Git & GitHub Masterclass for Beginners

Complete Course Notes — Version Control from Local to Remote

SECTION 1: Introduction to Version Control Systems

1.1 Why Do We Need Version Control?

Version Control Systems (VCS) solve two of the most common and frustrating problems every developer encounters.

Scenario 1 — The Broken Code Problem:

- You add a new feature to a working project — and it breaks everything
- You try to manually undo your changes, but the app still crashes — maybe you deleted one extra line by mistake
- Without VCS, getting back to the working state requires guesswork and manual line-by-line comparison
- With VCS, you can restore any previous snapshot of the entire project instantly

Scenario 2 — The Team Conflict Problem:

- Two developers edit the same file at the same time
- One person's changes overwrite the others without warning
- With VCS, changes are tracked per author, conflicts are flagged, and merging is controlled

1.2 What is a Version Control System?

A VCS is a system that records changes to files over time so that you can recall specific versions later.

VCS Feature	What It Does
Keeps all versions	Stores both new and old versions of every file — nothing is permanently lost
Documents every change	Each change is saved with a descriptive message explaining why it was made
Shows differences	Displays exactly what changed between any two versions
Enables collaboration	Multiple people can work on the same project simultaneously without overwriting each other
Time machine	Lets you restore the project to any previous state at any point in history
Audit trail	Records who made each change, when, and why

1.3 Git vs GitHub — Key Distinction

Tool	What It Is
Git	A local version control system. Tracks file changes and manage your project history on your computer.
GitHub	A remote hosting service for Git repositories. Enables cloud backup and team collaboration.
GitLab	Similar to GitHub — another remote hosting service with built-in CI/CD pipelines.
SVN (Subversion)	An older centralized version control system, still used in some organizations.

1.4 VCS in the Industry

- Used in virtually every technology company — development, QA, integration, automation, security
- Git and GitHub are required knowledge in most software engineering job interviews
- Open-source projects on GitHub allow anyone to contribute — improving your public profile
- Over 100 million repositories on GitHub; over 40 million registered users
- Microsoft acquired GitHub for approximately $7.5 billion in 2018

SECTION 2: Installing and Configuring Git

2.1 Downloading and Installing Git

- Go to google.com → search 'git download' → click git-scm.com/downloads
- Select your operating system: Windows, macOS, or Linux
- Choose 32-bit or 64-bit version based on your computer
- Run the installer and accept all defaults — no custom configuration needed for beginners
- Default editor: Vim (or change to Notepad or VS Code if preferred)
- After installation, launch Git Bash (Windows) or your Terminal (Mac/Linux)

2.2 Verifying the Installation

```
git --version
```

- If installed correctly, Git prints the version number (e.g. git version 2.27.0)
- If you see 'git is not recognized', Git is not installed correctly — re-run the installer

2.3 Getting Help from Git

```
git help
git help add       # opens browser page explaining the 'add' command
```

- Running git help shows a list of all common commands
- Running git help <command> opens a detailed web page with examples for that specific command

2.4 Initial Git Configuration

Git needs to know who you are so it can attach your name and email to every commit. This is especially important when working in a team.

```
git config --global user.name "Your Name"
git config --global user.email "you@example.com"
```

- The --global flag applies these settings to all Git projects on your computer
- To verify your settings:

```
git config --list
```

- You should see user.name and user.email listed correctly
- Every commit you make will be permanently stamped with this name and email — teammates can see who made each change

SECTION 3: Creating a Git Repository and the Basic Workflow

3.1 Creating Your First Git Project

- Create a new folder for your project (e.g. 'first-git-project' on your Desktop)
- Open Git Bash and navigate to the folder using the cd (change directory) command:
  ```
  cd /c/Users/YourName/Desktop/first-git-project
  ```
- Initialise Git in that folder:
  ```
  git init
  ```
- Git creates a hidden subdirectory called .git inside your project folder — this is where all version history is stored
- WARNING: Never delete the .git folder — doing so permanently destroys all project history
- Verify the repository was created:
  ```
  git status
  ```
- If successful, Git reports 'On branch master. Nothing to commit.' — no errors means Git is managing this folder

3.2 The Three-Stage Git Workflow

Every file in a Git project passes through three areas. Understanding this is the most important concept in Git.

Stage	What It Is	How to Move Into It
1. Working Directory	Your local files on disk — where you create and edit code	Files start here automatically when created
2. Staging Area	A holding zone where you select which changes should be included in the next snapshot	git add <filename>
3. Repository (.git)	The permanent history database — every committed snapshot lives here	git commit -m "message"

3.3 Core Workflow Commands

Command	What It Does
git status	Shows which files are untracked, modified, or staged — your dashboard for the current state of the project
git add <file>	Moves a specific file from the working directory to the staging area (start tracking it)
git add .	Stages ALL modified/new files at once — use with caution on large projects (may stage unwanted files)
git commit -m "msg"	Takes a snapshot of all staged files and saves it to the repository with your descriptive message
git log	Lists all commits in the repository — shows date, author, email, and message for each

3.4 Writing Good Commit Messages

- Be specific and descriptive — say WHAT changed and WHY, not just 'fixed stuff'
- Good examples: 'Added user login validation', 'Fixed null pointer error in payment module', 'Refactored database connection pool'
- Bad examples: 'Update', 'Fix', 'Changes', 'WIP'
- Use present tense (convention): 'Add feature' not 'Added feature'
- A good commit message tells a colleague exactly what this snapshot contains without them needing to read the code

3.5 Practical Example — First Commits

Creating and tracking your first file:

```
touch helloworld.py           # creates an empty file
# edit the file: print('Hello World')
git status                    # shows helloworld.py as untracked
git add helloworld.py         # move to staging area
git status                    # shows 'new file: helloworld.py'
git commit -m "Added helloworld.py file"
git log                       # shows one commit with date, author, message
```

Adding a second file and modifying the first:

```
touch greetings.py
# edit greetings.py: print('Welcome')
git add greetings.py
git commit -m "Added greetings.py file"
# modify helloworld.py: add print('Goodbye')
# modify greetings.py: add print('Welcome, my friend')
git status                    # shows both files modified
git add helloworld.py
git add greetings.py
git commit -m "Updated both files with additional output"
git log                       # shows 3 commits
```

SECTION 4: Viewing Differences and Removing Files

4.1 git diff — Comparing Versions

git diff shows exactly what changed in your files since the last commit — which lines were added (shown with +) and which were removed (shown with -).

```
git diff                        # compare working directory vs last commit
```

- Use case: you made changes yesterday but forgot to commit and can't remember what you did — git diff shows exactly what changed
- git diff can also compare two specific commits:
  ```
  git diff commit1 commit2      # compare any two commits using their IDs from git log
  ```
- And compare staging area vs repository:
  ```
  git diff --staged             # compare staged files vs last commit
  ```
- Use git log to find commit IDs (the long alphanumeric strings), then use them with git diff to compare any two points in history

4.2 Removing Files — git rm

- To remove a file from your Git repository (and from disk):
  ```
  git rm greetings.py
  ```
- To remove multiple files at once:
  ```
  git rm file1.py file2.py
  ```
- To remove an entire directory from Git:
  ```
  git rm -r directory-name/     # -r flag means recursive (includes subdirectories)
  ```
- After git rm, the removal is staged — commit it to make it permanent:
  ```
  git commit -m "Removed greetings.py — no longer needed"
  ```
- To view all files currently tracked by Git:
  ```
  git ls-files
  ```

SECTION 5: Tagging — Marking Important Points in History

5.1 What is a Git Tag?

As a project grows, the commit history can contain hundreds or thousands of entries. Tags mark specific important moments — typically product releases — so they can be found and restored quickly without scrolling through every commit.

Tag Type	How to Create It
Lightweight tag	git tag v1.0
Annotated tag	git tag -a v2.0 -m "message"

5.2 Common Tag Commands

Command	What It Does
git tag v1.0	Creates a lightweight tag called v1.0 at the current commit
git tag -a v2.0 -m "msg"	Creates an annotated tag with a message
git tag	Lists all existing tags
git tag --list	Lists all tags alphabetically
git tag -l "v1*"	Searches for tags matching a pattern (e.g. all tags starting with v1)
git show v1.0	Displays full details of a tag — commit info, author, message, changes made
git tag --delete v1.1.1	Deletes a specific tag (does not delete the commit — just the label)

5.3 Release Candidate Tags

- Common convention: v1.0-rc1, v1.0-rc2 (rc = release candidate — pre-release versions for testing)
- Search for all release candidates:
  ```
  git tag -l "*-rc*"
  ```
- Tags are typically used alongside commits to mark stable, deployable versions of a product

SECTION 6: Unstaging and Reverting Changes

6.1 Unstaging a File — Moving Back from Staging to Working Directory

A common mistake: you accidentally stage a file that should NOT be in the next commit. Here's how to fix it without losing your work.

```
# Accidentally staged both files
git add file1.py
git add file2.py    # this one was a mistake

# Fix: unstage file2.py only
git restore --staged file2.py

git status          # file2.py is back in working directory, file1.py still
staged
```

- git restore --staged <file> was introduced in Git 2.23 — the modern recommended approach
- Older equivalent (still seen in tutorials):
  ```
  git reset HEAD file2.py       # same result — available in older Git versions
  ```
- Key point: unstaging does NOT delete your changes — it just moves the file back to the working directory, ready to be staged separately in its own commit

6.2 Reverting a Modified File — Discarding Local Changes

You edited a file but then decided you want to abandon ALL changes since the last commit and start fresh from that point.

```
# Before: file is modified with unwanted changes
git status        # shows helloworld.py as modified

# Discard all changes — restore file to its last committed state
git restore helloworld.py

# After: file is back to exactly how it was at the last commit
```

- WARNING: git restore (without --staged) permanently discards your local changes — they cannot be recovered
- Only use this command when you are 100% certain you do not need those changes
- Data that has been committed can almost always be recovered — data in the working directory that was NEVER committed may be gone forever
- If unsure: consider stashing (git stash) to temporarily save changes before reverting — stashing is covered in the branching section

6.3 Unstage vs Revert — Quick Comparison

Action	Command
Unstage a file	git restore --staged <file>
Discard local edits	git restore <file>

SECTION 7: Remote Repositories and GitHub

7.1 Why Use a Remote Repository?

Git alone is excellent for solo work on your local machine. But for team projects, you need a shared online location where everyone can contribute. That is what remote repositories provide.

Benefit	Explanation
Collaboration	Multiple developers push their work to one shared repository — everyone sees the same codebase
Cloud backup	If your computer is lost or breaks, your code is safe on the remote server
Code review	Team members can review each other's changes before they reach the main codebase
Open source contribution	Others can find your public projects and contribute improvements
Visibility	Your GitHub profile acts as a developer portfolio — employers can see your work

7.2 Team Workflow with Remote Repositories

Example: three developers building a social media platform:

- John develops the 'upload posts' feature locally on his computer
- Emily develops the 'play videos' feature locally on her computer
- Mike develops the 'send messages' feature locally on his computer
- All three push their work to a shared remote repository on GitHub
- Each can pull the latest version to see each other's contributions
- The remote repository acts as the single source of truth for the entire project

7.3 Creating a GitHub Account and Repository

- Go to github.com → Sign up with username, email, and password → verify email
- After sign-in: click 'Create repository' (or the '+' icon in the top right)
- Fill in: repository name, optional description, and visibility

Setting	Options / Notes
Repository name	Short, descriptive, hyphen-separated — e.g. 'first-git-project'
Visibility	Public: anyone on the internet can view it. Private: only you and invited collaborators.
README file	Optional — a markdown file describing your project. Recommended for public projects.
.gitignore	Optional — pre-configured template to ignore common generated files for your chosen language
License	Optional — specifies how others may use your code (MIT, Apache, GPL, etc.)

- Previously private repos on GitHub required a paid plan (~$7/month) — they are now free for all users
- Approximately 30% of GitHub's 100M+ repositories are public

7.4 Connecting Your Local Repository to GitHub

After creating the remote repository on GitHub, link it to your local Git project:

```
# Step 1: Add GitHub as the remote (origin = the standard nickname for the
remote URL)
git remote add origin https://github.com/yourusername/your-repo.git

# Step 2: Verify the connection
git remote -v

# Step 3: Push your local commits to GitHub
git push origin master
```

- 'origin' is just a conventional alias for the remote URL — you could name it anything
- 'master' is the name of the branch you are pushing
- After pushing, refresh the GitHub page — all your files and commit history will appear online

SECTION 8: SSH — Secure Connection to GitHub

8.1 Why SSH?

SSH (Secure Shell) provides a secure, encrypted connection between your computer and GitHub. It replaces the need to enter a username and password every time you push or pull — and is significantly more secure than plain text connections.

Problem	SSH Solution
Plain connections can be intercepted — a 'man in the middle' can read your password	SSH encrypts all data so it is unreadable to anyone intercepting the connection
Typing password every time is slow and error-prone	Once SSH keys are configured, authentication happens automatically
Username/password can be phished or brute-forced	Private key never leaves your machine — cannot be guessed or phished

8.2 How SSH Key Pairs Work

- SSH generates two mathematically linked keys — a key pair:
  - Private key (id_rsa): stays on YOUR computer ONLY — never share this
  - Public key (id_rsa.pub): uploaded to GitHub — safe to share
- When you push to GitHub, your computer signs the request with the private key
- GitHub verifies the signature using the public key you uploaded — if they match, access is granted
- No password is transmitted — even if someone intercepts the connection, they cannot decrypt the data without your private key

8.3 Generating and Configuring SSH Keys

```
# Step 1: Generate the SSH key pair (RSA algorithm, 4096 bits)
ssh-keygen -t rsa -b 4096 -C "your@email.com"
# Press Enter to accept default file location (~/.ssh/id_rsa)
# Press Enter twice to skip passphrase (or set one for extra security)

# Step 2: Start the SSH agent
eval $(ssh-agent -s)

# Step 3: Add your private key to the agent
ssh-add ~/.ssh/id_rsa

# Step 4: Copy your public key to clipboard (Windows Git Bash)
clip < ~/.ssh/id_rsa.pub
```

Adding the public key to GitHub:

- GitHub → Settings → SSH and GPG keys → New SSH key
- Title: give it a descriptive label (e.g. 'My Laptop')
- Key: paste the copied public key content → click Add SSH Key
- After adding the key, git push will work without entering a password

SECTION 9: GitHub Features — Explore, Watch, Star, Issues

9.1 Exploring GitHub

- Explore tab: browse repositories based on your interests and past activity
- Topics: browse projects by subject area (Android, Arduino, C++, COVID-19, etc.)
- Trending: see what repositories are gaining the most stars this week/month
- Search: find any public repository — try 'coronavirus', 'machine learning', 'game engine'

9.2 Watch vs Star — What's the Difference?

Feature	Watch
What it does	Subscribe to a repository — receive notifications of ALL activity
Notifications	Yes — notified of commits, issues, pull requests, releases
Where it appears	Notifications inbox
Use when	You want to actively follow a project's development

9.3 Viewing a File's History — Raw, Blame, History

Button	What It Shows
Raw	The plain source code without GitHub's syntax highlighting or HTML formatting. Copy code from here to avoid pasting unwanted hidden characters into your IDE.
Blame	Shows who wrote each line of the file — which contributor committed every specific line. Useful for finding who introduced a bug or specific feature.
History	Shows all commits that modified this specific file — as opposed to the full project commit log which includes all files.

9.4 Issues — GitHub's Built-in To-Do Tracker

The Issues section is a project management tool built into every GitHub repository. Despite the negative-sounding name, issues are used for much more than bugs.

Issue Element	Purpose
Title	Short summary of the issue
Description / Body	Detailed explanation — what the problem is, steps to reproduce, proposed solution, relevant links
Labels	Categorise the issue — Bug (red), Enhancement (blue), Question, Documentation, Duplicate, etc. — enables filtering
Milestone	Group related issues into a project phase or feature — e.g. 'Version 2.0 Release'
Assignee	Assign a team member responsible for resolving this issue
Comments	Discussion thread — any collaborator can contribute insights, suggestions, or updates

- Issues function like a professional to-do list for your project — teams assign, prioritise, and track all work through them
- You can reference other issues or pull requests by number (#42) — GitHub auto-links them
- Issues can be opened by anyone (on public repos) — great for collecting bug reports from users

SECTION 10: .gitignore — Keeping Unwanted Files Out

10.1 What is .gitignore?

Some files in your project should never be committed to the repository. The .gitignore file tells Git which files and directories to ignore completely — they will not appear in git status, git add, or git commit.

File/Directory Type	Why to Ignore It
Build output (build/, dist/)	Auto-generated from source code — everyone should build on their own machine
Dependencies (node_modules/, venv/)	Very large — each developer installs these locally using package managers
IDE configuration (.vscode/, .idea/)	Personal editor settings that differ between developers
Temporary files (*.tmp, *.log)	Auto-created by apps, operating systems, and editors — not source code
Secrets (.env, config.ini)	CRITICAL — API keys, passwords, and tokens must NEVER be committed to a repo

10.2 Creating and Configuring .gitignore

```
touch .gitignore                # create the file (note the dot at the start)
```

Inside the .gitignore file (each line = one rule):

```
build/                          # ignore the entire build/ directory
*.txt                           # ignore ALL files ending in .txt
*.log                           # ignore all log files
node_modules/                   # ignore the node_modules directory
.env                            # ignore the environment variables file
```

- The asterisk (*) is a wildcard matching any characters — *.txt matches all text files
- A trailing slash (build/) means ignore a directory; no slash means ignore a file with that name
- Lines starting with # are comments

10.3 Committing and Sharing .gitignore

- The .gitignore file itself SHOULD be committed to the repository — this ensures all teammates use the same ignore rules

```
git add .gitignore
git commit -m "Added .gitignore rules"
git push origin master
```

- GitHub maintains official .gitignore templates for every major language and framework — search 'gitignore templates github' to find the right one for your project (Python, Java, Node.js, Unity, etc.)

SECTION 11: Fork and Clone — Working with Other People's Projects

11.1 Fork — Making Your Own Copy on GitHub

Fork creates a copy of someone else's GitHub repository in your own GitHub account. You can then freely modify your copy without affecting the original.

- Click the Fork button on any public GitHub repository → it appears in your account instantly
- Your forked copy is completely independent — changes you make do not affect the original
- When you want to suggest your changes to the original author, you create a pull request
- The original author receives your pull request, reviews it, and decides whether to merge it
- Fork is how the entire open-source ecosystem works — thousands of people contributing improvements to shared projects

11.2 Clone — Downloading to Your Local Machine

Clone downloads a repository from GitHub to your computer so you can work on it locally with your IDE, run tests, compile it, and make real changes.

```
git clone https://github.com/username/repository.git
```

- After cloning, the repository exists on your local machine — you can open it in any IDE
- Make your changes, commit them locally, then push back to your GitHub fork
- Then create a pull request from your fork to the original repository

11.3 Fork vs Clone vs Pull Request

Action	What It Does	When to Use It
Fork	Copies a repo from someone else's GitHub account to your GitHub account	When you want to contribute to a project you don't own
Clone	Downloads a GitHub repo (yours or forked) to your local computer	When you want to actively develop, test, and build the project locally
Pull Request	Proposes your changes from your fork to the original repository	When you're ready to suggest your contributions to the original author

SECTION 12: Branches — Independent Lines of Development

12.1 Why Use Branches?

The master (or main) branch holds the official, working production code. Developing new features or fixing bugs directly on master is risky — a mistake could break the entire deployed product for all users.

- Branches create a separate, isolated copy of the code where you can experiment freely
- Changes on a branch do NOT affect master until you deliberately merge them
- If something goes wrong on a branch, you can delete it and return to master instantly
- Multiple developers can work on different features simultaneously — each on their own branch
- Code is reviewed and tested on a branch BEFORE being merged into master

12.2 Branch Commands

Command	What It Does
git branch	Lists all local branches. The current branch is highlighted in green with *.
git branch <name>	Creates a new branch — copies the current branch state into the new one
git checkout <name>	Switches to the named branch — all future commits go to this branch
git checkout -b <name>	Creates AND switches to a new branch in one command (shortcut)
git push origin <branch>	Pushes the branch to GitHub — it appears as a separate branch in the remote repo
git merge <branch>	Merges the named branch into the current branch (usually master)

12.3 Practical Branching Workflow

```
# Step 1: Create a new branch for your feature
git branch descriptive-greetings

# Step 2: Switch to the new branch
git checkout descriptive-greetings

# Step 3: Verify you are on the correct branch
git status          # shows 'On branch descriptive-greetings'
git branch          # descriptive-greetings highlighted in green

# Step 4: Make changes, add, and commit as normal
# (edit greetings.py, create moonflight.py, etc.)
git add greetings.py
git commit -m "Added motivation phrase to greetings"
git add moonflight.py
git commit -m "Added moonflight feature"

# Step 5: Push the branch to GitHub
git push origin descriptive-greetings

# Step 6: Return to master (master is unaffected by the branch work)
git checkout master
```

12.4 Verifying Branch Isolation

- After pushing a branch, visit your repository on GitHub → click the branch dropdown
- Switch between master and descriptive-greetings — you will see different files and file contents
- Files added or modified on descriptive-greetings are INVISIBLE from master — isolation confirmed
- This isolation is what makes branches safe — team members can work independently without interfering with the deployed product

12.5 Branch Best Practices

- Never develop features directly on master — always create a dedicated branch
- Name branches descriptively: feature/user-login, bugfix/payment-crash, hotfix/null-pointer
- Keep branches focused on one feature or bug fix — smaller, focused branches are easier to review
- Delete branches after they are merged to keep the repository clean
- Require code review (pull request approval) before merging any branch into master
- The master branch should always contain working, deployable code

— End of Course Notes —