different hash value.**

Commonly used hash algorithms include **MD5, SHA-1, SHA-256, and SHA-3**. However, it's important to note that some older hash algorithms like MD5 and SHA-1 are no longer considered secure due to vulnerabilities that have been discovered over time. In modern applications, it's recommended to use stronger hash algorithms like SHA-256 or SHA-3.

Hash algorithms are used for various purposes, including data integrity verification, digital signatures, password hashing, and more.

For data integrity verification, the hash value of a file can be calculated before and after transmission, and the recipient can compare the received hash value to the original hash value to ensure that the file has not been tampered with during transmission.

Checksum
A checksum is **a value derived from a set of data,** often by performing a mathematical operation on the data. Checksums are commonly used in error detection in digital communication and storage systems.

The idea is that the sender calculates a checksum based on the data they're sending and includes it in the transmission. The recipient can then independently calculate a checksum based on the received data and compare it to the checksum sent by the sender. **If the checksums match, it suggests that the data was received correctly.**

Checkums can be as simple as a sum of all the bytes in a data block or more complex algorithms that involve bitwise operations. They're often used in networking protocols, file transfers, and storage systems to ensure that data hasn't been corrupted during transmission or storage.

A hash can be used as a checksum. While hash algorithms and checksums serve slightly different purposes, there is some overlap in how they can be used. In some cases, a hash value generated using a hash algorithm can function effectively as a checksum to detect errors or changes in data.

curriculum item #019-hash-algos

Git essentials: installing and using git

See your git version with git version

commands at the terminal

- git
- git version
- git help
    - EBNF

all commands will be highlighted in yellow the first time - for ease of reference

Extended Backus–Naur Form (EBNF) is a way to describe the grammar of a programming language, a file format, or any structured text. It helps you specify the "rules" for what sequences of symbols are valid in that language or format. EBNF is an extension of the original Backus–Naur Form (BNF), and it includes some additional notation to make it easier to write grammars.

Simple EBNF Elements

1. Terminal Symbols: These are the actual symbols of the language you're describing. They are usually in quotes. For example, `"a"` represents the letter a.

2. Non-terminal Symbols: These are placeholders for patterns of terminal symbols. They are usually written in angle brackets like `<expression>`.

3. Sequence: Writing two elements side-by-side indicates they should appear in sequence.
   <letter> <digit>

4. Choice: The pipe (`|`) symbol allows for choices between different elements.
   "a" | "b"

5. Repetition: Curly braces `{}` indicate zero or more repetitions of an element.
   { <digit> }

6. Optional: Square brackets `[]` indicate the element is optional.
   [<whitespace>]

7. Grouping: Parentheses `()` are used for grouping elements.
   ("a" | "b") "c"

Simple Example

Let's say we want to describe a simplified grammar for a calculator that can add and multiply numbers. The EBNF might look something like this:

```
<expression>  ::= <term> { ("+" | "-") <term> }
<term>        ::= <factor> { ("*" | "/") <factor> }
<factor>      ::= <number> | "(" <expression> ")"
<number>      ::= <digit> { <digit> }
<digit>       ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

Here, each rule defines what makes a valid `<expression>`, `<term>`, `<factor>`, `<number>`, or `<digit>`.

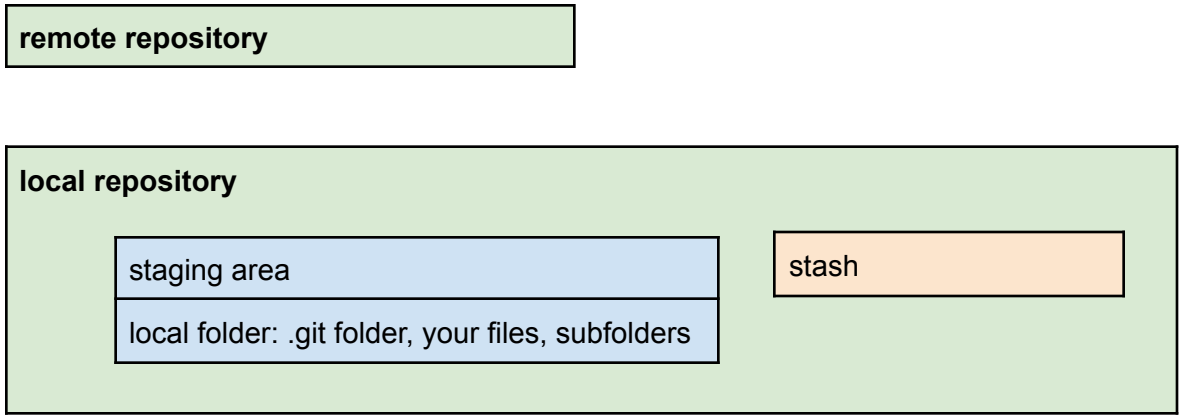
- `<expression>` is made up of one or more `<term>`s separated by either a `+` or `-`.
- `<term>` is made up of one or more `<factor>`s separated by either a `*` or `/`.

- `<factor>` could either be a `<number>` or another `<expression>` enclosed in parentheses.
- `<number>` is one or more `<digit>`s.
- `<digit>` is any single numeral from 0 to 9.

So, according to this simple grammar, "1+2", "3*4+2", and "(1+2)*3" are valid expressions.

curriculum item #020-git-version

How does git work



- your repository is your **REPOSITORY** of all of your file versions, aka, commits
- git init
- staging allows you to group files for a commit
    - illustration
        - add this to the stage
        - add that to the stage
        - don't add this other one
        - take this one off the stage
    - staging allows you to group code you want to commit in one commit
        - shipping analogy - staging christmas presents
            - I have a room full of presents
            - let me use my kitchen table as a stage
                - add presents going to family in Portland
                - get them all on the table - they're all staged
                - now ship them off (commit them)
                - add presents going to family in Hawaii
                - ….

curriculum item #021-how-does-git-work

Initialize a git repository with git init

- git init
- .git folder added
    - "." is a hidden folder

        - files & folders of note:
            - config file
                - local
                    - .git/config
                - global
                    - windows
                        - C:\Users\UserName\.gitconfig
                    - mac
                        - ~/.gitconfig or $HOME/.gitconfig
        - .git/head
            - what is the HEAD pointing to?
        - .git/refs/heads/
            - what is that HEAD shasum?
    - YOU CAN DELETE THIS .git FOLDER AND START ALL OVER AT ANY TIME - TOTALLY FINE!

curriculum item #022-initialize-a-repo

Basic git operations: staging, status, committing, and log

- add
    - git add <filename>
    - git add .
    - git add --all
- status
    - git status
- commit
    - git commit -m “<some message>”
- log
    - git log
    - git log --oneline

The command "git add ." is used to stage changes for commit. The dot (" . ") in this command is a shorthand notation that represents the current directory and all its contents.

curriculum item #023-moving-changes-to-staging

Looking at git commit messages in vs code

- git commit

now you write your message in an editor like vs code first line is main message; subsequent lines show up with git log # are comments

best practice: make your commit message descriptive: short, concise, evocative.

curriculum item #024-commit-messages-vs-code

Using git log to view repository history

Exploring Git Commit History: Basic Techniques with git log

- git log
- git log --oneline
    - git log --oneline -N

      where N is the number of commits you want shown

CHECKSUM commit ID
curriculum item #025=commits-part-1

How to navigate results in the terminal: shortcut keys

- space or f
- b
- ⬆ ⬇
- q

curriculum item #026-terminal-navigation

Filtering git log by author, keywords, and ranges

- git log --author=”Todd”
- git log --grep=”Happy”

  grep = "Global Regular Expression Print"
- git log <treeish>..<treeish>

  from..to

  example: git log <treeish>..HEAD
- git log <filename>

curriculum item #027-commits-part-2

Looking at options for git log: patch and stats

- git log -p

  shows every commit and the changes
- git log --stat

The **`-p`** option in the `git log` command stands for "patch." When you use **`git log -p`**, you'll not only see the commit history, but you'll also get t**he patch output** (i.e., the differences) for each commit. This makes it easier to understand what changes were introduced in each commit.

Normally, **`git log` just shows you a summary line for each commit: the commit hash,**

the author, the date, and the commit message.

The `-p` option expands on this by also showing the differences (in diff format) between each commit and its parent.

This can be very useful for code reviews or when you're trying to understand the history of a project. The diff output helps to highlight what was changed in the source code for each commit.

The **`git log --stat`** command displays the commit history along with some basic **statistics** for each commit, indicating **which files were modified** and **how many lines were added or deleted.**

The `--stat` option shows a diffstat of what each commit changes. This **makes it easier to understand the impact of each commit in terms of files and lines without showing the actual line-by-line changes (as you'd get with `git log -p`).**

This format can be very helpful for quickly scanning through the commit history to understand the breadth and impact of each commit without needing to see the detailed changes.

curriculum item #028-commits-part-3

Filtering commits using git log by time range

- git log --after=”4 weeks ago” --before=”2 days ago”
  shows every commit and the changes

The **`--after`** and **`--before`** options in the `git log` command filter the commit history based on the specified time range. **Git is quite flexible** in the time periods it accepts. You can use a variety of formats to specify the time, and yes, you can use minutes and hours as well.

Here are some examples of different time periods you can use:

- `--after="5 minutes ago"`
- `--before="3 hours ago"`
- `--after="2 days ago"`
- `--before="6 months ago"`
- `--after="1 year 2 months 3 days 4 hours ago"`

You can also use specific dates in a variety of formats:

- `--after="2021-01-01"`
- `--before="2021-12-31 23:59"`

You can even combine them:

- `git log --after="2 days ago" --before="3 hours ago"`

The flexibility allows you to narrow down your search quite effectively, making it easier to browse through the commit history.

Here's a more comprehensive list of units you can use:

- Minutes: `minutes` or `minute`
- Hours: `hours` or `hour`
- Days: `days` or `day`
- Weeks: `weeks` or `week`
- Months: `months` or `month`
- Years: `years` or `year`

Remember that you can combine them into a single statement like "1 year 2 months 3 days 4 hours ago" to be even more precise.

curriculum item #029-commits-part-4

Using git log to visualize your repository history

- git log --format=short
  oneline | short | medium (default) | full | fuller | email | raw
  - git log --oneline
- useful for branches
  - git log --graph
    for branches
  - git log --graph --all --oneline --decorate

curriculum item #030-commits-part-5

Exploring a directory tree

Exploring a directory tree in git

- git ls-tree <treeish>
- git help ls-tree

A TREE is a directory structure with sub-directories and files

In git, a "tree" refers to a fundamental data structure that represents the state of a directory at a specific point in time. It's essentially a snapshot of the directory's contents, including subdirectories and files, along with their metadata such as names, permissions, and pointers to the corresponding blobs (file content) or subtrees (subdirectory contents).

A tree object in git contains the following information for each entry in the directory:

1. Mode: Specifies whether the entry is a regular file, executable file, subdirectory, etc.
2. Type: Specifies whether the entry is a "**blob**" (file content) or a "tree" (subdirectory content).
3. Object Hash: The unique SHA-1 hash of the corresponding blob or subtree object.
4. Name: The name of the file or subdirectory.

Trees are an integral part of git's internal data structure that enables it to efficiently store and track changes to the contents of a repository. When you make a commit, git creates a new tree object that represents the state of the directory at that commit, linking to the appropriate blobs for file contents and subtrees for subdirectories. This structure allows git to maintain a complete history of changes while minimizing duplication of content.

To visualize it:

```
Tree Object:
- Mode: 100644 (file)
- Type: blob
- Object Hash: a1b2c3... (SHA-1 hash of file content)
- Name: example.txt

- Mode: 040000 (directory)
- Type: tree
- Object Hash: d4e5f6... (SHA-1 hash of subtree)
- Name: subdirectory
```

Internally, git uses a linked list architecture for its internal implementation of version control. The fundamental data structure in git is a **directed acyclic graph (DAG)** composed of commits. **Each commit points to its parent commit or commits, forming a linked list-like structure.**

It's important to note that git's internal representation is not a traditional linked list in the sense of a single chain of nodes. Instead, it's a more complex graph structure. **Each commit in git has a unique hash (SHA-1 or a more secure hash function in recent versions of git) that acts as its identifier. Each commit points to one or more parent commits, forming a history of changes.**

This graph structure allows git to efficiently represent branching and merging, which are core features of version control. When you create a new branch or perform a merge, git creates new commits with multiple parent references, forming a graph that shows how different branches and changes have evolved over time.

https://chat.openai.com/share/aa542aa4-b854-4fb3-b23c-941e9137dccb

In git, **"treeish" references** are ways to specify a specific state of the project's directory structure. They're typically used to refer to different snapshots of your repository's content. Here are some specific examples of treeish references:

1. **Commit Hash**: A commit hash uniquely identifies a specific commit. You can use the full or abbreviated hash to refer to the tree of that commit. For example:
   ```
   ```

```
git diff abc1234      # Diff against the commit with hash abc1234
git show def5678      # Show the changes in commit def5678
```

2. **Branch Name:** Branches in git point to specific commits. When you use a branch name as a treeish reference, git uses the latest commit in that branch's history. For example:

```
git diff main         # Diff against the latest commit on the 'main' branch
git log feature/xyz   # Show commit history on the 'feature/xyz' branch
```

3. **Tag Name:** Tags also point to specific commits. Using a tag name as a treeish reference will refer to the commit associated with that tag. For example:

```
git show v1.0.0       # Show the changes in the commit tagged as v1.0.0
```

4. **Ancestry Notations:** git provides notations for specifying commit ancestry. For instance, `^` and `~` can be used to reference parent commits and grandparent commits, respectively. For example:

```
git diff HEAD^        # Diff against the parent commit of the current commit
git diff HEAD~2       # Diff against the grandparent commit of the current commit
```

5. **Relative Refs:** You can use relative references to specify commits relative to a starting point. For instance, `HEAD~n` refers to the nth ancestor of the current commit. Also, you can use the caret (^) to move upwards in the commit hierarchy. For example:

```
git diff HEAD~3..HEAD~2   # Diff between the 3rd and 2nd ancestors of the current commit
git diff HEAD^^^..HEAD~2  # Same as above using caret notation
```

6. **Commit Ranges:** You can specify a range of commits using double-dot notation. This will include all the commits reachable from the second commit but exclude those reachable from the first commit. For example:

```
git log commit1..commit2   # Show commits reachable from commit1 exclusive to commit2 inclusive
```

7. **File Revisions:** You can also use treeish references to specify the state of a single file in a specific commit. For instance:

```
git show commit:path/to/file.txt   # Show the content of 'file.txt' in the specified commit
```

These are just a few examples of treeish references in git. They allow you to navigate and work with different snapshots of your repository's content using various forms of references.

curriculum item #031-explore-dir-tree-in-git

Comparing log & show in git

- git show <treeish>
- git log
  - view the history of commits in a repository branch
- git show
  - display info about a specific commit, including changes in that commit.

`git log` and **`git show`** are both commands in the git version control system, but they serve different purposes and provide different kinds of information about the commit history.

1. **git log**:
   The `git log` command is used to display the commit history in a repository branch. When you run `git log`, it shows a list of commits in reverse chronological order (newest to oldest), with each commit's hash, author, date, and commit message. By default, it displays a concise version of the commit information.

   You can use various options and flags with `git log` to customize the output. For example:
   git log -p Shows the commit history along with the diff introduced by each commit.
   git log --oneline Shows a simplified one-line representation of each commit.
   git log --author=<author> Filters the log to show commits made by a specific author.
   git log <branch> Shows the commit history of a specific branch.

2. **git show**:
   The `git show` command is used to display detailed information about a specific commit, including the commit's changes, author, date, and commit message. When you run `git show <commit_hash>`, it shows information about that specific commit. The commit hash can be obtained from the `git log` output.

   The `git show` command also displays the diff of the changes introduced by the commit, making it useful for reviewing the exact changes made in that commit. Additionally, you can specify options to control the display of information, such as `git show --stat` to show a summary of the changes.

curriculum item #032-git-show

Understanding ancestry terminology in git

commit - parent - grandparent - great grandparent ….

this way	or this way
• git show <treeish>^ • git show head ^^ • git show head ^^^	• git show <treeish>~1 • git show head~2 • git show head~3 • git show head~4

or use any reference to any treeish other than head, such as a checksum
curriculum item #033-git-ancestry

Exploring ancestry in a git directory tree

- git ls-tree <treeish>
- git ls-tree <treeish>^
- git ls-tree <treeish> <path>
- git ls-tree <treeish> <path>/

curriculum item #034-git-dir-tree

Global variables, local variables, & git config

An introduction to git config

local

- .git/config

global

- C:\Users\UserName\.gitconfig
    - ~/.gitconfig
- git config --list
- git config --user.name “Todd McLeod”
- git config --user.email “0074324@uk.gov”

curriculum item #035-global-git-config

Getting started with github

Creating a remote repo - step 0, create an account

- remote repos
    - GITHUB
    - GITLAB
    - BITBUCKET

curriculum item #036-github-create-account

Creating a remote repo - step 1, SSH

- google: how to setup an ssh key on github

Asymmetric Encryption (Public/Private Key Encryption) In asymmetric encryption, two different keys are used: a public key and a private key. The public key is used to encrypt data, and the corresponding private key is used to decrypt it. - Public Key: This is a key that is freely shared and can be distributed to others. Anyone can

use this key to encrypt a message intended for you.

- Private Key: This key is kept secret and only known to the owner. It is used to decrypt messages encrypted with the corresponding public key.

#How it Works
1. When Alice wants to send Bob an encrypted message, she looks up Bob's public key and uses it to encrypt the message.
2. The message is sent to Bob, who then uses his private key to decrypt it.

#Advantages and Disadvantages

- Advantages: More secure for key distribution; each participant has their own pair of keys, so compromising one doesn't lead to the compromise of others' communications.

- Disadvantages: Slower and computationally more intensive than symmetric encryption; often not suitable for encrypting large volumes of data directly.

Symmetric Encryption

In symmetric encryption, the same key is used for both encrypting and decrypting data.

#How it Works
1. Alice and Bob agree on a secret key (this is the tricky part in a pure symmetric system).
2. Alice uses this key to encrypt a message and sends it to Bob.
3. Bob uses the same key to decrypt the message.

#Advantages and Disadvantages

- Advantages: Faster and less computationally intensive; more suitable for bulk encryption of large volumes of data.

- Disadvantages: Key distribution is challenging and insecure. If someone gains access to the key, they can decrypt all messages encrypted with that key.

How They Relate in Secure Communication

In real-world applications, asymmetric and symmetric encryption are often used together to exploit the strengths of each.

1. Key Exchange: Asymmetric encryption is first used to securely exchange a symmetric key between two parties (Alice and Bob, for example). This could be done using algorithms like RSA or Diffie-Hellman.

    - Alice and Bob each generate a public-private key pair.
    - They exchange public keys.
    - Using their own private keys and each other's public keys, they derive a shared symmetric key.

2. Bulk Data Transfer: Once the symmetric key is securely exchanged, it is used for the bulk encryption and decryption of the actual data being communicated. This is usually done using faster symmetric algorithms like AES (Advanced Encryption Standard).

By using this combination, you get the best of both worlds: the secure key distribution of asymmetric encryption and the speed and efficiency of symmetric encryption.

curriculum item #037-github-ssh-key

Creating a remote repo - step 2, create repo

curriculum item #038-github-create-repo

Creating a remote repo - step 3, pushing code

- git push
- at terminal:
- cat .git/config
- git remote
- git remote -v

curriculum item #039-github-push-code

Seeing different versions of files at different points in time

curriculum item #040-seeings-different-versions

Understanding the local and remote repo

REMOTE (origin) MACHINE
main
24ad18d - 21475a2 - fe9cf34 - 968bdd5

LOCAL MACHINE
origin/main
24ad18d - 21475a2 - fe9cf34 - 968bdd5
main
24ad18d - 21475a2 - fe9cf34 - 968bdd5

curriculum item #041-understanding-local-remote

Ignoring files in a git repository

- .gitignore
- ignore a specific
    - FILE
        - example: filename.txt
    - FOLDER
        - example: logs/
    - FILES IN A FOLDER
        - example: stuff/*.stf

curriculum item #042-gitignore

Understanding the readme file

- readme.md

  md = markdown

curriculum item #043-markdown

Working with remote repositories in git

git clone

- git clone <info here> [<directory>]
- [<directory>] will give the folder a different name on your computer

curriculum item #044-git-clone

git fetch & merge

- git fetch
- git merge
- git pull

  git pull = git fetch + git merge

curriculum item #045-git-fetch-merge

git pull

- git fetch
- git merge
- git pull

  git pull = git fetch + git merge

curriculum item #046-git-pull

The pareto principle

pull before push

you will this 80% of the time (the pareto principle - Vilfredo Pareto & his tomatoes)

- git fetch or git pull
- git status
- git add .
- git commit -m “some message”
- git push

curriculum item #047-pareto

Using VS Code with git & GitHub

Installing VS Code

- INSTALLING
- extensions
    - go to the activity bar
        - if you don't see it
        - view / appearance / activity bar
    - go to extensions & search for "git"
        - github pull requests and issues
        - github repositories
        - git graph
        - gitlens
        - git history

curriculum item #048-install-vs-code

Launch VS Code from bash

- code
- code .

  launches vs code
- if this isn't working, in VS Code, command palette: "Shell"
    - shell command: install code command in path

curriculum item #049-launch-vs-from-bash

Terminal in VS Code items

- terminal
    - view terminal
    - terminal / new terminal

curriculum item #050-terminal-in-cs-code

Seeing differences in files

See differences - command line interface (CLI)

- git diff
  shows changes to file(s) not yet staged
- git diff --color-words
- git diff --cached
  shows changes to file(s) that have been staged
- git diff <commit>..<commit>
- git diff --color-words <commit>..<commit>
- git diff <branch>..<branch>

The **`--color-words`** flag in the **`git diff`** command is used to show changes with word-level granularity, as opposed to the default line-level granularity. This is particularly useful when you want to see exactly what words have been added or deleted within a line, without showing the entire line as added or removed.

By default, `git **diff` uses lines as the smallest unit of change.** When a word in a line changes, the whole line is considered changed, and `git diff` would show you the line removed and the line added. When you use **`--color-words`**, `git diff` will instead focus on individual words within those lines. Words that have been removed will be highlighted in one color, and words that have been added will be highlighted in another color, making it easier to see what exactly changed within the line.

curriculum item #051-git-diff

See differences - vs code (GUI)

- at terminal
  - code .
- make changes to files
- notice GIT icon with NUMBERS NOW
- click on the files to see the differences

curriculum item #052-differences-in-vs-code

See differences in GitHub (remote repo)

- click on COMMITS
- click on a COMMIT

curriculum item #053-see-diffs-on-gituhub

See a specific commit

- git log
- git show <treeish>
    - git show <commit>
    - git show HEAD
    - git show HEAD --color-words

Where is your **head** at?

In git, the term **"HEAD"** refers to a symbolic reference that points to the most recent commit in the currently checked-out branch. In simpler terms, it's a pointer to the latest commit in the branch you are currently working on.

The HEAD reference is important because it represents the state of your working directory and the commit you are currently on. When you make new commits, the HEAD reference is updated to point to the latest commit you've just created.

There are a few important things to note about the HEAD:

1. *Detached HEAD State:* Sometimes, the HEAD can be in a "detached" state, which means it is not pointing to the tip of a branch, but directly to a specific commit. This can happen, for example, when you check out a previous commit by its hash or when you're in the middle of a rebase or merge operation.

2. *Switching Branches:* When you switch to a different branch, the HEAD reference is updated to point to the latest commit of the newly selected branch.

3. *Creating New Commits:* When you create a new commit, the HEAD reference moves forward to point to this new commit, making it the latest commit in the current branch.

4. *Moving HEAD Manually:* While it's not common, you can manually move the HEAD reference using commands like `git reset` or `git checkout`, though these actions should be performed with caution as they can alter your commit history.

Overall, the HEAD reference is a crucial concept in git, as it helps you keep track of your current position within the commit history and your ongoing work.

curriculum item #054-git-show

Search for a specific commit with grep

- search for author
    - git log --author=todd
- search for a file
    - git log italy.txt
- search a commit message
    - git log --grep='<regular expression>'

grep = "Global Regular Expression Print"

we saw this before

- git log --grep=italy -i

curriculum item #055-search-grep

Pareto operations in vs code

- at terminal
  - code .
- make changes to files
- notice GIT icon with NUMBERS NOW
- add files with "+" to staging area
- give a COMMIT MESSAGE
- COMMIT
- PUSH (if connected to remote repo)
  - "sync changes" = push & pull
- "..." option

curriculum item #056-pareto-ops-vs-code

File and folder operations

Recovering a file's contents

delete the file, then:

- git pull
- git add .
- git commit -m "deletes this file"
- git push
- look at the remote repository

HEAD -> main: Local current branch.
origin/main: Remote equivalent of your local main branch as of last sync.
origin/HEAD: Default branch on the remote.

REMOTE (origin/HEAD) MACHINE
main
24ad18d - 21475a2 - fe9cf34 - 968bdd5

LOCAL MACHINE
origin/main
24ad18d - 21475a2 - fe9cf34 - 968bdd5
main
24ad18d - 21475a2 - fe9cf34 - 968bdd5

curriculum item #057-recover-file

Renaming a file - vs code (GUI)

- vs code
- bash
    - mv
- git
    - git mv ./oldfilename.txt ./newfilename.txt

curriculum item 058-rename-file

Adding two folders

- GIT DOESN'T TRACK EMPTY FOLDERS
- if you want an empty folder, add this dummy file to it:
    - .gitkeep

curriculum item #059-two-folders

Understanding restore, revert, and reset

Restore things in your working directory

- git status
- git restore --staged <treeish>
    - git restore --staged <filename>
    - git restore <filename>

`git restore`
undoes changes in your working directory or staging area. The `git restore` command can be used with various options to specify what you want to restore:

1. Working Directory Changes: If you've made changes to files in your working directory but haven't staged them, you can use `git restore <file>` to discard those changes and revert the file to the last committed version.
   git restore myfile.txt

2. Staging Area Changes: If you've staged changes using `git add` but want to unstage them, you can use the `--staged` option with `git restore`.
   git restore --staged myfile.txt

curriculum item #060-git-restore

Undo a commit with git revert

- git log --oneline
- git revert <treeish>

curriculum item #061-git-revert

Comparing restore, revert, reset - soft reset

- git reset --soft <treeish>
    - --soft
    - --mixed
    - --hard
- be careful using resets with shared repos

Git provides multiple commands to help you undo changes, but they serve different purposes and work in slightly different ways:

- **git restore**
- **git revert**
- **git reset**

RESTORE
undo changes in your **working directory** or **staging area**.
examples:

- **git restore <file>**
- restore --staged <file>
- git restore --source=HEAD --staged --worktree <file>
    - restores files in working directory and staging area

REVERT
create a new commit that **undoes** the changes made in a specific **previous commit**. This operation is safe to use on public commits.
git revert <commit_hash>

After running the command, Git will create a new commit that undoes the changes made in the commit identified by `<commit_hash>`. This is the safest way to "undo" a public commit.

	git restore	**git revert**	**git reset**
affects	Working directory and/or staging area	Commit history	HEAD, staging area, and/or working directory
changes history	no	yes, adds a new commit	yes, if applied to commits
safe in public repos	safe	safe	unsafe for public commits

RESET

a more powerful command; primarily operates on **commits** and the **staging area**. There are three types of resets:

--soft

- branch pointer & HEAD move
  - **git reset --soft <commit_hash>**
- **staging area** *preserved* *(no change)*
- **working directory** *preserved* *(no change)*
  - staged ***changes remain***
  - working directory ***changes remain***
  - ***keep the changes in your working directory and staging area but move the HEAD pointer to a specific commit***

--mixed` (default):

- branch pointer & HEAD move
  - **git reset --mixed <commit_hash>**
  - **git reset <commit_hash>**
- **staging area** *wiped* *(match repo you switch to)*
- **working directory** *preserved*
  - working directory ***changes remain***
  - ***move the HEAD pointer to a specific commit and update the staging area to match, but leave your working directory alone***

--hard

- branch pointer & HEAD move
  - **git reset --hard <commit_hash>**
- **staging area** *wiped* *(match repo you switch to)*
- **working directory** *wiped* *(match repo you switch to)*
  - ***move the HEAD pointer to a specific commit, and change both your staging area and working directory to match that commit***

Note: Using `git reset` on public commits, i.e., commits that you have already pushed to a shared repository, is generally discouraged as it can lead to a confusing history for other developers. This command should be used carefully.

curriculum item #062-git-reset-soft

git reset mixed

- git reset --mixed <treeish>

curriculum item #063-git-reset-mixed

git reset hard

- git reset --hard <treeish>
  - you can reset to another branch

curriculum item #064-git-reset-hard

Using git tag for semantic versioning

Semantic versioning

https://semver.org/

curriculum item #065-sem-version

Best documentation for git

- https://git-scm.com/book/en/v2
- https://git-scm.com/book/en/v2/Git-Basics-Tagging

curriculum item #066-git-doc

git tag

- see tags
    - git tag
- create tags
    - lightweight
        - git tag v1.0.1
    - annotated
        - git tag -a v1.0.2 -m "my version 1.0.2"
            - -a for annotated
            - -m for message
- Push tags to origin
    - git push origin --tags

curriculum item #067-git-tag

Copy and move a file in bash & git show a specific tag

- git show <commit hash> --color-words
- **git show v1.0.1**

In a Bash terminal, you can accomplish this using the `cp` command to create a copy of the file, and then use the `mv` command to move the copied file to a new directory. Alternatively, you can directly copy the file to the new directory in one step using `cp`.

Copy then Move

Suppose you have a file called `file.txt` in your current directory and you want to create a copy of it called `file_copy.txt`, then move it to a directory called `new_directory`. Here's how you can do it:

1. Copy the file

   ```bash
   cp file.txt file_copy.txt
   ```

   This creates a copy of `file.txt` and names it `file_copy.txt`.

2. Move the copied file

```bash
mv file_copy.txt new_directory/
```

This moves `file_copy.txt` to `new_directory`.

Copy Directly to New Directory

You can also do this in one step:

```bash
cp file.txt new_directory/file_copy.txt
```

This will create a copy of `file.txt` in `new_directory` and name it `file_copy.txt`.

Make sure the directory exists

Before moving or copying the file, make sure that the directory `new_directory` exists. If it doesn't, you can create it using:

```bash
mkdir new_directory
```

Summary

To sum it up, you can either:

- Copy the file and then move it:

  ```bash
  cp file.txt file_copy.txt
  mv file_copy.txt new_directory/
  ```

- Or copy it directly to the new directory:

  ```bash
  cp file.txt new_directory/file_copy.txt
  ```

Either way will result in `file_copy.txt` being placed into `new_directory`.

curriculum item #068-cp-mv-git-show

Tag an old commit

- post-facto tag
  - git tag -a v1.0.1 <commit hash>

curriculum item #069-tag-old-commit

Delete a tag

- post-facto tag - you can also do it without the `-a` flag
    - git tag v1.0.1 <commit hash>
- delete a tag
    - delete a local tag
        - git tag -d v0.0.4
    - delete a remote tag
        - git push origin --delete <tagname>

curriculum item #070-delete-tag

Working with branches in git & merging branches

An introduction to branches

- git branch
    - “*” shows you which branch you’re on
    - the primary branch naming convention: main or master
        - negative associations with “master”

Branches, margin, and pull requests

Branches, merging, and pull requests are related concepts but serve different purposes. Understanding their roles can help you collaborate more effectively in a codebase.

Branches

A branch in Git represents a separate line of development. You can think of a branch as a way to encapsulate changes that make up a feature or a fix. By developing on a branch, you can isolate your changes, making it easier to manage features, bug fixes, experiments, etc.

When you create a branch and switch to it, you start with a copy of the codebase as it exists in the branch you branched from. Any subsequent changes you make will only affect this new branch. This makes it possible to experiment freely and develop new features without affecting the main line of development, often referred to as the "main" or "master" branch.

Pull Requests

A Pull Request (PR), which is a concept native to platforms like GitHub, GitLab, and Bitbucket, is a request to merge one branch into another. It is essentially a mechanism for a developer to notify team members that they have completed a feature. The pull request provides the following functionalities:

1. Code Review: It opens the door for other developers to review, comment on, and eventually approve the changes.

2. Conflict Resolution: It provides a space to resolve any conflicts that may arise due to changes in the main line of development that happened concurrently with the feature development.

3. Documentation and Context: It offers an opportunity to document the changes, link related issues, and provide context around decisions made during development.

Relationship

Here's how branches and pull requests often interact in a typical workflow:

1. Create a Branch: You start by creating a new branch off the main branch (or another base branch if appropriate). This new branch will house the changes for the new feature you are developing or the bug you are fixing.

2. Development: You switch to the new branch and develop the feature, making one or many commits to this branch.

3. Push the Branch: After you've implemented the feature or fix, you push the branch to the remote repository.

4. Open a Pull Request: You then create a pull request, asking that the changes in your feature branch be merged into the main branch (or another target branch).

5. Review and Discuss: Team members review the code, discuss changes, and suggest modifications. You may push additional commits to your feature branch during this stage, which automatically get added to the existing pull request.

6. Resolve Conflicts: If there are conflicts with the main branch, they must be resolved in the scope of the pull request.

7. Merge: Once everyone is satisfied with the feature and all conflicts are resolved, the pull request is merged into the main branch, and the feature branch is often deleted.

8. Close the Pull Request: After merging, the pull request is closed, serving as a historical record of the changes made, reviews, discussions, and any linked issues.

So, in summary, branches serve as isolated environments for individual tasks or features, while pull requests are the bridge that helps integrate these isolated changes back into the main line of development, fostering collaboration and code quality along the way.

Here are some examples of how branches can be used:

Isolation of Work: Each branch can represent a separate line of development, such as a new feature, bug fix, or experiment. This isolation prevents changes in one branch from directly affecting the code in other branches.

Collaboration: Teams of developers can work concurrently on different branches, making it easier to manage and merge their changes later.

Versioning: Branches allow you to maintain multiple versions of your codebase simultaneously. This is useful for maintaining stable releases while actively developing new

features.

Feature Development: New features or enhancements can be developed in dedicated branches, making it easier to manage the development process and track progress.

Bug Fixing: Branches can be created specifically to address bugs in the code. Once fixed, the changes can be merged back into the main development line.

Experimentation: Developers can create branches to try out new ideas or approaches without affecting the main codebase. If the experiment is successful, the changes can be integrated; otherwise, the branch can be discarded.

The default branch in most git repositories is typically named "master" or **"main,**" but you can create additional branches with custom names. As you make changes in a branch, the branch's pointer advances with each new commit you make. When you create a new branch, it starts with the same content as the branch you created it from, but as you make changes in one branch, those changes don't affect the other branches until you explicitly merge them.

Merging is the process of integrating the changes from one branch into another. This allows you to bring the changes made in one branch into another branch, incorporating new features, bug fixes, and other updates.

curriculum item #071-git-branch

Creating a new branch

you branch from the branch you're on

- git branch name
    - no spaces in the name
- git branch
- we can explore in our computer's file directory:
    - .git/head
        - cat head
    - .git/refs/heads
        - ls -la
        - cat <branch name>
            - shows us the shasum

curriculum item #072-create-branch

Switching between branches

commit changes before switching to another branch

- git checkout <branch name>
- make changes
    - git add .
    - git commit -m "some message"
    - git log --oneline

curriculum item #073-switch-branch

Merging branches

When you are one branchX …

- git merge <branchY>

… branchX has branchY merged into it

best practice: make small changes, commit → **MERGE OFTEN** (get that code onto the company's server)

curriculum item #074-merge-branch

Seeing the differences between branches

- git diff <branch1>..<branch2>

  "from branch1 to branch2, how did things change?"

  "from a to b, how did things change?"

curriculum item #075-git-diff-branches

Deleting a branch

- git branch -d <branch name>
  - you can't delete a branch you're on
  - you must force delete a branch with commits that haven't been merged

curriculum item #076-del-branch

Visually seeing your branches with GAOD

- git log --graph
- git log --graph --all --oneline --decorate

  acronym: gaod

curriculum item #077-git-log-GAOD

Checking out a treeish, detached head, branching

- you can checkout a commit
  - but then your head is detached from the front of the branch
- two options from here
  - you can attached your head again
    - git checkout main
    - git switch -
  - you can branch off of this checked out commit
    - git switch -c <some-branch-name>
      - if you create a new branch
        - make changes
          - git add .
          - git commit -m "commit msg"
        - switch to main
          - git merge <branch name>
        - remove old branch

- git branch -d some-branch-name

The term **"detached head"** happens when YOUR HEAD (the thing on top of your shoulders that has eyes, a mouth, and a nose) is looking at something other than the most recent commit on a branch, eg, your head is somewhere than at the front of a branch.

In normal “git terminology” we would say the above like this, a **“detached head”** state happens when you are not on a named branch but are instead directly pointing to a specific commit in the repository's history.

This situation can occur when you checkout a commit hash, tag, or other reference that points to a specific commit.

When you are on a named branch, git knows which branch you are working on and updates that branch as you make new commits. However, when you are in a detached head state, git is not tracking any branch, and any new commits you make will not be associated with any branch.

This state is often encountered when you do things like:

1. Checking out a specific commit hash directly.
2. Checking out a tag.
3. Checking out a remote tracking branch directly (without creating a local branch).

For example, if you run the command `git checkout <commit_hash>`, you will end up in a detached head state. You can still make new commits, but it's important to note that ***these commits won't be part of any branch's history by default. If you switch to a different branch or commit, you might lose track of your commits in the detached head state, potentially making it difficult to reference them later.***

To avoid losing your work and to make it easier to manage your changes, it's generally recommended to create a new branch when you find yourself in a detached head state. This way, you can give a meaningful name to your branch and associate your commits with it, making them easier to manage and reference.

To recover from a detached head state and create a new branch from the current commit, you can use the command `git checkout -b <new_branch_name>`. This will create a new branch based on your current commit and switch to it, so you can continue your work on a named branch.

curriculum item #078-detached-head

Understanding fast-forward merges

You can create a branch and switch to it at the same time:

- git checkout -b <new-branch-name>

compare this to

- git branch <new-branch-name>
- git checkout new-branch-name

fast-forward
main: 24ad18d
branch2 24ad18d - 968bdd5
regular
main: 24ad18d - 20d8458 - b6a090a
branch2 24ad18d - 968bdd5

curriculum item #079-fast-forward

Regular merging - main changed, branch changed

You can create a branch and switch to it at the same time:

- git checkout -b <new-branch-name>

compare this to

- git branch <new-branch-name>
- git checkout new-branch-name

fast-forward
main: 24ad18d
branch2 24ad18d - 968bdd5
regular
main: 24ad18d - 20d8458 - b6a090a
branch2 24ad18d - 968bdd5

curriculum item #080-regular-merge

Further exploring merging branches in git

Seeing merged & no-merged

- git branch --merged
  - shows which branches were merged into the current branch
  - ***if you no longer need a branch, delete it***
- git branch --no-merged
  - shows which branches are not merged into the current branch

curriculum item #081-seeing-merged-nomerged

Understanding merge conflicts

- scenario:
- on two branches
  - same file is changed
- solution:
  - look at this and resolve

- nano or some other text editor
- vs code - a more GUI interface

Merge conflicts occur when there are competing changes in the same lines of a file or when one person has edited a file and another person has deleted that same file. When this happens, Git can't automatically determine what is correct. Conflicts only affect the developer doing the merge; other developers are not affected.

Here's a step-by-step guide on how to resolve your merge conflict in the terminal:

1. Check the Status of Your Repository
First, check what files are in conflict with the `git status` command:
git status

2. Open the File in a Text Editor
Open the conflicting file (`testtest` in your case) with a text editor of your choice. You'll see conflict markers like these:

<<<<<<< HEAD
Your changes in 'main' branch will be here
=======
Changes in 'branchTWO' branch will be here
>>>>>>> branchTWO

The `HEAD` marker indicates changes that you've made (in your `main` branch), and the `branchTWO` marker indicates changes that are in the `branchTWO` branch.

3. Resolve the Conflict
Edit the file to resolve the conflict. This could mean choosing one side's changes over the other, merging the changes manually, or, possibly, making entirely new changes.

Here's an example:

Before:
```bash
<<<<<<< HEAD
# This is the main version
=======
# This is the branchTWO version
>>>>>>> branchTWO
```

After resolving:
```bash
# This is the resolved version
```

4. Save and Close the File
After you've resolved the conflict, save the changes in your text editor and close it.

5. Stage the Resolved File
To stage your changes, use:
git add .

6. Commit the Changes
Finally, you need to commit the changes to complete the merge:
git commit -m "Resolved merge conflict"

And that's it! You've successfully resolved a merge conflict.

curriculum item #082-understanding-merge-conflicts

Understanding working directory changes

The working directory is not tied to any particular branch until you commit your changes. When you create a new branch and switch to it, Git adjusts only the branch pointer and some metadata. The actual file system doesn't change, and the files in your working directory remain the same.

So if you have made changes to files in your working directory and then you create and switch to a new branch, those uncommitted changes will still be there. This happens because those **changes are not yet committed to any branch—they're still in the working directory, which is separate from the Git repository database where commits are stored.**

curriculum item #083-working-dir-changes

Creating a merge conflict

curriculum item #084-creating-merge-conflict

Resolving a merge conflict - in nano

curriculum item #085-merge-resolve-nano

Resolving a merge conflict - in VS Code

curriculum item #086-merge-resolve-vs

Aborting a merge conflict

- git merge --abort

The `git merge --abort` command is used to abort the merge process and return to the pre-merge state. This can be useful if you've initiated a merge but then decide that you don't want to proceed with it, or if there are too many conflicts and you'd rather take a different approach to integrating changes.

When you perform a `git merge`, Git tries to automatically merge the changes from the specified branch into your current branch. If there are no conflicts, the merge will be successful, and there's nothing to abort. But if there are conflicts that can't be automatically resolved, you are left in a conflicted merge state. Running `git merge --abort` at this point will

abort the merge operation and return your working directory to the state it was in before you initiated the merge.

Remember, you can only abort a merge before you've committed it. Once the merge is committed, it's a part of the commit history, and aborting would require a different set of operations, such as `git reset` or `git revert`.

Note that `git merge --abort` is equivalent to `git reset --merge` when a merge conflict situation is present.

curriculum item #087-merge-abort

Keeping a branch up-to-date with main

While you are on a branch other than main …

- git merge <main>

… allows the branch you are on to remain current with main
curriculum item #088-keep-branch-to-date-main

Understanding stash in git

Working with stash in git

stash files

- git stash
- git stash push
- git stash push --include-untracked

  otherwise only files with merge conflict will be tracked

See your stashed files

- git stash list
- git stash show <stash id>

Recover files from a stash

- git stash pop <stash id>

Remove a stash

- git stash drop <stash id>
- git stash clear

Create a Branch from a Stash:

You can create a new branch from a specific stash and apply the stash to it:

- git stash branch new-branch-name <stash id>

In git, **"stash"** allows you to temporarily save changes that you've made to your **working directory and staging area** without committing them. This can be useful when you need to

switch to a different branch or work on something else for a while.

Here's how stash works:

1. Stash Save: You use the **`git stash`** command to save your changes. This command will take the changes from both your working directory and the staging area and store them as a new "stash" entry. Stashes are given default names like "stash@{0}", "stash@{1}", and so on, based on their creation order.

2. Stash Pop or Apply: Once your changes are stashed, you can revert your working directory and index to their state at the last commit. You can then switch branches or perform other operations. Later, when you're ready to continue working on your stashed changes, you can use the **`git stash pop`** or **`git stash apply`** command to retrieve and reapply the stashed changes to your working directory and index. The difference between these commands is that `pop` removes the stashed changes from the stash list after applying, while `apply` leaves the stash intact.

3. Stash List: You can view the list of stashes using the **`git stash list`** command. This will display a list of all the stashed changes you have, along with their names and the time they were stashed.

4. Stash Drop and Clear: If you decide that you no longer need a specific stash, you can use the **`git stash drop stash@{n}`** command, where `n` is the index of the stash you want to remove. To clear all stashes, you can use the **`git stash clear`** command.

Here's how you can use `git stash

1. Stash Changes:
To stash your changes, including both staged and unstaged modifications:
git stash
This will create a new stash with a default message indicating the changes stashed.

2. Stash with a Message:
You can provide a custom message to describe the stash:
git stash save "Custom stash message" deprecated

3. Stash Specific Files:
If you only want to stash changes from specific files
git stash push path/to/file1 path/to/file2

4. View Stashes:
git stash list

5. Apply Stash:
To apply the most recent stash (pop the stash) and remove it from the stash list:
git stash pop
To apply a specific stash (without removing it from the stash list):
git stash apply stash@{n}
Replace `n` with the index of the stash you want to apply.

6. Drop Stash:
To remove a specific stash without applying it:

git stash drop stash@{n}

7. Clear All Stashes:
git stash clear

8. Create a Branch from a Stash:
You can create a new branch from a specific stash and apply the stash to it:
git stash branch new-branch-name stash@{n}

Remember, using `git stash` temporarily saves your changes, and you can apply them later or discard them if necessary. It's important to note that if you have conflicts when applying a stash, you'll need to resolve those conflicts manually.

Also, if you're planning to use `git stash` while you have untracked files, you might want to use the `--include-untracked` or `-u` flag to stash those files as well:
git stash -u

curriculum item #089-git-stash

Having one branch track another branch

Understanding tracking

- cat .git/config
  - when you clone a remote repo, tracking is set up automatically
- git branch -u <tracked-branch> <this-branch>
  - git pull
- git branch --unset-upstream <this-branch>
- git pull . main

In git, the concept of one branch **"tracking"** another branch refers to setting up a relationship between two branches, **typically a local branch and a remote branch.** This relationship allows git to **streamline synchronize changes between the two branches,** making it easier to collaborate with others and manage codebase updates.

When one branch tracks another, it means that **the local branch is configured to push and pull changes to and from the remote branch.** This is particularly useful when working with a team or collaborating on a project, as it ensures that your local branch stays up to date with changes made by other team members in the remote branch.

Here's how you can set up tracking between branches:

1. Creating a New Branch and Tracking:
When you create a new branch from an existing branch, you can set it to track the source branch. This means that your new branch will automatically push and pull changes to and from the source branch.
git checkout -b new-branch origin/source-branch

2. Setting Up Tracking for an Existing Branch:
If you're already on a local branch and want to set it up to track a remote branch, you can use the **`--set-upstream-to` or `-u`** option with the `git branch` command.

git branch --set-upstream-to=origin/source-branch current-branch
or
git branch -u origin/source-branch current-branch

After setting up tracking, you can use commands like **`git pull`** and **`git push`** without specifying the remote branch explicitly, as git will understand the relationship and perform the operations on the tracked branches.

For example:
git pull This command will pull changes from the remote branch that your local branch is tracking.
git push This command will push your local changes to the remote branch that your local branch is tracking.

Remember that it's important to keep your local branch up to date with the changes in the remote branch by regularly pulling changes from the remote. This helps avoid conflicts and ensures a smoother collaboration process.

Having one branch track another branch

In git, you can set up **one local branch to track another local branch** using the **`git branch`** command with the **`--set-upstream-to`** option.

Here's how you can make your local branch "secondb" track the "main" branch:

git branch --set-upstream-to=master secondb

- **After running this command, the"secondb" branch will track the "main" branch.**
  - This means that when you're on the "secondb" branch and you run
    - **git pull**
      - git will pull changes from the "main" branch.
    - **git push**
      - git will push changes to the "main" branch.

Keep in mind that in more recent versions of git (2.23 and later), **git will automatically set up tracking relationships**
- when you use `git branch` to create a new branch based on another branch.

git checkout -b branch2 main

This will create a new branch "branch2" based on "main" and automatically set up tracking between the two branches.

We can verify tracking is occurring here:
- cat .git/config

Having one branch track another branch - test
- have one local branch “branch2” track the local branch “main”

- add and commit a file to “main”
- switch to “branch2”
- a message like this should be shown:
    - Your branch is behind 'main' by 1 commit, and can be fast-forwarded.
    - (use "git pull" to update your local branch)
- git pull

Having one branch track another branch - unset

- unset
- cat .git/config
- test
    - add and commit a file to “main”
    - switch to “branch2”
    - is a message shown?
    - does ‘git pull’ work?

To unset the tracking relationship between a local branch and another local branch in git, you can use the `--unset-upstream` option with the `git branch` command. Here's how you can do it:

git branch --unset-upstream branch2

Running either of these commands will remove the tracking relationship between the "branch2" branch and the "main" branch. After unsetting the upstream relationship, the "branch2" branch will no longer track the "main" branch, and you won't be able to use `git pull` or `git push` without specifying the remote and branch explicitly.

Remember that unsetting the upstream tracking relationship doesn't delete the branch itself or its content; it only removes the association that allows git to automatically determine the remote branch for pushing and pulling.

Using git pull from one local branch to another - pull vs merge

- git pull . main

git pull

- git pull = fetch and merge
- fetches changes
- merges changes
- **most commonly used with remote branches**

- **`git fetch . main`** fetches changes from the "main" branch in the current repository (indicated by `.`).
- `git merge FETCH_HEAD` merges the fetched changes into the current branch.
- In essence, this command is equivalent to running `git fetch . main` followed by `git merge FETCH_HEAD`.

git merge

- merges changes from one branch into another
- **most commonly used with local branches**

curriculum item #090-branch-tracking

Working with a remote repository

Pushing another branch to a remote repository

- push
    - git push origin <branch name>
    - git push -u origin <branch name>
        - for example: banch name = secondbanch

curriculum item #091-push-branch-to-remote

Checking out another branch on a remote repository

- getting a remote branch to work with
    - git checkout -b secondbranch origin/secondbranch
      or
    - git branch secondbranch origin/secondbranch
    - let's make some changes, add them, commit them
    - git push
        - git push origin secondbranch

curriculum item #092-checkout-remote-branch

Delete a remote branch

- delete a branch from local
    - git branch -D secondbranch
- delete a branch from remote
    - git push origin --delete secondbranch
- prune if needed
    - git fetch -p origin

To remove the `california` branch from a local repository:

1. Fetch the latest updates from the remote repo and prune
   git fetch -p origin

   The `-p` option stands for "prune." This command will remove any remote-tracking references that no longer exist on the remote (like `origin/california` in this case).

2 Check if the `california` branch is active locally
   Before deleting the local `california` branch, switch to another branch (e.g., `main`). If already on the `california` branch, switch using:
   git checkout main

3. Delete the local `california` branch
   git branch -d california

   The `-d` option will delete the branch safely (i.e., it prevents deletion if the branch contains changes that are not yet merged). If your friend is sure they want to delete it even with unmerged changes, they can use `-D` instead.

4. Verify that the branch is deleted
   git branch -a

   This command lists all the local branches. The `california` branch should no longer be listed.

curriculum item #093-delete-remote-branch

Forking a repo and pull requests

- (1) fork
- (2) clone your fork
- (3) create a branch
    - work on that branch, make changes, add, commit
- (4) push your branch with changes
    - git push origin <your-branch-name>
        - git push -u origin <your-branch-name>
- (5) open a pull request

curriculum item #094-pull-request

Congratulations!

Great job!

You've done it! After dedicating your time, energy, and intellectual curiosity to this course, you've successfully crossed the finish line. I am so proud of you!

🌟 From understanding the intricate details of git's internal structures to mastering advanced version control techniques, you've expanded your skill set in ways that will make a tangible impact in your coding projects.

🛠️ You've conquered commands, grappled with git branches, and navigated the labyrinthine paths of commit history. You've learned not just the 'how' but also the 'why'—gaining insights into what makes git an incredibly powerful tool for collaborative development.

🔗 Now, you're part of an elite group of developers who can optimize workflows, troubleshoot with ease, and collaborate efficiently in team environments. Your new skills are not just a badge of honor, but a toolkit of possibilities.

💡 As you move forward, don't forget that learning is a lifelong journey. The tech world is ever-evolving, and there's always more to discover. Keep exploring, keep asking questions, and most importantly, keep coding!
curriculum item #095-great-job

Bonus lecture

curriculum item #096-bonus-lecture