How to Use Git

A Guide for Dr. BC's CS 3383 Course

Presented by the Team Lead 1's of CS 3383:
Clive Miller, Asim Sapkota,
Johnathan Van Vliet, and Zuzanna Whitman

Written by Johnathan Van Vliet

Department of Computer Science
University of Idaho
February 2026

Contents

0.1 Introduction . . . 2
0.1.1 What is a Distributed Version Control System? . . . 2
0.1.2 What is GitHub? . . . 2
0.1.3 GitHub Desktop and GitHub CLI . . . 3
0.2 Setting up Git . . . 3
0.2.1 Installing Git . . . 3
0.2.2 Initial Setup . . . 5
0.2.3 Cloning and Authentication . . . 5
0.2.4 Initializing a Repository . . . 7
0.2.5 Initial Files . . . 8
0.2.6 Git LFS . . . 8
0.3 Git Basics . . . 9
0.3.1 Staging Changes . . . 10
0.3.2 Commits . . . 10
0.3.3 Pushing and Pulling Changes . . . 12
0.3.4 Creating a Branch . . . 12
0.3.5 Merging Branches . . . 13
0.3.6 Resolving Merge Conflicts . . . 14
0.4 Basic Workflow . . . 14
0.5 Disaster Recovery . . . 16
0.5.1 Reflog and Reset . . . 16
0.5.2 Abort . . . 17
0.5.3 Restoration of Deleted Files . . . 17
0.5.4 Undoing Commits . . . 18
0.5.5 Re-Cloning the Repository . . . 18
0.6 Tips and Tricks . . . 18
0.7 GitHub . . . 19
0.7.1 Issues . . . 19
0.7.2 Pull Requests . . . 19
0.7.3 Releases . . . 19
0.8 Glossary . . . 20
0.8.1 Terminology . . . 20
0.8.2 Commands . . . 20
0.9 References . . . 22

0.1 Introduction

Git is a free and open source distributed version control system (DVCS) designed with efficiency in mind. It was created by Linus Torvalds in 2005 due to the needs of the Linux kernel development community.

Each Git project is known as a repository, or a repo, and stores the projects' files and complete change history.

Today, it remains a crucial part of the software development industry. According to a StackOverflow survey from 2022, almost 97% of all professional developers use Git for their DVCS. For this reason, it is critical that all Computer Science students learn how to use it.

0.1.1 What is a Distributed Version Control System?

A Version Control System (VCS) is an application that tracks changes within a team's code and stores a complete history of all changes made. This allows for easier experimentation and collaboration. It acts as a way of protecting a team's source code from irreparable damage.

There are two most common types of Version Control Systems: Centralized and Distributed. A Centralized Version Control System (CVCS) has all files and repository history stored on a centralized server. A Distributed Version Control System (DVCS) has all files stored on all systems with access to the repository.

One Centralized VCS is called Subversion, or SVN. If you're curious about it, you can read more at their website: `https://subversion.apache.org/`

Distributed Version Control is like the democracy of Version Control Systems. Everyone has a complete history of the project on their side and can alter it as they see fit. It's a major reason why Git and other DVCS applications are so popular: they can be run by anybody on any computer without substantial cost.

0.1.2 What is GitHub?

GitHub is a website owned by Microsoft that was designed to work as a central hub for Git repositories. This adds a somewhat centralized element to the Distributed Version Control System that Git is. It helps streamline the collaboration process and provides a secure repository storage solution.

It is not, however, affiliated with Git in any way. Git is the Version Control System, and GitHub is the centralized hub for the Version Control System.

In fact, Git can be paired up with other web applications too, such as GitLab. GitLab offers its own features and, although less popular than GitHub, is a perfectly viable alternative to GitHub.

0.1.3 GitHub Desktop and GitHub CLI

GitHub Desktop is a free and open source application developed by the GitHub team as a GUI replacement for Git. It is often considered a less powerful, albiet more beginner-friendly version of Git. However, it is not industry standard and, for that reason, will not be recommended for this class.

GitHub CLI is a Git replacement CLI tool developed by the GitHub team as a way to bring GitHub into the terminal. It is designed as a way to pull GitHub features (like issues and pull requests) into the command line. It's not widely used, and although it pulls in many of the features of Git, it is not recommended due to it not being industry standard.

Clive has personal experience being asked about Git commands during an interview.

0.2 Setting up Git

This section will focus on the basic installation and configuraiton of Git, as well as creating and working with your first repository

0.2.1 Installing Git

Git is available on Windows, MacOS, and practically every version of Linux. The following sections list recommended setup instructions for the three major operating systems:

For **Windows**,

1. Download and run the Git installer from `https://git-scm.com/install`
2. Click **Next** to accept the GNU General Public License
3. Select an installation location or leave the default, then click **Next**
4. Leave the defaults in the component selection screen and click **Next**
5. Choose whether or not to create a Start Menu folder and click **Next**
6. Select your preferred text editor and click **Next**
7. Select *Override the default branch name for new repositories* and type `main` in the box, then click **Next**
8. Select *Git from the command line and also from 3rd-party software* and click **Next**
9. Select *Use bundled OpenSSH* and click **Next**
10. Select *Use the OpenSSL library* and click **Next**
11. Click *Checkout Windows-style, commit Unix-style line endings* and click **Next**
12. Choose your preferred console window (MinTTY is recommended) and click **Next**
13. Click *Fast-forward or merge* and click **Next**

14. Click *Git Credential Manager* and click **Next**
15. Check *Enable file system caching* and click **Next**
16. Leave both experimental options unchecked and click **Install**
17. Click **Finish**

For **MacOS**,

1. Install Homebrew if you don't already have it by opening a terminal and running `$ /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com /Homebrew/install/HEAD/install.sh)"`
2. Run `brew install git`

For **Linux** run the corresponding terminal command:

- **Debian/Ubuntu**: `apt-get install git`
- **Fedora 22 and newer**: `yum install git`
- **Fedora 21 and older**: `dnf install git`
- **Gentoo**: `emerge --ask --verbose dev-vcs/git`
- **Arch Linux**: `pacman -S git`
- **openSUSE**: `zypper install git`
- **Mageia**: `urpmi git`
- **Nix/NixOS**: `nix-env -i git`
- **FreeBSD**: `pkg install git`
- **Solaris 9/10/11**: `pkgutil -i git`
- **Solaris 11 Express, OpenIndiana**: `pkg install developer/versioning/git`
- **OpenBSD**: `pkg_add git`
- **Alpine**: `apk add git`
- **Slitaz**: `tazpkg get-install git`

Git can also be built from source. This is typically not recommended and is typically only needed if you are running a Linux distribution without a package manager or without a prebuilt Git package. Instructions can be found in the `README` file found at `https://github.com/git/git`

0.2.2 Initial Setup

For Git to fully integrate with GitHub, we must perform some initial setup.

First, set your username and email with the `git config` command...

```
git config --global user.name = "JoeVandal20"
git config --global user.email = "jvandal@uidaho.edu
```

...where `user.name` is the username you use for GitHub and `user.email` is the email you use for GitHub.

Additional Configuration for HTTPS Authentication

If you are authenticating to Git with HTTPS, also set Git to use the Git Credential Helper:

```
git config --global credential.helper cache
```

You can also set a custom cache timeout, as the default is 15 minutes before needing to reauthenticate. Do this by providing a timeout value in seconds:

```
git config --global credential.helper 'cache --timeout=3600'
```

For more information on authentication, read the next section

0.2.3 Cloning and Authentication

There are two main methods you can use to clone into a Git repository stored on GitHub: HTTPS or SSH. There are benefits and drawbacks to each method, as well as varying levels of complexity.

There is also Git Credential Manager, which will not be covered due to uncertainty in its development. Information on GCM can be found at its GitHub repository:
`https://github.com/git-ecosystem/git-credential-manager`

HTTPS Authentication

Using Git with HTTPS involves authenticating with your username and password each time you push/pull. It is simple to set up and requires no additional software, but is **extremely insecure**. Git stores your password as plaintext, meaning that anybody can see it at any time so long as they can access your `.gitconfig` file. It is possible to use a Personal Access Token generated by GitHub as a substitute, but this still allows anybody with access to your system access to your main repository. It is usually a better idea to use SSH authentication if possible.

However, if this is not possible for you, here are the instructions for use:

1. Generate a Personal Access Token by first going to `https://github.com/settings/tokens`. For the purpose of this course, select *Generate new token (classic)* as the newer token type does not work for repositories you do not own. Give it a name, set

the expiration, and give it at least the *repo* scope. You may give it a greater scope if you wish.

2. Once created, copy your personal access token. It should look something like: `ghp_ABCD123XXXXXXXXXXXXXXXXXXXXXXXX`
3. Clone your repository using the `git clone` command: `git clone https://github.com/username/repository-name.git`
4. When Git asks for a password, use your Personal Access Token

It is recommended to use a password manager to store and manage your Personal Access Token (However, 1Password has a custom SSH client that works wonderfully with GitHub).

SSH Authentication

Using Git with SSH involves authenticating using public key cryptography. It's more complicated to set up, but is extremely reliable and incredibly secure. It is also widely considered the industry standard.

Configuring SSH for Git is slightly different on Windows as it is for MacOS and Linux. Make sure to follow the proper set.

To generate your SSH keypair: For **Linux and MacOS** use Bash or Zsh,

1. Generate a SSH keypair using: `ssh-keygen -t ed25519 -C "your_email@example.com"` Press Enter to accept the default file location (or set a custom one if you wish) and then enter a secure passphrase for added security if you'd like.
2. Start the SSH agent with `eval "$(ssh-agent -s)"`
3. Add your SSH key to the SSH agent with `ssh-add PATH_TO_KEY/id_ed25519` By default, `PATH_TO_KEY` is `~/.ssh/`, so you would run the following command: `ssh-add ~/.ssh/id_ed25519`
4. Copy your public key to your clipboard. This works best by running `cat PATH_TO_KEY/id_ed25519.pub` and copying the output.

For **Windows** use PowerShell,

1. Generate a SSH keypair using: `ssh-keygen -t ed25519 -C "your_email@example.com"` Press Enter to accept the default file location (or set a custom one if you wish) and then enter a secure passphrase for added security if you'd like.
2. Run PowerShell as an administrator and run the following commands to start the SSH agent and configure it to start automatically:

```
Get-Service -Name ssh-agent | Set-Service -StartupType Automatic
Start-Service ssh-agent
```

3. Add your SSH key to the SSH agent with `ssh-add PATH_TO_KEY\id_ed25519`
   By default, `PATH_TO_KEY` is `C:\Users\YourUsername\.ssh\id_ed25519`, so you would run the following command:
   `ssh-add C:\Users\YourUsername\.ssh\id_ed25519`
4. Copy your public key (the entire contents of the id_ed25519.pub file)to your clipboard. This works best by running

```
Get-Content PATH_TO_KEY/id_ed25519.pub | Set-Clipboard
```

To add the key to GitHub,

1. Log into GitHub
2. Click your profile photo in the upper-right corner and select **Settings**
3. On the left, select **SSH and GPG keys**
4. Click the **New SSH key** button
5. Give it a name
6. Set key type of *Authentication Key*
7. Paste your key into the text box
8. Click **Add SSH key**

To test that the SSH keypair was set up successfully, run `ssh -T git@github.com`

Cloning

Once you are set up with either HTTPS or SSH authentication you can clone your project using one of the following commands:

- HTTPS: `git clone https://github.com/Username/RepoName.git`
- SSH: `git clone git@github.com:Username/RepoName.git`

0.2.4 Initializing a Repository

Before initializing your repository, it's recommended to set the default branch name (if you did not do so during installation) to "main", which is the current industry default. To do this, run `git config --global init.defaultBranch main`

To initialize a repository, open your terminal and navigate to your project directory with `cd`. Then, initialize the repository using `git init`. You can check the status of the repository with `git status`. Then, run `git add .` to stage all files for an initial commit and `git commit -m "Initial commit"` to commit the initial changes. These commands will be discussed in more detail in a later section.

Finally, add the remote destination with:

- HTTPS: `git remote add origin https://github.com/Username/RepoName.git`
- SSH: `git remote add origin git@github.com:Username/RepoName.git`

Some find it easier to create a repository on GitHub and then clone it using `git clone`.

0.2.5 Initial Files

There are three initial files all public repositories should have:

README.md

A `README.md` file contains information about your project, such as what it is, contributors, its purpose, compatibility, and whatever else may be relevant to a user. It is typically formatted in Markdown, denoted by the `.md` file extension. Markdown formatting follows an easy to understand but powerful syntax. Information on Markdown syntax can be found at `https://www.markdownguide.org/basic-syntax/`

.gitignore

A `.gitignore` file specifies a list of files, file types, and directories that Git should ignore. For example, adding `foo.txt` to a repo's `.gitignore` would prevent any files called `foo.txt` from being tracked by Git. Documentation on the intricacies of the `.gitignore` file can be found at Git's website: `https://git-scm.com/docs/gitignore`

It is generally good practice to add the following lines to your `.gitignore` to prevent personal config files from being tracked:

```
*.dSYM
*.DS_Store
*.vsconfig
.vscode/
```

You can find a template `.gitignore` specifically for Unity projects here:
`https://github.com/github/gitignore/blob/main/Unity.gitignore`

LICENSE

The `LICENSE` file contains the copyright license the project is published with. A copyright license is important because it allows you to decide what you want with your code.

The developers of GitHub created `https://choosealicense.com` to assist project leaders in choosing a license.

0.2.6 Git LFS

Git, by nature, is not designed to work with extremely large files. Because of this, some projects with many large files (such as a video game with sprites and audio) are slow or even completely incompatible with base Git. To mitigate this, Git LFS was created. It works

by storing the main file on an external server (such as GitHub) and storing a pointer to it inside Git.

On Windows, Git LFS should already be downloaded. To initialize it, run `git lfs install`. This only has to be done once per user account.

On MacOS, use Homebrew to install Git LFS with `brew install git-lfs` and then run `git lfs install`

On Linux, go to `https://git-lfs.com/` and install using the provided PackageCloud link. Then run `git lfs install`

To get Git LFS to properly track files, create a `.gitattributes` file. A Unity-specific `.gitattributes` file template can be found here:
`https://github.com/gitattributes/gitattributes/`

To add a filetype to the .gitattributes, the file can be edited directly or you can run `git lfs track "*.FILETYPE"`

0.3 Git Basics

To call Git from the command line, you start with the `git` command, followed by what you want to do.

An essential command to learn is `git status`. The output from this command shows the current status of the Git working tree: **Example:**

```
$git status
On branch main
Nothing to commit, working tree clean
```

Example:

```
$touch foo.c
$git status
On branch main
Your branch is up to date with 'origin/main'

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        foo.c
```

The following sections will showcase some of the most important Git commands. For more information on a specific command, you can find Git's documentation at `https://git-scm.com/docs`

0.3.1 Staging Changes

To add changes to Git, you must first stage them. This is done with the `git add` command. For example, if you made changes to a file called `foo.py`, you would stage it using `git add foo.py`. You can then check the status of the repo with `git status`

Some helpful tips and things to know for staging:

- Git handles pathnames relative to the parent directory of your repository.
- Use `git restore --staged` to remove a file from staging.
- Use `git add .` or `git add --all` to add all files to staging.

0.3.2 Commits

Once files are properly staged, the changes must be confirmed by Git. This is known as a commit, or committing your changes. This can be done with `git commit`.

It is generally best practice to provide a descriptive message alongside your commit, and you will be hard pressed to get Git or GitHub to accept a commit without a message. There are two ways to do this

Using the -m flag

The `-m` flag (`git commit -m`) is easy, versatile, and best for writing short commit messages. For example: `git commit -m "Fix the HTTPS error"` There are many instances in which you will want to be more descriptive. Using a second `-m` flag will essentially create a body/description for your commit:

```
git commit -m "Fix the HTTPS error" -m "Users were unable to establish a
secure connection with various endpoints. This was caused by an expired
certificate and an accidentally hardcoded HTTP request. The fix removes
that hardcoded HTTP request and replaces the expired certificate."
```

However, this is a bit difficult to type out on a terminal. So, many people instead opt to use a Git Editor.

Using the Git Editor

The Git Editor can be used to set the editor that Git will automatically open when a `git commit` is run *without* a message. By default, it is typically Vi, Vim, or Nano. This can be overwritten with a `git config`:

`git config --globl core.editor <PATH-TO-EDITOR> <FLAGS>`

- `git config`: The command
- `--global`: Makes the change work for all repositories
- `core.editor`: The value being changed

- `<PATH-TO-EDITOR>`: Needs to be set to either the full filepath or, if the application has been added to the system PATH variable, that can be used as well.
- `<FLAGS>`: Many (typically GUI-based) editors require the `--wait` flag so Git knows to wait until the file has been saved for it to finalize the commit message.

Some example commands are:

- `git config --global core.editor "code --wait"`
- `git config --global core.editor "atom --wait"`
- `git config --global core.editor "'C:\Program Files\Notepad++\notepad++.exe' -multiInst -nosession -notabbar"`
- `git config --global core.editor "vim"`
- `git config --global core.editor "nano"`

Commit Message Best Practices

The most commonly accepted rules / conventions on how to write a git commit message are as follows:

1. Limit the subject line to 50 characters
2. Capitalize only the first letter in the subject line
3. Don't put a period at the end of the subject line
4. Use the imperative mood
5. Describe the what and the why, but not the how

What is the Imperative Mood?

The imperative mood is an English language contention that is used to give commands and requests to people. To tell if you are using the imperative mood, try prepending "If applied, this commit will" to your commit message and seeing if it's grammatically correct. For example:

- "Fixing the HTTPS error"
  "If applied, this commit will fixing the HTTPS error"
- "Fixed the HTTPS error"
  "If applied, this commit will fixed the HTTPS error"
- "Fix the HTTPS error"
  "If applied, this commit will fix the HTTPS error"

Since the third message is grammatically correct, it is in the imperative mood.

What if My Team Doesn't Want to Use the Conventions?

Then don't. They are conventions, not steadfast regulations. The conventions are general best practices and guidelines, but what's most important is to find and use a system that works for you and your team.

0.3.3 Pushing and Pulling Changes

When changes are committed, they are not automatically stored remotely on GitHub, Gitlab, or wherever else your repository is stored if it's remote. They are instead stored locally on your system until they are pushed to the remote source. This can be done with the use of the `git push` command, though there are some things to note about this command.

A more descriptive syntax of the `git push` command is `git push <remote> <branch>`. This is important to understand for use with branches, which will be explained later.

Do not use `git pull --force` unless you know what you are doing. This can and will break things.

When changes are committed to the remote by other users (or by yourself on a different machine), they aren't downloaded to your local repository until you tell Git to grab them using `git pull`. When this is run, remote changes are synced with local changes and everything is updated.

If you are working on a branch, run `git pull origin main` often to keep merge conflicts small

0.3.4 Creating a Branch

A branch is a separate working track that can be used to make changes without affecting the main codebase. Once completed, the code can be merged back into the main codebase.

Every Git repository has a main or master branch. The standard for this branch is to be named "main" and it will be referred to as such for the duration of this manual.

To create a new branch, run `git branch <branch-name>` and then checkout the branch with `git checkout <branch-name>`. You can combine these commands with
`git checkout -b <branch-name>`.

You can then work on the branch as you normally would.

Changes on the branch can be pushed to the remote server using
`git push origin <branch-name>`.

A basic sample workflow would be:

```
git pull origin main
git branch test-feature
git checkout test-feature
git add .
```

```
git commit -m "Worked on feature"
git push origin test-feature
```

Private Branches

A private branch is a branch that you create and use locally on your own machine *without pushing to remote*. There are several reasons why you shouldn't use them and several instances in which it's acceptable to use them.

For this course, it is highly discouraged for the following reasons:

- It can isolate your work and drift away from the main branch, creating a mess of merge conflicts when it's eventually merged into main.
- It reduces visibility and might cause teammates to accidentally duplicate work because they don't know you've started it.
- There's no backup of your work in case something happens to your machine (loss, theft, damage, data corruption, etc)

Instead, push your branch to the remote even if you're the only one working on it, and merge your branches back into main often.

0.3.5 Merging Branches

Merging a branch in Git takes the history and changes from one branch and integrates them into another branch, typically main. The standard workflow for branches is:

1. Switch to the target branch:
   `git checkout main`
2. Update your local target branch:
   `git pull origin main`
3. Merge the branch:
   `git merge branch-name`

However, do note that you will rarely run `git merge` on your own machine to update main. Instead, you will push your branch to remote and open a pull request. This will allow your team to discuss and review the code, then merge the code once it's been accepted by the team.

0.3.6 Resolving Merge Conflicts

Sometimes, when pushing to or pulling from remote, or when merging branches, there is a marge conflict. Typically this means that a file was modified by multiple different people. When this happens, Git marks the file like this:

```
<<<<<<< HEAD
My new code
=======
The code my teammate pushed earlier
>>>>>>> main
```

To resolve the conflict, simply delete the markers and the version of the code you don't want, then stage and commit the change. If you are having trouble, sometimes it can be helpful to speak to the teammate(s) who wrote the conflicting change to find the best solution.

0.4 Basic Workflow

This section contains the basic workflow, from start to merge, using a real repository. You can find the final repository at `https://github.com/WhatALegend27/git-manual`

```
~/./repo$ git init
Initialized empty Git repository in .././repo/.git/

~/./repo$ touch README.md LICENSE .gitignore .gitattributes
~/./repo$ git status
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitattributes
        .gitignore
        LICENSE
        README.md
nothing added to commit but untracked files present (use "git add" to track)

~/./repo$ git add .
~/./repo$ git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
```

```
        new file:   .gitattributes
        new file:   .gitignore
        new file:   LICENSE
        new file:   README.md

~/./repo$ git commit -m "Initial commit"
[main (root-commit) ee0f2ce] Initial commit
 4 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 .gitattributes
 create mode 100644 .gitignore
 create mode 100644 LICENSE
 create mode 100644 README.md

 ~/./repo$ git remote add origin git@github.com:WhatALegend27/git-manual.git
 ~/./repo$ git push -u origin main
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 472 bytes | 472.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To github.com:WhatALegend27/git-manual.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.

~/./repo$ git branch write-readme
~/./repo$ git checkout write-readme
Switched to branch 'write-readme'

~/./repo$ echo "# Git Manual Repo for CS3383" >> README.md
~/./repo$ echo "*.dSYM" >> .gitignore
~/./repo$ echo "*DS_Store" >> .gitignore
~/./repo$ git add .
~/./repo$ git status
On branch write-readme
Changes to be committed:
  (use: "git restore --staged <file>..." to unstage)
        modified:   .gitignore
        modified:   README.md

~/./repo$ git commit -m "Edited readme and added macOS junk to .gitignore"
[write-readme 4a15d57] Edited readme and added macOS junk to .gitignore
2 files changed, 3 insertions(+)

~/./repo$ git checkout main
```

```
Switched to branch 'main'
Your branch is up to date with 'origin/main'.

~/./repo$ git merge write-readme
Updating ee0f2ce..4a15d57
Fast-forward
 .gitignore | 2 ++
 README.md  | 1 +
 2 files changed, 3 insertions(+)

~/./repo$ git branch -d write-readme
Deleted branch write-readme (was 4a15d57)

~/./repo$ git push
Enumerating objects: 7, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (4/4), 587 bytes | 587.00 KiB/s, done.
Total 4 (delta 1), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To github.com:WhatALegend27/git-manual.git
   ee0f2ce..4a15d57  main -> main
```

0.5 Disaster Recovery

It is difficult to truely break Git. However, things happen and sometimes data gets lost or overritten. Thankfully, there are ways to fix this:

0.5.1 Reflog and Reset

If you accidentally delete a branch or a significant amount of work gets wiped out, you can use `git reflog` to see a log of previous changes and restore them.

For example, running `git reflog` on my git-manual repository shows the following changes:

```
4a15d57 (HEAD -> main, origin/main) HEAD@{0}: merge write-readme...
ee0f2ce HEAD@{1}: checkout: moving from write-readme to main
4a15d57 (HEAD -> main, origin/main) HEAD@{2}: commit: Edited readme....
ee0f2ce HEAD@{3}: checkout: moving from main to write-readme
ee0f2ce HEAD@{4}: commit (initial): Initial commit
```

Each change is denoted with a specific hash. For example, the initial commit is `ee0f2ce`.

I can then use `git reset --hard <hash>` to change back to this previous state. Note that the `--hard` flag will overwrite any unsaved changes so make sure your working directory is

clean first. For example, `git reset --hard ee0f2ce` will restore me to the state it was after the initial commit.

0.5.2 Abort

Sometimes a command fails, or you begin a merge and there are a ton of merge conflicts. To return to exactly where you were before you started, you can run the command again with the `--abort` flag. For example, to abort a merge: `git merge --abort`.

0.5.3 Restoration of Deleted Files

If you accidentally deleted a file and have *not* committed the deletion yet, you can restore the deleted file with `git restore <filename>`.

If you have committed the deletion and want it back, you can use `git log -- <filename>` to find the commit hash where the file last existed and restore it with
`git checkout <commit-hash> -- <filename>`.

For example, I added a file `important.cs` to my repo, committed the change, and deleted the file. I now want to restore it:

First, I check the log:

```
~/./repo$ git log -- important.cs
commit 0352b69bed338c463e2cdace932c9bbdd06d04ed (HEAD -> main, origin/main)
Author: WhatALegend27 <jvanv06@gmail.com>
Date:   Mon Feb 9 22:21:51 2026 -0800

    Deleted unimportant file

    Probably won't need it later

commit bf7cd81f2e857058832bd4f2600448f9977d1285
Author: WhatALegend27 <jvanv06@gmail.com>
Date:   Mon Feb 9 22:21:13 2026 -0800

    This is really important

    Seriously. We need this
```

Then, I restore the file using the most recent commit where it existed. The hash for this is `bf7cd81f2e857058832bd4f2600448f9977d1285`. Note that the slash after `git checkout` is for readability and is not needed.

```
~/./repo$ git checkout\
            bf7cd81f2e857058832bd4f2600448f9977d1285 -- important.cs
~/./repo$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
 new file:   important.cs

~/./repo$ git commit -m "Phew, thought we lost it"
```

0.5.4 Undoing Commits

You can undo the last commit with `git reset`.

`git reset --soft HEAD~1` will undo the commit but keep changes staged. This is good for fixing typos or minor details.

`git reset --hard HEAD~1` will undo the commit and delete all changes.

0.5.5 Re-Cloning the Repository

If the local repository becomes so corrupted or confusing that nothing works, you can always re-clone the repository. It's typically a good idea to move the broken folder to a backup folder, then re-clone inside a new folder and manually copy over any files that weren't pushed yet.

0.6 Tips and Tricks

- Use SSH for authentication, as it is powerful, secure, and reliable
- Use descriptive commit messages
- If you are working on a branch, run `git pull origin main` often to keep merge conflicts small
- Avoid private branches
- Avoid working on `main` unless necessary
- NEVER use `git push --force`
- NEVER use the `--force` flag
- When dealing with a merge conflict, sometimes it's good to talk to your teammates

0.7 GitHub

Although GitHub was designed primarily as a place to remotely store and access Git repositories, it comes with a plethora of features to make development easier.

0.7.1 Issues

The issues section is a tab within your repository that users can use to report issues with your code, recommend features, or provide feedback. Although not incredibly useful on small class projects, they can be crucial for bug fixing and reports for larger projects. For example, the GitHub repository for Vim has over 1,600 open issues and 7,700 closed issues as of the time of writing.

0.7.2 Pull Requests

The pull requests section is a tab within your repository that code contributors can use to help streamline the merging of a branch into main. It allows a team to look at code and find bugs, ensure adherence to the team's style guide, and share knowledge.

Git repositories can also connect to tools that tests the code when a pull request is made. It ensures that the code in main is, most likely, operational at all times.

0.7.3 Releases

The releases section is a tab within your repository that you can use to publish finished builds of your code. It allows you to keep a complete version history of finished products without having to manage them within the repository itself.

With releases, you can keep the source code within the repository and finished builds within a completely different section. This keeps codebases smaller and easier to use and manage as a team.

0.8 Glossary

0.8.1 Terminology

Branch: A separate section of code to be worked on by a contributor. Typically meant for adding a feature or fixing a bug.

Codebase: A collection of source code.

Commit (Noun): A snapshot of the source code containing staged changes.

Commit (Verb): Taking a snapshot of the source code.

HEAD: A pointer to the most recent changes within a repository.

Origin: The address of a remote repository.

Pull Request: A feature on GitHub that allows a team to review a merge before it is completed.

Remote: A repository which you can push changes into and pull changes out of.

Repository/Repo: A collection of files tracked and secured by Git.

Stage: A list of files that will be committed.

Working Tree: All the files and folders within a Git repository excluding the .git folder.

0.8.2 Commands

`git add`: Used to add a file/directory to the staging area. All files can be added with `git add .` or `git add --all`.

`git branch`: Used to create a new branch.

`git checkout`: Used to switch to a different branch. Can also be used to restore deleted files. Can create a new branch using `git checkout -b <branch-name>`.

`git clone`: Used to clone a remote repository.

`git commit`: Used to create a commit. The `-m` flag can be used to add a message. Omitting the `-m` flag will open the Git Editor.

`git config`: Used to configure Git settings.

`git init`: Initializes a new repository.

`git log`: Shows a log of all previous commits.

`git merge`: Merges two branches together.

`git pull`: Pulls remote changes to the local repository.

`git push`: Pushes local changes to the remote repository.

`git reflog`: Shows a log of all times the Git HEAD has moved, which happens with every change even if it's not a commit.

`git reset`: Resets changes to a previous commit relative to the position of HEAD.

`git restore`: Restores a deleted file.

`git status`: Used to see the current status of the working tree.

0.9 References

Ali, A. (2025, July 13). *Imperative mood in English with Examples*. Englishan. https://englishan.com/imperative-mood/

Apache Subversion. (n.d.). https://subversion.apache.org/

Cameron McKenzie. (2024, September 25). *How to Write a Git Commit Message: Conventions & Best Practices* [Video]. YouTube. https://www.youtube.com/watch?v=9ilpKtF0KGQ

Getting Started - A Short History of Git. (n.d.). Git. https://git-scm.com/book/en/v2/Getting-Started-A-Short-History-of-Git

Git Ignore and .gitignore. (n.d.). W3Schools. https://www.w3schools.com/git/git_ignore.asp

Keeping your account and data secure. (n.d.). GitHub Docs. https://docs.github.com/en/authentication/keeping-your-account-and-data-secure

Olawanle, J. (2022, August 8). *How to set up Git for the first time on MacOS*. freeCodeCamp. https://www.freecodecamp.org/news/setup-git-on-mac/

Sharma, R. (2025, July 23). *How to use HTTPS or SSH for Git?* GeeksforGeeks. https://www.geeksforgeeks.org/git/how-to-use-https-or-ssh-for-git/

Stack Overflow Developer Survey 2022. (2022, May). Stack Overflow. https://survey.stackoverflow.co/2022/

What is version control? (n.d.). GitLab. https://about.gitlab.com/topics/version-control/