Terminology

### Branch

A branch represents an independent line of development. Branches serve as an abstraction for the edit/stage/commit process discussed in Git Basics, the first module of this series. You can think of them as a way to request a brand new working directory, staging area, and project history. New commits are recorded in the history for the current branch, which results in a fork in the history of the project.

### HEAD

Git’s way of referring to the current snapshot. Internally, the git checkout command simply updates the HEAD to point to either the specified branch or commit. When it points to a branch, Git doesn't complain, but when you check out a commit, it switches into a “detached HEAD” state.

### Hook

A script that runs automatically every time a particular event occurs in a Git repository. Hooks let you customize Git’s internal behavior and trigger customizable actions at key points in the development life cycle.

### Master

The default development branch. Whenever you create a git repository, a branch named "master" is created, and becomes the active branch.

### Tag

A reference typically used to mark a particular point in the commit chain. In contrast to a head, a tag is not updated by the commit command.

### Version Control

A system that records changes to a file or set of files over time so that you can recall specific versions later.

### Working Tree

The tree of actual checked out files, normally containing the contents of the HEAD commit's tree and any local changes you've made but haven't yet committed.

# Git commands

### git add

Moves changes from the working directory to the staging area. This gives you the opportunity to prepare a snapshot before committing it to the official history.

### git branch

This command is your general-purpose branch administration tool. It lets you create isolated development environments within a single repository.

### git checkout

In addition to checking out old commits and old file revisions, git checkout is also the means to navigate existing branches. Combined with the basic Git commands, it’s a way to work on a particular line of development.

### git clean

Removes untracked files from the working directory. This is the logical counterpart to git reset, which (typically) only operates on tracked files.

### git clone

Creates a copy of an existing Git repository. Cloning is the most common way for developers to obtain a working copy of a central repository.

### git commit

Takes the staged snapshot and commits it to the project history. Combined with git add, this defines the basic workflow for all Git users.

### git commit --amend

Passing the --amend flag to git commit lets you amend the most recent commit. This is very useful when you forget to stage a file or omit important information from the commit message.

### git config

A convenient way to set configuration options for your Git installation. You’ll typically only need to use this immediately after installing Git on a new development machine.

### git fetch

Fetching downloads a branch from another repository, along with all of its associated commits and files. But, it doesn't try to integrate anything into your local repository. This gives you a chance to inspect changes before merging them with your project.

### git init

Initializes a new Git repository. If you want to place a project under revision control, this is the first command you need to learn.

### git log

Lets you explore the previous revisions of a project. It provides several formatting options for displaying committed snapshots.

### git merge

A powerful way to integrate changes from divergent branches. After forking the project history with git branch, git merge lets you put it back together again.

### git pull

Pulling is the automated version of git fetch. It downloads a branch from a remote repository, then immediately merges it into the current branch. This is the Git equivalent of svn update.

### git push

Pushing is the opposite of fetching (with a few caveats). It lets you move a local branch to another repository, which serves as a convenient way to publish contributions. This is like svn commit, but it sends a series of commits instead of a single changeset.

### git rebase

Rebasing lets you move branches around, which helps you avoid unnecessary merge commits. The resulting linear history is often much easier to understand and explore.

### git rebase -i

The -i flag is used to begin an interactive rebasing session. This provides all the benefits of a normal rebase, but gives you the opportunity to add, edit, or delete commits along the way

### git reflog

Git keeps track of updates to the tip of branches using a mechanism called reflog. This allows you to go back to changesets even though they are not referenced by any branch or tag.

### git remote

A convenient tool for administering remote connections. Instead of passing the full URL to the fetch, pull, and push commands, it lets you use a more meaningful shortcut.

### git reset

Undoes changes to files in the working directory. Resetting lets you clean up or completely remove changes that have not been pushed to a public repository.

### git revert

Undoes a committed snapshot. When you discover a faulty commit, reverting is a safe and easy way to completely remove it from the code base.

### git status

Displays the state of the working directory and the staged snapshot. You’ll want to run this in conjunction with git add and git commit to see exactly what’s being included in the next snapshot.

# Appendix C: Git Commands

**git help**

**Getting and Creating Projects**

**git init**

To take a directory and turn it into a new Git repository so you can start version controlling it, you can simply run git init.

**git clone**

The git clone command is actually something of a wrapper around several other commands. It creates a new directory, goes into it and runs git init to make it an empty Git repository, adds a remote (git remote add) to the URL that you pass it (by default named origin), runs a git fetch from that remote repository and then checks out the latest commit into your working directory with git checkout.

the --bare option to create a copy of a Git repository with no working directory

the --recurse-submodules option to makecloning a repository with submodules a little simpler.

**Basic Snapshotting:**

For the basic workflow of staging content and committing it to your history, there are only a few basic commands.

**git add**

The git add command adds content from the working directory into the staging area (or “index”) for the next commit.

**git status**

The git status command will show you the different states of files in your working directory and staging area. Which files are modified and unstaged and which are staged but not yet committed

**git diff:**

The git diff command is used when you want to see differences between any two trees. This could be the difference between your working environment and your staging area (git diff by itself), between your staging area and your last commit (git diff --staged), or between two commits (git diff master branchB)

We use it to look for possible whitespace issues before committing with the --check option in Commit Guidelines.

We see how to check the differences between branches more effectively with the git diff A...B syntax in Determining What Is Introduced.

We use it to filter out whitespace differences with -b and how to compare different stages of conflicted files with --theirs, --ours and --base in Advanced Merging.

Finally, we use it to effectively compare submodule changes with --submodule in Starting with Submodules.

**git difftool:**

The git difftool command simply launches an external tool to show you the difference between two trees in case you want to use something other than the built in git diff command.

**git commit**

The git commit command takes all the file contents that have been staged with git add and records a new permanent snapshot in the database and then moves the branch pointer on the current branch up to it.

We first cover the basics of committing in Committing Your Changes. There we also demonstrate how to use the -a flag to skip the git add step in daily workflows and how to use the -m flag to pass a commit message in on the command line instead of firing up an editor

the --amend option to redo the most recent commit.

the -S flag sign commits cryptographically

**git reset:**

The git reset command is primarily used to undo things, as you can possibly tell by the verb. It moves around the HEAD pointer and optionally changes the index or staging area and can also

optionally change the working directory if you use --hard. This final option makes it possible for this command to lose your work if used incorrectly, so make sure you understand it before using it.

We first effectively cover the simplest use of git reset in Unstaging a Staged File, where we use it to unstage a file we had run git add on.

**git rm**

The git rm command is used to remove files from the staging area and working directory for Git

including recursively removing files and only removing files from the staging area but leaving them in the working directory with --cached.

**git mv**

The git mv command is a thin convenience command to move a file and then run git add on the new file and git rm on the old file.

**git clear**

The git mv command is a thin convenience command to move a file and then run git add on the new file and git rm on the old file.

**Branching and Merging**

There are just a handful of commands that implement most of the branching and merging

functionality in Git.

**git branch**

The git branch command is actually something of a branch management tool. It can list the branches you have, create a new branch, delete branches and rename branches

In Tracking Branches we use the git branch -u option to set up a tracking branch.

**git checkout**

The git checkout command is used to switch branches and check content out into your working directory.

We see how to use it to start tracking branches with the --track flag in Tracking Branches.

**git merge**

The git merge tool is used to merge one or more branches into the branch you have checked out. It will then advance the current branch to the result of the merge.

We use it to reintroduce file conflicts with --conflict=diff3 in Checking Out Conflicts.

We go into closer detail on its relationship with git reset in Reset Demystified.

**git mergetool**

The git mergetool command simply launches an external merge helper in case you have issues with a merge in Git.

**git log**

The git log command is used to show the reachable recorded history of a project from the most recent commit snapshot backwards.

There we look at the -p and --stat option to get an idea of what was introduced in each commit and the --pretty and --oneline options to view the history more concisely, along with some simple date and author filtering options.

we use it with the --decorate option to easily visualize where our branch

pointers are located and we also use the --graph option to see what divergent histories look like

In Private Small Team and Commit Ranges we cover the branchA..branchB syntax to use the git log command to see what commits are unique to a branch relative to another branch. In Commit Ranges we go through this fairly extensively

In Merge Log and Triple Dot we cover using the branchA...branchB format and the --left-right syntax to see what is in one branch or the other but not in both. In Merge Log we also look at how to use the --merge option to help with merge conflict debugging as well as using the --cc option to look at merge commit conflicts in your history.

In RefLog Shortnames we use the -g option to view the Git reflog through this tool instead of doing branch traversal.

In Searching we look at using the -S and -L options to do fairly sophisticated searches for something that happened historically in the code such as seeing the history of a function.

In Signing Commits we see how to use --show-signature to add a validation string to each commit in the git log output based on if it was validly signed or not.

**git stash**

The git stash command is used to temporarily store uncommitted work in order to clean out your working directory without having to commit unfinished work on a branch

**git tag**

The git tag command is used to give a permanent bookmark to a specific point in the code history. Generally this is used for things like releases. This command is introduced and covered in detail

**Sharing and Updating Projects**

**git fetch**

The git fetch command communicates with a remote repository and fetches down all the information that is in that repository that is not in your current one and stores it in your local database.

**git pull**

The git pull command is basically a combination of the git fetch and git merge commands, where Git will fetch from the remote you specify and then immediately try to merge it into the branch you’re on.

**git push**

The git push command is used to communicate with another repository, calculate what your local database has that the remote one does not, and then pushes the difference into the other repository. It requires write access to the other repository and so normally is authenticated somehow.

In Deleting Remote Branches we use the --delete flag to delete a branch on the server with git push.

We see how to use it to share tags that you have made with the --tags option in Sharing Tags.

In Publishing Submodule Changes we use the --recurse-submodules option to check that all of our submodules work has been published before pushing the superproject, which can be really helpful when using submodules.

**git remote**

The git remote command is a management tool for your record of remote repositories.

It is used in nearly every subsequent chapter in the book too, but always in the standard git remote add <name> <url> format

**git archive**

The git archive command is used to create an archive file of a specific snapshot of the project.

**git submodule**

The git submodule command is used to manage external repositories within a normal repositories. This could be for libraries or other types of shared resources. The submodule command has several sub-commands (add, update, sync, etc) for managing these resources.

**Inspection and Comparison**

**git show**

The git show command can show a Git object in a simple and human readable way. Normally you would use this to show the information about a tag or a commit.

**git shortlog**

The git shortlog command is used to summarize the output of git log. It will take many of the same options that the git log command will but instead of listing out all of the commits it will present a summary of the commits grouped by author.

**git describe**

The git describe command is used to take anything that resolves to a commit and produces a string that is somewhat human-readable and will not change. It’s a way to get a description of a commit that is as unambiguous as a commit SHA-1 but more understandable.We use git describe in Generating a Build Number and Preparing a Release to get a string to name our release file after.

**Debogage**

**git bisect**

The git bisect tool is an incredibly helpful debugging tool used to find which specific commit was the first one to introduce a bug or problem by doing an automatic binary search.

**git blame:**

The git blame command annotates the lines of any file with which commit was the last one to introduce a change to each line of the file and what person authored that commit. This is helpful in order to find the person to ask for more information about a specific section of your code.

**git grep**

The git grep command can help you find any string or regular expression in any of the files in your source code, even older versions of your project.

**Patching**

**git cherry-pick**

The git cherry-pick command is used to take the change introduced in a single Git commit and try to re-introduce it as a new commit on the branch you’re currently on. This can be useful to only take one or two commits from a branch individually rather than merging in the branch which takes all the changes.

**git rebase**

The git rebase command is basically an automated cherry-pick. It determines a series of commits and then cherry-picks them one by one in the same order somewhere else.

We also use it in an interactive scripting mode with the -i option

**git revert**

The git revert command is essentially a reverse git cherry-pick. It creates a new commit that applies the exact opposite of the change introduced in the commit you’re targeting, essentially undoing or reverting it.

**External Systems**

Git comes with a few commands to integrate with other version control systems.

**git svn**

The git svn command is used to communicate with the Subversion version control system as a client. This means you can use Git to checkout from and commit to a Subversion server.

**Administration**

**git gc**

The git gc command runs “garbage collection” on your repository, removing unnecessary files in your database and packing up the remaining files into a more efficient format.

**git fsck**

The git fsck command is used to check the internal database for problems or inconsistencies.

**git reflog**

The git reflog command goes through a log of where all the heads of your branches have been as you work to find commits you may have lost through rewriting histories.

We cover this command mainly in RefLog Shortnames, where we show normal usage to and how to use git log -g to view the same information with git log output.

We also go through a practical example of recovering such a lost branch in Data Recovery

**git filter-branch**

The git filter-branch command is used to rewrite loads of commits according to certain patterns, like removing a file everywhere or filtering the entire repository down to a single subdirectory for extracting a project

**Plumbing Commands**

ls-remote to look at the raw references on the server

ls-files to take a more raw look at what your staging area looks like.

rev-parse to take just about any string and turn it into an object SHA-1.