Github basic codes

**Installing Git**

**Windows Install**

Download Git For Windows Find the downloaded .exe file and open it to execute Git Bash.

**Configuring Git**

* git config --global user.name "hadush16"
* git config --global user.name "hadush12brhane@gmail.com"

Committing The most important Git feature!

* git status
* git init
* git add file1 file2
* Git commit -m “ “

DO NOT INIT A REPO INSIDE OF A REPO! Before running git init, use git status to verify that you are not currently inside of a repo.

Atomic Commits When possible, a commit should encompass a single feature, change, or fix. In other words, try to keep each commit focused on a single thing. This makes it much easier to undo or rollback changes later on. It also makes your code or project easier to review.

Ignoring Files

Create a file called .gitignore in the root of a repository. Inside the file, we can write patterns to tell Git which files & folders to ignore: .DS\_Store will ignore files named .DS\_Store folderName/ will ignore an entire directory \*.log will ignore any files with the .log extension

Git Branching

Branches are an essential part of Git! Think of branches as alternative timelines for a project. They enable us to create separate contexts where we can try new things, or even work on multiple ideas in parallel. If we make changes on one branch, they do not impact the other branches (unless we merge the changes)

Master? Main? In 2020, Github renamed the default branch from master to main. The default Git branch name is still master, though the Git team is exploring a potential change. We will circle back to this shortly.

* git branch view of => branches u are currently on
* git branch <branch name> creating of branches
* git switch < branch name>
* git checkout < branch name>
* git switch -c < branch name>
* git branch -v => v flag with git branch to view more information about each branch

Merging

To merge, follow these basic steps:

1. Switch to or checkout the branch you want to merge the changes into (the receiving branch)
2. Use the git merge command to merge changes from a specific branch into the current branch.
   1. git switch master
   2. git merge bugfix

Resolving Conflicts

Whenever you encounter merge conflicts, follow these steps to resolve them:

1. Open up the file(s) with merge conflicts
2. Edit the file(s) to remove the conflicts. Decide which branch's content you want to keep in each conflict. Or keep the content from both.
3. Remove the conflict "markers" in the document
4. Add your changes and then make a commit!
   1. git branch -d bugfix deleting => the branch

Git Diff

We can use the git diff command to view changes between commits, branches, files, our working directory, and more! We often use git diff alongside commands like git status and git log, to get a better picture of a repository and how it has changed over time.

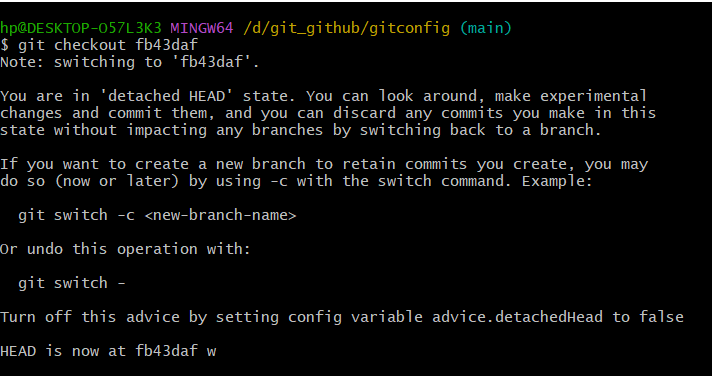
* git diff --staged
* git diff --cached
* git diff branch1..branch2 =>Comparing Branches
* git diff commit1..commit2 => Comparing Commits

Stashing

Git provides an easy way of stashing these uncommitted changes so that we can return to them later, without having to make unnecessary commits.

* git stash=> Now that I have stashed my changes, I can switch branches, create new commits,
* git stash pop
* git stash apply to apply whatever is stashed away,
* git stash -u=> to tell git stash to include those untracked files.
* git stash apply stash@{2}
* git stash drop stash@{2}
* git stash clear

**Undoing Stuff & Time Traveling**

****

* git switch -c newbranch

While in detached HEAD, make a new branch and switch to it. Head is now back to pointing at a branch reference!

* git checkout HEAD~1

HEAD~1 refers to the commit before HEAD (parent) HEAD~2 refers to 2 commits before HEAD (grandparent)

### **What’s happening**

* You checked out a **specific commit** (036cc3b) instead of a branch.
* That’s why Git says:  
    
    
   "HEAD detached at 036cc3b"

In this state, **new commits won’t be linked to any branch** — they’ll just “float” until you create a branch.

#### **Option 1: Go back to your main branch**

If you just want to return to your main branch:

git switch main

or (if your main branch is called master)

git switch master

Then check:

git status

Now your HEAD should no longer be detached.

#### 

#### **Option 2: Keep your current changes (create a new branch from this commit)**

If you want to **continue working from commit 036cc3b**:

git switch -c new-branch-name

Example:

git switch -c fix-detached-head

Now your new branch fix-detached-head starts from that commit, and you can commit safely.

#### **Option 3: Undo and go back to the previous state**

If you just want to go back to where you were before:

git switch -

### **Discarding Changes**

* git checkout HEAD <file>
* git checkout -- <file>

**Restore**

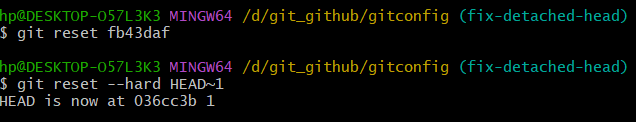
To restore the file to the contents in the HEAD, use git restore

* git restore <file-name>
* git restore --source HEAD~1 index.html
* git restore --staged <file-name>

**Git Reset**

git reset will reset the repo back to a specific commit. The commits are gone

* git reset <commit-hash>



**git revert**

If you want to reverse some commits that other people already have on their machines, you should use revert. If you want to reverse commits that you haven't shared with others, use reset and no one will ever know!

* git revert <commit-hash>

| Git | Github |
| --- | --- |
| Git is the version control software that runs locally on your machine. You don't need to register for an account. You don't need the internet to use it. You can use Git without ever touching Github. | Github is a service that hosts Git repositories in the cloud and makes it easier to collaborate with other people. You do need to sign up for an account to use Github. It's an online place to share work that is done using Git. |

**Collaboration**

If you ever plan on working on a project with at least one other person, Github will make your life easier!

**Cloning**

* git clone<url>

**Permissions?** Anyone can clone a repository from Github, provided the repo is public.

Register to Github

SSH Keys

1. Open terminal
2. Enter ls -al ~/.ssh

See #lists the file

### **1. Open Git Bash or PowerShell**

You can use **Git Bash** (installed with Git) or **Windows PowerShell**.

### **🔑 2. Generate a new SSH key**

Run this command (replace your email address):

ssh-keygen -t ed25519 -C "your\_email@example.com"

If your system doesn’t support ed25519, use:

ssh-keygen -t rsa -b 4096 -C "your\_email@example.com"

### **💾 3. Save the key**

When prompted:

Enter a file in which to save the key (/c/Users/YourName/.ssh/id\_ed25519):

Just press **Enter** to accept the default location.

Then you’ll be asked for a passphrase (optional but recommended).  
 If you don’t want one, press **Enter** twice.

### **🔍 4. Start the SSH agent**

eval "$(ssh-agent -s)"

### **➕ 5. Add your private key to the SSH agent**

ssh-add ~/.ssh/id\_ed25519

### **📋 6. Copy your public key**

To copy it to your clipboard:

clip < ~/.ssh/id\_ed25519.pub

Now your SSH key is copied!

### **🌐 7. Add the SSH key to your GitHub account**

1. Go to **GitHub → Settings → SSH and GPG keys**
2. Click **New SSH key**
3. Paste the key into the field
4. Give it a title (e.g., “My Windows Laptop”)
5. Click **Add SSH key**

### **✅ 8. Test the connection**

Run:

ssh -T git@github.com

If you see something like this:

Hi username! You've successfully authenticated, but GitHub does not provide shell access.

🎉 You’re done!

**Remote**

### **…or create a new repository on the command line**

1. echo "# marketing" >> README.md
2. git init
3. git add README.md
4. git commit -m "first commit"
5. git branch -M main
6. git remote add origin https://github.com/hadush16/marketing.git
7. git push -u origin main

### **…or push an existing repository from the command line**

1. git remote add origin https://github.com/hadush16/marketing.git
2. git branch -M main
3. git push -u origin main

**Delete the local branch**

If you just want to delete the branch on your computer:

git branch -d dogs

If Git says the branch is not fully merged and refuses, force it:

git branch -D dogs

## 

## **☁️ Delete the remote branch (on GitHub)**

If you want to delete it **from GitHub**, use:

git push origin --delete dogs

✅ That removes the dogs branch from your GitHub repository.

## **🔍 To confirm it’s gone:**

Check local branches:  
  
 git branch

Check remote branches:  
  
 git fetch -p

* git branch -r

## **To remove the submodule properly**

### **🪜 Step-by-step:**

1️⃣ **Deinitialize the submodule**

git submodule deinit -f flutter-complete-guide-course-resources

2️⃣ **Remove the submodule entry from the index**

git rm -f flutter-complete-guide-course-resources

3️⃣ **Delete the submodule’s files**

rm -rf .git/modules/flutter-complete-guide-course-resources

4️⃣ **Commit the changes**

git commit -m "Removed submodule flutter-complete-guide-course-resources"

5️⃣ **Push to GitHub**

git push origin main

🔁 Replace main with your actual branch name if different.

✅ Done

Git Fetch

1. git fetch <remote>
2. git fetch <remote> <branch> Ex: git fetch origin master
3. git checkout <origin/branch> or --track origin/puppies

Git Pulling

git pull = git fetch + git merge

**git fetch**

update the remote tracking branch with the latest changes from the remote repository

**git merge**

update my current branch with whatever changes are on the remote tracking branch

1. git pull <remote> <branch> Ex: git pull origin master

An even easier syntax! If we run ***git pull*** without specifying a particular remote or branch to pull from, git assumes the following: remote will default to origin branch will default to whatever tracking connection is configured for your current branch.

### GitHub – Summary

#### 🏛️ Public vs. Private Repositories

* **Public Repos: Visible to everyone on the internet; anyone can view them.**
* **Private Repos: Only accessible to the owner and invited collaborators.**

#### **🗑️ Deleting GitHub Repositories**

* **Only the repo owner (or those with the right permissions) can delete a repository.**

#### **👥 Adding Collaborators**

* **You can invite others to work on your private repositories.**
* **Collaboration allows multiple users to push, pull, and review code together.**

#### **📘 README Files**

* **Used to describe important details about a project:**
  + **What it does**
  + **How to use it**
  + **Why it’s important**
  + **Who maintains it**
* **Filename: README.md (written in Markdown)**
* **GitHub automatically displays it on the repo’s homepage.**

#### **✍️ Markdown Basics**

* **Simple syntax for formatting text in README files.**
* **Supports:**
  + **Headings (#, ##, ###, etc.)**
  + **Bold/italic text**
  + **Lists and links**
  + **Code blocks**
  + **Images**

#### **📄 GitHub Gists**

* **A lightweight way to share code snippets or small notes.**
* **Easier to create than full repositories, but with fewer features.**

#### **🌐 GitHub Pages**

* **Lets you host static websites (HTML, CSS, JS) directly from a GitHub repository.**
* **No server-side code (like Python or Node) is supported.**

**Types of GitHub Pages:**

1. **User Site: One per GitHub account (e.g., username.github.io).**
2. **Project Site: One per repository (e.g., username.github.io/repo-name).**

### **✅ In Short**

**GitHub is not just for storing code — it’s also a collaboration, documentation, and publishing platform.  
 It allows version control, teamwork, public or private sharing, and even simple website hosting.**

**git workflows (for collaboration)**

**Centralized Workflow**

1. Hadush clones the repo Ex master
2. Zereu clones the same repo Ex master
3. Hadudsh gets to work on a new feature! Adding and Committing all day long!
4. Hadush now pushes up his new work to Github
5. Zereu has also been hard at work on her own new feature.
6. He tries to push her new work up to Github, but she runs into trouble! As he is behind the master branch

* Git push origin master

**Fix**

* So he pulls to get the changes from origin master Forrest's work must be merged in. Hopefully this goes relatively smoothly!
* Git pull origin master
* Git commit -m “ merged “
* Now that she has merged the latest work from Github, she can push her master branch up!
* Git push origin master

**The Problem**

**Frequent merge conflicts:** Everyone edits the same branch, so conflicts happen often.

**Unstable main branch:** If someone pushes buggy or incomplete code, it affects everyone.

**No room for experimentation:** Developers can’t safely test new features without impacting others.

**Difficult collaboration:** Working together on the same branch leads to broken code and wasted time.

**Enter Feature Branches**

Rather than working directly on master/main, all new development should be done on separate branches!.

1. Hadush clones the repo Ex master
2. Zereu clones the same repo Ex master

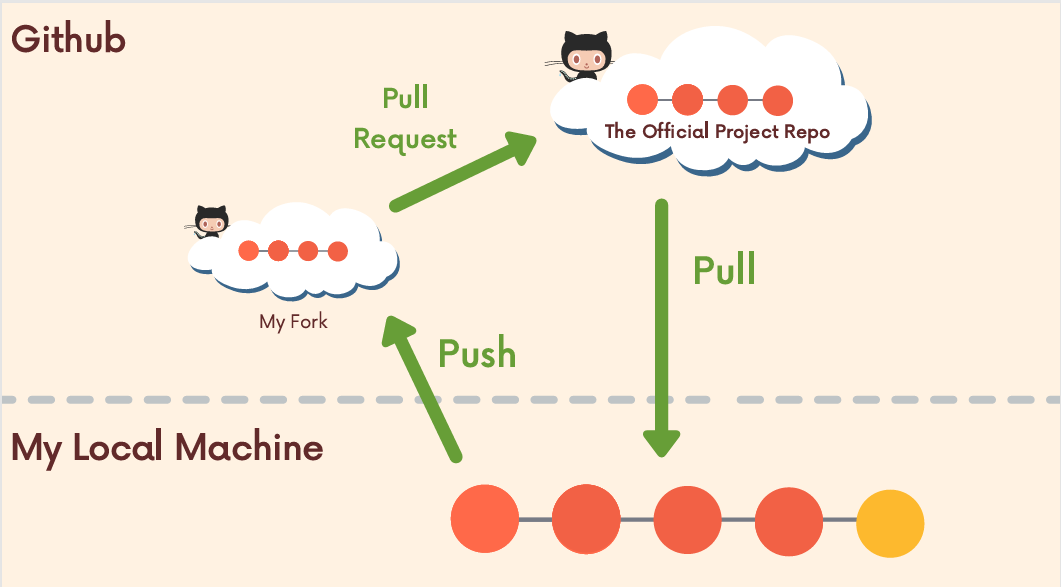
* Hadush starts work on a new feature. He does all this work on a separate branch!
* Hadush wants Zereu to take a look at his new feature. Instead of merging it into master, he just pushes his feature branch up to Github!
* Git status
* Git add index.html
* Git commit -m “ “
* Git remote -v
* Git push origin <H feature branch>
* Zeru hears from Hadush that he wants her to take a look at his new work. She pulls down his feature branch from Github.
* Git pull origin <main/master >
* Git git switch -c <Z feature branch>
* Git status
* Git add index.htm
* Git commit -m “ “
* Git push origin Z feature branch
* Git fetch origin master
* Git branch -r
* Git checkout origin/H feature branch
* Git switch <H feature branch>
* Git add index.html
* Git commit -m “edited H feature branch”
* Git push origin <H feature branch>
* Git switch origin <Z feature branch>
  + Git pull origin <h feature branch>
  + Git log
  + Git add …
  + Git commit -m “jj”
  + Git push origin <H feature branch>
* Zereu takes a look at the code and makes a couple improvements of his own. He adds/commits on the H feature branch and push to the H feature branch
* Hadush fetches from Github and sees that there is new work on the <H feature branch> branch. He pulls the changes down and continues work.
  + Git pull origin <H feature branch>
* Hadush decides he is happy with the new feature, so he merges it into master! Think as project owner
  + Git status
  + Git switch origin main
  + Git pull origin main
  + Git merge <h feature branch>
  + Git status
  + Git push origin main
* Zereu takes a look at the code and makes a couple improvements of his own
  + Git status
  + Git switch main
  + Git pull origin main

**Merging In Feature Branches**

* git fetch origin <main>
* git switch my-new-feature
* git merge master
* fix conflicts!
* git switch master
* git merge my-new-feature
* git push origin main

**Forking Github**

(and similar tools) allow us to create personal copies of other peoples' repositories. We call those copies a "fork" of the original. When we fork a repo, we're basically asking Github "Make me my own copy of this repo please" As with pull requests, forking is not a Git feature. The ability to fork is implemented by Github.



* Git status
* Git clone <url> for the my fork
* Git remote add upstream <url of original> for update
* Git remote -v to see the list of two
* Git pull upstream main to get updates from the original to my forked

**Rebasing vs merging**

There are two main ways to use the git rebase command:-

1. as an alternative to merging

* Git merge master
* Git rebase master
* Git rebase --continue

1. as a cleanup too

**Git Tags**

**Viewing Tags** git tag will print a list of all the tags in the current repository.

* git tag
* git tag -l "\*beta\*"
* git checkout Checking Out Tags
  + git tag <tagname> to create a lightweight tag,
  + ❯ git tag -a <tagname> we can also use the -m option to pass a message directly

Tagging Previous Commits So far we've seen how to tag the commit that HEAD references. We can also tag an older commit by providing the commit hash:

* git tag -a <tagname> <commit-hash>
* git tag -d <tagname> to delete
* git tag -f <tagname> to force
* git push --tags to push tags

**HASHING FUNCTIONS**

* **echo 'hello' | git hash-object --stdin -w**
* **git cat-file -p <object hash>**
* **$Ex. git cat-file -p 5ff7e655a8ecb969bcfb912f0853d14e122d56fd**
* **git cat-file -p a224f Output =>Wellcome**

**Git Reflog**

The git reflog command accepts subcommands show, expire, delete, and exists. Show is the only commonly used variant, and it is the default subcommand. git reflog show will show the log of a specific reference (it defaults to HEAD). For example, to view the logs for the tip of the main branch we could run git reflog show main.

* git reflog show HEAD

**Timed References**

* git reflog master@{one.week.ago}
* git checkout bugfix@{2.days.ago}
* git diff main@{0} main@{yesterday}

**=> Code . is the code use to open the vs code**

**Git reset - - hard <hash>**

**Reflogs Rescue**

**W**e can sometimes use reflog entries to access commits that seem lost and are not appearing in git log.

**Git Aliases**

Global Git Config Git looks for the global config file at either ~/.gitconfig or ~/.config/git/config. Any configuration variables that we change in the file will be applied across all Git repos. We can also alter configuration variables from the command line if preferred

**Adding Aliases** We can easily set up Git aliases to make our Git experience a bit simpler and faster. For example, we could define an alias "git ci" instead of having to type "git commit" Or, we could define a custom git lg command that prints out a custom formatted commit log